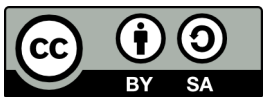
Optimisation discrète

INSA Rennes, 3MA

J. Omer, R. Texier-Picard*

Mars 2024



*INSA Rennes, rozenn.texier-picard@insa-rennes.fr

Contents

1 Introduction générale

Définition 1.1. Problème d'optimisation discrète

Un problème d'optimisation discrète consiste à rechercher le minimum d'une fonction de coût $f : X \to \mathbb{R}$ où X est un ensemble discret (autrement dit, tout point de X est isolé). Le plus souvent, on pourra supposer que X est fini, mais que son cardinal ne permet en général pas d'énumérer ses éléments en un temps raisonnable.

Ce cadre permet de capturer une partie prépondérante des problèmes d'optimisation rencontrés par les industriels, le secteur public et chacun d'entre nous, à titre personnel. Il permet par exemple de formaliser des questions telles que

- quelle tâche dois-je réaliser pour me préparer à aller en cours et dans quel ordre dois-je les effectuer ?
- quel moyen de transport et quel itinéraire suivre pour me rendre chez mes grands-parents ?
- comment construire un réseau de transport urbain minimisant les embouteillages ?
- comment utiliser au mieux les ouvriers et les machines d'une usine de fabrication de voitures ?

De façon générale, nous nommerons *graphe*, un ensemble de points et un ensemble de traits reliant des paires de points pour souligner une relation entre eux. Les graphes sont un outil de modélisation puissant, car ils donnent des possibilités de visualisation efficaces et permettent une plus grande prise à l'intuition. Dans ce cours, nous les utiliserons pour l'optimisation discrète, mais leur portée est bien plus vaste. On les retrouve par exemple en statistiques pour la modélisation probabiliste par des réseaux bayésiens, en apprentissage pour les méthodes de réseaux de neurones, mais aussi pour la représentation de notre généalogie (l'arbre généalogique), de nos réseaux sociaux ou de la fréquence d'apparition conjointes de mots en linguistique, etc.

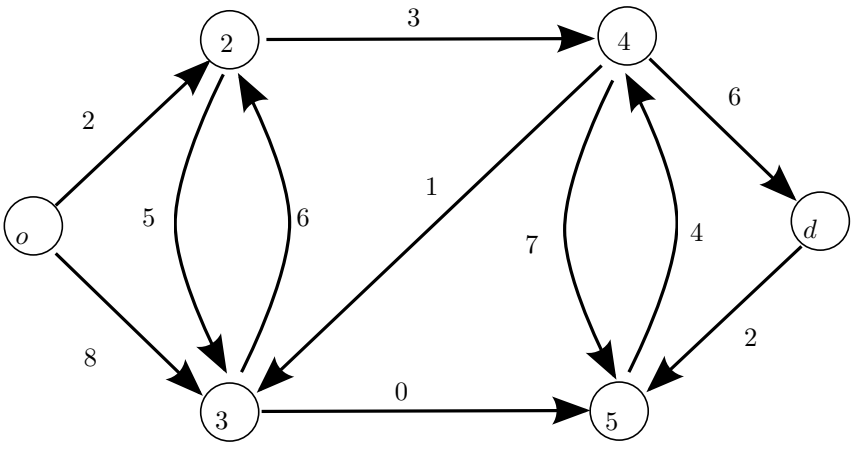


Figure 1: Quel est le plus court chemin de o à d?

Exemple 1 (Plus court chemin d'un point o à un point d). *La recherche systématique du plus court chemin dans un graphe sera vue pendant le 2ème CM, et nous l'illustrerons sur ce graphe lors du*

TD associé. Outre cette modélisation graphique du problème, il est également possible de considérer la modélisation algébrique suivante.

$$\begin{cases} \textit{min.} & \displaystyle\sum_{(i,j)\in A} l_{ij}x_{ij} \\ \textit{sous} & \displaystyle\sum_{(i,j)\in A} x_{ij} = \sum_{(j,i)\in A} x_{ji},\ \forall i \notin \{o,d\} \\ & \displaystyle\sum_{(o,j)\in A} x_{oj} = 1 \\ & \displaystyle\sum_{(i,d)\in A} x_{id} = 1 \\ & x_{ij} \in \{0,1\},\ \forall (i,j) \in A, \end{cases}$$

où A est l'ensemble des arcs, l_{ij} la longueur d'un arc (i,j) et x_{ij} est une variable de décision *prenant la valeur 1 si l'arc (i,j) est sur le chemin choisi et 0 sinon. Ce type de modèle est appelé* programme linéaire en nombres entiers *(PLNE). Comme vous devez le constater, la compréhension de ce formalisme est moins immédiate qu'un modèle graphique. Il a toutefois son intérêt, puisqu'il permet l'utilisation de codes informatiques très puissants pour ce type de modèles. Une introduction à la modélisation par PLNE sera donnée lors du dernier CM et du TP associé. L'étude approfondie des méthodes de résolution associées constituera le cœur du cours Recherche Opérationnelle de 4ème année.*

Exemple 2 (**Organisation d'une session d'examens).** *Les étudiants d'une promotion sont séparés en six groupes A, B, C, D, E, F, G et H. Ils doivent passer des examens dans différentes disciplines, chaque examen occupant une demi-journée.*

- *Chimie : Groupes A et B*
- *Electronique : Groupes C et D*
- *Informatique : Groupes C, E, F et G*
- *Mathématiques : Groupes A, E, F, et H*
- *Physique : Groupes B, F, G et H*

Afin d'accaparer l'espace dédié aux examens aussi peu de temps que possible, on cherche à minimiser la durée de la session d'examens.

Pour modéliser le problème, on peut représenter les disciplines par des sommets *et relier par des* arêtes *les sommets correspondant à des paires d'examens ayant au moins un groupe d'étudiants en commun. Représenter ce graphe.*

Le problème peut alors être traité en recherchant une façon de colorier tous les sommets du graphe avec un minimum de couleurs tout en s'assurant que des sommets voisins aient des couleurs différentes. On peut donc voir ce problème comme une application du célèbre problème de coloriage de graphe dont l'application première a été l'attribution de couleurs à des pays (ou à des régions) sur une carte géographique.

Exemple 3 (**Gestion d'un projet industriel.).** *Pour la réalisation d'un projet de recherche opérationnelle, il est nécessaire de:*

- *rédiger le cahier des charges (2 jours),*

- *effectuer une revue de littérature (3 jours),*
- *modéliser le problème (1 jour),*
- *adapter une méthode classique au problème traité (3 jours),*
- *rechercher des jeux de données pour les tests (2 jours),*
- *coder l'importation des jeux de données (1/2 jour),*
- *coder le squelette du code informatique (2 jours),*
- *coder la méthode de résolution (2 jours),*
- *effectuer les tests informatiques (3 jours),*
- *rédiger le rapport (2 jours).*

Le chef de projet dispose de suffisamment d'ingénieurs pour paralléliser les tâches du projet autant que possible. Toutefois, toutes les tâches ne peuvent pas être effectuées en même temps

- *il faut faire la revue de littérature avant le modèle,*
- *le modèle doit être développé avant la méthode de résolution,*
- *on ne peut pas coder la méthode avant de l'avoir déterminée,*
- *les tests ne peuvent pas être effectués avant que la méthode ne soit codée et les jeux de données importés,*
- *le squelette du code est nécessaire pour coder la méthode,*
- *il faut disposer des jeux de données pour pouvoir les importer.*

En revanche, on supposera que le cahier des charges et le rapport peuvent être rédigés en parallèle du reste.

Le chef de projet se demande alors quelle est la durée minimum sur laquelle son projet va s'étaler. Pour répondre à cette question, on peut représenter les étapes du projet sur un graphe dont chaque sommet correspond à une tâche et chaque relation de précédence est matérialisée par un arc étiqueté par la durée minimale séparant les deux tâches associées. ***Donner la représentation graphique du problème décrit ci-dessus.*** *Pour compléter le graphe, il est nécessaire d'ajouter un sommet débutà partir du duquel part un arc vers chaque tâche sans prédécesseur, et un sommet finvers lequel arrive un arc à partir de chaque tâche sans successeur.*

Dans ce graphe, chaque chemin est une suite de tâches du début à la fin du projet qui satisfait les contraintes de précédence. De plus, la durée du projet ne peut pas être inférieure à la somme des étiquettes sur les arcs d'un chemin (i.e. la longueur du chemin). Par conséquent, le plus long chemin du graphe donne la durée minimum du projet.

2 Graphes : définitions et concepts de base

2.1 Graphes : concepts orientés

Définition 2.1. Graphe orienté

Un graphe orienté $G = [S, A]$ est déterminé par :

- un ensemble S dont les éléments sont appelés *sommets* ou *nœuds*,
- un ensemble A dont les éléments sont des couples de sommets (i, j) appelés *arcs*. Le sommet initial, i, d'un arc est aussi appelé *origine* et le sommet terminal, j, est le *but*.

Graphiquement, les sommets d'un graphe sont généralement représentés par des points et les arcs sont représentés par des flèches allant de l'origine vers le but.

Notez qu'il est possible qu'un graphe contienne plusieurs arcs (i, j) entre deux mêmes sommets i et j. Cela peut par exemple être utilisé pour représenter un réseau de transport urbain où plusieurs moyens de transport différents peuvent être utilisés pour voyager d'un point à un autre (mais avec des coûts et des durées potentiellement différents). On parlera alors de p-graphe, où p est le nombre maximum d'arcs entre deux mêmes sommets. Si l'on veut appuyer sur le fait qu'il n'existe pas deux arcs ayant la même origine et le même but, on dira que G est un 1-graphe.

Définition 2.2. Successeur, prédécesseur, boucle, graphe simple

S'il existe un arc $(i, j) \in A$, on dit que i est un *prédécesseur* de j et j est un *successeur* de i.
Un arc $a = (i, i)$ est appelé une *boucle*. Si G est un 1-graphe et A ne contient aucune boucle, on dit que G est un graphe *simple*.

2.2 Graphes : concepts non-orientés

Lorsque le lien entre deux sommets ne nécessite pas de faire la distinction entre origine et but (cf l'exemple sur l'organisation d'une session d'examens), on parle alors de graphe *non-orienté*. Dans ce cas, les éléments de A sont appelés des *arêtes*. Graphiquement, une arête est simplement représentée par un segment entre deux sommets.

Pour marquer la distinction entre arcs et arêtes, une arête est notée comme un ensemble de deux sommets $\{i, j\}$ plutôt que comme un couple (i, j). Cela insiste sur le fait qu'une arête n'est pas ordonnée. Pour une arête $\{i, j\}$, les sommets i et j sont ses deux *extrémités*. Un graphe non-orienté est *simple* si et seulement si chaque paire de sommets est reliée par au plus une arête et que le graphe ne contient pas de boucle.

Définition 2.3. Graphe non-orienté sous-jacent

À partir d'un graphe orienté, on définit le *graphe non-orienté sous-jacent* en supprimant l'orientation de ses arcs. Réciproquement, on pourra munir un graphe non-orienté d'une *orientation* pour en faire un graphe orienté.

Dans la suite, on précisera si l'on se place dans le cas orienté ou non-orienté seulement lorsque cela ne pourra pas être déduit immédiatement du contexte. Par exemple, si l'on parle d'arcs, on pourra immédiatement déduire que l'on s'intéresse à un graphe orienté.

2.3 Principales définitions

Définition 2.4. Adjacence, incidence et degré

Deux sommets sont dits *adjacents* s'ils sont liés par un arc/arête. Plus simplement, on peut aussi dire que deux sommets sont *voisins*.
Un arc/arête est *incident* aux deux sommets qu'il relie. Pour un sous-ensemble $S' \subset S$, on définit:

$\delta^+(S') =$ ensemble des arcs ayant leur origine dans S'.

$\delta^-(S') =$ ensemble des arcs ayant leur but dans S'.

L'ensemble des arcs/arêtes incidents à un sommet de S' est noté $\delta(S')$ $(= \delta^+(S') \cup \delta^-(S')$ dans le cas orienté).
Le *demi-degré extérieur* du sommet i, noté $d_G^+(i)$, est le nombre d'arcs ayant i pour origine. Le *demi-degré intérieur* du sommet i, noté $d_G^-(i)$, est le nombre d'arcs ayant i pour but. Le *degré* du sommet i, noté $d_G(i)$, est le nombre d'arcs/arêtes ayant i pour extrémité. Dans un graphe orienté, on a $d_G(i) = d_G^-(i) + d_G^+(i)$.

Notez que parmi les définitions données ci-dessus, $\delta^+(S')$, $\delta^-(S')$, $d_G^+(i)$ et $d_G^-(i)$ n'ont de sens que dans un graphe orienté.

Définition 2.5. Symétrie dans les graphes

Un graphe orienté est *symétrique* si pour toute paire de sommets $\{i, j\}$, il y a autant d'arcs (i, j) que d'arcs (j, i).
Un 1-graphe orienté est *antisymétrique* si $(i, j) \in A \implies (j, i) \notin A$.

Exercice 1. *On s'intéresse à nouveau au graphe représenté sur la figure 1.*

i Donner le demi-degré intérieur du sommet 4, le demi-degré extérieur du sommet 5 et le degré du sommet 2.

ii Donner $\delta^+(o)$, $\delta^-(o)$ et $\delta(5)$.

iii Quel est le degré du sommet 3 dans le graphe non-orienté sous-jacent à ce graphe?

iv Ce graphe est-il symétrique ? Anti-symétrique ?

v Ce graphe est-il simple ? Son graphe non-orienté sous-jacent est-il simple ?

