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*.

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 arcs 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*.

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. Pour tout chemin γ, on notera sa longueur $|\gamma|$.

La *distance* entre deux sommets s et t est la longueur d'un chemin reliant s à t de longueur minimale. On la note $d(s, t)$.

Si γ_1 et γ_2 sont deux chemins tels que le sommet d'arrivée de γ_1 est le sommet de départ de γ_2, on notera $\gamma_1 @ \gamma_2$ la concaténation de γ_1 avec γ_2 privé de son premier sommet (pour éviter la répétition). On a alors $|\gamma_1 @ \gamma_2| = |\gamma_1| + |\gamma_2|$.

Dans le cas où G n'est pas orienté, on dit qu'il est « connexe » si pour tout $(s, t) \in S^2$, il existe un chemin de s à t. (Pour un graphe orienté, la notion de connexité présente quelques subtilités.)

Proposition 1.1. *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. C'est bien un chemin de G car celui-ci 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. Il est donc réduit à un seul sommet, son départ est aussi son arrivée, et donc $s = t$.

□

2 Implémentation

Notons n le nombre de sommets de G. En pratique, on numérotera les sommets de 0 à $n - 1$. Il y a plusieurs manières de représenter le graphe, en voici deux :

- par *matrice d'adjacence* : on utilise une matrice `m` de format $n \times n$, telle que pour tout $(i, j) \in [\![0, n[\![^2$, `m.(i).(j)` est un booléen qui indique s'il y a une arête pour passer du sommet i au sommet j.
  ```
  type graphe = bool array array; ;
  ```
- par *liste d'adjacence* : on utilise un tableau `t` de longueur n tel que pour tout $i \in [\![0, n[\![$, `t.(i)` contient la liste des voisins du sommet i.
  En Caml : `type graphe = int list array; ;`

Voici quelques avantages et inconvénients de ces deux approches :

- Avec une matrice d'adjacence, tester si un couple (s, t) de sommets est une arête se fait en $O(1)$. Avec le tableau de listes d'adjacences, ce sera en O(nombre de voisins de s).
- En revanche, récupérer tous les voisins d'un sommet est en $O\left(|S|\right)$ pour une matrice d'adjacence et $O(1)$ avec le tableau de 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.

Notons que $|A| \leq |S|^2$. Précisément, si on interdit les arêtes ayant le même sommet de départ et d'arrivée, le nombre maximal d'arêtes est $\frac{|S|(|S|-1)}{2}$. Un graphe qui réalise ce maximum est appelé une « clique ».

Définition 2.1. *Le nombre $|S| + |A|$ est appelé la taille du graphe G.*

Remarque : Dans chacune de ces deux implémentations, un graphe est modifiable, mais en pratique on modifiera rarement un graphe, sauf pour sa création.

3 Parcours d'un graphe : généralités

Tout comme pour un arbre, on peut parcourir un graphe en profondeur ou en largeur. Le parcours en profondeur explore un chemin jusqu'au bout avant de passer à un autre chemin, tandis que le parcours en largeur procède par cercles concentriques autour du point de départ. Nous verrons dans les exercices différentes situations où l'un ou l'autre de ces parcours est mieux adapté.

Comme on aura sans arrêt besoin d'utiliser la liste des voisins d'un sommet, la représentation par listes d'adjacence sera souvent plus pratique ici.

Il y a une difficulté supplémentaire par rapport aux arbres : c'est qu'il faut éviter de repasser sur un sommet déjà visité. On va donc enregistrer les sommets déjà visités.

3.1 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)

Imaginer que la zone noire est la zone déjà explorée, la zone blanche est complètement inconnue, et la zone grise est la frontière entre les deux.

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 démos 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 3.1. *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 3.2. *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. □

Pour la fin de cette partie on suppose G non orienté.

Définition 3.3.

- *Soit $X \in \mathcal{P}(S)$. On dit que X est une partie connexe de G lorsque pour tout $(s,t) \in X^2$, il existe un chemin dans G de s vers t qui reste dans X.*
- *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 deuxième 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 3.4. *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 3.2, 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 3.1, 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. □

3.2 Plan d'un algorithme impératif de parcours de graphe

3.2.1 L'algorithme

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

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

Algorithme 1 : Parcours de graphe, rédaction impérative

3.2.2 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.

- (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'algo 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).

3.2.3 En pratique ?

Les points restant à préciser :

- Comment enregistrons-nous les trois ensembles de sommets noirs, blancs, et gris ?
  - ◇ Pour enregistrer les sommets noirs, le plus simple est un tableau de booléens `dejaVu` de longueur $|S|$. Pour tout sommet s, `dejaVu.(s)` sera `true` si et seulement si s est noir.
    Cependant, si le nombre de sommet est grand, et si nous ne prévoyons pas de les visiter tous (par exemple si vous demandez à votre GPS l'itinéraire Pau-Arudy, il est probable que les données du Mexique n'interviendront pas), c'est un gaspillage de temps et de mémoire que de créer et d'initialiser un tableau de taille $|S|$, on préférera alors une table de hachage ou un arbre de recherche.
  - ◇ 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 ».
  - ◇ La manière d'enregistrer les sommets gris sera le point clef dans chaque algorithme.
- 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 très 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.

3.2.4 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 et gris. 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 occurrence en 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 contenant les gris). Le code devient le suivant. En pratique, ce sera souvent cette version qu'on utilisera.

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 a_visiter pour les sommets à
         visiter
4      Mettre s_0 dans a_visiter.
5      Initialiser des variables
6      tant que 𝒢 ≠ ∅ :
7          Retirer un sommet s de a_visiter
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 a_visiter
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

On remarque que :

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

Dans l'analyse ci-dessous (terminaison et complexité), nous supposons que l'opération « extraire un gris » permet de fournir un sommet vraiment gris en temps $O(1)$, ce qui est le cas si il n'y a pas de doublon dans la structure des gris.

3.2.5 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 munit de l'ordre lexicographique peut aussi servir de variant de boucle, il faut prouver le théorème suivant :

Théorème 3.5. *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.

3.2.6 Complexité

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 3.2.5.

◇ 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 risque 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 :

```
1  pour tout voisin t de s :
2      si t n'est pas noir :
3          mettre t dans aVisiter
```

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'algo.

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).
- Initialiser n, mettre s_0 dans `aVisiter` : $O(1)$.
- Faire tourner la boucle « tant que » (en pratique, tester se `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 $O(|S| + |\mathcal{A}|)$.

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.

