Louis Jachiet & Bertrand Meyer

CSC-4MI13 : PERLES DE PROGRAMMATION DE STRUCTURES DE DONNÉES ET ALGORITHMES

Louis Jachiet & Bertrand Meyer

Automne 2024

1. Version du 25 octobre 2024.

CSC-4MI13 : PERLES DE PROGRAMMATION DE STRUCTURES DE DONNÉES ET ALGORITHMES

Louis Jachiet & Bertrand Meyer

Ces notes sont en cours d'écriture et contiennent probablement des erreurs ou des coquilles. Tout retour sera apprécié.

TABLE DES MATIÈRES

CHAPITRE 1

STRUCTURES DE DONNÉES SUR INTERVALLES

Dans tout ce chapitre nous nous intéresserons à des structures de données capables de répondre à divers problèmes.

1.1. Un problème général

Les structures de données que nous allons étudier dans ce chapitre vont répondre à des variantes (ou extensions) du problème suivant.

***Problème 1.1* (Requête somme sur tableau avec mise à jour)**
Étant donné un monoïde $(M, +)$ et un tableau T de longueur n d'éléments on souhaite une structure de données capable de répondre rapidement aux requêtes suivantes :

- *`Construction`(T), qui prend en argument un tableau T de longueur n et initialise la structure avec le contenu de T*
- *`Somme`(i, j), qui prend en arguments $0 \leq i \leq j < n$ et la structure doit renvoyer la valeur de $\sum_{i \leq k < j} T[k]$.*
- *`MiseÀJour`(i, v), qui prend en argument $0 \leq i < n$ et doit modifier la tableau T stocké de façon à ce que $T[i] = v$.*

Un algorithme naïf pour ce problème. — Un premier algorithme auquel on peut penser consiste à simplement stocker le tableau lors de `Construction`, à faire la somme pour `Somme` et à modifier le tableau pour `MiseÀJour`. Les complexités des opérations sont $O(n)$ pour `Construction` et `Somme` mais $O(1)$ pour `MiseÀJour`.

Un algorithme avec précalcul. — Un second algorithme consiste à précalculer les réponses aux requêtes `Somme` lors de `Construction`. On peut alors répondre à `Somme` immédiatement mais la `MiseÀJour` est aussi coûteuse que la reconstruction complète du tableau. Attention, pour obtenir une construction rapide, il faut remarquer que $\texttt{Somme}(i, j+1) = \texttt{Somme}(i, j) + T[j]$ quand $j > i$. Les complexités des opérations sont donc $O(n^2)$ pour `Construction` et `MiseÀJour` mais $O(1)$ pour `Somme`.

Une technique avancée : la Square-root decomposition. — Le Square-root decomposition est une technique qui peut intervenir dans différents pans de l'algorithmique et qui permet souvent de d'améliorer un algorithme inefficace.

Le principe général de cette technique est d'améliorer un algorithme A travaillant sur un objet de taille n en coupant cet objet en n/B bouts chacun de taille B (en général on choisit B proche de $\sqrt{n}$).

Nous allons maintenant présenter un exemple de cette technique dans le cadre de notre problème avec l'amélioration des algorithmes ci-haut.

Square-root decomposition sur algorithme naïf. — Pour cela, on va choisir un nombre B (que l'on précisera plus tard) et l'on va découper notre tableau de valeurs en n/B tableaux contenant au plus B valeurs. Pour gérer une requête construction, il suffit de faire n appels à des requêtes construction sur les tableau de taille B via l'algorithme naïf, cela donne un temps de calcul $n/B \times O(B) = O(n)$ pour la construction. Pendant la construction on va aussi calculer S_i, la somme pour chaque bloc i de taille B des éléments qu'il contient (remarquer que cela n'augmente pas la complexité).

Ensuite pour faire une requête `Somme`(i,j) on regarde les blocs $b_i = \lfloor i/B \rfloor$ et $b_j = \lfloor j/B \rfloor$ et la réponse notre requête peut alors se calculer comme la somme d'une requête sur le bloc b_i, la somme des S_k pour $i < k < j$ plus une requête sur le bloc b_j dans le cas $j \neq i$. La complexité de somme est alors $O(B) \times 2 + O(n/B) = O(n/B + B)$.

Enfin pour faire une requête `MiseÀJour`(i, v) on va faire une requête `MiseÀJour` sur le bloc où est stocké i puis recalculer la somme de ce bloc. Tout cela peut se faire en $O(B)$.

En choisissant $B = \sqrt{n}$, on a alors les complexités $O(n)$ pour `Construction`, $O(\sqrt{n})$ pour `Somme` et `MiseÀJour`. On remarque que cet algorithme n'est pas strictement meilleur que l'algorithme naïf car le temps de `MiseÀJour` à augmenté de $O(1)$ à $O(\sqrt{n})$.

Square-root decomposition sur algorithme avec précalcul. — Une manière d'appliquer la Square-root decomposition sur l'algorithme avec précalcul consiste à découper en blocs de taille B et à appliquer l'algorithme avec précalcul sur chaque bloc. Comme dans le cas naïf on va aussi calculer pour chaque bloc i sa somme $S[i]$ mais on va utiliser l'algorithme de précalcul sur le tableau S. La construction va donc coûter $O(B^2)$ pour chacun des n/B blocs puis $O((n/B)^2)$ pour la structure gérant S ce qui donne un total de $O(n^2/B^2 + nB)$.

Pour les requêtes `Somme`, les requêtes sont transformés en deux requêtes sur des blocs de taille B et une requête sur S, chacune de ses 3 requêtes prenant un temps $O(1)$ ce qui donne un total de $O(1)$.

Enfin pour faire une `MiseÀJour` on doit mettre à jour le bloc dans lequel on veut modifier l'indice mais aussi la structure pour S ce qui coûte $O(B^2 + (n/B)^2)$.

De nouveau on peut choisir ici $B = \sqrt{n}$ ce qui donne les complexités $O(n^{3/2})$ pour la construction, $O(1)$ pour les requêtes et $O(n)$ pour les mises à jour, ce qui est une amélioration sur toutes les opérations par rapport à l'algorithme avec précalcul.

Si on choisit $B = n^{1/3}$ on obtient $O(n^{4/3})$ pour la construction et la mise à jour ce qui est aussi une alternative intéressante.

Peut-on faire une "Recursive square-root decomposition" ? — Une extension naturelle de la Square-root decomposition consiste à appliquer la square-root decompostion à un algorithme issu de la square-root decomposition voire à itérer un nombre important de fois ce procédé. Il y a des problèmes pour lesquels cette idée fonctionne (les arbres de van Emde Boas peuvent être vus comme une application de cette technique) mais dans le cas qui nous intéresse nous allons voir une solution plus simple et efficace utilisant des arbres binaires.

Solution récursive utilisant des arbres binaires. — Cette fois-ci on propose une approche récursive qui fonctionne de la façon suivante, si $n = 1$ alors on stocke la valeur lors de la construction, on répond 0 pour $\texttt{Somme}(0, 0)$ ou $\texttt{Somme}(1, 1)$ et la valeur pour `Somme(0,1)` tandis qu'une requête de mise à jour consiste à simplement changer la valeur.

Si $n > 1$ alors on va stocker n, la somme de toutes les valeurs dans T ainsi que deux sous-structures, une "gauche" de taille $k = \lceil n/2 \rceil$ qui gère $T[0]$ à $T[k-1]$ et une "droite" qui gère de $T[k-1]$ à $T[n-1]$. Ces structures forment alors un arbre binaire complet tassé à gauche (voir la figure 1.1).

Pour faire une requête $\texttt{Somme}(i, j)$ on commence par regarder si $i = 0$ et $j = n$ et dans ce cas on renvoie la somme totale. Sinon, si $j < k = \lceil n/2 \rceil$ on renvoie $\texttt{Somme}(i, j)$ sur la sous-structure de gauche. Sinon, si $i \geq k$ alors on renvoie $\texttt{Somme}(i-k, j-k)$. Enfin aucune de ces conditions n'est vérifiée on renvoie alors $\texttt{Somme}(i, k) + \texttt{Somme}(0, j-k)$.

Pour le calcul de la complexité, il faut faire attention aux i j extrêmes, c'est-à-dire quand $i = 0$ ou $j = n$. Notre structure va faire divers appels récursifs mais, à chaque

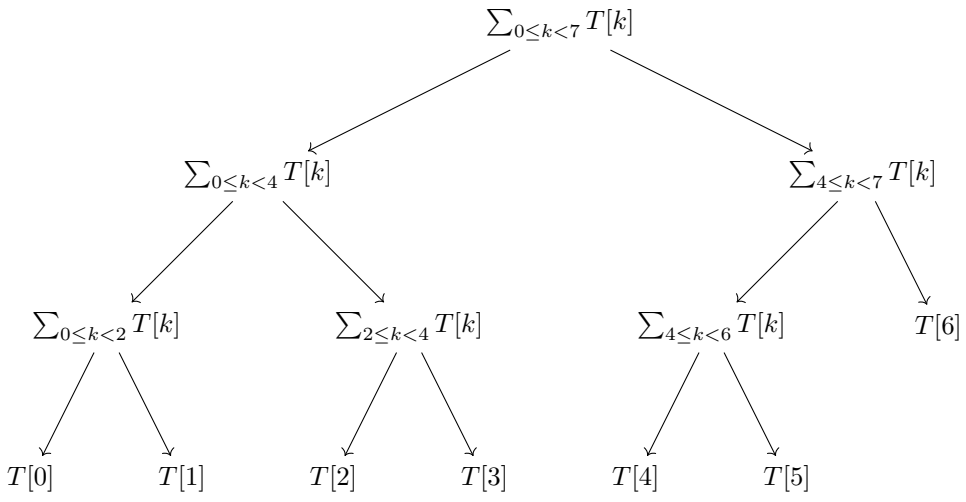


FIGURE 1.1. Exemple d'arbre pour T de taille 7

profondeur, il ne peut y avoir qu'un appel avec un i non extrême et qu'un appel avec un j non extrême, ce qui signifie qu'à chaque profondeur il n'y a au plus que 4 appels (dont 2 qui terminent immédiatement). La complexité de `Somme` est donc $O(\log_2(n))$ (le temps par nœud est $O(1)$).

Pour `MiseÀJour` l'algorithme est similaire mais plus simple, à chaque fois on met à jour récursivement la bonne sous-structure puis on recalcule la somme du nœud courant (qui est la somme des deux sous-nœuds).

1.2. Vers des problèmes plus généraux : mise à jour sur intervalle

Nous allons maintenant étudier des problèmes plus généraux et nous verrons comment la structure d'arbre peut-être étendue pour ces problèmes.

Problème 1.2 **(Requête somme sur tableau avec mise à jour sur intervalle)**
Il s'agit d'une variante du problème précédent :

- *`Construction`(T), qui prend en argument un tableau T de longueur n et initialise la structure avec le contenu de T*
- *`Somme`(i, j), qui prend en arguments $0 \leq i \leq j < n$ et la structure doit renvoyer la valeur de $\sum_{i \leq k < j} T[k]$.*
- *`MiseÀJour`(i, j, v), qui prend en argument $0 \leq i \leq j \leq n$ et doit modifier la tableau T stocké de façon à ce que $T[k] += v$ pour tout $i \leq k < j$. Pour cette requête on va supposer qu'il est possible de calculer en $O(1)$ la valeur $n \times v = v + \cdots + v$ pour tout n.*

Pour répondre à ce problème nous allons reprendre la structure de données à base d'arbres binaire de la section précédente. Comme dans la version précédente, chaque nœud va stocker le nombre n d'éléments et la sommes des éléments qu'il contient mais il pourra aussi stocker une valeur spéciale V, indiquant que l'on doit ajouter à tous les éléments la valeur V.

La construction s'effectue comme la construction dans le cas de mise à jour d'un unique élément. Pour `MiseÀjour` et `Somme` on va introduire une fonction `MarqueNoeud`(u, V) qui marque que le nœud u a tous ses éléments remplacés par V (et donc dans le cas d'une feuille on remplace la valeur qu'elle contient et dans le cas d'un noeud interne il faut changer la somme de u en lui ajoutant $n \times V$ et marquer la valeur spéciale de u).

On va aussi introduire une fonction `RentreNoeud` qui sera appelée dès que les fonctions `Somme` et `MiseÀJour` arrivent récursivement dans un nœud u. Cette fonction regarde si V est défini dans le nœud u, si oui, elle le supprime ce marquage et appelle `MarqueNoeud`(w, V) sur chacun des fils w de u.

La gestion des requêtes `Somme` est très similaire au cas des `Somme` avec mise à jour d'un unique élément à la différence qu'au début de chaque appel sur un noeud on fait appel à `RentreNoeud`.

La gestion des requêtes `MiseÀJour`(i, j, V) se fait de la façon suivante : on descend récursivement dans l'arbre comme dans le cas des requêtes `Somme` mais au lieu de faire

la somme des noeuds pour lesquels on appel avec des $i = 0, j = n$, on marque ces nœuds avec la valeur V.

1.3. Vers des problèmes plus généraux : rapport de points et d'intervalles

Problème 1.3 **(Rapport de point).** — *On souhaite une structure de données capable de stocker une liste de valeurs d'un ensemble totalement ordonnées (souvent $\mathbb{R}$) et qui supporte efficacement les requêtes :*

- `ajoute`(x) *qui ajoute x,*
- `retire`(x) *qui retire x,*
- `suivant`(x)*, qui renvoie le plus petit y dans la structure tel que $y > x$,*
- `précédant`(x)*, qui renvoie le plus grand y dans la structure tel que $y < x$.*

Solution. — On remarque qu'un Arbre Binaire de Recherche (ABR) contient déjà toute l'information qu'il faut pour répondre à ce problème, les requêtes `suivant` ou `précédant` allant chercher les noeuds qui suivent ou précèdent dans l'ordre infixe. Noter qu'en maintenant l'ABR équilibré, on obtient une complexité $O(\log(n))$ pour toutes les opérations.

Problème 1.4 **(Requête d'intervalle, variante dynamique)**

On souhaite une structure de données capable de stocker une liste d'intervalles ($[a_i; b_i[$) et qui supporte efficacement les requêtes :

- `Construction`$()$*, qui renvoie une structure vide (sans intervalles)*
- `Ajoute`$([a; b[)$*, qui prend deux réels $a \leq b$ et ajoute à la structure l'intervalle $[a; b[$;*
- `Retire`$([a; b[)$*, qui prend deux réels $a \leq b$ et retire à la structure l'intervalle $[a; b[$;*
- `valeurPoint`(x)*, qui prend un réel x et renvoie la liste des intervalles $[a_i; b_i[$ qui contiennent x (i.e. $x \in [a_i; b_i[$).*

Solution. — L'idée ici est de considérer l'ensemble $E = \cup\{a_i, b_i\}$ des nombres qui apparaissent comme la borne gauche ou droite d'un intervalle et de maintenir un ABR sur E. Les insertions ne posent pas de vrais problèmes mais les rotations sont, elles, assez compliquées à gérer pour les informations que l'on va vouloir maintenir. Le plus simple est donc de supposer que l'on connaît à l'avance un ensemble E' pas trop gros (toutes les complexités dépendront de ce E') qui contient E et l'on fait un ABR sur E'.

Dans l'ABR que l'on a, on définit récursivement l'intervalle $I(u)$ pour chaque noeud u. Pour la racine il s'agit de l'intervalle $[-\infty; +\infty[$ et pour un noeud u qui correspond à l'intervalle $[a; b]$ et donc la clef est c, son fils gauche correspond à $[a; c[$ tandis que le fils droit correspond à $[c; b[$.

Dans cet ABR, chaque noeud va stocker une partie des intervalles ajoutés. Quand on ajoute un intervalle $[a_i; b_i[$, il sera stocké dans tous les noeuds u tel que $I(u) \subseteq [a_i; b_i[$ sauf ceux dont le parent p est aussi tel que $I(u) \subseteq [a_i; b_i[$.

On peut facilement montrer que l'ajout se fait en $O(\log n)$ et que l'intervalle ne sera ajouté qu'à $2\log n$ noeuds.

Pour savoir la liste des intervalles qui coupent un point x, il suffit de renvoyer la liste des intervalles qui apparaissent dans un noeud du chemin de la feuille f tel que $I(f)$ contient x jusqu'à la racine (il n'y a pas de doublons). Cette feuille f peut être trouvée à l'aide d'un algorithme classique de recherche dans un ABR.

1.4. Structure somme pour les groupes

Considérons le problème original mais sans mise à jour et pour les groupes.

***Problème 1.5* (Requête somme sur tableau).** — *Étant donné un monoïde $(M, +)$ et un tableau T de longueur n d'éléments on souhaite une structure de données capable de répondre rapidement aux requêtes suivantes :*

- *`Construction`(T), qui prend en argument un tableau T de longueur n et initialise la structure avec le contenu de T*
- *`Somme`(i, j), qui prend en arguments $0 \leq i \leq j < n$ et la structure doit renvoyer la valeur de $\sum_{i \leq k < j} T[k]$.*

Solution. — Une solution pour ce problème consiste à précalculer le tableau T_Σ tel que $T_\Sigma[k] = \sum_{i<k} T[i]$. La requête peut alors être répondu en $O(1)$ via le calcul $T_\Sigma[j] - T_\Sigma[i]$.

1.5. Structure somme pour les lois absorbantes

Considérons le problème original, sans mise à jour et pour les monoïdes "absorbants" c'est-à-dire des monoïdes commutatifs, associatifs et tels que $\forall a, b, (a+b)+a = (a+b) = (a+b)+b$.

***Problème 1.6* (Requête somme sur tableau).** — *Étant donné un monoïde absorbant $(M, +)$ et un tableau T de longueur n d'éléments on souhaite une structure de données capable de répondre rapidement aux requêtes suivantes :*

- *`Construction`(T), qui prend en argument un tableau T de longueur n et initialise la structure avec le contenu de T*
- *`Somme`(i, j), qui prend en arguments $0 \leq i \leq j < n$ et la structure doit renvoyer la valeur de $\sum_{i \leq k < j} T[k]$.*

Solution. — Une solution pour ce problème consiste à précalculer les $\log n$ tableaux $T_i[j] = \sum_{j \leq k < j+2^i} T[k]$. Une requête `somme`$(i, j)$ devient alors $T_k[i] + T_k[j - 2^k]$ pour k maximal tel que $i + 2^k \leq j$ c'est à dire $k \leq \log_2(j - i)$.

CHAPITRE 2

COUPLAGES

2.1. Généralités sur les couplages

2.1.1. Couplages maximum. — Dans tout ce chapitre, nous travaillons avec un *graphe non-orienté* $G = (V, E)$, c'est-à-dire un couple où V est un ensemble fini de *sommets* et E un ensemble de paires (non ordonnées) de sommets.

Définitions 2.1. — Un *couplage* dans un graphe non orienté $G = (V, E)$ est un ensemble d'arête $M \subseteq E$ dont les extrémités sont toutes distinctes.

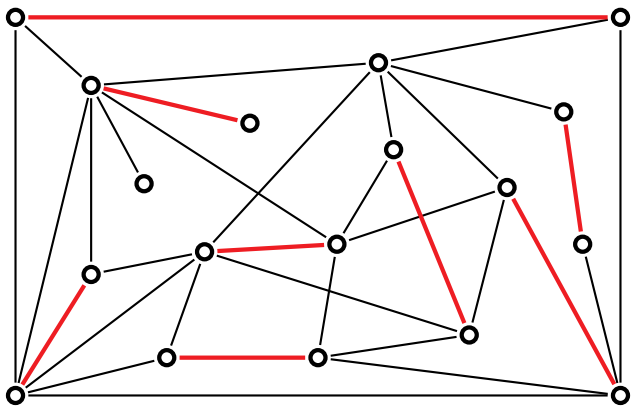


FIGURE 2.1. Couplage de cardinal 8.

Un couplage est dit *maximal* s'il est maximal pour la relation d'ordre « inclusion », c'est-à-dire qu'il n'existe pas d'arête $e \in E$ telle que $M \cup \{e\}$ est encore un couplage.

Un couplage est dit *maximum* s'il est maximum pour le cardinal, c'est-à-dire qu'il n'existe pas de couplage M' avec $|M| < |M'|$.

Nous notons $\nu(G)$ le cardinal d'un couplage maximum.

Remarque 2.2. — Le terme *maximal* fait référence au fait que M est maximal pour la relation d'ordre $\subseteq$ (inclusion) sur l'ensemble 2^E (l'ensemble des parties de E), c'est-à-dire que pour tout couplage M' tel que $M \subseteq M'$, on a en fait $M = M'$.

Nous nous intéressons à deux problèmes.

***Problème 2.3* (Couplage maximal).** — *Étant donné un graphe non orienté $G = (V, E)$, déterminer un couplage maximal.*

***Problème 2.4* (Couplage maximum).** — *Étant donné un graphe non orienté $G = (V, E)$, déterminer un couplage de cardinal maximum.*

Le problème 2.3 peut se résoudre facilement par une stratégie gloutonne. Un couplage maximal a souvent peu d'intérêt en soit. Cependant, il est en général intéressant dans la pratique d'initialiser les algorithmes de recherche de couplage maximum avec un couplage maximal pour gagner du temps à leur démarrage.

Définitions 2.5. — Nous disons qu'un sommet v d'un graphe non orienté est *couvert* par un couplage M de ce graphe s'il existe une arête appartenant à M qui est incidente à v

Un couplage M est dit *parfait* s'il couvre tous les sommets du graphe.

Un couplage M d'un graphe G est dit *presque parfait* s'il couvre tous les sommets de G sauf un.

Nous remarquons que le nombre de sommets d'un graphe $G = (V, E)$ non couverts par un couplage M est exactement

$$|V| - 2|M|. \tag{1}$$

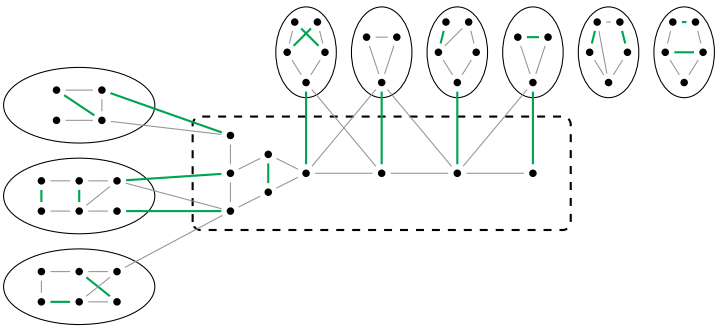
Le cas du couplage parfait correspond au cas idéal, où l'on parvient à réduire le nombre de sommets non couverts à zéro. En général, dans un graphe (V, E), il y a toujours au moins $|V| - 2\nu(G)$ sommets qui restent découverts par un couplage quelconque. Notons que toute minoration *a priori* du nombre de sommets que nous ne parviendrons jamais à couvrir nous aide à prévoir la taille d'un couplage maximum.

Définition 2.6. — Étant donnés un graphe non orienté $G = (V, E)$ et un ensemble de sommets quelconque $X \subseteq V$, nous notons $q_G(X)$ le *nombre de composantes connexes de cardinal impair* de $G \setminus X$, où $G \setminus X$ désigne le graphe obtenu en retirant de G tous les sommets de X ainsi que toute arête incidente à un sommet de X.

Proposition 2.7. — *Dans tout graphe non orienté $G = (V, E)$, le nombre de sommets non couverts par un couplage M satisfait*

$$q_G(X) - |X| \leq |V| - 2|M|.$$

Démonstration. — Soit G un graphe non orienté et M un couplage. Si $q_G(X) - |X|$ est négatif, il n'y a rien à prouver. Nous raisonnons dans le cas où $q_G(X) > |X|$.



Un sommet d'une composante connexe de $G \setminus X$ ne possède des voisins dans G que dans la même composante connexe ou dans X. Comme $q_G(X) > |X|$, il y a au moins $q_G(X) - |X|$ composantes qui ne sont pas reliées à X par une arête du couplage M. Mais comme ces composantes sont de cardinal impair, au moins un de leur sommet n'est pas couvert par M. D'où l'inégalité. □

Corollaire 2.8. — *Si le graphe $G = (V, E)$ possède un couplage parfait, alors, pour tout ensemble de sommet $X \subseteq V$, on a*

$$q_G(X) \leq |X|$$

Remarque 2.9. — Le corollaire 2.8 est en réalité une condition nécessaire et suffisante (théorème de Tutte). Nous constaterons que l'algorithme que nous présenterons dans ce qui suit fournira un couplage maximum M et un ensemble de sommets X tel que $|V| - 2\nu(G) = q_G(X) - |X|$ (formule de Tutte-Berge), ce qui fournit un certificat d'optimalité.

2.1.2. Chemins augmentants. — Rappelons le mécanisme principal sur lequel sont basés la plupart des algorithmes combinatoires de recherche de couplage maximum.

Définitions 2.10. — Soit $M \subseteq E$ un couplage d'un graphe non orienté $G = (V, E)$.
Une chaîne $P \subseteq E$ est dite alternée par rapport au couplage M, ou tout simplement *M-alternée*, si l'ensemble d'arêtes $P \setminus M$ forme aussi un couplage de G.
Une chaîne P est dite augmentante par rapport au couplage M, ou tout simplement *M-augmentante*, s'il s'agit d'une chaîne M-alternée entre deux sommets non couverts par M.

Remarque 2.11. — Dans une chaîne M-alternée, les arêtes sont toutes disjointes. Seules les extrémités de la chaîne peuvent apparaître deux fois parmi les sommets.

Théorème 2.12 **(Lemme de Berge).** — *Un couplage est maximum si et seulement s'il n'existe pas de chaîne augmentante par rapport à ce couplage.*

Démonstration. — Soit M un couplage dans un graphe G. S'il existe une chaîne M-augmentante P, alors la différence symétrique $M' = M \triangle P$ est encore un couplage de cardinal supérieur à celui de M et M n'est pas un couplage maximum. D'autre part, s'il existe un couplage M'' tel que $|M''| > M$, alors la différence symétrique

$D = M \triangle M''$ est l'union de cycles de longueur paire formés d'autant d'arêtes de M que de M'' et de chaînes alternées par rapport à M et M''. Comme $|M''| > M$, au moins une chaîne possède une arête de plus dans M'' que dans M : c'est une chaîne M-augmentante. □

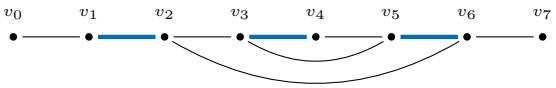
2.2. Révisions du cas biparti

TODO

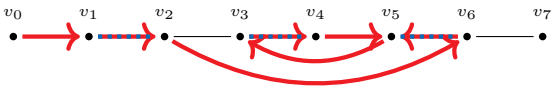
2.3. L'algorithme d'Edmonds

2.3.1. Parcours de graphe insuffisant. — La recherche d'un chemin M-augmentant ne peut pas se faire simplement par un parcours de graphe au départ d'un sommet non couvert.

Exemple 2.13. — Dans le graphe ci-dessous



un parcours de graphe au départ du sommet v_0 qui alterne les arêtes hors de M et les arêtes de M pourrait être amené à construire le chemin



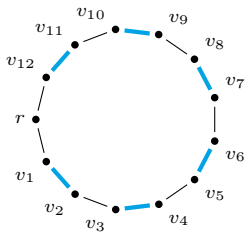
et se retrouver coincé, et ne pas découvrir qu'il existe un chemin augmentant, à savoir le chemin $(v_0, v_1, \ldots, v_7)$ qui est bien un chemin augmentant.

2.3.2. Facteurs critiques et contraction de fleurs. — Commençons par une définition en toute généralité.

Définitions 2.14. — Un graphe $C = (W, F)$ est appelé *facteur critique* si, pour tout sommet $v \in W$, le graphe $G \setminus v$ admet un couplage parfait.

Remarque 2.15. — Un facteur critique possède un nombre impair de sommets.

Exemple 2.16. — Un cycle de longueur impaire est un facteur critique, comme le montre le schéma suivant, dans lequel le sommet supprimé a été noté r et les sommets restant ont été nommés $(v_i)_{1 \leq i \leq |C|-1}$



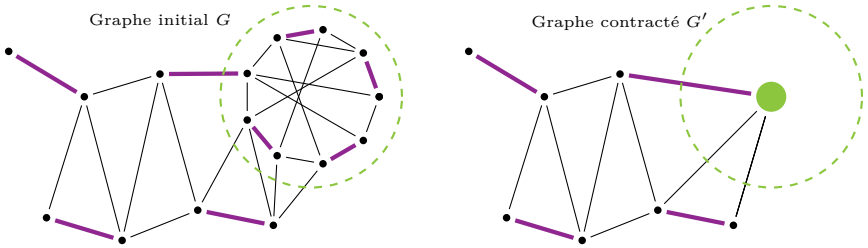
Dans le cas où un couplage existe déjà, nous pouvons rendre le vocabulaire plus précis.

Définitions 2.17. — Une *fleur* dans un graphe $G = (V, E)$ par rapport à un couplage M est un sous-graphe induit $C = (W, F)$ de G qui est un facteur critique et dont l'ensemble des arêtes F possède $\frac{|W|-1}{2}$ arêtes de M (autrement dit, la restriction du couplage M au facteur C fournit un couplage presque parfait de C).

