Chapitre 6

Flots

Sommaire

Les réseaux de flots modélisent des réseaux de transport, dans lesquels une source produit un certain matériau que l'on veut acheminer le long du réseau jusqu'à un certain puits. Le flot se réfère à la quantité de produit que l'on achemine de la source au puits, et l'on veut pouvoir maximiser la quantité de produit acheminée — on recherche donc un *flot maximum* — sachant que les canaux du réseau possèdent chacun une certaine capacité qu'on ne peut dépasser. On verra dans ce chapitre deux méthodes permettant de résoudre ce problème, celle de Ford et Fulkerson ainsi que celle de Dinitz, de même que diverses applications. On fera les deux hypothèses suivantes dans tout ce chapitre :

1. **toutes les *capacités* et tous les *flots* manipulés prennent exclusivement des valeurs entières** [1], et
2. **les arcs du réseau n'existent que dans un seul sens** [2].

6.1 Notions de base

Les graphes peuvent servir à représenter des réseaux de transport, par lesquels on décide d'acheminer un certain produit. Les arcs nous disent quelle quantité de produit peut circuler entre deux sommets, et le but est d'acheminer une quantité maximale d'une *source* du réseau (un sommet de degré entrant nul) vers un *puits* du réseau (un sommet de degré sortant nul).

1. Pour des raisons théoriques qu'on ne développera pas ici ; voir Cormen et al. [1] pour plus de détails.
2. Cette hypothèse n'est pas essentielle au bon fonctionnement des algorithmes que l'on verra mais facilite la discussion.

Définition 25. Un *réseau de flot* est un graphe orienté et pondéré $G = (V, A, c)$ dans lequel chaque arc (u, v) possède une *capacité* $c(u, v) \geq 0$. Ce réseau possède une *source* s (un sommet de degré entrant nul) et un *puits* t (un sommet de degré sortant nul), et l'on suppose que tous les autres sommets appartiennent à un chemin de s à t.

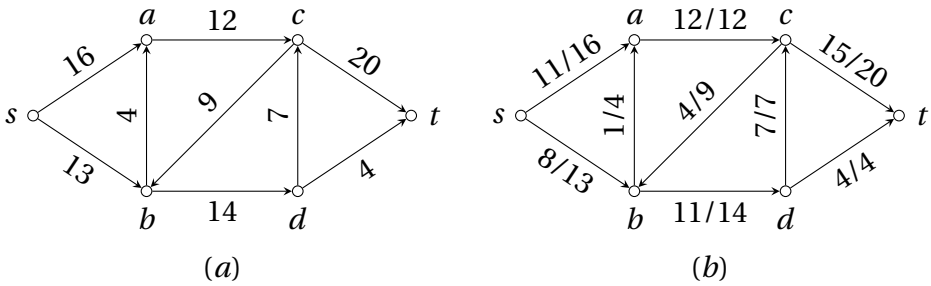
Comme précisé dans l'introduction, on exige généralement que les arcs n'existent que dans une seule direction : si $(u, v) \in A$, alors $(v, u) \notin A$.

Définition 26. Un *flot* dans un réseau de flot $G = (V, A, c)$ est une fonction $f : V \times V \to \mathbb{R}$ satisfaisant les deux propriétés suivantes :

1. **contrainte de capacité** : $\forall\ u, v \in V : 0 \leq f(u, v) \leq c(u, v)$;
2. **conservation des flots** : $\forall\ u \in V \setminus \{s, t\} : \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$.

Les deux propriétés sont intuitives : la première formalise le fait que l'on ne peut pas dépasser la capacité d'un arc, et la seconde exprime le fait que la quantité de flot entrant dans un sommet égale ce qui en sort. La valeur de la fonction est nulle pour les paires de sommets non adjacents.

Exemple 42. (a) Un réseau de flot [1] ; (b) le même réseau muni d'un flot. La notation x/y exprime que x unités de flot circulent le long de l'arc de capacité y.



(a) (b)

Définition 27. La *valeur* d'un flot f est

$$|f| = \sum_{v \in V} f(s, v) - f(v, s).$$

Il s'agit donc de la quantité qui sort de la source moins ce qui y arrive[3]. Le *problème du flot maximum* est de trouver, pour un réseau de flot G donné, un flot f de valeur maximum, c'est-à-dire que pour tout flot f' sur G, on a $|f| \geq |f'|$.

Définition 28. Soit G un réseau de flot muni d'un flot f. On dit que l'arc $(u, v) \in A(G)$ est *saturé (par le flot f)* si $f(u, v) = c(u, v)$.

3. La source étant de degré entrant nul, cette quantité sera en général nulle, mais cette définition plus générale sera utile dans le cas des réseaux résiduels qu'on verra plus loin.

6.2 La méthode de Ford-Fulkerson

La méthode de Ford-Fulkerson consiste à démarrer avec un flot initial nul, puis à augmenter ce flot à chaque itération en trouvant un "chemin augmentant" dans un "réseau résiduel" (ces termes seront définis plus loin). Les arcs du chemin augmentant dans le réseau résiduel nous disent quels arcs changer dans le réseau original pour augmenter la valeur du flot. Cela ne signifie pas que le flot augmente sur *tous* les arcs du réseau : il est possible qu'on doive faire chuter le flot de certains arcs pour pouvoir augmenter la valeur totale du flot, comme on le verra plus loin.

La méthode se termine quand il n'existe plus de chemin augmentant. Intuitivement, l'existence de ce chemin dans le réseau résiduel nous informe qu'il est encore possible d'augmenter la quantité de flot circulant sur certains arcs, et l'arc de capacité minimale de ce chemin nous dira de combien exactement on peut augmenter le flot — on ne pourra pas faire mieux que ce minimum si l'on veut augmenter le flot le long de tout le chemin puisqu'on doit respecter la propriété 1 des flots. L'algorithme 28 illustre cette méthode, dont on détaillera les composantes plus loin. Remarquons que l'on n'a précisé nulle part *comment* trouver les chemins augmentants : les algorithmes de parcours qu'on a déjà vus nous offrent plusieurs options pour ce faire, et on verra que la manière dont on choisit les chemins aura un impact énorme sur la complexité de l'algorithme.

