Graphes

C. Charignon

Computer science is no more about computers than astronomy is about telescopes.

Edsger Dijkstra

Table des matières

Première partie

Cours

1 Vocabulaire

Nous fixons pour tout le chapitre un ensemble S que nous appelons ensemble des *sommets*, et un ensemble $\mathcal{A}$ de couples d'arêtes que nous appelons ensemble des *arêtes*. Le couple $(S, \mathcal{A})$ s'appelle alors un *graphe* (orienté). On notera $G = (S, \mathcal{A})$ dans la suite.

A priori, étant donné $(i, j) \in S^2$, si une arête (i, j) est présente dans $\mathcal{A}$, il n'y a aucune raison pour que (j, i) y soit aussi. C'est pourquoi on dit que le graphe est orienté. Lorsqu'on représente un graphe orienté, on représente l'arête (i, j) par une flèche de i vers j. Le sommet i est alors l'*origine* de l'arête, et j son *extrémité*.
Remarque : On réserve parfois le terme « arête » à un couple $(i, j) \in \mathcal{A}$ tel que $(j, i) \in \mathcal{A}$, utilisant le terme « arc » dans le cas contraire.

Le nombre d'arêtes qui arrivent à un sommet s s'appelle le *degré entrant* de s. Le nombre d'arêtes qui partent de s s'appelle son *degré sortant*.

Lorsque pour tout $(i, j) \in S^2$, $(i, j) \in \mathcal{A} \Leftrightarrow (j, i) \in \mathcal{A}$, on dit que le graphe est non orienté. Dans ce cas, on représente graphiquement les arêtes par de simples traits et non plus des flèches.

Une suite $(s_0, ..., s_n)$ de sommets telle que $\forall i \in [\![0, n[\![$, $(s_i, s_{i+1}) \in A$ s'appelle un *chemin*.
Un chemin d'au moins trois sommets dont le point de départ est égal au point d'arrivée est un *cycle*.
Si γ est un chemin de s vers t, nous noterons $s \overset{\gamma}{\rightsquigarrow} t$.
La *longueur* d'un chemin est le nombre d'arêtes qui le compose.
La *distance* entre deux sommets s et t est la longueur d'un chemin reliant s à t de longueur minimale. On la note $d(s, t)$. S'il n'existe aucun chemin de s à t, on convient que $d(s, t) = \infty$.
Dans un graphe non orienté, un ensemble A de sommets est dit *connexe* si pour tout $(s, t) \in A^2$, il existe un chemin de s à t (pour un graphe orienté, la notion de connexité présente quelques subtilités dont nous ne parlerons pas ici).

Définition 1.1. *Soit γ_1 et γ_2 deux chemins dans G. On suppose que le dernier sommet de γ_1 et aussi le premier de γ_2. On notera alors $\gamma_1 + \gamma_2$ la concaténation de γ_1 et γ_2, c'est-à-dire la suite des sommets de g_1, sauf le dernier, suivie des sommet de γ_2.*

Remarque : La notation + est choisie par analogie avec la notation Python. En effet si γ_1 et γ_2 sont représentés par des objets de type `List`, alors leur concaténation s'obtient par `gamma1[:-1] + gamma2`.
En Caml, on utiliserait @ plutôt que +.

Proposition 1.2. *Si G est non orienté et connexe, la fonction d définie ci-dessus est bien une distance.*

Démonstration :

- *Inégalité triangulaire :* Soit $(s, t, u) \in S^3$. Soient $l = d(s, t)$ et $m = d(t, u)$. Soient γ un chemin de longueur minimale de s à t, et δ un chemin de longueur minimale de t à u. Alors $\gamma + \delta$ est un chemin de s à u, qui est de longueur $l + m$.Ceci prouve que $d(s, u) \leqslant l + m = d(s, t) + d(t, u)$.
- *Symétrie :* Soit $(s, t) \in S^2$. Soit γ un chemin de longueur minimale de s vers t. Notons γ^T le chemin renversé. Il existe bien car G est non orienté, Et il relie t à s, prouvant que $d(t, s) \leqslant d(s, t)$.
  Le même raisonnement en échangeant les rôles de s et t prouve que $d(s, t) \leqslant d(t, s)$, d'où égalité.
- Soit $s \in S$. Le chemin (s) relie s à s et est de longueur 0. Il est donc de longueur minimale, et $d(s, s) = 0$.
- Soient $(s, t) \in S^2$. Supposons $d(s, t) = 0$. Soit γ un chemin de longueur minimal de s à t. Il est de longueur 0 donc ne contient aucune arête. Donc son départ est aussi son arrivée, donc $s = t$.

□

2 Exemples

- Graphe d'itinéraires ;
- Réseau social ;

- Jeux au tour par tour : chaque état du jeu est un sommet, chaque coup possible une arête ;
- Graphe des dépendances entre des modules Python ;
- Graphe de contraintes entre différentes tâches (par exemple recette de cuisine)

3 Implémentation

Notons n le nombre de sommets de G.
Si on veut simplifier l'impléentation, on peut faire en sorte que $S = [\![0, n[\![$.

- par *matrice d'adjacence* : on utilise une matrice `m` de format $n \times n$, telle que pour tout $(i, j) \in S^2$, `m[i][j]` est un booléen qui indique s'il y a une arête pour passer du sommet i au sommet j.
  Si $S = [\![0, n[\![$, ce sera une matrice au sens habituel : un tableau de tableaux. Pour un ensemble S plus général on peut utiliser un dictionnaire de dictionnaires.
- par *listes d'adjacence* : Pour tout $s \in S$, on enregistre le tableau (type **`list`**) des voisins de s. Si $S = [\![0, n[\![$, on prendra un tableau de tableaux. Sinon, un dictionnaire de tableaux.

Voici quelques avantages et inconvénients de ces deux approches :

- Avec une matrice d'adjacence, il est immédiat de savoir si on peut passer d'un sommet à un autre.
- En revanche, il est plus long de récupérer la liste des voisins d'un sommet puisqu'il faut parcourir toute une ligne de la matrice.
- L'implémentation par matrice d'adjacence peut être modifiée de façon immédiate pour enregistrer en plus une étiquette à chaque arête. Par exemple, dans le graphe d'une carte routière, on pourra enregistrer sur chaque arête le temps de parcours de cette arête, et ainsi programmer facilement une fonction qui calcule le temps de parcours d'un chemin. Ceci dit, ça n'est pas très compliqué non plus de rajouter des étiquettes dans l'implémentation par listes d'adjacence.
- L'implémentation par matrice ne permet pas de mettre plusieurs arrêtes entre deux mêmes sommets. Ceci dit, la définition mathématique donnée ci-dessus non plus...

Au niveau occupation mémoire :

- La matrice d'adjacence occupe un espace en $O\left(|S|^2\right)$
- Le tableau des listes d'adjacence occupe un espace en $O\left(|S| + |A|\right)$.

Ainsi, lorsque le nombre d'arêtes est petit devant $|S|^2$, ce qui est souvent le cas, le tableau des listes d'adjacence occupe moins d'espace.

Remarque : Dans chacune de ces deux implémentations, un graphe est modifiable.

4 Parcours d'un graphe : généralités

4.1 Principe et invariants de boucle

Voici le vocabulaire que j'utiliserai dans la suite de ce cours.
À chaque instant, l'ensemble des sommets sera partitionné en trois :

- Les sommets non découverts, que nous appellerons les sommets « blancs » ;
- Les sommets découverts mais pas encore traités, que nous appellerons sommets « gris ». D'une manière ou d'une autre (en fonction du type de parcours, mais aussi de l'implémentation choisie), ces sommets sont enregistrés comme devant être visités prochainement ;
- Les sommets traités, que nous appellerons « noirs ». Il s'agit d'éviter d'y revenir.

Quel que soit l'algorithme employé et son but, les invariants de boucles suivant seront toujours maintenus :

- (VN) Les voisins d'un sommet noir sont noirs ou gris ;
- (VG) Un sommet gris a au moins un voisin noir ; (en pratique c'est le sommet à partir duquel il a été découvert)

En outre, à chaque étape de notre algorithme, nous ferons en sorte que :

- (CC) Les seuls changements de couleur possibles pour un sommet sont de blanc vers gris et de gris vers noir.

Le terme « étape » ci-dessus est un peu imprécis... Disons qu'il s'agira d'un tour de la boucle principale, ou d'un appel récursif de la fonction principale. Autrement dit dans les démonstrations ci-dessous, vous pourrez remplacer « étape » par «tour de boucle » ou « appel récursif » selon le contexte.

Il conviendra donc de faire en sorte lors de la programmation que ces invariants soient bien conservés. Si tel est le cas, on aura les premières conséquences suivantes :

Lemme 4.1. *On suppose les assertions (VN), (VG), et (CC) vérifiées. Si de plus, il n'y a plus de sommet gris alors les sommets blancs sont « déconnectés » des sommets noirs. Précisément, il n'existe aucun chemin d'un sommet noir vers un sommet blanc.*

N.B. En pratique il n'y a plus de sommets gris lorsque l'algorithme se termine.

Démonstration : Supposons l'existence d'un chemin c de longueur n tel que c_0 est noir et c_n est blanc. Soit $i \in [\![0, n]\!]$, minimal tel que c_i n'est pas noir. Alors $i > 0$ car c_0 est noir, donc c_{i-1} existe et est noir. C'est un voisin noir de c_i donc par (VN) c_i est noir ou gris. Il n'est pas noir par définition, et ne peut être gris par hypothèse. Contradiction. □

Lemme 4.2. *Notons $\mathcal{N}_0$ l'ensemble des sommets noirs au début de l'exécution. On suppose qu'un algorithme maintenant les assertions (VN), (VG), et (CC) a été employé.*
Alors pour tout sommet s noir ou gris, il existe un chemin d'un élément de $\mathcal{N}_0$ vers s.

Démonstration : On prouve que l'assertion « pour tout sommet s noir ou gris, il existe un chemin d'un élément de $\mathcal{N}_0$ vers s » est un invariant de boucle.

- *Initialement :* Au début de l'exécution, tout sommet noir est dans $\mathcal{N}_0$, et donc trivialement relié à un élément de $\mathcal{N}_0$. Et tout sommet gris est voisin d'un sommet de $\mathcal{N}_0$ par (VG), il est donc relié à $\mathcal{N}_0$ par un chemin de longueur 1.
- *Hérédité :* Supposons la propriété vérifiée à une certaine étape. À l'étape suivante :
  - ◇ Les sommet noirs étaient auparavant noirs ou gris (par (CC)) et donc reliés à $\mathcal{N}_0$ par hypothèse de récurrence.
  - ◇ Les sommet gris sont reliés aux noirs (par (VG)) et dont à $\mathcal{N}_0$ comme on vient de le voir.