3.3 En largeur

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

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

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

3.3.1 Programmation

Pas de surprise : pour un parcours en largeur, nous utilisons une file d'attente pour enregistrer les sommets gris.

```
1
2  let enfile_liste l f =
3      (* Enfile le contenu de la liste l dans la file d'attente f *)
4      List.iter
5      (fun x-> Queue.add x f)
6      l
7  ;;
8
9  let cc_largeur g sd =
10
11   (* Création des variables *)
12   let n=Array.length g in
13   let aVisiter = Queue.create()
14   and dejaVu = Array.make n false
15   and res = ref [] in
16
17   (* Initialisation *)
18   dejaVu.(sd) <- true;
19   enfile_liste g.(sd) aVisiter;
20
21   (* Boucle principale *)
22   while not (Queue.is_empty aVisiter) do
23     let s = Queue.take aVisiter in
24     if not dejaVu.(s) then begin
25         res := s:: !res;
26         dejaVu.(s) <- true;
27         enfile_liste (List.filter (fun x-> not dejaVu.(x)) g.(s) ) aVisiter
28       end
29   done;
30
31   !res
32 ;;
```

La version ci-dessous autorise les doublons dans la file d'attente. Par définition d'une file d'attente, ce sera toujours la première occurrence d'un sommet dans la file qui sera traitée : on peut donc sans incidence décider de ne pas mettre une deuxième fois dans `aVisiter` un sommet qui y est déjà. Pour ce, il faudra maintenir également un tableaux `gris` pour se souvenir des sommets déjà mis dans la file.

Plus généralement, voici le squelette d'un parcours en largeur en Caml :

```
1    let cc_largeur g sd =
2
3   (* Création des variables *)
4   let n=Array.length g in
5   let aVisiter = Queue.create()
6   and dejaVu = Array.make n false
7   and (* ... créer d'autres variables...*)  in
8
9   (* Initialisation *)
10  dejaVu.(sd) <- true;
11  enfile_liste g.(sd) aVisiter;
12
13  (* Boucle principale *)
14  while not (Queue.is_empty aVisiter) do
15    let s = Queue.take aVisiter in
16    if not dejaVu.(s) then begin
17        (* ... Faire des trucs avec s ... *)
18        dejaVu.(s) <- true;
19        enfile_liste (List.filter (fun x-> not dejaVu.(x)) g.(s) ) aVisiter
20      end
21  done;
```

```
22
23    (* ... renvoyer un résultat ... *)
24  ;;
```

3.3.2 Invariant de boucle spécifique

Notons sd le sommet de départ. Voici l'invariant de boucle spécifique à un parcours en largeur :

« 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 : $\mathcal{N} = \overline{\mathcal{B}}(sd, d-1) \cup \mathcal{S}(sd, d) \setminus \mathcal{G}$.»

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 (créer un tableau de booléens appelé `gris_ou_noir`), on se compliquerait un peu la programmation mais on se simplifierait un peu la preuve.

Démonstration :

- *Initialisation :* Initialement, `aVisiter` contient les voisins de s_0 et $\mathcal{N} = \{s_0\}$. Donc $d = 1$ et $k = n-1$ 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 le 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 k la nouvelle longueur de la file -1.

□

3.3.3 Application : plus court chemin

Utilisons un parcours en largeur pour obtenir une fonction qui calcule la distance entre deux sommets. Cette fonction renverra -1 dans le cas où les deux sommets ne sont pas reliés.

Programmation Voici les spécificités de cette fonction :

- Soit sd le sommet de départ. Nous allons maintenir un tableau `distance` tel qu'à chaque instant, pour tout $t \in S$, `distance.(t)` contient $d(sd, t)$ si `t` est noir ou gris, et -1 sinon.
- Au passage, ce tableau permet de savoir si un sommet est noir ou gris ; il peut être utilisé pour éviter les doublons dans la file d'attente comme dit dans 3.3.1. Dès lors, nous sommes sûrs que la file ne contient à chaque instant que des sommets gris. Il est alors inutile de vérifier si le sommet extrait n'est pas noir.
- On peut arrêter la boucle dès qu'on a calculé la distance de sd au sommet d'arrivée.

```
1  let distance sd sa g=
2
3    (* Création des variables *)
4    let n=Array.length g in
```

```
5   let aVisiter = Queue.create()
6   and dejaVu = Array.make n false
7   and distance = Array.make n (-1) in
8
9   (* Initialisation *)
10  distance.(sd) <- 0 ;
11  Queue.add sd aVisiter;
12
13  (* boucle interne *)
14  let rec visite_voisins s = function
15      (* Fonction chargée d'ajouter les voisins de s dans la file d'attente, et de calculer
            ↪ leur distance à sd.*)
16    |[] -> ()
17    |t::q when distance.(t) = -1 -> (* t est blanc *)
18      distance.(t) <- distance.(s)+1;
19      Queue.add t aVisiter;
20      visite_voisins s q
21    |t::q -> visite_voisins s q (* t gris : déjà dans la file d'attente, on le saute.*)
22  in
23
24
25  (* Boucle principale *)
26  while not (Queue.is_empty aVisiter)  && distance.(sa) = -1 do
27    let s = Queue.take aVisiter in
28        dejaVu.(s) <- true;
29        visite_voisins s  g.(s)
30  done;
31
32  distance.(sa);;
```

Démonstration Démontrons que la propriété « Pour tout $t \in S$, `distance.(t)` contient $d(sd, t)$ si `t` est noir ou gris, et -1 sinon. De plus àVisiter ne contient que des sommets gris et n'a pas de doublons.» est bien un invariant :

- *Initialisation :* Après initialisation, le seul sommet non blanc est `sd`, qui est bien à distance 0 de `sd`.
- *Hérédité :* Plaçons nous au début d'une itération où notre invariant est vrai. Soit $n, s_0, \dots, s_{n-1}, d, k$ comme dans la partie 3.3.2 (existe car (IPL) est vrai). On extrait donc s_0 de la file d'attente puis on lance la boucle interne sur ses voisins.

  Comme s était gris (par l'invariant), alors au vu de (IPL), il n'était pas à distance $d-1$ de sd, et donc il est à distance d. Alors par hypothèse de récurrence, `distance.(s0)=d`.

  Ensuite, pour tout t voisin de s_0 :

  - ◇ Si t était noir ou gris, par hypothèse de récurrence `distance.(t)` contenait déjà $d(sd, t)$, et n'est pas modifié.
  - ◇ Sinon, d'après (IPL), t est au moins à distance $d+1$ de sd. Or, le chemin consistant à passer par s_0 via un plus court chemin, puis à prendre l'arête (s_0, t) est de longueur $d+1$. C'est donc un plus court chemin de sd à t, et $d(sd, t) = d+1$, qui est bien la valeur mise dans `distance.(t)`.

  De plus, les seuls sommets rajoutés dans la file sont blancs (c'est-à-dire avaient un `distance` égale à -1), donc n'étaient pas déjà dans la file par hypothèse de récurrence. Donc la file n'a toujours pas de doublons.

  Enfin, le seul sommet qui passe noir est s_0, et comme la file n'a pas de doublons, s_0 n'est plus dans la file, donc aucun nouveau sommet noir dans la file.

Ainsi, cette propriété est bien un invariant de boucle.

