Algorithmes et structures de données

Adrien Vannson

Table des matières

1 Structures de données **2**
1.1 Union-Find 2

2 Graphes **2**
2.1 Plus courts chemins 2
2.1.1 Floyd-Warshall 2
2.2 Arbre du DFS 3
2.3 Composantes fortement connexes 3
2.3.1 Algorithme de Kosaraju 3
2.3.2 Algorithme de Tarjan 4
2.4 Arbre couvrant minimal 6
2.4.1 Algorithme de Prim 6
2.4.2 Algorithme de Kruskal 7
2.5 Flots 7
2.5.1 Définitions 7
2.5.2 Capacités résiduelles 8
2.5.3 Algorithme d'Edmonds-Karp 8
2.5.4 Théorème Flot-Max / Coupe-Min 8
2.6 Couplages 9
2.6.1 Chemins d'augmentation 9
2.6.2 Recherche d'un couplage maximum 10

3 Chaînes de caractères **10**
3.1 Algorithme de Knuth-Morris-Pratt 10

4 Géométrie **11**
4.1 Enveloppe convexe 11

1 Structures de données

1.1 Union-Find

L'union-find est une structure de données permettant de maintenir à jour une partition d'un ensemble. Il supporte deux opérations principales :

— La recherche de la composante d'un élément : à chaque élément, elle associe un représentant ne dépendant que de la composante dans laquelle il se trouve.

— La fusion des composantes dans lesquelles se trouvent deux éléments.

Pour cela, il utilise une structure de forêt dans laquelle chaque noeud correspond à un élément et contient deux informations :

— Le père du noeud s'il n'est pas une racine, lui-même sinon.

— Un entier valant le nombre de noeuds du sous-arbre dont il est la racine si c'est une racine et 0 sinon.

Cette forêt est telle que deux éléments sont dans la même composante si et seulement si leurs noeuds correspondant appartiennent au même arbre. Le représentant associé à un noeud est alors la racine de l'arbre dans lequel celui-ci se trouve.

```
1  Fonction initialiser () :
2      Pour chaque noeud :
3          noeud.père = noeud
4          noeud.taille = 1
5
6  Fonction représentant (noeud) :
7      Si noeud.père != noeud:
8          noeud.père = représentant(noeud.père)
9      Renvoyer noeud.père
10
11 Fonction unir (noeud1, noeud2) :
12     noeud1 = représentant(noeud1)
13     noeud2 = représentant(noeud2)
14
15     Si noeud1 != noeud2 :
16         Si noeud2.taille > noeud1.taille :
17             Échanger noeud1 et noeud2
18
19         noeud1.taille += noeud2.taille
20         noeud2.père = noeud1
```

2 Graphes

2.1 Plus courts chemins

2.1.1 Floyd-Warshall

L'algorithme de Floyd-Warshall permet de calculer la longueur du plus court chemin entre chaque paire de noeuds d'un graphe G orienté et pondéré avec une complexité en $\Theta(|V|^3)$. Les pondérations peuvent éventuellement être négatives, mais il ne doit pas exister de cycle négatif dans G.

Implémentation

```
1  On initialise dist comme étant la matrice d'adjacence du graphe
2  Pour chaque noeud p:
3    Pour chaque noeud d:
4      Pour chaque noeud f:
```

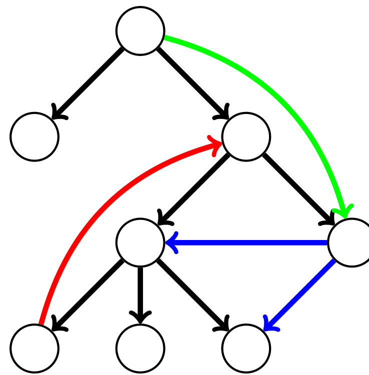
```
5        dist[d][f] = min(
6          dist[d][p] + dist[p][f],
7          dist[d][f]
8        )
```