On appelle *base* d'une fleur C par rapport au couplage M l'unique sommet de C non couvert par une arête du couplage M.

L'intérêt d'une fleur est qu'il s'agit localement d'un « paquet » de sommets que l'on peut recouvrir presque totalement par un couplage tout en laissant un dernier sommet non couvert qui peut être n'importe lequel d'entre eux. Il est donc tentant de raisonner globalement en contractant une fleur en un seul sommet résultant. C'est ce que propose de faire le résultat suivant.

Définition 2.18. — On appelle *contraction* d'un ensemble de sommet W dans un graphe $G = (V, E)$ le nouveau graphe dont l'ensemble des sommets est $V \setminus W \cup \{w\}$ (où w est un sommet frais) et dont l'ensemble des arêtes contient les arêtes $\{u, v\}$ si $u, v \in V \setminus W$ et $\{u, v\}$ est une arête de E ainsi que les arêtes $\{u, w\}$ si $u \in V \setminus W$ et s'il existe un sommet $v \in W$ tel que $\{u, v\}$ est une arête de E.



Nous observons que, lors de la contraction d'une fleur par rapport à un couplage M dans un graphe $G = (V, E)$, l'ensemble d'arête induit M' par M dans le graphe contracté $G' = (V', E')$ reste un couplage du graphe G'. Dans cette transformation, le nombre de sommets de G non couverts M, à savoir $|V| - 2|M|$, reste identique au nombre de sommets non couverts par M' dans G', à savoir $|V'| - 2|M'|$.

De plus, pour tout couplage M' du graphe contracté G', du fait qu'un fleur est un facteur critique, il est possible de de relever M' en un couplage M tel que $|V|-2|M| = |V'| - 2|M'|$.

Proposition 2.19. — *Soit G un graphe, M un couplage de G et $C = (W, F)$ une fleur par rapport à M. Notons G' le graphe obtenu en contractant W en un unique sommet et M' le couplage induit. Alors M est un* couplage maximum *de G si et seulement si M' est un couplage maximum de G'.*

Démonstration. — Cette proposition découle de l'observation vue plus haut. □

2.3.3. Forêts alternées et algorithme d'Edmonds. — Nous décrivons de manière informelle l'*algorithme du couplage maximum d'Edmonds*. Les détails de l'implémentation seront vus en TP.

Comme dans le cas biparti, nous recherchons un chemin augmentant afin d'appliquer le lemme de Berge (cf. théorème 2.12). Néanmoins, cette recherche pouvant échouer et détecter à la place une fleur, nous envisageons de contracter toute fleur rencontrée. Comme dans le cas biparti, la construction d'un chemin augmentant nous amène à manipuler implicitement des forêts indiquant des chemins M-alternants (même si, dans le cas biparti, les forêts sont en général cachées derrière le fait qu'on effectue des parcours de graphes et qu'il y a implicitement des arborescences de parcours).

Définition 2.20. — Une *forêt alternée par rapport à un couplage M* dans un graphe $G = (V, E)$ est une forêt (W, F), avec $W \subseteq V$ et $F \subseteq E$, telle que

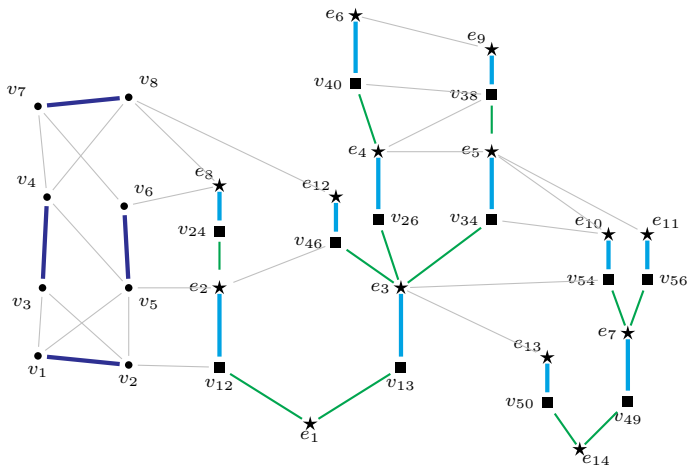
1. tout sommet de V non couvert par M est dans W
2. chaque composante connexe contient exactement un sommet non couvert par M, que l'on fixe comme racine
3. les sommets à distance impaires de la racine sont de degrés 2 dans (W, F).
4. tout chaîne élémentaire dans (W, F) issue d'une racine est une chaîne M-alternée.

Définition 2.21. — Une *forêt alternée par rapport à un couplage M* dans un graphe $G = (V, E)$ étant fixée, nous disons qu'un sommet $v \in V$

1. *hors de la forêt* s'il appartient à $V \setminus W$
2. *externe* s'il appartient à W et s'il se trouve à distance paire de la racine de l'arbre le contenant.,
3. *interne* s'il appartient à W et s'il se trouve à distance impaire de la racine de l'arbre le contenant..

Par ailleurs, pour tout sommet $v \in W$, nous notons $P(v)$ l'unique chaîne reliant v à la racine de l'arbre le contenant.

Exemple 2.22. — Dans cet exemple, les sommets hors de la forêt sont notés par un cercle, les sommets externes par une étoile et les sommets internes par un carré. La forêt comporte deux composantes connexes : l'arbre enraciné en e_1 et l'arbre enraciné en e_{14}.



Exemple 2.23. — Un couplage M d'un graphe $G = (V, E)$ étant fixé, il est facile de construire une *forêt M-alternée triviale* $(W, \emptyset)$ en considérant l'ensemble W des sommets de V non couverts par M. Dans cette forêt, tous les arbres sont réduits à un seul sommet.

Dans ce qui suit, nous décrivons une procédure qui, à partir d'un certain couplage M et d'une certaine forêt M-alternée (W, F) conduit à l'un des cas suivants

— déclarer le couplage M maximal,

— transformer le couplage M en un couplage possédant une arête de plus et remplacer la forêt (W, F) par la forêt triviale (cf. exemple 2.23),

— transformer la (W, F) en une forêt plus grande,

— contracter une fleur dans le graphe.

Voici les détails. Fixons un sommet externe x et identifions un voisin y de x dans G. Nous distinguons trois cas :

— [Grossir la forêt] Cas où le sommet y est hors de l'arbre. Notons que y est forcément couvert par M par une arête qui n'est pas $\{x, y\}$. Nous faisons grossier la forêt en ajoutant l'arête $\{x, y\}$ et l'arête du couplage M couvrant y.

— [Augmenter le couplage] Cas où le sommet y est un sommet externe situé dans une composante connexe de (W, F) différente de celle de x. Le chaine $C = P(x) \cup \{x, y\} \cup P(y)$ est une chaîne M augmentante. Nous remplaçons M par $M \triangle C$ et ré-initialisons la forêt à la forêt triviale.

— [Contracter une fleur] Cas où le sommet y est un sommet externe situé dans la même composante connexe de (W, F) que celle de x. Soit r le premier sommet de la chaîne $P(x)$ qui appartient aussi à la chaîne $P(y)$. Soit r est la racine et est externe, soit r n'est pas la racine, mais alors r est de degré 3 dans (W, F) et donc est un sommet externe. Donc $C = P(x)_{|[x,r]} \cup \{x, y\} \cup P(y)_{|[y,r]}$ est un cycle de longueur impaire : c'est un facteur critique. Nous décidons de le contracter.

En épuisant les trois cas pour tout choix de x et de y, nous pouvons nous amener à la situation où tout voisin y de tout sommet externe x est un sommet interne.

Lemme 2.24. — *Soit G un graphe, M un couplage et (W, F) une forêt M-alternée dans laquelle les voisins de tout sommet externe sont des sommets internes. Alors le couplage M est un couplage maximum de G.*

Démonstration. — Notons $X \subseteq W$ l'ensemble des sommets internes. Notons s le nombre de sommets internes et t le nombre de sommets externes. Nous pouvons observer que le nombre de sommets de V non couverts par M est exactement $t - s$.

Dans le graphe $G \setminus X$, tout sommet externe se retrouve déconnecté des autres sommets. Ainsi, les composantes connexes de $G \setminus X$ sont soit des sommets externes isolés, soit des grappes de sommets de $V \setminus W$. Les sommets hors de l'arbres sont deux à deux couverts par une arête de M, donc ils formes des composantes connexes de cardinal pair. Les seules composantes connexes de $G \setminus X$ de cardinal impair sont les sommets externes isolés : ainsi $q_G(X) = t$.

D'après la formule de Tutte-Berge (cf. lemme 2.7), un couplage maximum laisse un nombre de sommets découverts $|V| - 2\nu(G)$ qui est au moins $q_G(X) - |X|$, c'est-à-dire ici $t - s$ sommets. Cette inégalité est atteinte ici, ce qui prouve l'optimalité du cardinal du couplage M. □

Théorème 2.25. — *L'algorithme d'Edmonds se termine et construit un couplage de cardinal maximum.*

Démonstration. — Il est facile de se convaincre que l'application répétée de la procédure décrite ci-dessus se termine : on peut considérer la quantité $(|V| - 2|M|, |V| - |W|, |V|)$ comme variant pour l'établir. Par ailleurs, le lemme 2.24 confirme la correction de l'algorithme. □

Exemple 2.26. —

TRAVAUX PRATIQUES : IMPLÉMENTATION DE L'ALGORITHME DU COUPLAGE MAXIMUM D'EDMONDS

Objectif. — Le but de ce TP est d'implémenter l'algorithme d'Edmonds de construction d'un *couplage de cardinal maximum dans un graphe non orienté*. Le TP a été écrit en s'appuyant sur [**KV12**] qui renvoie à [**LP09**] pour les détails d'implémentation. N'hésitez pas à remonter à ces deux sources en cas de suspicion d'erreur dans l'énoncé de ce TP.

Langage de programmation. — Le TP est proposé en Python. Vous êtes libre de l'adapter à un autre langage de programmation. Si vous êtes mal à l'aise avec les rudiments de programmation orientée objet en Python, vous pouvez consulter les pages consacrées dans un manuel scolaire de lycée (par exemple chapitre 2 de [**CGG$^+$22**]).

Conseils. — Faites beaucoup de tests au fur et à mesure. N'hésitez pas à coder des fonctions qui vérifient les invariants attendus pour vous assurer de la justesse de votre code.

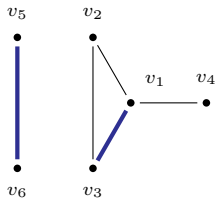
2.4. Mise en place de fonctions utilitaires

Nous présentons tout couplage M dans un graphe $G = (V, E)$ par une involution, c'est-à-dire une application $\mu : V \to V$ telle que

$$\mu \circ \mu = \mathrm{Id}_V \quad \text{et} \qquad \forall v \in V,\ \mu(v) = v \text{ ou } \{v, \mu(v)\} \in E. \tag{$\clubsuit_1$}$$

Nous obtenons le couplage M comme l'ensemble des arêtes $\{\{v, \mu(v)\}\,;\ v \in V \text{ t.q. } v \neq \mu(v)\}$.

Exemple 2.27. — Le graphe Γ_1 suivant est muni d'un couplage formé de deux arêtes.



$$\begin{aligned}\mu(v_1) &= v_3\\ \mu(v_2) &= v_2\\ \mu(v_3) &= v_1\\ \mu(v_4) &= v_4\\ \mu(v_5) &= v_6\\ \mu(v_6) &= v_5\end{aligned}$$

Question 2.1. — Initialiser un fichier contenant le code suivant. Un fichier squelette est disponible sur le site du cours E-Campus pour vous aider. Il pourra être prudent de prévoir, dès à présent, une fonction vérifiant l'invariant ($\clubsuit_1$) que l'on appellera à chaque fois que l'attribut μ est modifiée.

```
1  class Graph():
2      def __init__(self, vertices = [], edges = []) -> None:
3          self.V = []
4          self.E = {}
5          self.mu = {}
6          for v in vertices:
7              self.add_vertex(v)
8          for (u,v) in edges:
9              self.add_edge(u,v)
10
11     def add_vertex(self, v):
12         assert(v not in self.V)
13         self.V.append(v)
14         self.E[v] = []
15         self.mu[v] = v
16
17     def add_edge(self, u, v):
18         assert(u in self.V)
19         assert(v in self.V)
20         self.E[u].append(v)
21         self.E[v].append(u)
```

La version de l'algorithme d'Edmonds présentée en cours s'appuie sur des opérations contractions et amènent à manipuler des graphes dont les sommets sont en réalité des grappes de sommets entre lesquels il faut se souvenir de certains couplages déjà construits. La présentation suivante autorise une forêt à comporter des fleurs [(1)].

Définitions 2.28. — Une *forêt générale de fleurs* par rapport à un couplage M dans un graphe non orienté $G = (V, E)$ est un sous-graphe (W, F), avec $W \subseteq V$ et $F \subseteq E$, muni d'une partition de W en $W_1, \ldots, W_k$ telle que, pour tout indice i,

1. chaque graphe induit par W_i est une fleur par rapport à M (*cf* définition 2.17)

1. En toute rigueur, il ne s'agit plus d'une forêt.

2. et en contractant chaque W_i, on obtient une forêt alternée F' (*cf* définition 2.20).

Nous parlons de *fleur interne*, ou respectivement de *fleur externe*, selon que sa contraction est un sommet interne, ou respectivement externe, dans la forêt alternée F'. Les sommets de W appartenant à une *fleur interne*, respectivement à une *fleur externe*, sont dit internes, respectivement externes. Un sommet v de $V \setminus W$ est dit *hors de la forêt*. Dans ce qui suit, nous n'aurons besoin que de fleurs externes : les fleurs internes ne comporteront qu'un seul sommet. Nous parlons alors de *forêt spéciale de fleurs*.

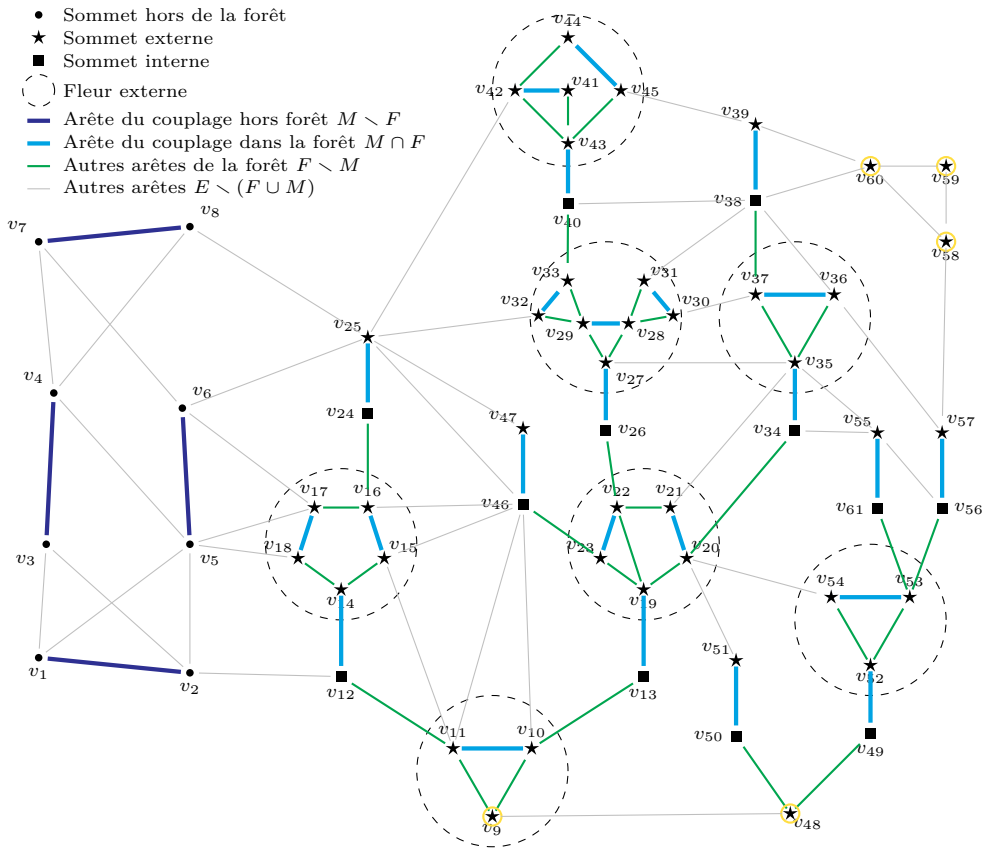


FIGURE 2.2. Graphe Γ_2 : une forêt spéciale de fleurs, disponible sur E-Campus

Nous gardons en mémoire une référence la base (*cf.* définition 2.17) d'une fleur à l'aide d'une application $\rho : V \to V$ telle que :

$$\rho(x) = \begin{cases} x & \text{si } x \text{ n'est pas un sommet externe} \\ y & \text{si } x \text{ est un sommet externe et } y \text{ est} \\ & \text{la base de la fleur externe contenant } x \end{cases} \qquad (\clubsuit_2)$$

Question 2.2. — Modifier la classe `Graph` afin que chaque instance dispose d'un attribut `rho`, de type dictionnaire $V \to V$. [Indication : initialement, pour tout sommet $v \in V$, on souhaite avoir $\rho(v) = v$.]

Définition 2.29. — On dit qu'une fleur par rapport au couplage M admet une *décomposition en oreilles M-alternée* si en notant r la base de la fleur, il existe une suite P_1, P_2, …, P_k de chaînes M-alternante de longueur impaire telle que, pour tout indice i compris entre 1 et k, seules les extrémités de P_i appartiennent à $\{r\} \cup P_1 \cup P_2 \cup \cdots \cup P_{i-1}$ et telle que la fleur est la réunion $\{r\} \cup P_1 \cup \cdots \cup P_k$.

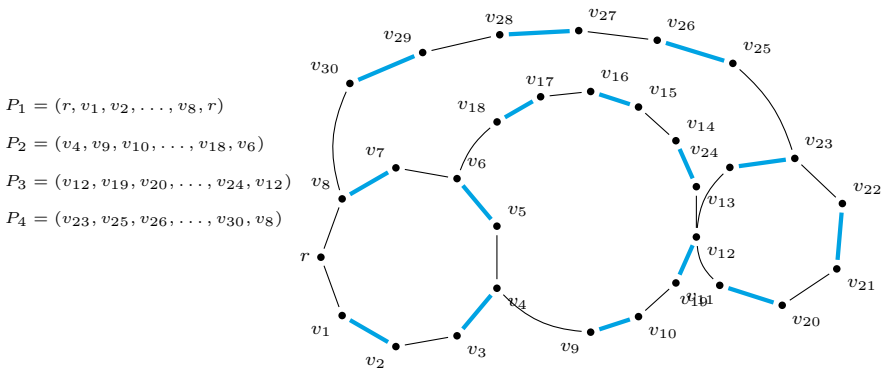


FIGURE 2.3. Graphe Γ_3 admettant une décomposition en oreilles M-alternées.

Dans ce qui suit, nous ne rencontrerons que des fleurs admettant une décomposition en oreilles M-alternée [2] et nous ne les manipulerons qu'à travers cette décomposition. Pour tout sommet v appartenant à une oreille P_i distinct des extrémités de la chaîne P_i, l'un des voisins de v dans P_i est $\mu(v)$, nous notons $\phi(v)$ l'autre voisin. Nous posons encore $\phi(r)$. Il est clair que la connaissance de μ et ϕ suffit pour retrouver la décomposition en oreilles.

2. En vérité, nous pourrions démontrer que cette décomposition existe toujours pour n'importe quelle fleur.

Exemple 2.30. — Dans le graphe Γ_3 de la figure 2.3, les valeurs des fonctions $\mu : V \to V$ et de $\phi : V \to V$ sont données par

$\mu(r) = r$	$\phi(r) = r$	$\mu(v_1) = v_2$	$\phi(v_1) = r$	$\mu(v_2) = v_1$	$\phi(v_2) = v_3$
$\mu(v_3) = v_4$	$\phi(v_3) = v_2$	$\mu(v_4) = v_3$	$\phi(v_4) = v_5$	$\mu(v_5) = v_6$	$\phi(v_5) = v_4$
$\mu(v_6) = v_5$	$\phi(v_6) = v_7$	$\mu(v_7) = v_8$	$\phi(v_7) = v_6$	$\mu(v_8) = v_7$	$\phi(v_8) = r$
$\mu(v_9) = v_{10}$	$\phi(v_9) = v_4$	$\mu(v_{10}) = v_9$	$\phi(v_{10}) = v_{11}$	$\mu(v_{11}) = v_{12}$	$\phi(v_{11}) = v_{10}$
$\mu(v_{12}) = v_{11}$	$\phi(v_{12}) = v_{13}$	$\mu(v_{13}) = v_{14}$	$\phi(v_{13}) = v_{12}$	$\mu(v_{14}) = v_{13}$	$\phi(v_{14}) = v_{15}$
$\mu(v_{15}) = v_{16}$	$\phi(v_{15}) = v_{14}$	$\mu(v_{16}) = v_{15}$	$\phi(v_{16}) = v_{17}$	$\mu(v_{17}) = v_{18}$	$\phi(v_{17}) = v_{16}$
$\mu(v_{18}) = v_{17}$	$\phi(v_{18}) = v_6$	$\mu(v_{19}) = v_{20}$	$\phi(v_{19}) = v_{12}$	$\mu(v_{20}) = v_{19}$	$\phi(v_{20}) = v_{21}$
$\mu(v_{21}) = v_{22}$	$\phi(v_{21}) = v_{20}$	$\mu(v_{22}) = v_{21}$	$\phi(v_{22}) = v_{23}$	$\mu(v_{23}) = v_{24}$	$\phi(v_{23}) = v_{22}$
$\mu(v_{24}) = v_{23}$	$\phi(v_{24}) = v_{12}$	$\mu(v_{25}) = v_{26}$	$\phi(v_{25}) = v_{23}$	$\mu(v_{26}) = v_{25}$	$\phi(v_{26}) = v_{27}$
$\mu(v_{27}) = v_{28}$	$\phi(v_{27}) = v_{26}$	$\mu(v_{28}) = v_{27}$	$\phi(v_{28}) = v_{29}$	$\mu(v_{29}) = v_{30}$	$\phi(v_{29}) = v_{28}$
$\mu(v_{30}) = v_{29}$	$\phi(v_{30}) = v_8$				

En toute généralité, nous travaillerons avec une fonction $\phi : V \to V$ satisfaisant

$$\phi(x) = \begin{cases} x & \text{si } x \in V \setminus W \text{ ou si } x \text{ est la base d'une fleur externe} \\ y & \text{où } \{x, y\} \text{ est l'unique arête de } W \setminus M \\ & \text{si } x \text{ est un sommet interne} \\ y & \text{où } \{x, y\} \text{ est une arête de } W \setminus M \text{ tel que} \\ & \mu \text{ et } \phi \text{ forment une décomposition en oreille } M\text{-alternée} \\ & \text{si } x \text{ est un sommet externe} \end{cases} \qquad (\clubsuit_3)$$

Question 2.3. — Modifier la classe `Graph` afin que chaque instance dispose d'un attribut `phi`, de type dictionnaire $V \to V$. [Indication : initialement, pour tout sommet $v \in V$, on souhaite avoir $\phi(v) = v$.]

Proposition 2.31. — *Soit $G = (V, E)$ un graphe, M un couplage, (W, F) une forêt spéciale de fleurs par rapport à M. Un sommet $v \in V$ est*

- *externe si et seulement si $\mu(v) = v$ ou $\phi(\mu(v)) \neq \mu(v)$;*
- *interne si et seulement si $\phi(\mu(v)) = \mu(v)$ et $\phi(v) \neq v$;*
- *hors de la forêt si et seulement si $\mu(v) \neq v$ et $\phi(v) = v$ et $\phi(\mu(v)) = \mu(v)$.*

Question 2.4. — À l'aide de la proposition 2.31, enrichir la classe `Graph` d'une méthode `is_outer(self, v)` dont la valeur de retour est `True` si et seulement si le sommet $v \in V$ est externe. On pourra écrire des tests basés sur le graphe Γ_2, également repris dans l'un des fichiers disponibles sur E-Campus, ainsi que des tests basés sur des graphes que l'on inventera.

Question 2.5. — À l'aide de la proposition 2.31, enrichir la classe `Graph` d'une méthode `is_inner(self, v)` dont la valeur de retour est `True` si et seulement si le sommet $v \in V$ est interne.

Question 2.6. — À l'aide de la proposition 2.31, enrichir la classe `Graph` d'une méthode `is_outside(self, v)` dont la valeur de retour est `True` si et seulement si le sommet $v \in V$ est hors de la forêt.

Pour tout sommet externe v, nous définissons le chemin $P(v)$ comme la chaîne obtenue en appliquant alternativement les fonctions μ et ϕ jusqu'à tomber sur un point fixe.

$$v,\ \mu(v),\ \phi(\mu(v)),\ \mu(\phi(\mu(v))),\ \phi(\mu(\phi(\mu(v)))),\ \ldots$$

Proposition 2.32. — *Soit $G = (V, E)$ un graphe, M un couplage, (W, F) une forêt spéciale de fleurs par rapport à M et v un sommet externe. Notons q la racine de l'arbre de la forêt spéciale contenant v. Alors la chaîne $P(v)$ est une chaîne M-alternée allant de v à q.*

Question 2.7. — Enrichir la classe `Graph` d'une méthode `path(self, v)` dont la valeur de retour est la chaîne $P(v)$, présentée sous forme de liste de sommets. Nous recommandons de lever une exception [3] lorsque la précondition « le sommet v est externe » n'est pas remplie. On pourra écrire des tests basés sur les graphes Γ_2 et Γ_3, repris dans les fichiers disponibles sur E-Campus, ainsi que des tests basés sur des graphes que l'on inventera.

Question 2.8. — Enrichir la classe `Graph` d'une méthode `_are_disjoint(self, u, v)` dont la valeur de retour est un booléen indiquant si les sommets externes u et v appartiennent à deux composantes connexes différentes de la forêt spéciale de fleurs. Nous recommandons de lever une exception `ValueError` lorsque la précondition « les sommets u et v sont externes » n'est pas remplie.

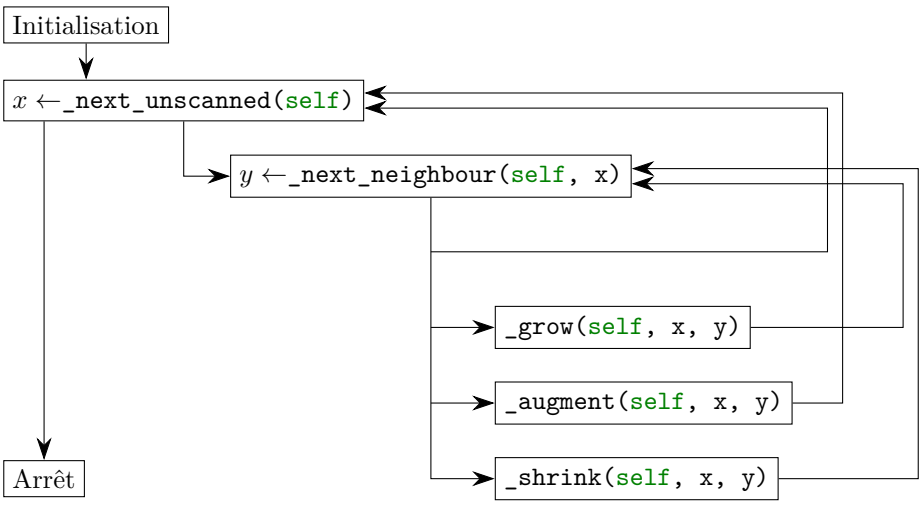
2.5. Déroulement de l'algorithme

L'exécution de l'algorithme d'Edmonds nécessite de se souvenir quels sommets ont déjà été étudiés.

Question 2.9. — Modifier la classe `Graph` afin que chaque instance dispose d'un attribut `scan`, de type dictionnaire $V \to \{\mathsf{Faux}, \mathsf{Vrai}\}$. Initialement, pour tout sommet $v \in V$, on souhaite que `G.scan[v]` égale `False`.

Nous décomposons une étape de l'algorithme d'Edmonds en cinq phases correspondant aux cinq méthodes suivantes : `_next_unscanned(self)`, `_next_neighbour(self, x)`, `_grow(self, x, y)`, `_augment(self, x, y)`, `_shrink(self, x, y)`, qui seront écrite dans les question 2.10 à 2.14. Les méthodes `_next_unscanned(self)` et respectivement `_next_neighbour(self, x)` ont un sommet comme valeur de retour, que l'on conviendra de noter x et respectivement y ; les trois dernières méthodes travaillent par effet en modifiant en place les attributs μ, ϕ, ρ etc et n'ont pas de valeur de retour. Le flot de contrôle (organisé à la question 2.15) est passé d'une méthode à une autre selon le schéma suivant :

3. Par exemple avec la syntaxe `raise ValueError(f"Vertex {v} should be outer")`.



Question 2.10. — Enrichir la classe `Graph` d'une méthode `_next_unscanned(self)` dont la valeur de retour est, si possible, un sommet externe x non balayé (i.e. tel que `self.scan[x]` égale `False`) et le booléen `False` sinon.