Pour conclure quant à la validité de notre programme, il ne reste plus qu'à dire qu'à la fin de la boucle, il y a deux possibilités (prendre la négation de la condition du «tant que») :

- `distance.(sa)!= -1`, ce qui signifie que `sa` est noir, et donc `distance.(sa)` contient bien $d(sd, sa)$
- ou alors la file est vide, ce qui signifie qu'il n'y a plus de gris. D'après le lemme 3.1, cela signifie que sa n'est pas accessible depuis sd.

Remarque : Si vous avez cette preuve à faire dans un sujet de concours, il faudra remplacer «noir» par «tel que `deja_vu` est vrai» et «gris» par « dans la file ».

3.4 En profondeur

Dans un parcours en profondeur, on suit un chemin aussi loin que possible avant de passer à un autre. Concrètement, on explore en priorité les voisins du dernier sommet exploré.

3.4.1 Impératif

On utilise simplement une pile pour contenir les éléments à visiter. De cette manière on explorera en priorité les derniers sommets empilés, c'est-à-dire les voisins du dernier sommet traité.

Ci-dessous un code pour calculer la composante connexe d'un sommet de départ. Je prends soin de renvoyer les sommets dans l'ordre où ils ont été visités, afin de pouvoir vérifier facilement qu'ils ont été vus dans l'ordre d'un parcours en profondeur.

```
let parcours_profondeur_imp g sd=
  let n=Array.length g in
  let dejaVu=Array.make n false in
  let aVisiter=Stack.create()
  and res=ref [] in

  Stack.push sd aVisiter;


        let rec parcours_voisins =
            List.iter (fun x-> Stack.push x p)
        in

  while not( Stack.is_empty aVisiter) do
    let s= Stack.pop aVisiter in
    if not dejaVu.(s) then begin
      dejaVu.(s)<-true;
      res:=s::!res;
      parcours_voisins g.(s) aVisiter
    end
  done;
  List.rev !res;;
```

N.B. Ce code a la même complexité que la version en largeur, puisque l'étude de la complexité avait été faite sans savoir quelle structure serait utilisée pour enregistrer les sommets à visiter. Nous avions simplement supposé que les opérations d'ajout et d'extraction seraient en $O(1)$, ce qui est le cas pour les piles.

En résumé, ce code est tout à fait équivalent à la version en largeur : sur cet exemple c'est juste une question de goût.

On peut alléger la pile en évitant d'y mettre les noirs :

```
let parcours_profondeur_imp g sd=
  let n=Array.length g in
  let dejaVu=Array.make n false in
  let aVisiter=Stack.create()
  and res=ref [] in
  Stack.push sd aVisiter;

        let rec parcours_voisins =
            List.iter (fun x-> Stack.push x p)
        in

  while not( Stack.is_empty aVisiter) do
    let s= Stack.pop aVisiter in
    if not dejaVu.(s) then begin
      dejaVu.(s)<-true;
      res:=s::!res;
      parcours_voisins
                          (List.filter (fun x-> not dejaVu.(x)) g.(s))
                          aVisiter
    end
```

```
  done;
  List.rev !res;;
```

Par contre attention : si on faisait en sorte de ne pas mettre de gris dans la pile (autrement dit, si on n'autorisait pas de doublon dans `aVisiter`), on n'obtiendrait pas un parcours en profondeur d'abord. En effet, un sommet aperçu plusieurs fois se retrouverait trop profond dans la pile et serait traité trop tard. Par exemple sur :

```
let exemple=[| [1;3];
               [2;4];
               [3];
               [];
               [] |];;
```

En partant de 0, 3 devient gris dès le début. Supposons que le parcours parte sur 1, puis 2. Alors il devrait ensuite aller sur 3. Mais lors de la visite de 2 nous ne remettons pas 2 au dessus de la pile, sous prétexte qu'il est déjà quelque part dans la pile, alors nous allons partir sur 4 avant, ce qui ne donne pas un parcours en profondeur d'abord car la branche 0,1,2 aurait pu être prolongée.

3.4.2 Révision : parcours récursif d'un arbre avec nombre de fils non borné

Lorsqu'on décrit un graphe par le tableau des listes de voisins, la situation est très proche de celle d'un arbre de valence non bornée, ou pour chaque nœud on dispose de la liste de ses fils.

C'est pourquoi avant d'écrire une version récursive d'un parcours de graphe, nous allons réviser les parcours d'arbre.

On définit le type :

```
type 'a arbre = Noeud of  'a* 'a arbre list;;
```

Par exemple écrivons une fonction pour tester l'appartenance d'un élément à un arbre :

```
(* En profondeur *)
let rec appartientProf x = function
  (* Cette fonction reçoit un œnud *)
  | Noeud(e, _) when x=e -> true
  | Noeud(_, fils) -> appartientProfForet x fils

and appartientProfForet x = function
  (* Celle-ci reçoit une liste d'arbre. Initialement, la liste des fils du œnud précédent.*)
  |[] -> false
  |a::q -> appartientProf x a || appartientProfForet x q
;;
```

3.4.3 Récursif

Commençons par une version en pseudo-Caml :

```
let parcours_prof g sd=
  let n =Array.length g in
  let dejaVu=Array.make n false in

  let rec aux s=
      if not dejaVu.(s) then begin
      dejaVu.(s) <- true;
      (* ... Faire des trucs avec s ... *)
      Pour tout t voisin de s :
          lancer aux t et faire quelque chose avec le résultat
      end

  in  aux sd ;;
```

Comme précédemment, la boucle interne (« pour tout t voisin de s ») pourra être traduite par une fonction récursive, ou par de la programmation fonctionnelle mais cette dernière méthode est souvent un peu plus difficile ici. Voici ce que donnerait la première. On utilise deux fonctions mutuellement récursives, comme pour le parcours d'un arbre au paragraphe précédent. La première fonction est conçue pour traiter un sommet, la seconde une liste de sommets.

```
let parcours_prof g sd=
  let n = Array.length g in
  let dejaVu=Array.make n false in

  let rec visite_sommet s=
      if not dejaVu.(s) then begin
      dejaVu.(s) <- true;
      (* ... Faire des trucs avec s et visite_voisins g.(s) ... *)

      and visite_voisins = function
      |[] -> ... (* cas d'arrêt *)
      |t :: autres_voisins -> (* Faire quelque chose avec (visite_sommet t) et
          ↪ (visite_voisins autres_voisins) *)

  in  visite_sommet sd ;;
```

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.

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.