Démonstration. La correction de l'algorithme est assurée par l'invariant de boucle suivant, vérifié au début de chaque itération de la boucle principale : pour toute paire de noeuds d'indices respectifs d et f, $dist[d][f]$ est la longueur du plus court chemin du noeud d'indice d vers le noeud d'indice f n'utilisant comme noeuds intermédiaires que des noeuds d'indice dans $[\![0, p-1]\!]$. □

2.2 Arbre du DFS

Il est possible de représenter un graphe orienté en mettant en avant les différentes propriétés de ses arcs pouvant être rencontrées lors d'un parcours en profondeur, ce qui peut par exemple aider à adapter l'algorithme du DFS à différents problèmes. Pour cela, on effectue un parcours en profondeur du graphe, puis on représente le graphe selon l'arbre des appels de la fonction récursive.

Figure 1 – Arbre du DFS



Les arcs peuvent être classés en quatre catégories :

- → les *arcs de découverte* pointent vers un noeud n'ayant pas encore été parcouru
- → les *arcs arrières* pointent vers un noeud en cours de parcours
- → les *arcs avants* pointent vers un noeud parcouru qui est un descendant du noeud actuel dans l'arbre d'appels
- → les *arcs transverses* pointent vers un noeud parcouru qui n'est pas un descendant du noeud actuel dans l'arbre d'appels

2.3 Composantes fortement connexes

2.3.1 Algorithme de Kosaraju

L'algorithme de Kosaraju permet de calculer les composantes fortement connexes d'un graphe orienté. Pour cela, il effectue successivement deux DFS. Le premier permet de trouver un parcours suffixe du graphe, c'est-à-dire un parcours suffixe de l'arbre des appels du DFS. Le deuxième est effectué sur le graphe transposé $G^\top$ (graphe obtenu en inversant les arcs de G), en découvrant les noeuds dans l'ordre inverse du parcours suffixe. Il permet de déterminer les composantes fortement connexes du graphe.

Implémentation

```
Trouver un parcours suffixe du graphe
Pour chaque noeud v dans l'ordre inverse du parcours suffixe :
    Si v n'a pas encore été visité :
        Trouver l'ensemble des noeuds non-visités ayant accès à v
        Cet ensemble est la composante fortement connexe contenant v
        Marquer tous ses noeuds comme visités
```

Démonstration. Montrons que l'ordre de parcours obtenu avec le premier DFS permet effectivement de trouver les composantes fortement connexes lors du deuxième DFS. Soit v un noeud sur lequel débute un parcours du deuxième DFS.

- Soit u un noeud de la composante fortement connexe à laquelle v appartient. Il existe un chemin de v vers u dans $G^\top$. De plus, v est le premier noeud de sa composante fortement connexe à être rencontré, donc aucun des noeuds permettant de se rendre de v vers u n'a été initialement visité. Ainsi, u sera détecté lors du parcours.
- Soit u un noeud parcouru lors de l'appel du deuxième DFS depuis le noeud v. Montrons que u appartient à la composante fortement connexe contenant v.
  - Il existe un chemin de v vers u dans $G^\top$, donc il existe un chemin de u vers v dans G.
  - On suppose qu'il n'existe pas de chemin de v vers u dans G. Le point précédent assure que v apparaît avant u dans le parcours suffixe de G, donc que u a déjà été parcouru lorsque le parcours de v commence. Cela est absurde car u devrait être parcouru à nouveau lors de l'exploration depuis v.