Définition 2.6. Sous-graphes et graphe complémentaire

Le *sous-graphe restreint* à un sous-ensemble d'arcs/arêtes A' est donné par $G' = (S, A')$.
Le *sous-graphe induit* par un sous-ensemble de sommets S' est le graphe dont l'ensemble des sommets est S' et dont les arcs/arêtes sont ceux ayant leurs **deux** extrémités dans S'.
Le *graphe complémentaire* d'un graphe simple $G = [S, A]$ est le graphe $\overline{G} = [S, \overline{A}]$ où

$$(i, j) \in \overline{A} \Leftrightarrow (i, j) \notin A.$$

(Remplacer les arcs (i, j) par des arêtes $\{i, j\}$ pour le cas non-orienté.)

Exemple 4. *Supposons que G décrive le réseau routier Européen et que (S', A') correspond au réseau routier Français. Le sous-graphe de G restreint à A' contient une carte des villes Européennes, mais n'inclue que les routes françaises. Le sous-graphe de G induit par S' correspond uniquement à une carte routière de la France.*

Définition 2.7. Graphes complets et cliques

Un graphe non-orienté $G = (S, A)$ est *complet* si pour toute paire de sommets distincts i et j de S, A contient l'arête $\{i, j\}$.
Si un sous-graphe induit par un sous-ensemble de sommets $K \subset S$ est complet on dit que K est une *clique* de G.

Les concepts de graphe complet et de clique sont le plus souvent utiles pour des graphes non-orientés, mais on peut tout de même les étendre au cas orienté en définissant un graphe orienté complet comme un graphe dont le sous-graphe non-orienté sous-jacent est complet.

Exercice 2. Graphes complets et cliques
Le graphe ci-dessous est-il complet ? Sinon, comment pourrait-on le modifier pour qu'il le devienne ?
Donner une clique de cardinalité maximum dans ce graphe.

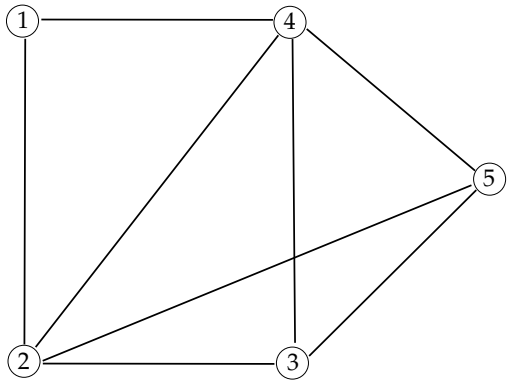


Figure 2: Graphe complet ? Cliques ?

Nous pouvons désormais passer aux notions qui nous permettront d'effectuer les opérations sur les

graphes qui semblent les plus naturelles à beaucoup d'entre nous. Il est en effet facile de voir une carte de transport comme un graphe. Dès lors, on y cherchera naturellement des itinéraires.

Définition 2.8. Chemin, chaîne et connexité

i Un *chemin* de *cardinalité* l est une suite d'arcs $C = (a_1, \ldots, a_l)$ telle que pour tout $i = 1, \ldots, l-1$ le but de a_i coïncide avec l'origine de a_{i+1}.

Dans le cas d'un graphe non-orienté, on parlera de *chaîne*. Pour qu'une suite d'arêtes $C = (a_1, \ldots, a_l)$ soit une chaîne, il suffit alors que pour tout $i = 2, \ldots, l-1$, a_i ait une extrémité commune avec a_{i-1} et l'autre commune avec a_{i+1}.

ii On parle de chemin/chaîne *élémentaire* lorsque la suite d'arcs/arêtes associée ne passe jamais deux fois par le même sommet.

iii Un graphe non-orienté est *connexe* si pour toute paire de sommets du graphe, il existe une chaîne joignant i et j. Un graphe orienté est *connexe* si son graphe non-orienté sous-jacent est connexe.

iv Un graphe orienté est *fortement connexe*, si pour toute paire de sommets i et j, il existe un chemin joignant i et j.

On peut noter que dans le cas d'un graphe simple, un chemin ou une chaîne est parfaitement définie à l'aide de la suite des sommets qu'il ou elle rencontre $(i_0, i_1, \ldots, i_l)$. Comme pour un arc, on parle d'*origine* et de *but* d'un chemin $C = (a_1, \ldots, a_l)$ pour désigner ses sommets initiaux et finaux (autrement dit, l'origine de a_1 et le but de a_l). Sans orientation, les notions de but et d'origine n'ont pas de sens, donc on parle simplement d'*extrémités* d'une chaîne.

Par ailleurs, si un graphe n'est pas connexe, on peut identifier un ensemble de sous-graphes induits connexes maximaux pour l'inclusion (c'est-à-dire les plus grands sous-graphes connexes). Ces sous-graphes sont les *composantes connexes* du graphe.

Définition 2.9. Circuit, cycle et arbre

i Un *circuit* (resp. un *cycle*) est un chemin (resp. une chaîne) dont les extrémités coïncident.

ii Un circuit/cycle est *élémentaire* s'il ne rencontre jamais deux fois le même sommet (sauf celui qui se trouve à ses deux extrémités).

iii Un graphe non-orienté est *acyclique* s'il ne contient aucun cycle.

iv Si G est un graphe acyclique et connexe, on dit que G est un *arbre*. S'il est acyclique, mais non-connexe, ses composantes connexes forment une *forêt* (chaque composante connexe est un arbre).

Remarque 1. *On peut également voir un cycle/circuit élémentaire comme un cycle/circuit minimal pour l'inclusion, dans le sens qu'il ne contient strictement aucun autre cycle/circuit.*

Par ailleurs, un cycle/circuit élémentaire est un sous-graphe dont tous les sommets sont de degré 2.

Proposition 2.10. Caractérisation des arbres

Soit $G = (S, A)$ un graphe non-orienté. Le graphe G est un arbre si et seulement si G est connexe et que $|A| = |S| - 1$.

Démonstration : *Par récurrence sur le nombre de sommets $n = |S|$. C'est évidemment vrai si $n = 1$. On suppose que c'est vrai pour $n \geq 1$ et l'on considère un graphe $G = (S, A)$ où $|S| = n + 1$.*

On suppose d'abord que G est un arbre et l'on montre qu'il contient alors un sommet degré égal à 1. Pour cela, on part d'un sommet arbitrairement choisi i_1. S'il est de degré 1, on a fini, sinon on construit le début d'une chaîne en allant vers un voisin, i_2, de i_1. Si i_2 est de degré 1, on a fini, et sinon on est en mesure d'étendre la chaîne vers un voisin de i_2 autre que i_1. Ce processus se termine au plus tard après avoir ajouté $n + 1$ sommets dans la chaîne, sinon on passe deux fois par le même sommet, ce qui est impossible dans un arbre (car acyclique).
Notons alors i ce sommet de degré 1. Si l'on retire i de G ainsi que l'arête incidente à i, on obtient un graphe connexe (S', A') tel que $|S'| = n$, donc $|A'| = |S'| - 1$ par l'hypothèse de récurrence.

Réciproquement, si $|A| = |S| - 1 = n$ et que G est connexe, il existe nécessairement un sommet de degré 1 (pour G connexe, tous les sommets sont de degré au moins 1 et la somme des degrés vaut deux fois le nombre d'arêtes). On peut donc à nouveau utiliser l'hypothèse de récurrence pour conclure en supprimant ce sommet du graphe.

Exercice 3. Chemins et chaînes
On s'intéresse au graphe G dessiné en Figure 3 :

i Trouver un chemin d'origine o et de but d de cardinalité minimum.

ii Trouver une chaîne d'extrémités o et d de cardinalité minimum.

iii Le graphe est-il connexe ? Fortement connexe ?

iv Créer un graphe G' en supprimant tous les arcs dont une extrémité est dans $\{2, 3\}$ et l'autre dans $\{4, 5\}$. Donner les composantes connexes de G'.

v Donner un circuit élémentaire de cardinalité maximum dans G.

vi Supprimer des arcs de G jusqu'à ce que son graphe sous-jacent non-orienté soit un arbre.

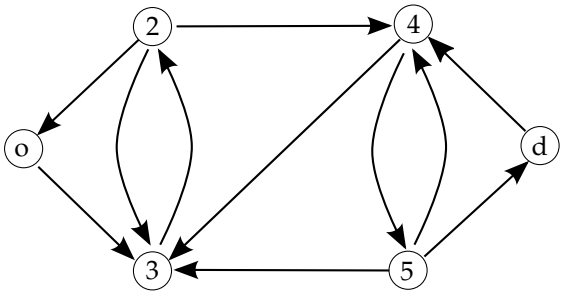


Figure 3: Chemins et chaînes

Exercice 4. Chemins et chaînes, bis

On considère le graphe orienté $G = (S, A)$ de la figure 4.

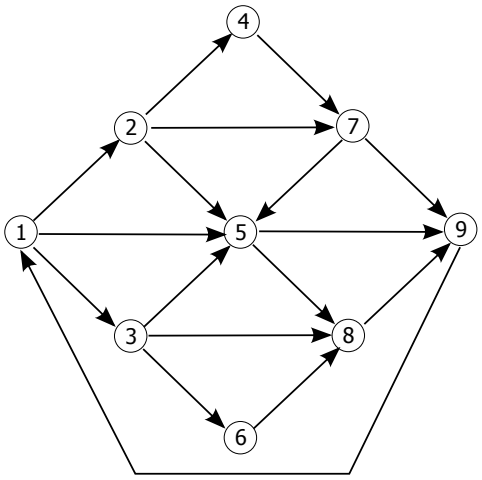


Figure 4: Exemple pour l'exercice 4

i Donner la liste des degrés entrants et des degrés sortants des trois premiers sommets. Quelle relation existe-t-il entre la somme des degrés entrants et des degrés sortants à l'échelle du graphe ?

ii Donner un chemin élémentaire de G contenant six arcs ainsi qu'une chaîne élémentaire du graphe non-orienté sous-jacent contenant huit arêtes.

iii Déterminer un circuit élémentaire formé de sept arcs et un cycle élémentaire du graphe non-orienté sous-jacent à G formé de neuf arêtes.

iv Un arbre couvrant pour un graphe non-orienté est un sous-graphe restreint à un sous-ensemble d'arêtes qui est un arbre (on doit conserver tous les sommets). Déterminer un arbre couvrant du graphe non-orienté sous-jacent à G ayant six feuilles (une feuille d'un arbre est un sommet de degré 1).

v Le graphe G est-il fortement connexe ?

2.4 Représentation d'un graphe

Hormis la donnée d'une liste de sommets et d'une listes d'arcs/arêtes, il existe plusieurs façons de représenter un graphe dont l'utilité est intimement liée aux opérations que l'on souhaite réaliser dans le graphe. On décrit succinctement les principaux modes de représentation des graphes ci-dessous. Le projet sera notamment l'occasion de vous poser la question d'une représentation efficace d'un graphe. Pour la suite, on note $n = |S|$, le nombre de sommets du graphe et $m = |A|$, le nombre d'arcs/arêtes du graphe.

Un premier mode de représentation des graphes est matriciel. Pour cela, on numérote les arcs/arêtes de 1 à m et les sommets de 1 à n. On peut représenter un graphe orienté par une *matrice d'incidence*, $M \in \mathbb{R}^{n\times m}$, telle que

- $M_{ij} = 1$ si le sommet i est l'origine de l'arc j,
- $M_{ij} = -1$ si le sommet i est le but de l'arc j,
- $M_{ij} = 0$ si le sommet i n'est pas une extrémité de l'arc j.

Dans les lignes, les valeurs 1 et -1 repéreront donc les arcs dont les sommets sont une extrémité. On pourra aussi utiliser cette représentation pour les graphe non-orientés en mettant des 1 dans l'élément M_{ij} si le sommet i est une extrémité de l'arête j.

Lorsque G est un 1-graphe, on peut le représenter par une *matrice d'adjacence*, $M \in \mathbb{R}^{n\times n}$. Dans le cas orienté (non-orienté) $M_{i,j}$ vaut 1 si $(i,j) \in A$ ($\{i,j\} \in A$) et 0 sinon. Ainsi, la matrice d'adjacence associée à un graphe non-orienté est nécessairement symétrique.

Le second mode de représentation fait appel à des *listes d'incidence ou d'adjacence*. La représentation par liste d'incidence d'un graphe orienté $G = (S, A)$ se fait par l'énumération des ensembles $\delta^+(i)$ (ou $\delta^-(i)$ selon les besoins de l'application) pour tout $i \in S$. Dans le cas non-orienté, on énumère simplement $\delta(i)$ pour tout $i \in S$.

Dans la représentation par listes d'adjacence, dans le cas orienté, on énumère les ensembles de successeurs (ou de prédécesseurs) de tous les sommets. Dans le cas non-orienté, on énumère les ensembles de voisins de chaque sommet.