L'algorithme d'Edmonds se poursuit par l'appel `_next_neighbour(self, x)` si le sommet x existe et s'arête sinon.

Question 2.11. — Enrichir la classe `Graph` d'une méthode `_next_neighbour(self, x)` dont la valeur de retour est,

- si possible, un voisin y du sommet x tel que y soit hors de la forêt ou (y est un sommet externe avec $\rho(y) \neq \rho(x)$)
- et qui est, sinon, le booléen `False`.

L'algorithme d'Edmonds se poursuit comme suit :

- Si le sommet y n'existe pas, l'attribut `self.scan[x]` passe à `True` et `_next_unscanned` est appelé à nouveau.
- Sinon si le sommet y est hors de la forêt, `self._grow(x, y)` est appelé.
- Sinon si le sommet y est un sommet externe et $P(x)$ et $P(y)$ sont sommet-disjoints, `self._augment(x, y)` est appelé.
- Sinon, `self._shrink(x, y)` est appelé.

Question 2.12. — Enrichir la classe `Graph` d'une méthode `_grow(self, x, y)` qui effectue $\phi(y) \leftarrow x$

Après un grossissement de la forêt spéciale de fleur, l'algorithme d'Edmonds se poursuit en appellant `self._next_neighbour(x)`.

Question 2.13. — Enrichir la classe `Graph` d'une méthode `_augment(self, x, y)` qui implémente les instructions suivantes :

Algorithme 2.1 : Augmentation

Entrées : Sommets externes x et y voisins dans G et issus de deux arbres distincts de la forêt (W, F)

pour *v sommet de la chaîne $P(x)$ à distance impaire de x sur $P(x)$ ou sommet de la chaîne $P(y)$ à distance impaire de y sur $P(y)$* **faire**
  $\mu(\phi(v)) \leftarrow v$
  $\mu(v) \leftarrow \phi(v)$
$\mu(x) \leftarrow y$
$\mu(y) \leftarrow x$
pour $v \in V$ **faire**
  $\phi(v) \leftarrow v$
  $\rho(v) \leftarrow v$
  $\mathsf{scan}(v) \leftarrow \mathsf{Faux}$

Après une augmentation du couplage, l'algorithme d'Edmonds se poursuit en appellant `self._next_unscanned()`.

Question 2.14. — Enrichir la classe `Graph` d'une méthode `_shrink(self, x, y)` qui implémente les instructions suivantes

Algorithme 2.2 : Contraction

Entrées : Sommets externes x et y voisins dans G et issus du même arbre de la forêt (W, F)

$r \leftarrow$ premier sommet commun z à $P(x)$ et $P(y)$ en partant de x ou y tel que $\rho(z) = z$.
pour *v sommet de $P(x)$ compris entre r et x et à distance impaire de x ou sommet de $P(y)$ compris entre r et y et à distance impaire de y tel que $\rho(\phi(v)) \neq r$* **faire**
  $\phi(\phi(v)) \leftarrow v$
si $\rho(x) \neq r$ **alors**
  $\phi(x) \leftarrow y$
si $\rho(y) \neq r$ **alors**
  $\phi(y) \leftarrow x$
pour *$v \in V$ tel que $\rho(v)$ est un sommet compris entre r et x sur la chaîne $P(x)$ ou est un sommet compris entre r et y sur la chaîne $P(y)$* **faire**
  $\rho(v) \leftarrow r$

Après une contraction d'une fleur, l'algorithme d'Edmonds se poursuit en appellant `self._next_neighbour(x)`.

Question 2.15. — Enrichir la classe `Graph` d'une méthode `maximum_matching(self)` dont l'effet est de calculer un couplage de cardinal maximum en orchestrant les appels aux méthodes des question 2.10 à 2.14.

2.6. Pour aller plus loin

Nous terminons ce TP avec quelques résultats de correction et de complexité.

Lemme 2.33. — *À chaque étape de l'exécution de l'algorithme d'Edmonds ainsi présenté,*

1. *L'invariant* ($\clubsuit_1$) *est vérifié : autrement dit, l'application* μ *définit un couplage* M*.*
2. *Les arêtes* $\{v, \mu(v)\}$ *et* $\{v, \phi(v)\}$ *forment une forêt spéciale de fleurs* F *par rapport à* M *avec, en plus, des arêtes isolées du couplage.*
3. *Les invariants* ($\clubsuit_2$) *et* ($\clubsuit_3$) *sont vérifiés par rapport à* F *et* M*.*

Théorème 2.34. — *L'algorithme du couplage maximum d'Edmonds répond correctement en un temps* $O(n^3)$ *où* n *est le nombre de sommets de* G*.*

Exercice 2.35. — Démonter la proposition 2.31.

Exercice 2.36. — Démonter la proposition 2.32.

Exercice 2.37. — Démonter le lemme 2.33.

Exercice 2.38. — Démonter le théorème 2.34.

2.7. Cas de test

Dans l'exemple de la figure 2.2, la fonction ρ relative au graphe Γ_2 correspond à

$$\begin{array}{ccccc}
\mu(v_1)=v_2 & \mu(v_2)=v_1 & \mu(v_3)=v_4 & \mu(v_4)=v_3 & \mu(v_5)=v_6 \\
\mu(v_6)=v_5 & \mu(v_7)=v_8 & \mu(v_8)=v_7 & \mu(v_9)=v_9 & \mu(v_{10})=v_{11} \\
\mu(v_{11})=v_{10} & \mu(v_{12})=v_{14} & \mu(v_{13})=v_{19} & \mu(v_{14})=v_{12} & \mu(v_{15})=v_{16} \\
\mu(v_{16})=v_{15} & \mu(v_{17})=v_{18} & \mu(v_{18})=v_{17} & \mu(v_{19})=v_{13} & \mu(v_{20})=v_{21} \\
\mu(v_{21})=v_{20} & \mu(v_{22})=v_{23} & \mu(v_{23})=v_{22} & \mu(v_{24})=v_{25} & \mu(v_{25})=v_{24} \\
\mu(v_{26})=v_{27} & \mu(v_{27})=v_{26} & \mu(v_{28})=v_{29} & \mu(v_{29})=v_{28} & \mu(v_{30})=v_{31} \\
\mu(v_{31})=v_{30} & \mu(v_{32})=v_{33} & \mu(v_{33})=v_{32} & \mu(v_{34})=v_{35} & \mu(v_{35})=v_{34} \\
\mu(v_{36})=v_{37} & \mu(v_{37})=v_{36} & \mu(v_{38})=v_{39} & \mu(v_{39})=v_{38} & \mu(v_{40})=v_{43} \\
\mu(v_{41})=v_{42} & \mu(v_{42})=v_{41} & \mu(v_{43})=v_{40} & \mu(v_{44})=v_{45} & \mu(v_{45})=v_{44} \\
\mu(v_{46})=v_{47} & \mu(v_{47})=v_{46} & \mu(v_{48})=v_{48} & \mu(v_{49})=v_{52} & \mu(v_{50})=v_{51} \\
\mu(v_{51})=v_{50} & \mu(v_{52})=v_{49} & \mu(v_{53})=v_{54} & \mu(v_{54})=v_{53} & \mu(v_{55})=v_{61} \\
\mu(v_{56})=v_{57} & \mu(v_{57})=v_{56} & \mu(v_{58})=v_{58} & \mu(v_{59})=v_{59} & \mu(v_{60})=v_{60} \\
\mu(v_{61})=v_{55} &&&&
\end{array}$$

Dans l'exemple de la figure 2.2, la fonction ρ relative au graphe Γ_2 est donnée par

$$\begin{array}{ccccc}
\rho(v_1)=v_1 & \rho(v_2)=v_2 & \rho(v_3)=v_3 & \rho(v_4)=v_4 & \rho(v_5)=v_5 \\
\rho(v_6)=v_6 & \rho(v_7)=v_7 & \rho(v_8)=v_8 & \rho(v_9)=v_9 & \rho(v_{10})=v_9 \\
\rho(v_{11})=v_9 & \rho(v_{12})=v_{12} & \rho(v_{13})=v_{13} & \rho(v_{14})=v_{14} & \rho(v_{15})=v_{14} \\
\rho(v_{16})=v_{14} & \rho(v_{17})=v_{14} & \rho(v_{18})=v_{14} & \rho(v_{19})=v_{19} & \rho(v_{20})=v_{19} \\
\rho(v_{21})=v_{19} & \rho(v_{22})=v_{19} & \rho(v_{23})=v_{19} & \rho(v_{24})=v_{24} & \rho(v_{25})=v_{25} \\
\rho(v_{26})=v_{26} & \rho(v_{27})=v_{27} & \rho(v_{28})=v_{27} & \rho(v_{29})=v_{27} & \rho(v_{30})=v_{27} \\
\rho(v_{31})=v_{27} & \rho(v_{32})=v_{27} & \rho(v_{33})=v_{27} & \rho(v_{34})=v_{34} & \rho(v_{35})=v_{35} \\
\rho(v_{36})=v_{35} & \rho(v_{37})=v_{35} & \rho(v_{38})=v_{38} & \rho(v_{39})=v_{39} & \rho(v_{40})=v_{40} \\
\rho(v_{41})=v_{43} & \rho(v_{42})=v_{43} & \rho(v_{43})=v_{43} & \rho(v_{44})=v_{43} & \rho(v_{45})=v_{43} \\
\rho(v_{46})=v_{46} & \rho(v_{47})=v_{47} & \rho(v_{48})=v_{48} & \rho(v_{49})=v_{49} & \rho(v_{50})=v_{50} \\
\rho(v_{51})=v_{51} & \rho(v_{52})=v_{52} & \rho(v_{53})=v_{52} & \rho(v_{54})=v_{52} & \rho(v_{55})=v_{55} \\
\rho(v_{56})=v_{56} & \rho(v_{57})=v_{57} & \rho(v_{58})=v_{58} & \rho(v_{59})=v_{59} & \rho(v_{60})=v_{60} \\
\rho(v_{61})=v_{61} &&&&
\end{array}$$

Dans l'exemple de la figure 2.2, la fonction ϕ relative au graphe Γ_2 correspond à

$$\begin{array}{ccccc}
\phi(v_1)=v_1 & \phi(v_2)=v_2 & \phi(v_3)=v_3 & \phi(v_4)=v_4 & \phi(v_5)=v_5 \\
\phi(v_6)=v_6 & \phi(v_7)=v_7 & \phi(v_8)=v_8 & \phi(v_9)=v_9 & \phi(v_{10})=v_9 \\
\phi(v_{11})=v_9 & \phi(v_{12})=v_{11} & \phi(v_{13})=v_{10} & \phi(v_{14})=v_{14} & \phi(v_{15})=v_{14} \\
\phi(v_{16})=v_{17} & \phi(v_{17})=v_{16} & \phi(v_{18})=v_{14} & \phi(v_{19})=v_{19} & \phi(v_{20})=v_{19} \\
\phi(v_{21})=v_{22} & \phi(v_{22})=v_{21} & \phi(v_{23})=v_{19} & \phi(v_{24})=v_{16} & \phi(v_{25})=v_{25} \\
\phi(v_{26})=v_{22} & \phi(v_{27})=v_{27} & \phi(v_{28})=v_{27} & \phi(v_{29})=v_{27} & \phi(v_{30})=v_{28} \\
\phi(v_{31})=v_{28} & \phi(v_{32})=v_{29} & \phi(v_{33})=v_{29} & \phi(v_{34})=v_{20} & \phi(v_{35})=v_{35} \\
\phi(v_{36})=v_{35} & \phi(v_{37})=v_{35} & \phi(v_{38})=v_{37} & \phi(v_{39})=v_{39} & \phi(v_{40})=v_{33} \\
\phi(v_{41})=v_{43} & \phi(v_{42})=v_{43} & \phi(v_{43})=v_{43} & \phi(v_{44})=v_{42} & \phi(v_{45})=v_{43} \\
\phi(v_{46})=v_{23} & \phi(v_{47})=v_{47} & \phi(v_{48})=v_{48} & \phi(v_{49})=v_{48} & \phi(v_{50})=v_{48} \\
\phi(v_{51})=v_{51} & \phi(v_{52})=v_{52} & \phi(v_{53})=v_{52} & \phi(v_{54})=v_{52} & \phi(v_{55})=v_{55} \\
\phi(v_{56})=v_{53} & \phi(v_{57})=v_{57} & \phi(v_{58})=v_{58} & \phi(v_{59})=v_{59} & \phi(v_{60})=v_{60} \\
\phi(v_{61})=v_{53} &&&&
\end{array}$$

Dans l'exemple de la figure 2.2, les chemins P dans le graphe Γ_2 sont les suivants :

$$P(v_9) = [v_9]$$
$$P(v_{10}) = [v_{10}, v_{11}, v_9]$$
$$P(v_{11}) = [v_{11}, v_{10}, v_9]$$
$$P(v_{14}) = [v_{14}, v_{12}, v_{11}, v_{10}, v_9]$$
$$P(v_{15}) = [v_{15}, v_{16}, v_{17}, v_{18}, v_{14}, v_{12}, v_{11}, v_{10}, v_9]$$
$$P(v_{16}) = [v_{16}, v_{15}, v_{14}, v_{12}, v_{11}, v_{10}, v_9]$$
$$P(v_{17}) = [v_{17}, v_{18}, v_{14}, v_{12}, v_{11}, v_{10}, v_9]$$
$$P(v_{18}) = [v_{18}, v_{17}, v_{16}, v_{15}, v_{14}, v_{12}, v_{11}, v_{10}, v_9]$$
$$P(v_{19}) = [v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{20}) = [v_{20}, v_{21}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{21}) = [v_{21}, v_{20}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{22}) = [v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{23}) = [v_{23}, v_{22}, v_{21}, v_{20}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{25}) = [v_{25}, v_{24}, v_{16}, v_{15}, v_{14}, v_{12}, v_{11}, v_{10}, v_9]$$
$$P(v_{27}) = [v_{27}, v_{26}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{28}) = [v_{28}, v_{29}, v_{27}, v_{26}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{29}) = [v_{29}, v_{28}, v_{27}, v_{26}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{30}) = [v_{30}, v_{31}, v_{28}, v_{29}, v_{27}, v_{26}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{31}) = [v_{31}, v_{30}, v_{28}, v_{29}, v_{27}, v_{26}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{32}) = [v_{32}, v_{33}, v_{29}, v_{28}, v_{27}, v_{26}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{33}) = [v_{33}, v_{32}, v_{29}, v_{28}, v_{27}, v_{26}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{35}) = [v_{35}, v_{34}, v_{20}, v_{21}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{36}) = [v_{36}, v_{37}, v_{35}, v_{34}, v_{20}, v_{21}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{37}) = [v_{37}, v_{36}, v_{35}, v_{34}, v_{20}, v_{21}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{39}) = [v_{39}, v_{38}, v_{37}, v_{36}, v_{35}, v_{34}, v_{20}, v_{21}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{41}) = [v_{41}, v_{42}, v_{43}, v_{40}, v_{33}, v_{32}, v_{29}, v_{28}, v_{27}, v_{26}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{42}) = [v_{42}, v_{41}, v_{43}, v_{40}, v_{33}, v_{32}, v_{29}, v_{28}, v_{27}, v_{26}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{43}) = [v_{43}, v_{40}, v_{33}, v_{32}, v_{29}, v_{28}, v_{27}, v_{26}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{44}) = [v_{44}, v_{45}, v_{43}, v_{40}, v_{33}, v_{32}, v_{29}, v_{28}, v_{27}, v_{26}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{45}) = [v_{45}, v_{44}, v_{42}, v_{41}, v_{43}, v_{40}, v_{33}, v_{32}, v_{29}, v_{28}, v_{27}, v_{26}, v_{22}, v_{23}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{47}) = [v_{47}, v_{46}, v_{23}, v_{22}, v_{21}, v_{20}, v_{19}, v_{13}, v_{10}, v_{11}, v_9]$$
$$P(v_{48}) = [v_{48}]$$
$$P(v_{51}) = [v_{51}, v_{50}, v_{48}]$$
$$P(v_{52}) = [v_{52}, v_{49}, v_{48}]$$
$$P(v_{53}) = [v_{53}, v_{54}, v_{52}, v_{49}, v_{48}]$$
$$P(v_{54}) = [v_{54}, v_{53}, v_{52}, v_{49}, v_{48}]$$
$$P(v_{55}) = [v_{55}, v_{61}, v_{53}, v_{54}, v_{52}, v_{49}, v_{48}]$$
$$P(v_{57}) = [v_{57}, v_{56}, v_{53}, v_{54}, v_{52}, v_{49}, v_{48}]$$
$$P(v_{58}) = [v_{58}]$$
$$P(v_{59}) = [v_{59}]$$
$$P(v_{60}) = [v_{60}]$$

CHAPITRE 3

STRUCTURES DE DONNÉES PERSISTENTES ET FONCTIONNELLES PURES

Détour par la programmation fonctionnelle

La *programmation fonctionnelle* est un paradigme de programmation dans lequel chaque instruction élémentaire est une fonction et tout programme est une combinaison d'appels de fonctions. Dans la programmation fonctionnelle *pure* on exclut des instructions élémentaires toutes celles qui ont des *effets de bord*, c'est-à-dire qui modifient l'état du programme, comme, par exemple, les tableaux ou les variables non constantes.

Cela rapproche fortement les fonctions des langages fonctionnels des fonctions mathématiques : si on appelle une même fonction plusieurs fois, on obtiendra le même résultat de nouveau.

Bien que la programmation fonctionnelle peut sembler être une contrainte dans l'écriture de programme, elle permet de raisonner sur la programmation de façon plus mathématique. Le paradigme de la programmation fonctionnelle est peu utilisé pour programmer (seul Haskell est un peu populaire parmi les langages fonctionnels purs), mais des idées développées dans le cadre de ces langages ont eu de très diverses applications :

— dans certaines structures de données, comme celles que nous allons voir ;

— dans les systèmes de fichiers (le Copy On Write des systèmes de fichiers modernes n'est qu'une application des techniques que nous allons voir) ;

— dans le calcul parallèle notamment dans les frameworks Big Data (comme Spark) ou Erlang. Cet usage dans le calcul parallèle n'est pas surprenant : si l'on est sûr que des fonctions f et g n'ont aucun effet de bord, alors on peut toujours calculer f et g en parallèle.

Les algorithmes écrits en langages fonctionnels peuvent toujours être traduits dans des langages impératifs mais l'étude des structures de données "purement fonctionnelles" permet, comme nous allons le voir, certains algorithmes capable de "partage" ou de "remonter dans le temps".

Remarque 3.1. — OCaml est un langage fonctionnel mais impur. Il existe en effet des ref, des tableaux, des enregistrements mutables mais le langage reste fait pour facilement éviter ces constructions.

Un exemple fondamental : la liste

Version impérative. — Dans un langage impératif comme C, on peut faire une structure de liste d'entiers et l'on peut insérer après un élément dans une liste de la façon suivante :

```
22 struct liste {
23   int v;
24   struct liste * suivante ;
25 };
26
27 void insere_apres(liste * l, int v ) {
28   struct liste * maille = malloc(sizeof(liste)) ;
29   maille->v = v;
30   maille->suivante = l->suivante ;
31   l->suivante = maille ;
32 }
```

Version fonctionnelle. — Dans un langage fonctionnel (même impur comme OCaml) on s'interdira en général de modifier une liste déjà existante et toute modification sur une liste va plutôt créer une nouvelle liste de la façon suivante :

```
33 let rec insere_au_n_ieme_element n v l =
34   if n = 0
35   then v::l
36   else (List.hd l)::(insere_au_n_ieme_element (n-1) v (List.tl l))
```

Copie gratuite. — Cet exemple permet de capturer un intérêt fondamental des structures de données fonctionnelles : la copie est gratuite. Dans l'exemple C de liste, si on possède deux pointeurs vers la même liste, alors toute modification sur l'une impacte une modification sur l'autre. Dans l'exemple OCaml, si on prend une liste l et qu'on appelle insère dessus, on possède deux listes, la liste l originale et la copie avec l'insertion faite. Aucune modification ne pourra modifier notre version de l. Il est donc totalement inutile d'avoir une opération de copie : un pointeur vers une liste l pointera toujours vers la même liste l.

Noter que la copie et l'original partagent toute la partie qui n'a pas été visitée mais comme cette partie ne peut pas être modifiée ce n'est pas un problème et même au contraire cela permet d'avoir une utilisation mémoire minimale dans le cas où les deux listes sont utilisées.

Calcul de la taille. — Calculer la taille d'une liste prend un temps $O(n)$ car il faut parcourir toute la liste. Dans le cadre des listes C c'est difficilement évitable car une variable peut maintenir un pointeur vers le milieu de la structure et y ajouter des éléments. Dans le cas fonctionnelle on peut toujours rajouter l'information de la taille, comme la liste ne peut pas être modifiée, on est garantis que la taille est correcte. Ce n'est pas en général ce que l'on fait car cela coûte un entier par maille de la liste mais ça ne change l'occupation mémoire que d'un facteur constant. On peut désormais supposer que, si besoin, la taille d'une liste se calcule en $O(1)$.

Telle que présentée en version fonctionnelle, une liste est en fait exactement la même chose qu'une pile. Nous allons maintenant étudier un cas plus subtil, la file (First-In-First-Out)

3.1. La file

Une file naïve. — Une manière très naïve d'obtenir une file c'est d'utiliser la pile-liste, pour ajouter on empile, et pour supprimer on dépile tout, on renvoie le dernier et on rempile le reste. Ça fonctionne mais ce n'est pas très efficace car on paie $O(n)$ par élément retiré.

```
37 type 'a file_naive = ('a list * 'a list)
38
39 let enfile_naif x file = x::file
40
41 let rec defile_naif file =
42   | [x] -> x, []
43   | a::q ->
44      let der,reste = defile_naif q in
45      (der,a::reste)
46   in
```

La double pile. — On peut améliorer cette méthode en considère que notre file est en fait une pile des objets devant (que l'on a dans l'ordre de sortie) et celle des objets derrière (que l'on a dans l'ordre d'insertion). Si l'on essaie d'insérer dans la file, il suffit d'insérer dans la pile de derrière. Si l'on essaie de sortir le premier élément d'une file non vide il y a deux cas :

- si la pile devant est non vide et il suffit de retirer un élément,
- si la pile de devant est vide on va devoir prendre tous les objets de la pile de derrière et les insérer dans la pile de devant ; ça aura bien l'effet de retourner la pile et donc les premiers seront les derniers. On se ramène alors au cas de la pile de vant qui est non vide.

```
47 type 'a file_naive = ('a list * 'a list)
48
49 let enfile_naif x (f_der,f_dev) = (x::f_der, f_dev)
50
```

```
51 let rec defile_naif (f_der, f_dev) = match f_dev with
52   | [] -> defile ([], List.rev f_der)
53   | x::t -> x, (f_der,t)
```

En faisant une analyse pire cas, on remarque que celui-ci reste $O(n)$ car il faut parfois retourner la pile de derrière. Dans le cas où il n'y a qu'une seule pile et n opérations enfiler/défiler on peut calculer qu'il y a un total de $O(n)$ opérations ce qui donne une complexité moyenne est $O(1)$.

Nous avons envie de dire que la complexité moyenne ou amortie est $O(1)$ mais, en programmation fonctionnelle, cela n'est vrai que si l'on n'utilise qu'une copie de la file. En effet, il est possible de créer une file en enfilant n valeurs (sans aucune opération défiler) puis de faire n copies de cette file (on rappelle que la copie est toujours $O(1)$). On peut ensuite faire une opération défiler sur chaque copie ce qui va coûter à chaque fois $O(n)$. Au total, les $3n$ opérations mettront un temps de $O(n^2)$.

Vers une bonne solution : la quadruple pile. — On peut encore améliorer la méthode précédente en utilisant un quadruplet de pile. En effet considérons l'algorithme qui retourne la pile de derrière pour la mettre devant. Cette algorithme peut être vu comme un algorithme de pile : on part d'une pile P_1 à retourner et mettre dans P_2, à chaque étape on prend le haut de P_1 et on l'empile dans P_2.

L'idée générale de notre algorithme quadruple pile est d'avoir 4 pilesă: il y aura toujours celle de devant, et celle de derrière mais il y aura aussi deux piles pour les retournements "en cours". On appelle *opération de transfert* l'action de prendre un élément de P_1 si c'est non vide et de le mettre sur P_2

À chaque enfilement on insère dans la pile de derrière et l'on fait 3 opérations de transfert. Pour chaque opération défilement, si la pile de devant est non vide, on retire de la pile de devant puis on fait 3 opérations de transfert. Si la pile de devant est vide, on peut finir le retournement en cours puis mettre devant égal à P_2, P_2 égal à vide, P_1 égal à derrière et derrière devient la pile vide.

Cet algorithme est correct mais il n'est pas $O(1)$ car il est possible que le retournement en cours ne soit pas fini. Pour que cela arrive il faut que l'on commence un retournement avec K éléments alors que le retournement précédent avait $K' < K/3$ éléments, ce qui signifie que tant que le nombre d'éléments dans derrière est inférieur à trois fois la taille du retournements il n'y a pas de problème.

```
54 type 'a file_4 = ('a list * 'a list * 'a list * 'a list)
55
56 let transfert (p1, p2, p3, p4) = match p2 with
57   | [] -> (p1,p2,p3,p4)
58   | x::t -> (p1, t, x::p3, p4)
59
60 let tri_transfert f = f |> transfert |> transfert |> transfert
61
62 let enfile_4 (der, sas_der, sas_dev, dev) x =
63   (x::der, sas_der, sas_dev, dev) |> tri_transfert
64
```

```
let rec defile_4 (der, sas_der, sas_dev, dev) =
  match dev with
  | x::t -> x, tri_transfert (der, sas_der, sas_dev, t)   (* on a un
  ↪  élément devant *)
  | [] ->
     begin
       match sas_der with
       | [] -> defile_4 ([], der, [], sas_dev)
       | x::t -> defile_4 (der, t, x::sas_dev, [])
     end
```

La file temps-réel. — L'idée de la file temps-réel reprend l'idée de la quadruple file mais dès que le nombre d'éléments dans la pile de derrière atteint le nombre d'éléments dans les piles de retournement on commence à calculer deux objets : l'union de la pile de derrière et de la pile retournée et la pile des éléments enfilés après le début du calcul de l'union. Comme dans le cas de la quadruple file, cette union sera calculée à la vitesse de quelques opérations par enfilement/défilement avec l'objectif de finir en n étapes où n est la taille des deux piles à unir. Deux cas peuvent se produire :

— soit pendant le calcul de l'union on vide la pile de devant, auquel on abandonne le calcul de l'union, à ce moment la pile de derrière contient au plus $2n$ éléments ce qui est plus petit que le nombre d'éléments de la nouvelle pile de devant multiplié par 3 ;

— et sinon pendant le calcul de l'union la pile de devant ne devient jamais vide et donc on finit l'union, on remplace la pile p_2 par le résultat de cette union et on remplace la pile de derrière par la pile des éléments ajoutés après le début du calcul de l'union. Noter que le nouveau P_2 a une taille $2n$ mais le nouveau derrière a une taille inférieure à n (cela poserait problème si le nouveau derrière était déjà trop grand).

```
type 'a union_computation =
  | PasDemaree
  | RetourneDer of 'a list * 'a list  * 'a list
  | RetourneDev of 'a list * 'a list * 'a list
  | EmpileDev of 'a list * 'a list

let une_etape_union = function
  | RetourneDer(x::der, dev, der_ret) ->
  ↪  RetourneDer(dev,dev,x::der_ret)
  | RetourneDer([], dev, der_ret) -> RetourneDev(dev, der_ret, [])
  | RetourneDev(x::dev, der_ret, dev_ret) -> RetourneDev(dev,
  ↪  der_ret, x::dev_ret)
  | RetourneDev([], der_ret, dev_ret) -> EmpileDev(dev_ret, der_ret)
  | EmpileDev(x::t, fini) -> EmpileDev(t, x::fini)
```

```
86    | EmpileDev([], f) -> EmpileDev([], f)
87    | PasDemaree -> PasDemaree
```

Arbres de recherche équilibrés 2-3

On va présenter des arbres de recherche 2-3. Dans ces arbres, les feuilles portent entre 1 et 3 étiquettes et les noeuds internes ont entre 2 et 4 enfants. Les noeuds internes portent aussi des étiquettes avec un nombre d'étiquettes qui est l'arité du noeud moins 1. On va aussi s'assurer que, dans ces arbres, les feuilles différentes feuilles sont à la même profondeur.

On dit qu'un arbre 2-3 est *normal* si tous les noeuds et feuilles ont, au plus, 2 étiquettes. Pour insérer dans arbre normal on commence par descendre dans la feuille où l'on veut insérer, on insère l'étiquette puis lors de la remontée on va normaliser l'arbre. Pour cela, dès que l'on remonte dans un noeud interne n, si un des enfants u de n a trois étiquettes e_1, e_2, e_3 éventuellement associées à quatre sous-arbres a_0, a_1, a_2, a_3. Si c'est le cas on remplace dans la liste des enfants de n, le noeud u par la paire de noeuds v, w qui ont pour étiquettes e_1 et e_3 et pour sous arbres a_0, a_1 (pour v) et a_2, a_3 (pour w).