Algorithme 28 : FORDFULKERSON(G)

Entrées : un réseau de flot G.
Sortie : un flot maximum pour G.

```
1  flot ← tableau associatif (clés = G.arcs(), valeurs = 0);
2  source ← unique sommet de degré entrant nul de G;
3  puits ← unique sommet de degré sortant nul de G;
4  G_f ← G;                // au départ, le réseau et son résiduel coïncident
5  chemin ← CHEMINAUGMENTANT(G_f, source, puits);
6  tant que chemin ≠ NIL faire
7      AUGMENTERFLOT(flot, chemin);
8      METTREAJOURRÉSIDUEL(G, G_f, chemin.arcs(), flot);
9      chemin ← CHEMINAUGMENTANT(G_f, source, puits);
10 renvoyer flot;
```

Quelle que soit la manière dont on sélectionne le chemin augmentant, le flot augmente bien strictement à chacune des itérations de notre algorithme (voir Cormen et al. [1, corollaire 26.3 page 720]). Ce n'est pas nécessairement évident, car augmenter le flot sur certains "arcs retours", qui peuvent appartenir à un chemin augmentant, pourrait faire diminuer le flot dans le graphe original.

6.2.1 Réseaux résiduels et chemins augmentants

La notion de réseau résiduel se base sur la notion de *capacité résiduelle*, qui représente la quantité dont le flot peut encore augmenter (ou diminuer) sur chacun des arcs d'un réseau donné. L'exemple 43 illustrera les notions présentées dans les définitions qui suivent.

Définition 29. Soit $G = (V, A, c)$ un réseau muni d'un flot f. La *capacité résiduelle* d'une paire de sommets $u, v \in V$ est

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{si } (u,v) \in A, \\ f(v,u) & \text{si } (v,u) \in A, \\ 0 & \text{sinon .} \end{cases}$$

Définition 30. Soit G un réseau de flot muni d'un flot f. Le *réseau résiduel* G_f est le sous-graphe de G induit par les arcs de capacité résiduelle non nulle. Autrement dit, il s'agit du graphe $G_f = (V, A_f, c_f)$ où $A_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$.

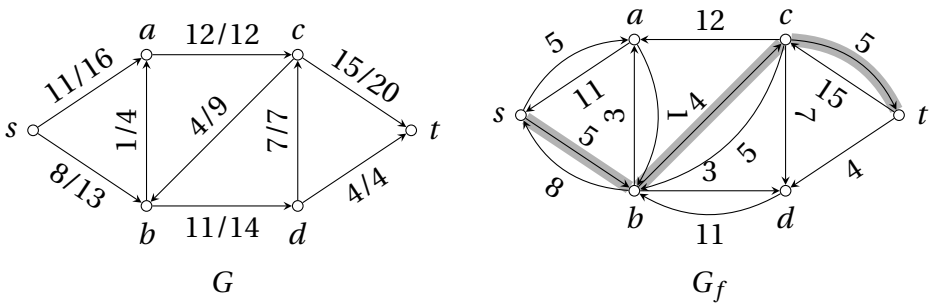
On constate dans G_f l'apparition d'arcs qui n'appartiennent pas au réseau de départ. Ces arcs nous permettent de représenter le fait que l'on peut réduire le flot le long de l'arc (u,v) en renvoyant des unités de flot sur un arc retour artificiel (v,u).

Définition 31. Un *chemin augmentant* est un chemin simple (sans répétition de sommets) de s à t dans le réseau résiduel G_f. Notons $c_f(P) = \min_{(u,v)\in A(P)} c_f(u,v)$; le *flot associé au chemin augmentant* P est donné par

$$f_P(u,v) = \begin{cases} c_f(P) & \text{si } (u,v) \in P, \\ 0 & \text{sinon .} \end{cases}$$

Sa valeur est donc tout simplement $|f_P| = c_f(P)$.

Exemple 43. Revoici le réseau G muni d'un flot de l'exemple 42, et le réseau résiduel G_f correspondant, avec un chemin augmentant surligné.



Une fois qu'on a identifié un chemin augmentant, on peut augmenter le flot de départ de la capacité résiduelle minimale présente sur ce chemin.

Définition 32. L'*augmentation* du flot f par f', notée $f \uparrow f'$, est une fonction de $V \times V$ vers $\mathbb{R}$ définie par

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{si } (u,v) \in A, \\ 0 & \text{sinon .} \end{cases}$$

Cette notation exprime l'augmentation du flot f par une certaine quantité (d'où le $+f'(u,v)$)

et la réduction qui découle de ce qu'on peut renvoyer en arrière (d'où le $-f'(v,u)$).

Dans l'exemple 43, la capacité résiduelle la plus petite du chemin augmentant identifié est 4, et on va donc pouvoir augmenter de 4 le flot sur tous les arcs de ce chemin dans le réseau de départ. Attention, comme certains arcs représentent des retraits de flots, augmenter le flot de 4 sur ces arcs signifie qu'on va en réalité diminuer le flot de 4 sur l'arc existant réellement dans le réseau de départ (voir par exemple l'arc (b,c) du réseau résiduel, qui nous mènera à *réduire* de 4 le flot sur l'arc (c,b) du réseau de départ). L'algorithme 29 implémente l'approche que l'on vient d'expliquer.

Algorithme 29 : AUGMENTERFLOT(f, P)

Entrées : un flot f, un chemin P.
Résultat : le flot f augmente au maximum le long du chemin P.

```
1 capacité_min ← min {c | (u, v, c) ∈ P.arcs()};
2 pour chaque (u, v, c) ∈ P.arcs() faire
3 |   si (u, v) ∈ f alors f[(u, v)] ← f[(u, v)]+ capacité_min ;
4 |   sinon f[(v, u)] ← f[(v, u)] − capacité_min ;
```

Il n'est pas nécessaire d'écrire un algorithme calculant le réseau résiduel, car il serait inefficace de le recalculer en entier à chaque itération du calcul de flot. Au lieu de cela, on peut partir d'un résiduel qui est exactement le réseau de départ, et mettre à jour ce réseau en n'utilisant que les arcs du chemin augmentant que l'on a trouvé. Pour cela, on peut :

1. calculer les capacités résiduelles de chaque arc du réseau de départ en utilisant le flot actuel ;
2. calculer les capacités résiduelles de chaque arc *inverse* du réseau de départ en utilisant le flot actuel ; et
3. remplacer les arcs du chemin trouvé et leurs inverses dans G_f par des arcs pondérés par les capacités résiduelles que l'on vient de calculer, en supprimant les arcs de poids nul ainsi créés.

L'algorithme 30 montre le résultat de cette approche.

6.2.2 L'approche d'Edmonds-Karp

Comme on l'a précisé, la méthode de Ford-Fulkerson ne précise pas comment on doit sélectionner le chemin augmentant. L'approche d'Edmonds-Karp, qu'on va suivre ici, consiste simplement à sélectionner le plus court chemin de s à t, ce qui revient à effectuer un parcours en largeur au départ de s.