Dans le choix de la représentation, il faut garder à l'esprit que les représentations par liste sont beaucoup plus compactes, et qu'elles permettent un accès très rapide à certaines informations. Par exemple, si l'on exécute un algorithme qui s'intéresse uniquement aux successeurs des sommets, le recours à une liste d'adjacence permet d'accéder aux successeurs de façon immédiate (i.e. sans parcourir tous les éléments de lignes ou de colonnes d'une matrice d'adjacence ou d'incidence). Toutefois, la représentation matricielle peut avoir un intérêt, car elle permet de recourir au calcul matriciel dans certaines applications. C'est par exemple le cas dans la proposition suivante.

Proposition 2.11. Matrice d'adjacence itérée

Soit M la matrice d'adjacence d'un 1-graphe orienté, et $p \in \mathbb{N}^*$, alors $(M^p)_{ij}$ est égale au nombre de chemins différents de longueur p ayant i pour origine et j pour but (ici M^p correspond bien à M élevée à la puissance p pour le produit matriciel usuel).

Démonstration : *La démonstration se fait par récurrence. Le résultat est vrai par définition de la matrice d'adjacence pour $p = 1$. S'il est vrai pour p, on écrit que $(M^{p+1})_{ij} = \sum_{k=1}^{n} (M^p)_{ik} M_{kj}$. Les chemins de longueur $p+1$ allant de i à j sont des chemins de longueur p de i à k, $(k,j) \in \delta^-(j)$,*

(c'est-à-dire que k est un prédécesseur de j) prolongé par l'arc (k, j). Par l'hypothèse de récurrence, on sait que pour $k \in S$, il y a $(M^p)_{ik}$ chemins distincts de longueur p allant de i à k. Si $M_{kj} = 1$, il y a donc $(M^p)_{ik}$ chemins de i à j de longueur $p+1$ passant par k juste avant j, et il n'y en a aucun si $M_{kj} = 0$. En faisant la somme sur les sommets k, on obtient bien le nombre total de chemins de longueur $p+1$ entre i et j.

Exercice 5. Représentations de graphes.
Donner les représentations par matrice / liste d'incidence / d'adjacence du graphe de la figure 3.

2.5 Algorithmes de parcours

Le parcours d'un graphe consiste à se déplacer sur les sommets de ce graphe en suivant ses arcs ou arêtes selon des règles fixées arbitrairement. En pratique, le parcours d'un graphe servira à déterminer un ordre sur ses sommets. Cela pourra être utile par exemple si une action doit être effectuée sur chaque sommet du graphe et que l'ordre de traitement a une importance.

2.5.1 Parcours des arbres

Dans la mesure où les arbres sont connexes et acycliques, les arbres sont les graphes pour lesquels le parcours est le plus simple à effectuer. En effet la connexité implique qu'il y a au moins une chaîne entre chaque paire de sommets, puis, en l'absence de cycles, on sait qu'il y a exactement une chaîne entre chaque paire de sommets.

À partir d'un sommet arbitrairement choisi, r, on peut définir une orientation de tout arbre tel qu'il existe un chemin de r vers chaque autre sommet du graphe.

Un arbre muni d'une orientation est appelé une *arborescence*, et le sommet r est alors la *racine* de l'arborescence. Le *niveau* k d'une arborescence contient alors tous les sommets atteint par un chemin de longueur k depuis r. De plus on parlera de *fils* d'un sommet pour parler d'un successeur d'un sommet et de *père* d'un sommet pour parler de son unique prédécesseur. Les sommets sans successeur (sans fils) dans l'arborescence sont nommées des *feuilles* de l'arbre.

Exercice 6. Arborescences
En la représentant de façon à mettre en évidence ses différents niveaux, donner l'arborescence obtenue en prenant le sommet 1 pour racine. Quels sont le(s) fils du sommet 5 ?

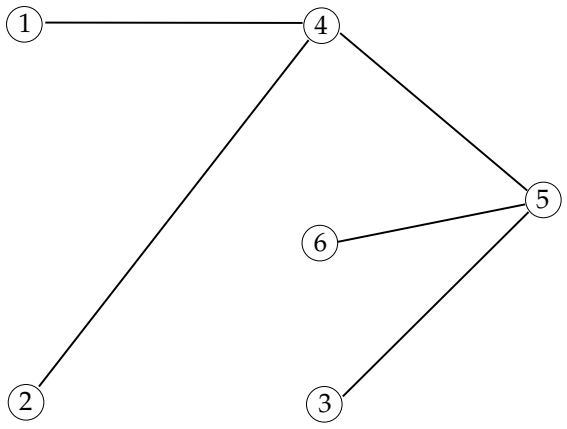


Figure 5: Arbre et arborescence

En pratique, il arrive souvent que l'on parle d'arbre en se référant à une arborescence, mais sans pour autant faire apparaître explicitement d'orientation sur les arêtes. Le contexte permettra d'éviter les ambiguïtés par la définition d'une racine ou par la présence d'une orientation évidente (ex. : un arbre généalogique).

Nous pouvons désormais décrire les deux principales méthodes de parcours des arbres : le *parcours en largeur* et le *parcours en profondeur*. Dans le parcours en largeur, on commence par définir une racine arbitraire (à moins que celle-ci ne soit fixée par le contexte), puis l'exploration de l'arbre se fait niveau par niveau, en partant de la racine. L'ordre dans lequel on explore les sommets d'un même niveau de l'arbre n'est pas dicté par la méthode. Il peut être fixé arbitrairement ou selon ce qui est le plus pratique ou efficace. De façon plus formelle, un parcours en largeur peut être fait en suivant le pseudo-code de l'algorithme 5. Dans ce pseudo-code, on utilise une *file* pour stocker les sommets à traiter. Les files obéissent au principe de First-in, First-out selon lequel le premier élément à sortir d'une file est le premier à y être entré (principe de la file d'attente). Notez que l'algorithme ne retourne rien de particulier. Il dépend d'un traitement particulier à réaliser sur les sommets, et va donc simplement produire l'ordre dans lequel les sommets sont traités. Si on le souhaite, on peut toutefois stocker l'ordre dans lequel les sommets ont été traités et retourner cet ordre.

```
1 Choisir une racine et orienter l'arbre en conséquence;
2 Mettre la racine au début d'une file;
3 Traiter le premier sommet, s, de la file et le retirer de la file;
4 Ajouter les successeurs de s à la fin de la file;
5 Si la file n'est pas vide retourner à l'étape 3;
```

Algorithme 1 : Parcours en largeur d'un arbre

Dans un parcours en profondeur, on part de la racine, puis on traite un fils du sommet qui vient d'être traité jusqu'à atteindre une feuille de l'arbre. On remonte alors jusqu'au dernier sommet traité ayant un fils non traité, puis on reprend le parcours en partant de ce fils. Pour formaliser cet algorithme, on utilisera une pile, c'est-à-dire une liste obéissant au principe de Last-in, First-out.

```
1 Choisir une racine et orienter l'arbre en conséquence;
2 Mettre la racine au début d'une pile;
3 Traiter le premier sommet, s, de la pile et le retirer de la pile;
4 Ajouter les successeurs du sommet au début de la pile;
5 Si la pile n'est pas vide retourner à l'étape 3;
```

Algorithme 2 : Parcours en profondeur d'un arbre

Exercice 7. Parcours d'un arbre
Effectuer un parcours en largeur et un parcours en profondeur en partant du sommet 1 sur l'arbre représenté sur la figure 5. Vous pourrez décrire le résultat des algorithmes par les séquences de traitement des sommets obtenues.

2.5.2 Parcours de graphes quelconques

Pour un graphe quelconque, le parcours se heurte à deux difficultés : la présence de circuits peut amener à traiter un même sommet plusieurs fois en passant par des chemins différents, et les algorithmes vus plus haut peuvent s'arrêter sans avoir parcouru tous les sommets si le graphe n'est pas fortement connexe. Pour éviter de traiter plusieurs fois un même sommet, il suffit d'ignorer le premier sommet de la pile/file s'il est déjà traité. Afin de s'assurer de traiter tous les sommets on répète l'algorithme de parcours en

partant d'un nouveau sommet non-traité arbitraire tant que tous les sommets ne sont pas traités. Je donne ci-dessous les pseudo-codes pour le parcours en largeur et en profondeur de graphes orientés. Pour un graphe non-orienté, il suffit d'ajouter les voisins non-traités plutôt que les successeurs non traités à l'étape 4.

1 Mettre un sommet non-traité du graphe au début d'une file vide;
2 Soit s le premier sommet de la file;
3 Retirer s de la file;
4 Si s n'est pas traité, le traiter et ajouter ses successeurs non-traités à la fin de la file ;
5 Si la file n'est pas vide retourner à l'étape 2;
6 Si certains sommets n'ont pas été traités, retourner à l'étape 1;

Algorithme 3 : Parcours en largeur d'un graphe orienté quelconque

1 Mettre un sommet non-traité du graphe au début d'une pile vide ;
2 Soit s le premier sommet de la pile ;
3 Retirer s de la pile;
4 Si s n'est pas traité, le traiter et ajouter ses successeurs non-traités au début de la pile ;
5 Si la pile n'est pas vide retourner à l'étape 2;
6 Si certains sommets n'ont pas été traités, retourner à l'étape 1 ;

Algorithme 4 : Parcours en profondeur d'un graphe orienté quelconque

Exercice 8. Parcours de graphe
Effectuer un parcours en largeur et un parcours en profondeur en partant du sommet o sur le graphe représenté sur la figure 1. Une bonne représentation du parcours trouvé consiste à reproduire le graphe en ne conservant que les arcs qui ont été suivis pour parcourir des sommets, et en numérotant les sommets dans leur ordre de parcours.

On donne désormais le premier résultat de *complexité* du cours. Dans tout le cours on parlera de complexité pour désigner une complexité en temps d'exécution des algorithmes. Cette dernière sera toujours mesurée au pire cas, en fonction de la taille des données d'entrée. Dans le cas des graphes, il s'agira en général du nombre de sommets n et du nombre d'arêtes m. Il s'agira alors de compter l'ordre de grandeur du nombre d'opérations de base effectuées au pire cas jusqu'à ce que l'algorithme termine. Par opérations de base, on désigne les opérations arithmétiques simples (addition, soustraction, multiplication, division, comparaison de deux nombres), l'accès à un élément donné d'un vecteur et l'affectation d'une valeur à une variable.

Proposition 2.12. Complexité

Un algorithme de parcours est exécuté en au plus $\mathcal{O}(m+n)$ opérations de base.

Démonstration : *On ajoute les successeurs d'un sommet dans la pile/file au plus une fois, donc on fait au plus m ajouts de sommets dans la pile/file. Le premier sommet de la file sera donc au plus m fois un sommet ajouté en tant que successeur et au plus n fois un sommet ajouté à la ligne 1. À chaque itération, on accède au premier sommet de la pile/file, on le retire de la pile/file, puis on teste s'il a*

été traité et si tous les sommets sont traités. Toutes ces opérations correspondent à des opérations de base, donc on effectue au plus $\mathcal{O}(m+n)$ opérations en plus de l'ajout de successeurs. Cela donne la complexité annoncée.

Notez que si le graphe est connexe, la complexité est en $\mathcal{O}(m)$, car tous les sommets ajoutés dans la pile/file sauf le premier le sont en tant que successeur d'un sommet non-traité (c'est-à-dire que chaque ajout correspond à un arc).

2.5.3 Application des algorithmes de parcours

Les algorithmes ont deux applications directes pour déterminer les composantes connexes d'un graphe et rechercher des circuits. Pour déterminer les composantes connexes d'un graphe, il suffit de lancer l'un des algorithmes de parcours de graphe vu ci-dessus. Quel que soit l'algorithme choisi, on sait que si un sommet s a été choisi à l'étape 1, alors tous les sommets atteignables depuis s seront traités à la prochaine exécution de l'étape 6. Cela signifie que la composante connexe à laquelle appartient s est identifiée entre l'étape 1 et l'étape 6. Ainsi, on passera par l'étape 1 autant de fois qu'il y a de composantes connexes dans le graphe.

Pour effectuer une recherche de circuit dans un graphe donné, il faut modifier l'algorithme de parcours en profondeur en rangeant les sommets dans trois ensembles au cours de l'algorithme. Un premier ensemble de sommets (que l'on repérera par la couleur blanche) correspond aux sommets non-traités. Le second ensemble (les sommets noirs) contient les sommets traités dont tous les successeurs sont également noirs. Le dernier ensemble (les sommets gris) contient les sommets traités ayant des successeurs blancs. Les sommets commencent tous en blanc, puis on colore en gris les sommets au moment où ils sont traités. Dès qu'un sommet gris n'a plus de successeur blanc (ou pas de successeur tout court), on le colore en noir.

On détecte la présence d'un circuit si l'on ajoute un sommet gris à la pile à l'étape 4 du parcours en profondeur. Réciproquement, on conclut que le graphe est sans circuit si cela ne se produit pas. En outre, ce même algorithme nous permet de déterminer un *ordre topologique* lorsque le graphe ne contient pas de circuit.