Remarquer qu'après l'insertion il y a au plus un seul noeud qui est non normal dans l'arbre mais en le réparant avec la description ci-dessus on risque de rendre son parent non normal, qui va rendre son parent non normal, etc. jusqu'à remonter à la racine. Si la racine est non normale, on la scinde en deux de la même façon et ajoute une nouvelle racine qui pointe vers les deux bouts de la scission. Noter que c'est la seule manière de changer la profondeur des feuilles.

Gérer la suppression est un peu plus compliqué mais se gère de façon assez similaire : on cherche le noeud qui porte l'étiquette à supprimer. Si c'est un noeud interne on remplace son étiquette par la plus petite étiquette d'une feuille dans le sous-arbre concerné puis on supprime cette étiquette là (pour qu'elle n'apparaisse pas deux fois). Pour supprimer l'étiquette d'une feuille, on la retire mais on risque alors d'avoir un arbre qui a une feuille sans étiquette. On peut corriger ce problème avec une normalisation étendue qu'on applique au parent : on commence par regarder si un des sous-arbres manque de sous-arbres, dans ce cas on fusionne avec le frère puis on normalise comme dans le cas de l'insertion (car le frère en question a pu devenir trop gros).

Tas binomiaux

Todo, voir wikipédia `https://fr.wikipedia.org/wiki/Tas_binomial` et `https://en.wikipedia.org/wiki/Binomial_heap`

En résumé, pour tout type α qui supporte la comparaison en $O(1)$ on peut faire un tas avec les propriétés suivantes :

— $insert(x, M)$

— $extract\_min(x, M)$

— $meld(M_1, M_2)$

Et qui gère toutes ces opération avec une complexité $O(\log n)$ en pire cas. En regardant précisément on peut même voir et implémenter l'insertion pour que la complexité amortie soit $O(1)$. En effet, de la même manière que si on calcule $x + 1, x + 1 + 1, \ldots$ alors la retenue se propage sur k bits que tous les 2^k fois où l'on fait $+1$, on aura ici que l'on ne fait $O(k)$ opérations que tous les 2^k.

Idée sous jacente au tas binomial penché (Skew Binomial Heap)

Nous allons maintenant présenter une amélioration des tas binomiaux qui permettent de faire une insertion en O(1) mais pour cela nous allons d'abord présenter les nombres binaires penchés (Binary Skew Numbers). Dans ce système de notation de nombre, les chiffres sont 0, 1 et 2 mais le 2 ne peut apparaître qu'une seule fois et qu'en position du chiffre le moins significatif non nul.

Formellement si on a un nombre $b_0 \ldots b_k$ avec b_0 le chiffre le moins significatif sa valeur est $\sum_i b_i \times (2^i - 1)$. Voilà quelques exemples :

Binaire penché	**Décimal**
0	0
1	1
2	2
10	3
11	4
12	5
20	6
100	7
101	8

Dans ce système de nombre on voit que faire l'opération incrémenter (c'est-à-dire faire $+1$) est toujours possible en $O(1)$ si l'on garde un pointeur vers le 2 s'il existe.

Arbres binomiaux penchés

Un arbre binaire penché de rang 0 est un arbre qui contient une unique feuille. Un arbre binaire de rang $r + 1$ peut être construit soit à partir de deux arbres A et B de rang r, soit à partir de deux arbres de rang r plus une valuer v :

— soit on ajoute B aux enfants de A (cas de deux arbres)

— soit en créant un nud n parent de A et B (cas de deux arbres et de v avec v plus petit que les valeurs de A et B)

— soit en ajoutant B et un arbre penché de rang 0 aux enfants de A (cas de deux arbres et v qui n'est pas l'élément minimal).

On peut facilement montrer que tout arbre de rang r contient plus de 2^r valeurs. Pour stocker les enfants d'un arbre binai

CHAPITRE 4

MATROÏDES

De nombreux problèmes d'optimisation combinatoire se présentent sous la forme d'une question de *sélection d'éléments* de poids optimum et sujets à une certaine contrainte, à savoir : « étant donné un ensemble fini E muni d'une fonction de poids $w : E \to \mathbb{R}$, que l'on étend à 2^E par la formule

$$\forall X \subseteq E, \qquad w(X) = \sum_{x \in X} w(x),$$

trouver un sous-ensemble $X \subseteq E$ de poids maximum et qui obéit à certaines contraintes ».

4.1. Structures mathématiques

4.1.1. Systèmes d'indépendance. — Dans la plupart des cas, le jeu de contraintes obéit à la définition suivante.

Définition 4.1. — On appelle *système d'indépendance* tout couple $(E, \mathcal{I})$ où E est un ensemble fini, dit aussi ensemble de base, et $\mathcal{I} \subseteq 2^E$ est une collection de parties de E, dite aussi ensemble des *ensembles indépendants*, telle que

1. l'ensemble vide est indépendant : $\emptyset \in \mathcal{I}$;
2. tout sous-ensemble d'un ensemble indépendant reste indépendant : autrement dit, les conditions $A \subseteq A'$ et $A' \in \mathcal{I}$ impliquent que $A \in \mathcal{I}$ (on parle de proprité d'*hérédité*) ;

Vocabulaire 4.2. — Les éléments de $2^E \setminus \mathcal{I}$ sont appelés les *ensembles dépendants*. Les ensembles indépendants maximaux dans le poset $(2^E, \subseteq)$ sont appelés les *bases*. Les ensembles dépendants minimaux dans le poset $(2^E, \subseteq)$ sont appelés les *circuits*.

Deux problèmes peuvent avoir un intérêt dans ce cadre :

***Problème 4.3* (Indépendant de poids maximum).** — *Étant donné un système d'indépendance $(E, \mathcal{I})$ et une fonction de poids $w : E \to \mathbb{R}$, déterminer un ensemble indépendant I qui maximise la quantité $w(I) = \sum_{e \in I} w(e)$.*

***Exemple 4.4* (Clique de cardinal maximum).** — Trouver un ensemble de *sommets formant une clique* et de cardinal maximum dans un graphe non orienté $G = (V_G, E_G)$ relève du problème 4.3. L'ensemble de base est V_G ; la collection d'indépendants est formée des ensembles des cliques ; la fonction poids est la fonction constante égale à 1.

***Exemple 4.5* (Sac à dos).** — Dans le *problème du sac à dos*, on dispose d'un ensemble d'objet E, d'une taille des objets $t : E \to \mathbb{R}_{>0}$, d'une valeur des objets $v : E \to \mathbb{R}$ et d'un capacité du sac B. L'ensemble de base des E ; la collection des indépendants est l'ensemble

$$\mathcal{I} = \left\{ A \subseteq E; \sum_{e \in A} t(a) \leq B \right\};$$

la fonction poids est la fonction coût.

***Problème 4.6* (Base de poids minimum).** — *Étant donné un système d'indépendance $(E, \mathcal{I})$ et une fonction de poids $w : E \to \mathbb{R}$, déterminer une base B qui maximise la quantité $w(B)$.*

***Exemple 4.7* (Voyageur · euse de commerce).** — Le problème du *voyageur · euse de commerce* dans un graphe non orienté $G = (V_G, E_G)$ relève du problème 4.6. L'ensemble de base est E_G ; la collection d'indépendants est formée des sous-ensembles d'arêtes d'un tour hamiltonien.

En toute généralité, les problèmes 4.3 et 4.6 contiennent des problèmes de la classe **NP** : il est donc délicat de vouloir les étudier directement.

Définitions 4.8. — Soit $(E, \mathcal{I})$ un système d'indépendance. On appelle *rang* l'application $\mathrm{rg} : 2^E \to \mathbb{N}$ donnée par

$$\mathrm{rg} : \begin{cases} 2^E & \to & \mathbb{N} \\ X & \mapsto & \max\{|I|; I \in \mathcal{I}, I \subseteq X\} \end{cases}$$

On appelle *fermeture* [1] de l'ensemble X l'ensemble $\sigma(X)$ donné par

$$\sigma : \begin{cases} 2^E & \to & 2^E \\ X & \mapsto & \{y \in E; \mathrm{rg}(X \cup \{y\}) = \mathrm{rg}(X)\} \end{cases}$$

Nous observons qu'un ensemble indépendant est un ensemble dont le rang égale son cardinal.

1. Lorsque E est un espace vectoriel, nous retombons sur la notion d'espace vectoriel engendré par la famille de vecteurs X.

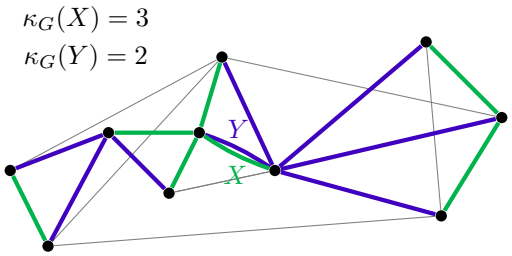
4.1.2. Matroïdes. — Nous ajoutons un troisième axiome, qui rappelle l'essence de la notion d'indépendance, au sens de l'algèbre linéaire, dans le cas d'une famille de vecteurs .

Définition 4.9. — Un système d'indépendance $\mathcal{M} = (E, \mathcal{I})$ est un *matroïde* si, pour tout couple d'indépendants I et $J \in \mathcal{I}$ tel que $|I| < |J|$, il existe un élément $x \in J \setminus I$ tel que $I + x := I \cup \{x\}$ est encore indépendant (propriété d'augmentation).

Exemple 4.10. —

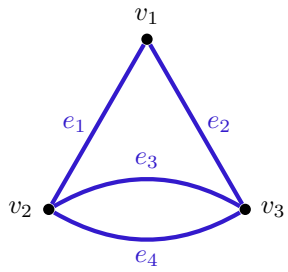
1. Let n and k be integers with $k \leq n$ and $\mathcal{I}$ be the set of subsets of $[n]$ containing at most k elements. Then $\mathcal{U}_{n,k} = ([n], \mathcal{I})$ is a matroid, called the *uniform matroid*.
2. Let E be a set partionned into (disjoint) subets
$$E = E_1 \sqcup E_2 \sqcup \cdots \sqcup E_\ell$$
and $(k_1, \ldots, k_\ell)$ be positive integers. Let
$$\mathcal{I} = \{X \subseteq E;\ \forall 1 \leq i \leq \ell,\ |X \cap E_i| \leq k_i\}$$
Then $(E, \mathcal{I})$ is a matroid called the *partition matroid*. You could see it as a direct sum of uniform matroids.
3. Let A be a matrix over some field with n columns and A_i denote the i-th column vector of A. Let $\mathcal{I}$ be the set of $I \subseteq [n]$ such that the family $\{A_i, i \in I\}$ is linearly independant. Then $([n], \mathcal{I})$ is a matroid, known as the *vector* or *linear matroid* of A.
4. Let $G = (V, E)$ be a graph. Let $I \subseteq E$ belong to $\mathcal{I}$ if I contains no cycle. In other words, $\mathcal{I}$ is the set of *forests* over G. Then $(E, \mathcal{I})$ is a matroid, called the *graphic matroid* of G.

Démonstration. — Augmentation property for graphic matroids : Let X and $Y \in \mathcal{I}$ be two forests with $|X| < |Y|$. Let's call $\kappa_G(X)$ the number of connected components of X on the nodes of G. We have $\kappa_G(X) = |V| - |X|$. So $\kappa_G(X) > \kappa_G(Y)$. This implies that there is an edge e in $Y \setminus X$ that connects two of the connected components of X. Moreover, $X + e$ is still a forest, which concludes the proof.



Check the rest as an exercise. □

Exercice 4.11. — What are all the independent sets of the graphic matroid given by the following graph ?

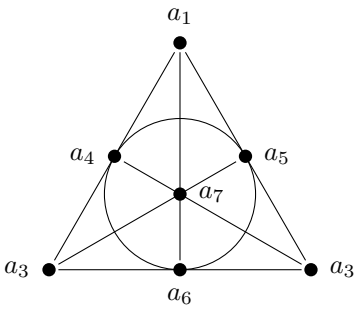


Exercice 4.12. — Show that the uniform matroid $\mathcal{U}_{4,2}$ can be expressed a vector matroid over $\mathbb{F}_3$.

***Exemple 4.13* (Fano matroid).** — Let A be the matrix over $\mathbb{F}_2$

$$\begin{array}{c} a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7 \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

that can be visually seen on the following picture (where the lines represent the independent sets)



***Proposition 4.14* (Matching matroid).** — *Let $G = (V, E)$ be a graph and $\mathcal{I}$ be the collection of the sets of vertices $U \subseteq V$ that can be covered by a matching in G. Then $(V, \mathcal{I})$ is a matroid, called the* matching matroid.

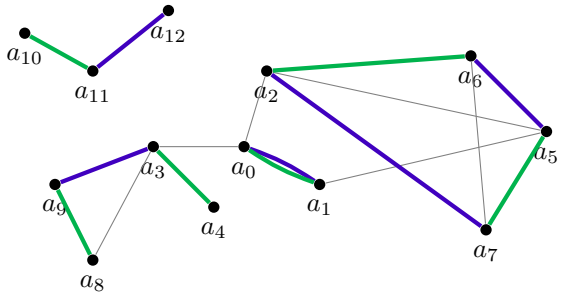
Démonstration. — It is quite clear that the empty set of vertices is independent and that the hereditary property holds.

Let X and $Y \subseteq V$ be to independent sets of vertices. Let $M_X = (V_X, E_X)$ and $M_Y = (V_Y, E_Y)$ be two matchings that contain them. If there exists a vertex v in $Y \setminus X$ that belongs to V_X, then $X + v$ is still covered by M_X. We restrict to the case where any vertex of $Y \setminus X$ is not already covered by M_X :

$$\forall v \in V_X, \quad v \in Y \Rightarrow v \in X. \qquad (*)$$

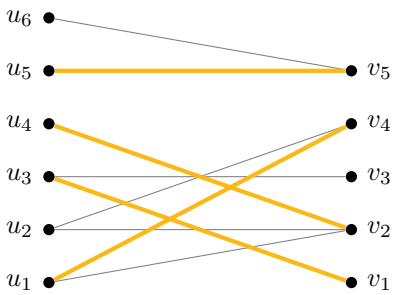
Let us restrict ourselves to the subgraph $\Gamma = (V_X \cup V_Y, E_X \Delta E_Y)$ (where Δ stands for the symmetric difference) and eliminate vertices of degree zero in Γ. Every vertex of Γ has at most one incoming edge from X and one from Y, and they are distinct.

As a consequence, Γ is a reunion of disjoint cycles or paths with edges stemming alternatively from M_X and M_Y.



Since $|Y| > |X|$, there is a connected component of Γ that contains more vertices of Y than X and we can assume WLOG that Γ is connected. (it is thus a even cycle or a path). We select an element $v \in Y \setminus X$. If Γ has an even number of vertices, there is a matching that contains all the vertices of Γ and so $X + v$ is still independant. If Γ has an odd number m of vertices, Γ has to be a path. Because of $(*)$, Y must be exactly $X + v'$ where v' is the only element not covered yet by M_X, so $X + v$ is indeed covered by a matching, namely M_Y. □

Corollaire 4.15 **(Transversal matroid).** — *Let $G = (U \sqcup V, E)$ be a bipartite graph and $\mathcal{I}$ be the collection of set of vertices $X \subseteq U$ that are the endpoints of some matching in G. Then $(V, \mathcal{I})$ is a matroid, called the* transversal matroid.



In the above picture, $\{u_5\}$ or $\{v_6\}$ are independent sets but $\{u_5, u_6\}$ is not independent.

Remarque 4.16. — It is important to note that in the corollary 4.15, the edges themselves do not form a matroid : when we add a new vertex to an independent set, we might need to select a matching with many new edges to prove that it remains independent.

Exercice 4.17. — Let $\mathcal{M} = (E, \mathcal{I})$ be a matroid and k a natural number. Define $\mathcal{I}' = \{X \in \mathcal{I};\ |X| \leq k\}$. Show that $(E, \mathcal{I}')$ is also a matroid, called the *k-truncation* of the matroid $\mathcal{M}$.

Proposition 4.18. — *Soit $\mathcal{M} = (E, \mathcal{I})$ système d'indépendance. Les propriétés suivantes sont indépendantes :*

1. *pour tout couple d'indépendants I et $J \in \mathcal{I}$ tel que $|I| < |J|$, il existe un élément $x \in J \setminus I$ tel que $I + x := I \cup \{x\}$ est encore indépendant ;*
2. *pour tout ensemble $X \subseteq E$, toutes les bases de X sont de même cardinal.*

Démonstration. — Supposons que la propriété d'augmentation est vérifiée dans $\mathcal{M} = (E, \mathcal{I})$. Supposons qu'il existe deux bases B et B' telles que $|B| < |B'|$, alors on pourrait compléter B avec un élément x de B' et $B + x$ serait encore indépendant. Mais alors, B se serait pas maximal et ne serait pas une base. Aussi deux bases ont toujours le même cardinal.

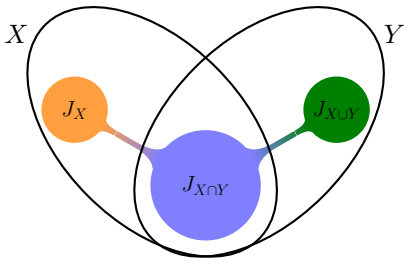
Supposons inversement que deux bases ont toujours le même cardinal et considérons deux indépendants I et J avec $|I| < |J|$. Alors I ne peut pas être une base de $K = I \cup J$. Mais alors un indépendant I' tel que $I \subset I' \subseteq K$. Par hérédité, il existe un élément x de $I' \setminus I$ tel que $I + x$ est indépendant. De plus $x \in J \setminus I$. Nous venons de prouver la propriété d'augmentation. □

Théorème 4.19. — *Soit E un ensemble fini et $r : 2^E \to \mathbb{N}$ une application. Les affirmations suivantes sont équivalentes :*

1. *Le couple $(E, \mathcal{I})$ où*
$$\mathcal{I} = \{I \subseteq E; r(I) = |I|\};$$
est un matroïde dont la fonction rang est l'application r.
2. *Pour tous sous-ensembles $X, Y \subseteq E$, on a*
   (a) *$r(X) \leq |X|$;*
   (b) *si $X \subseteq Y$, alors $r(X) \leq r(Y)$;*
   (c) *$r(X \cup Y) + r(X \cap Y) \leq r(X) + r(Y)$.*

Démonstration. — Supposons que $(E, \mathcal{I})$ est un matroïde dont r est la fonction rang.

1. Les propriétés (2a) et (2b) sont claires.
2. Démontrons la propriété (2c). Commençons par nous donner une base $J_{X\cap Y} \subseteq X \cap Y$ de $X \cap Y$ que l'on étend d'abord avec des éléments $J_X \subseteq X$ en une base de X puis par des éléments $J_{X\cup Y} \subseteq Y$ en une base de$X \cup Y$. Nous avons donc
$$\mathrm{rg}(X \cap Y) = |J_{X\cap Y}|, \quad \mathrm{rg}(X) = |J_{X\cap Y}| + |J_X|$$
$$\mathrm{rg}(X \cup Y) = |J_{X\cap Y}| + |J_X| + |J_{X\cup X}|.$$



Or $J_{X\cap Y} \cup J_{X\cup Y}$ est un indépendant de Y, donc $\mathrm{rg}(Y) \geq |J_{X\cap Y}| + |J_{X\cup Y}|$. En conséquence, nous avons

$$\mathrm{rg}(X) + \mathrm{rg}(Y) \geq \mathrm{rg}(X \cup Y) + \mathrm{rg}(X \cap Y)$$

comme attendu.

Inversement, soit $r : 2^E \to \mathbb{N}$ une application vérifiant les propriétés (2a), (2b) et (2c). Nous introduisons l'ensemble $\mathcal{I} = \{I \subseteq E; r(I) = |I|\}$ et cherchons à démontrer que $(E, \mathcal{I})$ est un matroïde dont r la fonction rang.

1. D'abord, $0 \leq r(\emptyset) \leq |\emptyset| = 0$, ce qui implique que $r(\emptyset) = |\emptyset|$. Donc $\emptyset$ appartient à $\mathcal{I}$.
2. Soit $J \in \mathcal{I}$, autrement dit tel que $r(J) = |J|$, et soit $I \subseteq J$. On a la suite d'inégalité

$$\begin{aligned} |I| + |J \smallsetminus I| &= |J| \\ &= r(J) \\ &= r(I \cup (J \smallsetminus I)) \\ &\leq r(I) + r(J \smallsetminus I) - r(I \cap (J \smallsetminus I)) \quad \text{en utilisant (2c)} \\ &\leq r(I) + |J \smallsetminus I| + 0 \quad \text{en utilisant (2a)} \\ &\leq |I| + |J \smallsetminus I| \quad \text{en utilisant (2a)} \end{aligned}$$

   ce qui prouve que les inégalités sont en vérité toutes des égalités. En particulier, on a $r(I) = |I|$, ce qui montre que I appartient à $\mathcal{I}$.
3. Soient I et J deux éléments de $\mathcal{I}$ tels que $|I| < |J|$. Comme $r(I) = |I|$ et $r(J) = |J|$, on a encore $r(I) < r(J)$. D'après (2b), on a $r(I \cup J) \geq r(J)$, d'où

$$r(I) < r(I \cup J).$$

   Notons $x_1, \ldots x_t$ les éléments de $J \smallsetminus I$. Raisonnons par l'absurde et supposons que, pour tout indice k, l'ensemble $I + x_k$ n'appartient pas à $\mathcal{I}$. On a alors forcément $r(I + x_k) \geq r(I) = |I|$ et $r(I + x_k) < |I + x_k| = |I| + 1$, autrement dit, on a $r(I + x_k) = r(I)$. Démontrons par récurrence sur le cardinal que, pour tout $A \subseteq J \smallsetminus I$,

$$r(I \cup A) \leq r(I)$$

   Le cas où $|A| = 1$ vient d'être vérifié. Dans le cas où $|A| \geq 2$, nous découpons A en $A = A' + x_k$ où A' est un ensemble de cardinal $|A'| = |A| - 1$. On a

$$r(I \cup A) \leq r(I \cup A' + x_k) \leq r(I + A') + r(I + x_k) - r(I) = r(I).$$

   En conséquence, lorsque A vaut $J \smallsetminus I$, on obtient

$$r(I \cup J) = r(I \cup (J \smallsetminus I)) \leq r(I)$$

   Mais alors,

$$r(I) < r(I \cup J) \leq r(I)$$

   ce qui est absurde.

Les deux implications sont prouvées et démontrent l'équivalence entre 1. et 2. □

Exemple 4.20. — 1. In the uniform matroid $\mathcal{U}_{n,k}$, the rank of a set X is $\min(|X|, k)$, bases are subsets with exactly k elements, circuits are subsets with exactly $k+1$ elements.

2. In a linear matroid, the rank and bases are the usual notions of linear algebra.
3. In a graphic matroid over a graph G on n vertices, the rank is $\mathrm{rg}(X) = n - \kappa_G(X)$ where $\kappa_G(X)$ is the number of connected components of X (also counting isolated vertices), a base is a spanning tree and a circuit is a cycle of the graph.

Théorème 4.21. — *Let $\mathcal{M} = (E, \mathcal{I})$ be a matroid, S an independent set and $e \in E$ an element such that $S + e$ is dependent, then $S + e$ contains a unique circuit $C(S, e)$.*

Démonstration. — Suppose $S + e$ contains two distinct circuits C_1 and C_2. Since $C_1 \not\subseteq C_2$, we can consider $f \in C_1 \setminus C_2$. Moreover, by minimality of C_1, $C_1 \setminus f$ is independent. We can extend it to a base X of $S + e$, so $|X| = \mathrm{rg}(S + e) = |S|$ and X is simply $X = S + e \setminus f$. Now, since $C_2 \subseteq S + e$ and $f \notin C_2$, C_2 is included in X and is by consequence independent, which is a contradiction, since C_2 is dependent. □

4.2. Algorithme glouton

Reprenons le problème de la recherche d'une base de poids minimum (voir 4.6) dans le cas où le système d'indépendance est un matroïde.

Du point de vue algorithmique, nous devons nous demander comment représenter un matroïde $\mathcal{M} = (E, \mathcal{I})$. Stocker exhaustivement l'ensemble des indépendants $\mathcal{I}$ n'est pas réaliste : nous nous retrouverions en général à utiliser un espace en $O(2^n)$ pour un ensemble de base E ayant n éléments. Dans tout ce qui suit, nous supposons disposer d'un oracle capable de décider (de préférence en temps polynomial en n) si un sous-ensemble de E est indépendant ou non. Nous notons C la complexité d'un appel.

Algorithme 4.1 : Algorithme glouton du meilleur inséré

Entrées : Matroïde pondéré $\mathcal{M} = (E, \mathcal{I}, w)$
Sorties : Une base $B \in \mathcal{I}$ de poids maximum.

```
1 Trier les éléments E = {e_1, ..., e_n} de sorte que
    w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_n).
2 S ← ∅
3 pour i ∈ ⟦1, n⟧ faire
4     si S + e_i ∈ I alors
5         S ← S + e_i
6 retourner S
```

Remarque 4.22. — 1. Dans l'algorithme 3, on peut remplacer minimum par maximum en renversant l'ordre de tri des éléments.

2. Dans le cas du matroïde graphique, on retrouve exactement l'algorithme de Kruskal qui calcule un arbre couvrant de poids minimum.

Théorème 4.23. — *L'algorithme glouton du meilleur inséré (algorithme 3) calcule une base de poids minimum en temps $O(n \log n + nC)$ où n est le cardinal de l'ensemble de base V et C est la complexité d'un appel à l'oracle.*

Démonstration. — Soit $S = \{s_1, \dots, s_k\}$ la valeur de retour de l'algorithme et $T = \{t_1, \dots, t_k\}$ une base de $\mathcal{M}$ de poids plus léger. Nous avons noté les éléments de sorte que

$$w(s_1) \leq w(s_2) \leq \cdots \leq w(s_k)$$
$$w(t_1) \leq w(t_2) \leq \cdots \leq w(t_k)$$

Puisque $w(S) > w(T)$, nous ne pouvons avoir $w(s_i) \leq w(t_i)$pour tout indice i. Il doit exister un indice ℓ, que nous choisissons minimum, tel que $w(s_\ell) > w(t_\ell)$. À présent, posons $A = \{s_1, \dots, s_{\ell-1}\}$ et $B = \{t_1, \dots, t_\ell\}$ qui sont deux ensembles indépendants en vertu de la propriété d'hérédité. Comme $|B| > |A|$, il existe un élément e de $B \setminus A$ tel que $A + e$ est indépendant. Mais $w(e) \leq w(t_\ell) < w(s_\ell)$ ce qui montre que l'algorithme glouton a sélectionné s_ℓ par erreur à la place de e. Ceci ne peut pas se produire. □

Théorème 4.24 **(Rado 1957, Edmonds 1971).** — *Un système d'indépendance $(E, \mathcal{I})$ est un matroïde si et seulement si l'algorithme glouton du meilleur inséré fournit une solution optimale au problème de la base de poids minimum pour n'importe quelle fonction de poids $w : E \to \mathbb{R}$.*

Démonstration. — The sufficient condition has just been proven.

Suppose that $\mathcal{I}$ does not have hereditary property, *i.e.* there are two sets $X \subset Y$ such that $X \notin \mathcal{I}$ but $Y \in \mathcal{I}$. We assign the weights as follows :

$$\forall x \in E, \qquad \begin{cases} w(x) = 10 & \text{if } x \in X \\ w(x) = 1 & \text{if } x \in Y \setminus X \\ w(x) = 0 & \text{otherwise} \end{cases} .$$

It is quite clear that the maximum weight of an independant set is the weight of Y, namely $m = 9|X| + |Y|$. The greedy algorithm investigates first all the elements of X, then the elements of $Y \setminus X$ and then the remaining one. To be able to match with m, it needs to accept every element from Y when they are presented to him. Now this means that at the $|X|$-th step, the algorithm recognise X as an independent set, which is not the case. So the greedy algorithm is not optimal.

From now on, we assume that $\mathcal{I}$ has the hereditary property. We assume that there are to independant sets I and J with $|I| < |J|$, such that for all $x \in J \setminus I$, $I + x \notin \mathcal{I}$. We set the weights as follows :

$$\forall x \in E, \qquad \begin{cases} w(x) = 1 + \frac{1}{2|I|} & \text{if } x \in I \\ w(x) = 1 & \text{if } x \in J \setminus I \\ w(x) = 0 & \text{otherwise} \end{cases} .$$

Again, the greedy algorithm will first examine the elements of I, keep them (because of the heredity assumption) and then reject all the remaining elements of J. The

weight of the greedy solution is thus

$$|I|\left(1+\frac{1}{2|I|}\right)=|I|+\frac{1}{2}$$

On the other hand, J is an independant set of weight $|J|\geq|I|+1>|I|+\frac{1}{2}$. So the greedy algorithm is not selecting an optimal solution. □

***Exercice 4.25* (On-time task scheduling).** — We are given n tasks $(t_i)_{i\leq n}$ that can be processed in a unit of time and with a potential (positive) benefit $(p_i)_{i\leq n}$ that we earn provided the task t_i is performed by the deadline d_i.