Par récurrence, notre invariant de boucle est bien maintenu. □

Définition 4.3. *Soit $s \in S$. On appelle composante connexe de s dans G la plus grande partie de S qui est connexe.*

Ainsi la composante connexe de s est l'ensemble des sommets accessibles depuis s.
Pour justifier la validité de la définition, il faut s'assurer de l'existence et l'unicité d'une partie de S maximale parmi les parties connexes contenant s. pour ce, on vérifie qu'il s'agit de $\bigcup_{Y \text{ connexe tq } s \in Y} Y$. Le cours de maths parlera plus en détails de la notion de connexité.

On démontre alors qu'un parcours de graphe partant d'un sommet s_0 et vérifiant (VG), (VN), et (CC) parcourt exactement la composante connexe de s.

Proposition 4.4. *On suppose qu'il n'y a plus de sommet gris, que les invariants (VG), (VN), et (CC) ont été maintenus pendant toute l'exécution du programme, et qu'initialement un seul sommet, noté s_0 était noir. Alors l'ensemble des sommets noirs est la composante connexe de s_0.*

Démonstration :

- D'après le lemme 4.2, les éléments de $\mathcal{N}$ sont tous reliés à s_0 et donc forment une partie connexe contenant s_0.
- Ensuite soit X une partie de S plus grande que $\mathcal{N}$. Soit $t \in X \setminus \mathcal{N}$. Donc t est blanc par hypothèse. Et par le lemme 4.1, t n'est pas reliés à s_0. Donc X n'est pas connexe.

En conclusion, $\mathcal{N}$ est une partie connexe contenant s_0 maximale. C'est donc la composante connexe de s_0. □

4.2 L'algorithme de base

4.3 Les invariants de boucle

Vérifions que nos trois propriétés (VG), (VN), et (CC) sont vérifiées. Ici, une «étape» est un tour de la boucle « tant que ».

Entrées : Un graphe (A, S) et un sommet de départ s_0

1 **début**
2 $n \leftarrow |S|$
3 Créer trois ensembles de sommets $\mathcal{N}$, $\mathcal{G}$, et $\mathcal{B}$. Initialement, $\mathcal{B} = S$ et les autres sont vides
4 Peindre s_0 en gris (c'est-à-dire le retirer de $\mathcal{B}$ pour le mettre dans $\mathcal{G}$)
5 *Initialiser des variables*
6 **tant que** $\mathcal{G} \neq \emptyset$ **:**
7 Retirer un sommet s de $\mathcal{G}$
8 *Faire des trucs avec s*
9 Mettre tous les voisins de s qui sont blancs dans $\mathcal{G}$
10 Peindre s en noir
11 **fin**
12 *Renvoyer un résultat*
13 **fin**

Algorithme 1 : Parcours de graphe, rédaction impérative

- (CC) : Les seuls changements de couleurs sont lignes 9 et 10. À la ligne 9, des sommets gris ou blancs deviennent gris, et à la ligne 10, un sommet gris devient noir.
- (VG) : Les sommets qui deviennent gris sont les sommets voisins d'un sommet noté `s` dans l'algorithme qui devient noir. Et qui restera toujours noir d'après (CC). Donc ils ont et auront toujours un voisin noir.
- (VN) : lorsqu'on peint un sommet en noir (ligne 10), on a fait en sorte que tous ses voisins soient noirs ou gris (ligne 9).

4.4 En pratique ?

Les points restant à préciser :

- Comment enregistrons-nous les trois ensembles de sommets noirs, blancs, et gris ?
  - ◇ Pour les noirs : on peut utiliser un tableau de booléens. Je l'appellerai souvent `déjàVu`, ainsi `déjàVu[s]` est vrai ssi s est noir.
    Cependant un graphe peut être énorme, et il y a probablement de nombreux sommets qui resteront blancs pendant toute l'exécution du programme (par exemple si vous demandez à votre GPS l'itinéraire Pau-Arudy, il est probable que les données du Mexique n'interviendront pas). Il ne serait pas judicieux de charger en mémoire tous ces sommets inutiles. On peut dont préférer utiliser un dictionnaire plutôt qu'un tableau. En outre cela permet que les sommets soient désignés par n'importe quel objet (persistant : on ne peut pas utiliser d'objet mutable comme clef d'un dictionnaire) et non pas uniquement par les entiers de 0 à $|S| - 1$.
  - ◇ Pour les gris : c'est le point crucial, voir plus loin.
  - ◇ Une fois que nous aurons décidé comment enregistrer les gris, il sera inutile de gérer une structure pour les blancs : ce seront simplement « tous les autres ».
- Comment choisir le prochain sommet gris ? Souvent cela n'a pas d'importance pour le résultat final : nous avons vu que quoiqu'il arrive, nous parcourrons la composante connexe de s_0. Mais parfois au contraire il est important de traiter les sommets dans un ordre précis. La réponse à cette question influera directement sur le choix de la structure de données pour enregistrer les gris.

4.5 En autorisant les répétitions dans les gris

Enfin, toujours au niveau de la gestion des gris, il y aura un autre détail technique à régler : autorisons-nous les répétitions dans la structure contenant les gris ? Dans l'algorithme tel qu'écrit ci-dessus, ce n'est pas le cas, puisque nous parlons de *l'ensemble* des gris. En outre, nous prenons soin d'ajouter à g uniquement des sommets blancs. Mais cela nécessite un peu d'efforts car il faudra faire en sorte d'être capable de tester rapidement si un sommet est gris ou blanc pour savoir si nous l'insérons. En outre, nous verrons que pour le parcours « en profondeur », nous ne pouvons tout simplement pas éliminer les doublons.

Si finalement nous autorisons les doublons, il se pourra qu'un sommet soit traité, et donc devienne noir, alors qu'une autre version est encore présent dans la structure des gris. Il faudra donc faire attention au moment d'extraire un élément à visiter que celui-ci est vraiment gris (n'a pas été peint en noir depuis son insertion dans la structure

Entrées : Un graphe (A, S) et un sommet de départ s_0

```
1  début
2      n ← |S|
3      Créer une structure 𝒩 pour enregistrer les sommets noirs, et une structure àVisiter pour les sommets à
         visiter
4      Mettre s_0 dans àVisiter.
5      Initialiser des variables
6      tant que 𝒢 ≠ ∅ :
7          Retirer un sommet s de àVisiter
8          si s n'est pas noir :
9              Faire des trucs avec s
10             Mettre tous les voisins de s qui ne sont pas noirs dans àVisiter
11             Peindre s en noir
12         fin
13     fin
14     Renvoyer un résultat
15 fin
```

Algorithme 2 : Parcours de graphe, rédaction impérative, doublons autorisés dans gris

contenant les gris). Le code devient le suivant. En pratique, ce sera souvent cette version qu'on utilisera. J'ai remplacé l'identifiant « $\mathcal{G}$ » par `àVisiter` puisqu'il ne contient ppas que des gris dans cette nouvelle version.

On remarque que :

- les sommets gris sont alors les sommets dans `àVisiter` qui ne sont pas noirs ;
- je n'ai plus utilisé le mot « ensemble » maintenant que les doublons sont autorisés.

4.6 Terminaison

Dans la version simple (algorithme 1) le nombre de sommets non noirs est un variant de boucle, d'où la terminaison.

Dans le cas où les doublons sont autorisés (algorithme 2), prendre comme variant de boucle le couple (nombre de sommets non noirs, taille de la structure contenant les gris), muni de l'ordre lexicographique. En effet à chaque étape :

- Soit le sommet s extrait était vraiment gris, il passe noir et la première composante de notre couple diminue de 1 ;
- Soit s était noir. Il a été retiré de $\mathcal{G}$ et aucun autre changement de couleur n'a été effectué, donc la seconde composante du couple diminue de 1.

Remarque : En première année on a utilisé comme variant de boucle uniquement des entiers naturels. Pour montrer qu'un couple d'entiers naturels muni de l'ordre lexicographique peut aussi servir de variant de boucle, il faut prouver le théorème suivant :

Théorème 4.5. *Il n'existe pas de suite u à valeur dans $\mathbb{N}^2$, strictement décroissante pour l'ordre lexicographique, dont le domaine de définition soit infini.*

Niveau vocabulaire, on dit qu'une relation d'ordre $\leqslant$ sur un ensemble E est bien fondée lorsqu'il n'existe pas dans E de suite infinie strictement décroissante. Ainsi une quantité peut servir de variant de boucle pour prouver la terminaison d'un algorithme si et seulement si c'est une suite strictement décroissante dans un ensemble muni d'une relation d'ordre bien fondée. Et le théorème ci-dessus se reformule en « l'ordre lexicographique sur $\mathbb{N}^2$ est bien fondé ».

4.7 Complexité

Nous supposons que l'extraction et l'insertion d'un sommet gris (algo 1) ou d'un élément de `àVisiter` (algo 2|) est en $O(1)$.

Le calcul de la complexité d'un parcours de graphe est plus subtil que l'on a l'habitude.
En première approche, nous pouvons la majorer ainsi :

- **Algo 1** :
  - ◇ À chaque étape de la boucle tant que, un nouveau sommet gris devient noir, et aucun sommet noir ne peut changer de couleur (par (CC)). Donc le nombre d'étapes de cette boucle est au plus $|S|$ (le nombre de sommets non noirs est un variant de boucle, comme dit à la partie 4.6.

◇ Le « mettre tous les voisins de s qui sont blancs dans $\mathcal{G}$» cache une boucle qui s'exécute sur tous les voisins de s (pour tester s'ils sont blancs et le cas échéant les enfiler). On peut majorer le nombre d'itérations de cette boucle par $|A|$, même si on voit bien que cette majoration est grossière.

Nous obtenons une complexité en $O\left(|S| \times |V|\right)$.

- **Algo 2 :**
  - ◇ Puisque les doublons sont possibles dans `aVisiter`, la boucle while presque de s'exécuter plus de fois que dans la version précédente. Nous pouvons cependant remarquer que l'insertion d'un nouveau sommet ne se fait qu'à la ligne 10 et que cette ligne ne peut être exécutée qu'une fois pour chaque valeur de $s \in S$ et pour chaque t voisin de s, autrement dit qu'une fois par arête. Ainsi, au plus $|A|$ valeurs peuvent transiter dans `aVisiter`, donc la boucle while tourne au plus $|A|$ fois.
  - ◇ Comme précédemment, la boucle interne « pour tout sommet non noir voisin de s » s'exécute au plus $|A|$ fois.

  Nous obtenons une complexité en $O\left(|A|^2\right)$, ce qui dans la majorité des cas est moins bon que pour la première version.

Mais cette estimation, quoique correcte [1], n'est pas assez précise. En effet, la borne maximale de $|\mathcal{A}|$ opérations pour boucle interne ne peut être atteinte à chaque itération.

Pour obtenir une estimation plus précise, nous allons étudier séparément chaque opération élémentaire et voir combien de fois au maximum elle peut être exécutée durant toute l'exécution de l'algorithme.

On se base sur l'algorithme 2 car c'est le plus simple à implémenter, vérifiez que le résultat sera le même sur l'algorithme 1.