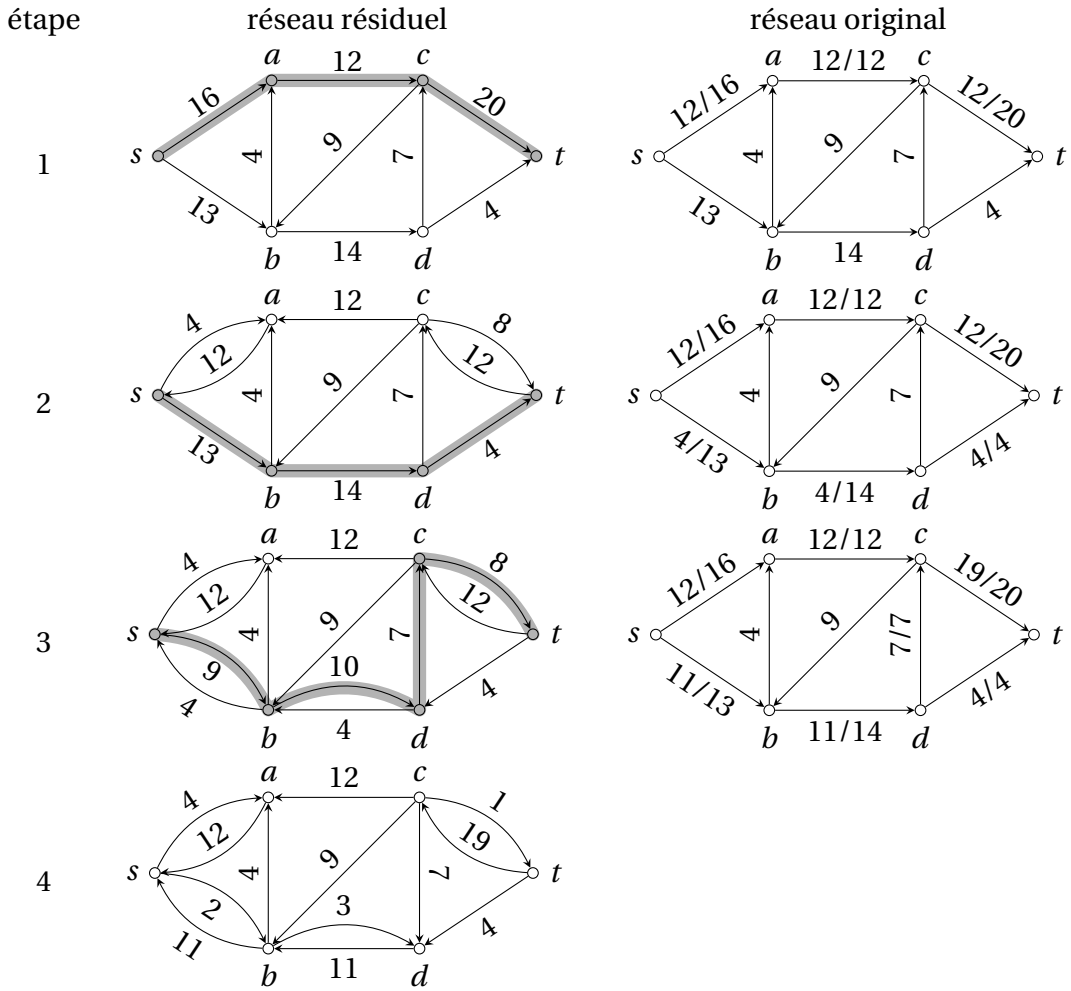
Exemple 44. Voici les étapes de l'algorithme d'Edmonds-Karp sur le graphe de l'exemple 43 :

Algorithme 30 : METTREAJOURRÉSIDUEL(G, G_f, arcs, f)

Entrées : un réseau G, le réseau résiduel correspondant G_f, un ensemble d'arcs dans G_f, et un flot f.

Résultat : modifie les arcs de G_f suivant l'augmentation de flot qu'ils ont subi.

```
// calculer les capacités résiduelles
c_f ← tableau associatif;
pour chaque (u, v) ∈ arcs faire
    si G.contient_arc(v, u) alors échanger u et v;
    c_f[(u, v)] ← G.poids_arc(u, v) − f[(u, v)] ;
    c_f[(v, u)] ← f[(u, v)] ;
// remplacer les arcs concernés de G_f en supprimant ceux dont le
   poids devient nul
pour chaque (u, v) ∈ arcs faire
    si c_f[(u, v)] > 0 alors G_f.ajouter_arc(u, v, c_f[(u, v)]) ;
    sinon G_f.supprimer_arc(u, v) ;
```



(fin exemple 44)

En principe, n'importe quel chemin de s à t dans le réseau résiduel fait l'affaire, et n'importe quel algorithme de parcours à partir de s peut convenir. En pratique, on se rend compte qu'il n'y a aucune raison de préférer le parcours en profondeur qui peut produire des chemins

certes valides mais plus longs, car on augmente surtout le risque de se retrouver avec plus d'arcs et éventuellement des arcs de capacité moindre, ce qui limite d'autant l'augmentation du flot.

On opte donc pour un parcours en largeur de notre graphe au départ de s, dans lequel on stocke les parents des sommets rencontrés pour pouvoir ensuite reconstruire le chemin ainsi obtenu de s à t s'il existe. L'algorithme 32 implémente cette approche, et utilise l'algorithme 31 pour reconstruire le chemin explicitement.