1. Show that if a set of tasks can be completed on time with a certain schedule, they can be scheduled in the order of increasing deadlines.
2. Let $\mathcal{I}$ be the set of all $I\subseteq[n]$ such that all the tasks of I can be completed on time. Show that $([n],\mathcal{I})$ is a matroid.
3. Find an algorithm that computes an optimal ordering of the tasks.

***Exercice 4.26* (Experiment design).** — You're farming herbs on your balcony and you would like understand through experiments the relationship between the yield Y of your production and the different minerals $(X_i)_{i\leq m}$ that you can add to the soil. You assume a linear relationship :

$$Y=p_1x_1+\cdots+p_mx_m$$

where x_i is the added quantity of X_i. Unfortunately, you have access only to different fertilizers $(F_j)_{j\leq n}$, available in the supermarket at the costs of $(c_j)_{j\leq n}$, that contain $a_{i,j}$ units of X_i. Design an algorithm that selects which fertilizers to buy in order to determine all the coefficients $(p_i)_{i\leq m}$ at the least cost.

Exercice 4.27. —

1. Given a rank n matroid $\mathcal{M}=(E,\mathcal{I})$ and a black and white coloring of the elements of E, find an algorithm that computes a base with k black elements and $n-k$ white elements or reports correctly that this is impossible.
2. A communication network is modeled by a connected graph $G=(V,E)$ on n nodes and is owned by two compagnies *Bloom Connected* and *Serena Networks*. The two companies have contracted with your firm *Prodigy Solutions* to upgrade their network from fiber-optic to thunder-eXtra-light cables. Due to the costs, only a minimal number of connections will be upgraded at the moment, but the new network should be still connected. Your business relations people have just signed a contract stipulating that only k connections of *Bloom Connected* and $n-1-k$ connections of *Serena Networks* will be upgraded. *Prodigy Solutions* is clueless about the feasibility of the contract. Since you're the smartest kid of the crew, they ask you to come up with a solution by tomorrow 8am, while they go out to celebrate the new signed contract. What do you suggest ?
3. The firm *Euforins* is doing food analysis for the two major industries *Pindus* and *Ficcard*. Each analysis takes a unit of time to be processed and Euforins has only one machine to do it. The samples $(S_i)_{i\leq n}$ to analyse are usually given

with a mandatory deadline d_i by which the results should be sent back. Due to tabloid allegations that vegetarian ingredients have been found in certain wapiti dishes, the need for analysis has exploded by 700 % and it is no longer possible to do them all. Euforins' sales department has agreed to garantee k analysis for Pindus and $n - k$ analysis for Ficcard. Your significant other has been asked to quickly reorganise the scheduling and calls you at Prodigy Solutions. Can you help him/her ? If you succeed, s/he promises to share some of the left-over samples with you on your next date.

Exercice 4.28. — Describe a greedy removal algorithm for maximum independent sets of a matroid. Prove that it works.

4.3. Intersection de deux matroïdes

4.3.1. Intersection de systèmes d'indépendance. —

Définition 4.29. — Soient deux systèmes d'indépendance $\mathscr{S}_1 = (E, \mathcal{I}_1)$ et $\mathscr{S}_2 = (E, \mathcal{I}_2)$ portant sur le même ensemble de base. On appelle *intersection des systèmes* $\mathscr{S}_1$ *et* $\mathscr{S}_2$ le couple $(E, \mathcal{I}_1 \cap \mathcal{I}_2)$, qui est encore un système d'indépendance.

Plus généralement, nous pouvons définir l'intersection de plusieurs systèmes d'indépendance.

Proposition 4.30. — *Tout système d'indépendance est l'intersection de plusieurs matroïdes.*

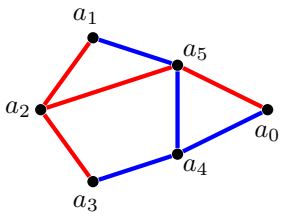
Démonstration. — Soit $(E, \mathcal{I})$ un système d'indépendance. Pour tout circuit C de $(E, \mathcal{I})$ nous posons

$$\mathcal{M}_C = (E, \mathscr{C}) \quad \text{où} \quad \mathscr{C} = \{X \in 2^E;\ C \smallsetminus X \neq \emptyset\}.$$

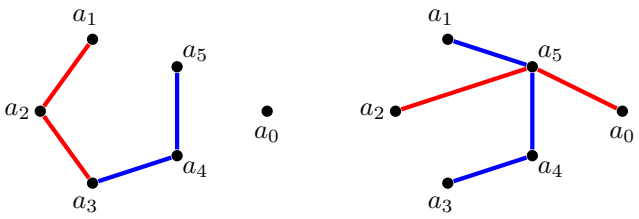
Le couple $\mathcal{M}_C$ est un matroïde : nous reconnaissons le matroïde de partition, dont les indépendants sont formés de moins de $|C| - 1$ éléments provenant de C et d'un nombre quelconque d'élément de $E \smallsetminus C$. Dans le matroïde $\mathcal{M}_C$, l'ensemble C n'est pas indépendant. De plus, tout indépendant de $(E, \mathcal{I})$ est encore un indépendant de $\mathcal{M}_C$. Par conséquent, l'intersection $\bigcap_{C \text{ circuit}} \mathcal{M}_C$ correspond à $(E, \mathcal{I})$. □

Remarque 4.31. — Seule la propriété d'hérédité est stable par intersection. La propriété d'augmentation ne l'est pas. Ainsi, l'intersection de deux matroïdes n'est pas un matroïde en général.

Par exemple, dans l'exercice 4.27, on recherche un arbre couvrant, avec exactement k arêtes bleues et $n-k-1$ arêtes rouges dans un graphe dont les arêtes sont bicolorées, ce qui peut s'exprimer comme l'intersection du matroïde graphique usuel et du matroïde de partition où les ensembles indépendants sont les ensembles comportant $\leq k$ arêtes bleues et $\leq n-k-1$ arêtes rouges. Dans ce cas, l'intersection n'est pas un matroïde.



Pour voir cela, considérons le cas où, dans le graphe ci-dessus, on demande ≤ 3 arêtes bleues et ≤ 2 arêtes rouges. L'exemple ci-après montre que la propriété d'augmentation n'est pas satisfaite.

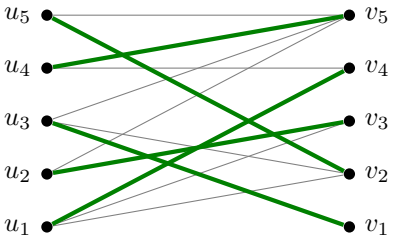


We now give different examples where matroid intersection nicely expresses classical notions

Exemple 4.32 **(Bipartite matching).** — Recall remark 4.16. Let $G = (U \sqcup V, E)$ be a bipartite graph and $\mathcal{I} \subseteq 2^E$ the set of matchings. Let $\delta(v)$ denote the set of edges incident to the vertex v. Then $(E, \mathcal{I})$ is not a matroid but the intersection of two partition matroids $\mathcal{I}_U \cap \mathcal{I}_V$ where

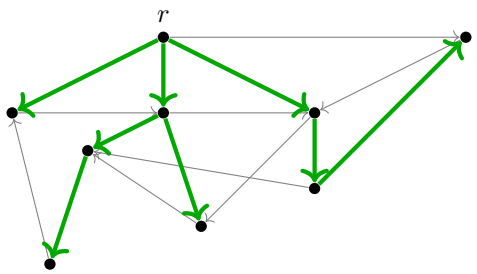
$$\mathcal{I}_U = \{X \subseteq E;\ \forall u \in U, |X \cap \delta(u)| \leq 1\}$$

$$\mathcal{I}_V = \{X \subseteq E;\ \forall v \in V, |X \cap \delta(v)| \leq 1\}.$$



Exemple 4.33 **(Arborescences).** — Recall that an r-arborescence in a directed graph $D = (V, A)$ is a spanning tree with arcs oriented away from the root r. This can be encoded with the intersection of two matroids. Let A^* be the set of non-oriented arcs in A and $\delta^-(v)$ the set of incoming edges in the node v. Now let $\mathcal{I}_1$ be be the independent sets of the graphic matroid on (V, A^*) and let $(A^*, \mathcal{I}_2)$ be the partition matroid with

$$\mathcal{I}_2 = \{X \subseteq A;\ \forall v \in V \setminus r, |X \cap \delta^-(v)| \leq 1\}.$$



***Exemple 4.34* (Colorful spanning tree).** — This follows exercise 4.27. Suppose that the set of edges E of graph (V, E) on n nodes is colored by ℓ colors $E = E_1 \sqcup E_2 \sqcup \cdots \sqcup E_\ell$ and that $n-1 = k_1 + \cdots + k_\ell$, than a spanning tree of G with k_i edges of color i is a maximal element of the intersection of $\mathcal{I}_1 \cap \mathcal{I}_2$ where $(E, \mathcal{I}_1)$ is the usual graphic matroid and $(E, \mathcal{I}_2)$ is the partition matroid.

Exemple 4.35. — The existence of two edge disjoint spanning trees in a graph can be decided using matroids intersection. Applications of this include for example questions about the rigidity of the graph (cf. [**Lee04**] chap. 3) or game theory (Shannon switching game, cf. [**Law01**], chap 8.).

Exercice 4.36. — Recall that the HAMILTONIAN PATH PROBLEM (Π) is the following NP-hard decision problem : given a directed graph $D = (V, A)$ and two nodes a and b, decide if there exists a directed path from a to b that visits every vertex of the graph D exactly once.

Show that this problem can be reduced to the computation of a maximum set the intersection of three matroids. What can you conclude ?

4.3.2. Algorithme d'Edmonds pour l'intersection de deux matroïdes. — Dans cette partie, nous nous intéressons au problème suivant, qui est une spécialisation du problème 4.3.

***Problème 4.37* (Intersection de deux matroïdes).** — *. Étant donné deux matroïdes $(E, \mathcal{I}_1)$ et $(E, \mathcal{I}_2)$ portant sur le même ensemble de base et une fonction de poids $w : E \to \mathbb{R}$, déterminer un ensemble indépendant I de $\mathcal{I}_1 \cap \mathcal{I}_2$ qui maximise la quantité $w(I) = \sum_{e \in I} w(e)$.*

4.3.2.1. *Une borne a priori sur l'optimum.* — Comme dans chaque problème d'optimisation combinatoire, commençons par chercher des informations *a priori* sur l'optimum. Nous verrons que le lemme suivant pourra nous servir de critère d'arrêt de l'algorithme d'Edmonds.

***Lemme 4.38* (Certificat d'optimalité).** — *Soient deux matroïdes $\mathcal{M}_1 = (E, \mathcal{I}_1)$ et $\mathcal{M}_2 = (E, \mathcal{I}_2)$, U un sous-ensemble quelconque de E, $\overline{U} = E \setminus U$ son complémentaire et S un élément de $\mathcal{I}_1 \cap \mathcal{I}_2$, alors*

$$|S| \leq \mathrm{rg}_{\mathcal{M}_1}(U) + \mathrm{rg}_{\mathcal{M}_2}(\overline{U}).$$

De plus, en cas d'égalité, il est certain que

$$|S| = \max_{I \in \mathcal{I}_1 \cap \mathcal{I}_2} |I|.$$

Démonstration. — Soit S un élément de $\mathcal{I}_1 \cap \mathcal{I}_2$ et U une partie quelconque de E. Découpons I en deux morceaux : $J_1 = S \cap U$ et $J_2 = S \cap \overline{U}$. Par hérédité, J_1 et J_2 sont encore des indépendants de $\mathcal{M}_1$ et $\mathcal{M}_2$. De plus, comme $J_1 \subseteq U$, on doit avoir $\mathrm{rg}_{\mathcal{M}_1}(J_1) \leq \mathrm{rg}_{\mathcal{M}_1}(U)$. De même, on a $\mathrm{rg}_{\mathcal{M}_2}(J_2) \leq \mathrm{rg}_{\mathcal{M}_2}(\overline{U})$. Mais $\mathrm{rg}_{\mathcal{M}_1}(J_1) = |J_1|$ et $\mathrm{rg}_{\mathcal{M}_2}(J_2) = |J_2|$. Ainsi

$$|S| = |J_1| + |J_2| \leq \mathrm{rg}_{\mathcal{M}_1}(U) + \mathrm{rg}_{\mathcal{M}_2}(\overline{U}).$$

Le cas d'optimalité est clair. □

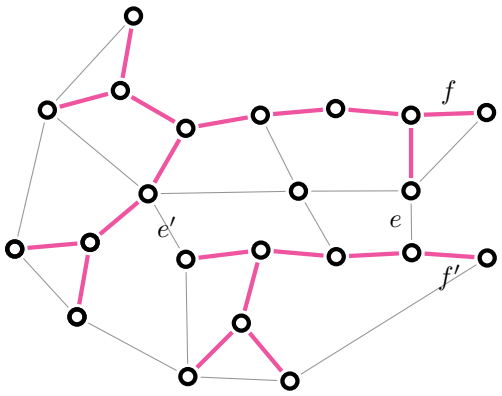
4.3.2.2. *Les idées de l'algorithme.* — Nous connaissons déjà un algorithme pour résoudre le problème 4.37 dans un cas particulier, à savoir l'algorithme de recherche d'un couplage parfait dans un graphe biparti grâce à des chemins augmentants (voir section 2.2). Nous renvoyons à l'exemple 4.32 pour les détails : rappelons simplement que, dans cet exemple, le matroïde $\mathcal{M}_1$, respectivement $\mathcal{M}_2$, exprime alors la condition « les sommets de gauche, respectivement de droite, sont couverts au plus une fois ». Dans cet algorithme, à chaque étape, nous utilisons une suite d'arêtes e_0, f_1, e_1, …, f_t, e_t (qui forment un chemin alternant de longueur $2t + 1$) comme suit : nous ajoutons e_0 au couplage, nous retirons f_1, nous ajoutons e_1, etc. L'arête e_0 sert à augmenter le rang au sens du premier matroïde $\mathcal{M}_1$ (un sommet supplémentaire est couvert à gauche) et la suite $(f_1, e_1, \ldots, f_t, e_t)$ ne sert qu'à changer d'indépendant au sens de $\mathcal{M}_1$. Vis-à-vis du second matroïde, l'arête e_t sert à augmenter le rang au sens de $\mathcal{M}_2$ (un sommet supplémentaire est couvert à droite) et la suite $(f_t, e_{t-1}, \ldots, e_1, f_1, e_0)$ ne sert qu'à changer d'indépendant au sens de $\mathcal{M}_2$.

Nous aimerions nous inspirer de cette démarche dans le cas de l'intersection de deux matroïdes. Nous démarrons avec $S = \emptyset$ qui est un élément trivial de $\mathcal{I}_1 \cap \mathcal{I}_2$. À chaque, nous cherchons à remplacer S par un nouvel élément de S' de $\mathcal{I}_1 \cap \mathcal{I}_2$ tel que $|S'| = |S| + 1$. Précisément, nous cherchons une suite e_0, f_1, e_1, …, f_t, e_t où e_0, e_1, … e_n appartiennent à $\overline{S} = E \smallsetminus S$ et f_1, …, f_t appartiennent à S de telle manière que

$$S' = S \cup \{e_0, e_1, \ldots, e_t\} \smallsetminus \{f_1, \ldots, f_t\}$$

soit encore dans $\mathcal{I}_1 \cap \mathcal{I}_2$.

Dans le cas de l'exemple 4.32, nous sommes aidés par le fait que les arêtes que nous ajoutons n'interfèrent pas entre elles sur le caractère « être indépendant au sens du matroïde $\mathcal{M}_1$ ou $\mathcal{M}_2$ ». En effet, les arêtes qu'il est possible de choisir pour couvrir un certain sommet sont toutes distinctes lorsqu'on passe à un autre sommet du même côté de la bipartition. Dans le cas de deux matroïdes quelconques, faire des échanges conjoints n'est pas possible. Par exemple, dans le matroïde graphique suivant



Nous pouvons échanger f avec e ou échanger f' avec e' et garder un indépendant de même cardinal, mais nous ne pouvons pas échanger $\{f, f'\}$ avec $\{e, e'\}$: cela crée un circuit. Nous avons besoin du lemme suivant qui nous précise dans quelles conditions nous pouvons procéder à un échange entre éléments de l'ensemble de base.

Définition 4.39. — Pour tout ensemble indépendant I d'un matroïde $\mathcal{M} = (E, \mathcal{I})$ et pour tout élément $x \in E$, nous notons $C(I, e)$ l'unique circuit de $I + e$ dans $\mathcal{M}$ si $I + e \notin \mathcal{I}$ et posons encore $C(I, e) = \emptyset$ sinon (cf. théorème 4.21).

***Lemme 4.40* (Frank 1981).** — *Soient $\mathcal{M} = (E, \mathcal{I})$ un matroïde, $I \in \mathcal{I}$ un ensemble indépendant. Soient $e_1, \ldots, e_t$ des éléments de $\overline{I} = E \setminus I$ et $f_1, \ldots, f_t$ des éléments de I tels que, pour tout indice k compris entre 1 et t,*

(i) $f_k \in C(I, e_k)$ *et*

(ii) $f_k \notin C(I, e_{k+1}) \cup C(I, e_{k+2}) \cup \cdots \cup C(I, e_t)$.

Alors $I \cup \{e_1, \ldots, e_t\} \setminus \{f_1, \ldots, f_t\}$ est encore indépendant dans $\mathcal{M}$.

Démonstration. — Pour tout indice k compris entre 0 et t, notons I_k l'ensemble

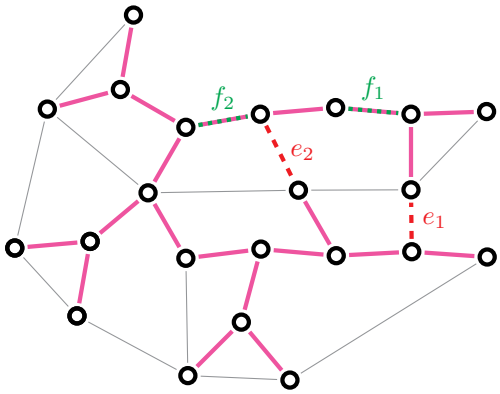
$$I_k = I \cup \{e_1, \ldots, e_k\} \setminus \{f_1, \ldots, f_k\}.$$

Démontrons par récurrence sur l'entier k que I_k est indépendant.

Pour $k = 0$ et $I_0 = I$, il n'y a rien à faire.

Supposons avoir démontré que I_{k-1} est indépendant pour un certain indice k compris entre 1 et t. Si $I_{k-1} + e_k$ est indépendant, il n'y a rien à faire : I_k est indépendant par hérédité. Sinon, $I_{k-1} + e_k$ contient un unique circuit, disons le circuit C. Nous affirmons que le circuit $C(I, e_k)$ est aussi inclus dans $I_{k-1} + e_k$. En effet, supposons qu'il existe un élément z de $C(I, e_k)$ qui n'est pas dans $I_{k-1} + e_k$: ce serait forcément un élément de I (et non pas e_k) que l'on ne retrouverait pas dans I_{k-1}, autrement dit, il s'agirait de l'un des éléments f_1, f_2, …, f_{k-1}. Mais alors, nous contredirions l'hypothèse (ii). Comme les circuits sont minimaux pour l'inclusion, nous devons avoir l'égalité $C = C(I, f_k)$. Mais alors, d'après l'hypothèse (i), l'élément f_k provient du circuit créé par l'ajout de e_k. En le retirant, $I_k = I_{k-1} + e_k \setminus f_k$ redevient indépendant, comme voulu. □

Exemple 4.41. — Pour illustrer le lemme 4.40, considérons l'exemple suivant dans le matroïde graphique.



Il est clairement possible de substituer e_1 et e_2 à f_1 et f_2 : cela provient du fait que f_1 n'appartient pas au cycle crée par l'ajout de e_2.

À présent, organisons nos pensées concernant les échanges qu'il est possible de faire. Notre esprit se porte naturellement sur une modélisation par un graphe.

Définition 4.42. — Soient deux matroïdes $\mathcal{M}_1 = (E, \mathcal{I}_1)$ et $\mathcal{M}_2 = (E, \mathcal{I}_2)$ et soit S un élément de $\mathcal{I}_1 \cap \mathcal{I}_2$. Nous appelons *graphe d'échange*, et notons $\mathcal{G}(S)$, le graphe biparti orienté dont les sommets sont S et $\overline{S} = E \setminus S$ et dont les arcs sont $(x, \overline{x}) \in S \times \overline{S}$ si $S \setminus x + \overline{x} \in \mathcal{I}_1$ et $(\overline{x}, x) \in \overline{S} \times S$ si $S \setminus x + \overline{x} \in \mathcal{I}_2$.

Par ailleurs, nous identifions deux ensembles : l'ensemble $X_1 = \{u \in \overline{S};\ S+u \in \mathcal{I}_1\}$ que nous appelons *source* et l'ensemble $X_2 = \{v \in \overline{S};\ S + v \in \mathcal{I}_2\}$ que nous appelons *puits*.

Nous renvoyons à la figure 4.1 pour une illustration.

Proposition 4.43. — *Si*

$$e_0 - f_1 - e_1 - \cdots - f_t - e_t$$

est un plus court chemin dans le graphe d'échange $\mathcal{G}(S)$ allant de la source X_1 vers le puits X_2, alors on peut remplacer S par

$$S' = S \cup \{e_0, e_1, \ldots, e_t\} \setminus \{f_1, \ldots, f_t\}$$

et l'ensemble S' appartient toujours à $\mathcal{I}_1 \cap \mathcal{I}_2$ avec un élément de plus que S.

Démonstration. — Par définition de la source X_1, l'ensemble $S + e_0$ est un indépendant de $\mathcal{M}_1$. Montrons que le lemme 4.40 s'applique dans le matroïde $\mathcal{M}_1$ à $S + e_0$, les éléments $e_1, \ldots, e_t$ et les éléments $f_1, \ldots, f_t$. La propriété (i) du lemme est vérifiée par construction du graphe d'échange. Si la propriété (ii) du lemme était violée, nous aurions $f_k \in C(S, e_j)$, autrement dit nous aurions dans le graphe d'échange une arête entre f_k et un certain e_j avec $k < j$. Mais alors, nous pourrions raccourcir le chemin en

$$e_0 - f_1 - e_1 - \cdots - f_k - e_j - \cdots - f_t - e_t$$

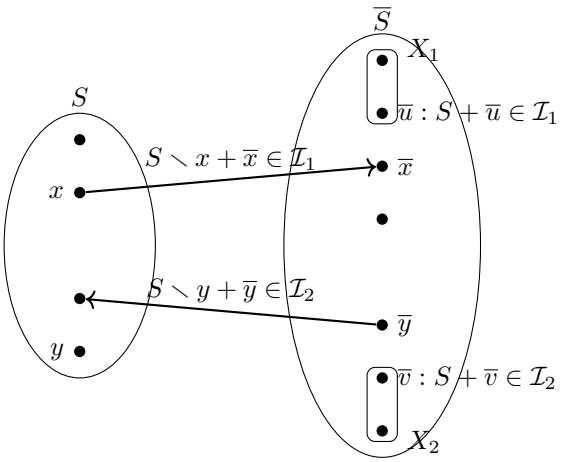


FIGURE 4.1. Graphe d'échange

ce qui contredit la minimalité du plus court chemin. En conséquence, nous avons démontré que $S' \in \mathcal{I}_1$.

Par définition de X_2, l'ensemble $S + e_t$ est un indépendant de $\mathcal{M}_2$. Avec des arguments semblables à ceux vus ci-dessus, le lemme 4.40 s'applique dans le matroïde $\mathcal{M}_2$ à $S + e_t$, les éléments $e_{t-1}, e_{t-1}, \ldots, e_1, e_0$ et les éléments $f_t, f_{t-1} \ldots, f_2, f_1$. ce qui montre que $S' \in \mathcal{I}_2$. □

4.3.3. Présentation et correction de l'algorithme. — À ce stade, nous obtenons l'algorithme suivant.

Algorithme 4.2 : Cardinal maximum dans une intersection de deux matroïdes

Entrées : Deux matroïdes $\mathcal{M}_1 = (E, \mathcal{I}_1)$ et $\mathcal{M}_2 = (E, \mathcal{I}_2)$ sur un même ensemble de base

Sorties : Ensemble $S \in \mathcal{I}_1 \cap \mathcal{I}_2$ de cardinal maximum

```
1 S ← ∅
2 répéter
3 |   Construire le graphe d'échange G(S)
4 |   Calculer un plus court chemin e_0 − f_1 − e_1 − ··· − f_n − e_n entre la
  |    source X_1 et le puits X_2.
5 |   si le chemin existe alors
6 |   |   S ← S ∪ {e_0, e_1, ..., e_n} ∖ {f_1, ..., f_n}.
7 jusqu'à le puits X_2 n'est pas accessible depuis la source X_1 dans le
    graphe d'échange G(S) ;
8 retourner S
```

Remarque 4.44. — En pratique, il vaut mieux démarrer avec un ensemble S non vide si on en connait déjà un. On peut par exemple en chercher un avec une stratégie gloutonne.

Lemme 4.45. — *Soit S un sous-ensemble de E. On suppose qu'il n'existe pas de chemin entre la source X_1 et le puits X_2 dans le graphe d'échange $\mathcal{G}(S)$. Alors l'ensemble S est de cardinal maximum dans $\mathcal{I}_1 \cap \mathcal{I}_2$.*

Démonstration. — La proposition 4.43 couvre le cas où il existe un chemin reliant la source au puits.

Supposons qu'il n'existe pas chemin allant de X_1 à X_2. Notons $\overline{U}$ l'ensemble des sommets accessibles dans le graphe d'échange $\mathcal{G}(S)$ depuis X_1. Par hypothèse, $\overline{U} \cap X_2 = \emptyset$.

Nous affirmons que $\mathrm{rg}_{\mathcal{M}_2}(\overline{U}) = |S \cap \overline{U}|$. Comme $S \cap \overline{U}$ est un indépendant de $\mathcal{M}_2$, nous avons au moins $\mathrm{rg}_{\mathcal{M}_2}(\overline{U}) \geq |S \cap \overline{U}|$. Si l'inégalité n'avait pas lieu, nous pourrions compléter $(S \cap \overline{U})$ par un élément $\overline{x}$ appartenant à $\overline{U} \setminus S$ tel que $(S \cap \overline{U}) + \overline{x} \in \mathcal{I}_2$. Mais, comme $\overline{U}$ est disjoint de X_2, on doit avoir que $\overline{x} \notin X_2$, ce qui signifie que $S + \overline{x}$ est dépendant dans $\mathcal{M}_2$. En conséquence, $C(S, \overline{x})$ est bien un circuit et doit contenir un élément x tel que $S + \overline{x} \setminus x$ est indépendant dans $\mathcal{I}_2$. De plus x ne peut venir de $S \cap \overline{U}$ car $(S \cap \overline{U}) + \overline{x} \in \mathcal{I}_2$, donc x provient de $S \setminus \overline{U}$. Mais alors, $(\overline{x}, x)$ est un arc du graphe $\mathcal{G}(S)$ sortant de $\overline{U}$ menant vers un sommet hors de $\overline{U}$: cela contredit la définition de $\overline{U}$.

Posons $U = E \setminus \overline{U}$. Prouvons à présent que $\mathrm{rg}_{\mathcal{M}_1}(U) = |S \cap U|$. Si ce n'était pas le cas, il existerait un élément $\overline{x}$ dans $U \setminus S$ tel que $S \cap U + \overline{x}$ soit indépendant dans $\mathcal{M}_1$. Comme $\overline{x} \notin \overline{U}$, on a $\overline{x} \notin X_1$, donc $X + \overline{x}$ est dépendant dans $\mathcal{M}_1$. Alors, l'ensemble $C(X, \overline{x})$ est bien un circuit et contient un élément $x \in S \cap \overline{U}$ tel que $S + \overline{x} \setminus x \in \mathcal{I}_1$. On trouvé un arc $(x, \overline{x})$ qui sort de $\overline{U}$ ce qui est impossible.

Nous pouvons à présent appliquer le lemme 4.38 qui certifie l'optimalité. □

Théorème 4.46 **(Edmonds).** — *L'algorithme d'Edmonds résout correctement le problème de l'intersection de deux matroïdes en temps $O\left(r^2 n(1 + C_1 + C_2)\right)$ où r est le cardinal du résultat, n est le cardinal de l'ensemble de base E et C_1, respectivement C_2, la complexité de l'oracle décidant l'indépendance dans le matroïde $\mathcal{M}_1$, respectivement dans $\mathcal{M}_2$.*

Démonstration. — La preuve de correction provient de la proposition 4.43 et du lemme 4.45.