Avant de commencer, détaillons la boucle interne :

```
pour tout voisin t de s :
    si t n'est pas noir :
        mettre t dans aVisiter
peindre s en noir
```

Cette boucle est effectuée au plus une fois pour tout sommet $s \in S$, car elle n'est lancée que si s n'est pas noir et elle se termine en peignant s en noir, et un sommet noir le reste jusqu'à la fin de l'algo par (CC).

Ainsi, les opérations « tester si t est noir » et « mettre t dans `aVisiter` » sont exécutées au plus pour tout $s \in S$ et pour tout t voisin de S, ce qui revient à dire « pour tout $(s,t) \in \mathcal{A}$. Ce qui fait $|\mathcal{A}|$ fois, sur *toute* l'exécution de l'algorithme.

Maintenant voyons la complexité créée par chaque opération de base, ligne après ligne :

- Créer les structures au plus $O\left(|S|\right)$ si on crée des tableaux de $|S|$ cases. (Avec des dictionnaires ce serait en $O(1)$).
- Initialiser n, mettre s_0 dans `aVisiter` : $O(1)$.
- Faire tourner la boucle « tant que » (en pratique, tester si `aVisiter` est vide) : $O(|A|)$ comme expliqué ci-dessus.
- Extraire un sommet de `aVisiter` : exécuté au plus $O(|A|)$ fois. Si cette opération est bien en $O(1)$, elle contribue à la complexité totale à raison de $O(1) \times O(|A|)$ soit $O(|A|)$).
- Tester si s est noir : idem, donc $O(|A|)$.
- Gérer la boucle interne : $O(|A|)$ comme dit ci-dessus.
- Tester si t est noir, et le cas non échéant le mettre dans `aVisiter` : $O(|A|)$.
- Traiter le sommet s, et le peindre en noir : sera exécuté au plus une fois par sommet, donc contribue à la complexité totale en $O(|S|)$.
- Renvoyer le résultat : $O(1)$.

Au total, nous trouvons une complexité en $O\left(|S| + |\mathcal{A}|\right)$.

Remarques :

- Dans le cas extrême où $\mathcal{A} = S^2$ (graphe complet), on retrouve le $O\left(|S|^2\right)$ du calcul grossier.

- Le nombre $|S| + |A|$ est parfois appelé taille du graphe, puisque c'est ce nombre qui indique si le graphe sera compliqué à parcourir. Rappelons que c'est aussi la place mémoire occupée par le graphe si on utilise le tableau des listes d'adjacence, ce qui est le plus adapté pour un parcours de graphe en général.

1. Rappelons que $O\left(|A| \times |V|\right)$ signifie *au plus* de l'ordre de $|A| \times |V|$.

5 Parcours en profondeur

5.1 Programmation

Dans la suite, à titre d'exemple pour implanter nos algorithmes, écrivons des programmes qui renvoient la composante connexe d'un sommet initial.

Le choix le plus simple pour `àVisiter` est d'utiliser le type `List` de Python, et ses opérations `append` et `pop`.

```
def profondeur(g, départ):
    n=len(g)#nb de sommets
    déjà_vu = [False for i in range(n)]
    à_visiter =[départ] # sommets « gris »

    res=[] # Contient les sommets déjà vu. On pourrait se passer de cette variable.

    while len(à_visiter) > 0 :
        s = à_visiter.pop()
        if not déjà_vu[s]:
            res.append(s)
            déjà_vu[s] = True
            for t in g[s]:
                à_visiter.append(t)

    return res
```

5.2 Ordre de parcours des sommets

Suivons sur un exemple l'ordre dans lequel les sommets sont parcourus. À chaque étape, le prochain sommet visité est le dernier qui a été découvert. Ainsi, l'algorithme va-t-il suivre un chemin aussi loin que possible, puis une fois arrivé à une impasse, reprendre sur un nouveau chemin.

Un tel parcours est dit « en profondeur » car on va le plus « profond » possible au bout d'un chemin avant d'en essayer un autre.

On obtient ce comportement car dans `àVisiter`, lorsqu'on en sort un élément, c'est le dernier élément qui y a été mis qu'on récupère.

5.3 Notion de pile

Une structure de données permettant d'insérer et d'extraire des éléments, et telle que l'élément est extrait est le dernier qui y a été inséré s'appelle une structure de « pile ». Pour expliquer ce nom, imaginez une pile d'assiette : l'élément qu'on peut extraire facilement est celai au sommet de la pile, c'est bien le dernier qui a été empilé. On dit qu'il s'agit d'une structure de type « FILO » : « First in, last out ».

Ainsi, les tableaux Python (type « `List` ») munis des opération `pop` et `append` forment une structure de piles.

Les piles interviennent dans de nombreuses autres circonstances, mais le plus souvent elles enregistrent un ensemble de tâches à effectuer. Penser à une pile de dossiers à traiter sur un bureau.

6 Parcours en largeur

6.1 Principe

Un parcours en largeur traite les sommets du plus proche du sommet de départ au plus éloigné.

À chaque étape nous voudrons que le sommet qui devient noir soit le sommet gris le plus proche du départ.

Pour y parvenir, nous ferons aussi en sorte que les prochains sommets blancs qui deviennent gris soient également les sommets blancs les plus proches du départ.

Ainsi, dans `àVisiter`, les sommets les plus « vieux » seront les plus proches du départ.

Et il faut donc à chaque étape traiter le sommet de `àVisiter` le plus « vieux », c'est-à-dire celui qui y a été placé il y a le plus longtemps.

En modifiant le code précédent, il suffit de remplacer la ligne `s = àVIsiter.pop()` par `àVisiter.pop(0)`. Mais cette opération est en $O($`len(àVisiter)`$)$: cette méthode ne serait donc pas efficace.

6.2 File d'attente

Définition 6.1. *Une file d'attente est une structure de donnée permettant deux opérations :*

- *Insérer un élément ;*
- *Extraire un élément. L'élément extrait est alors celui qui y avait été inséré en premier.*

Ainsi une file d'attente est-elle une structure de type FIFO, pour « first in first out ».
On peut créer des files d'attentes efficaces grâce à deux piles **cf exercice : ??**.
Mais Python propose un type de file d'attente dans la bibliothèque `collection` : c'est la classe `deque` (« double ended queues ») : il s'agit en réalité de files à deux bouts. On peut extraire et insérer aux deux bouts. Pour créer une file vide :

```
import collections
test = collections.deque()
```

Ensuite les méthodes `append` et `pop` permettent d'ajouter et d'extraire les éléments comme pour le type `List` (« à droite »). Ceci nous redonne des piles... Cependant il existe aussi `appendleft` et `popleft` qui permettent d'ajouter et d'extraire un élément à gauche. Si on utilise par exemple des `appendleft` et des `pop` on obtient une structure de file d'attente (les éléments circulant « de gauche à droite »).

```
import collections

def nouvelle_file():
    return collections.deque()
def enfile(x,f):
    f.appendleft(x)
def defile(f):
    return f.pop()
```

6.3 Programmation

On obtient alors un parcours en largeur ainsi :

```
def largeur(g, départ):
    n = len(g)#nb de sommets
    déjà_vu = [False for i in range(n)]
    à_visiter = nouvelle_file()
    enfile(départ, à_visiter)

    res=[]

    while len(à_visiter) > 0 :
        s = defile(à_visiter)
        if not déjà_vu[s]:
            res.append(s)
            déjà_vu[s] = True
            for t in g[s]:
                enfile(t, à_visiter)

    return res
```

On vérifie sur des exemples que les sommets sont bien parcourus dans l'ordre voulu.
Remarque : On peut préférer utiliser un tableau pour marquer les gris plutôt que les noirs.

6.4 Invariant de boucle spécifique

Notons s_d le sommet de départ. Voici l'invariant de boucle spécifique à un parcours en largeur. Le principe est qu'à chaque instant, il existe d tel que dans `àVisiter` se trouve la fin de la sphère de rayon d centrée en s_d et le début de la sphère de rayon $d+1$. Et les sommets déjà visités (c'est-à-dire noirs) sont ceux de $\overline{\mathcal{B}}(s_d, d)$, hormis les gris.

« Soit n le nombre d'éléments dans `aVisiter`, et $\overleftarrow{(s_0, \dots, s_{n-1})}$ ces éléments, s_0 étant le prochain à sortir. Alors il existe $k \in [\![1, n[\![$ et $d \in \mathbb{N}$ tel que :

1. les sommets $s_0, \dots, s_k$ sont à distance au plus d de sd ;

2. les sommets $s_{k+1}, \dots, s_{n-1}$ sont à distance au plus $d+1$ de sd ;
3. les sommets noirs sont les sommets à distance $d-1$ de s_0 ainsi que les sommets à distance d qui ne sont pas gris. En formule : $\mathcal{N} = \overline{\mathcal{B}}(s_d, d-1) \cup \mathcal{S}(sd, d) \setminus \mathcal{G}$.
4. $\overline{\mathcal{B}}\left(s_d, d\right) = \mathcal{N} \cup \{s_0, ..., s_k\}$. »

Nous noterons (IPL) cette propriété dans la suite.

Remarque : Le « au plus » dans les deux premiers points est dû au fait qu'un même sommet peut figurer en double dans la file après avoir été découvert depuis un sommet à distance $d-1$ mais aussi depuis un sommet à distance d. Ainsi, si on évitait de mettre dans la file un sommet qui y est déjà (comme on le fait dans 6.5), on se compliquerait un peu la programmation mais on se simplifierait un peu la preuve.

Démonstration :

- *Initialisation :* Initialement, `aVisiter` contient s_d, $\mathcal{G} = \{s_d\}$ et $\mathcal{N} = \emptyset$. Donc $d = 0$ et $k = 0$ conviennent.
- *Hérédité :* Supposons notre invariant vrai en début d'une itération. Prenons les notations $n, s_0, \dots, s_{n-1}$ et d comme dans l'énoncé de l'invariant.
  - ◇ *Cas 1, si k>1 :*
    - ◦ Si s_0 est déjà noir, il ne se passe rien à part que s_0 a été enlevé de la file des gris. On vérifie alors que l'invariant est encore vrai en remplaçant k par $k-1$ et en conservant d.
    - ◦ Sinon, s_0 devient noir, et ses voisins non noirs sont ajoutés en fin de file. Ils sont à distance 1 et s_0 et donc à distance $\leqslant d+1$ de sd. De plus, s_0 n'étant pas noir, il n'est pas à distance $\leqslant d$ de sd (point 3 de l'invariant) donc il est à distance d de sd.

      On voit alors que l'invariant est encore vrai en fin d'itération, en remplaçant k par $k-1$ et en conservant d (comme avant donc).
  - ◇ *Cas 2, si k=1* Même chose qu'avant sauf que l'invariant sera vérifié à la prochaine étape avec $d+1$ au lieu de d et en prenant pour k la nouvelle longueur de la file -1.