□

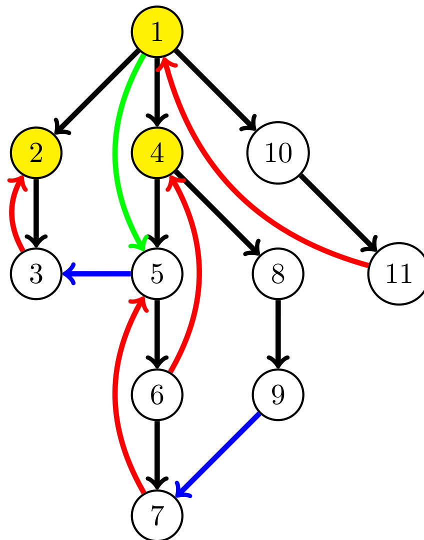
2.3.2 Algorithme de Tarjan

L'algorithme de Tarjan permet de calculer les composantes fortement connexes d'un graphe orienté $G = (V, E)$. Il effectue un parcours en profondeur du graphe, on utilisera donc la représentation graphique décrite précédemment qui est adaptée à cette situation.
On définit tout d'abord la racine d'une composante fortement connexe comme étant le premier noeud de la composante rencontré lors du parcours du graphe. Elles sont représentées en jaune sur le schéma. On remarque qu'une composante fortement connexe est un sous-arbre de l'arbre des appels du DFS enraciné sur la racine de la composante, privé des éventuels sous-arbres correspondant à d'autres composantes. Pour trouver les composantes fortement connexes, il suffit donc de trouver l'ensemble de leurs racines.
On effectue un DFS sur le graphe, en maintenant à jour :

- Des identifiants pour chacun des noeuds : ils sont donnés par ordre croissant au fur et à mesure du parcours
- Une pile contenant les noeuds dont la composante n'a pas encore été parcourue complètement, ajoutés dans leur ordre de parcours
- L'état de chaque noeud, parmi les états suivants :
  - À visiter : le noeud est encore inconnu
  - Dans la pile : le noeud a été découvert, mais sa composante fortement connexe n'est pas encore complètement connue
  - Visité : le parcours du noeud est terminé, et sa composante fortement connexe est complètement connue

La fonction `parcourir` renvoie le plus petit identifiant d'un noeud présent dans la pile et accessible directement depuis le sous-arbre enraciné sur le noeud d'appel, ou éventuellement $+\infty$ si aucun noeud ne convient.
Implémentation

Figure 2 – Exemple




```
Fonction parcourir (noeud):
    Donner un identifiant au noeud actuel
    Ajouter le noeud à la pile et mettre à jour son état

    idMinAccessible = idNoeud

    Pour chaque voisin:
        Si le voisin est dans la pile:
            idMinAccessible = min(idVoisin, idMinAccessible)
        Si le voisin n'a pas encore été visité:
            idMinAccessible = min(parcourir(voisin), idMinAccessible)

    Si idMinAccessible == idNoeud:
        La composante fortement connexe du noeud est composée des noeuds
          de la pile à partir du noeud actuel
        Enlever ces noeuds de la pile, et les marquer comme visités
        Renvoyer +oo

    Renvoyer idMinAccessible

Pour chaque noeud du graphe:
    Si le noeud est inconnu:
        Appeler la fonction parcourir sur ce noeud
```

Montrons que la fonction parcourir a le comportement attendu. On suppose que, lors d'un appel à la fonction parcourir, la boucle sur tous les voisins vient de se terminer.

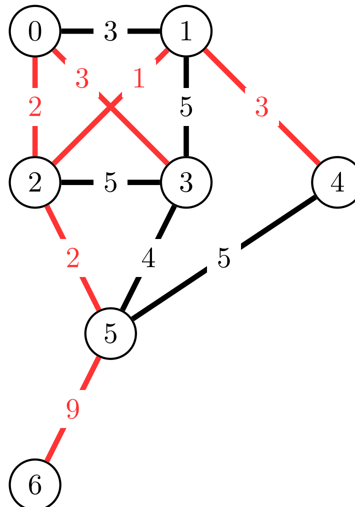
— Si idMinAccessible vaut l'identifiant du noeud, alors il est impossible de sortir du sous-arbre depuis celui-ci, sauf éventuellement vers une composante fortement connexe déjà complète. Le noeud est donc une racine, et la composante fortement connexe correspondante est composée de l'ensemble des noeuds au-dessus du noeud actuel dans la pile. On ajoute la composante à celles déjà détectées, et on continue l'exploration.