La complexité se déduit du calcul suivant

1. La construction du graphe d'échange demande la création de $O(r)$ sommets à gauche (ceux de S) et $O(n)$ sommets à droite (ceux de $E \setminus S$). Pour chaque couple de sommet, il faut vérifier si un arc existe dans chaque sens. Pour cela il faut tester un indépendance dans chaque matroïde, pour un coût $C_1 + C_2$ par couple de sommets. Cette étape coûte au total $O(rn(1 + C_1 + C_2))$.
2. La construction de la source et du puit demande un appel à l'oracle par élément de E, soit un total de $O(n(C_1 + C_2))$ appels.

3. La recherche d'un plus court chemin peut se faire par un parcours de graphe en largeur, ce qui coûte $O(n + rn) = O(rn)$ opérations.
4. Remplacer S avec le chemin obtenu se fait en $O(n)$.

Toutes ces étapes sont répétées $O(r)$ fois. Au total, la complexité est bien $O(r^2 n(1 + C_1 + C2))$. □

Nous tirons une autre conséquence :

***Théorème 4.47* (Edmonds, 1970).** — *Soient deux matroïdes* $\mathcal{M}_1 = (E, \mathcal{I}_1)$ *et* $\mathcal{M}_2 = (E, \mathcal{I}_2)$. *On a l'*inégalité min-max *suivante :*

$$\max_{I \in \mathcal{I}_1 \cap \mathcal{I}_2} |I| = \min_{U \subseteq E} \left[\mathrm{rg}_{\mathcal{M}_1}(U) + \mathrm{rg}_{\mathcal{M}_2}(\overline{U}) \right]. \tag{2}$$

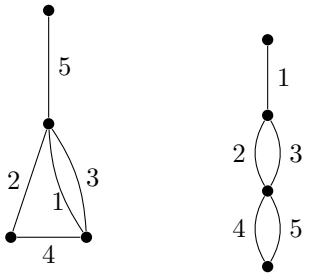
Remarque 4.48. — The algorithm 4.2 is a general purpose polynomial time algorithm for matroid intersection. For specific instances with given types of matroids, there can be faster algorithms. For example, although bipartite matching can be seen as a matroid intersection good algorithms have been developped in this specific case.

Remarque 4.49. — Given a weight function on the elements of the matroids, algorithm 4.2 can be adjusted to account for weights on the elements of E, *i.e.* to compute a set S that reaches

$$\max_{S \in \mathcal{I}_1 \cap \mathcal{I}_2} w(S).$$

Call $\sum_{i=0}^{n} w(e_i) - \sum_{i=1}^{n} w(f_i)$ the incremental weight of an augmenting sequence $e_0 - f_1 - e_1 - \cdots - f_n - e_n$. In algorithm 4.2, it is actually enough to start with the empty set and to use at every step the shortest path among all the sequences $e_0 - f_1 - e_1 - \cdots - f_n - e_n$ with maximal incremental weight in order to compute the weighted maximum.

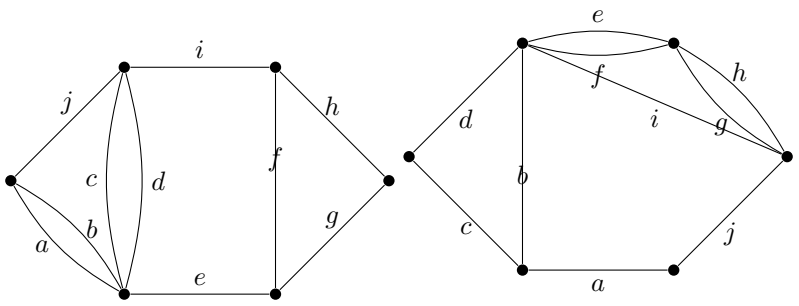
Exercice 4.50. — Consider the intersection of the two graphic matroids given by the following graphs.



Suppose the algorithm has already selected the set $S = \{2, 4\} \in \mathcal{I}_1 \cap \mathcal{I}_2$.

1. Find a maximal set from S by applying algorithm 4.2.
2. Find an augmenting sequence that is not the shortest and such that the new set S would no longer be in $\mathcal{I}_1 \cap \mathcal{I}_2$.

Exercice 4.51. — Find a maximum independent set in the intersection of the two graphic matroids given by following graphs :



Exercice 4.52. — You are in charge of the music program of the world-famous *Freezing Alien* festival. At this festival, each of the five continents should be represented by a group. Now, since the audience is opinionated about what they like to listen to, they would be miffed if one of the following music styles were not represented : *Old-days Hipster Bebop Jazz*, *Ultimate Steampunk Opera*, *Survivalist Whistling Choral*, *Post Cybergoth Übertrance*, and *Synthetic Bubblegum Pop-rock*. Since the festival takes place in the middle of the Antartic desert, you cannot really afford to invite more than five groups. Yet, if you are cool enough to come up with an algorithm that, given a world-wide list of all the music groups, computes a short list with exactly $(c_i)_{i\leq n}$ groups from country $(C_i)_{i\leq n}$ and $(s_j)_{j\leq m}$ groups with music style $(S_j)_{j\leq m}$, you would have a hell of a lot of swag !

Exercice 4.53. — We are day one after the end of the world. Only six cities have survived : Berlin, London, Madrid, New-York, Paris, and Rome. You have only found a boat, a bus, a plane and two trains under the rubble. You have to reorganise the world public transport system so that all the cities are connected together. How can you model the situation ? Suggest an algorithm to solve it.

Exercice 4.54. — Given an undirected graph $G = (V, E)$, an *orientation* is a directed graph $D = (V, A)$ with a bijection $\phi : A \to E$ such that $\phi((u, v)) = \{u, v\}$: in other words, each edge $\{u, v\} \in E$ is given a direction, either (u, v) or (v, u).

Given tupple of integers $(k_v)_{v\in V}$, show that the problem of finding an orientation such that

$$\forall v \in V, \qquad \delta^-(v) = k_v,$$

or showing that none exists, can be solved using the matroid intersection algorithm.

***Exercice 4.55* (König theorem).** — We call *vertex cover* of graph G a selection of vertices of G such that any edge of G is incident to one of the selected vertices. Let G be a bipartite graph. Prove with theorem 4.47 that any maximum matching and minimum vertex cover have the same cardinality

***Exercice 4.56* (Disjoint bases).** — 1. Let $\mathcal{M} = (E, \mathcal{I})$ be a matroid and k an integer. Then $\mathcal{M}$ possesses k disjoint bases if and only if for any set $X \in 2^E$,

$$|E \setminus X| \geq k\left(\mathrm{rg}_{\mathcal{M}}(E) - \mathrm{rg}_{\mathcal{M}}(X)\right).$$

[*Hint : use the ground set* $E \times [k]$ *to define two judicious matroids.*]

2. Deduce the following theorem due to Nash-Williams. A connected graph $G = (V, E)$ contains k edge disjoint spanning trees if and only if for any partition $(V_i)_{i \leq \ell}$ of the set of vertices V, the total number of eges between nodes belonging to different parts is at least $k(\ell - 1)$.

TRAVAUX PRATIQUES : IMPLÉMENTATION DE L'ALGORITHME DE L'INTERSECTION DE MATROÏDES D'EDMONDS

Objectif. — Le but de ce TP est d'implémenter l'algorithme d'Edmonds de construction d'un *indépendant de cardinal maximum dans l'intersection de deux matroïdes* génériques.

Langage de programmation. — Le TP est proposé en Python. Vous êtes libre de l'adapter à un autre langage de programmation. Si vous êtes mal à l'aise avec les rudiments de programmation orientée objet en Python, vous pouvez consulter les pages consacrées dans un manuel scolaire de lycée (par exemple chapitre 2 de [**CGG+22**]).

Conseils. — Faites beaucoup de tests au fur et à mesure. N'hésitez pas à coder des fonctions qui vérifient les invariants attendus pour vous assurer de la justesse de votre code.

4.4. Construction de matroïdes classiques

Question 4.1. — Initialiser un fichier `matroid.py` contenant le code suivant. Un fichier squelette est disponible sur le site du cours E-Campus pour vous aider.

```
88 class Matroid():
89     def __init__(self, E={}):
90         self.elements = {e for e in E}
91 
92     def is_independent(self, X):
93         # this method should be implemented at the level of the
           ↪ subclasses
94         raise NotImplementedError
```

4.4.1. Implémentation du matroïde de partition. — Pour représenter un matroïde de partition $\mathcal{M} = (E, \mathcal{I})$ dont les couleurs des éléments constituent l'ensemble C, nous nous appuyons sur deux applications $\chi : E \to C$ et $\mu : C \to \mathbb{N}$ où $\chi(e)$ est la couleur de l'élément e et $\mu(c)$ est le nombre maximum d'éléments

de couleur c autorisés dans un indépendant (voir exemple 4.10). Autrement dit, les indépendants sont alors

$$\mathcal{I} = \{I \subseteq E;\ \forall c \in C,\ |I \cap \chi^{-1}(c)| \leq \mu(c)\}$$

Question 4.2. — Écrire une classe `PartitionMatroid` qui hérite de la classe `Matroid` et tel que la syntaxe `M = PartitionMatroid(chi, mu)` permette de déclarer un matroïde de partition à partir de deux dictionnaires χ et μ. On pourra utiliser les données de la section 5.9 pour écrire des tests.

4.4.2. Implémentation du matroïde graphique. — Nous aurons besoin premièrement d'une structure permettant de détecter des cycles dans un graphe non orienté.

Question 4.3. — À titre préliminaire en vue de la question 4.4, écrire une classe `UnionFind` dans un fichier autonome (par exemple `unionfind.py`) qui implémente la structure « unir et trouver » permettant de représenter des partitions d'ensembles. On prévoira un deux méthodes `union(self, u, v)` et `find(self, v)` en plus du constructeur. Il est attendu que la méthode `union` ait pour valeur de retour un booléen qui indique si une union entre deux parts a réellement eu lieu.

Question 4.4. — Écrire une classe `GraphicMatroid` qui hérite de la classe `Matroid` et tel que la syntaxe `M = GraphicMatroid(E)` permette de déclarer un matroïde graphique à partir d'une liste d'arête E. On pourra utiliser les données de la section 5.9 pour écrire des tests.

4.5. Algorithme d'Edmonds

Question 4.5. — À partir de la définition 4.42, écrire une classe `ExchangeGraph` telle que la syntaxe `G = ExchangeGraph(S, M1, M2)` obéisse à la spécification suivante :
Préconditions : Les objets $\mathcal{M}_1$ et $\mathcal{M}_2$ sont des matroïdes ayant le même ensemble de base, l'ensemble S est un indépendant de $\mathcal{M}_1 \cap \mathcal{M}_2$.
Postcondition : L'objet G possède les attributs :

- `G.edges` contenant les arcs du graphe d'échange $\mathcal{G}(S)$ sous forme de dictionnaire $S \to 2^{\overline{S}} + \overline{S} \to 2^S$ ayant un sommet comme clé et ayant la liste de ses voisins comme valeur associée,
- `G.source` contenant l'ensemble X_1,
- `G.sink` contenant l'ensemble X_2.

Question 4.6. — En lien avec la proposition 4.43, enrichir la classe `ExchangeGraph` d'une méthode `_shortest_path(self)` dont la valeur de retour est un plus court chemin allant de `self.source` vers `self.sink`.
En phase de déboggage, on pourra vérifier que le chemin obtenu est de longueur paire et alterne entre éléments de $\overline{S}$ et de S.

Question 4.7. — Écrire une fonction `maximum_intersection(M1, M2)` dont la valeur de retour est un ensemble indépendant dans l'intersection de matroïde $\mathcal{M}_1 \cap \mathcal{M}_2$ de cardinal maximum.

Question 4.8. — Confronter votre implémentation avec la complexité avancée par le théorème 4.46 dans chacun des cas de la section 5.9.

4.6. Pour aller plus loin

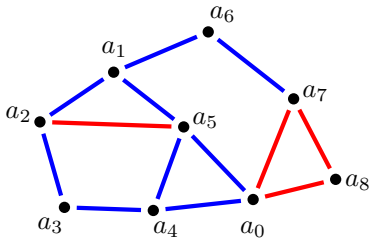
Question 4.9. — Écrire une classe `VectorMatroid` qui hérite de la classe `Matroid` et qui permette de déclarer un matroïde linéaire sur le corps fini à deux éléments.

Question 4.10. — Résoudre le problème de programmation compétitive suivant : `https://codeforces.com/gym/102156/problem/D`

4.7. Cas de tests

Les données des cas de test suivant sont disponibles sur E-Campus.

4.7.1. Forêt bicolore. — Dans la figure 4.2, on cherche une forêt de cardinal maximum composée de moins de 4 arêtes bleues et moins de 4 arêtes rouges. Ces contraintes peuvent se voir l'intersection de deux matroïdes portant sur les arêtes : un matroïde graphique et un matroïde de partition. La couleur d'un élément est indiquée par l'application χ ; le maximum par couleur est indiqué par l'application μ.



$\chi(a_0a_4)$	=	bleu
$\chi(a_0a_5)$	=	bleu
$\chi(a_0a_7)$	=	rouge
$\chi(a_0a_8)$	=	rouge
$\chi(a_1a_5)$	=	bleu
$\chi(a_1a_6)$	=	bleu
$\chi(a_2a_1)$	=	bleu
$\chi(a_2a_3)$	=	bleu
$\chi(a_2a_5)$	=	rouge
$\chi(a_3a_4)$	=	bleu
$\chi(a_4a_5)$	=	bleu
$\chi(a_6a_7)$	=	bleu
$\chi(a_7a_8)$	=	rouge
μ(rouge)	=	4
μ(bleu)	=	4

FIGURE 4.2. Arbre couvrant dans un graphe bicolore

4.7.2. Couplage dans un graphe biparti. — Dans la figure 4.3, on cherche un couplage de cardinal maximum, c'est-à-dire un ensemble d'arêtes dont les extrémités possèdent des couleurs et des formes toutes distinctes.

4.7.3. Arborescence. — Dans la figure 4.4, on cherche un ensemble d'arcs avec au plus un arc entrant par sommet et et qui forme un graphe acyclique lorsqu'on oublie l'orientation.

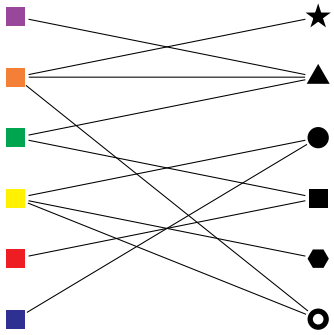


FIGURE 4.3. Graphe biparti et couplages

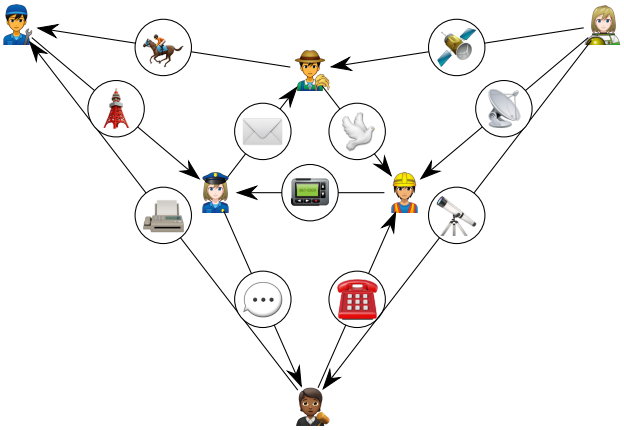


FIGURE 4.4. Graphe orienté et arborescence

CHAPITRE 5

FLOTS DE COÛT MINIMUM

5.1. Le problème

5.1.1. Formulation du problème. — Nous adoptons les notations suivantes : si $G = (V, E)$ est un graphe, si S et $T \subseteq V$ sont des ensembles de sommets, la notation $E(S, T)$ désigne l'ensemble des arcs allant de S vers T, autrement dit, nous posons :

$$E(S, T) = \{uv;\ u \in S, v \in T \text{ et } uv \in E\}.$$

Définition 5.1. — Soient $G = (V, E)$ un graphe orienté, $u : E \to \mathbb{R}_{\geq 0}$ une application appelée *capacité*, $b : V \to \mathbb{R}$ une application telle que $\sum_{v \in V} b(v) = 0$. On appelle *b-flot* toute fonction $f : E \to \mathbb{R}_{\geq 0}$ telle que

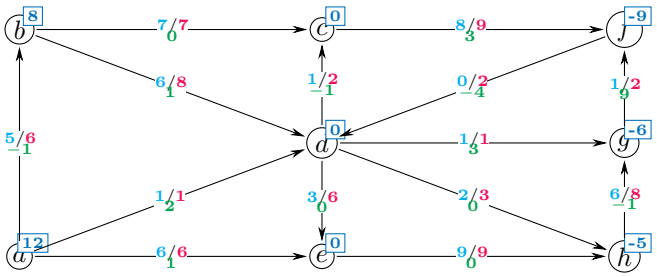
$$\forall e \in E, \quad f(e) \leq u(e)$$

et

$$\forall v \in V, \quad \sum_{e \in E(\{v\}, V \smallsetminus \{v\})} f(e) - \sum_{e \in E(V \smallsetminus \{v\}, \{v\})} f(e) = b(v).$$

Vocabulaire 5.2. — Les sommets se classent en trois catégories : lorsque b est strictement positif, on parle d'*offre* : il s'agit des sommets où le flot est produit ; lorsque b est strictement négatif, on parle d'*offre* : il s'agit de sommets où le flot est consommé ; lorsque b est nul, on parle de *transbordement* : il s'agit de sommets où le flot est en transit.

Exemple 5.3. — Voici un exemple de b-flot. Le flot f est indiqué en bleu clair, les capacités sont indiquées en rouge, la fonction b est indiquée en bleu sombre.



Les nombres indiquées en vert désignent une fonction coût, qui sera introduite plus tard.

On parle de *circulation* lorsque b est la fonction toute à zéro.

Lemme 5.4. — *Si $g : E \to \mathbb{R}_{\geq 0}$ est une circulation, alors il existe une famille $\mathscr{C}$ de circuit de G de cardinal au plus $|E|$ et des réels positifs $(\varphi_\chi)_{\chi\in\mathscr{C}}$ tels que la circulation f se décompose comme la combinaison linéaire de fonctions indicatrices*

$$g = \sum_{\chi\in\mathscr{C}} \varphi_\chi \mathbf{1}_\chi,$$

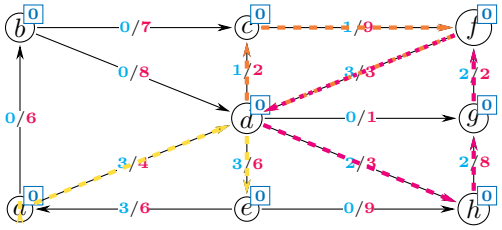
où

$$\mathbf{1}_\chi : \begin{cases} E & \to & \mathbb{R} \\ e & \mapsto & \begin{cases} 1 & \text{si } e \in \chi \\ 0 & \text{sinon} \end{cases} \end{cases}$$

Démonstration. — Sans perte de généralité, on peut exclure tout arc e de E tel que $g(e) = 0$. Nous démontrons le résultat par récurrence sur $|E|$. Si E est vide, il n'y a rien à faire. Sinon, nous sélectionnons un arc $e = uv$ (avec $g(e) > 0$). Comme g est une circulation, $b(v)$ est nul et il doit exister un arc sortant du sommet v (avec $g(v) > 0$). En répétant l'argument, on peut construire un chemin. Ce chemin doit repasser par un sommet déjà rencontré, ce qui fournit un certain circuit χ. Nous notons $\gamma = \max_{e\in\chi} f(e)$. Nous observons que $g - \gamma\mathbf{1}_c$ est encore une circulation et que, pour un certain arc e de χ, $g(e) = 0$. Aussi, nous pouvons appliquer notre hypothèse de récurrence à $g - \gamma\mathbf{1}_c$ et conclure. □

Exemple 5.5. — Dans la circulation g suivante, on reconnait trois cycles : $\chi_1 = (ad, de, ea)$, $\chi_2 = (cf, fd, dc)$ et $\chi_3 = (dh, hg, gf, fd)$ avec des poids $\varphi_1 = 3$, $\varphi_2 = 1$ et $\varphi_3 = 2$. On a bien

$$g = \varphi_1\mathbf{1}_{\chi_1} + \varphi_2\mathbf{1}_{\chi_2} + \varphi_3\mathbf{1}_{\chi_3}.$$



***Problème 5.6* (Flot de coût minimum).** — *Étant donné un $G = (V, E)$ un graphe orienté, $u : E \to \mathbb{R}_{\geq 0}$ une application capacité, $b : V \to \mathbb{R}$ une application telle que $\sum_{v \in V} b(v) = 0$ et $c : E \to \mathbb{R}$ une application appelée* coût unitaire*, déterminer un b flot qui minimise la fonction objectif*

$$c(f) = \sum_{e \in E} c(e) f(e).$$

Exemple 5.7. — Dans l'exemple 5.3, le coût $c(f)$ vaut 38.

Remarque 5.8. — Le problème du flot de coût minimum admet facilement une présentation sous la forme d'un *programme linéaire*[1]. Dans ce cours, nous nous intéressons à des algorithmes combinatoires. On pourra consulter [**Bie15**] pour plus de liens avec l'optimisation linéaire.

Exercice 5.9. — Nous modifions le problème 5.6 en gardant les mêmes notation, le même objectif, les mêmes contraintes et en introduisant une contrainte supplémentaire. Nous supposons donnée une application $l : E \to \mathbb{R}_{\geq 0}$ avec, pour tout arc $e \in E$, l'inégalité $l(e) \leq u(e)$. Nous imposons la nouvelle contrainte suivante sur le b-flot f :

$$\forall e \in E, \quad l(e) \leq f(e) \leq u(e).$$

Montrer que ce nouveau problème peut se ramener au problème 5.6 et vice versa.

5.1.2. Lien avec des problèmes classiques. — Nous pouvons reformuler des problèmes classiques comme des problèmes de flot de coût minimum.

5.1.2.1. *Plus courts chemins.* — Le problème du plus court chemin dans un graphe pondéré $G = (V, E, c)$ entre un sommet source s et un sommet puits t peut être vu comme un problème de b-flot : on fixe une offre $b(s) = 1$, une demande $b(t) = -1$ et des sommets de transbordement $b(v) = 0$ pour tout sommet v de $V \setminus \{s, t\}$. On fixe comme capacité la fonction u toute à un. On utilise comme fonction de coûts la fonction c.

Le flot transporté de s vers t passe entièrement par un plus court chemin au sens de la fonction coût c.

5.1.2.2. *Flot de valeur maximum.* — Nous rappelons les définitions suivantes :

Définitions 5.10. — Un *réseau* est un quadruplet (G, u, s, t) où $G = (V, E)$ est un graphe, $u : E \to \mathbb{R}_{\geq 0}$ une fonction capacité, s un sommet particulier appelé *source*, t un sommet particulier appelé *puits*.

Dans ce contexte, un flot est une fonction $f : EE \to \mathbb{R}_{\geq 0}$ telle que, pour tout arc $e \in E$, $f(e) \leq u(e)$ et telle que, pour tout sommet distinct de la source et du puits, on a $\sum_{e \in E(\{v\}, V \setminus \{v\})} - \sum_{e \in E(V \setminus \{v\}, \{v\})} = 0$. Autrement dit, tous les sommets sont des sommets de transbordement sauf le sommet source qui est un sommet d'offre et le sommet puits qui est un sommet de demande.

1. Voir par exemple le cours FMA-0EL01-TP dans l'école

Enfin, nous appelons *valeur du flot* la quantité

$$|f| = \sum_{e \in E(\{v\}, V \setminus \{v\})} - \sum_{e \in E(V \setminus \{v\}, \{v\})}$$

qui correspond à la quantité de flux transportée de s vers t.

Problème 5.11 **(Flot maximum dans un réseau [3])**

Étant donné un réseau (G, u, s, t), déterminer un flot f qui maximise la valeur du flot $|f|$.

Nous pouvons transformer une instance du problème 5.11 en une instance du problème 5.6 comme suit. Nous conservons le graphe $G = (V, E)$. Nous ajoutons un arc allant de t vers s de capacité $u(ts) = +\infty$. Nous fixons comme fonction b la fonction toute à zéro (autrement dit, nous cherchons une circulation). Enfin, nous utilisons comme fonction de coût la fonction c telle que $c(e) = 0$ pour tout arc $e \in E$ et $c(ts) = -1$.

Il est clair que les flots au sens classique sont en bijection avec les b-flots, ici des circulations, dans le nouveau réseau. De plus, la valeur du flot correspond à l'opposé du coût puisque tout le flux de s vers t revient de t vers s par un arc de coût unitaire -1.

5.1.2.3. *Affectation de coût minimum.* — Les problèmes d'affectations entre ressources et tâches peuvent de formuler comme des problèmes de couplage dans des graphes bipartis.

Problème 5.12 **(Affectation de coût minimum).** — *Étant donnés $G = (U \sqcup V, E)$ un graphe biparti admettant un couplage parfait (cf. definition 2.5) et $c : E \to \mathbb{R}$ une fonction de coût sur les arêtes, trouver un couplage parfait M qui minimise le coût*

$$c(M) = \sum_{e \in E} c(e).$$

Nous pouvons transformer une instance du problème 5.12 en une instance du problème 5.6 comme suit. Dans le graphe G, nous orientons les arêtes de E de U vers V. Nous utilisons comme fonction b la fonction définie par $b(v) = 1$ si $v \in U$ et $b(v) = -1$ si $v \in V$. Nous utilisons comme fonction capacité la fonction u toute à un. La coût de chaque arc est donné par c.

5.2. Critère d'optimalité

Définitions 5.13. — Soient $G = (V, E)$ un graphe orienté, $u : E \to \mathbb{R}_{\geq 0}$ une application capacité et $f : E \to \mathbb{R}$ un flot.

— Pour tout arc $e = uv \in E$, on appelle *arc opposé* l'arc $\overleftarrow{e} = vu$. On note $\overleftarrow{E}$ l'ensemble des arcs opposés de E

3. Il s'agit du problème qui a été introduit dans le cours FMA-0EL01. Notez bien que, dans ce contexte, il n'y a pas de fonction coût.

— On appelle *graphe symétrisé* de G le graphe $\overleftrightarrow{G} = (V, \overleftrightarrow{E})$ où $\overleftrightarrow{E} = E \cup \overleftarrow{E}$. Dans le cas où E possède déjà deux arcs opposés, le graphe $\overleftrightarrow{G}$ est, en toute rigueur, un multigraphe dans lequel il peut y avoir deux arcs parallèles, l'un provenant de E et l'autre de $\overleftarrow{E}$.

— On étend la fonction *coût* au graphe symétrisé par

$$\begin{array}{rcl} \forall e \in E, \quad \overleftrightarrow{c}(e) & = & c(e). \\ \forall e \in \overleftarrow{E}, \quad \overleftrightarrow{c}(e) & = & -c(\overleftarrow{e}). \end{array}$$

— On appelle *capacité résiduelle par rapport au flot f* l'application $u_f : \overleftrightarrow{E} \to \mathbb{R}_{\geq 0}$ définie par

$$\begin{array}{rcl} \forall e \in E, \quad u_f(e) & = & u(e) - f(e) \\ \forall e \in \overleftarrow{E}, \quad u_f(e) & = & f(\overleftarrow{e}) \end{array}$$

— On appelle *graphe résiduel par rapport au flot f* le graphe $G_f = (V, E_f)$ où

$$E_f = \left\{ e \in E \cup \overleftarrow{E} \,;\, u_f(e) > 0 \right\}.$$

— On appelle *chemin f-augmentant*, respectivement *circuit f-augmentant*, tout chemin, respectivement tout circuit, Γ dans le graphe résiduel G_f. La valeur d'un chemin ou d'un circuit augmentant est la quantité

$$\gamma = \min_{e \in \Gamma} u_f(e).$$

qui est strictement positive. Le *coût d'un chemin* est, selon l'habitude,

$$\overleftrightarrow{c}(\Gamma) = \sum_{e \in \Gamma} \overleftrightarrow{c}(e).$$

— Enfin, *augmenter le flot f* le long d'un chemin ou d'un circuit f-augmentant Γ de valeur γ revient à construire un nouveau flot $f' : E \to \mathbb{R}_{\geq 0}$ selon les règles suivantes :

$$\begin{array}{rcl} \forall e \in E \cap \Gamma, \qquad f'(e) & = & f(e) + \gamma \\ \forall e \in E \cap \overleftarrow{\Gamma}, \qquad f'(e) & = & f(e) - \gamma \\ \forall e \in E \smallsetminus (\Gamma \cup \overleftarrow{\Gamma}), \quad f'(e) & = & f(e) \end{array}$$

qui est de coût $c(f') = c(f) + \gamma \cdot \overleftrightarrow{c}(\Gamma)$.

