Graphes

À la fin de ce chapitre, je sais :

- ☞ parcourir un graphe en largeur et en profondeur
- ☞ calculer les composantes connexes d'un graphe
- ☞ faire le lien entre la colorabilité et la satisfiabilité
- ☞ calculer un arbre recouvrant
- ☞ utiliser un graphe biparti

A Parcours d'un graphe

a Parcours en largeur

Parcourir en largeur un graphe signifie qu'on cherche à visiter tous les voisins situé à une même distance d'un sommet avant de parcourir le reste du graphe.

Vocabulary 1 — Breadth First Search ⇜⇝ Parcours en largeur

Le parcours en largeur d'un graphe (cf. algorithme 1) est un algorithme à la base de nombreux développement comme l'algorithme de Dijkstra et de Prim (cf. algorithmes **??** et **??**). Il utilise une file FIFO [1] `F` afin gérer la découverte des voisins dans l'ordre de la largeur du graphe.

Pour matérialiser le parcours en profondeur, on opère en repérant les sommets à visiter. Lorsqu'un sommet est découvert, il intègre l'ensemble des éléments à visiter, c'est à dire la file `F`. Lorsque le sommet a été traité, il quitte la file. Il est donc également nécessaire de garder la trace du passage sur un sommet afin de ne pas traiter plusieurs fois un même sommet : si un sommet a été visité alors il intègre l'ensemble des éléments visités. En Python, cet ensemble `V` peut être implémenté par un type `list` ou un type `set`.

1. First In Fisrt Out

Au fur et à mesure de sa progression, cet algorithme construit un arbre de parcours en largeur dans le graphe. La racine de cet arbre est l'origine du parcours. Comme un sommet de cet arbre n'est découvert qu'une fois, il a au plus un parent. L'algorithme 1 ne renvoie rien mais garde la trace du parcours dans cet arbre dans une structure de type liste (C) qui enregistre le chemin parcouru.

Algorithme 1 Parcours en largeur d'un graphe

```
1 : Fonction PARCOURS_EN_LARGEUR(G, s)                    ▷ s est un sommet de G
2 :     F ← une file FIFO vide                             ▷ F comme file FIFO
3 :     V ← un ensemble vide                               ▷ V comme visités
4 :     C ← une liste vide                                 ▷ C comme chemin
5 :     ENFILER(F,s)
6 :     AJOUTER(V,s)
7 :     tant que F n'est pas vide répéter
8 :         v ← DÉFILER(F)
9 :         AJOUTER(C, v)
10 :        pour chaque voisin x de v dans G répéter
11 :            si x ∉ V alors                             ▷ x n'a pas encore été découvert
12 :                AJOUTER(V, x)
13 :                ENFILER(F, x)
14 :    renvoyer C          ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

La complexité de cet algorithme est lié, comme toujours, aux structures de données utilisées. Soit G un graphe d'ordre n et de taille m implémenté par une liste d'adjacence. On a choisi une file FIFO pour laquelle les opérations ENFILER et DÉFILER sont en $O(1)$. On parcours tous les sommets et chaque liste d'adjacence est parcourue une fois. Ces opérations sont donc en $O(n+m)$.

R Si l'on avait choisi un type tableau dynamique (typiquement le type `list` en Python) au lieu d'une file FIFO pour implémenter F, alors l'opération DÉFILER ferait perdre du temps : en effet, le tableau serait réécrit dans sa totalité à chaque fois qu'une opération DÉFILER aurait lieu car on retirerait alors le premier élément du tableau et il faudrait donc allouer un autre espace mémoire à ce nouveau tableau.

R L'ensemble V n'est pas indispensable dans l'algorithme 1. On pourrait se servir du dictionnaire qui enregistre le parcours. Néanmoins, son utilisation permet de bien découpler la sortie de l'algorithme (le chemin C) de son fonctionnement interne.

b Parcours en profondeur

Parcourir en profondeur un graphe signifie qu'on cherche à emprunter d'abord les arêtes du premier sommet trouvé avant de parcourir les voisins de ce sommet et le reste du graphe.

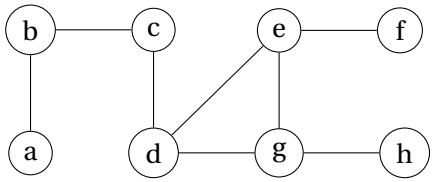

FIGURE 1 – Exemple de parcours en largeur au départ de a : a ⟶ b ⟶ c ⟶ d ⟶ e ⟶ g ⟶ f ⟶ h.

Vocabulary 2 — Depth First Search ⇜⇝ Parcours en profondeur

Le parcours en profondeur d'un graphe (cf. algorithme 2) est un algorithme qui utilise une pile P afin gérer la découverte des voisins dans l'ordre de la profondeur du graphe. Il peut aussi s'exprimer récursivement (cf. algorithme 3).

Algorithme 2 Parcours en profondeur d'un graphe

```
1 : Fonction PARCOURS_EN_PROFONDEUR(G, s)                    ▷ s est un sommet de G
2 :     P ← une file vide                                    ▷ P comme pile
3 :     V ← un ensemble vide                                 ▷ V comme visités
4 :     C ← un liste vide                                    ▷ C comme chemin
5 :     EMPILER(P,s)
6 :     AJOUTER(V,s)
7 :     tant que P n'est pas vide répéter
8 :         v ← DÉPILER(P)
9 :         AJOUTER(C, v)
10 :        pour chaque voisin x de v dans G répéter
11 :            si x ∉ V alors                               ▷ x n'a pas encore été découvert
12 :                AJOUTER(V, x)
13 :                EMPILER(P, x)
14 :    renvoyer C          ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