— On suppose à présent que idMinAccessible est strictement inférieur à l'identifiant du noeud. Le noeud correspondant appartient à une composante fortement connexe dont la racine est toujours en cours de visite (sinon, la composante aurait été trouvée). C'est donc un noeud ascendant du noeud actuel, et il est accessible depuis celui-ci. Ainsi, le noeud actuel n'est pas une racine. On renvoie la valeur demandée et on continue le parcours.

2.4 Arbre couvrant minimal

Soit $G = (V, E)$ un graphe pondéré non-orienté connexe. Un *arbre couvrant* de G est un arbre inclus dans G contenant tous les noeuds de G. On définit le *poids* d'un arbre couvrant comme étant la somme des pondérations de ses arêtes. Un *arbre couvrant minimal* est alors un arbre couvrant de poids minimal.

Figure 3 – Arbre couvrant minimal



2.4.1 Algorithme de Prim

L'algorithme de Prim est un algorithme calculant un arbre couvrant minimal d'un graphe. Son fonctionnement est semblable à celui de l'algorithme de Dijkstra. Il met à jour un arbre A initialement vide en parcourant le graphe de la façon suivante : tant que des noeuds n'ont pas été visités, c'est-à-dire tant que A n'est pas couvrant, une arête à ajouter à A est sélectionnée de manière gloutonne : c'est celle reliant un sommet visité à un sommet non-visité dont la pondération est minimale. Une fois que tous les sommets du graphe ont été visités, l'arbre obtenu est un arbre couvrant minimal.
Implémentation

```
1  A := arbre vide
2  Noter un sommet quelconque du graphe comme visité
3  Tant qu'il existe des sommets non-visités :
4      a := arête reliant un sommet visité et un sommet non-visité de
5           pondération minimale
6      Ajouter a à A
7      Marquer le sommet à l'extrémité de a comme visité
```

Pour rechercher l'arête à ajouter à A, il est possible d'utiliser une file à priorité contenant l'ensemble des arêtes pouvant être utilisées. La complexité obtenue est alors en $O(|E| \log |E|)$. Il est également possible de maintenir, pour chaque noeud non-visité, l'arête de pondération minimale le reliant à un noeud visité, si elle existe. Dans ce cas, la complexité est en $O(|V|^2)$.

Démonstration. L'algorithme renvoie bien un arbre couvrant du graphe G. Pour montrer qu'il est minimal, il suffit de montrer qu'à chaque étape de l'algorithme, il existe un arbre couvrant de G dans lequel A est inclus.

— Initialement, A est vide, donc la propriété est vérifiée.

— On suppose la propriété vérifiée avant qu'une arête a ne soit ajoutée à A. On note $\widetilde{A}$ un arbre couvrant minimal dans lequel A est inclus. Si a appartient à $\widetilde{A}$, la propriété reste

vérifiée. On suppose donc que a n'appartient pas à $\widetilde{A}$. a relie un sommet visité u à un sommet non-visité v. Par définition d'un arbre, il existe un chemin c de u vers v dans $\widetilde{A}$. Or, u est visité et v ne l'est pas, donc il existe une arête $\widetilde{a}$ dans c reliant un sommet visité à un sommet non-visité. Si sa pondération était strictement inférieure à celle de a, elle aurait été sélectionnée par l'algorithme, ce qui n'est pas le cas. Ainsi, l'arbre couvrant obtenu à partir de $\widetilde{A}$ en enlevant $\widetilde{a}$ puis en rajoutant a reste minimal. La propriété reste donc vérifiée.

Une fois tous les sommets visités, tout arbre couvrant contenant A est égal à A, ce qui assure que A est minimal. □

2.4.2 Algorithme de Kruskal

L'algorithme de Kruskal est un algorithme glouton permettant de déterminer un arbre couvrant minimal d'un graphe avec une complexité temporelle en $\Theta\left(|E|\log(|E|)\right)$.