Nous aurons besoin du résultat intermédiaire :

Lemme 5.14. — *Soit $G = (V, E)$ un graphe orienté muni de capacités $u : E \to \mathbb{R}_{\geq 0}$. Soient deux flots f et $f' : E \to \mathbb{R}_{\geq 0}$ pour la même fonction b. On note $g : \overleftrightarrow{E} \to \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ l'application définie par*

$$\begin{array}{rcl} \forall e \in E, \quad g(e) & = & \max\left(f'(e) - f(e), 0\right) \\ \forall e \in \overleftarrow{E}, \quad g(e) & = & \max\left(f(\overleftarrow{e}) - f'(\overleftarrow{e}), 0\right) \end{array}$$

Alors g forme une circulation dans le graphe résiduel G_f de coût

$$c(g) = c(f') - c(f).$$

Démonstration. — TODO □

Théorème 5.15. — *Soit (G, u, b, c) une instance du problème du flot de coût minimum. Un b-flot est de coût minimum si et seulement s'il n'existe pas de circuit f-augmentant de coût strictement négatif.*

Démonstration. — S'il existe un circuit f-augmentant Γ de coût $\overleftrightarrow{c}(\Gamma)$ strictement négatif, nous pouvons augmenter le flot f le long du circuit Γ et obtenir un nouveau b-flot de coût $c(f') = c(f) + \gamma \cdot \overleftrightarrow{c}(\Gamma)$ qui est strictement inférieur au coût $c(f)$. Ceci prouve que le flot f n'est pas de coût minimum.

Réciproquement, supposons qu'il existe un flot f' de coût strictement inférieur à celui de f. D'après le lemme 5.14, nous pouvons construire une circulation g dans le graphe résiduel de coût $c(g) = c(f') - c(f)$ strictement négatif. D'après le lemme 5.4, cette circulation est une combinaison linéaire à coefficient strictement positifs d'indicatrices sur des circuits, à savoir

$$g = \sum_{\chi \in \mathscr{C}} \varphi_\chi \mathbf{1}_\chi,$$

où $\mathscr{C}$ est une famille de circuits et $(\varphi_\chi)_{\chi \in \mathscr{C}}$ est une famille de réels strictement positifs. Mais le coût $c(g)$ vaut $\sum_{\chi \in \mathscr{C}} \varphi_\chi c(\chi)$, ce qui prouve que l'un au moins des circuits χ est de coût strictement négatif, fournissant un circuit f-augmentant de coût strictement négatif. □

Corollaire 5.16. — *Il est possible de vérifier qu'un b-flot est de coût minimum en temps $O(nm)$ (grâce à l'algorithme de Bellman-Ford).*

Afin de résoudre naïvement le problème 5.6, nous pourrions initialiser f avec un b-flot quelconque (grâce à l'algorithme de Ford et Fulkerson) et nous contenter de rechercher des circuits de coût négatif afin d'augmenter f le long de ces circuits jusqu'à ce que la procédure s'arrête. La difficulté consiste à trouver une suite de circuits augmentants efficace afin que la procédure se termine rapidement.

5.3. Algorithme 1 : par élimination de circuits de coût moyen minimaux

5.3.1. Circuits de coût positif. — Nous aurons besoin des résultats classiques suivants.

Définition 5.17. — Soit $G = (V, E)$ un graphe orienté muni d'une fonction coût $c : E \to \mathbb{R}$. Nous appelons *potentiel* toute fonction $p : V \to \mathbb{R}$. Nous définissons le *cout réduit* par rapport au potentiel p comme le nouveau coût $c_p : E \to \mathbb{R}$ où

$$\forall e = uv \in E, \quad c_p(uv) = c(uv) + p(u) - p(v).$$

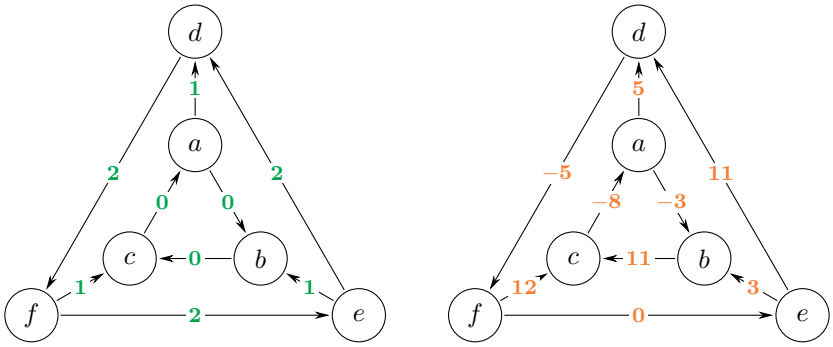
Remarque 5.18. — Un graphe orienté $G = (V, E)$ muni d'une fonction coût $c : E \to \mathbb{R}$ et un potentiel $p : V \to \mathbb{R}$ étant donnés, nous observons que, pour tout circuit χ, les coûts $c(\chi)$ et$c_p(\chi)$ sont égaux. En effet,

$$\sum_{e \in \chi} c_p(e) = \sum_{e=uv \in \chi} c(e) + p(u) - p(v) = \sum_{e \in \chi} c(e).$$

Exemple 5.19. — Dans le graphe suivant, on réduit les coûts par rapport au potentiel p où

$$p(a) = 1 \quad p(b) = 4 \quad p(c) = -7 \quad p(d) = -3 \quad p(e) = 6 \quad p(f) = 4.$$

A gauche, nous présentons le graphe initial (G, c) ; à droite, le graphe modifié (G, c_p).



Définitions 5.20. — Soit $G = (V, E)$ un graphe orienté munit d'une fonction coût $c : E \to \mathbb{R}$ et d'un potentiel $p : V \to \mathbb{R}$. On dit que le coût est *conservatif* s'il n'existe pas de circuit de coût strictement négatif.
On dit que p forme un *potentiel réalisable* si pour tout arc

$$c_p(e) \geq 0.$$

Théorème 5.21. — *Soit $G = (V, E)$ un graphe orienté muni d'une fonction coût $c : E \to \mathbb{R}$. Le graphe (G, c) admet un potentiel réalisable si et seulement si les coûts c sont conservatifs.*

Démonstration. — Si p est un potentiel réalisable, alors, pour tout circuit χ, d'après la remarque, on a

$$0 \leq \sum_{e \in \chi} c_p(e) = c_p(\chi) = c(\chi)$$

ce qui montre que le circuit χ est de coût positif et que c est conservatif.

Inversement, si les coûts c sont conservatifs, on ajoute un sommet frais au graphe G, disons le sommet s et, pour tout sommet $v \in V$, on ajoute des arcs sv de coût nul. Comme G ne possède pas de circuit de coût strictement négatif, il est possible d'exécuter l'algorithme de Moore-Bellman-Ford [4], ce qui détermine une fonction $\ell : V \to \mathbb{R}$ où, pour tout sommet v, $\ell(v)$ est le coût d'un plus court chemin allant de s à v. Or la postcondition de l'algorithme est précisément que ℓ satisfait, pour tout arc $e = uv \in E$,

$$\ell(u) \leftarrow \ell(v) + c(uv)$$

ce qui montre que ℓ est un potentiel réalisable. □

Nous pouvons donc reformuler le théorème 5.15 ainsi :

4. Dans cet algorithme, on initialise une fonction $\ell : V \to \mathbb{R}$ à $+\infty$ et $\ell(s) = 0$. On répète ensuite, pour tout arc $e = uv$, $\ell(v) \leq \min(\ell(v), \ell(u) + c(uv)$ jusqu'à stabilisation.

Corollaire 5.22. — *Soit (G, u, b, c) une instance du problème du flot de coût minimum. Un b-flot est de coût minimum si et seulement si le graphe (G_f, c) admet un potentiel réalisable.*

Remarque 5.23. — En vérité, le lien entre optimalité de f et potentiel réalisable peut encore s'expliquer par de la dualité en optimisation linéaire. Le problème du flot f de coût minimum s'écrit aussi sous la forme du programme linéaire

$$\text{Maximiser} \quad \sum_{i=1}^{k} -c_e f_e \tag{3}$$

$$\text{sujet à} \quad \begin{cases} \forall v \in V, & \displaystyle\sum_{e \in E(v, V \setminus v)} f_e - \sum_{e \in E(V \setminus v, v)} f_e & = & b_v \\ \forall e \in E, & f_e & \geq & u_e \\ \forall e \in E, & f_e & \in & \mathbb{R}_{\geq 0} \end{cases}$$

Le dual de ce programme est

$$\text{Minimiser} \quad \sum_{v \in V} b_v p_v + \sum_{e \in E} u_e z_e \tag{4}$$

$$\text{sujet à} \quad \begin{cases} \forall e = vw \in E, & p_v - p_w + z_e & \geq & -c_e \\ \forall v \in V, & p_v & \in & \mathbb{R} \\ \forall e \in E, & z_e & \in & \mathbb{R}_{\geq 0} \end{cases}$$

Un flot $f^\star$ est optimal si et seulement s'il existe une solution duale réalisable $(p^\star, z^\star)$ telle que $f^\star$ et $(p^\star, z^\star)$ vérifient les écarts complémentaires :

$$z_e(u_e - f_e^\star) = 0 \quad \text{et} \quad f_e^\star(c_e + p_v^\star - p_w^\star + z_e^\star) = 0$$

Donc le flot $f^\star$ est optimal si et seulement s'il existe $(p^\star, z^\star)$ tel que

$$0 = -z_e^\star \leq c_e + p_v^\star - p_w^\star \text{ pour } e = vw \in E \text{ avec } f_e^\star < u_e \text{ et}$$

$$c_e + p_v^\star - p_w^\star = -z_e^\star \text{ pour } e = vw \in E \text{ avec } f_e^\star > 0.$$

Ce qui est équivalent à l'existence d'un $p^\star$ tel que $c_e + p_v^\star - p_w^\star \geq 0$ pour tout arc $e = vw$ appartenant au graphe résiduel $G_{f^\star}$.

La proposition suivante servira abondamment dans des démonstrations à suivre.

Proposition 5.24. — *Soit (G, u, c, b) une instance du problème du flot de coût minimum et $p : V \to \mathbb{R}$ un potentiel. Un b-flot f est de coût minimum pour les coûts c si et seulement s'il est aussi minimum pour les coûts réduits c_p.*

Démonstration. — Pour tout b-flot f, on a

$$\begin{aligned} c_p(f) &= \sum_{e\in E} c_p(e)f(e) \\ &= \sum_{e=uv\in E} (c(e)+p(u)-p(v))\, f(e) \\ &= \sum_{e\in E} c(e)f(e) + \sum_{v\in V} p(v)b(v) \\ &= c(f) + \sum_{v\in V} p(v)b(v) \end{aligned}$$

Comme les coûts $c(f)$ et $c_p(f)$ sont égaux à une constante additive près, un flot est minimum pour c si et seulement s'il est minimum pour c_p. □

5.3.2. Circuit de coût moyen minimum. —

Définition 5.25. — Soit $G=(V,E)$ un graphe orienté, $c : E \to \mathbb{R}$ une fonction coût. Le *coût moyen* du circuit $\chi = (e_1, \dots, e_t)$ est la quantité

$$\frac{c(\chi)}{|\chi|} = \frac{1}{t}\sum_{e\in\chi} c(e).$$

***Problème 5.26* (Circuit de coût moyen minimum).** — *Étant donné un graphe orienté $G=(V,E)$, trouver un circuit χ de coût moyen minimum.*

Notons n le nombre de sommet du graphe $G=(V,E)$. Afin de résoudre le problème 5.26, nous introduisons, pour tout entier $k \in [\![0,n]\!]$ et pour tout sommet v, le nombre $d_k(v)$ qui est le coût minimum d'un chemin terminant en v et utilisant exactement k arcs. La quantité $d_k(v)$ se calcule facilement grâce aux relations

$$\forall v\in V,\quad d_0(v)=0 \qquad \text{et} \qquad \forall k\in\mathbb{N},\quad d_{k+1}(v) = \min_{e=uv\in E} d_k(u)+c(e). \tag{5}$$

Théorème 5.27. — *Le circuit de coût moyen minimum d'un graphe orienté $G=(V,E)$ muni d'une fonction coût $c : E \to \mathbb{R}$ est de coût moyen*

$$\min_{v\in V}\max_{k\in[\![0,n-1]\!]} \frac{d_n(v)-d_k(v)}{n-k}.$$

Démonstration. — (Voir thm 8.10 p 111 dans [**Sch03**] ou 7.13 p 163 dans [**KV12**]) Quitte à translater la fonction coût c d'une constante, on peut supposer que le coût moyen minimum vaut 0. Autrement dit, nous faisons l'hypothèse que tout circuit χ est de coût $c(\chi)$ positif et qu'il existe un circuit, disons χ_0, de coût nul. Nous voulons montrer que

$$M = \min_{v\in V}\max_{k\in[\![0,n-1]\!]} \frac{d_n(v)-d_k(v)}{n-k}.$$

vaut zéro.

Soit v_0 un sommet de V atteignant le minimum M. Notons π un chemin en n arcs terminant en v_0 et de coût $d_n(v_0)$. Comme π possède n arcs, π contient au moins un

cycle, disons χ et on peut noter π' les arcs restants (qui forment un chemin terminant en v_0). Mais alors, en appelant k_0 le nombre d'arcs dans π', on a

$$d_n(v_0) = c(\pi) = c(\pi') + \underbrace{c(\chi)}_{\geq 0} \geq c(\pi') \geq d_{k_0}(v_0).$$

Par conséquent, M est bien positif.

Notons w un des sommets du circuit χ_0 de coût nul. Appelons t son nombre d'arcs. Le minimum $\min_{r\in\mathbb{N}} d_r(w)$ est notamment atteint sur un chemin élémentaire (car les circuits sont de coût ≥ 0) et, quitte à rajouter des copies de χ_0, on peut trouver un chemin formé de r arcs avec $n - t \leq r \leq n$. Mais alors, observons que $n - r \leq t$. Nous coupons le cycle χ_0 en deux chemins : le chemin π formé des $n - r$ arcs de χ_0 en partant de w et allant jusqu'à un certain sommet z et le chemin π' formé des arcs $t - (n - r)$ arcs restants allant de z à w. On a alors d'une part

$$d_n(z) \leq d_r(w) + c(\pi)$$

et d'autre part, pour tout entier k,

$$d_k(z) + c(\pi') \geq d_{k+(t-(n-r))}(w) \geq d_r(w) \geq d_n(z) - c(\pi)$$

Mais ceci implique que

$$d_n(w) - d_k(w) \leq c(\chi_0) = 0,$$

ce qui prouve que $M = 0$. □

On en déduit immédiatement l'algorithme suivant :

Algorithme 5.1 : Algorithme du circuit de coût moyen minimum

Entrées : Graphe orienté $G = (V, E)$ muni d'un coût $c : E \to \mathbb{R}$
Sorties : Circuit de coût minimum

$n \leftarrow |V|$
pour $v \in V$ **faire**
  $d_0(v) \leftarrow 0$
pour $k \in [\![1, n]\!]$ **faire**
  pour $v \in V$ **faire**
    $d_k(v) \leftarrow +\infty$
    pour w *prédécesseur de* v **faire**
      si $d_{k-1}(w) + c(wv) < d_k(v)$ **alors**
        $d_k(v) \leftarrow d_{k-1}(w) + c(wv)$
        $p_k(v) \leftarrow w$
si *pour tout sommet* $v \in V$, $d_n(v) = +\infty$ **alors**
  G est sans circuit
sinon
  Soit $v_0 \in V$ minimisant $\min_{v \in V} \max_{k \in [\![0, n-1]\!]} \frac{d_n(v) - d_k(v)}{n-k}$
  Soit χ un circuit quelconque dans la suite de sommets $p_n(x), p_{n-1}(p_n(x)), p_{n-2}(p_{n-1}(p_n(x))), \ldots,$
  retourner χ

Proposition 5.28. — *L'algorithme 6 calcule correctement un circuit de coût moyen minimum en temps* $O(mn)$.

5.3.3. Élimination du circuit de coût moyen minimum. —

Algorithme 5.2 : Algorithme par élimination du circuit de coût moyen minimum

Entrées : Graphe orienté $G = (V, E)$, capacité $u : E \to \mathbb{R}_{\geq 0}$, offres et demandes $b : V \to \mathbb{R}$ telle que $\sum_{v \in V} b(v) = 0$, coûts $c : E \to \mathbb{R}$.
Sorties : b-flot f de coût $c(f)$ minimum

Trouver un b-flot f
répéter
  Trouver un circuit χ dans le graphe résiduel G_f de coût moyen minimum.
  si $c(\chi) < 0$ **alors**
    Augmenter f le long du circuit χ.
jusqu'à *le circuit* χ *est de coût positif ou n'existe pas* ;
retourner f

Proposition 5.29. — *Dans l'algorithme par élimination (algorithme 6), le nombre d'itérations est au plus* $4nm^2 \lceil \ln n \rceil$.

Démonstration. — Notons $n = |V|$ et $m = |E|$. Appelons f_0 le flot obtenu à la ligne 1 de l'algorithme 6 et appelons f_k le flot après l'étape k. Notons $G_k = (V, E_k)$ le graphe résiduel manipulé après l'étape k. Notons χ_k le circuit utilisé pour passer de G_k à G_{k+1} et γ_k sa valeur. On a donc

$$c(f_{k+1}) = c(f_k) + \gamma_k c(\chi_k).$$

Nous notons $\mu_k = \frac{c(\chi_k)}{|\chi_k|}$ le coût moyen du cycle utilisé à l'étape k. À cause des règles de l'algorithme, ce coût est toujours strictement négatif (Attention, il faudra s'en souvenir dans les inégalités quand on multiplie par μ_k !).

Lemme 5.30. — *On a, pour tout entier k et si les valeurs sont définies,*

1. $\mu_k \leq \mu_{k+1}$
2. $\left(1 - \frac{2}{n}\right) \mu_k \leq \mu_{k+m}$

Preuve du lemme. — La preuve ne dépend par de la valeur de k. Nous détaillons chaque point en se limitant au cas $k = 0$.

1. La valeur μ_0 correspond au plus petit réel tel que, en translatant les coûts c par μ_0, les coûts deviennent conservatifs. Mais alors, d'après le théorème 5.21, il existe un potentiel $p : V \to \mathbb{R}$ tel que

   $$\forall e = uv \in E, \quad c(e) - \mu_0 + p(u) - p(v) \geq 0$$

   La proposition 5.24, nous pouvons raisonner sur le coût réduit c_p. De plus, à cause de la remarque 5.18, nous constatons que l'algorithme s'exécute de manière identique si nous remplaçons les coûts c par les coûts réduits c_p. Bref, nous pouvons raisonner, sans perte de généralité, dans le cas où :

   $$\forall e = uv \in E, \quad c(e) \geq \mu_0. \tag{6}$$

   Il vient de plus que, pour tout arc $e \in \chi_0$, on a l'égalité $c(e) = \mu_0$.

   Les arêtes du graphe $G_1 = (V, E_1)$ viennent soit de E_0, soit de $\overleftarrow{\chi_0}$. Dans le premier cas, nous savons que $c(e) \geq \mu_0$; dans le second, nous avons $c(\overleftarrow{e}) = \mu_0$, donc $c(e) = -\mu_0 \geq \mu_0$ (car $\mu_0 < 0$). Le coût moyen d'un cycle dans G_1 est donc supérieur à μ_0 et on a bien $\mu_1 \geq \mu_0$.

2. Lorsqu'on passe de G_0 à G_m, on supprime toujours au moins un arc du cycle employé pour l'augmentation. Par conséquent, il y a parmi les circuits $\chi_1, \ldots, \chi_m$ au moins deux d'entre eux, disons les circuits χ_h et $\chi_{h'}$, qui contiennent un arc a dans l'un et son opposé $\overleftarrow{a}$ dans l'autre. Nous choisissons h' aussi petit que possible, de sorte que les arcs de $\chi_{h'}$ viennent soit de E_h soit de $\overleftarrow{\chi_h}$. Par une manipulation identique à la preuve du point 1, nous pouvons nous ramener au cas où les arcs de χ_h soient de coût μ_h et celle de $\chi_{h'}$ sont de coût $\geq \mu_h$ en général et de coût $-\mu_h$ pour $\overleftarrow{a}$. Mais alors,

   $$\mu_{h'} = \frac{c(\chi_{h'})}{|\chi_{h'}|} \geq \frac{(|\chi_{h'}| - 1)\mu_h - \mu_h}{|\chi_{h'}|} \geq \left(1 - \frac{2}{n}\right) \mu_h$$

□

Posons $t = nm\lceil \ln n \rceil$. À cause du lemme, en utilisant la majoration classique $\left(1 - \frac{1}{\alpha}\right)^{\alpha} < e^{-1}$ avec $\alpha = \frac{n}{2}$, on a

$$0 > \mu_t \geq \left(1 - \frac{2}{n}\right)^{n\lceil \ln n \rceil} \mu_0 \geq e^{-2\lceil \ln n \rceil} \mu_0$$

et encore, en relâchant la majoration, on peut encore écrire :

$$0 > \mu_t \geq \frac{1}{2n} \mu_0.$$

Lemme 5.31. — *Pour tout indice i, il existe un arc a dans le circuit χ_k qui ne réapparait plus dans les circuit χ_h pour $h \geq k + t$.*

Preuve du lemme. — Ici encore, on se contente de regarder le cas où $k = 0$. Comme précédemment, on se ramène à des coûts réduits, de sorte que l'équation 6 soit vérifiée dans chacun des graphes résiduels. Les arcs de $c(\chi_0)$ sont de coût μ_0 et $\mu_0 \leq 2n\mu_t$ (comme vu plus haut). Nous fixons un tel arc a_0. Supposons que les flots résiduels évoluent : pour un certain $h > t$, $f_h(a_0) \neq f_t(a_0)$. Comme $c(a_0) < 2n\mu_t$, on a en relâchant $c(a_0) < \mu_t$. A cause de l'équation 6, a_0 n'appartient pas au graphe résiduel G_t ce qui se traduit par le fait que $f_t(a_0) = c(a_0)$. On doit donc avoir que $f_h(a_0) < f_t(a_0)$. La fonction $f_t - f_h$ est une circulation. D'après le lemme 5.4, l'ensemble d'arcs E_h contient un circuit χ qui utilise l'arc a_0 tel que E_t contient $\overleftarrow{\chi}$. Mais alors, à cause de l'équation 6, pour tout arc e de χ, $-c(e) = c(\overleftarrow{e}) \geq \mu_t$. Ceci conduit à

$$c(\chi) = c(a_0) + c(\chi \setminus a) < 2n\mu_t - (|\chi| - 1)\mu_t \leq n\mu_t \leq n\mu_h \leq |\chi|\mu_h$$

ce qui contredit la minimalité de μ_t. □

Comme E et $\overleftarrow{E}$ contiennent m arcs, le lemme implique que le nombre d'itérations est au plus $4mt$. □

Théorème 5.32. — *L'algorithme par élimination (algorithme 6) revoie une réponse correcte en temps $O(n^2 m^3 \lceil \ln n \rceil)$.*

5.4. Algorithme 2 : par des courts chemins successifs

Proposition 5.33. — *Soit (G, u, b, c) une instance du problème de flot de coût minimum et f un b-flot de coût minimum. Soit Γ un plus court chemin (au sens des coûts c) allant d'un certain sommet s à un certain sommet t. Soit f' le flot obtenu en augmentant f le long de Γ par une certaine valeur γ. Alors, en posant $b : V\mathbb{R}$ avec*

$$\begin{array}{rcl} \forall v \in V \setminus \{s, t\}, \quad b'(v) & = & b(v) \\ b'(s) & = & b(s) - \gamma \\ b'(t) & = & b(t) + \gamma \end{array}$$

le flot f' est un b'-flot de coût minimum dans (G, u, b', c).

Démonstration. — Comme f est minimum, d'après le théorème 5.15 le graphe résiduel G_f ne contient pas de circuits négatifs pour la fonction coût c. Mais alors d'après le théorème 5.21, il existe un potentiel $p : V \to \mathbb{R}$ réalisable. Quitte à modifier p sur l'ensemble des sommets accessibles depuis s, nous supposons de plus que p coïncide avec $\mathrm{dist}_c(s, v)$ là où cette distance est finie.

Montrons que p est encore un potentiel réalisable dans le graphe $G_{f'}$. Soit $e = uv$ un arc de $G_{f'}$. Si e était déjà un arc dans G_f, rien ne change et $c(e)+p(u)-p(v) \geq 0$. Si e n'était pas déjà un arc dans G_f, alors $\overleftarrow{e}$ est dans Γ, donc $c(\overleftarrow{e}) = p(u) - p(v)$. Mais $c(\overleftarrow{e}) = -c(e)$. Donc $c(e)+p(u)-p(v) \geq 0$ comme voulu. Ceci montrer que p est encore réalisable dans $G_{f'}$, donc que $G_{f'}$ ne possède pas de circuit de coût strictement négatif. Ainsi f' est de coût minimum. □

On en déduit l'algorithme suivant :

Algorithme 5.3 : Algorithme par plus courts chemins

Entrées : Graphe orienté $G = (V, E)$, capacité $u : E \to \mathbb{R}_{\geq 0}$, offres et demandes $b : V \to \mathbb{R}$ telle que $\sum_{v \in V} b(v) = 0$, coûts $c : E \to \mathbb{R}$ conservatifs.

Sorties : b-flot f de coût $c(f)$ minimum

Initialiser $b' \leftarrow b$ et $f \leftarrow 0_{E \to \mathbb{R}}$.
répéter
  Choisir un sommet s avec $b'(s) > 0$
  Choisir un sommet t avec $b'(t) < 0$
  Calculer un plus court chemin Γ de s vers t
  si *t n'existe pas* **alors**
    Il n'y a pas de b-flot.
  Calculer $\gamma \leftarrow \min\left(\min_{e \in \Gamma} u_f(e), b'(s), -b'(t)\right)$
  Augmenter f le long de Γ de γ.
  $b'(s) \leftarrow b'(s) - \gamma$, $b'(t) \leftarrow b'(t) + \gamma$,
jusqu'à *b' est tout à zéro* ;
retourner f

Théorème 5.34. — *Si les capacités u et les demandes ou offres b sont toutes entières, l'algorithme par les plus courts chemins successifs peut s'exécuter en temps $O(nm + (m + n \log n)B)$ où $B = \sum_{v \in V, b(v) > 0} b(v)$.*

Démonstration. — Comme les valeurs de c et b sont entières, la quantité $\sum_{v \in V, b'(v) > 0} b'(v)$ diminue au moins d'une unité à chaque étape. On peut l'utiliser comme variant de boucle. Ainsi, il y a au plus B itérations de la boucle.

La recherche de plus court chemin peut se faire avec l'algorithme de Dijkstra en temps $O(m + n \log n)$ quand il est implémenté avec un tas de Fibonacci. □

TRAVAUX PRATIQUES : FLOTS DE COÛT MINIMUM

Objectif. — Le but de ce TP est d'implémenter l'algorithme de Goldberg et Tarjan d'élimination du cycle de coût moyen minimum et l'algorithme de Busacker et Gowen par les plus courts chemins.

Langage de programmation. — Le TP est proposé en Python. Vous êtes libre de l'adapter à un autre langage de programmation. Si vous êtes mal à l'aise avec les rudiments de programmation orientée objet en Python, vous pouvez consulter les pages consacrées dans un manuel scolaire de lycée (par exemple chapitre 2 de [**CGG+22**]).

Conseils. — Faites beaucoup de tests au fur et à mesure. N'hésitez pas à coder des fonctions qui vérifient les invariants attendus pour vous assurer de la justesse de votre code.

Dans tout le TP, nous supposons que les graphes sont antisymétriques : autrement dit, l'ensemble des arcs ne contient pas un arc et son opposé.

5.5. Mise en place de fonctions utilitaires

Question 5.1. — Initialiser un fichier `flow.py` contenant le code suivant. Un fichier squelette est disponible sur le site du cours E-Campus pour vous aider.

```
95  class Graph():
96      def __init__(self, u, b, c):
97          assert(u.keys() == c.keys())
98          self.V = list(b.keys())
99          self.E = list(u.keys())
100         self.succ = {v : [] for v in V}
101         self.pred = {v : [] for v in V}
102         for (v,w) in self.E:
103             self.succ[v].append(w)
104             self.pred[w].append(v)
105         self.u = u
106         self.b = b
```

```
107         self.c = c
108
109     def is_flow(self):
110         """Permet de controler que f est un b-flot."""
111         assert(0 <= self.f[e] for e in self.E)
112         assert(self.f[e] <= self.u[e] for e in self.E)
113         assert(sum(self.f[e] for e in self.E if e[0]==v or e[1]==v)
            ↪  == self.b[v] for v in self.V)
114
115     def set_flow(self, f):
116         self.f = f
117
118     def get_flow_cost(self):
119         return sum(self.f[e]*self.c[e] for e in self.E)
```

Question 5.2. — Présenter les cas de test donnés ci-après comme instance de la classe `Graph`.

Nous représentons un chemin ou un circuit f-augmentant par une liste de sommets. Dans le cas d'un circuit, on demande que le premier et le dernier sommet de la liste soient identiques.

Question 5.3. — Enrichir la classe `Graph` d'une méthode `get_value(self, p)` qui renvoie la valeur du chemin augmentant. On pourra lever une erreur dans le cas où le chemin n'est pas f-augmentant.