Algorithme 31 : RECONSTRUIRECHEMIN(G, début, fin, parents)

Entrées : un graphe orienté pondéré G, deux sommets début et fin, et les parents des sommets du graphe.
Sortie : un chemin de début à fin dans G, ou NIL s'il n'en existe pas.

```
1  chemin ← GrapheOrientéPondéré();
2  v ← fin;
3  tant que v ≠ début faire
4      si parents[v] = NIL alors renvoyer NIL;
5      chemin.ajouter_arc(parents[v], v, G.poids_arc(parents[v], v));
6      v ← parents[v];
7  renvoyer chemin;
```

Algorithme 32 : CHEMINAUGMENTANT(G, source, puits)

Entrées : un graphe orienté pondéré G, et deux sommets source et puits.
Sortie : un chemin de source à puits dans G, ou NIL s'il n'en existe pas.

```
1  déjà_visités ← tableau(G.nombre_sommets(), FAUX);
2  a_traiter ← file();
3  a_traiter.enfiler(source);
4  parents ← tableau(G.nombre_sommets(), NIL);
5  tant que a_traiter.pas_vide() faire
6      sommet ← a_traiter.défiler();
7      si sommet = puits alors arrêter;
8      si ¬ déjà_visités[sommet] alors
9          déjà_visités[sommet] ← VRAI;
10         pour chaque suivant ∈ G.successeurs(sommet) faire
11             a_traiter.enfiler(suivant);
12             si parents[suivant] = NIL alors parents[suivant] ← sommet ;
13 renvoyer RECONSTRUIRECHEMIN(G, source, puits, parents);
```

6.2.3 Correction et liens avec les coupes

La correction de la méthode de Ford et Fulkerson découle d'un théorème connu sous le nom de *max-flow min-cut theorem*, qui comme son nom l'indique met en relation les flots maximums avec des "coupes" de poids minimum. Avant d'exposer ce théorème, nous avons besoin de quelques définitions.

Définition 33. Une *coupe* (S, T) dans un réseau de flot avec source s et puits t est une partition de ses sommets en deux ensembles S et T tels que $s \in S$ et $t \in T$. Le *flot associé à la coupe* (S, T) est le flot circulant dans les arcs joignant S à T :

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u).$$

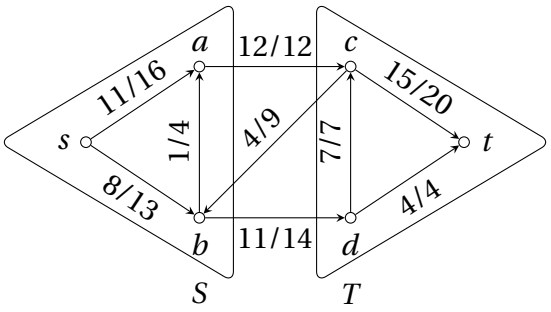
La *capacité* de la coupe (S, T) est la somme des capacités des arcs joignant S et T :

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v).$$

Enfin, la coupe (S, T) est *minimum* si pour toute coupe (S', T') du réseau, on a $c(S,T) \leq c(S',T')$.

Attention, la capacité d'une coupe (S, T) n'implique que les arcs allant de S à T, alors que le flot associé prend également en compte les arcs allant de T à S.

Exemple 45. [1] Voici un réseau de flot avec une coupe de capacité $12 + 14 = 26$ (la somme des capacités des arcs allant de S à T). La valeur du flot associé est $12+11-4 = 19$ (la somme des flots sur les arcs allant de S à T moins la somme des flots sur les arcs allant de T à S).



Le lien entre les flots et les coupes est formalisé par les résultats suivants. Intuitivement, tout comme la capacité d'un arc limite le flot pouvant circuler d'un sommet u à un sommet v, la capacité d'une coupe limite la quantité de flot pouvant aller d'un morceau de la coupe vers l'autre.

Lemme 6.2.1. Soit f un flot sur un réseau de flot G, et (S, T) une coupe sur G. Alors $|f| = f(S,T)$.

Démonstration. Voir Cormen et al. [1, pages 721–722]. □

Corollaire 6.2.1. Pour tout flot f sur un réseau de flot G et pour toute coupe (S, T) sur ce même réseau, on a $|f| \leq c(S,T)$.

Démonstration. Découle des définitions et du Lemme 6.2.1 :

$$\begin{aligned}
|f| &= f(S,T) && \text{(Lemme 6.2.1)}\\
&= \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) && \text{(par définition)}\\
&\leq \sum_{u\in S}\sum_{v\in T} f(u,v)\\
&\leq \sum_{u\in S}\sum_{v\in T} c(u,v) && \text{(propriété 1 des réseaux de flots)}\\
&= c(S,T) && \text{(par définition).}
\end{aligned}$$

□

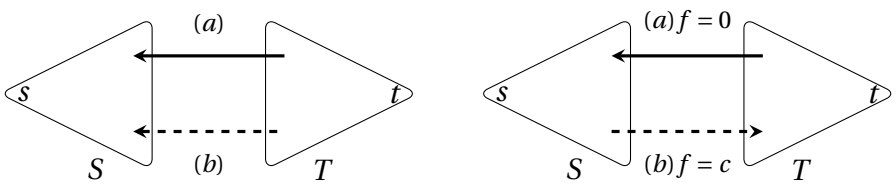
Avant d'énoncer le théorème *max-flow min-cut*, intéressons-nous au problème du calcul d'une coupe minimum qui nous sera utile dans la preuve de celui-ci. Si l'on a déjà calculé un flot maximum pour le réseau donné, on peut calculer une coupe minimum en parcourant le réseau résiduel correspondant à partir du sommet source, et en stockant dans un ensemble S les sommets ainsi atteints : la coupe $(S, V(G) \setminus S)$ est une coupe minimum, comme le prouve le résultat suivant.

Proposition 6.2.1. Soit G un réseau de flot de source s, f un flot pour ce réseau, G_f le réseau résiduel associé, et S l'ensemble des sommets de G_f accessibles à partir de s. Si G_f ne contient pas de chemin augmentant, alors $(S, V(G) \setminus S)$ est une coupe minimum pour G.

Démonstration. Soit $T = V(G_f) \setminus S$. Pour montrer que (S,T) est bien une coupe, il suffit de remarquer que l'absence de chemin de s à t garantit que $S \subset V(G)$ et donc que T contiendra au moins t. Pour prouver que cette coupe est minimum, on remarque également que :

1. tous les arcs entre S et T dans G_f sont orientés de T vers S ; en effet, un arc orienté (u,v) de S vers T nous permettrait d'ajouter v à S, ce qui n'est pas possible par définition de S ;
2. un arc (v,u) de T à S appartient à l'une des deux catégories suivantes :
   (a) soit cet arc appartient à $A(G)$, auquel cas $f(v,u) = 0$ sinon on aurait $(u,v) \in A(G_f)$, impossible comme expliqué ci-dessus ;
   (b) soit c'est l'arc (u,v) qui appartient à $A(G)$, auquel cas $f(u,v) = c(u,v)$ sinon on aurait $(u,v) \in A(G_f)$, impossible comme expliqué ci-dessus ;

On a donc la situation suivante :



G_f G

Par le Lemme 6.2.1, on sait que pour toute coupe (X, Y), la valeur d'un flot sur G est $|f| = f(X,Y) = \sum_{u\in X}\sum_{v\in Y} f(u,v) - \sum_{u\in X}\sum_{v\in Y} f(v,u)$. Donc la valeur de notre flot est

$$\begin{aligned} |f| = f(S,T) &= \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) && \text{(Lemme 6.2.1)} \\ &= \sum_{u\in S}\sum_{v\in T} f(u,v) && \text{(arcs de type } (a)) \\ &= \sum_{u\in S}\sum_{v\in T} c(u,v) && \text{(arcs de type } (b)). \end{aligned}$$

en vertu des points (a) et (b) ci-dessus. Cette dernière quantité est exactement la capacité de notre coupe, qui est donc bien minimum à cause de la minoration du Corollaire 6.2.1. □

Nous pouvons maintenant énoncer et prouver le théorème *max-flow min-cut*.