Complétons le squelette ci-dessus pour obtenir une fonction qui renvoie la composante connexe du sommet de départ :

```
let parcours_prof g sd=
  let n = Array.length g in
  let dejaVu=Array.make n false in

  let rec visite_sommet s=
      (* Renvoie la liste des sommets accessibles à partir de s et pas encore vus *)
      if not dejaVu.(s) then begin
          dejaVu.(s) <- true;
          s::visite_voisins g.(s)
      else
          []

   and visite_voisins = function
      (* Renvoie la liste des sommets accessibles depuis un des sommets de la liste passée en
          ↪ argument et pas déjà vus *)
      |[] -> []
      |t :: autres_voisins -> (visite_sommet t) @ (visite_voisins autres_voisins)

  in  visite_sommet sd ;;
```

3.4.4 Note sur l'ordre d'évaluation des argument d'une fonction

Considérons la version suivante de la fonction précédente :

```
let parcours_prof2 g sd=
  let n = Array.length g in
  let dejaVu=Array.make n false in

```

```
  let rec visite_sommet s=
      (* Renvoie la liste des sommets accessibles à partir de s et pas encore vus.
      Appelé uniquement sur un sommet par encore traité.*)

      dejaVu.(s) <- true;
      s::visite_voisins g.(s)

   and visite_voisins = function
      (* Renvoie la liste des sommets accessibles depuis un des sommets de la liste passée en
          ↪ argument et pas déjà vus *)
      |[] -> []
      |t :: autres_voisins when not dejaVu.(t) -> (visite_sommet t) @ (visite_voisins
          ↪ autres_voisins)
      |_ :: autres_voisins -> (visite_voisins autres_voisins)

  in  visite_sommet sd ;;
```

La différence est que nous testons si un sommet a déjà été traité juste *avant* de lancer la fonction `visite_sommet` sur celui-ci, alors que dans la version précédente, nous le testons juste *après*, c'est-à-dire en entrée de cette fonction.

Nous constatons sur l'exemple suivant :

```
   (* (non orienté)
 1  -- 3 -- 4 --5
 |     |        |
 0 -- 2 -------6
 *)
let exemple3 =
    [| [1;2];
       [0;3];
       [0;3];
       [1;2;4];
       [3;5];
       [4;6];
       [2;5] |];;
```

Que nous obtenons des sommets en double (`[0 ; 1 ; 2 ; 3 ; 1 ; 4 ; 5 ; 6]` sur ma machine). Pourtant notre tableau `dejaVu` aurait du nous permettre justement d'éviter de passer plusieurs fois par le même sommet. Que s'est-il passé ?

Le problème vient de l'ordre d'évaluation des arguments du `@`. Nous pensions que dans `(visite_sommet t)@ ↪ (visite_voisins autres_voisins)`, serait évalué en premier le `visite_sommet t`. Mais c'est au contraire `visite_voisins autres_voisins` qui l'a été. Dès lors, pour certaines valeurs de t, le sommet t a été visité lors de `visite_voisins autres_voisins` et donc figure dans le résultat de cet appel. En effet, `visite_sommet t` n'ayant pas encore été lancé, t n'était pas encore marqué comme déjà vu. Ensuite, l'appel à `visite_sommet t` marque enfin t comme déjà vu, et le met lui aussi dans le résultat renvoyé.

Si on veut corriger cette fonction, il faut s'assurer de l'ordre d'évaluation des arguments du `@` ainsi :

```
  |t :: autres_voisins when not dejaVu.(t) ->
      let l1 = visite_sommet t
      in l1 @ (visite_voisins autres_voisins)
```

3.4.5 Spécificité d'un parcours en profondeur

L'intérêt d'un parcours en profondeur, dans sa version récursive 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.

En outre, les fonctions auxiliaires peuvent directement renvoyer un résultat, alors que les boucles while des versions impératives ne peuvent que faire muter des variables. Cela peut permettre un code plus court et plus clair. Dans l'exemple de la composante connexe ci-dessus, les fonctions auxiliaires renvoient directement des listes de sommets.

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)*.

4 Plus court chemin dans un graphe orienté

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 le temps nécessaire pour parcourir cette arête. Le but est 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 positifs. On note la conséquence suivante : un chemin de longueur minimal entre deux sommets ne passe au plus qu'une fois par sommet.

4.1 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 $\delta(s,t) = \infty$.

Remarque : Dans Ocaml, on obtient le flottant ∞ en tapant `infinity`.

Remarque : 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.

Lemme 4.1. *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 :

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

□

Lemme 4.2. *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 cours que γ_1. Alors $\eta_1 @ \gamma_2$ est un chemin de s à t strictement plus court que γ, ce qui est absurde. □

4.2 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 $(i,j) \in [\![0, n[\![^2$, $m_{i,j} = \rho(i,j)$.
  Le type Caml serait alors défini par `type graphe_pondere = float array array` .
- **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).*
  Le type Caml est alors défini par `type graphe_pondere = (int*float)list array`.

4.3 Algorithme de Floyd-Warshall

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 4.2), 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.

4.4 Algorithme de Dijkstra

4.4.1 Principe

L'algorithme de Dijkstra, inventé en 1959 par Edsger Dijkstra [2], 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 3.1 seront toujours en vigueur. Nous

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

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 4.3. *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.
  - ◇ **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 $|c| = |c_1| + |c_2| = d(s_d, s) + \rho(s, t) =$ `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 $\mathcal{N}'$.

4.4.2 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

4.4.3 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. La structure la plus adaptée que nous ayons à disposition à cet effet est la file de priorité, implantée par un tas-min. Ceci permettra d'effectuer les extractions de minimum ainsi que les insertions en temps logarithmique en le nombre d'éléments dans le tas.

Mais il faut faire attention à deux points :

- L'élément à extraire n'est pas le sommet le plus petit, mais le sommet dont la distance estimée à d est minimale. Nous pouvons modifier nos tas pour pouvoir indiquer n'importe quelle relation d'ordre ; et ici l'ordre utilisé sera `fun x y -> distance.(x) <= distance.(y)`. Au passage, on programmera un tas-min au lieu des tas-max vus dans le chapitre précédent [3].
- Lorsqu'on change la distance d'un sommet t, il faut pouvoir le déplacer dans le tas pour le mettre à sa nouvelle place. On utilise ici la fonction `remonteASaPlace` déjà vue. Mais pour ça, il faut connaître la position de t dans le tableau qui enregistre le tas. Nous allons créer et maintenir un autre tableau `pos` dont le but sera de retenir la position de chaque sommet gris dans le tableau qui implémente le tas.
  Concrètement, on aura donc pour tout sommet $s \in \mathcal{G}$, `gris.(pos.(s)) = s`.

Un implémentation d'une telle structure de tas-min est disponible dans le fichier `tasMinPourDijkstra.ml`.

3. De toute façon, il suffit d'inversion la relation d'ordre pour passer de tas-min à tas-max.

4.4.4 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, extraction et ici remonteASaPlace) passe de $O(1)$ à $O(\log n)$ où n est le nombre d'éléments présents dans cette file de priorité. Comme le nombre d'éléments présents est au plus $|S|$, cela nous fait un $O(\log |S|)$.

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

4.4.5 Implantation à l'aide d'un tas persistant

Une manière d'éviter les difficultés mentionnées dans la partie 4.4.3, et de réduire le risque d'erreurs (par exemple mettre à jour le tas avant d'avoir mis à jour dist...) est d'utiliser des tas persistants. Le tas contiendra des couples de la forme *(distance estimée depuis sd, le sommet)* .