□

Comme on l'a déjà mentionné, cet invariant de boucle se simplifie lorsqu'on s'assure de ne pas créer de doublons dans `àVisiter`. En effet, dans ce cas, `àVisiter` ne contient que des sommets gris. Elle ne contient donc pas de sommet à distance $\leq d-1$ de s_d par le point 3. Le point 1 indique alors que $s_0, ..., s_k$ sont à distance exactement d de s_d. Et grâce au point 4 et au fait que `àVisiter` n'a pas de doublons, on déduit que $s_{k+1}, ..., s_{n-1}$ sont à distance exactement $d+1$ de s_d.

À titre d'exercice, on peut programmer un parcours un largeur sans file d'attente, en utilisant un simple tableau que contiendra à chaque étape une sphère centrée en s_d. Voir l'exercice 7.

6.5 Application : distance et plus court chemin

Lors d'un parcours en largeur, on visite les sommets par cercles concentriques autour du sommet de départ, donc du plus proche au plus éloigné. Notons s_d le sommet de départ. Si on fixe un sommet s_a d'arrivée, lorsqu'on atteint s_a on l'atteint via un plus court chemin depuis le sommet de départ. On peut donc déduire d'un parcours en largeur un algorithme de recherche de plus court chemin.

6.5.1 Distance

On commence par la distance. On va maintenir un tableau `dist` vérifiant l'invariant de boucle : « Pour tout $i \in [\![1, n[\![$, `dist[i]` contient $+\infty$ ou $d(s_d, i)$. »

Comme on l'a vu lors de la preuve de l'invariant de boucle du parcours en largeur, au moment où on extrait un sommet s de `àVisiter`, s'il était à distance d de s_d, alors ses voisins non gris sont à distance $d+1$ de s_d.

Preuve : Nous admettons le fait que les sommets sont traités du plus proche au plus lointain de s_d. Précisément les sommets gris deviennent noirs dans l'ordre de leur proximité avec s_d, et le sommets blancs deviennent gris de la même manière.

Soit γ un chemin constitué d'un pcc de s_d à s puis de l'arête (s, t). Montrons qu'il s'agit d'un pcc de s_d à t. L'idée est que si t est un voisin blanc de s en passe de devenir gris, c'est qu'il est parmi les blancs à distance minimale de s_d. Supposons qu'il existe un chemin c de s_d à t strictement plus court que γ. Soit u le sommet avant t dans c.

- Si u est noir, alors t est déjà gris par (VN) : impossible.

- Si u est gris, alors $d(s_d, u) = d(s_d, t) - 1 < |\gamma| - 1 = d(s_d, s)$: u est un sommet gris plus proche que s de s_d. Donc u aurait devenir noir avant s : impossible.
- Si u est blanc, c'est un sommet blanc strictement plus proche de s_d que t : il aurait du passer gris avant.

Ceci nous permet de remplir le tableau `dist` à chaque fois que nous insérons un nouvel élément dans `àVisiter`.

En outre, il nous faudra un moyen de savoir si un sommet t est gris. C'est très simple : il suffira de regarder si `dist[t]` vaut -1.

Voici le résultat :

```
def distance(g, sd, sa):
    n = len(g)
    dist = [-1 for i in range(n)]
    à_visiter = nouvelle_file()

    enfile(sd, à_visiter)
    dist[sd] = 0

    while len(à_visiter) > 0 and dist[sa] == -1:
        s = defile(à_visiter)
        # Dans cette version, àVisiter ne contient que des gris -> inutile de regarder si s
            ↪ est noir.
        for t in g[s]:
            if dist[t] == -1 : #t est blanc
                dist[t] = dist[s] + 1
                enfile(t, à_visiter)
    return dist[sa]
```

Liste des modifications par rapport au programme pour calculer une composante connexe :

- Plus de variable `res`.
- Mais un tableau `dist`.
- Le tableau est rempli à chaque fois qu'on découvre un nouveau sommet. On vérifie au préalable si le sommet est effectivement nouveau.
- On est dans une version sans doublon dans `àVisiter`. Donc pas besoin de tester si un sommet extrait est noir.

6.5.2 Plus court chemin

Si nous disposions d'une structure persistante pour enregistrer des chemins, pour laquelle la copie soit en $O(1)$ [2] il suffirait de maintenir un tableau `chemin` tel que pour tout i, `chemin[i]` contient un plus court chemin de s_d vers i si i a été découvert (gris ou noir) et `[]` sinon.

Comme ce n'est pas le cas nativement dans Python, on va garder en mémoire un tableau `pred` tel que pour tout sommet i, `pred[i]` contiendra à chaque instant -1 ou le sommet depuis lequel i a été découvert. Comme le parcours est en largeur, `pred[i]` sera alors un prédécesseur de i le long d'un plus court chemin de s_d vers i. Une fois le parcours en largeur terminé, ou juste une fois s_a atteint, on pourra reconstruire un plus court chemin à partir des informations contenues dans `pred` (méthode de type « programmation dynamique », au programme en option informatique et en tronc commun un deuxième année).

```
def pcc(g, départ, arrivée):

    n=len(g)#nb de sommets
    déjà_vu = [False for i in range(n)]
    à_visiter = nouvelle_file()
    enfile(départ, à_visiter)
    pred = [-1 for i in range(n)]

    while len(à_visiter) > 0 and not déjà_vu[arrivée]:
        s = defile(à_visiter)
        if not déjà_vu[s]:
            déjà_vu[s] = True
            for t in g[s]:
```

2. C'est le cas en Caml !

```
                if pred[t] == -1 : # t est blanc
                    enfile(t, à_visiter)
                    pred[t] = s


    if déjà_vu[arrivée]:
        # Maintenant, on reconstruit le chemin
        s=arrivée
        res=[s]
        #Invariant de boucle : res contient un pcc de s à arrivée
        while s != départ :
            s=pred[s]
            res.append(s)
        res.reverse()
        return res
    else:# Pas de chemin entre départ et arrivée
        return []
```

7 Parcours en profondeur, version récursive

Dans cette version, on crée une fonction auxiliaire récursive qui prend en entrée le sommet actuel.

```
def profondeur_réc(g, départ):
    """ Encore une fois on renvoie la composante connexe de départ dans g."""
    n = len(g)
    déjà_vu = [False for i in range(n)]
    res = []

    def visite_sommet(s):
        if not déjà_vu[s]:
            res.append(s)
            déjà_vu[s] = True
            for t in g[s]:
                visite_sommet(t)

    visite_sommet(départ)
    return res
```

Cette méthode permet de se passer de la variable `aVisiter` des exemples précédents, et ainsi d'obtenir un code un peu plus court. Cependant, il y a quand même une pile dans cette affaire : c'est la pile des appels. C'est elle qui se souvient des prochains sommets à visiter. Simplement, cette pile est gérée par Python lui-même. À cause des limitations de la pile des appels de Python, ce programme ne pourra pas être utilisé pour de gros graphes. En Caml en revanche ce sera le plus pratique.

Remarque : Tester si un sommet a est noir avant de le traiter peut être effectué :

- Avant l'appel à `visite_sommet(a)`, c'est-à-dire dans `visite_voisins`.
- Au début de l'appel à `visite_sommet(a)`. Commencer par tester si le sommet en entrée est noir. C'est ce qui a été fait ci-dessus.

Si on veut continuer d'employer le vocabulaire coloré introduit au début du cours :

- Les sommets noirs sont enregistrés grâce à un tableau de booléens `dejaVu` ;
- Les sommets gris sont ceux qui sont dans la pile des appels et qui ne sont pas encore marqués `dejaVu.` ;
- Les sommets blancs sont les autres.

7.1 Intérêt de cette dernière version

Le premier intérêt de cette version est qu'elle est plus courte. Les gris sont gérés automatiquement par la pile des appels de Python, et nous n'avons pas à gérer nous même une pile, file, ou autre. Attention du coup à la limitation de la taille de la pile des appels de Python.

Un deuxième intérêt est qu'on peut facilement lorsqu'on arrive à un sommet savoir de quel sommet on vient. Il suffit de le rajouter en argument supplémentaire aux fonctions récursives.

cf exercice : 11, 10, 9 (question 3).

Remarque : Dans une autre version d'un parcours de graphe, si on a besoin de savoir de quel sommet on est arrivé, on peut enregistrer dans `aVisiter` des couples *(sommet à visiter, sommet depuis lequel on l'a découvert)*.

8 Plus court chemin dans un graphe pondéré

À présent on considère un graphe tel que chaque arête est munie d'un nombre, qu'on appellera son poids. Par exemple, ce poids peut représenter la longueur cette arête. Le but est alors de déterminer le plus court chemin entre deux sommets du graphe.

Notons tout de suite que si le graphe présente un cycle de poids total négatif, alors pour tout couple de sommet dans la composante connexe de ce cycle, il n'existe pas de plus court chemin. En effet, en tournant en rond sur ce cycle, on peut obtenir des chemins de poids total arbitrairement petit.

Nous supposerons dans la suite que tous les poids sont strictement positifs. On note la conséquence suivante : un chemin de longueur minimal entre deux sommets ne passe au plus qu'une fois par sommet.

8.1 Notations

Pour toute arête $(s,t) \in \mathcal{A}$, on notera $\rho(s,t)$ son poids. Pour tout $(s,t) \in S^2$ qui n'est pas une arête, on notera $\rho(s,t) = \infty$. De plus on posera pour tout $s \in S$, $\rho(s,s) = 0$.

Soit c un chemin, dont on note n le nombre de sommets et $s_0, ... s_{n-1}$ ces sommets. On appelle longueur de c, et on notera $|c|$ le nombre $\sum_{i=0}^{n-1} \rho(s_i, s_{i+1})$. Si c est une suite de sommets qui n'est pas un chemin dans G, on notera $|c| = \infty$.

Pour tous sommets $(s,t) \in S^2$, on notera, si s et t sont reliés, $d(s,t)$ la distance de s à t, c'est-à-dire la longueur d'un plus court chemin de s à t. Si s et t ne sont pas reliés, on notera $\delta(s,t) = \infty$.

Remarques :

- Dans Python, on obtient le flottant ∞ en tapant `float("inf")`.
- A priori, d n'est pas symétrique, car le poids d'une arête n'est pas forcément le même que le poids de l'arête inverse. L'arête inverse peut même ne pas exister si le graphe est orienté.

On rappelle qu'on note $+$ l'opération de concaténation de deux chemins. C'est presque la concaténation des tableaux Python : il faut juste éviter le doublon entre le sommet d'arrivée du premier chemin et le départ du second.