Théorème 6.2.1. [1] Soit f un flot dans un réseau $G = (V, A, c)$ de source s et de puits t. Les conditions suivantes sont équivalentes :

f est un flot maximum ;

2. le réseau résiduel G_f ne contient pas de chemin augmentant ;
3. il existe une coupe (S, T) pour G telle que $|f| = c(S, T)$.

Démonstration. On peut prouver les implications séparément :

1. (1) ⇒ (2) : par contraposée : si G_f contient un chemin augmentant, alors ce chemin contient par définition un arc qui nous permet d'augmenter strictement le flot le long de tout le chemin, et f n'est donc pas maximum puisqu'on peut l'augmenter.
2. (2) ⇒ (3) : si G_f ne contient pas de chemin augmentant, on peut obtenir une coupe minimum par l'algorithme énoncé avant la Proposition 6.2.1.
3. (3) ⇒ (1) : le Corollaire 6.2.1 nous dit que pour tout flot, on a $|f| \leq c(S, T)$; donc s'il existe une coupe pour laquelle il y a égalité, alors le flot est maximum puisqu'il n'est pas possible de dépasser cette valeur.

□

6.2.4 Complexité

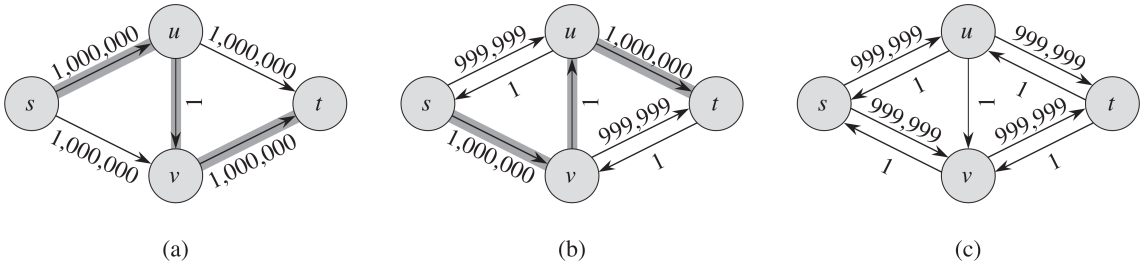
Le réseau $G = (V, A, c)$ de flot donné étant supposé connexe, on a $|A| \geq |V| - 1$ et donc $|V| = O(|A|)$.

1. construire le réseau résiduel nécessite de recopier chaque sommet et chaque arc au plus deux fois, ce qui donne du $O(|V| + |A|) = O(|A|)$;
2. calculer un chemin augmentant nécessite d'effectuer un parcours puis de revenir en arrière pour reconstruire le chemin, ce qui donne du $O(|V| + |A|) = O(|A|)$.

Il nous reste donc à savoir combien d'itérations l'algorithme va effectuer. Comme un chemin augmentant augmente d'au moins 1 unité le flot actuel à chaque itération, on peut déjà observer qu'on passe au plus $|f^*|$ fois dans la boucle, où $|f^*|$ est la valeur d'un flot maximum. Mais cela ne suffit pas pour affirmer que l'algorithme s'exécute en temps polynomial : il suffit qu'un arc possède une capacité exponentielle en la taille du réseau pour contredire cette affirmation. Et même si toutes les capacités étaient polynomiales, il n'est pas difficile

de trouver des cas où le temps d'exécution résultant (à savoir $O(|f^*||A|)$) est loin d'être satisfaisant, comme le montre l'exemple suivant.

Exemple 46. Un exemple d'instance pathologique quand on se contente d'une augmentation de 1 à chaque étape de la méthode de Ford-Fulkerson (source : Cormen et al. [1]).



L'approche d'Edmonds-Karp ne sélectionne pas nécessairement le meilleur chemin à chaque étape, mais possède l'avantage d'une garantie sur le nombre d'itérations effectuées : on peut prouver, et on l'admettra tel quel, que le nombre d'augmentations effectuées — et donc d'itérations de la boucle — est $O(|V||A|)$. Au final, l'algorithme de Ford et Fulkerson basé sur l'approche d'Edmonds et Karp possède donc une complexité en $O(|V||A|^2)$.

6.3 L'algorithme de Dinitz

Dinitz a également proposé un algorithme permettant de calculer un flot maximum dans un réseau, qui permet d'obtenir un flot maximum en temps $O(|V|^2|A|)$ plutôt qu'en $O(|V||A^2|)$ pour Edmonds-Karp, ce qui est donc une amélioration puisque $|A| \geq |V|$ pour un réseau connexe. Il est également possible, à l'aide de structures de données plus avancées, de réduire sa complexité à $O(|V||A|\log|V|)$; le lecteur curieux consultera l'article de Sleator et Tarjan [5] sur les *link-cut trees* ou *dynamic trees*.

L'algorithme de Dinitz suit la méthode de Edmonds et Karp, à deux différences près :

1. on maintient une structure contenant *tous* les plus courts chemins de s à t dans le réseau résiduel ;
2. au lieu d'augmenter le flot sur un seul chemin de s à t dans le réseau résiduel, on l'augmente sur *tous* les chemins trouvés dans la structure sus-mentionnée : ce n'est donc plus directement sur le réseau résiduel que l'on travaille.

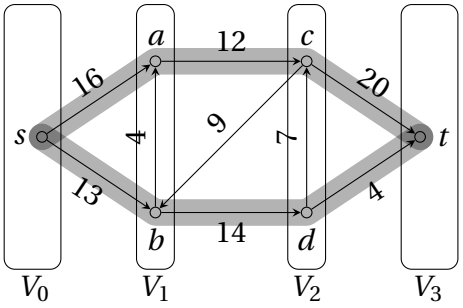
Lorsque le flot a été augmenté sur tous les chemins, on recalcule la structure sur le réseau résiduel résultant, et l'on réitère le calcul de flot. La structure que l'on construit sur base du réseau résiduel est la suivante.