Au lieu de modifier le tas lorsque nous trouvons une manière plus rapide d'atteindre un gris, nous allons insérer de nouveau le sommet avec sa nouvelle distance à *sd* estimée. De sorte que pour être plus précis, notre tas contiendra des couple *(dist.(s) au moment de l'insertion de ce couple, s)*.

Ainsi un même sommet sera présent plusieurs fois dans le tas. En particulier, le tas pourra contenir des sommets noirs (qui étaient gris au moment de l'insertion et sont devenus noir ensuite), et on prendra soin de vérifier si le sommet extrait est gris avant de le traiter.

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 $|A|$. Ainsi le nombre maximal d'élément dans le tas sera $|A|$, et la complexité des opérations de base sur les tas sera de $O(\log |A|)$ au lieu de $O(\log |S|)$ de la version avec un tas mutable. Mais $|A| \leqslant |S|^2$ donc $\log(|A|) \leqslant 2\log(|S|)$ et $O(\log |A|) = O(\log |S|)$. Ainsi la complexité est-elle toujours la même.

```
#use "tas_persistants.ml";;

type graphe = (int*float) list array;;
type tas = (float*int) arbre;;

let dijk (g:graphe) sd sa =

  (* Création des variables *)
  let n = Array.length g in
  let dist = Array.make n infinity
  and dejaVu = Array.make n false in


  (* boucle interne, chargée de traiter les voisins d'un sommet s qu'on vient de peindre en
     ↪ noir *)
  let rec parcours_voisins s (gris:tas) = function

    |[] -> boucle gris (* fin de la boucle interne, on passe au tour suivant de la boucle
       ↪ externe.*)

    |(t, _) :: autres_voisins when dejaVu.(t) -> (* t est noir *)
      parcours_voisins s gris autres_voisins

    |(t, dst) :: autres_voisins when dist.(t) = infinity -> (* t est blanc *)
      dist.(t) <- dist.(s) +. dst;
      parcours_voisins s (entasse (-. dist.(t), t) gris) autres_voisins (* t a été rajouté
         ↪ dans les gris *)

    |(t,dst) :: autres_voisins -> (* t est gris *)
      if dist.(s) +. dst < dist.(t) then begin (* Il vaut mieux passer par s pour atteindre t
         ↪ *)
        dist.(t) <- dist.(s) +. dst;
        parcours_voisins s (entasse (-. dist.(t), t) gris) autres_voisins (* nouvelle valeur
           ↪ de dist.(t) mise dans le tas. Elle ira alors au dessus de la précédente. *)
        end
      else
        parcours_voisins s gris autres_voisins
```

```
37
38
39      and
40
41
42    (* boucle principale *)
43        boucle (gris:tas) =
44      (* gris : tas des prochains sommets à visiter. Tas de couples (-d, s) où d est le contenu
            ↪ de dist.(s) au moment de l'insertion. Un mêmem sommet peut être présent plusieurs
            ↪ fois. *)
45      if gris = Vide then dist.(sa) (* qui vaut ∞+ *)
46      else let ((_,s), suiteGris) = extraitMax gris in
47           if s = sa then dist.(s)
48           else if dejaVu.(s) then boucle suiteGris (* Possible à cause des doublons dans gris
                 ↪ *)
49           else begin
50               dejaVu.(s) <- true;
51               parcours_voisins s suiteGris g.(s)
52             end
53    in
54
55    (* Initialisation des variables et lancement de la boucle *)
56    dist.(sd) <- 0.;
57    boucle (entasse (0.,sd) Vide)
58  ;;
```

4.4.6 Application concrète

Les fichiers `aretes.csv` et `sommets.csv` contiennent les données d'un graphe routier des Pyrénées Atlantiques. Nous allons les charger pour lancer Dijkstra dessus.

La fonction suivante permet de supprimer le dernier caractère d'une chaîne de caractère. Lorsque la chaîne sera une ligne d'un fichier, le caractère enlevé sera le caractère de fin de ligne `'\n'`. À noter que pour un fichier créé sous windows, il y aurait deux caractères de fin de ligne à supprimer : `"\r\n"`.

```
2  let strip c=
3    let n=String.length c in
4    String.sub c 0 (n-1);;
```

Ensuite la fonction `decoupe_chaine` prend une chaîne `c` et un caractère `s` et renvoie la liste des chaînes obtenues en découpant `c` selon les `s`. C'est donc la méthode `split` de Python.

```
8  let split chaine sep=
9    let n=String.length chaine in
10
11   let rec prochain_morceau i=
12     if i=n || chaine.[i]=sep then i
13     else prochain_morceau (i+1)
14   in
15
16   let rec aux i=
17     if i>=n then []
18     else
19       let j = prochain_morceau i in
20       (String.sub chaine i (j-i)) :: aux (j+1)
21
22   in aux 0;;
```

On passe à la lecture du fichier des sommets. Cette fonction renverra une table de hachage qui associe un indice i à chaque sommet s, ainsi qu'un tableau `t` tel que pour tout i, `t.(i)` contient le sommet d'indice i. Si n est le nombre de sommets, les indices utilisés seront $[\![0, n[\![$.

```
let chargesommets chemin =
   (* Renvoie le tableau indice -> sommet
      et la liste d'asso sommet -> indice *)


  let dico_sommets = Hashtbl.create 42 in
  let f = open_in chemin in

  let rec aux i =
    (* Remplit le dico sommet -> num
       i est le prochain numéro à utiliser.
       Renvoie en outre le nombre de sommets *)
    try
      let ligne = split (input_line f) ';' in
      Hashtbl.add dico_sommets (List.hd ligne) i;
      aux (i+1)
    with
    |End_of_file -> i

  in

  let nb_sommets = aux 0 in
  close_in f;

  (* À présent le tableau i -> nom du sommet *)
  let tab= Array.make nb_sommets "" in

  Hashtbl.iter
    (fun s i -> tab.(i) <- s)
    dico_sommets;


  tab, dico_sommets
;;
```

La fonction `graphe` suivante permet alors d'obtenir le graphe.

```
let graphe  cheminSommets cheminAretes =
  let tab, dico = chargesommets cheminSommets in
  let n = Array.length tab in
  let g = Array.make n []
  and f = open_in cheminAretes in

  let rec aux ()=
    try
      match split (strip (input_line f)) ';' with
      | [s; t; d] -> let i= Hashtbl.find dico s and j = Hashtbl.find dico t and
          ↪ dist=float_of_string d in
         g.(i) <- (j, dist) :: g.(i);
         g.(j) <- (i, dist) :: g.(j);
         aux ()
      |_-> failwith "fichier mal formé"
    with
    |End_of_file -> ()

  in
  aux ();
  close_in f;
  dico, tab, g;;
```

Pour vérifier que tout va bien on peut écrire une fonction `voisins` qui renvoie la liste des voisins d'un sommet. Elle prend en entrée le nom du sommet (et pas son numéro) et renvoie les noms des voisins.

```
let voisins dico tab s=
  List.map
    (fun (j,d)-> (tab.(j),d))
    g64.(Hashtbl.find dico s)
;;
```

On obtient alors le programme final en lançant le programme de Dijkstra sur le graphe qu'on vient de créer. On récupère la liste des numéros des sommets le long d'un plus court chemin : il ne reste qu'à récupérer les noms de ces sommets.

```
#use "dijkstra_persistant.ml";;

let gps d a =
  let i= Hashtbl.find dico64 d and j = Hashtbl.find dico64 a in
  let dist, l= dijk_chemin g64 j i in
  dist, List.map (fun i-> tab64.(i)) l
;;
```

Deuxième partie

Exercices

Exercices : graphes

1 Implémentation d'un graphe

Exercice 1. *! Opérations élémentaires sur les graphes

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

Pour les trois premiers, 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.