Lemme 8.1. *Soient γ_1 et γ_2 deux chemins tels que le sommet d'arrivée de γ_1 soit le sommet de départ de γ_2. Alors $|\gamma_1 + \gamma_2| = |\gamma_1| + |\gamma_2|$.*

Démonstration : Notons $k, l \in \mathbb{N}^2$ et $s_0, ..., s_k$, $t_0, ..., t_l$ les sommets de γ_1, γ_2 respectivement. Par hypothèse, $s_k = t_0$ et $\gamma_1 + \gamma_2 = (s_0, ..., s_k, t_1, ..., tl)$. On calcule :

$$\begin{aligned}
|\gamma_1 + \gamma_2| &= \sum_{i=0}^{k-1} \rho(s_i, s_{i+1}) + \rho(s_k, t_1) + \sum_{i=1}^{l-1} \rho(t_i, t_{i+1}) \\
&= |\gamma_1| + \rho(t_0, t_1) + \sum_{i=1}^{l-1} \rho(t_i, t_{i+1}) \\
&= |\gamma_1| + \sum_{i=0}^{l-1} \rho(t_i, t_{i+1}) \\
&= |\gamma_1| + |\gamma_2|
\end{aligned}$$

□

Lemme 8.2. *Soit $(s,t) \in S^2$ et γ un plus court chemin de s à t. Soit $u \in \gamma$, et $\gamma_1 : s \rightsquigarrow u$, $\gamma_2 : u \rightsquigarrow t$ tel que $\gamma = \gamma_1 + \gamma_2$.*

Alors γ_1 est un plus court chemin de s vers u, et γ_2 est un plus court chemin de u vers t.

Démonstration : Supposons qu'il existe $\eta_1 : s \rightsquigarrow u$ strictement plus court que γ_1. Alors $\eta_1 + \gamma_2$ est un chemin de s à t strictement plus court que γ, ce qui est absurde. □

8.2 Implémentation

On peut enregistrer la longueur de chaque arête par une petite modification du type choisit pour enregistrer le graphe.

- **Matrice d'adjacence :** il suffit d'enregistrer dans chaque case de la matrice la longueur de l'arête correspondante. Ainsi on crée une matrice m telle que pour tout $(i,j) \in [\![0,n[\![^2$, $m_{i,j} = \rho(i,j)$.
- **Tableau de listes d'adjacence :** Pour tout $i \in [\![0,n[\![$, on enregistre dans `g[i]` une liste de couples *(voisin de i, longueur de l'arête y menant)*. Ou alors un dictionnaire *voisin de i* $\to$ *longueur de l'arête y menant.*

8.3 Algorithme de Floyd-Warshall

Hors programme en tronc commun 2021

L'algorithme de Floyd-Warshall permet de calculer toutes les distances entre deux sommets du graphe. On suppose donnée la matrice d'adjacence m de G.

Notons pour tout $(i,j,k) \in [\![0,n[\![^2 \times [\![0,n]\!]$, $d^k_{i,j}$ la longueur d'un plus court chemin de i à j qui ne passe que par des sommets de $[\![0,k[\![$ si un tel chemin existe, et $+\infty$ sinon. On calcule facilement ces nombres par récurrence sur k :

- **Initialisation :** Pour $k = 0$, on n'a le droit à aucun sommet intermédiaire, donc pour tout $(i,j) \in [\![0,n[\![^2$, $d^0_{i,j} = m.(i).(j)$.
- **Hérédité :** Fixons $(i,j) \in [\![0,n[\![^2$ et $k \in [\![0,n[\![$. Voyons comment calculer $d^{k+1}_{i,j}$ en fonction des $d^k_{\cdot,\cdot}$. Soit c un plus chemin de i à j restant dans $[\![0,k[\![$ (s'il en existe, dans le cas contraire $d^{k+1}_{i,j} = +\infty$).
  - ◇ si c passe par k, il n'y passe qu'une seule fois (graphe à poids positifs, donc pas de cycle dans les plus courts chemins). Donc les autres sommets de c sont dans $[\![0,k-1]\!]$. Donc c est constitué d'un chemin de i à k inclus dans $[\![0,k-1]\!]$ de longueur minimale (par le lemme 8.2), puis d'un chemin de k à j inclus dans $[\![0,k-1]\!]$ de longueur minimale. D'où :
  $$d^{k+1}_{i,j} = d^k_{i,k} + d^k_{k,j}.$$
  - ◇ si c ne passe par k, il est déjà inclus dans $[\![0,k-1]\!]$. Donc :
  $$d^{k+1}_{i,j} = d^k_{i,j}.$$

  Ainsi, $d^{k+1}_{i,j}$ est égal à $d^k_{i,k} + d^k_{k,j}$ ou à $d^k_{i,j}$. Comment savoir lequel des deux ? Rappelons que nous sommes dans le cas où $d^{k+1}_{i,j} \neq \infty$. Donc si l'un des deux nombres $d^k_{i,k} + d^k_{k,j}$ ou $d^k_{i,j}$ est infini, $d^{k+1}_{i,j}$ est égale à l'autre.
  Et si aucun de ces nombres est infini, on peut fixer c_1 un plus court chemin de i vers j à étapes dans $[\![0,k[\![$, et c_2 un chemin formé d'un plus court chemin de i à k à étapes dans $[\![0,k[\![$ et d'un plus court chemin de k vers j à étapes dans $[\![0,k[\![$, de sorte que vu ce qui précède, $d^{k+1}_{i,j} = |c_1|$ ou $|c_2|$. En particulier, $d^{k+1}_{i,j} \geqslant \min(|c_1|,|c_2|)$.
  Mais par ailleurs, c_1 et c_2 sont deux chemins de i vers j à étapes dans $[\![0,k+1[\![$, donc par définition de $d^{k+1}_{i,j}$, $d^{k+1}_{i,j} \leqslant \min(|c_1|,|c_2|)$.
  D'où :
  $$d^{k+1}_{i,j} = \min\left(d^k_{i,k} + d^k_{k,j}\ ,\ d^k_{i,j}\right).$$
  Et cette formule fonctionne aussi lorsque un ou plusieurs de ces flottants valent ∞.

Dès lors l'algorithme de Floyd-Warshall consiste tout simplement à créer une matrice `distance`, à effectuer une boucle « pour k de 0 à $n-1$ » et à faire en sorte que pour tout k, à la fin de l'itération k de la boucle, pour tout $(i,j) \in [\![0,n[\![^2$, `distance.(i)[j]` contient $d^k_{i,j}$.

Remarque : Exemple typique de programmation dynamique.

Remarque : On constate que $d^k_{i,k} = d^{k+1}_{i,k}$ et $d^k_{k,j} = d^{k+1}_{k,j}$ (à cause du fait qu'un plus court chemin n'a pas de cycle). De sorte qu'on peut aussi utiliser les formules :

$$d^{k+1}_{i,j} = \min\left(d^k_{i,k} + d^{k+1}_{k,j}\ ,\ d^k_{i,j}\right).$$

ou

$$d^{k+1}_{i,j} = \min\left(d^{k+1}_{i,k} + d^k_{k,j}\ ,\ d^k_{i,j}\right).$$

Autrement dit si lors du calcul de $d^k_{i,j}$ on utilise une case du tableau qui a déjà été mise à jour, cela ne change pas le résultat.

8.4 Algorithme de Dijkstra

8.4.1 Principe

L'algorithme de Dijkstra, inventé en 1959 par Edsger Dijkstra [3], est utilisé pour trouver un plus court chemin entre deux sommets fixés du graphe.

Fixons donc deux sommets s_d et s_a et cherchons un plus cours chemin de s_d vers s_a.

Il s'agit d'une variante d'un parcours en largeur, nous gardons donc le vocabulaire et les notations des sommets noirs, gris, et blancs. Les invariants de boucle (VN), (VG), et (CC) de la partie 4.1 seront toujours en vigueur. Nous voulons toujours parcourir les sommets du plus proche au plus éloigné, autrement dit par cercles concentriques autour de s_d.

On maintiendra un tableau `dist` tel que pour tout $s \in S$, `dist[s]` contiendra à chaque instant la distance actuellement estimée de s_d à s. Il s'agit de la longueur d'un plus court chemin de s_d vers s parmi les chemin déjà découverts, autrement dit la longueur d'un plus court chemin *noir* de s_d vers s. Si aucun chemin noir n'existe de s_d à s (ce qui revient à dire si s est blanc), on mettra $+\infty$.

Voici l'invariant de boucle précis qu'on maintiendra :

1. (DN) Pour tout $s \in \mathcal{N}$, `distance[s]` $= \delta(d, s)$. Autrement dit, `dist[s]` contient $d(s_d, s)$. En outre, cette distance peut être réalisée par un chemin qui reste dans $\mathcal{N}$.
2. (DG) Pour tout $s \in \mathcal{G}$, `dist[s]` contient la longueur du plus petit chemin de s_d à s *passant uniquement par ses sommets de* $\mathcal{N}$.
3. (DB) Pour tout $s \in \mathcal{B}$, `dist[s]` contient ∞.

Pour rappel, les propriétés vérifiées par n'importe quel parcours de graphe :

- (VN) Les voisins d'un sommet noir sont noirs ou gris ;
- (VG) Un sommet gris a au moins un voisin noir ;
- (CC) Les seuls changements de couleur possibles pour un sommet sont de blanc vers gris et de gris vers noir.

Voici le lemme crucial sur lequel repose l'algorithme :

Lemme 8.3. *Supposons que le tableau* `dist` *vérifie effectivement les trois points ci-dessus. Soit s le sommet de $\mathcal{G}$ tel que* `dist[s]` *est minimale.*
Alors `dist[s]` $= d(s_d, s)$.

Démonstration : Soit c un chemin de longueur minimale de s_d à s. Supposons $|c| <$ `dist[s]`. D'après (DG), c passe par un sommet non noir. Soit t le premier sommet non noir rencontré par c. Par (VN), t est gris. Soit c' la partie de c de d jusqu'à t. On a $d(s_d, t) = |c'|$, sans quoi on pourrait raccourcir c' et donc aussi c. En outre, c' ne passe que par des sommets intermédiaires noirs, donc d'après 2, $|c'| =$ `dist[t]`. Ainsi, `dist[t]` $= \delta(d, t)$.
Au final, `dist[t]` $= |c'| \leqslant |c| <$ `dist[s]`.
Donc `dist[t] < dist[s]` ce qui contredit l'hypothèse du lemme. □

Ainsi, dans les conditions du lemme, on va faire passer le sommet s de $\mathcal{G}$ à $\mathcal{N}$, autrement dit le colorier en noir. Pour préserver (DG) et (VN), il faudra colorier en gris les voisins de s non noirs, et mettre à jour leur distance à d.

Notons $\mathcal{N}' = \mathcal{N} \cup \{s\}$ et $\mathcal{G}'$ l'ensemble obtenu en rajoutant à $\mathcal{G}$ les voisins de s qui étaient blancs.
Soit t un voisin de s non noir. Voyons comment mettre à jour `dist[t]` :

- Si t était blanc, c'est qu'il n'y a aucun chemin de d à t qui reste dans $\mathcal{N}$. Ainsi, pour relier d à t en restant dans le noir, la seule possibilité est de prendre un plus court chemin noir de d à s, puis l'arête (s, t). La longueur de ce chemin est $dist[s] + \rho(s, t)$. Nous devons donc exécuter :
  `dist[t] <- dist[s] +` $\rho(s, t)$
- Si t était déjà gris, soit c un chemin restant dans $\mathcal{N}'$ de longueur minimale de d à t. On reproduit le même raisonnement qu'on a déjà utilisé pour l'algorithme de Floyd-Warshall : on distingue selon que c passe par s ou pas.
  - ◇ **Si c ne passe pas par s :** Alors c est aussi un plus court chemin de s_d à s restant dans $\mathcal{N}$. Donc par (DG), `dist[s]` est déjà égal à $|c|$. Rien à faire dans ce cas.