Implémentation

```
1 Initialiser un union-find sur l'ensemble des noeuds du graphe
2 Créer un arbre vide
3 Pour chaque arête par pondération croissante :
4     Si l'arête permet de relier deux noeuds non-connectés :
5         Ajouter l'arête à l'arbre
6         Fusionner les noeuds dans l'union-find
7         Si l'arbre est couvrant :
8             Renvoyer cet arbre
```

Démonstration. L'algorithme renvoie bien un arbre couvrant de G. Pour montrer qu'il est de poids minimal, on considère l'invariant suivant, vérifié au début de chaque itération de la boucle principale : il existe $\widetilde{A}$ un arbre couvrant minimal de G contenant toutes les arêtes de l'arbre A construit par l'algorithme. Au début de l'exécution, A ne contient aucune arête, donc l'invariant est vérifié. Lorsqu'une arête a est ajoutée à A :

- Si a appartient déjà à $\widetilde{A}$, l'invariant reste vérifié.
- On suppose que l'arête a n'appartient pas à $\widetilde{A}$, et on note (u, v) les noeuds qu'elle relie. $\widetilde{A}$ est un arbre, donc il existe un chemin C de u vers v dans $\widetilde{A}$. L'arête a doit être ajoutée à A, donc il n'existe pas de chemin de u vers v dans A. Ainsi, il existe une arête a' de C n'appartenant pas à A, on note u' et v' les noeuds qu'elle relie. On suppose cette arête de pondération strictement inférieure à celle de a. L'ordre de parcours des arêtes assure qu'elle a été parcourue précédemment, sans avoir été ajoutée à A. Il donc un chemin de u' vers v' dans A. Or, les arêtes de A appartiennent à $\widetilde{A}$, donc il existe un chemin de u' vers v' dans $\widetilde{A}$ n'utilisant pas l'arête a', ce qui est absurde car a' appartient à l'arbre $\widetilde{A}$. Ainsi, a' est de pondération supérieure ou égale à celle de a. En remplaçant a' par a dans $\widetilde{A}$, on obtient un arbre couvrant minimal de G auquel toutes les arêtes de A appartiennent : l'invariant est donc vérifié.

□

2.5 Flots

2.5.1 Définitions

Soit $G = (V, E)$ un graphe orienté et (s, p) deux sommets distincts de V que l'on appellera respectivement *source* et *puits*. On associe à chaque arc $e \in E$ un réel positif représentant sa capacité. Pour tout $(u, v) \in V^2$, on notera $c(u, v)$ la somme des capacités des arcs de u vers v.

Définition. $F = (G, c, s, p)$ *constitue un réseau de flot.*

Définition. *Un flot f est une application de V^2 dans $\mathbb{R}$ telle que :*

- *Pour tout* $(u,v) \in V^2$, $f(u,v) \leq c(u,v)$
- *Pour tout* $(u,v) \in V^2$, $f(u,v) = -f(v,u)$
- *Pour tout* $u \in V \setminus \{s,p\}$, $\sum_{v \in V} f(u,v) = 0$
- $\sum_{v \in V} f(s,v) \geq 0$ *et* $\sum_{v \in V} f(p,v) \leq 0$

Définition. *La valeur d'un flot f est* $|f| = \sum_{v \in V} f(s,v)$.

2.5.2 Capacités résiduelles

Pour tout $(u,v) \in V^2$, on notera $c_f(u,v) = c(u,v) - f(u,v)$ la capacité résiduelle de u vers v. Cela correspond à la capacité disponible de u vers v après l'application du flot f. Le réseau résiduel correspondant à (F, f) est obtenu en remplaçant dans F les capacités des arcs par leurs capacités résiduelles.

Définition. *Un chemin augmentant est un chemin de s vers p sur lequel toutes les capacités résiduelles sont strictement positives.*

2.5.3 Algorithme d'Edmonds-Karp

L'algorithme d'Edmonds-Karp permet de calculer le flot maximal pouvant traverser un réseau avec une complexité temporelle en $O(|V||E|^2)$. Sa complexité est également en $O(S)$ où S est la somme des capacités des arcs du graphe. Pour cela, il commence par considérer un flot nul qu'il améliore ensuite tant que possible ; le flot finalement obtenu étant de valeur maximale.