Un ordre topologique est une numérotation des sommets de 1 à n telle qu'il ne peut exister un arc du sommet numéroté i vers le sommet numéroté j que si $i < j$. Plus formellement, un ordre topologique est une bijection $\sigma : S \to \{1, \ldots, n\}$ telle que $(i,j) \in A \implies \sigma(i) < \sigma(j)$. On peut trouver un ordre topologique dans un graphe si et seulement si ce dernier ne contient pas de circuit. Pour déterminer un tel ordre par l'algorithme vu plus haut, il suffit d'affecter le numéro n au premier sommet colorié en noir, puis d'affecter le plus grand entier non affecté à chaque fois qu'un sommet est colorié en noir. Le pseudo-code ci-dessous formalise cet algorithme.

```
1  Initialiser k = n;
2  Mettre un sommet non-traité du graphe au début d'une pile vide ;
3  Soit s le premier sommet de la pile ;
4  Retirer s de la pile et le colorier en gris s'il est blanc;
5  Si l'un des successeurs de s est gris : STOP, le graphe contient un circuit;
6  Ajouter tous les successeurs blancs de s au début de la pile;
7  si aucun successeur de s n'est blanc (ou pas de successeur) alors
8  |  colorier_noir(s, k);
9  Si la pile n'est pas vide retourner à l'étape 3;
10 Si tous les sommets n'ont pas été traités, retourner à l'étape 2;
```

Algorithme 5 : Détection de circuit et tri topologique

Le pseudo-code ci-dessus fait appel à une fonction auxiliaire colorier_noir(s, k) lorsque tous les successeurs du sommet s sont noirs. Cette fonction sert à colorier s en noir et à lui affecter un rang dans l'ordre topologique en construction, puis à faire de même récursivement sur ces prédécesseurs. On formalise cette fonction comme suit.

```
1 colorier s en noir, affecter σ(s) = k et poser k = k − 1;
2 pour t prédécesseur gris de s faire
3 |   si tous les successeurs de t sont noirs alors
4 |   |   colorier_noir(t,k)
```

Algorithme 6 : colorier_noir(s, k)

Exercice 9. Détection de circuit
À l'aide d'un algorithme de parcours, déterminer si le graphe ci-dessous contient un circuit et s'il n'en contient pas, donner un ordre topologique de ses sommets.

Même question après avoir inversé le sens de l'arc $(6, 4)$.

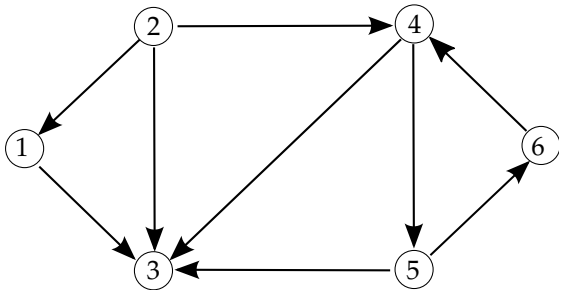


Figure 6: Ce graphe contient-il un circuit ?

Exercice 10. Parcours et connexité

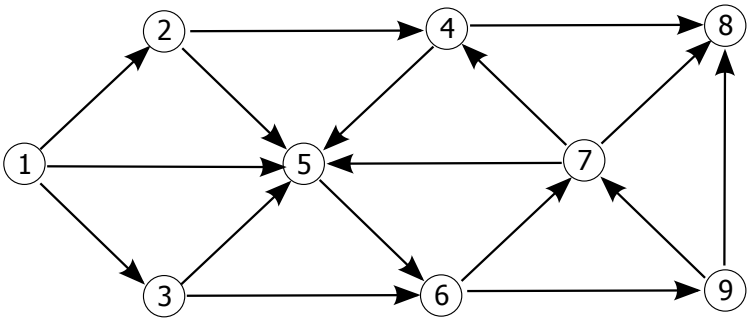


Figure 7: Exemple pour l'exercice 10

i Préciser la différence entre le parcours en largeur et le parcours en profondeur d'un arbre.

ii Comment peut-on déterminer la liste des sommets accessibles à partir d'un sommet source s d'un graphe orienté (c'est-à-dire les sommets que l'on peut relier à s par un chemin) ?

iii Comment appliquer cet algorithme pour déterminer si un graphe orienté est fortement connexe ?

iv Dans la suite, on considère le graphe orienté $G = (S, A)$ donné en Figure 7. En modifiant légèrement l'algorithme de la question 2 pour marquer les arcs empruntés à chaque fois qu'un sommet est ajouté à la liste des accessibles, il est possible de construire un arbre donnant un chemin unique de la source vers chaque sommet accessible. Déterminer les arbres pour le parcours en largeur et pour le parcours en profondeur en prenant le sommet 1 comme source.

v Le graphe de la figure 7 est-il acyclique ? Si oui, modifier la numérotation des sommets de façon à ce qu'elle donne un ordre topologique (c'est-à-dire une numérotation des sommets par les entiers $1, \ldots, |S|$, telle que $i < j$ pour tout $(i, j) \in A$).

vi Même question que la précédente après avoir retiré l'arc $(5, 6)$.

Exercice 11. Graphes bipartis et coloration de graphe

i En vous référant à ses niveaux après un choix arbitraire d'une racine, montrer comment colorer les sommets d'un arbre à l'aide de deux couleurs, afin que pour chaque arête $\{i, j\}$ de l'arbre, les sommets i et j soient de couleurs différentes.

ii Soit T un arbre couvrant d'un graphe non-orienté connexe $G = (V, A)$. En utilisant la question précédente, prouver que G est biparti si pour toute arête $\{k, l\}$ n'appartenant pas à T, les sommets k et l sont de couleurs différentes (un graphe est biparti si et seulement s'il existe une partition de ses sommets en deux ensembles V_1, V_2, tels que pour tout $\{i, j\} \in A : i \in V_1$ et $j \in V_2$ ou $j \in V_1$ et $i \in V_2$).

iii Grâce à cette caractérisation, décrire un algorithme en $O(m)$ (m est le nombre d'arêtes) permettant de déterminer si un graphe connexe est biparti ou non.

3 Algorithmes de graphe polynomiaux

3.1 Plus court et plus long chemin

On considère un graphe orienté $G = [S, A]$ et l'on associe une longueur l_a à chaque arc $a \in A$. Le problème du plus court chemin d'un sommet o à un sommet d de G consiste à chercher le chemin de o à d tel que la somme des longueurs des arcs le composant soit minimale.

Dans le cas où G ne contient pas de circuit de longueur strictement négative, on prouve qu'il existe toujours un chemin de longueur minimum sans circuit. Pour cela, il suffit de dire que s'il existe un circuit dans un chemin de o à d, on peut le supprimer pour obtenir un nouveau chemin de o à d de longueur inférieure ou égale (car le circuit a une longueur positive ou nulle). Cet argument a permis de développer des algorithmes s'exécutant en temps polynomial pour résoudre ce problème. Dans la suite, nous supposerons donc que G ne contient pas de circuit de longueur strictement négative (ce qui n'exclut en revanche pas que certaines longueurs soient négatives).

Remarque 2. *Si le graphe contient un circuit de longueur strictement négative et qu'un des sommets du circuit est accessible depuis o, il existe un chemin de longueur aussi petite qu'on le souhaite pour aller de o à d. Il suffit pour cela de parcourir le circuit de longueur négative autant que nécessaire. Dans ce cas, il sera plus intéressant de rechercher un chemin élémentaire de longueur minimum dans le graphe. Le problème devient alors difficile au sens que l'on ne connaît pas d'algorithme pouvant le résoudre en un temps polynomial.*

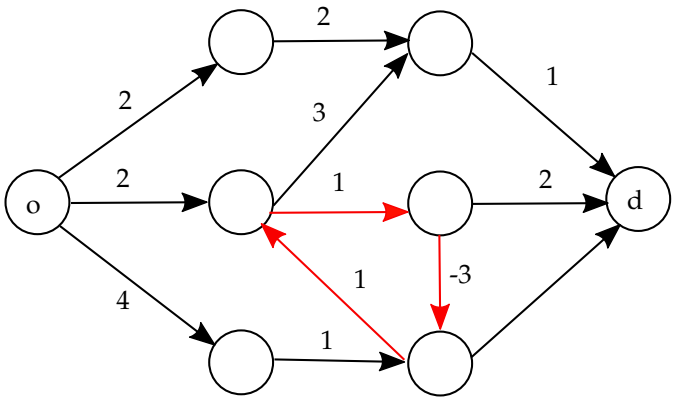


Figure 8: Graphe contenant un circuit de longueur négative

3.1.1 Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford repose sur les principes de la programmation dynamique. La programmation dynamique est une famille de méthodes algorithmiques permettant de résoudre des problème d'optimisation dont la solution peut être déduite à partir de la solution d'instances plus petites du même problème. On calcule alors la solution du problème de façon ascendante en résolvant des problèmes de plus en plus gros.

Pour le problème de plus court chemin, on constate que pour tout $s \in S$, un chemin de o à s contenant au plus $k+1$ arcs consiste en un chemin contenant au plus k arcs et allant de o à un prédécesseur t de s auquel on ajoute l'arc (t, s). Notons donc $L(s, k)$ la longueur du plus court chemin de o à s contenant *au plus* k arcs. Notre observation implique que :

$$L(s, k+1) = \min \left\{ L(s, k), \min_{t \in \delta^-(s)} \{L(t, k) + l_{ts}\} \right\}.$$

Cette formule de récurrence indique que si l'on sait trouver les plus courts chemins à partir de o en 0 arc, on sait calculer les plus courts chemins à partir de o en au plus k arcs pour tout $k \in \mathbb{N}$. De plus, on sait qu'il existe un chemin optimal sans circuit. Sachant qu'un chemin sans circuit ne passe jamais deux fois par le même sommet, il existe un chemin optimal contenant au plus $n-1$ arcs. Ainsi, $L(d, n-1)$ correspond nécessairement à la longueur du plus court chemin de o à d. Pour formaliser la procédure, on pourra suivre le pseudo-code ci-dessous.

1 Poser $L(o,0) = 0$ et $L(s,0) = +\infty, \forall s \neq o$;
2 **pour** *k allant de 0 à $n-2$* **faire**
3 | **pour** *tout sommet s de S* **faire**
4 | | $L(s,k+1) = \min\left\{L(s,k), \min_{(t,s)\in\delta^-(s)}\{L(t,k) + l_{ts}\}\right\}$;
5 | | **si** $L(s,k+1) < L(s,k)$ **alors**
6 | | | predecesseur$(s) = \text{argmin}_{(t,s)\in\delta^-(s)}\{L(t,k) + l_{ts}\}$;
7 | **si** $L(s,k+1) = L(s,k), \forall s \in S$ **alors**
8 | | STOP, l'algorithme a convergé;

Algorithme 7 : Algorithme de Bellman-Ford

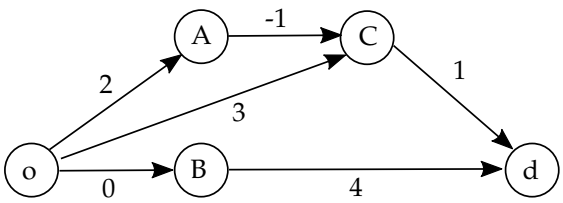
Le test ajouté à l'étape 7 sert à écourter l'exécution lorsqu'aucune longueur n'est mise à jour à une itération donnée. En effet, si cette condition est satisfaite, les itérations suivantes ne donneront pas plus de changement dans le calcul des longueurs minimum. Par ailleurs, l'opération réalisée à l'étape 6 permet de mettre à jour le meilleur prédécesseur de chaque sommet quand sa distance minimum à o est mise à jour. Ainsi, à la fin de l'algorithme, on pourra construire un *arbre des plus courts chemins* à l'aide du meilleur prédécesseur de chaque sommet. En particulier, cela donne le plus court chemin de o à d.

La bonne pratique est de toujours justifier qu'un algorithme converge bien vers la solution que l'on recherche, puis d'étudier sa complexité au pire cas. Ici, la convergence a été justifiée avant la description de l'algorithme. Pour ce qui est de la complexité dans le pire cas, on a vu que l'algorithme effectue au plus n itérations, car le chemin de longueur minimum contient au plus $n-1$ arcs. À chaque itération, on applique la formule de récurrence à chaque sommet s, ce qui amène à faire une comparaison et une addition pour chaque prédécesseur de s. Regarder tous les prédécesseurs de tous les sommets équivaut à regarder tous les arcs, donc on réalise de l'ordre de m opérations de base à chaque itération. Au final, la complexité au pire cas est donc en $\mathcal{O}(n \times m)$.