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 ») ;
- Une rédaction à l'aide de `List.iter`.

1. Écrire une fonction qui compte le nombre d'arêtes d'un graphe.
2. 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.
3. 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.
4. $\star$ Écrire une fonction `miroir` qui renvoie le graphe retourné : ses sommets sont les mêmes que ceux du graphe d'origine et ses arêtes sont $\{(t, s)\ ;\ (s, t) \in \mathcal{A}\}$.
5. $\star$ 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 $(j, i) \in \mathcal{A}$.
6. 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.
7. (**) 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.

Exercice 2. * Conversion

Écrire les fonction de conversion pour passer d'un graphe décrit par sa matrice d'adjacence au graphe décrit par le vecteur des listes d'adjacence, et réciproquement.

Exercice 3. ** En pratique : numérotation des sommets

En pratique, les sommets d'un graphe sont rarement les entiers de 0 à $n-1$, où n est le nombre de sommets. Il faut donc numéroter les sommets si l'on veut utiliser une matrice d'adjacence ou un tableau de listes d'adjacence comme dans le cours.

Nous supposons ici donnée une liste de couples qui représente un graphe : chaque couple est une arête, et les sommets sont l'ensemble des éléments intervenant dans un de ces couples (ce qui implique qu'il n'y a pas de sommet isolé). Nous ne savons rien du type des sommets. Nous voulons alors, en notant toujours n le nombre de sommets, attribuer à chaque sommet un numéro de 0 à $n-1$, et construire le graphe sous la forme vue en cours.

1. Quelle structure de donnée sera à votre avis la plus adaptée pour enregistrer l'indice associé à chaque sommet ? Et pour enregistrer le sommet associé à chaque indice ?
2. Écrire une fonction `numerotation` prenant en entrée une liste de couples (type `('a * 'a) list`) représentant un graphe, et renvoyant :
   - une fonction `int_of_sommet : 'a -> int` renvoyant l'indice de chaque sommet ;
   - une fonction `sommet_of_int : int -> 'a` renvoyant le sommet associé à un indice ;
   - le tableau de listes d'adjacence `g` du graphe.
3. Reprendre une des fonctions du cours ou du TD et en déduire une version qui prendra en entrée le graphe donné par une liste de couples.

2 Parcours de graphes

Exercice 4. * Graphe connexe
Écrire une fonction pour tester si un graphe est connexe.

Exercice 5. *! Toutes les composantes connexes
Écrire une fonction qui renvoie la liste des composantes connexes d'un graphe.

Exercice 6. *! Reconnaître le parcours
Pour chacun des algorithmes suivants :

- Reconnaître le type de parcours.
- Déterminer son but. Préciser le rôle des arguments et les valeurs renvoyées.
- Donner le rôle et le type des variables utilisées.
- Faire le lien avec le vocabulaire utilisé cours : reconnaître les sommets blancs, gris et noirs.
- L'implémenter en Caml si ce n'est pas déjà fait.

1.

Entrée : un arbre $G = (S, A)$, orienté, un sommet de départ s_0
Sortie : un tableau de booléens, un tableau de prédécesseurs

$F \leftarrow$ `creer_file_vide()`
Enfiler s_0 dans F
$b_s \leftarrow$ Faux pour tout $s \in S$ (* un tableau de booléens pour chaque sommet, tous faux *)
$b_{s_0} \leftarrow$ Vrai
$\pi_s \leftarrow s$ pour tout $s \in S$ (* un tableau de prédecésseurs pour chaque sommet, initialement $\pi[s] = s$ *)
tant que *F est non vide* **:**
  $s \leftarrow$ defiler(F)
  pour *s' voisin de s tel que $b_{s'}$ est Faux* **:**
    $b_{s'} \leftarrow$ Vrai ; $\pi_{s'} \leftarrow s$
    enfiler s' dans F
  fin
fin
Renvoyer b, π

Algorithme 4 : centrale 2018

2. Dans les deux programmes ci-dessous (mines 2016), les fonctions `ajout` et `contient` et la constante `dicoVide` implémentent des dictionnaires persistants. Les fonctions `liste_enfilee`, `defiled` et la constante `fileVide` implémentent des files persistantes. La fonction `recupere_liens` appelée sur l'adresse d'une page internet renvoie les adresses des pages accessibles depuis celle-ci.

```
let rec parcourt depart n =

 let rec aux n aVisiter dejaVisitees =
  if n=0 then []
  else if aVisiter = fileVide then [] (* On a visité le web en entier! *)
  else match defiled aVisiter with
       | page, suite when contient page dejaVisitees -> aux n suite dejaVisitees
       | page, suite   -> let nelles_pages = recupere_liens page in

                  (page, nelles_pages)
                  :: aux
                          (n-1)
                          (liste_enfilee suite nelles_pages)
                          (ajout page true dejaVisitees)

  in
  aux n (enfile depart fileVide) (dicoVide)
;;
```

3.

```
let rec crawler_imp n pageDepart =
  let aVisiter = Queue.create ()
  and dejaVisitees = Hashtbl.create 42
  and res = ref []
  and n_restant = ref n in
  Queue.add pageDepart aVisiter;

  while not (Queue.is_empty aVisiter) && !n_restant>0 do
    let page = Queue.take aVisiter in
    if not (Hashtbl.mem dejaVisitees page) then begin
        let nelles_pages = recupere_liens page in

        List.iter
          (fun p -> Queue.add p aVisiter)
          nelles_pages;

        Hashtbl.add dejaVisitees page true;

        res:= (page, nelles_pages) :: !res;
        decr n_restant

      end
  done;
  List.rev !res
;;
```

4. Dans ce programme (CCP 2016 modifié), le graphe modélise un réseau de transport. Pour toute arête (u, v), on dispose d'un flottant $r(u, v)$ qui représente la capacité maximale de cette arête (quantité maximale de matière qui peut traverser cette arrête sur une période donnée).

   L'appel à `miseAJour(G,`α`)` a pour effet de supprimer du graphe toutes les arêtes (u, v) telles que $r(u, v) \leqslant \alpha$.