Implémentation

```
1 Initialiser f comme étant un flot nul
2 Créer le réseau résiduel correspondant à f
3 Tant qu'il existe un chemin augmentant dans le réseau résiduel :
4     Utiliser ce chemin pour améliorer le flot
5     Mettre à jour le graphe résiduel
```

La recherche d'un chemin augmentant est effectuée à l'aide d'un BFS dans le graphe résiduel.

Démonstration.

- Terminaison : $|f|$ croît strictement à chaque itération de la boucle principale, et la valeur d'un flot est majorée par la somme de la capacité des arcs. Ainsi, **lorsque les capacités sont des entiers**, l'algorithme termine. Cet argument prouve également la complexité en $O(S)$.
- Correction : Lorsque l'algorithme termine, il n'existe plus de chemin augmentant, le flot obtenu est donc de valeur maximale.

□

2.5.4 Théorème Flot-Max / Coupe-Min

Théorème. *Soit $G = (V, E)$ un graphe orienté pondéré, et s et p deux sommets de G. La valeur maximale d'un flot de s vers p est égale à la valeur minimale d'une coupe entre s et p.*

Démonstration.

- Le flot max doit être réparti parmi les arcs de toute coupe, donc $|FlotMax| \leq |CoupeMin|$

FIGURE 4 – Graphe G, flot max

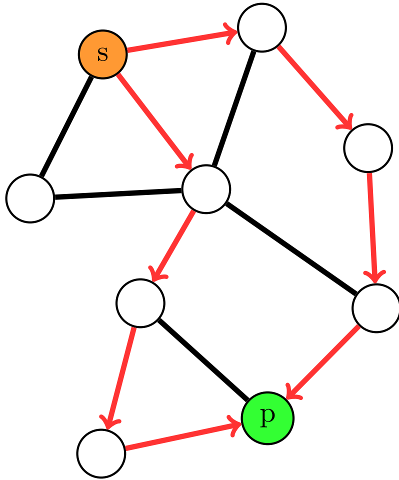
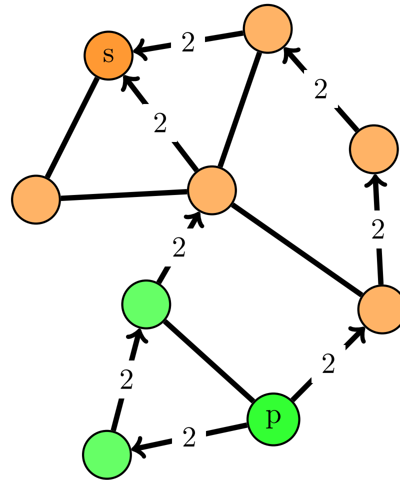


FIGURE 5 – Graphe résiduel G'



- On note G' le graphe résiduel correspondant à un flot max de G, A l'ensemble des noeuds accessibles depuis S dans G' et B l'ensemble des noeuds n'appartenant pas à A. Il n'existe pas de chemin améliorant, donc $p \in B$. On considère $\mathcal{C}$ l'ensemble des arcs de G partant d'un noeud de A vers un noeud de B.
  - $s \in A$, $p \in B$ et il n'existe pas d'arc hors de $\mathcal{C}$ d'un noeud de A vers un noeud de B, donc $\mathcal{C}$ est une coupe.
  - Dans le graphe résiduel G', il n'existe pas d'arc de A vers B, donc les arcs de B vers A ne sont pas utilisés et ceux de A vers B sont saturés. Ainsi, la somme des capacités des arcs de A vers B est égale au flot max. Or, cette somme est égale à la somme des pondérations des arcs de $\mathcal{C}$, donc $|CoupeMin| \leq |FlotMax|$

Finalement, $|CoupeMin| = |FlotMax|$ □

2.6 Couplages

Soit $G = (V, E)$ un graphe biparti dont on note (X, Y) une partition des sommets adaptée, c'est-à-dire telle que toute arête relie un noeud de X avec un noeud de Y. Un *couplage* C de G est une partie de E telle que tout noeud ne soit relié qu'à au plus une arête de C. Un couplage est dit *maximum* si et seulement si il est de cardinal maximal. Pour la suite, les arêtes d'un graphe appartenant à un couplage seront représentées en rouge et celles n'y appartenant pas seront représentées en noir.