Remarque 3. *Dans le calcul de la complexité, on a implicitement supposé que tout sommet autre que o a au moins un prédécesseur (et donc $m \geq n$). Il est facile d'imposer cette hypothèse, car les sommets sans prédécesseur peuvent simplement être ignorés dans une recherche de plus courts chemins.*

Remarque 4. *En fait, l'algorithme de Bellman-Ford ne retourne pas uniquement la longueur du plus court chemin de o à d : il retourne la longueur des plus courts chemins de o vers tous les autres sommets.*

Exercice 12 (Bellman-Ford)**.** *Appliquer l'algorithme de Bellman-Ford pour trouver les plus courts chemins de o vers tous les autres sommets du graphe suivant*



3.1.2 Algorithme de Dijkstra

Dans le cas particulier où tous les arcs sont de longueur positive ou nulle, il est possible de faire mieux que l'algorithme de Bellman-Ford. L'algorithme qui utilise le mieux cette hypothèse est l'algorithme de Dijkstra dont on donne un pseudo-code ci-dessous. Dans la description de cet algorithme, $L(s)$ correspond à la longueur d'un chemin de o à s (on montrera qu'à la fin de l'algorithme, il s'agit de la longueur du plus court chemin de o à s). Ce qu'il est essentiel de remarquer ici est que l'algorithme effectue exactement n itérations dans la boucle tant que , mais qu'à chacune de ces itérations, les successeurs d'un seul sommet sont examinés. En fait, nous verrons dans la preuve de convergence de l'algorithme qu'une fois qu'un sommet s est marqué, $L(s)$ contient la longueur du plus court chemin de o à s.

```
1 Poser L(o) = 0 et L(s) = +∞, ∀s ≠ o;
2 Initialiser M = ∅ // Aucun sommet n'est marqué initialement
3 tant que M ≠ S faire
4 |   Soit s = argmin_{t∈S\M}{L(t)};
5 |   Ajouter s à M // marquer le sommet non-marqué minimisant L
6 |   pour tout successeur t de s tel que t ∉ M faire
  |   |   // mise-à-jour des successeurs de s uniquement
7 |   |   si L(s) + l_st < L(t) alors
8 |   |   |   Poser L(t) = L(s) + l_st;
9 |   |   |   Stocker predecesseur(t) = s;
```

Algorithme 8 : Algorithme de Dijkstra

Preuve de l'algorithme : On montre qu'à la fin de l'algorithme de Dijkstra, L contient la longueur du plus court chemin de o vers chaque sommet. On note $L^*(s)$ cette plus petite distance de o à s pour tout $s \in S$.

Pour parvenir au résultat, on montre par récurrence qu'à chaque itération, le sommet marqué s vérifie $L(s) = L^*(s)$. Cela assurera que $L(t) = L^*(t)$ pour tout sommet marqué, t, et donc pour tous les sommets de S à la fin de l'algorithme.

L'hypothèse de récurrence est évidemment vraie à l'itération 1 (on marque le sommet o), donc on se place à une itération ultérieure et l'on suppose que $L(t) = L^*(t), \forall t \in M$. On note alors s le sommet marqué à cette itération, et l'on considère un chemin quelconque de o à s. Notons alors t le premier sommet non-marqué du chemin (on pose $t = s$ si tous les sommets sont marqués). Tous les sommets avant t étant marqués, l'hypothèse de récurrence assure que la longueur du chemin partiel de o à t est supérieure ou égale à $L(t)$. Par positivité des longueurs des arcs, cela implique que la longueur du chemin complet est elle aussi supérieure ou égale à $L(t)$. De plus, $L(s) \leq L(t)$ sinon on aurait marqué t à la place de s. Le chemin est donc de longueur supérieure ou égale à $L(s)$. On en déduit que tout chemin de o à s est de longueur supérieure ou égale à $L(s)$, c'est-à-dire $L(s) = L^*(s)$.

Complexité standard : Un sommet est marqué à chaque itération, donc il y a exactement n itérations. À chaque itération, on doit chercher le sommet non-marqué de longueur minimale. Au pire, on cherche donc le minimum d'un vecteur à n éléments. En notant $\theta(n)$ le nombre d'opérations de base nécessaires pour cela, on effectue donc $n \times \theta(n)$ opérations pour choisir le sommet à marquer. Par la suite on effectue une comparaison et une addition pour chaque successeur du dernier sommet marqué. Donc, sur l'ensemble de l'algorithme, on fait m additions et comparaisons pour la mise-à-jour des longueurs.

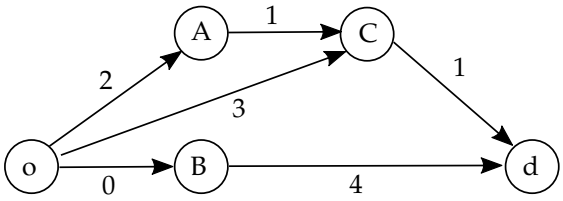
Ainsi la complexité de l'algorithme de Dijkstra est en $\mathcal{O}(m + n \times \theta(n))$.

Soit $\theta(n)$ le nombre d'opérations de base nécessaires pour trouver le minimum d'une liste de n éléments.
L'algorithme de Dijkstra a une complexité en $\mathcal{O}(\mathbf{m} + \mathbf{n} \times \boldsymbol{\theta}(\mathbf{n}))$.

Vers une complexité optimale : *Le temps d'accès au minimum d'une liste dépend des structures de données utilisées. L'algorithme le plus évident va effectuer exactement n comparaisons, mais des structures plus efficaces permettent de trouver le minimum en $\mathcal{O}(\log(n))$ opérations. En Python, on rangera la liste des distance dans une "heap queue list" dont le fonctionnement est décrit sur la page* `https://docs.python.org/2/library/heapq.html`.

Ainsi, on dira le plus souvent que la complexité de Dijkstra est en $\mathcal{O}(m + n\log(n))$. Si le graphe est connexe, il contient au moins $n-1$ arcs, et s'il s'agit d'un 1-graphe, il en contient au plus n^2. Ainsi, selon la prépondérance de m devant $n\log(n)$, la complexité pourra se comporter comme m ou comme $n\log(n)$.

Exercice 13 (Dijkstra)**.** *Appliquer l'algorithme de Dijkstra pour trouver les plus courts chemins de o vers tous les autres sommets du graphe suivant*



3.1.3 Plus long chemin dans un graphe sans circuit de longueur positive

Les deux algorithmes vus plus haut peuvent être adaptés pour calculer les plus longs chemins à partir d'un sommet o vers tous les autres sommets dans un graphe sans circuit de longueur positive. Dans le cas général où le graphe peut contenir des circuits de longueur positive, on retombe dans le cas difficile où le graphe peut contenir des chemins arbitrairement longs et où l'on est plutôt intéressé par les chemins *élémentaires* les plus longs.

L'adaptation de l'algorithme de Bellman-Ford est immédiate, il suffit de considérer le pseudo-code suivant et les résultats de convergence et de complexité sont inchangés.

```
Poser L(o,0) = 0 et L(s,0) = −∞, ∀s ≠ o;
pour k allant de 0 à n − 2 faire
    pour tout sommet s de S faire
        L(s,k+1) = max{ L(s,k), max_{(t,s)∈δ⁻(s)} {L(t,k) + l_ts} };
        si L(s,k+1) > L(s,k) alors
            predecesseur(t) = argmax_{(t,s)∈δ⁻(s)} {L(t,k) + l_ts};
    si L(s,k+1) = L(s,k), ∀s ∈ S alors
        STOP, l'algorithme a convergé;
```

Algorithme 9 : Algorithme de Bellman-Ford pour les plus longs chemins

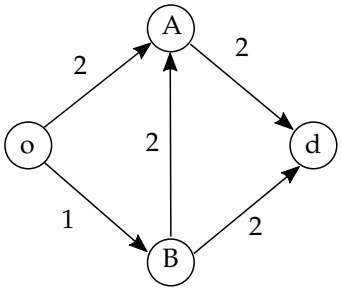


Figure 9: Contre-exemple pour l'adaptation directe de Dijkstra à la recherche de plus longs chemins

L'adaptation de l'algorithme de Dijkstra est connue sous le nom d'algorithme de Bellman. **L'algorithme de Bellman ne peut être appliqué que si le graphe ne contient pas de circuit.** Cette adaptation demande un peu plus d'effort que pour l'algorithme de Bellman-Ford. En effet, si l'on se contente de marquer le sommet non-marqué maximisant L à chaque itération et de changer le sens des inégalités, l'algorithme ne produit pas les plus longs chemins. Il suffit de considérer l'exemple très simple dessiné sur la figure 9 pour s'en convaincre. En fait, on utilise plutôt l'absence de circuit pour adapter Dijkstra.

```
1 Poser L(o) = 0 et L(s) = −∞, ∀s ≠ o;
2 Aucun sommet n'est marqué initialement;
3 tant que il y a des sommets non marqués faire
4 |   Marquer un sommet s non marqué dont tous les prédécesseurs sont marqués;
5 |   pour tout successeur non-marqué t de s faire
6 |   |   si L(s) + l_st > L(t) alors
7 |   |   |   Poser L(t) = L(s) + l_st;
8 |   |   |   Stocker predecesseur(t) = s;
```

Algorithme 10 : Algorithme de Bellman

En l'absence de circuit, on est sûr qu'à chaque itération, il existe un sommet non-marqué dont tous les prédécesseurs sont marqués. Sachant que si tous les prédécesseurs d'un sommet s sont marqués, la valeur de $L(t)$ ne pourra plus augmenter, il est aisé de montrer par récurrence que Bellman retourne bien les plus longs chemins vers chaque sommet.

Par ailleurs, comme Dijsktra, l'algorithme effectue $\mathcal{O}(m)$ opérations pour mettre à jour les valeurs de L. Mais il peut par la même occasion mettre à jour le nombre de prédécesseurs non-marqués de chaque successeur du dernier sommet marqué (au moment où l'on marque un sommet, chacun de ses successeurs perd un prédécesseur non-marqué). Ainsi, l'accès à un sommet sans prédecesseur ne coûte pas plus cher que le maintien des nombres de prédécesseurs. Au final, la complexité de l'algorithme de Bellman est donc en $\mathcal{O}(m)$.

Remarque 5. *Pour traiter des graphes non-orientés, on interprète l'absence d'orientation d'une arête comme la possibilité de l'emprunter dans les deux sens, ce qui équivaut à avoir deux arcs de même longueur dans des sens opposés. Les algorithmes vus pour les plus courts chemins s'adaptent donc sans difficulté. En revanche, le graphe orienté ainsi obtenu est nécessairement cyclique, donc on est forcé de chercher un plus long chemin* élémentaire*, ce qui est bien plus difficile.*

Exercice 14. Dijsktra et Bellman
On considère le graphe orienté suivant représenté sur la figure 10.

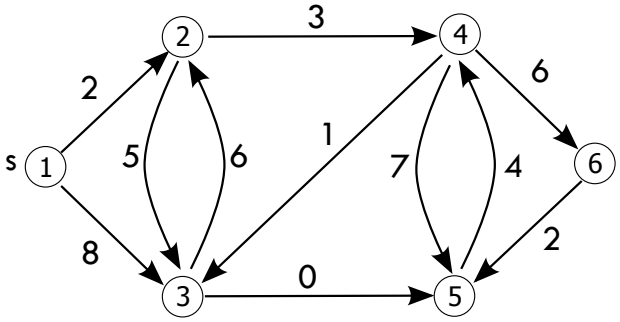


Figure 10: Exemple pour l'exercice 14

i *Quelle hypothèse permet d'affirmer que l'algorithme de Dijkstra trouve le plus court chemin d'un sommet s vers tous les autres sommets d'un graphe orienté ?*

ii *Quelle est la complexité de l'algorithme ?*

iii *Appliquer l'algorithme de Dijsktra au graphe ci-dessus pour obtenir les plus courts chemins à partir du sommet 1. Représenter la solution sous la forme d'un arbre de parcours de longueurs minimum.*

iv *Supprimer deux arcs de façon à ce que le graphe ne contienne aucun circuit.*

v *Expliquer dans quelle mesure l'étape précédente a un impact sur le calcul du plus long chemin dans le graphe, puis appliquer l'algorithme de Bellmann pour calculer les plus longs chemins à partir du sommet 1 vers tous les autres sommets du graphe. Représenter la solution sous la forme d'un arbre de parcours de longueurs maximum.*

Exercice 15 (**Modélisation par un plus court chemin).** *L'objet de cet exercice est le rangement optimal de livres ordonnés selon leur taille de façon à minimiser leur coût de stockage.*

Soit un ensemble de livres dont l'épaisseur et la hauteur sont connues. Supposons que les hauteurs des livres ont été ordonnées de façon croissante $H_1 < H_2 < \ldots < H_n$, et notons L_i la somme des épaisseurs des livres ayant la même hauteur H_i, $1 \leq i \leq n$ (bien noter que plusieurs livres peuvent avoir la même hauteur). Ainsi, il faut que l'étagère contenant les livres de hauteurs H_i soit au moins de longueur L_i.

Le coût d'une étagère de hauteur H_i et de longueur x_i est donné par $F_i + C_i x_i$, où F_i est un coût fixe et C_i le coût par unité de longueur de l'étagère $C_1 \leq C_2 \leq \ldots \leq C_n$. Notons que pour faire des économies sur le coût fixe des étagères, on peut utiliser une étagère de hauteur H_i pour ranger des livres plus petits, mais si elle contient des livres de hauteur H_j avec $j < i$, alors elle doit contenir tous les livres de hauteur H_k avec $j \leq k \leq i$.

i *Formuler ce problème comme un problème de plus court chemin.*

ii *Résoudre le problème de rangement optimal de livres pour les données du tableau 2. Utiliser l'algorithme de Dijkstra pour trouver le plus court chemin.*

3.2 Arbre couvrant de poids minimum

Exemple 5. *Une entreprise souhaite installer un réseau de type fibre dans cinq villes A, B, C, D, E. Le réseau doit permettre les communications entre toutes paires de villes, mais la longueur du chemin*

i	1	2	3	4
H_i (cm)	15	20	23	30
L_i (cm)	100	300	200	300
F_i	1000	1200	1100	1600
C_i	5	6	7	9

Table 1: Données de l'exercice 15

parcouru n'a pas d'importance. Seules certaines villes peuvent être directement raccordées les unes aux autres. Une estimation du coût de fabrication des lignes joignant ces paires de villes est donnée sur le graphe ci-dessous.