   On ne donne pas le rôle de `reconstitue` : à vous de le deviner.

   Entrées :
   Un graphe $G = (V_G, E_G)$, deux sommets s et t, une fonction $r : E_G \to \mathbb{R}^+$

   pour tout $(u, v) \in E_G$ **faire**
   ⌊ $x(u, v) \leftarrow 0$
   Marque$[t] \leftarrow$ vrai
   tant que Marque$[t]$ **faire**
   | **pour tous les** $u \in V_G$ **faire**
   | | Marque $[u] \leftarrow$ faux
   | | Pred $[u] \leftarrow$ NULL
   | Marque $[s] \leftarrow$ vrai
   | $S \leftarrow \{s\}$
   | ********** Phase 1 **********
   | **tant que** $S \neq \emptyset$ et *Non(Marque* $[t]$ *))* **faire**
   | | Choisir $u \in S$
   | | $S \leftarrow S \backslash \{u\}$
   | | **pour tout** v **tel que** $(u, v) \in E_G$ **faire**
   | | | **si** $r(u, v) > 0$ et *Non(Marque* $[v]$ *))* **alors**
   | | | | Pred $[v] \leftarrow u$
   | | | | Marque $[v] \leftarrow$ vrai
   | | | | $S \leftarrow S \cup \{v\}$
   | ********** Phase 2 **********
   | **si** *Marque* $[t]$ **alors**
   | | $P \leftarrow$ **reconstitueP(Pred, s ,t)**
   | | $\alpha \leftarrow \min \{r(u, v)\ ;\ (u, v) \in P\}$
   | | **MiseAJour**(G, α)
   renvoyer P

Exercice 7. * 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 8. *!! Plus court chemin

Modifier la fonction qui calcule la distance entre deux sommets vue en cours en une fonction qui calcule un plus court chemin entre deux sommets. On propose deux méthodes (on note `sd` le sommet de départ du parcours) :

- Maintenir un tableau `chemin` de listes tel que à chaque itération, pour tout sommet `t`, `chemin.(t)` contient un plus court chemin de `sd` à `t` si `t` est noir ou gris, ou `[]` sinon.
- Maintenir un tableau `pred` d'entiers tel que pour tout $t \in S$, `pred.(t)` contient le sommet à partir duquel t a été découvert s'il est noir ou gris, ou -1 sinon. Une fois rempli ce tableau, il est facile de récupérer un plus court chemin.

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.

3 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. Pour tout $(i, j) \in [\![0, n[\![\times [\![0, p[\![$, la case (i, j) sera représentée par le sommet numéro $pi + j$ du graphe.

Le fichier `lib_laby.ml` fournit un labyrinthe exemple et une fonction pour afficher un labyrinthe. On rappelle la marche à suivre pour charger ce fichier dans le toplevel : `#directory "/adresse/de/mon/répertoire/";;` puis `#use "lib_laby.ml";;`.

1. Écrire une fonction prenant deux points et un labyrinthe et renvoyant le chemin reliant ces deux points dans ce labyrinthe.
2. Soit $k \in [\![0, np[\![$ le numéro d'un sommet. Quelles sont les coordonnées (i, j) de la case correspondante ? Quel est le numéro des cases situées à sa droite, sa gauche, au dessus et en dessous ? À quelle condition ces cases appartiennent-elles encore à la matrice ?
3. Écrire une fonction `voisins` prenant en entrée n, p et le numéro k d'une case, et renvoyant la liste des numéros des cases voisines qui sont encore dans la matrice.
4. Écrire une fonction `voisins_desordre` ayant le même rôle que `voisins` mais renvoyant la liste mélangée dans un ordre aléatoire.
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. * 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 `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 numéros des sommets du graphe à créer.
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 affichanc six points sous forme d'un fichier csv.

4 Graphes pondérés

Exercice 13. * 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 $\leq d$ de s_0.

Exercice 14. ** Variantes sur Dijkstra

- Écrire une fonction renvoyant un plus court chemin entre deux ensembles de sommets.
- (***) Écrire une fonction prenant trois ensembles D, A et I et renvoyant un plus court chemin d'un sommet de D vers un sommet de A qui passe par (au moins) un sommet de I.
- (****) Écrire une fonction prenant trois ensembles D, A et I et renvoyant un plus court chemin d'un sommet de D vers un sommet de A qui passe par une arête inclue dans I.

Exercice 15. ** Optimisation de Dijkstra : A^*

On suppose connue une fonction `aVolDOiseau` prenant en entrée deux sommets du graphe et renvoyant un flottant, auquel on pense comme à la distance à vol d'oiseau entre les deux sommets, vérifiant que pour tout $(i,j) \in S^2$, $d(i,j) \geqslant$`aVolSOiseau i j`.

Nous supposerons dans la suite que le tableau contenant les distances estimées au point de départ s'appelle `distance`, que le sommet de départ s'appelle `d` et celui d'arrivée `a`.

1. Une première optimisation simple est la suivante : lorsqu'on explore un sommet s, si `distance.(s) + aVolDOiseau s a > distance.(a)` alors le sommet s est « inutile » : ne pas explorer ses voisins.

2. L'algorithme A^* est une variante de celui de Dijkstra qui explore en priorité les sommets dont la distance à l'arrivée à vol d'oiseau est la plus faible. En pratique, lorsqu'on recherche le sommet gris à distance minimale de la source, on utilisera comme distance non plus la distance estimée, mais la somme de la distance estimée et de la distance à vol d'oiseau. On peut démontrer que cet algorithme fournit toujours un plus court chemin.
   Programmer cet algorithme.

5 Exercices théoriques

Exercice 16. ** 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 17. * 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.

Exercice 18. * Composantes connexes et diagonalisation du laplacien**

Soit $(S, \mathcal{A})$ un graphe qu'on notera G. Soit $n = |S|$, on suppose que $S = [\![0, n[\![$. Soit M sa matrice d'adjacence. Soit $\Delta \in \mathcal{M}_n(\mathbb{Z})$ telle que :

$$\forall (i,j) \in [\![0,n[\![^2,\ \Delta_{i,j} = \left| \begin{array}{ll} \deg(i) & \text{si } i = j \\ -1 & \text{si } i \neq j \text{ et } (i,j) \in \mathcal{A} \\ 0 & \text{sinon} \end{array} \right.$$

($\deg(i)$ est le nombre d'arêtes issues de i.)

Cette matrice s'appelle le laplacien de G. Soit $(e_0, \dots, e_{n-1})$ la base canonique de $\mathbb{R}^n$. On identifiera M et Δ à leurs applications linéaires canoniquement associées sur $\mathbb{R}^n$.

