Chapitre 11. Chaînes et connexité

11.1 Chaînes

Une chaîne dans un graphe est une suite alternée de sommets et d'arêtes :

$$[s_0, a_1, s_1, a_2, s_2, \ldots, a_n, s_n]$$

où l'arête a_i est adjacente au sommet s_{i-1} qui la précède et au sommet s_i qui la suit ; on dit que cette chaîne relie les sommets s_0 et s_n, ou qu'elle représente un chemin de s_0 vers s_n.

Un cycle est une chaîne dont les deux extrémités coïncident ($s_0 = s_n$) ; on peut choisir n'importe quel sommet du cycle comme sommet de départ.

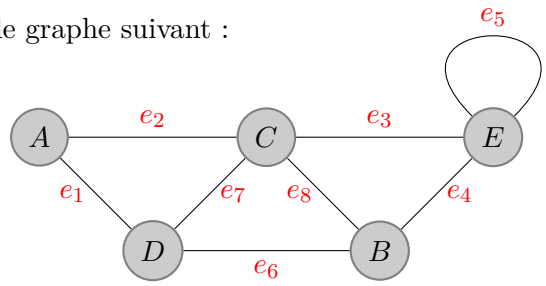
Le nombre d'arêtes n est la longueur de la chaîne (ou du cycle quand la chaîne se trouve être un cycle). Une chaîne peut être réduite à un seul sommet, elle est alors de longueur nulle. C'est d'ailleurs même un cycle.

Une chaîne est élémentaire si ses sommets et arêtes sont *distincts* deux à deux, sauf les sommets extrêmités qui peuvent être égaux (c'est dans ce cas un cycle élémentaire). Plus formellement :

$$\begin{aligned}&\Gamma = [s_0, a_1, s_1, a_2, s_2 \ldots a_n, s_n] \text{ est élémentaire si et seulement si}\\ &\quad \forall\ 0 \leq i < j \leq n, s_i \neq s_j \text{ ou } \{i, j\} = \{0, n\}\\ &\quad \text{et } \forall\ 1 \leq i < j \leq n, a_i \neq a_j\end{aligned}$$

Il ne peut donc y avoir d'allers-retours de la forme $[s, a, t, a, s]$ (où a désigne une arête qui relie les sommets s et t).

Exercice 11.1.1 Soit le graphe suivant :



1. Donner plusieurs exemples de chaînes élémentaires entre A et E.
2. Donner un exemple de chaîne non élémentaire entre A et E.
3. Donner les cycles élémentaires de longueur 4 de ce graphe.

11.2 Démonstration par récurrence

Au chapitre précédent nous avons vu la distinction entre preuve directe et preuve par l'absurde. Il y a une sous-catégorie des preuves directes qui est particulièrement puissante : les preuves par récurrence (aussi appelé induction). Elles utilisent deux étapes :

- On vérifie la propriété sur un cas de base. En général c'est plutôt trivial.

- On suppose que la propriété est vraie pour un ensemble de cas donné. On montre alors que la propriété est vraie pour un nouveau cas.

La situation d'utilisation la plus simple est une récurrence sur les entiers : on montre la propriété pour $n = 0$. On suppose que la propriété est vraie pour un n donné, et on montre qu'elle est alors vraie pour $n + 1$. On a alors une preuve directe pour tout $n \geq 0$, puisqu'il suffit de partir du cas de base 0 et d'appliquer la récurrence autant de fois que nécessaire pour parvenir au n souhaité.

Pour simplifier les démonstrations sur les chaînes, on définit deux opérateurs sur les chaînes.

Si $\Gamma_1 = [s_0, a_1, s_1, \ldots, s_n]$ et $\Gamma_2 = [t_0, b_1, t_1, \ldots, t_m]$ avec $s_n = t_0$, on pose la concaténation : $\Gamma_1 + \Gamma_2 = [s_0, a_1, s_1, \ldots, s_n, b_1, t_1, \ldots, t_m]$ dont la longueur $(n+m)$ se trouve être la somme des longueurs de Γ_1 (n) et Γ_2 (m).

Si $\Gamma = [s_0, a_1, s_1, \ldots, a_n, s_n]$, on pose l'inversion : $\overline{\Gamma} = [s_n, a_n, s_{n-1}, \ldots, a_1, s_0]$.

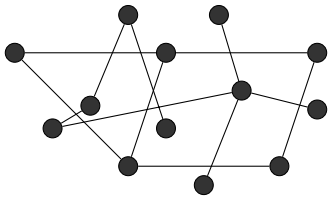
Exercice 11.2.1 Soit Γ une chaîne d'extrémités s et t. On suppose que Γ *n'est pas* élémentaire.