Algorithme 3 Parcours en profondeur d'un graphe (version récursive)

```
1 : Fonction REC_PARCOURS_EN_PROFONDEUR(G, s, V)             ▷ s est un sommet de G
2 :     AJOUTER(V, s)                                        ▷ s est marqué visité
3 :     pour chaque voisin x de s dans G répéter
4 :         si x ∉ V alors                                   ▷ x n'a pas encore été découvert
5 :             REC_PARCOURS_EN_PROFONDEU(G, x, V)
```

La complexité de cet algorithme est lié, comme toujours, aux structures de données utilisées. Soit G un graphe d'ordre n et de taille m implémenté par une liste d'adjacence. On a choisi une pile LIFO pour laquelle les opérations EMPILER et DÉPILER sont en $O(1)$. On parcours tous

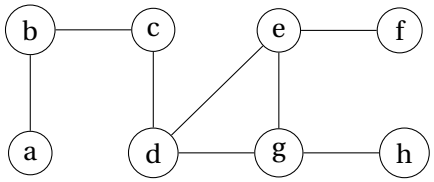

FIGURE 2 – Exemple de parcours en profondeur au départ de a : a ⟶ b ⟶ c ⟶ d ⟶ g ⟶ h ⟶ e ⟶ f

les sommets et chaque liste d'adjacence est parcourue une fois. Ces opérations sont donc en $O(n+m)$.

B Composantes connexes d'un graphe non orienté

Le parcours d'un graphe non orienté à partir d'un sommet de départ permet de lister les composantes connexes de ce sommet. Pour obtenir l'ensemble des composantes connexes d'un graphe non orienté, il suffit donc d'effectuer un parcours complet, c'est à dire en choisissant un nouveau sommet de départ parmi ceux que l'on a pas encore atteint.

L'algorithme 1 de parcours en largeur ne s'applique qu'à un graphe connexe. Mais on peut le modifier simplement pour trouver toutes les composantes connexes comme le montre l'algorithme 4.

Algorithme 4 Composantes connexes d'un graphe non orienté

```
1 : Fonction COMP_CONNEXES(G)
2 :     F ← une file FIFO vide                                  ▷ F comme file FIFO
3 :     V ← ∅                                                   ▷ L'ensemble des sommets visités
4 :     C ← un dictionnaire vide                                ▷ Dictionnaire des composantes connexes
5 :     tant que on n'a pas visité tous les sommets répéter
6 :         s ← un sommet de G pas visité choisi au hasard
7 :         ENFILER(F,s)
8 :         AJOUTER(V,s)
9 :         AJOUTER(C[s],[s])                                   ▷ Composante connexe liée à s
10 :        tant que F n'est pas vie répéter
11 :            v ← DÉFILER(F)
12 :            AJOUTER(C[s], v)
13 :            pour chaque voisin x de v dans G répéter
14 :                si x ∉ V alors                              ▷ x n'a pas encore été découvert
15 :                    AJOUTER(V, x)
16 :                    ENFILER(F, x)
17 :    renvoyer C
```

C Détection de cycles dans un graphe non orienté

Trouver les cycles éventuels d'un graphe est important, notamment lorsqu'on cherche un arbre couvrant de ce graphe. On peut modifier l'algorithme de parcours en profondeur pour détecter les cycles d'un graphe : en effet, le parcours en profondeur produit un arbre et on sait qu'il n'y a pas de cycle dans un arbre. Si, au cours du parcours en profondeur, on détecte une arête qui atteint un sommet déjà visité, alors on a trouvé un cycle. L'algorithme ainsi modifié est donnée ci-dessous.

Algorithme 5 Détection de cycles : le graphe est-il acyclique ?

```
1 : Fonction ACYCLIQUE(G, s, V)                          ▷ s est un sommet de G
2 :     AJOUTER(V, s)                                    ▷ s est marqué visité
3 :     pour chaque voisin x de s dans G répéter
4 :         si x ∉ V alors                               ▷ x n'a pas encore été découvert
5 :             ACYCLIQUE(G, x, V)
6 :         sinon
7 :             renvoyer Faux
8 :     renvoyer Vrai
```

R En fait, il y a d'autres méthodes plus efficaces pour détecter les cycles. L'idée est de garantir que lorsqu'on ajoute un sommet à un arbre, celui-ci fait parti d'une autre composante connexe. La structure de donnée Union-Find permet de gérer l'évolutions de composantes connexes dans un algorithme de manière efficace.

D Tri topologique

E Colorabilité d'un graphe

F Arbre recouvrant

Algorithme de Kruskal. Les détails dimplémentation sont lais- sés à lappréciation du professeur. On fait le lien avec la notion dalgorithme glouton étudiée dans le programme de tronc com- mun.

Recherche d'un cycle calcul des composantes connexes lien avec les tas / file de priorité

G Couplage dans un graphe biparti

Recherche dun couplage de cardinal maximum dans un graphe biparti par des chemins augmentants. On se limite à une ap- proche élémentaire. Lalgorithme de Hopcroft-Karp nest pas au programme.