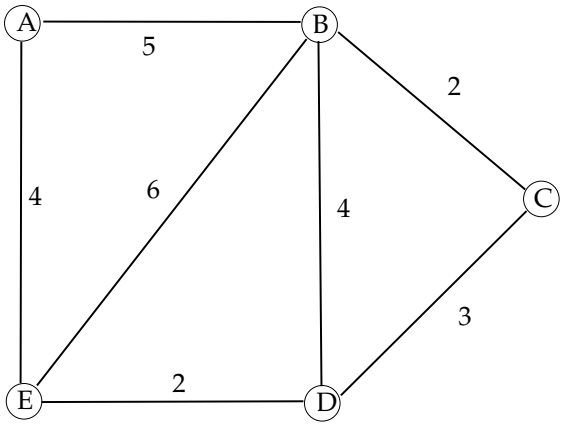


Figure 11: Prix des liaisons directes.

Quelles liaisons choisir pour construire le réseau à moindre coût ?

On se donne un graphe non-orienté connexe $G = [S, A]$ et un coût c_a pour chaque arête a. Le problème de l'arbre couvrant de poids minimum consiste à rechercher un sous-ensemble d'arêtes $A' \subset A$ tel que le sous-graphe de G restreint à A' est un arbre et dont la somme des coûts des arêtes est minimum. Ce problème peut être résolu très efficacement à l'aide des algorithmes de Kruskal ou de Prim.

3.2.1 Algorithme de Kruskal

Sachant qu'un arbre est un graphe connexe et acyclique, l'algorithme de Kruskal ajoute itérativement l'arête de coût minimum à condition que celle-ci ne crée pas de cycle. Comme nous le démontrons ci-dessous, cet algorithme simple répond effectivement au problème. On donne d'abord le pseudo-code ci-dessous.

```
1 Trier A par coût d'arête croissant;
2 Initialiser T = [S, ∅] // l'arbre en construction
3 tant que T a moins de n − 1 arêtes faire
4 |  si la première arête de A ne crée pas de cycle dans T alors
5 |  |  Ajouter a à T;
6 |  Supprimer a de A.
7 Retourner T;
```

Algorithme 11 : Algorithme de Kruskal

Preuve de l'algorithme : Par construction de T, celui-ci ne peut pas contenir de cycle. De plus l'algorithme s'arrête quand T possède $n-1$ arêtes. Le sous-graphe retourné est donc bien un arbre.

Pour montrer qu'il est de poids minimum, on montre par récurrence que les k premières arêtes ajoutées à T appartiennent bien à un arbre de poids minimum. C'est bien évidemment vrai pour $k = 0$, donc on le suppose pour un certain $0 \leq k \leq n-2$. On note alors T_k, l'arbre couvrant de poids min. auxquelles appartiennent les k premières arêtes ajoutées à T et a la $(k+1)$ème arête ajoutée à T.

Si $a \in T_k$, l'hypothèse de récurrence est vérifiée pour $k+1$.

Si $a \notin T_k$, $T_k \cup \{a\}$ est connexe avec n sommets et n arêtes donc il contient un cycle. Ce cycle contient donc nécessairement une arête e qui n'a pas été ajoutée à T par l'algorithme de Kruskal. Si $c_e < c_a$, e aurait été choisie à la place de a, donc on a $c_e \geq c_a$. Ainsi, $(T_k \setminus \{e\}) \cup \{a\}$ a un coût au plus égal à celui de T_k. De plus, ce graphe a été obtenu en supprimant une arête appartenant à un cycle dans un graphe connexe, donc il est aussi connexe. Finalement, il s'agit d'un graphe à n sommets et $n-1$ arêtes donc il s'agit bien d'un arbre de poids minimum.

Complexité standard : La premiere étape est faite une fois pour toutes. De nombreux algorithmes standard ont été développés pour trier une liste de taille m. La complexité des meilleurs d'entre eux (cf. le tri fusion par exemple) est en $\mathcal{O}(m \log(m))$.

Par la suite, l'algorithme de Kruskal effectue au plus m itérations (pour parcourir toute la liste triée). À chaque itération, l'étape prépondérante est la détection de cycle dans $T \cup \{a\}$. Nous avons déjà vu que cela peut se faire par un parcours en profondeur. Sachant que T contient au plus $n-1$ arêtes, la complexité de cette étape est donc en $\mathcal{O}(n)$. On déduit que la boucle est effectuée $\mathcal{O}(m \times n)$.

Au final, la complexité du tri est négligeable devant celle de la boucle. D'où l'exécution de Kruskal est en $\mathcal{O}(m \times n)$.

L'algorithme de Kruskal a une complexité en $\mathcal{O}(\mathbf{m \times n})$.

Comme ce sera souvent le cas pour les algorithmes vus dans ce cours, l'implémentation naïve donne une complexité acceptable, mais il est possible de faire (beaucoup) mieux.

Vers une complexité optimale : *Supposons que pour tout $s \in S$, l'on maintienne la liste des sommets que l'on atteint depuis s dans T. Au moment d'ajouter une arête $\{u, v\}$ dans T, on détecte alors un cycle en regardant simplement si v est dans la liste des sommets que l'on peut atteindre depuis u. Pour une structure de données bien choisie, le maintien des listes de sommets accessibles et la détection de cycle peuvent alors être faits en $\mathcal{O}(\log(n))$. Ainsi, l'exécution de la boucle est maintenant en $\mathcal{O}(m \log(n))$, ce qui devient comparable à la complexité du tri. (Cette structure de données est généralement nommée Union-Find, et consiste à ranger les données sous la forme d'une forêt.)*

3.2.2 Algorithme de Prim

La seconde approche consiste à faire grossir un arbre au fil des itérations, en ajoutant une arête et un sommet à chaque itération. Le résultat est connu sous le nom d'algorithme de Prim (un algorithme similaire est celui de Solin).

```
1 Soit s ∈ S arbitrairement choisi;
2 Initialiser T = ({s}, ∅) // l'arbre en construction
3 pour k allant de 1 à n − 1 faire
4 |   Soit a une arête de coût minimum parmi celles ayant exactement une extrémité dans T;
5 |   Ajouter a à T, ainsi que l'extrémité de a qui n'est pas encore dans T;
6 Retourner T;
```

Algorithme 12 : Algorithme de Prim

Pour clarifier l'algorithme ci-dessus (si besoin), notez qu'à chaque itération, un sous-ensemble de sommets S' a été ajouté à T et les sommets de $S \setminus S'$ attendent d'y être ajoutés. On cherche alors l'arête de poids minimum parmi celles qui ont exactement une extrémité dans S' et l'autre dans $S \setminus S'$. C'est cette arête que l'on ajoute à T avec l'une de ses extrémités.

Preuve de l'algorithme : On remarque d'abord qu'à chaque itération on ajoute à T une arête dont seulement une extrémité est dans T. On est donc sûr que cette arête ne peut pas créer de cycle dans l'arbre en construction et que ce dernier est connexe. Ainsi, T est un arbre à chaque étape de l'algorithme de Prim.

La démonstration la plus simple emploie la même récurrence que pour l'algorithme de Kruskal. On suppose donc qu'à l'itération $1 \leq k \leq n-2$, l'arbre T peut être complété de façon à former un arbre couvrant minimum T_k. On pose ensuite S_k et A_k les ensembles de sommets et d'arêtes tels que $T = [S_k, A_k]$ et on note s_{k+1} et a_{k+1}, le sommet et l'arête ajoutés à T à l'itération $k+1$.

Il reste à démontrer que le graphe $[S_k \cup \{s_{k+1}\}, A_k \cup \{a_{k+1}\}]$ peut être complété pour former un arbre couvrant de poids min. T_{k+1}. Pour cela considérer également les deux cas $a_{k+1} \in T_k$ et $a_{k+1} \notin T_k$. Laissée en exercice (voir indices dans l'énoncé du dernier exercice de la section).

La distinction principale entre les algorithmes de Prim et de Kruskal est dans les opérations effectuées dans la boucle principale. Dans Kruskal, on détecte l'existence d'un cycle, tandis que l'on recherche une arête de coût minimum dans Prim.

Complexité standard : Ici, il suffit de s'intéresser aux opérations effectuées dans la boucle pour. Il y a au plus n itérations. Si l'on note S_k les sommets de T à une itération k, on cherche alors l'arête de coût minimum entre S_k et $S \setminus S_k$. Pour cela, on peut maintenir la liste contenant les coûts minimum des arêtes joignant s à S_k pour tout $s \in S \setminus S_k$. De façon naïve, cela peut être fait en $\mathcal{O}(n)$: pour chaque sommet $s' \notin S_k$, on compare la valeur actuelle au coûts de l'arête $\{s, s'\}$ si elle existe. La plus petite des valeurs de coût de cette liste peut être mise à jour dans la même boucle. Au final, l'implémentation standard de Prim a donc une complexité $\mathcal{O}(n^2)$. (Démonstration laissée en exercice) **L'algorithme de Prim a une complexité en $\mathcal{O}(\mathbf{n^2})$.**

Vers une complexité optimale : *Une implémentation plus avancée fait appel à une structure de tas min. binaire dont la forme est celle d'un arbre binaire équilibré (exactement deux fils pas sommet*

de l'arbre sauf au niveau précédant les feuilles). On atteint aussi une complexité $\mathcal{O}(m\log(n))$. La meilleure implémentation connue à ce jour atteint même une complexité en $\mathcal{O}(m + n\log(n))$.

Exercice 16. *Appliquer successivement l'algorithme de Prim et de Kruskal pour traiter l'exemple de la figure 11.*

Exercice 17.

i Rappeler l'utilité des algorithmes de Prim et de Kruskal et donner leur complexité.

ii Construire un graphe non-orienté connexe possédant 6 sommets et 12 arêtes auxquelles vous affecterez arbitrairement des poids entiers et positifs entre 1 et 10.

iii Appliquer les algorithmes de Prim et de Kruskal au graphe que vous avez construit. Vous prendrez soin de dessiner le graphe construit par les algorithmes et d'indiquer ce qui est fait à chaque itération.

iv Pour $1 \leq k \leq |S|$, on note $T_k = (S_k, A_k)$ le sous-graphe de G construit par l'algorithme de Prim à l'itération k. Montrer qu'à toute itération k de l'algorithme de Prim, il existe un arbre couvrant de poids minimum, $T_{\min}$, tel que T_k est un sous-arbre de $T_{\min}$. En déduire la convergence de l'algorithme de Prim.

3.3 Recherche d'un flot maximum

Dans toute cette section, on considère un graphe orienté simple et antisymétrique $G = [S, A]$ dans lequel on associe une capacité positive K_a à chaque arc $a \in A$ (K_a est potentiellement infinie). Par ailleurs G est muni d'un *sommet source* s et d'un *sommet puits* t, tels que s n'a pas de prédécesseur et t n'a pas de successeur.

3.3.1 Flot, flot compatible et flot maximum

Un *flot* dans un graphe est une valuation des arcs de G respectant la loi de conservation des flux, aussi appelée *loi de Kirchhoff*. La loi de Kirchhoff consiste à imposer que la somme des flots sur les arcs entrants dans un sommet est égale à la somme des flots sur les arcs sortant de ce sommet, pour tout sommet autre que la source et le puits. Il s'agit d'une loi physique valable pour un grand nombre de problème impliquant le transport de matière. Dans l'étude d'un réseau de transport routier, il sera par exemple naturel d'imposer que la somme des véhicules entrant dans un carrefour est égale à la somme des véhicules qui en sortent. Plus formellement, un flot peut être défini comme une fonction $f : A \to \mathbb{R}$ telle que

$$\forall u \in S \setminus \{s, t\} : \sum_{a \in \delta^-(u)} f(a) = \sum_{a \in \delta^+(u)} f(a).$$

Par ailleurs, on dira qu'un flot f est *compatible* (ou *réalisable*) s'il est positif et respecte les capacités des arcs, c'est-à-dire si $0 \leq f(a) \leq K_a, \forall a \in A$. Par exemple, le flot nul respecte la loi de Kirchhoff et il est compatible quelles que soient les capacités.

Proposition 3.1. Théorème de décomposition des flots

Tout flot $s-t$ non-nul et à valeurs positives peut s'écrire comme une combinaison linéaire à coefficients positifs de flots dont chacun vaut 1 sur un chemin élémentaire de s à t ou sur un circuit élémentaire et 0 sur les autres arcs.