2.6.1 Chemins d'augmentation

Définition. *Un chemin d'augmentation d'un graphe biparti* $G = (V, E)$ *muni d'un couplage* C *est une suite finie* $v = (v_0, ..., v_{p-1})$ *de sommets de* V *telle que* $v_0 \in X$, $v_{p-1} \in Y$, *les sommets* v_0 *et* v_{p-1} *ne soient reliés à aucune arête de* C *et :*

$$\forall i \in [\![0, p-2]\!], \{v_i, v_{i+1}\} \in \begin{cases} E \setminus C & \text{si } i \text{ est pair} \\ C & \text{si } i \text{ est impair} \end{cases}$$

Théorème. *Un couplage* C *d'un graphe biparti* $G = (V, E)$ *est maximum si et seulement si il n'existe pas de chemin d'augmentation pour ce couplage.*

Démonstration.

— On suppose qu'il existe un chemin d'augmentation $v = (v_0, ..., v_{p-1})$. La différence symétrique de C et de $\{\{v_i, v_{i+1}\}, i \in [\![0, p-2]\!]\}$ est alors un couplage de cardinal $|C|+1$. Ainsi, s'il existe un chemin d'augmentation, le couplage n'est pas maximum.

— On suppose réciproquement qu'il n'existe pas de chemin d'augmentation. Il existe C_0 un couplage maximum de G. On note D la différence symétrique de C et de C_0. C et C_0 sont des couplages, donc chaque sommet de G est relié à au plus deux arêtes de D. Ainsi, les arêtes de D forment des *cycles* ou des *chaînes* d'arêtes, par exemple :

Figure 6 – Cycle

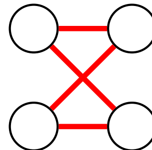
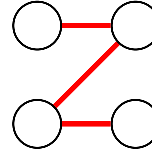


Figure 7 – Chaîne



On suppose qu'il existe une chaîne composée d'un nombre impair d'arêtes. Si les arêtes à ses extrémités sont dans C, elle permet d'obtenir un chemin d'augmentation pour C_0, et si les arêtes à ses extrémités sont dans C_0, elle permet d'obtenir un chemin d'augmentation pour C. Or, cela est absurde car il n'existe de chemin d'augmentation ni pour C (par hypothèse), ni pour C_0 (il est maximum). Toutes les chaînes sont ainsi composées d'un nombre pair d'arêtes. Finalement, $|C| = |C_0|$, donc C est maximum.

□

2.6.2 Recherche d'un couplage maximum

Soit $G = (V, E)$ un graphe biparti. Pour construire un couplage maximum de G, on peut améliorer tant que possible un couplage initialement vide en trouvant et utilisant des chemins d'augmentation. L'algorithme décrit a une complexité en $O(|V||E| + |V|^2)$.

```
1 Créer un couplage vide C
2 Tant qu'il existe un chemin d'augmentation :
3     L'utiliser pour augmenter de un la taille de C
```

Démonstration.

— L'entier $|C|$ est incrémenté à chaque itération de la boucle principale. Or, $|C| \leq \frac{|V|}{2}$, donc l'algorithme termine, et la boucle principale est exécutée au plus $|V|$ fois.

— À la fin de l'exécution de l'algorithme, il n'existe plus de chemin d'augmentation pour (G, C) : le théorème précédent assure donc que C est maximum. Ainsi, l'algorithme renvoie un couplage maximum.

— Chaque itération de la boucle principale nécessite un parcours du graphe pour rechercher un chemin d'augmentation et un parcours des arêtes constituant le chemin d'augmentation. Ces opérations s'effectuent en $O(|E|)$. Finalement, la complexité temporelle de l'algorithme est en $O(|V||E|)$.