3. En 20mn à la terrasse d'un café avec sa femme dit la légende.

◇ **Si c passe pas par s :** notons c_1 la partie jusqu'à s et c_2 la partie de s à t. Comme les arêtes de G ont des poids positifs, on peut supposer que c n'a pas de cycle. Si c_2 passe par un sommet intermédiaire x, ce dernier est dans $\mathcal{N}$. Donc il existe un plus court chemin de s_d à x restant dans $\mathcal{N}$. On peut alors remplacer le début de c par celui-ci, on obtient un plus court chemin de s_d à t qui ne passe pas par s et on est ramené au premier cas.
Sinon, c_2 est constitué uniquement de l'arête (s,t), Donc $|c| = |c_1|+|c_2| = d(s_d,s)+\rho(s,t) = \texttt{dist[s]}+\rho(s,t)$.

Pour résumer tous les cas, il suffit d'effectuer :

```
dist[s] = min (dist[t],  dist[s] + ρ(s,t))
```

même preuve que pour Floyd-Warshall :

- D'une part la valeur à mettre est $\geqslant min\big(dist[t], (dist[s]+\rho(s,t)\big)$ car on sait que c'est l'un de ces deux nombres.
- D'autre part elle est $\leqslant min\big(dist[t], (dist[s]+\rho(s,t)\big)$ car ces deux nombres sont des longueurs de chemins de s_d à t avec étapes dans $\mathcal{N}'$.

8.4.2 Algo simplifié

Voici une version simplifiée de l'algorithme, dans laquelle on ne se préoccupe pas encore de savoir comment seront enregistrés les sommets noirs, gris, et blancs. On suppose que la commande `extraitMin gris` permet de retirer de `gris` le sommet à distance minimale, et de renvoyer ce sommet.

Entrées : m : matrice d'adjacente d'un graphe g, s_d et s_a : deux sommets de g
Sorties : la distance de s_d à s_a
Variables locales :

- n : entier, nombre de sommets du graphe
- `dist` tableau de n flottants comme ci-dessus
- `gris` type à définir, pour enregistrer les sommets gris
- `fini` booléen qui indique si on a atteint a.

```
1  début
2      n ← nombre de ligne de m
3      distance.(s_d) ← 0
4      fini ← faux
5      Insérer s_d dans gris
6      tant que non fini et gris est non vide :
7          s ← extraitMin gris
8          si s = s_a :
9              fini ← vrai
10         sinon :
11             pour t voisin de s :
12                 si t est blanc :
13                     dist[t] ← dist[s]+m[s][t]
14                     ajouter t dans gris
15                 sinon :
16                     dist[t] ← min dist[t] (dist[s]+m[s][t])
17                 fin
18             fin
19         fin
20     fin
21     Renvoyer dist.(s_a)
22 fin
```

Algorithme 3 : Dijkstra

8.4.3 Implantation à l'aide d'un tas mutable

Les deux opérations que nous effectuons sur l'ensemble `gris` sont ajouter un élément, et extraire un minimum.

Il existe une structure spécialement adaptée à cette situation : c'est le tas (ici un tas-min pour être précis). L'étude de cette structure est inclus dans le programme de l'option informatique, mais non du tronc commun. Nous nous contenterons pour notre part d'utiliser la bibliothèque « heapq » (« heap » signifie « tas » et « q » est pour « queue ». Ainsi cette bibliothèque implémente des files par tas. De telles files ne sont pas des files d'attente, on les appelle des « files de priorité ».) Les extractions de minimum ainsi que les insertions seront en temps logarithmique en le nombre d'éléments dans le tas.

En pratique, un tas est représenté en Python par le type `List` : les éléments sont rangés dans un simple tableau. Les opérations à employer sont `heappush` pour entasser un nouvel élément et `heappop`[4] poun exatraire le minimum.

Afin que le sommet extrait soit le sommet à distance de s_d minimal, et non le sommet de numéro minimal, nous mettrons dans notre tas des couples $(d(s_d, s), s)$.

Nous ne pouvons pas de manière simple éviter les doublons dans `àVisiter`. Ainsi un même sommet figurera peut-être plusieurs fois, avec différentes distances dans ce tas. Mais ce sera toujours la version avec la plus petite distance qui sortira en premier : il conviendra d'ignorer les apparitions suivantes.

```
from heapq import heappop, heappush

def dijkstra(g, sd, sa):
    """
    Entrée : g, graphe orienté
             sa, sd, deux sommets d'icelui
    Sortie : distance de sd à sa
    """

    àVisiter=[(0., sd)]  # On va manipuler àVisiter uniquement via
    # heappush et heappop : ce sera une file de priorité
    dist = {sd: 0.}
    déjàVu = {}


    while len(àVisiter) > 0:
        # prendre le sommet s de àVisiter
        # pour lequel dist est minimal
        d, s = heappop(àVisiter)

        if s not in déjàVu:

            for (t, la) in g[s]:  #la: longueur de l'arête
                # rajouter t dans àVisiter (si besoin)
                # voir si il faut modifier dist[t]
                if t in dist:
                    if dist[s]+la < dist[t]:
                        # Il vaut mieux passer par s
                        dist[t] = dist[s] + la  # màj dist[t]
                        heappush(àVisiter, (dist[t], t))  # màj dans àVisiter
                else:
                    # t était blanc
                    dist[t] = dist[s]+la  # on initialise dist[t]
                    heappush(àVisiter, (dist[t], t))
            déjàVu[s] = True

    return dist.get(sa, float("inf"))
```

Dans la variante suivante, on s'arrête dès qu'on a rencontré le sommet d'arrivée. En outre, on renvoie non plus seulement la distance, mais aussi un plus court chemin. Pour ce, on procède exactement comme dans un graphe non pondéré dans un parcours en largeur.

```
def reconstruire_chemin(pred, sd, sa):
    res=[sa]
    s=sa
```

4. Il y a forcément un jeu de mot à faire avec ce nom.

```
    while s!=sd:
        s=pred[s]
        res.append(s)
    res.reverse()
    return res


def pcc(g, sd, sa):
    déjàVu = {}
    àVisiter = [(0., sd)]
    dist = {sd: 0.}
    pred = {}

    while len(àVisiter) > 0:
        d, s = heappop(àVisiter)
        if s==sa:
            return reconstruire_chemin(pred, sd, sa)

        if s not in déjàVu:
            for t, la in g[s]:
                if d + la < dist.get(t, float('inf')):
                    dist[t] = d+la
                    heappush(àVisiter, (dist[t], t))
                    pred[t] = s
            déjàVu[s] = True

    # Si on finit la boucle sans avoir rencontré sa, c'est que celui-ce n'est
    # pas relié à sd
    return []
```

8.4.4 Complexité

Cet algorithme a la même structure que les parcours de graphe déjà étudié. La différence est que la file d'attente ou la pile devient ici une file de priorité, et la complexité des opération de base (ajout et extraction) passe de $O(1)$ à $O(\log n)$ où n est le nombre d'éléments présents dans cette file de priorité.

Comme vu lors de l'étude de la complexité d'un parcours de graphe dans cette situation, le nombre d'ajouts effectués dans la structure contenant les gris est au plus $|A|$. Cela nous fait un $O(\log|A|)$ pour la complexité des opérations de base.

Mais comme $|A| \leq |S|^2 = O\left(|S|\right)$, on a $\log|A| = O\left(\log|S|\right)$.

Au final on trouve que la complexité de l'algorithme de Dijkstra est $O\left((|A|+|S|)\log|S|\right)$.

8.5 A*

L'algorithme de Dijkstra visite les sommets dans l'ordre de leur proximité avec le sommet de départ ; on peut s'imaginer une sorte de spirale. Or dans de nombreux cas, on suit dans quelle direction se trouve l'arrivée, et on pourrait donc orienter la recherche dans cette direction. C'est l'objet de la variante étudiée ici, appelée « algorithme A^* »

Nous fixons une fonction $h : S \to \mathbb{R}$. En pratique, pour tout $s \in S$, $h(s)$ sera supposée représenter une approximation de $d(s_a, s)$. Cette fonction sera appelée une « heuristique ». Un exemple simple d'heuristique dans le cas d'une recherche d'itinéraire est la distance à vol d'oiseau.

Choisir à chaque étape de visiter le sommet pour lequel l'heuristique est minimal donnerait un algorithme glouton. Très rapide, mais ne donnant en général pas un plus court chemin.

L'algorithme A^* peut être vu comme un intermédiaire entre cet algorithme glouton et Dijkstra : à chaque étape nous choisissons comme prochain sommet gris à visiter celui pour lequel $s \mapsto dist[s] + h(s)$ est minimal.

Théorème 8.4. *Si pour tout $s \in S$, $h(s) \leqslant d(s, s_a)$, alors l'algorithme A^* fournit un plus court chemin entre s_d et s_a.*

Ainsi, à condition que l'heuristique ne surestime jamais la vraie distance, l'algorithme A^* fournit, contrairement à l'algorithme glouton, le bon résultat. Donc dès que nous disposons d'une heuristique fiable, nous avons tout intérêt à l'employer !