1. Montrer que Γ contient un cycle de longueur non nulle.
2. En déduire qu'il existe une chaîne de longueur strictement plus petite que celle de Γ, qui relie s et t.
3. En déduire par récurrence sur le nombre de cycles inclus de longueur non nulle que pour toute chaîne Γ formant un chemin de s à t, il existe une chaîne *élémentaire* formant un chemin de s à t.

11.3 Connexité

Un graphe est connexe si, par définition : pour tous sommets s et t, il existe une chaîne qui relie s et t.

Exercice 11.3.1 Le graphe suivant est-il connexe ?



Le graphe du TGV (page 59) est-il connexe ?

Exercice 11.3.2 Indiquer si les propositions suivantes sont vraies ou fausses en justifiant votre réponse (rappel : un sommet isolé est un sommet de degré zéro) :

1. Si un graphe n'a pas de sommet isolé, alors il est connexe.
2. Si un graphe possède un sommet isolé, alors il n'est pas connexe.
3. Pour qu'un graphe à plusieurs sommets soit connexe il est nécessaire que tous ses sommets soient de degré supérieur ou égal à 1.

Exercice 11.3.3

1. La proposition suivante :

   « un graphe est connexe si et seulement si il existe un sommet s_0 qui peut être relié (par des chaînes) à tous les autres sommets »

   est-elle vraie ? Justifier ou donner un contre-exemple.

2. Question de révision : Même question pour la proposition suivante :

   « un graphe est connexe si et seulement si il existe une chaîne passant (au moins une fois) par chaque sommet du graphe ».

11.4 Exercices de révisions et compléments

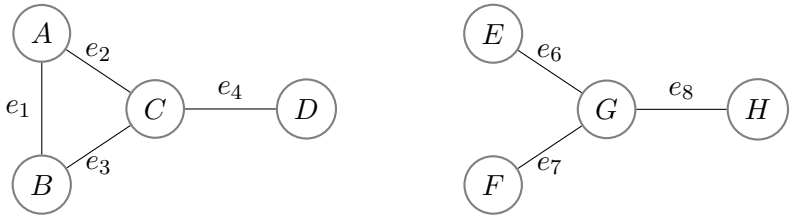
Dans les exercices suivants, une composante connexe d'un graphe $G = (S, A)$ est un sous-ensemble maximal de sommets tels que deux quelconques d'entre eux soient reliés par une chaîne. Pour tout sommet d'un graphe, la composante connexe à laquelle appartient ce sommet est donc l'ensemble des sommets auquel il est relié par au moins une chaîne.

Exercice 11.4.1 Montrer qu'un graphe est connexe si et seulement s'il ne contient qu'une composante connexe.

Exercice 11.4.2 On suppose qu'un graphe G comporte exactement deux sommets de degré impair. En utilisant la formule des poignées de main et en raisonnant par l'absurde, démontrer que ces deux sommets sont dans la même composante connexe.

Exercice 11.4.3 (extrait DST 2014-15 et 2016-17) Dans un graphe connexe, un *isthme* est une arête dont la suppression détruit la connexité du graphe. Autrement dit, une arête e est un isthme d'un graphe connexe G si et seulement si le graphe $G \setminus e$ (le graphe G privé de l'arête e) se décompose en deux composantes connexes.

1. Pour chacun des graphes suivants expliciter tous les isthmes. (Il y en a 4 sur l'ensemble des deux graphes).



2. Soit G un graphe cubique (c'est-à-dire un graphe dont tous les sommets sont de degré 3), connexe, contenant un isthme e. En appliquant la formule de poignées de mains à chacune des composantes connexes, montrer que les deux composantes connexes de $G \setminus e$ ont un nombre impair de sommets.

3. Montrer qu'un graphe connexe dont tous les sommets sont de degré pair ne possède pas d'isthme.

11.5 Algorithmes

Accessibilité

On dit que le sommet t est accessible à partir du sommet s s'il existe une chaîne reliant s et t. En théorie, il est facile de construire progressivement l'ensemble A des sommets accessibles depuis s en utilisant l'algorithme suivant :

1. initialisation : $A = \{s\}$;
2. tant qu'il existe un sommet $x \notin A$ qui est voisin d'un sommet de A, ajouter x à A.

Lorsque cet algorithme se termine, l'ensemble A contient tous les sommets accessibles depuis s.

Pour savoir si le sommet t est accessible depuis s, il suffit donc de tester si le sommet t appartient à l'ensemble A, alors t est accessible depuis s, sinon il ne l'est pas.