Définition 34. Soit $G = (V, A)$ un graphe orienté, et s un sommet de ce graphe. Le *graphe de parcours en largeur de G au départ de s* est le graphe H défini par :
— $V(H) = \cup_{i=0}^{d} V_i$, où V_i est l'ensemble des sommets de G à distance i de s et d est la distance du sommet le plus éloigné de s ;
— $E(H) = \cup_{i=1}^{d} E_i$, où $E_i = \{(u, v) \mid u \in V_{i-1}, v \in V_i\}$.

Il s'agit donc d'un graphe acyclique similaire à l'arbre de parcours en largeur au départ de s, si ce n'est que cette fois-ci, on sauvegarde tous les plus courts chemins issus de s au lieu

d'un seul par sommet.

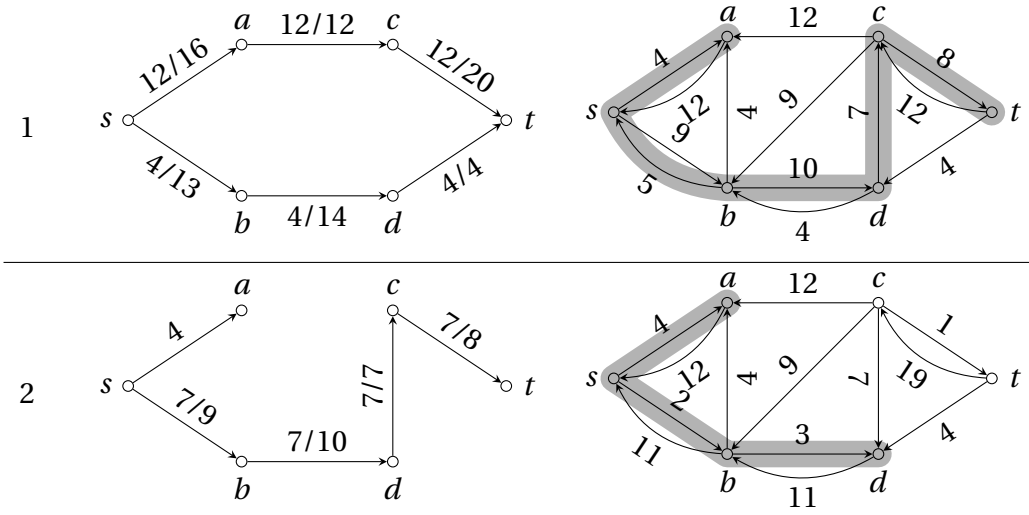
Exemple 47. Voici un graphe de parcours en largeur au départ de s pour le réseau de flot de l'exemple 42 et la partition correspondante des sommets selon leur distance à s :



Exercice 11. Écrivez l'algorithme DAGLARGEUR prenant en entrée un graphe orienté G, un sommet s et un sommet t et renvoyant le graphe acyclique d'exploration en largeur de G au départ de s jusque t.

Dans le contexte qui nous intéresse, on veut calculer un flot maximum d'un sommet s à un sommet t dans un réseau G. On note $G_L(s,t)$ le graphe obtenu à partir du graphe de parcours en largeur de G au départ de s en supprimant les chemins ne se terminant pas par t. La seule source de $G_L(s,t)$ est donc s, et son seul puits est donc t.

Exemple 48. Voici les étapes de l'algorithme de Dinitz sur le réseau de l'exemple 42, où l'on a condensé l'augmentation de flot sur chaque chemin de s à t en une seule étape. À chaque étape, on augmente au maximum le flot sur le chemin sélectionné, et on met à jour le graphe d'exploration en largeur manipulé. Lorsque plus aucun chemin de s à t n'existe dans ce graphe, on construit le nouveau réseau résiduel résultant, on recalcule le graphe d'exploration en largeur, et l'on recommence. L'algorithme se termine quand il n'est plus possible de construire un graphe d'exploration en largeur au départ de s qui contiendrait également t.



L'algorithme 33 illustre cette approche, en cherchant à chaque itération un flot dit *bloquant*

dans le DAG de parcours en largeur G_L ; il est tel qu'une fois l'augmentation réalisée, G_L ne contient plus de chemin de la source au puits, d'où l'adjectif "bloquant". Chaque fois que l'on arrive à trouver un flot bloquant, le flot actuel subit les augmentations correspondantes. Il faut ensuite mettre à jour le réseau résiduel et recalculer G_L, puisqu'on ne peut par définition plus augmenter le flot dans le G_L actuel.

Algorithme 33 : DINITZ(G)

Entrées : un réseau de flot G.
Sortie : un flot maximum pour G.

```
1  flot ← tableau associatif (clés = G.arcs(), valeurs = 0);
2  source ← unique sommet de degré entrant nul de G;
3  puits ← unique sommet de degré sortant nul de G;
4  G_f ← G;
5  G_L ← DAGLARGEUR(G, source, puits);
6  arcs ← FLOTBLOQUANT(G_L, f, source, puits);
7  tant que arcs ≠ ∅ faire
8      METTREAJOURRÉSIDUEL(G, G_f, arcs, flot);
9      G_L ← DAGLARGEUR(G_f, source, puits);
10     arcs ← FLOTBLOQUANT(G_L, f, source, puits);
11 renvoyer flot;
```

Algorithme 34 : FLOTBLOQUANT(G_L, f, source, puits)

Entrées : un graphe orienté acyclique pondéré G_L, un flot f, une source et un puits.
Résultat : le flot f augmente au maximum le long de chaque chemin de G_L, qui est mis à jour au fur et à mesure ; renvoie l'ensemble des arcs qui ont subi un changement de flot.

```
1  arcs ← ∅;
2  chemin ← CHEMINAUGMENTANT(G_L, source, puits);
3  tant que chemin ≠ NIL faire
4      AUGMENTERFLOT(f, chemin);
5      METTREAJOURDAGLARGEUR(G, G_L, chemin.arcs(), f);
6      arcs ← arcs ∪ chemin.arcs();
7      chemin ← CHEMINAUGMENTANT(G_L, source, puits);
8  renvoyer arcs;
```

Il est possible d'améliorer la version basique de l'algorithme de Dinitz présenté ici. Outre l'utilisation des *link / cut trees* mentionnée plus haut, on pourrait :

— nettoyer G_L en supprimant les éléments qui deviennent inutiles ; les suppressions d'arcs peuvent en effet donner lieu à :

1. des chemins issus de s qui n'arriveront pas à t, ce qu'on peut repérer en cherchant les sommets de degré sortant nul ;
2. des chemins arrivant en t mais ne remontant pas jusqu'à s, ce qu'on peut repérer en cherchant les sommets de degré entrant nul ;
3. ou encore d'autres composantes inaccessibles à partir de s et ne menant pas non plus à t.