En pratique, il suffit au moment de mettre un sommet t dans la file de priorité `àVisiter` de l'associer non pas au flottant `dist[t]` mais à `dist[t]+h(t)`. On peut donc transformer Dijkstra en A^* en changeant une ligne de code ! Attention tout de même : en sortant un sommet s de la file, on ne récupère plus la valeur de `dist[s]` au passage.

```
242 def pcc_A_etoile(g, sd, sa, h):
243     """
244     h est 'lheuristique.
245     """
246     déjàVu = {}
247     àVisiter = [(0., sd)]
248     dist = {sd: 0.}
249     pred = {}
250 
251     while len(àVisiter) > 0:
252         _, s = heappop(àVisiter)  # Ne pas récupérer le flottant devant s : ce 'nest plus
            ↪ dist[s]
253         if s == sa:
254             return reconstruire_chemin(pred, sd, sa)
255 
256         if s not in déjàVu:
257             for t, la in g[s]:
258                 if dist[s] + la < dist.get(t, float('inf')):
259                     dist[t] = dist[s]+la
260                     heappush(àVisiter, (dist[t]+h(t), t))  # modif ici
261                     pred[t] = s
262             déjàVu[s] = True
263     return []
```

Pour conclure voici une utilisation concrète de cet algo. En utilisant la bibliothèque Python `osmnx` on peut récupérer le graphe de n'importe quelle ville du monde. Le graphe obtenu est dans le format de la bibliothèque `networkx` ; les exemple ci-dessous montrent comment on peut le manipuler. Un gros intérêt est que le graphe contient de nombreuses données en plus des longueurs des arêtes, en particulier les coordonnées gps des sommets. Ceci permet de calculer les distances à vol d'oiseau, pour implémenter A^*.

```
1 ''
269 import osmnx, math
270 g = osmnx.graph_from_place("64000 Pau")
271 
272 ## Exemple de manipulation:
273 
274 # Un sommet quelconque:
275 s = list(g.nodes)[0]
276 
277 # ses voisins
278 voisins_de_s = list(g[s].keys())
279 # nb: On peut parcourir les voisins de s par « for t in g[s] »
280 t = voisins_de_s[0]
281 
282 #longueur de l'arête (s,t)
283 g[s][t]["length"]
284 
285 #nom de la rue correspondante
286 g[s][t]["name"]
287 
288 
289 def coords(g, s):
290     """
291     Sortie : le couple (longitude, latitude) du sommet s
292     """
293     return tuple(map(float, (g.nodes[s]["x"], g.nodes[s]["y"])))
294 
295 
296 def distance_euc(g, s, t):
```

```
    """
    Sortie : distance à vol d'oiseau entre les sommets s et t, en mètres.
    """
    R_TERRE = 6360000  # en mètres
    k = math.pi/180
    lon1, lat1 = coords(g, s)
    lon2, lat2 = coords(g, t)
    lat1*=k
    lat2*=k
    lon1*=k
    lon2*=k
    dx = R_TERRE * math.cos(lat1) * (lon2-lon1)
    dy = R_TERRE * (lat2-lat1)
    return (dx**2 + dy**2)**.5


## Dijkstra classique, programmé pour un graphe networkx
# de plus j'affiche le nombre de sommets visité afin de comparer avec A*
def pcc_pour_nx(g, sd, sa):
    déjàVu={}
    àVisiter = [(0., sd)]
    dist={sd: 0.}
    pred={}

    while len(àVisiter)>0:
        d, s = heappop(àVisiter)
        if s==sa:
            print(f"Nb de sommets visités : {len(dist)}")
            return reconstruire_chemin(pred, sd, sa)

        if s not in déjàVu:
            for t in g[s]:#
                la = float(g[s][t]["length"])
                if d + la < dist.get(t, float('inf')):
                    dist[t] = d+la
                    heappush(àVisiter, (dist[t], t))
                    pred[t]=s
            déjàVu[s]=True
    return []


## Et enfin, A*
def pcc_A_etoile(g, sd, sa):
    déjàVu={}
    àVisiter = [(0., sd)]
    dist={sd: 0.}
    pred={}

    while len(àVisiter)>0:
        _, s = heappop(àVisiter)
        if s==sa:
            print(f"Nb de sommets visités : {len(dist)}")
            return reconstruire_chemin(pred, sd, sa)

        if s not in déjàVu:
            for t in g[s]:#
                la = float(g[s][t]["length"])
                if dist[s] + la < dist.get(t, float('inf')):
                    dist[t] = dist[s]+la
                    heappush(àVisiter, (dist[t]+distance_euc(g, t, sa), t))
                    pred[t]=s
            déjàVu[s]=True
    return []
```

Deuxième partie

Exercices

Exercices : graphes

1 Implémentation d'un graphe

Exercice 1. *! Opérations élémentaires sur les graphes

Dans la suite, on note $G = (\mathcal{A}, \mathcal{S})$ le graphe manipulé.

- Pour les questions 1, 3 et 4, on écrira deux versions des fonctions demandées ci-dessous : une pour un graphe décrit par sa matrice d'adjacence, l'autre pour un graphe décrit par son tableau de listes d'adjacence. Pour les suivants, on écrira uniquement une version pour un graphe représenté par un tableau de listes.
- Préciser à chaque fois le type des entrées et des sorties ainsi que la complexité.
- Pour les fonctions signalées par $\star$, on demande en outre une rédaction en français de l'algorithme (avec des « $\forall s \in \mathcal{A}$ » puis des « $\forall t$ voisin de s »);

1. Écrire une fonction qui renvoie le nombre d'arêtes d'un graphe.
2. Écrire une fonction prenant une matrice d'adjacence et un tableau de listes d'adjacence et indiquant si les graphes que représentent ces deux objets sont les mêmes.
3. Écrire une fonction prenant en entrée un graphe et une liste de sommets et vérifiant si cette liste de sommets est un chemin existant dans le graphe.
4. Écrire une fonction `clique` prenant en entrée un entier n et renvoyant le graphe à n sommets tels que pour tout couple (s,t) de sommets distincts, (s,t) est une arête.
5. $\star$ Écrire une fonction `miroir` qui renvoie le graphe retourné : ses sommets sont les mêmes que ceux du graphe d'origine et ses arêtes sont $\{(t,s)\ ;\ (s,t) \in \mathcal{A}\}$.
6. $\star$ Écrire une procédure pour désorienter un graphe. Ainsi pour tout $(i,j) \in S^2$, si $(i,j) \in \mathcal{A}$ faire en sorte que $(j,i) \in \mathcal{A}$.
7. Un sommet est appelé un « puits » lorsqu'il ne mène à aucun autre sommet. Écrire une fonction pour déterminer si un graphe possède un puits.
8. (**) Un sommet est appelé un « puits total » lorsque c'est un puits et qu'il est accessible depuis tous les autres sommets (il y a donc $|\mathcal{S}| - 1$ arêtes qui y aboutissent).
   (a) Démontrer qu'un puits total, s'il existe, est unique.
   (b) $\star$ Écrire une fonction pour renvoyer, s'il en existe, un puits total du graphe passé en argument. La complexité de votre fonction devra être en $O\left(|A| + |S|\right)$.

2 Parcours de graphes quelconques

Pour les exercices suivants, n'importe quel parcours de graphe fonctionne.

Exercice 2. * Graphe connexe

Écrire une fonction pour tester si un graphe est connexe.

Exercice 3. *! Toutes les composantes connexes

Écrire une fonction qui renvoie la liste des composantes connexes d'un graphe.

Exercice 4. ** Calcul de signature

On fixe $n \in \mathbb{N}^*$. Le but de cet exercice est d'écrire une fonction pour calculer la signature d'une permutation. Par commodité, nous appellerons $\mathfrak{S}_n$ l'ensemble des permutations de $[\![0, n[\![$. Une permutation σ sera représentée par un tableau `s` tel que pour tout $i \in [\![0, n[\![$, $\sigma(i) =$ `s[i]`.

Les élèves de l'option informatique peuvent coder les fonctions demandées en Caml.

1. Écrire une fonction naïve basée sur la définition et calculer sa complexité.
2. À présent, on propose de décomposer la permutation étudiée en cycles. Tout au long du calcul, nous allons maintenir un tableau de booléens `dejaVu`, tel que pour tout $i \in [\![0, n[\![$, `dejavu[i]` indique si i a déjà été mis dans un cycle.
   (a) Écrire une fonction `unCycle` prenant en entrée une permutation σ et un entier $i \in [\![0, n[\![$, et renvoyant le cycle de σ qui contient i. Cette fonction prendra en outre le tableau `dejaVu` et le mettra à jour à chaque nouvel élément rajouté dans le cycle.
   (b) En déduire une fonction calculant la décomposition en cycles d'une permutation.
   (c) En déduire une fonction pour calculer la signature d'une permutation. Calculer sa complexité.

3 Piles et files

Exercice 5. ** Parenthésage

Le but est de prendre en entrée une chaîne de caractères et de déterminer si elle est bien parenthésée.

1. Pour commencer on ne prend en compte que les parenthèses classiques "(" et ")". Il n'y a pas besoin de pile, un simple entier contenant le nombre de parenthèses ouvertes mais pas encore fermées rencontrées suffira.
2. Maintenant, on suppose qu'il y a différents types de parenthèses. Par exemple, la chaîne suivante est mal parenthésée : " `{bla (blabla} blu)`".

   On suppose disposer d'un dictionnaire qui associe à toute parenthèse ouvrante la parenthèse fermante correspondante. Par exemple `parentheses = { '(':')', '{':'}', '[':']'}`.

   On propose alors d'utiliser une pile de caractères. Chaque parenthèse ouvrante sera empilée. Et lorsqu'on rencontre une parenthèse fermante, on dépile et on vérifie que le type de parenthèse ouvrante dépilé est le bon.

Exercice 6. ** (option info) Dérécursivisation

Dérécursifier la procédure qui affiche les mouvement nécessaires à la résolution du jeu des tours de Hanoï. Utiliser une pile pour enregistrer les prochains mouvements à faire. Par exemple on pourra mettre dans la pile un quadruplet `(n, d, a, i)` pour signifier qu'il faudra déplacer n anneaux de la tige d vers la tige a en utilisant i comme tige intermédiaire.

4 Parcours de graphe spécifiques

Exercice 7. ** Parcours en largeur sans file d'attente.

Écrire une version d'un parcours en largeur sans file d'attente. À la place, on utilisera un tableau contenant à chaque étape une sphère centrée sur le sommet de départ. Il sera plus pratique d'utiliser un tableau ou un dictionnaire pour retenir les sommets gris.

Exercice 8. * Avec une matrice d'adjacence ?

Quelle serait la complexité d'un parcours de graphe si on représentait le graphe par sa matrice d'adjacence ?

Exercice 9. **! Calcul d'une sphère ou d'une boule

1. Écrire une fonction prenant en entrée un graphe g, un sommet d et un entier n et renvoyant l'ensemble des sommets à distance n de s, autrement dit la sphère $\mathcal{S}(s, d)$.
2. Écrire une fonction renvoyant la liste des sommets accessibles en au plus n arêtes. On calcule donc la boule fermée $\overline{\mathcal{B}}(d, n)$.
3. (***) Écrire enfin une fonction qui renvoie la liste des sommets accessibles en n arêtes, autrement dit pour lesquels il existe un chemin de longueur n y menant.

5 Gros exercices, ou petits problèmes

Exercice 10. ** Labyrinthe

Soit $(n, p) \in \mathbb{N}^2$. Nous allons manipuler des labyrinthes dans une matrice de n lignes et p colonnes. Un labyrinthe sera un graphe dont les sommets sont les $n \times p$ cases de la matrice.

N.B. Un sommet est donc un couple de deux entiers. Il sera alors indispensable d'utiliser un dictionnaire pour enregistrer un graphe. Par exemple si `g` est le dictionnaire des listes d'adjacence d'un graphe, et si `g[(2,3)] = [ (2,2), (3,3)]`, cela signifie que la case $(2, 3)$ est reliée aux cases $(2, 2)$ et $(3, 3)$.

Le fichier `lib_laby.py` fournit un labyrinthe exemple et une fonction pour afficher un labyrinthe.