Exercice 11.5.1 On travaille sur le graphe G de l'exercice 11.1.1. Appliquer à la main sur papier (pas de python) l'algorithme ci-dessus de construction de l'ensemble A des sommets accessibles à partir de E en choisissant toujours à l'étape 2 le sommet ayant le nom le plus petit dans l'ordre alphabétique. Recommencer en faisant varier le sommet de départ.

Expérimentation

Pour rappel, les fonctions qui permettent de manipuler le marquage d'un sommet sont les suivantes :

`marquerSommet(s:sommet)`
 	
`demarquerSommet(s:sommet)`
	marque le sommet `s`, par exemple :
	`marquerSommet(bdx)`
	démarque le sommet `s`, par exemple :
	`demarquerSommet(bdx)`

`estMarqueSommet(s:sommet) -> bool`	retourne `True` si `s` est marqué, `False` sinon, par exemple : `b = estMarqueSommet(bdx)`

Note : le marquage d'un sommet est indépendant de la coloration d'un sommet.

Exercice 11.5.2 TP Écrire les fonctions suivantes :

1. `toutDemarquer(G:graphe)` démarque tous les sommets du graphe G ;
2. `sommetAccessible(G:graphe)->sommet` retourne un sommet non marqué ayant au moins un voisin marqué, ou bien retourne `None` s'il n'existe pas de tel sommet ;

Dans l'interprète python,

- appeler `toutDemarquer` sur le graphe du TGV,
- appeler `marquerSommet` pour marquer l'un des sommets (ne pas choisir Strasbourg),
- afficher le graphe pour vérifier le résultat,
- appeler `sommetAccessible` et stocker le sommet retourné dans une variable,

- marquer ce sommet,
- afficher le graphe pour vérifier le résultat,
- appeler `sommetAccessible` de nouveau, un autre sommet est retourné,
- continuer ainsi à marquer quelques sommets. Quand cela s'arrêtera-t-il ?

Plutôt que de faire tout cela à la main, écrire une fonction `marquerAccessibles(G:graphe,s:sommet)` qui effectue les étapes suivantes :

1. elle marque d'abord le sommet `s`,
2. elle utilise ensuite une boucle `while` : tant que `sommetAccessible` retourne un sommet (et non `None`), elle le marque.

Pour que cela soit moins coûteux, arrangez votre code de façon à n'appeler `sommetAccessible` qu'une seule fois par tour de boucle `while`.

Exercice 11.5.3 TP Définir les fonctions suivantes :

1. `sommetsTousMarques(G:graphe)->bool` teste si tous les sommets du graphe G sont marqués ;
2. `estConnexe(G:graphe)->bool` teste si le graphe G est connexe. Il suffit de commencer par démarquer tout le graphe, puis piocher un sommet au hasard (utiliser `elementAleatoireListe`), appeler `marquerAccessibles`, et tester enfin si tous les sommets se retrouvent marqués.

Tester avec le graphe du TGV et celui de la figure 8.2, page 62.

Ouvrir le graphe *Power Grid* qui représente le réseau électrique américain, à télécharger via http://dept-info.labri.fr/ENSEIGNEMENT/INITINFO/ressources-initinfo/power.gml. Vérifier que ce réseau est connexe.

Exercice 11.5.4 TP En s'inspirant très largement de la fonction `marquerAccessibles` de l'exercice 11.5.2, écrire une fonction :
`estAccessibleDepuis(G:graphe, s:sommet, t:sommet) -> bool` qui retourne `True` si le sommet t est accessible depuis le sommet s dans le graphe G, et `False` sinon.

Attention : il faut commencer par démarquer tout le graphe, car les sommets que l'on marque restent marqués !

Tester avec différents sommets du graphe de l'Europe.

Il est possible d'améliorer l'algorithme afin qu'il se termine parfois plus rapidement. Pour cela, il suffit de stopper la construction de A dès qu'à l'étape 2 on a $x = t$, car cela signifie alors que t est accessible depuis s. Corriger la boucle de `estAccessibleDepuis` pour effectuer cette optimisation.

Une composante connexe d'un graphe $G = (S, A)$ est un sous-ensemble maximal de sommets tels que deux quelconques d'entre eux soient reliés par une chaîne. Pour tout sommet d'un graphe, la composante connexe à laquelle appartient ce sommet est donc l'ensemble des sommets auquel il est relié par au moins une chaîne. Un graphe connexe est donc un graphe qui possède une seule composante connexe.

Exercice 11.5.5 Montrer que l'ensemble A calculé par l'algorithme d'accessibilité correspond à l'ensemble des sommets appartenant à la composante connexe du sommet s choisi initialement.