Démonstration : *Soit f un flot non-nul à valeurs positives sur G. Il existe un arc (s_0, s_1) sur lequel le flot est strictement positif. Par conservation des flux en s_1, le flot sortant de s_1 est > 0 à moins que $s_1 = t$. Si $s_1 \neq t$, on peut alors trouver un arc (s_1, s_2) sur lequel le flot est strictement positif. Par récurrence, on construit donc un chemin $(s_0, \dots, s_n)$ de longueur n le long duquel le flot est strictement positif ou un chemin $(s_0, \dots, s_p)$ de longueur $p < n$ tel que $s_p = t$. Dans le second cas, on répète le processus mais en cherchant un arc entrant dans s_0, (s_{-1}, s_0) sur lequel le flot > 0 (il existe par conservation des flux en s_0 à moins que $s_0 = s$). On prolonge donc le chemin construit ci-dessus pour former le chemin $(s_{-q}, \dots, s_0, \dots, s_p)$ sur lequel le flot est strictement positif. On s'arrête dès que $p - q = n$ ou $s_{-q} = s$.*

À l'issue du du processus décrit ci-dessus, on a donc un chemin de longueur n ou un chemin de s à t de longueur $< n$ le long duquel le flot est > 0.

i Si le chemin est de longueur n, il contient nécessairement un circuit élémentaire et l'on note $r > 0$ la valeur du plus petit flot le long de ce circuit. On a alors $f = f_1 + rc_1$ ou c_1 est un flot valant 1 sur ce circuit et 0 ailleurs et $f_1 = f - rc_1$. Par ailleurs, $f - rc_1$ est nécessairement nul sur l'arc où le flot valait r donc f_1 est nul sur au moins un arc de plus que f.

ii Si le chemin va de s à t, il peut aussi contenir un circuit, et dans ce cas on peut aussi exprimer f sous la forme $f_1 + rc_1$. Si le chemin ne contient pas de circuit, il s'agit d'un chemin élémentaire. On note $r > 0$ le plus petit flot le long de ce chemin, et l'on exprime f sous la forme $f_1 + rp_1$ où p_1 vaut 1 sur les arcs du chemins et 0 ailleurs. De plus, $f - rp_1$ vaut 0 sur au moins un arc de plus que f (celui dont le flot valait r).

Au final, en répétant ce processus, on identifie à chaque fois un circuit ou un chemin $s-t$ élémentaire le long duquel le flot est strictement positif et l'on augmente strictement le nombre d'arcs sur lequel le flot est nul. La décomposition est donc effectuée après au plus m répétitions. Faite en cours.

Corollaire 3.2.

Le flot sortant de s est égal au flot entrant dans t.

Démonstration : *Après avoir utilisé le théorème de décomposition, on montre que le flot sortant de s est égal au flot le long des chemins élémentaires de s à t. Le résultat se déduit par conservation du flux le long de ces chemins.*

Exercice 18. *On représente ci-dessous un graphe avec capacité muni d'une source s et d'un puits t : les capacités sont entre crochets et l'arc de retour est en pointillés.*

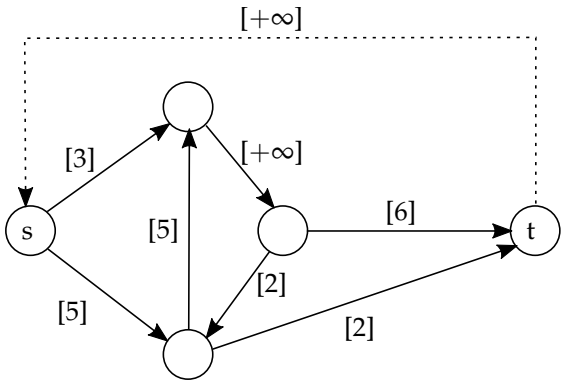


Figure 12: Graphe orienté avec capacités

Dessiner un flot compatible non nul sur ce graphe.

Définition 3.3. Valeur d'un flot

La valeur d'un flot, f, notée $\mathrm{val}(f)$, est donnée par la valeur du flot sortant de s, ou de façon équivalente, la valeur du flot entrant en t.

Définition 3.4. flot $s-t$ maximum

Un flot $s-t$ maximum est un flot compatible de valeur maximum.

Remarque 6. *Du fait que le flot entrant dans t est égal au flot sortant de s, il est classique d'ajouter un* arc de retour (s,t) *avec une capacité infinie et d'imposer la loi de Kirchhoff sur* ***tous*** *les sommets y compris s et t.*

Exercice 19. *Trouver un flot $s-t$ maximum sur le graphe de la figure 12.*

Dans la suite, nous ne représenterons pas l'arc de retour. Ainsi, on reste dans la situation où s n'a pas de prédécesseurs et t n'a pas de successeurs.

Remarque 7. *L'étude faite dans ce chapitre peut être généralisée au cas où chaque arc a est muni d'une borne inférieure l_a et d'une borne supérieure u_a. La recherche d'un flot compatible n'est pas triviale dans ce cas puisque le flot nul n'est en général pas compatible. L'exercice 24 montre la voie à suivre pour transformer la recherche d'un flot compatible dans un tel graphe borné en la recherche d'un flot maximal dans un graphe muni uniquement de capacités.*

3.3.2 Coupe minimum

Définition 3.5. Coupe $s-t$

Une *coupe* $s-t$ est une partition de S en deux ensembles S_1 et S_2 tel que $s \in S_1$ et $t \in S_2$.
La *capacité* (ou plus généralement, la *valeur*) d'une coupe (S_1, S_2) est la somme des capacités des arcs allant de S_1 vers S_2. On la note $K(S_1, S_2)$.
Une *coupe* $s-t$ *minimum* est une coupe $s-t$ dont la capacité est minimum.

Remarque 8. *Pour rappel $\{S_1, S_2\}$ constitue une partition de S si et seulement si $S_1 \cup S_2 = S$ et $S_1 \cap S_2 = \emptyset$.*

Remarque 9. ***J'insiste sur le fait que l'on ne considère pas les arcs allant de S_2 vers S_1 dans le calcul de la capacité de la coupe !***

Exemple 6. *Rennes est traversée par 7 ponts enjambant la Vilaine. Une voiture voulant se rendre des Gayeulles (s) à Bréquigny (t) sans sortir de Rennes (ni emprunter la Rocade) passe nécessairement par l'un de ses ponts. Ainsi le flot maximum des Gayeulles à Bréquigny est borné par la somme des capacités (du nord vers le sud) des ponts enjambant la Villaine. Par ailleurs, on définit une coupe du réseau routier rennais (S_1, S_2) tel que S_1 inclue tous les nœuds du réseau situés au nord de la Villaine et S_2 ceux situés au sud de la Villaine. La capacité de cette coupe est elle aussi égale à la somme des capacités (du nord vers le sud) des ponts enjambant la Villaine.*

On peut désormais s'intéresser à un résultat classique de la théorie des graphes : le théorème flot max./coupe min. Pour une source s et un puits t bien identifiés, on note $F^{\max}$ la valeur du flot s-t maximum dans et $C^{\min}$ la capacité de la coupe s-t minimum.

Proposition 3.6. Lemme du Théorème flot max./coupe min.

On a $F^{\max} \leq C^{\min}$.

Démonstration : *Soit un flot maximum de s à t. En applicant le théorème de décomposition à f on peut écrire f comme une combinaison linéaire à coefficients positifs de flots unitaires sur des circuits élémentaires de G et sur des chemins élémentaires de s à t. Les circuits de la décomposition ne passent pas par s ni par t puisque ceux-ci n'ont que des arcs sortant ou entrant (respectivement). La valeur du flot maximum est donc égale à la somme des flots passant sur les chemins élémentaires. Par ailleurs pour tout coupe $s-t$, (S_1, S_2), chacun de ces chemins élémentaires contient un arc allant de S_1 vers S_2. Le flot maximum est donc borné par la somme des capacités des arcs allant de S_1 vers S_2, c'est-à-dire par la capacité de la coupe. Sachant que c'est vrai pour une coupe s-t quelconque, c'est également vrai pour une coupe minimum, ainsi $F^{\max} \leq C^{\min}$.*

Remarque 10. *Une autre preuve consiste à partir d'une coupe $s-t$ minimum (S_1, S_2) et de construire un flot de même valeur que la coupe pour montrer $F^{\max} \geq C^{\min}$. Pour cela, on procède en fait de façon similaire à ce que fait l'algorithme de Ford-Fulkerson ci-dessous. On cherche des chemins de s à t sur lesquels on fait passer autant de flot que possible jusqu'à avoir* saturé *tous les arcs allant de S_1 à S_2 (un arc est saturé lorsque le flot qui y passe est égal à sa capacité).*

3.3.3 Algorithme de Ford-Fulkerson

Définition 3.7. Graphe d'écart

À tout arc $(i,j) \in A$, on associe un arc de retour (j,i) et l'on note A^R l'ensemble des arcs de retour ($\left|A^R\right| = |A|$).
À tout flot f sur G on associe un graphe d'écart $G_f = [S, A_f]$ où A_f est défini de la façon suivante :

$$A_f = \{(i,j) \in A : f(i,j) < K_{(i,j)}\} \cup \{(j,i) \in A^R : f(i,j) > 0\}.$$

La capacité de l'arc $a \in A_f$ dans le graphe d'écart G_f est notée $K_a(G_f)$ et est calculée de la façon suivante. Pour $(i,j) \in A$ tel que $f(i,j) < K_{(i,j)}$, la capacité de (i,j) dans G_f est donnée par $K_{ij}(G_f) = K_{(i,j)} - f(i,j)$. Pour $(i,j) \in A$ tel que $f(i,j) > 0$, la capacité de $(j,i) \in A^R$ dans G_f est donnée par $K_{(j,i)}(G_f) = f(i,j)$.

Remarque 11. *On peut noter que si f est le flot nul, $G_f = G$.*

Le graphe d'écart permet de mieux modéliser les modifications que l'on peut apporter à un flot tout en conservant sa compatibilité. Une augmentation x du flot sur un arc (i,j) peut être vue comme faire passer un flot égal à x sur l'arc (i,j) dans G_f. Si ce flot respecte la capacité dans G_f, i.e., si $0 \leq x \leq K_{(i,j)} - f(i,j)$, on sait que $f(i,j) + x \leq K_{(i,j)}$, donc le flot continue à respecter la capacité de l'arc. Inversement, une diminution y du flot sur un arc (i,j) consiste à faire passer un flot égal à y sur l'arc (j,i) dans G_f. Si ce flot respecte la capacité dans G_f, i.e., si $0 \leq y \leq f(i,j)$, on sait que $f(i,j) - y \geq 0$, donc le flot reste positif. Ainsi, tout flot compatible dans G_f, Δf, correspond à une modification réalisable du flot f : pour tout $(i,j) \in A$, on augmente $f(i,j)$ de $\Delta f(i,j)$ si $(i,j) \in G_f$, et on diminue $f(i,j)$ de $\Delta f(j,i)$ si $(j,i) \in G_f$.

Exercice 20. *On considère le graphe déjà vu plus tôt sur lequel on fait passer un flot f. Sur la figure ci-dessous, on note $f_a/[K_a]$ le flot et la capacité de chaque arc.*

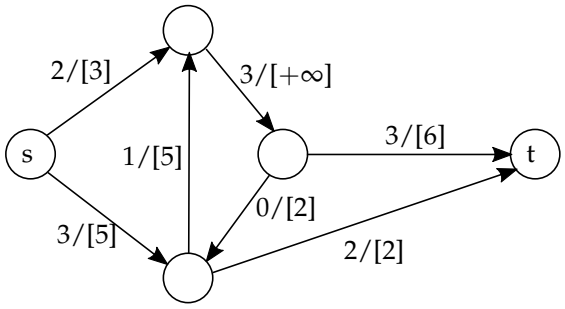


Figure 13: Exemple de flot compatible

i Donner le graphe d'écart associé à ce flot.

ii Trouver un chemin de s à t dans le graphe d'écart.

iii Quel est la valeur maximale du flot que l'on peut faire passer sur ce chemin ?

iv Modifier f à l'aide de ce flot. Quel changement observe-t-on sur la valeur du flot ?

Proposition 3.8.

Soit f un flot compatible sur G et Δf un flot compatible sur G_f tel que Δf vaut r sur un chemin élémentaire de s à t dans G et 0 ailleurs. Alors la modification de f associée à Δf donne lieu à une augmentation de la valeur du flot égale à r.

Démonstration : *Il suffit de remarquer que les arcs incidents à s sortent tous de s. Donc l'arc sortant de s dans le chemin élémentaire de G_f défini dans l'énoncé de la proposition est aussi dans G. Ainsi, on ajoute bien r à la valeur du flot sortant de s.*

Les éléments définis ci-dessus mènent naturellement à l'algorithme de Ford-Fulkerson. En partant d'un flot compatible f quelconque, on sait que si l'on trouve un chemin de s à t dans G_f, alors tout flot non-nul le long de ce chemin mène à une augmentation de la valeur du flot. On applique donc ce processus itérativement tant que cela est possible. Sa validité sera démontrée dans la preuve de l'algorithme.

```
1 Initialiser f_a := 0, ∀a ∈ A // on part du flot nul, toujours compatible
2 Initialiser G_f := G // le graphe d'écart est le graphe lui-même pour le flot nul
3 tant que il existe un chemin élémentaire L de s à t dans G_f faire
4 |   Soit K(L) := min_{a∈L} {K_a(G_f)} // valeur maximum du flot sur L dans G_f
  |   // Mise à jour du flot
5 |   pour (i, j) ∈ L faire
6 |   |   Si (i, j) ∈ A: f(i, j) := f(i, j) + K(L);
7 |   |   Si (i, j) ∉ A: f(j, i) := f(j, i) − K(L);
8 |   Mettre à jour G_f;
9 Retourner f;
```

Algorithme 13 : Algorithme de Ford-Fulkerson

Complexité standard : Si toutes les capacités sont entières, chaque itération donne lieu à une augmentation de la valeur du flot au moins égale à 1. Si l'on note $F^{\max}$ la valeur du flot max., l'algorithme effectue donc au plus $F^{\max}$ itérations. À chaque itération, on recherche un chemin de s à t dans G_f (complexité en $\mathcal{O}(m)$) et l'on met à jour le flot et le graphe d'écart le long de ce chemin (au plus m arcs donc complexité en $\mathcal{O}(m)$).