1. Écrire une fonction prenant deux points et un labyrinthe et renvoyant un chemin reliant ces deux points dans ce labyrinthe.
2. Écrire une fonction `case_valide` prenant une case c et les dimensions (n, p) d'un labyrinthe et indiquant si c est une case du labyrinthe (numéro de ligne dans $[\![0, n[\![$ et numéro de colonne dans $[\![0, p[\![$).
3. Écrire une fonction `voisins` prenant en entrée une case c et les dimensions (n, p) du labyrinthe et renvoyant le tableau des cases voisines de c qui sont dans le labyrinthe.
4. Modifier la fonction précédente afin que les voisins soient renvoyés dans un ordre aléatoire. On pourra utiliser la fonction `shuffle` du module `random`.
5. Écrire une fonction créant le graphe où depuis chaque case on peut aller à chaque case voisine tant qu'elle appartient encore à la matrice. Cette fonction est un échauffement et ne sera pas utilisée dans la suite.

6. On désire à présent créer un labyrinthe aléatoire. Pour ce, on part d'un graphe `laby` sans aucune arête. On effectue un parcours en profondeur du graphe complet créé à la question précédente, et à chaque fois qu'on avance vers un nouveau sommet, on rajoute dans `laby` l'arête utilisée. Il est inutile d'avoir réellement créé en mémoire le graphe complet de la question précédente, l'usage de la fonction `voisins` suffit.

Exercice 11. * (option info) Graphes et arbres**

1. Écrire une fonction qui teste si un graphe non orienté connexe possède un cycle. Quel type de parcours utilisez-vous ?
2. On rappelle qu'un arbre est par définition un graphe non orienté connexe sans cycle. Écrire une fonction qui teste si un graphe est un arbre.
3. Écrire une fonction prenant un arbre décrit par le type Caml `type 'a arbre ='a * 'a arbre list` et renvoie cet arbre décrit comme un graphe par listes d'adjacence. On supposera que les étiquettes des nœuds de l'arbre sont les noms des sommets du graphe à créer (en particuliers, toutes les étiquettes de l'arbre doivent être distinctes)..
4. Enfin, écrire une fonction qui prend en entrée un arbre décrit comme un graphe, et qui renvoie cet arbre décrit par le type `arbre` ci-dessus. On prendra n'importe quel sommet comme racine de l'arbre. On mettra en étiquette de chaque nœud le numéro du sommet correspondant.
5. Soit g un graphe. Un arbre couvrant de g est un arbre inclus dans g et contenant tous ses sommets. Écrire une fonction prenant un graphe et renvoyant un arbre couvrant de ce graphe.

Exercice 12. ** Lire le résultat d'un dé à partir d'une photo

On suppose qu'on a pris en photo une face d'un dé, et qu'on a obtenu une matrice de 0 et de 1 : un 0 indique un pixel blanc, et un 1 un pixel noir.

Écrire une fonction qui renvoie le nombre de points noirs sur la face du dé.

Pour tester votre programme, un fichier `dé.csv` est à disposition : il contient le tableau des pixels d'un dé affichant six points sous forme d'un fichier csv.

6 Graphes pondérés

Exercice 13. *! Plus court chemin par l'algorithme de Dijkstra

Modifier l'algorithme de Dijkstra vu en cours pour qu'en plus de la distance entre les deux sommets, il indique le chemin à prendre.

Exercice 14. * Calcul de boule

Écrire une variante du programme de Dijkstra fait en cours prenant en entrée un sommet s_0 et un flottant d et renvoyant la liste des sommets à distance $< d$ de s_0.

7 Exercices théoriques

Exercice 15. ** Arbres

Par définition, un arbre (déraciné) est un graphe non orienté connexe et sans cycle ; un cycle étant un chemin partant et arrivant au même sommet et n'empruntant pas deux fois la même arête.

Soit (A, S) un graphe non orienté, qu'on notera G. Montrer que les assertions suivantes sont équivalentes :

1. G est un arbre (déraciné) ;
2. Pour tout $(s,t) \in S^2$, il existe un unique chemin simple de s vers t (un chemin simple est un chemin qui n'emprunte pas deux fois une même arête) ;
3. G est connexe, mais pour tout $(s,t) \in A$, $(S, A \setminus \{(s,t),(t,s)\})$ ne l'est plus ;
4. G est acyclique mais pour tout $(s,t) \in S^2 \setminus A$, $\left(S, A \cup \{(s,t),(t,s)\}\right)$ ne l'est plus ;
5. G est connexe et $|A| = |S| - 1$;
6. G est acyclique et $|A| = |S| - 1$.

Exercice 16. * Puissances de la matrice d'adjacence**

1. Soit G un graphe, n son nombre de sommets et M sa matrice d'adjacence, dans laquelle on a mis des 1 ou des 0 pour indiquer la présence ou l'absence d'une arête .
   Montrer que pour tout $k \in \mathbb{N}$, et tout $(i,j) \in [\![0,n[\![^2$, $\left(M^k\right)_{i,j}$ est le nombre de chemins à k arêtes de i jusqu'à j.

2. Soit $n \in [\![2, \infty[\![$, et G le graphe dont la matrice d'adjacence est $\begin{pmatrix} 0 & 1 & & & \\ \vdots & \ddots & \ddots & (0) & \\ \vdots & (0) & \ddots & \ddots & \\ 0 & & & \ddots & 1 \\ 1 & 1 & 0 & \dots & 0 \end{pmatrix}$. On note A cette matrice.

   Calculer A^n de deux manières :

   (a) En interprétant A^n grâce à la question précédente ;

   (b) En calculant le polynôme caractéristique et en utilisant le théorème de Cayley-Hamilton.

Quelques indications

1 7.b) Créer un tableau pour enregistrer le degré entrant de chaque sommet.

2 Prendre n'importe quel parcours et regarder si à son issue tous les sommets ont été traités, autrement dit si tous les sommets sont noirs.

3 Partir d'une fonction qui renvoie la composante connexe d'un sommet de départ.
Appeler cette fonction depuis tout sommet blanc, en prenant soin de conserver les sommets noirs entre deux appels.

4

5
1. Il y a deux choses à vérifier : le nombre de parenthèses ouvrantes non fermées doit être à chaque instant positif, et doit être à la fin nul.
2. Récupérer au préalable la liste des parenthèses ouvrantes et des parenthèes fermantes, et écrire une petite fonction pour récupérer la parenthèse ouvrante qui correspond à une parenthèse fermante.

9 Pour les deux premiers, il est plus naturel d'utiliser des parcours en largeur. On peut procéder comme dans le programme pour calculer la distance entre deux points en maintenant un tableau pour enregistrer la distance de chaque sommet au sommet de départ.
Pour le dernier, on peut s'inspirer de la version récursive du parcours en profondeur. Inutile d'utiliser le tableau pour indiquer les sommets déjà traités, en revanche mettre en argument des fonctions auxiliaires le nombre d'arêtes encore utilisables (n initialement).

10
1. C'est du cours, n'importe quel parcours de graphe fonctionnera.
2.
3. Une méthode pour mélanger une liste : associer à chaque élément un flottant aléatoire, trier selon ces flottants, puis les éliminer.
4.
5. Prendre un parcours en profondeur comme en cours. Simplement, au lieu de `g[s]` pour obtenir les voisins du sommet s, utiliser `voisins(s, n, p)`.

11 1) Lors d'un parcours en profondeur, on a trouvé un cycle dès qu'on retombe sur un sommet déjà visité autre que le sommet depuis lequel l'appel récursif courant a été lancé.
Mettre en argument supplémentaire de `visite_sommet` le sommet précédent. Et dans `visite_voisins` les deux sommets précédents.

12 Cela revient à compter les composantes connexes d'un graphe. Votre première étape sera de créer ce graphe.
On pourra prendre comme sommets les pixels noirs et comme arêtes les couples de pixels noirs côte à côte.

13 Parallèlement au tableau `distance`, maintenir un autre tableau `plusCourtChemin` tel que pour tout sommet s, `plusCourtChemin[s]` contient de le plus court chemin trouvé pour l'instant de la source jusqu'à s.

16 Récurrence.

Quelques solutions

1

4

```python
def signature (s):
    res=0
    for i in range (n):
        for j in range (i+1, n):
                if s[i]>s[j]:
                    res+=1

    if (res % 2)==0 :
        return 1
    else:
        return -1

def unCycle(s,i,DejaVu):
    res=[]
    a=i
    while not DejaVu[a]:
        DejaVu[a]=True
        res.append(a)
        a=s[a]
    return res

def decomposeCycle(s):
    res=[]
    dejaVu=[False for i in range(len(s))]
    for i in range(len(s)):
        if not dejaVu[i]:
            res.append(unCycle(s,i,dejaVu))
    return res

def signatureBis(s):
    decompositions=decomposeCycle(s)
    res=1
    for i in range (len(decompositions)):
        res*=(-1)**(len(decompositions[i])-1)
    return res

def signatureBis(s):
    decompositions=decomposeCycle(s)
    res=1
    for c in decompositions:
        res*=(-1)**(len(c)-1)
    return res
```

5

7

```python
def largeur_par_sphères(g, départ):
    n = len(g)#nb de sommets
    gris = [False for i in range(n)]
    sphere = [départ]
    res = [départ]
    gris[départ]=True
    while len(sphere)>0:
        sphere_suivante = []
        for s in sphere:
            for t in g[s]:
                if not gris[t]:
                    sphere_suivante.append(t)
                    res.append(t)
```

```
                    gris[t] = True
        sphere = sphere_suivante
    return res
```

8 La différence est que la recherche des voisins d'un sommet se fait dans tous les cas en $O\left(|S|\right)$. On obtient une complexité en $O\left(|S|^2\right)$.

9

10

11

12

13

16 1.

2. (a) Pour tout $i \in [\![1, n-1]\!]$, il y a exactement deux chemins issus de i : l'un est $(i, (i)\%n+1, (i+1)\%n+1, \dots, (i+n-1)\%n+1)$ (le % est le reste de la division euclidienne.), et l'autre emprunte le raccourci $n \to 2$, ce qui fait qu'il aboutit à $(i+n)\%n+1$. Les points d'arrivée de ces deux chemins sont i et $i+1$ respectivement. Ainsi, dans A^n, il y a deux coefficients non nuls dans la ligne i, ces deux coefficients valent 1, et ce sont $A_{i,i}$ et $A_{i,i+1}$.
   Remarque : Les formules seraient plus simples en indiçant les lignes et colonnes à partir de 0.

   Pour $i = n$, il y a un troisième chemin, qui emprunte deux fois le raccourci. Ainsi il y a trois coefficients qui valent 1 dans la ligne n, ce sont $A_{n,n}$, $A_{n,1}$ et $A_{n,2}$.

   (b) En développant selon la dernière ligne, on trouve que $\chi(A) = X^n - X - 1$. Par le théorème de Cayley Hamilton, $\chi(A) = 0$ donc $A^n = A + 1$.