Connexité

L'algorithme mis en œuvre dans l'exercice 11.5.3 teste si le graphe G est connexe avec le même genre d'algorithme : on choisit un sommet s, on construit l'ensemble A des sommets accessibles depuis s, puis l'on teste si tous les sommets de G appartiennent à A.

Exercice 11.5.6 Montrer que quand G est connexe le résultat de cet algorithme ne dépend pas du choix du sommet initial s.

Exercice 11.5.7 Dans le cas où l'algorithme termine sans marquer tous les sommets (le graphe G n'est pas connexe), l'ensemble A obtenu est seulement l'une des composantes connexes du graphe G.

Écrire une fonction `sommetNonMarque(G:graphe)->sommet` qui retourne un sommet non marqué.

Un sommet retourné par cette fonction appartient donc à une composante connexe dont les sommets ne sont pas encore marqués. Écrire une fonction `nbComposantesConnexes(G:graphe)->int` qui calcule le nombre de composantes connexes du graphe G.

Exercice 11.5.8 Combien le graphe stocké dans le fichier `europe.dot` possède-t-il de composantes connexes ? Comptez manuellement les composantes connexes en dessinant le graphe. Comparez le résultat obtenu avec celui renvoyé par la fonction `nbComposantesConnexes`. *Note* : la Russie est absente du graphe, d'où une composante connexe surprenante — laquelle ?

Exercice 11.5.9 Améliorer la fonction précédente afin qu'elle colore les composantes connexes de différentes couleurs. Pour ce faire, vous pourrez utiliser une palette préfédinie de couleurs (*cf.* Annexe : palettes page 95).

Arêtes (hors-programme)

Pour traiter les arêtes le module de manipulation de graphes contient les fonctions suivantes :

`listeAretesIncidentes(s:sommet)`	retourne la liste des arêtes issues du sommet `s`, par exemple : `L = listeAretesIncidentes(bdx)`
`sommetVoisin(s:sommet, a:arete)`	retourne le voisin du sommet `s` en suivant l'arête `a`, par exemple : `s = sommetVoisin(bdx, a)`

`marquerArete(a:arete)`
 	
`demarquerArete(a:arete)`
	marque l'arête `a`, par exemple :
	`marquerArete(a)`
	démarque l'arête `a`, par exemple :
	`demarquerArete(a)`

Ainsi le fragment de code

```
for t in listeVoisins(s):
```

peut être remplacé par :

```
for a in listeAretesIncidentes(s):
  t = sommetVoisin(s,a)
```

ce qui est un peu plus compliqué, mais permet d'accéder à l'arête a qui relie les sommets s et t. *Note* : si le graphe n'est pas simple plusieurs arêtes incidentes à s peuvent mener au même voisin t, qui dans ce cas apparaît aussi plusieurs fois dans la liste des voisins de s.

Exercice 11.5.10 Modifier la fonction `sommetAccessible` de l'exercice précédent pour marquer l'arête qui relie ce sommet non marqué à son voisin marqué. Ne pas oublier de modifier aussi la fonction `toutDemarquer` pour démarquer les arêtes en même temps que les sommets.

Tester à nouveau la fonction `estAccessibleDepuis` comme dans l'exercice 11.5.4, puis afficher le graphe de l'Europe. Les arêtes marquées apparaissent avec une épaisseur et une couleur différentes des arêtes non marquées : que remarque-t-on ?

11.6 Exercices et notions complémentaires

Parcours aléatoire

Le module de manipulation de graphes contient la fonction :

`melange(u:list) -> list`	retourne une copie de la liste `u` dont les éléments ont été permutés de façon aléatoire, par exemple : `L = melange(L)`

Par exemple l'instruction :

```
for s in melange(listeSommets(G)):
```

parcourt la liste des sommets du graphe G dans un ordre aléatoire.

Exercice 11.6.1 TP Modifier la fonction `sommetAccessible` de l'exercice précédent pour que le résultat ne dépende plus de l'ordre dans lequel sont rangés les sommets du graphe, ni de l'ordre des arêtes incidentes à un sommet.

Tester à nouveau la fonction `estAccessibleDepuis` comme dans l'exercice 11.5.4, puis afficher le graphe de l'Europe : le chemin entre la Belgique et la Hongrie (par exemple) devrait changer à chaque exécution de l'algorithme.

Complexité de l'algorithme de connexité

Dans cette section on note S l'ensemble des sommets d'un graphe G, et A l'ensemble des arêtes ; $|S|$ désigne alors le nombre de sommets du graphe, et $|A|$ le nombre d'arêtes.