Question 5.4. — Enrichir la classe `Graph` d'une méthode `augment(self, p)` qui augmente le flot f le long du chemin p.

5.6. Algorithme par élimination des circuits de coût moyen minimum

Question 5.5. — Enrichir la classe `Graph` d'une méthode `shortest_paths_to_dest(self)` qui construit les valeurs $d_k(v)$ (voir équation 5) et les prédécesseurs $p_k(v)$ pour les coûts résiduels c_f.

Question 5.6. — Écrire une fonction `find_circuit(p)` qui recherche un circuit dans une liste d'arcs p formant un chemin.

Question 5.7. — Enrichir la classe `Graph` d'une méthode `min_meancost_circuit(self)` qui renvoie un circuit de coût moyen minimum.

Question 5.8. — Enrichir la classe `Graph` d'une méthode `min_cost_flowGT(self)` transforme le flot courant en un flot de coût minimum.

5.7. Algorithme par des chemins augmentants

Question 5.9. — Enrichir la classe `Graph` d'une méthode `shortest_paths_from_src(self, s)` qui explore le graphe résiduel G_f par des plus courts chemins et renvoie, si possible, un plus court chemin de s vers un sommet t tel que $b(t) < 0$.

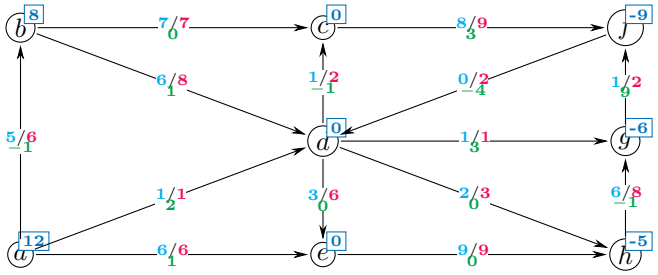
Question 5.10. — Enrichir la classe `Graph` d'une méthode `min_cost_flowBG(self)` transforme le flot courant en un flot de coût minimum en utilisant l'algorithme de Busacker et Gowen (avec des chemins augmentants).

5.8. Pour aller plus loin

Question 5.11. — La suite OR-Tools de Google contient des outils pour résoudre le problème du flot de coût minimum (voir `https://developers.google.com/optimization/flow/mincostflow`). Reprendre la question 5.2 et transformer les cas de test en instance de leur bibliothèque `min_cost_flow`.

5.9. Cas de tests

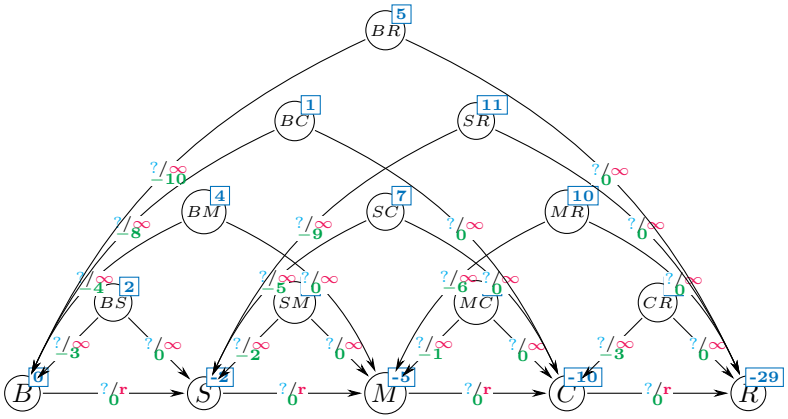
5.9.1. Exemple d'instance du problème du flot de coût minimum. — Dans ce problème de transport, il y a deux sommets d'offre, trois sommets de transbordement et trois sommets de demande.



5.9.2. La péniche le long du Rhin. — Une péniche de capacité r descend le Rhin. Il y a $b_{i,j}$ conteneurs qui sont en attente de transport entre la ville i et j. Le profit du déplacement d'un conteneur entre l'origine et la destination est $c_{i,j}$ par conteneur acheminé. Quel bénéfice maximum le batelier peut-il engranger ?

$b_{i,j}/c_{i,j}$	Strasbourg	Mayence	Cologne	Rotterdam
Bâle	$b_{b,s} = 2$, $c_{b,s} = 3$	4 / 4	1 / 8	5 / 10
Strasbourg		1 / 2	7 / 5	11 / 9
Mayence			2 / 1	10 / 6
Cologne				3 / 3

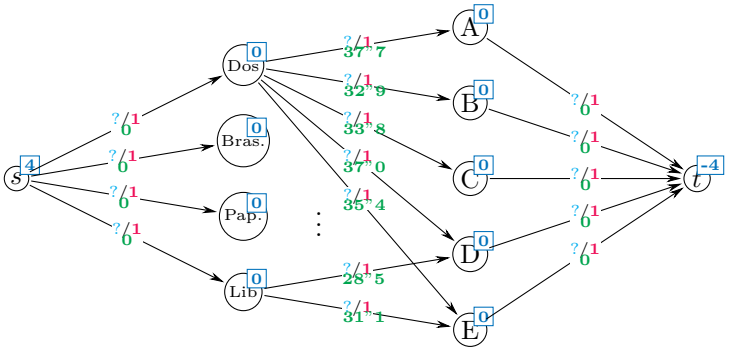
On modélise la situation par le graphe suivant.



5.9.3. Le relais 4 nages. — Dans le relais quatre nages, quatre nageurs se relaient et doivent chacun nager dans une nage différente. Quelle équipe un entraineur doit-il aligner pour le relais 4 nages à partir des temps d'entrainement de ses 5 nageurs ?

	Aya	Badri	Côme	Dalia	Erynn
Dos	37"7	32"9	33"8	37"0	35"4
Brasse	43"4	33"1	42"2	34"7	41"8
Papillon	33"3	28"5	38"9	30"4	33"6
Libre	29"2	26"4	29"6	28"5	31"1

On modélise la situation comme un problème de couplage de cardinal maximum dont le coût est minimum.



CHAPITRE 6

ÉLÉMENTS DE CORRIGÉ

6.1. Couplages

Solution 2.35.

Solution 2.36.

Solution 2.37. 1. L'invariant ($\clubsuit_1$) porte sur la fonction μ qui n'est modifiée que par `_augment(self, x, y)`. D'après la proposition 2.31, les chemins $P(x)$ et $P(y)$ sont alternés, donc $P = P(x) \cup \{(x, y)\} \cup P(y)$ représente bien un chemin M alterné. Les modifications reflètent le changement $M \leftarrow M \triangle P$.

2. À l'initialisation ou après `_augment(self, x, y)`, les arêtes $\{v, \mu(v)\}$ et $\{v, \phi(v)\}$ forment la forêt spéciale de fleurs par rapport à M triviale. Après `_grow(self, x, y)`, la forêt a bien grandi. Il nous faut surtout vérifier la fonction `_shrink(self, x, y)` conserve la structure de forêt de M-fleurs spéciale. Pour cela nous allons montrer que l'ensemble de sommet B, formé de tous les sommets v tels que $\rho(v)$ appartient à $P = P(x) \cup \{x, y\} \cup P(y)$, est une fleur par rapport à M. Nous observons plusieurs points :

   — La fleur de racine r, disons F_0, est laissée inchangée.

   — Les valeurs de la fonction ϕ sont transformées de sorte que, le long des chemins $P = P(x) \cup \{x, y\} \cup P(y)$ (tronqué de la partie dans F_0) le couple (μ, ϕ) forme une oreille M-alternée (dont les extrémités se rattachent à la lfeur de racine r)

   — Pour les autres sommets restant, les valeurs de μ et de ϕ ne changent pas. Ainsi, en partant d'un sommet v ou de son voisin $\phi(v)$ et en suivant le chemin M-alterné $P(v)$ jusqu'à retomber soit dans P soit dans la fleur de racine r, on retrouve une décomposition en oreilles M-alternée, ce qui finit de prouver que B est une fleur.

   Le couplage indiqué par μ n'ayant pas changé, lorsqu'on contracte B (et les autres fleurs qui existaient déjà), on retrouve bien une forêt M-alternée.

3. Le fait que les invariants ($\clubsuit_2$) et ($\clubsuit_3$) soient vérifiés par rapport à F et M découle de ce que nous venons d'expliquer.

Solution 2.38. La correction partielle découle du lemme 2.33 et de la proposition 2.31. Pour que l'algorithme a trouvé un couplage maximum quand il s'arête, nous répétons les arguments déjà vus pour la présentation naïve de l'algorithme d'Edmonds. Soit X l'ensemble des sommets internes et B l'ensemble des bases de fleurs. On a alors

$$|B| = |X| + |V| - 2|M|$$

Mais par ailleurs, les fleurs sont les composantes connexes impaires de $G \setminus X$, donc $q_G(X) = |B|$. Nous avons donc trouvé un couplage qui atteint l'inégalité $|V| - 2\nu(G) \geq q_G(X) - |X|$ (cf proposition 2.7), ce qui prouve l'optimalité de M.

Concernant le temps d'exécution, nous notons que vérifier le statut (hors forêt, interne ou externe) se fait en temps constant grâce à la proposition 2.31. Chacune des trois sous-fonctions (grossir la forêt, augmenter le couplage ou contracter la fleur) s'exécute en temps linéaire. Entre deux augmentations, il ne peut y avoir qu'un nombre linéaire de grossissement ou de contraction (puisque le nombre de points fixes de ϕ décroit strictement). De plus, chaque arête n'est étudiée qu'une seule fois. En conséquence, le temps passé entre deux augmentations est au plus quadratique. Comme il ne peut y avoir qu'un nombre linéaire d'augmentations, l'algorithme se termine en temps cubique.

6.2. Matroïdes

Solution 4.11. $\mathcal{I} = \{\emptyset, \{e_1\}, \{e_2\}, \{e_3\}, \{e_4\}, \{e_1, e_2\}, \{e_1, e_3\}, \{e_1, e_4\}, \{e_2, e_3\}, \{e_2, e_4\}\}$

Solution 4.12. We consider the matrix

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & -1 \end{pmatrix}.$$

Solution 4.25.

1. Let A be a set of tasks and suppose that they can be performed on time in the order
$$t_1, t_2, \ldots, t_n.$$
By definiton, this means that for any $k \leq n$, $k \leq d_k$. Now assume there are two indices $i < j$ such that $d_i > d_j$. Reorder the tasks as
$$t_1, \ldots, t_{i-1}, t_j, t_{i+1}, \ldots, t_{j-1}, t_i, t_{j+1}, \ldots, t_n.$$
We have $d_j \geq j > i$, so t_j will be completed on time. We have $d_i > d_j \geq j$ so t_i will be completed on time in the new schedule too.

   Since any permutation can be decomposed into transposition, this shows that A can be completely ordered by increasing deadlines.

2. Two answers are possible.

   (a) We check axiom by axiom that $([n], \mathcal{I})$ is a matroid. If A is a set of task that can be completed on time and ordered by deadlines, and $A' \subseteq A$, then any task of A' will be done even earlier than in A, so $A' \in \mathcal{I}$.

   Let A and B be in $\mathcal{I}$ with $|B| = |A| + 1$ (this case is enough). We reason by induction. If $A = \emptyset$, it's clear that A can be augmented by the unique element of B. In the general case, we have the two scheduling

A	t'_1	t'_2				t_n	
B	t_1	t_2				t'_n	t_{n+1}

If t_{n+1} is not in A, we can add it to A and it will be done by the deadline. If $t_{n+1} \in A$, there is an index k such that $t_{n+1} = t'_k$. By induction, there is some task τ in $\{t_i\}_{i\leq k}$ such that $\{t_i\}_{i<k} + \tau \in \mathcal{I}$. Now we consider the scheduling

$$t'_1, \dots, t'_{k-1}, \tau, t_{k+1}, \dots, t_n, t_k.$$

which is valid.

(b) We consider the bipartite graph on the set of vertices $([\max_{i\leq n} d_i] \sqcup [n])$ where a left vertex i represent the i-th time slot and a right vertex k represent task t_k. Moreover, we connect a left vertex i with a right vertex k if $i \leq d_k$, which means that it is helpful to perform activity t_k at time i. Now an assignment of tasks is a matching in this bipartite graph. Since we are only interested in the tasks that are done, we exactly recognise a transversal matroid (cf. corollary4.15).

3. We deduce the following greedy algorithm. The variable t stands for the time.

Algorithme 6.1 : Greedy algorithm for task scheduling

Input : List S of tasks $(t_i)_{i\leq n}$ with potential benefits $(p_i)_{i\leq n}$ and deadlines $(d_i)_{i\leq n}$
Output : Sequence of selected task with maximal profit

```
Sort the elements T = {t_1, ..., t_n} such that p_1 ≥ p_2 ≥ ··· ≥ p_n.
S ← ∅
for i to n do
    if S + t_i feasible on time then
        S ← t_i
return S
```

We maintain S as a sorted list. Insertion of an element is $O(\log |S|)$. Since checking if $S \in \mathcal{I}$ requires $|S|$ comparison, the overall cost of the algorithm is in $O(n^2)$. (A naive approach would test all possible orders in $O(n!)$)

Solution 4.26. The information given by all the experiments can be summerised by the system :

$$\begin{cases} p_1 a_{1,1} + \cdots + p_m a_{m,1} &= y_1 \\ p_1 a_{1,2} + \cdots + p_m a_{m,2} &= y_2 \\ & \vdots \\ p_1 a_{1,n} + \cdots + p_m a_{m,n} &= y_n \end{cases}$$

which is a system of n linear equation with m unkowns.

Provided $n \geq m$, our problem has a solution which consists in selecting n equations that are linearly independent. We introduce the linear matroid $\mathcal{M} = (E, \mathcal{I})$ where $E = \{F_1, \ldots, F_n\}$ and $X \subseteq E$ is independent is the corresponding colomn vectors of the matrix $(a_{i,j})$ are linearly independent in $\mathbb{R}^n$ (see the third example of 4.10).

A minimum base is exactly what we are looking for. It can be done with the greedy algorithm. Note that it is easy to check whether a set X belongs to $\mathcal{I}$: one just need to compute the rank of matrix, which a classical question of linear algebra.

Solution 4.27. 1. Let's call $k_{\min}$ and $k_{\max}$ the minimal and maximal number of black elements that can be found in an independent set. It's easy to compute them along with an independent set ($I_{\min}$ and $I_{\max}$) by using the greedy maximum weight algorithm with simply assigning a weight of 0 (respectively 1) to black edges and 1 (respectively 0) to white edges.

   Now let's show that it is possible to construct a base with k black elements for any $k \in [k_{\min}, k_{\max}]$. Start with $I = I_{\min}$. For any element e among the black elements of $I_{\max} \setminus I_{\min}$, add the element e to I, only one circuit C is created (see theorem 4.21), and at least one element of C is not in $I_{\max}$, delete this element. At the end of the procedure, I is an independent set with $k_{\max}$ black elements, and every integer value between $k_{\min}$ and $k_{\max}$ has been covered.

2. The problem is to find a bi-colored spanning tree. We consider the graphic matroid on G. We just need to apply the previous algorithm specialized to graphs.

3. We are back with the matroid that was used in exercise 4.25

Solution 4.36. Suppose we are given an instance $D = (V, A)$, a, b of (Π). Let $\mathcal{M}_1 = (A, \mathcal{I}_1)$, $\mathcal{M}_2 = (A, \mathcal{I}_2)$, $\mathcal{M}_3 = (A, \mathcal{I}_3)$ be the three matroids constructed as follows on the set of edges E of a non-oriented version of D. For $\mathcal{M}_1$, we simply take the graphic matroid on a non-oriented version of D. Now, if $X \subseteq E$, we write $X \cap A^+(v)$ for the set of outgoing arcs in A corresponding to eges of X. We define the following partition matroids

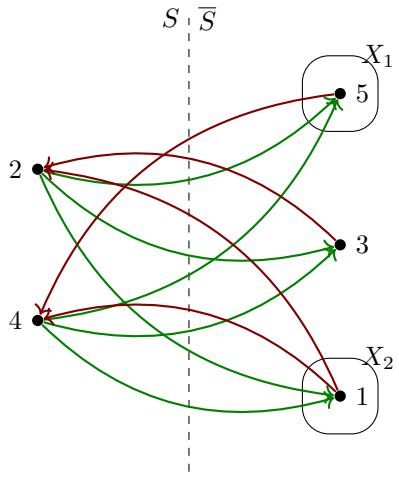
$$\mathcal{I}_2 = \{X \in 2^E;\ |X \cap A^+(b)| \leq 0 \text{ and } \forall v \in V,\ |X \cap A^+(v)| \leq 1\}$$

$$\mathcal{I}_3 = \{X \in 2^E;\ |X \cap A^-(a)| \leq 0 \text{ and } \forall v \in V,\ |X \cap A^-(v)| \leq 1\}$$

We have just encoded the fact that an element of $\mathcal{I}_1 \cap \mathcal{I}_2 \cap \mathcal{I}_3$ is a simple directed path from a to b in D. To answer our decision problem, it suffices to check is a maximal set in this intersection has $|V| - 1$ elements or less.

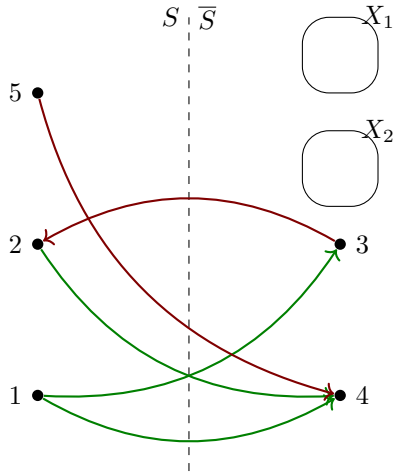
This proves that maximum intersection of three or more matroid is NP-hard.

Solution 4.50. 1. The exchange graph is simply



The shortest path from X_1 to X_2 is $5-4-1$. So we build $S' = S \setminus \{4\} \cup \{1, 5\} = \{1, 2, 5\}$.

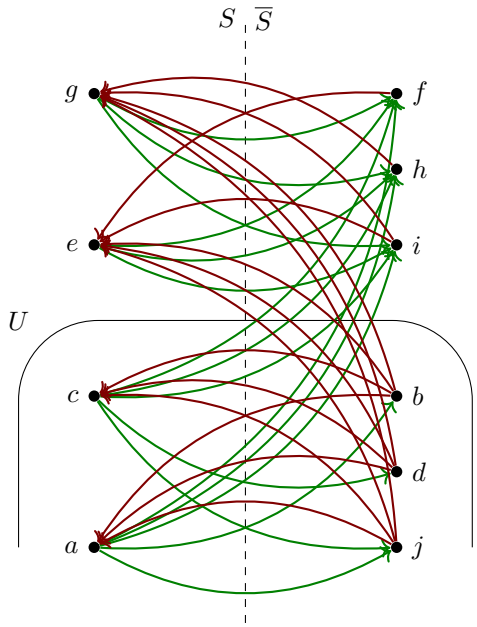
The new exchange graph is



There is nothing to improve. Besides, we could take $U = \emptyset$ as suggested by the algorithm and observe that we have the min-max equality.

2. We could have wrongly selected the path $5 - 4 - 3 - 2 - 1$ and obtained $S' = \{1, 3, 5\}$ which is not $\mathcal{M}_1$-independent.

Solution 4.51. To accelerate the intersection algorithm, we start by greedily increase a set from $\emptyset$ to set in $\mathcal{I}_1 \cap \mathcal{I}_2$. We obtain $S = \{a, c, e, g\}$ Now the exchange graph for S is



$X_1 = \{f, h, i\}$, $X_2 = \{b, d, j\}$

We obtain in particular $U = \{a, c, b, d, j\}$. We note that $\mathrm{rg}_{\mathcal{M}_1}(U) = 2$, $\mathrm{rg}_{\mathcal{M}_2}(\overline{U}) = \mathrm{rg}_{\mathcal{M}_2}(\{g, e, f, h, i\}) = 2$ and the min-max equality is achieved.

Solution 4.52. Let B be the set of all music bands. Let us denote U_i the subset of bands from country C_i and V_j the subset of bands with music style S_j. We clearly have

$$B = \bigsqcup_i U_i = \bigsqcup_j V_j$$

Let $\mathcal{M}_1$ and $\mathcal{M}_1$ be the partition matroids on the first and second partition with

$$\mathcal{I}_1 = \{X \subseteq B,\ |X \cap U_i| \leq c_i\}$$

$$\mathcal{I}_2 = \{X \subseteq B,\ |X \cap V_j| \leq s_j\}$$

With the algorithm 4.2 for matroid intersection, we can compute a maximal set in $\mathcal{I}_1 \cap \mathcal{I}_2$.

Solution 4.53. Let's consider the graphic matroid $\mathcal{M}_1 = (E, \mathcal{I}_1)$ on the complete graph with one node per city. Let T be the set of all available transports, and B be the bipartite graph between E and T with edges $((c, c'), t)$ if the transport t can run between the cities c and c'. Typically, we don't suggest travelling from Madrid to Berlin by boat whereas we think that London New-York by plane is ok. Let $\mathcal{M}_2 = (E, \mathcal{I}_2)$ be the transversal matroid on this graph. Now what we are looking for is a maximal set in $\mathcal{I}_1 \cap \mathcal{I}_2$, which can be found with the algorithm 4.2.

Solution 4.54. We consider the set $E^+ = \bigcup_{\{u,v\} \in E} \{(u, v), (v, u)\}$: E is the set of arcs from E with the two possible orientations of each edge. The first condition we

need to express is the fact that at most, one arc out of a pair is selected. Let's call

$$\mathcal{I}_1 = \{X \subseteq E^+; \forall \{u, v\} \in E,\ |X \cap \{(uv), (vu)\}| \leq 1\},$$

$(E^+, \mathcal{I}_1)$ is a partition matroid. On the other hand, let's call call

$$\mathcal{I}_1 = \{X \subseteq E^+; \forall v \in V,\ |X \cap \delta^-(v)| \leq k_v\},$$

where $\delta^-(v)$ is the set of incoming vertices of E^+ in v. Again $(E^+, \mathcal{I}_2)$ is a partition matroid.

Now deciding if an orientation exists amounts to checking that $\sum_v k_v = |E|$ and that there is a maximum comon independent set in $\mathcal{I}_1 \cap \mathcal{I}_2$ of size $|E|$. This can be done with algorithm 4.2 in polynomial time.

Solution 4.55. Let's take a bipartite graph $G = (V_1 \sqcup V_2, E)$. As usual when it comes to prove a min-max equality, proving that the maximum is at most equal to the minimum is easy. Given a matching M and a vertex cover C, every edge of M is incident to a vertex of M, and no pair of eges is incident to the same summit, so $|M| \leq |C|$.

Now, let's use the two partition matroids introduced in the example 4.32. Let $M \in \mathcal{I}_1 \cap \mathcal{I}_2$ be a maximum common independent set. We apply theorem 4.47 : there is some set $U \subseteq E$ such that

$$|M| = \mathrm{rg}_{\mathcal{M}_1}(U) + \mathrm{rg}_{\mathcal{M}_2}(\overline{U}).$$

But (recall examples 4.20) the $\mathcal{M}_1$-rank of a set of edges U is exactly the number of vertices of V_1 that are incident to U, and similarly the $\mathcal{M}_2$-rank of a set of edges U is the number of incident vertices of V_2. Let C be the set of vertices of $V_1 \cup V_2$ incident to U. Since M is maximal, C is a vertex cover of G with $|C| = |M|$. This concludes the proof.

Solution 4.56. 1. First, suppose that we have k disjoint bases $(B_i)_{i \leq k}$. Then for any set $X \in 2^E$,

$$|E \setminus X| \geq \sum_{\ell=1}^{k} |(E \setminus X) \cap B_i|.$$

But $|(E \setminus X) \cap B_i| = \mathrm{rg}_{\mathcal{M}}(E) - |X \cap B_i|$ and $|X \cap B_i| \leq \mathrm{rg}(X)$. So

$$|E \setminus X| \geq k \cdot (\mathrm{rg}_{\mathcal{M}}(E) - \mathrm{rg}_{\mathcal{M}}(X))$$

It remains to prove the converse inequality. We are going to express the fact that we have disjoint bases as an intersection of 2 matroids $\mathcal{M}_1'$ and $\mathcal{M}_2'$. We start by replicating all the elements of E k times and work on $E' = E \times [k]$. Our first matroid is the direct sum matroid :

$$\mathcal{I}_1' = \{X' \subseteq E';\ \forall i \leq k, \{x \in E; (x, i) \in X'\} \in \mathcal{I}\}.$$

Our second matroid is a partition matroid that specifies that only one element among its copies is selected :

$$\mathcal{I}_2' = \{X' \subseteq E';\ \forall x \in E, \{i \in [k]; (x, i) \in X'\} \leq 1\}.$$

We observe that there are k disjoint bases iff a maximum common independent set S in $\mathcal{I}_1' \cap \mathcal{I}_2'$ is of size $\geq k \cdot \mathrm{rg}_{\mathcal{M}}(E)$.

Theorem 4.47 tells us that

$$|S| = \min_{U \subseteq E'} \mathrm{rg}_{\mathcal{M}'_1}(U) + \mathrm{rg}_{\mathcal{M}'_2}(\overline{U})$$

But for a given U, suppose $(x,i) \in \overline{U} = E' \setminus U$, then, (x,i') belongs to the span of $\overline{U}$ for any $i' \in [k]$. So if we consider

$$U' = \{x \in E;\ \forall i \in [k], (x,i) \notin U\}$$

we have

$$\mathrm{rg}_{\mathcal{M}'_1}(U) + \mathrm{rg}_{\mathcal{M}'_2}(\overline{U}) \geq \mathrm{rg}_{\mathcal{M}'_1}(U') + \mathrm{rg}_{\mathcal{M}'_2}(\overline{U'})$$

Moreover, U' has the shape $U' = X \times [k]$ for $X \in 2^E$ and thus

$$\mathrm{rg}_{\mathcal{M}'_1}(U') + \mathrm{rg}_{\mathcal{M}'_2}(\overline{U'}) = k \cdot \mathrm{rg}_{\mathcal{M}}(X) + |E \setminus X|$$

Using our hypothesis, we get

$$\mathrm{rg}_{\mathcal{M}'_1}(U') + \mathrm{rg}_{\mathcal{M}'_2}(\overline{U'}) \geq k \cdot \mathrm{rg}_{\mathcal{M}}(E)$$

which is what we wanted to prove.

2. The necessity of the condition is quite clear. If we shrink each part into one node, the remaining edges of the disjoint spanning trees need to also contain k edge-disjoint spanning trees. Thus, there are more than $k(\ell - 1)$ edges.

   Finding k disjoint spanning trees is exactly the problem discussed in the previous question specialised to the case of the graphic matroid. In view of what we have just proven, we need to show that for any set $X \in 2^E$,

   $$|E \setminus X| \geq k\left(\mathrm{rg}_{\mathcal{M}}(E) - \mathrm{rg}_{\mathcal{M}}(X)\right).$$

   Remember (cf. examples 4.20) that the rank in this case is just $n - \kappa_G(X)$ where $\kappa_G(X)$ is the number of connected components of X and $n = |V|$. So we need to show

   $$|E \setminus X| \geq k\left(\kappa_G(X) - 1\right).$$

   Let $C_1, \ldots, C_\kappa$ be the connected components of X. We call V_i the set of vertices belonging to C_i. The familly $(V_i)_{i \leq \kappa}$ is a partition of V. Our hypothesis is exactly that

   $$|E \setminus X| \geq k\left(\kappa - 1\right),$$

   which we wanted to show.

6.3. Flots

INDEX

BIBLIOGRAPHIE

[Bie15] M. Bierlaire – *Optimization : Principles and Algorithms*, EPFL Press, Lausanne, 2015.

[CGG+22] C. Chevalier, G. Grimaud, B. Groz, P. Marquet, M. Nancel, C. Pelsser, X. Redon, T. Vantroys & E. Waller – *Numérique et Sciences de l'informatique - Spécialité Terminale*, Éditions Hachette Éducation, 2022, Consultable sur `https://mesmanuels.fr/acces-libre/9782017189992`.

[CH19] I. Charon & O. Hudry – *Introduction à l'optimisation continue et discrète*, Lavoisier, 2019.

[KV12] B. Korte & J. Vygen – *Combinatorial optimization : Theory and algorithms*, 5th éd., Springer Publishing Company, Incorporated, 2012.

[Law01] E. Lawler – *Combinatorial optimization*, Dover Publications Inc., Mineola, NY, 2001, Networks and matroids, Reprint of the 1976 original.

[Lee04] J. Lee – *A first course in combinatorial optimization*, Cambridge Texts in Applied Mathematics, Cambridge University Press, Cambridge, 2004.

[LP09] L. Lovasz & M. D. Plummer – *Matching theory*, American Mathematical Society, Providence, 2009.

[Sch03] A. Schrijver – *Combinatorial Optimization - Polyhedra and Efficiency (3 volume, a,b, & c)*, 1st éd., Springer, Berlin, 2003.