Algorithme 35 : METTREAJOURDAGLARGEUR(G, G_L, arcs, f)

Entrées : un réseau de flot G, un graphe orienté acyclique pondéré G_L, un ensemble d'arcs, et un flot f.

Résultat : le poids des arcs spécifiés de G_L est mis à jour sur base de la valeur du flot associé ; les arcs dont le poids devient nul sont supprimés.

```
1 pour chaque (u, v, c) ∈ arcs faire
2     si G.contient_arc(u, v) alors                    // arc original du réseau
3         capacité_initiale ← G.poids_arc(u, v) ; flot_actuel ← f[(u, v)];
4     sinon                                            // arc inverse du réseau
5         capacité_initiale ← G.poids_arc(v, u) ; flot_actuel ← f[(v, u)];
6     si flot_actuel = capacité_initiale alors G_L.supprimer_arc(u, v) ;
7     sinon G_L.ajouter_arc(u, v, capacité_initiale − flot_actuel) ;
```

— si on a la garantie que G_L a été nettoyé comme expliqué ci-dessus, on peut chercher un chemin de s à t plus efficacement : on remonte de t à s en n'examinant qu'un seul de ses prédécesseurs à chaque étape, et on s'arrête quand on retombe sur s.

6.4 Applications des flots

L'étude des flots a été motivée au début de ce chapitre par un problème d'acheminement d'une quantité d'un produit dans un réseau. Outre les applications où l'on a directement besoin de résoudre cette famille de problèmes, et les applications légèrement plus éloignées d'optimisation du trafic (par exemple : il vaut mieux répartir le trafic sur un réseau routier pour éviter les congestions qu'envoyer tout le monde sur le même plus court chemin, dont la capacité est limitée), il existe quelques autres problèmes que l'on peut résoudre à l'aide d'algorithmes de calcul de flots maximum avec lesquels les liens sont moins évidents. On va en examiner quelques-uns dans cette section.

6.4.1 Couplages dans un graphe biparti

Un certain nombre d'applications nécessite de trouver des arêtes ou des arcs dans un graphe dont les extrémités sont disjointes. Plus formellement, on cherche un *couplage*, dans le sens défini ci-dessous dans le cas d'un graphe non orienté (mais qui se généralise facilement aux graphes orientés).

Définition 35. Soit $G = (V, E)$ un graphe. Un *couplage* dans G est un ensemble $F \subseteq E$ d'arêtes deux à deux disjointes ; on dit que le couplage est *maximum* s'il n'existe aucun autre couplage possédant plus d'arêtes (ou d'arcs), et *parfait* s'il couvre tous les sommets du graphe. Si le graphe est pondéré, le *poids* du couplage est la somme des poids de ses arêtes.

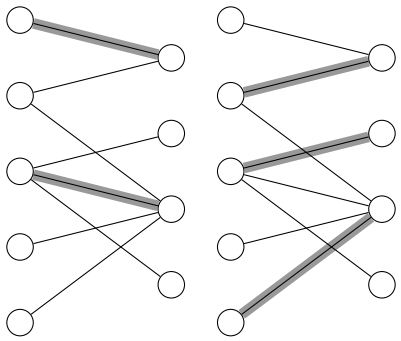
Dans le cas des graphes bipartis, les arêtes du couplage relient toujours deux sommets n'appartenant pas à la même classe, et la recherche d'un tel couplage intervient dans un grand nombre d'applications assez naturelles dont voici quelques exemples :

— dans le cadre d'un projet informatique, on dispose d'une équipe de n développeurs

que l'on désire affecter à n tâches sur lesquelles ils travailleront en parallèle ; le chef de projet connaît les compétences de chacun, et peut donc construire un graphe biparti dans lequel il relie un développeur à une tâche s'il estime le développeur capable de la réaliser. Un couplage parfait dans ce graphe lui permet donc d'occuper tous les développeurs en parallèle tout en étant sûr que toutes les tâches seront réalisées.

— un particulier veut effectuer des travaux de rénovation dans sa maison ; il crée la liste des n tâches à réaliser, et demande ensuite à n entrepreneurs quels seraient leurs tarifs pour chacune de ces tâches à réaliser. Ceci lui permet de construire un graphe biparti dans lequel chaque entrepreneur est relié à une tâche par une arête dont le poids est le prix demandé par l'entrepreneur. Un couplage parfait de poids minimum dans ce graphe permet au particulier de réaliser tous les travaux au coût le plus bas.

Exemple 49. Voici un graphe biparti avec deux couplages : le premier (a) est maximal, car on ne peut pas lui rajouter d'arête tout en conservant un couplage, mais pas maximum. En effet, le second couplage (b) possède une arête de plus ; et on verra plus loin qu'il est bien maximum, dans le sens où il n'existe pas d'autre couplage de taille plus grande sur ce graphe.



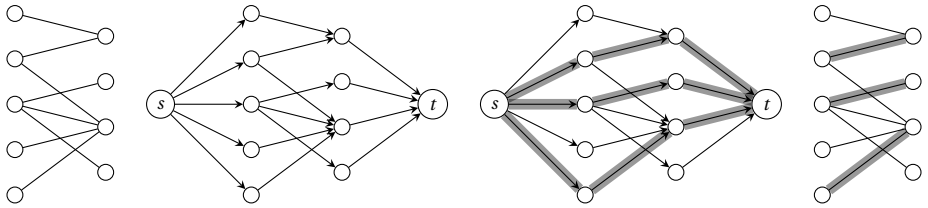
(a) (b)

Les techniques de calcul d'un flot maximum nous permettent de trouver un couplage maximum dans un graphe biparti $G = (V_1, V_2, E)$ comme suit :

1. orienter toutes les arêtes de G de V_1 vers V_2 ;
2. rajouter une source s artificielle ayant pour successeurs tous les sommets de V_1 ;
3. rajouter un puits t artificiel ayant pour prédécesseurs tous les sommets de V_2 ;
4. affecter à tous les arcs une capacité de 1 ;
5. trouver un flot maximum dans le réseau résultant.

Les arêtes du couplage seront les arêtes de G saturées par le flot.