Exercice 11.6.2

1. Lorsque l'on exécute le code suivant :

```
for s in listeSommets(G):
    for t in listeVoisins(s):
        op(s,t)
```

   combien de fois l'opération `op(s,t)` est-elle exécutée ? (pensez à utiliser la formule de poignée de main)

2. En déduire que la complexité *au pire* de la fonction `sommetAccessible` de l'exercice 11.5.2 est *proportionnelle* à $|A| + |S|$.

   Note : les opérations de marquage des sommets (y compris les tests) sont des opérations élémentaires effectuées en temps constant.

3. Estimer de même la complexité (au pire) des fonctions `estConnexe` (exercice 11.5.3) et `estAccessibleDepuis` (exercice 11.5.4).

4. (bonus++) Il existe des méthodes plus sophistiquées pour programmer ces algorithmes, qui fournissent des fonctions de complexité (au pire) proportionnelle à $(|A| + |S|)$. Dans `sommetAccessible` l'inefficacité vient du fait que l'on reparcourt entièrement le graphe pour trouver un sommet non encore marqué avec voisin marqué. Pour être plus efficace, il suffit de se *souvenir* de ce que l'on vient de marquer, en utilisant l'algorithme suivant :
   a) Marquer `s`
   b) Mettre dans `L` la liste contenant seulement `s`
   c) Tant que `L` n'est pas vide :
      - Mettre dans `L2` la liste vide
      - Pour chaque élément `a` de la liste `L` :
        - Pour chaque voisin `b` de `a` :
          - Si `b` n'est pas marqué :
            - Marquer `b`
            - Ajouter `b` à la liste `L2`
      - Mettre la liste `L2` dans `L`

   Écrivez une nouvelle version de `sommetAccessible` utilisant cet algorithme, essayez-le sur le graphe *Power Grid*, constatez la différence de temps d'exécution par rapport à la version précédente.
5. Montrez que chaque sommet n'est traité qu'une seule fois par la première boucle "Pour".
6. Montrez que chaque arête n'est traitée que deux fois par la deuxième boucle "Pour".
7. En déduire que la complexité est $\mathcal{O}(|A| + |S|)$.

Chemin

Exercice 11.6.3 On voudrait maintenant obtenir un chemin. Il existe des algorithmes efficaces pour obtenir un chemin optimal, mais pour simplifier nous allons écrire un algorithme relativement peu efficace, et qui ne trouvera pas forcément un chemin optimal.

L'idée est d'utiliser une fonction *récursive* : pour savoir si l'on peut aller de `s` à `t`, il suffit de tester pour chaque voisin `v` de `s` si l'on peut aller de `v` à `t`, et le cas échéant d'ajouter `s` au chemin obtenu entre `v` et `t`. Puisqu'on cherche un seul chemin, on peut le retourner dès que l'on en a trouvé un.

i. Pourquoi ce n'est pas aussi simple ?

Pour éviter ce problème, il suffit de *marquer* le sommet `s` avant de parcourir ses voisins, et en tête de fonction, vérifier si le sommet est déjà marqué, auquel cas on retourne tout de suite `None`.

ii. Écrire donc la fonction `chemin(G:graphe,s:sommet,t:sommet)->list` qui retourne un chemin de `s` à `t` s'il en existe un (sous forme de la liste des sommets à parcourir), et sinon `None`. N'oubliez pas de nettoyer le graphe au début, il vous faudra donc écrire deux fonctions : l'une qui nettoie le graphe et appelle simplement l'autre, cette dernière s'appelant elle-même récursivement. N'oubliez pas non plus le cas de base où `t == s`.

iii. Déterminer un chemin entre `Espagne` et `Allemagne` dans le graphe de l'Europe.

11.7 L'essentiel du chapitre

Une chaîne dans un graphe est une suite alternée de sommets et d'arêtes :

$$[s_0, a_1, s_1, a_2, s_2, \dots a_n, s_n]$$

où l'arête a_i est adjacente au sommet s_{i-1} qui la précède et au sommet s_i qui la suit.

Un cycle est une chaîne dont les deux extrémités coïncident ($s_0 = s_n$).

Le nombre d'arêtes n est la longueur de la chaîne.

Une chaîne est élémentaire si ses sommets et arêtes sont *distincts* deux à deux, sauf les sommets extrêmités qui peuvent être égaux (c'est dans ce cas un cycle élémentaire).

Un graphe est connexe si pour tous sommets s et t, il existe une chaîne qui relie s et t.

On dit que le sommet t est accessible à partir du sommet s s'il existe une chaîne reliant s et t.

Une preuve par récurrence se démontre en montrant une hypothèse :

- sur un cas de base.
- sur un cas en supposant qu'elle a déjà été montrée sur des cas plus petits.