□

3 Chaînes de caractères

3.1 Algorithme de Knuth-Morris-Pratt

Soit $S = s_0...s_{n-1}$ une chaîne de caractères. L'algorithme de Knuth-Morris-Pratt (KMP) permet de calculer, pour tout k, la longueur l_k du plus long suffixe strict de $s_0...s_{k-1}$ étant

également un préfixe de S. Il peut être utilisé pour rechercher en temps linéaire une chaîne de caractères dans un texte.
Implémentation

```
1 Fonction kmp(s):
2   l := tableau de taille |s|+1
3   l[0] := -1
4   Pour k de 1 à |s| inclus :
5     t := l[k - 1]
6     Tant que t != -1 et s[t] != s[k - 1]:
7       t := l[t]
8     l[k] := t + 1
9   Renvoyer l
```

Démonstration.

— Une chaîne vide n'admet aucun suffixe strict : on choisit $l_0 = -1$ par convention. Le choix de cette valeur permet d'éviter de distinguer certains cas particuliers dans la suite de l'algorithme.

— Soit $k \in [\![1, n]\!]$. $t+1$ est la taille d'un suffixe strict de $s_0...s_{k-1}$ préfixe de S si et seulement si $s_{k-t-1}...s_{k-2}$ est préfixe de S et $s_{k-1} = s_t$. Ainsi, pour calculer le plus long suffixe strict de $s_0...s_{k-1}$ étant un préfixe de S, il suffit d'énumérer les suffixes de $s_0...s_{k-2}$ préfixes de S par taille décroissante, et de renvoyer le premier dont la longueur t vérifie $s_{k-1} = s_t$. Montrons que les sous-chaînes énumérées par l'algorithme sont bien les suffixes de $s_0...s_{k-2}$ préfixes de S. On note B le plus long suffixe strict d'une chaîne A dont il est également un préfixe. Alors, A est de la forme $B...B$, où les deux B peuvent éventuellement se chevaucher. Les préfixes-suffixes stricts de A sont B et les préfixes-suffixes stricts de B.

□

Pour utiliser cet algorithme pour rechercher les occurrences d'une chaîne de caractères S dans un texte T, il suffit de considérer la chaîne de caractères $S\#T$, où $\#$ est un caractère n'apparaissant ni sans S, ni dans T. Les occurrences de S dans T se terminent aux positions $k - 2|S|$ où k vérifie $l_k = |S|$.

Par exemple, pour recherche la chaîne $S = ababc$ dans le texte $T = abababcd$, l'algorithme construit le tableau suivant :

Chaîne		a	b	a	b	c	#	a	b	a	b	a	b	c	d
l_k	-1	0	0	1	2	0	0	1	2	3	4	3	4	5	0

4 Géométrie

4.1 Enveloppe convexe

On considère un ensemble A de n points du plan de coordonnées (x_i, y_i) dont on cherche à calculer l'enveloppe convexe. Pour cela, il suffit de calculer d'abord la partie supérieure de l'enveloppe, puis sa partie inférieure. Le calcul de la partie inférieure pourra s'effectuer de manière similaire à celui de la partie supérieure, on ne décrira donc que le calcul de la partie supérieure. On effectue un balayage des points de A par abscisses décroissantes, en maintenant sur une pile l'ensemble des points de A parcourus et appartenant à la frontière de l'enveloppe convexe actuelle. À chaque tour de boucle, on enlève de la pile les points n'appartenant plus à l'enveloppe convexe, puis on ajoute le point actuel.
Implémentation

```
1 Créer une pile de points vide
2 Pour chaque point de A par abscisses décroissantes :
```

```
3     Tant que le point en haut de la pile peut être supprimé :
4         Le supprimer
5     Ajouter le point actuel à la pile
```

L'algorithme décrit a une complexité temporelle en $O(n \log(n))$ car il nécessite un tri des points.