1. Que vaut la somme des coefficients sur une ligne ou sur une colonne de Δ ?
2. En déduire que $\mathrm{Ker}(\Delta) \neq \{0\}$. Donner explicitement un vecteur dans $\mathrm{Ker}(\Delta)$.
3. Soit X une composante connexe de G. Soit E_X le sous-espace de $\mathbb{R}^n$ engendré par $\left\{ e_i \mid i \in X \right\}$. Montrer que E_X est stable par Δ et par M.
4. On reprend les notations précédentes. Montrer que E_X contient un élément non nul du noyau de Δ.
5. (Avec le cours de maths sur la géométrie euclidienne) Démontrer en une ligne que Δ et M sont diagonalisables.
6. Soit $X \in \mathcal{M}_{n,1}(\mathbb{R})$ un vecteur colonne, dont on note $(x_0, \dots, x_{n-1})$ les coefficients. Montrer que :

$$X^T \Delta X = \frac{1}{2} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} M_{i,j} (x_i - x_j)^2$$

7. On suppose ici que G est connexe. Démontrer que $\dim(\mathrm{Ker}(\Delta)) = 1$.
8. Dans le cas général, prouver que la dimension du noyau de Δ est égale au nombre de composantes connexes de G.

Quelques indications

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

3 Pour enregistrer l'indice d'un sommet, prendre un dictionnaire (table de hachage, ABR, voire liste d'association), et pour le sommet d'un indice, un simple tableau fonctionne parfaitement.

4 Prendre n'importe quel parcours et regarder si à son issue tous les sommets ont été traités.

5 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.

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.(k)` pour obtenir les voisins du sommet k, utiliser `voisins_desordre n p k`.

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.

17 Récurrence.

18 6) Remarquer que pour tout i, $\sum_{j=0}^{n-1} M_{i,j}x_i^2 = \deg(i)x_i^2$.

7) Utiliser la question précédente.

Quelques solutions

1

2

3

7 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

1.

```
let trajet depart arrivee laby n p =
  let noir = Array.make (n*p) false in

  let rec visite_sommet k chemin =
    (* chemin : chemin parcouru pour arriver à k, k inclus
     Renvoie [] s'il est impossible d'atteindre arrivee en prolongeant chemin *)
    noir.(k) <- true;
    if k = depart then chemin
    else
      visite_voisins k chemin laby.(k)

  and visite_voisins k chemin = function
    (* chemin : chemin parcouru pour arriver à k, k inclus
     Renvoie [] s'il est impossible d'atteindre arrivee en prolongeant chemin *)
    |[] -> [] (* Pas possible d'atteindre l'arrivée *)
    |l::autres_voisins when not noir.(l) ->
      let essai = visite_sommet l (l::chemin) in
      if essai = [] then
        visite_voisins k chemin autres_voisins
      else
        essai
    |_::autres_voisins -> visite_voisins k chemin autres_voisins

  in
  visite_sommet arrivee [arrivee]
;;
```

2.

3.

```
let voisins n p k=
  (* liste des voisins de la case k dans un labyrinthe de format (n,p) *)
  let i, j = k/p, k mod p in
    (if j>0 then [k-1] else [])
    @ (if j<p-1 then [k+1] else [])
    @ (if i>0 then [k-p] else [] )
    @ (if i < n-1 then [k+p] else [])
;;
```

```
(* La fonction sort de Caml demande une fonction qui à x y associe 1 si x>y, -1 si x < y
   ↪ et 0 sinon *)
(* Mais on peut tout simplement utiliser un bon vieu tri fusion *)
let ordre_pour_caml x y =
  if x>y then 1
  else if x=y then 0
  else -1
;;

List.sort ordre_pour_caml [0;5;6;2;6];;

let melanged l =
```

```
  (* Renvoie l mélangée *)
  let avec_float = List.map
                     (fun x -> (Random.float 1.,x))
                     l
  in
  List.map
    snd
    (List.sort ordre_pour_caml avec_float)
;;
melanged [1;2;3;4;5];;


let voisins_desordre n p k=
  melanged (voisins n p k)
;;
```

4.

5.

6.

```
let nouveau_laby n p =

  let noir = Array.make (n*p) false
  and laby = Array.make (n*p) []
  in

  let ajoute_arete k l=
    (* Crée l'arête (k,l) dans laby *)
    laby.(k) <- l::laby.(k);
    laby.(l) <- k::laby.(l)
  in

  let rec visite_sommet k =
    noir.(k) <- true;
    visite_voisins k (voisins_desordre  n p k)

  and visite_voisins k = function
    |[] -> ()
    |l::autres_voisins when not noir.(l) -> begin
      ajoute_arete k l;(* On ajoute un passage de k vers l *)
      visite_sommet l;
      visite_voisins k autres_voisins
      end
    |_::autres_voisins -> visite_voisins k autres_voisins

  in visite_sommet 0;
     laby
;;
```

11

12

14

- Initialement, mettre tous les sommets de l'ensemble de départ gris. Et arrêter l'algo dès qu'un sommet de l'ensemble d'arrivée a été peint en noir.
- Une fois noircis tous les sommets de I, réinitialiser le tableau `dist`, en ne conservant que les valeurs des sommets de I. Relancer alors un Dijkstra, en prenant I comme ensemble initial de sommets gris. Garder une sauvegarde du tableau permettant de retrouver les plus courts chemins jusqu'à ceux-ci.
- J'ignore actuellement comment faire et serai très reconnaissant à qui me l'indiquera !

15

17 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 $A^n = A + 1$.

18 1. La somme vaut 0.

2. On constate que le vecteur $(1, \dots, 1)$ est dans le noyau de Δ.
3. Soit $i \in X$. On a $\Delta(e_i) = \deg(i)e_i - \sum_{j \text{ voisin de } i} e_j \in E_X$.
4. Soit $e_X = \sum_{i \in X} e_i$. On constate que $\Delta(e_X) = 0$.
5. Les matrices M et Δ sont symétriques réelles (G n'est pas orienté !) donc diagonalisables par le théorème spectral.
6. Calcul direct.
7. On a déjà vu que $\dim(\mathrm{Ker}(\delta)) \geqslant 1$. Réciproquement, soit $X \in \mathrm{Ker}(\Delta)$, notons $(x_0, \dots, x_{n-1})$ ses coefficients. On a par la question précédente $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} M_{i,j}(x_i - x_j)^2 = 0$. Lorsqu'une somme de termes positifs est nulle, c'est que chaque terme est nul. On déduit que pour tout $(i,j) \in S^2$, si $M_{i,j} \neq 0$ alors $x_i = x_j$. D'où, pour tout $(i,j) \in S^2$, si i et j sont reliés par un chemin, alors $x_i = x_j$. Comme G est connexe, on obtient $x_0 = x_1 = \cdots = x_{n-1}$. Donc $X \in \mathrm{Vect}\big((1, \dots, 1)\big)$.
8. Dans le cas général, on voit que si $X \in \mathrm{Ker}(\Delta)$, alors ses coordonnées sont égales au sein de chaque composante connexe de G. Donc $X \in \mathrm{Vect}\Big\{ e_C \;\Big|\; C \text{ composante connexe de } G \Big\}$.