Exemple 50. Voici comment on peut trouver un couplage maximum dans un graphe biparti en trouvant un flot maximum :



(1) (2) (3) (4)

(fin exemple 50)

L'algorithme 36 montre le pseudocode résultant.

Algorithme 36 : COUPLAGEBIPARTI(G)

Entrées : un graphe biparti connexe $G = (V_1 \cup V_2, E)$.
Sortie : un couplage maximum pour G.

```
1 H ← GrapheOrientéPondéré();
2 H.ajouter_sommet(s);
3 H.ajouter_sommet(t);
4 pour chaque {u, v} ∈ G.aretes() avec u ∈ V1 et v ∈ V2 faire
5 |   H.ajouter_arc(s, u, 1); H.ajouter_arc(u, v, 1); H.ajouter_arc(v, t, 1);
6 flot ← DINITZ(H);
7 renvoyer {{u, v} | {u, v} ∈ E et flot[(u, v)] = 1};
```

Correction

Montrons pour commencer que l'approche proposée est correcte, et ce quel que soit l'algorithme utilisé pour obtenir le flot maximum.

Proposition 6.4.1. Soit $S \subseteq A$ les arcs auxquels un flot de 1 est affecté après le calcul d'un flot maximum f dans G' ; alors l'ensemble M des arcs de S dont les extrémités sont toutes deux dans G est un couplage maximum pour G.

Démonstration. On a deux propriétés à prouver, et on peut les prouver par contradiction :

1. M est un couplage : si l'origine (ou la destination) de a_1 et de a_2 coïncident, alors un flot de 2 sort de l'origine de a_1 (ou rentre dans la destination de a_2), ce qui contredit le principe de conservation des flots puisque l'origine de a_1 (resp. sa destination) ne peut recevoir (resp. produire) qu'un flot de 1.
2. M est maximum : supposons qu'il existe un autre couplage M' avec $|M'| > |M|$; les arêtes de ce couplage correspondent à des arcs dans G' que l'on peut connecter à s et à t pour obtenir un flot f' de valeur supérieure à f, ce qui contredit l'hypothèse que f est maximum.

□

Complexité

On aurait pu utiliser n'importe quel algorithme de calcul de flot, en particulier l'approche d'Edmonds-Karp, mais on a choisi ici celui de Dinitz pour obtenir une meilleure complexité. La complexité que l'on obtiendra sera bien meilleure dans les deux cas que celle que l'on a obtenue pour le calcul de flot traditionnel, d'une part parce que le graphe de départ est

particulier, et d'autre part parce que tous les arcs ont le même poids unitaire. Si l'on avait choisi l'algorithme d'Edmonds-Karp, on aurait obtenu l'analyse suivante :

— la construction du réseau se fait en $O(|V|)$;
— la valeur $|f^*|$ d'un flot maximum f^* est $O(|V|)$: au mieux, on peut utiliser à pleine capacité tous les arcs issus de la source ;
— comme on l'a vu dans l'analyse de complexité de l'algorithme de Ford-Fulkerson, on peut borner le temps d'exécution par $O(|f^*||A|)$.

On en conclut donc que l'on peut trouver un couplage maximum dans un graphe biparti par cette méthode en temps $O(|V||E|)$, au lieu du $O(|V||A|^2)$ que l'on avait obtenu avant.

Si l'on utilise comme ici l'algorithme de Dinitz, la complexité est encore meilleure mais l'analyse est plus complexe. On retiendra simplement que l'algorithme 36, quand on utilise bien l'algorithme de Dinitz, possède une complexité en $O(\sqrt{|V|}|E|)$.

On ne peut malheureusement pas utiliser directement la même transformation dans le cas où le couplage à trouver utilise des arêtes pondérées (que l'on cherche à maximiser ou à minimiser le coût total). En effet, dans le cas où tous les arcs du réseau ont le même poids unitaire, un chemin utilisant un arc le saturera forcément ; mais dans le cas où les coûts ne sont plus unitaires, un chemin utilisant un arc ne le saturera pas forcément, et on ne peut donc plus garantir qu'un sommet ne sera pas utilisé plusieurs fois.

6.5 Pour aller plus loin

Flots. Cormen et al. [1] expliquent avec beaucoup plus de détails des techniques plus avancées pour calculer des flots efficacement. À l'heure actuelle, l'algorithme possédant la meilleure complexité pour le calcul d'un flot maximum est celui de Orlin [3], qui s'exécute en temps $O(|V||A|)$. Cela ne veut pas nécessairement dire que l'on peut automatiquement oublier tout ce qui a été fait avant : comme on l'a vu dans le cas des couplages, certains algorithmes nous permettent de donner des garanties plus fortes sur le temps d'exécution quand on les exécute sur des graphes particuliers comme les graphes bipartis, et un algorithme de meilleure complexité pour le cas général ne sera pas automatiquement meilleur qu'un autre dans tous les cas particuliers.

Couplages. De nombreuses autres variantes du problème de couplage que l'on a mentionné ont été étudiées. On peut envisager entre autres les cas suivants :

— le graphe d'entrée n'est pas biparti, ce qui peut se produire par exemple quand on veut grouper des éléments deux par deux sans avoir de raison d'exclure certaines associations ;
— le graphe d'entrée est orienté, ce qui permet de représenter des préférences : par exemple, pour représenter des demandeurs d'emplois qui ont chacun des préférences pour les entreprises qui les intéresse, sachant que les entreprises elles-mêmes ont également des préférences pour certains candidats qui ne coïncident pas forcément avec celles de ces derniers.

Le lecteur curieux pourra consulter Plummer et Lovász [4] pour en savoir plus sur la théorie développée pour résoudre ces problèmes, ainsi que Manlove [2] pour en découvrir d'autres applications.

Bibliographie

[1] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, ET C. STEIN, *Introduction to Algorithms*, MIT Press, 3ème édition, 2009.

[2] D. F. MANLOVE, *Algorithmics of Matching Under Preferences*, vol. 2 de Series on Theoretical Computer Science, WorldScientific, 2013.

[3] J. B. ORLIN, *Max flows in $O(nm)$ time, or better*, dans Symposium on Theory of Computing Conference (STOC), édité par D. Boneh, T. Roughgarden, et J. Feigenbaum, Palo Alto, CA, Juin 2013, ACM, pages 765–774.

[4] M. D. PLUMMER ET L. LOVÁSZ, *Matching Theory*, North-Holland, 1986.

[5] D. D. SLEATOR ET R. E. TARJAN, *A data structure for dynamic trees*, Journal of Computer and System Sciences, 26 (1983), pages 362–391.