Au final la complexité de Ford-Fulkerson est en $\mathcal{O}(\mathbf{mF}^{\max})$.

Preuve de l'algorithme : À la fin de l'algorithme de Ford-Fulkerson, il n'existe plus de chemin allant de s à t dans le graphe d'écart. On note alors (S_1, S_2) la coupe $s-t$ telle que S_1 regroupe les sommets accessibles depuis s dans le graphe d'écart et S_2 est formé des autres sommets. Ainsi, l'on sait qu'il n'existe pas de chemin entre s et un sommet quelconque de S_2 dans G_f et que tout chemin allant de s à t dans G passe par au moins un arc allant de S_1 à S_2 (ce dernier point est vrai pour toute coupe $s-t$). En sommant sur tous les chemins, on déduit que la somme des flots sur les arcs de S_1 à S_2 est au moins égale à la valeur du flot f. De plus, par définition de la coupe, ces arcs sont saturés par f sinon certains éléments de S_2 seraient accessibles à partir de s dans G_f. On en déduit que la somme des flots sur les arcs de S_1 à S_2 est exactement égale à la valeur de la coupe (S_1, S_2).

En notant F, la valeur du flot f et C la valeur de la coupe (S_1, S_2), il vient que $C \leq F$. Sachant que l'on a déjà démontré que $F^{\max} \leq C^{\min}$, cela n'est possible que si $F = F^{\max} = C^{\min} = C$. Laissée en exercice.

La preuve de l'algorithme de Ford-Fulkerson permet de conclure la preuve du théorème de flot max./coupe min. et elle donne également une méthode pour construire une coupe min. en même temps qu'un flot max. Ce dernier point est tout aussi important à retenir car des problèmes pratiques apparaîtront parfois plus naturellement sous la forme d'une recherche de coupe min. que de flot max.

Proposition 3.9. Théorème flot max./coupe min.

La valeur du flot maximum entre s et t est égale à la capacité de la coupe s-t minimum.

Démonstration : *Il s'agit de montrer que l'on a $F^{\max} = C^{\min}$ alors que l'on a déjà vu que $F^{\max} \leq C^{\min}$.*

Pour montrer l'égalité, on peut se référer directement à la preuve de l'algorithme de Ford-Fulkerson ci-dessus. À la fin de l'algorithme, on montre en effet que l'on a construit une coupe s-t et un flot s-t de mêmes valeurs. En notant F et C les valeurs de ce flot et de cette coupe, on a évidemment $F \leq F^{\max}$ et $C^{\min} \leq C$. Donc par l'inégalité déjà acquise :

$$F \leq F^{\max} \leq C^{\min} \leq C.$$

Si $F = C$, on obtient nécessairement $F^{\max} = C^{\min}$.

Le paramètre $F^{\max}$ utilisé dans l'étude de la complexité de l'algorithme n'est pas une donnée du problème. Si les capacités sont finies, les capacités des arcs sortant de s et celles des arcs entrant dans t donnent des bornes sur $F^{\max}$. Toutefois, on peut a priori faire grandir ces bornes autant que l'on veut pour un même graphe. La complexité ne dépend plus seulement du nombre de données d'entrée, mais aussi de la valeur numérique de ces données. On dira dans ce cas que l'on a affaire à une complexité *pseudo-polynomiale*. Ce cas est bien entendu beaucoup moins favorable qu'une complexité polynomiale dans le cas général.

Exemple 7. *On considère l'exemple ci-dessous où K est un entier quelconque.*

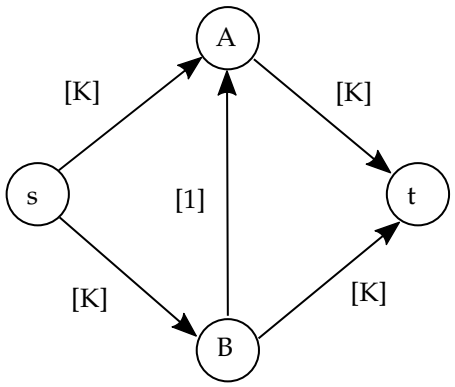


Figure 14: Graphe sur lequel l'application de Ford-Fulkerson peut mal tourner

Si l'on fait augmenter le flot le long du chemin (s, B, A, t), le flot augmente au plus de 1 du fait de la capacité sur (A, B). Le graphe d'écart mis à jour ne contient pas (B, A), mais il contient (A, B) avec une capacité égale à 1. Le chemin (s, A, B, t) existe dans le graphe d'écart, et l'on peut aussi y faire passer un flot de 1. Ainsi, une nouvelle itération de Ford-Fulkerson permet augmente à nouveau le flot de 1 et le graphe d'écart contiendra l'arc (B, A). On comprend bien que si l'on continue comme cela, on parvient à la solution optimale en $2K$ itérations où K peut être arbitrairement grand.

Si toutefois, on choisit plutôt de faire augmenter le flot le long de (s, A, t) puis le long de (s, B, t), on parvient au même résultat en 2 itérations.

Vers une complexité optimale : *L'exemple ci-dessus illustre la possibilité d'améliorer l'algorithme de Ford-Fulkerson. En fait, si l'on choisit un chemin de s à t dont le nombre d'arcs est minimum, il suffit d'au plus $nm/2$ itérations pour converger. Sachant qu'un parcours en largeur produit naturellement un tel chemin lorsque t est accessible depuis s, il est facile de modifier Ford-Fulkerson pour atteindre une complexité en $\mathcal{O}(nm^2)$.*

Exercice 21. *Montrer que l'on peut effectivement trouver le chemin de s à t contenant le moins d'arcs à l'aide d'un parcours en largeur (par récurrence, en se référant aux niveaux d'un arbre de parcours en largeur).*

Exercice 22. *Appliquer l'algorithme de Ford-Fulkerson pour trouver un flot maximum sur le graphe de la figure 13.*

Exercice 23. *Appliquer l'algorithme de Ford-Fulkerson pour trouver le flot maximum entre les sommets 1 et 6 dans le graphe de la figure 15. On utilisera un algorithme de parcours en largeur pour détecter les chemins allant de la source au puits. Préciser la coupe de capacité minimum obtenue à la fin de l'algorithme.*

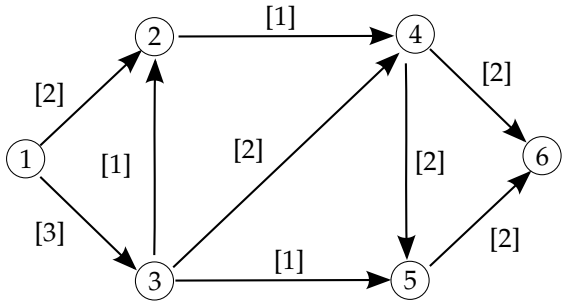


Figure 15: Graphe de l'exercice 23

Exercice 24. *Le problème du flot réalisable consiste à identifier un flot f dans un graphe orienté $G = (S, A)$ tel que :*

$$\sum_{j:(i,j)\in A} f_{ij} - \sum_{j:(j,i)\in A} f_{ji} = D(i), \qquad \forall i \in S$$

$$0 \leq f_{ij} \leq U_{ij}, \qquad \forall (i,j) \in A$$

où $D(i)$ est un entier donné pour chaque $i \in S$.

i (a) *Donner une condition nécessaire sur la somme des* $D(i), i \in S$, *pour qu'il existe un flot réalisable.*

(b) *Ajouter un sommet source* s, *un sommet puits* t *et un ensemble d'arcs tels que le problème puisse être résolu par la recherche d'un flot maximum dans le graphe obtenu. Bien expliquer et justifier le lien entre les deux problèmes.*

ii On considère maintenant le problème de flot compatible à variables bornées, à savoir :

$$\sum_{j:(i,j)\in A} f_{ij} - \sum_{j:(j,i)\in A} f_{ji} = 0, \qquad \forall i \in S \setminus \{s,t\}$$
$$L_{ij} \leq f_{ij} \leq U_{ij}, \qquad \forall (i,j) \in A$$

On suppose ici que toutes les bornes inférieures sont positives ou nulles.

(a) *Adapter la méthode vue ci-dessus pour trouver un flot compatible avec les bornes. Pour cela, on se ramènera à un problème de flot réalisable équivalent par un changement de variable pertinent.*

(b) *Appliquer cet algorithme pour trouver un flot compatible entre les sommets 1 et 7 dans le graphe de la figure 16.*

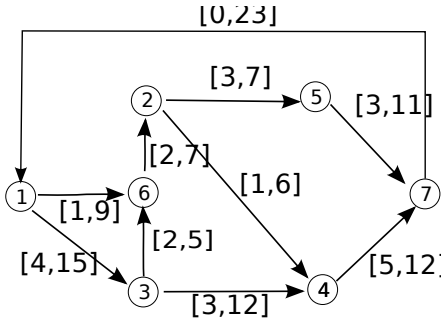


Figure 16: Graphe de l'exercice 24

3.4 Problème d'affectation simple

Définition 3.10. Problème d'affectation simple

Soient I et J deux ensembles finis de même cardinal : $|I| = |J| = n$. Soit une matrice de coût $C \in (\mathbb{R}_+)^{n \times n}$. Le problème d'affectation simple est la recherche d'une bijection $\sigma : I \to J$ minimisant

$$\sum_{i \in I} C_{i,\sigma(i)}.$$

Dit autrement, dans cette section, nous chercherons à affecter chaque élément d'un ensemble I à un élément d'un ensemble J de façon à ce que chaque élément de J soit affecté à un seul élément de I et que le coût d'affectation soit minimum. En se focalisant sur la matrice de coût, on peut aussi voir le problème comme la recherche de n éléments de la matrice tels qu'il y en ait un par ligne et un par colonne et que leur somme soit minimum.

Exemple 8. *Des parents organisent le dîner de Noël chez eux ce soir et ils comptent mettre à contribution leurs quatre enfants : Anne, Bob, Cécile et Daniel. Les parents souhaitent demander à leurs enfants de faire le ménage, dresser la table, s'occuper du vin et faire un gâteau. Selon les affinités, l'âge et la débrouillardise de chacun, les parents notent l'inaptitude de chacun pour chaque tâche par un entier positif :*

	ménage	*table*	*vin*	*gâteau*
A	10	10	0	5
B	5	5	$+\infty$	5
C	2	2	5	5
D	5	2	$+\infty$	2

Pour justifier les valeurs apparaissant dans la colonne du vin, les parents avancent par exemple qu'Anne fait des études pour devenir sommelière, Bob n'a que 13 ans, et, pour une raison qui leur échappe, Daniel n'aime pas l'alcool. À qui confier chaque tâche pour que le dîner soit aussi réussi que possible ?

Dans une première étape, on s'aperçoit qu'une analyse de la fonction de coût donne déjà certaines informations sur la solution optimale.

Proposition 3.11. Calcul immédiat d'une borne inférieure

Les deux valeurs

$$\sum_{i \in I} (\min_{j \in J} C_{i,j}) \quad \text{et} \quad \sum_{j \in J} (\min_{i \in I} C_{i,j})$$

sont des bornes inférieures sur la valeur de l'affection de coût minimum. Ainsi, on obtient une borne inférieure sur le coût d'affectation minimum en considérant :

$$\max \left\{ \sum_{i \in I} \left(\min_{j \in J} C_{i,j} \right) \; ; \; \sum_{j \in J} \left(\min_{i \in I} C_{i,j} \right) \right\}.$$

Démonstration : *La première des deux sommes correspond à l'affectation de chaque élément de I à l'élément de J qui minimise le coût d'affectation. On a donc bien affecté chaque élément de I, mais pas nécessairement chaque élément de J. La seconde correspond à l'affectation de chaque élément de J à l'élément de I qui minimise le coût d'affectation. On est donc dans la situation inverse où tous les éléments de J sont affectés, mais pas forcément tous ceux de I. Dans chacun des cas, une affectation réalisable (i.e. bijective) est nécessairement de coût supérieure ou égal. Faite en cours.*

Remarque 12. *Nous verrons dans le cours Recherche opérationnelleque chacune de ces deux bornes est en fait la valeur optimale d'une relaxation du problème. Dans ce cas, une relaxation est une version moins contrainte du problème initial. Ayant moins de contraintes, la relaxation a nécessairement une meilleure solution optimale.*

Proposition 3.12.

Soit σ^* une affectation de coût minimum pour la matrice C et soit $r \in \mathbb{R}$. Alors σ^* reste optimale pour la matrice de coût obtenue en ajoutant r à tous les éléments d'une ligne de C. Il en va de même si l'on ajoute r aux éléments d'une colonne.

Démonstration : *Soit $i' \in I$ et considérons la matrice de coût C' obtenue en ajoutant r à la ligne i' de C. Pour toute affectation σ, cette nouvelle matrice donne un coût :*

$$\sum_{i \in I} C'(i) = \sum_{i \neq i'} C_{i,\sigma(i)} + (C(i', \sigma(i')) + r) = r + \sum_{i \in I} C_{i,\sigma(i)}.$$

Le coût de l'affectation pour C' est donc le même que son coût pour C à une constante près. Ainsi, une affectation de coût minimum pour C l'est aussi pour C' et réciproquement. Faite en cours.

Proposition 3.13.

Soit σ une affectation et C une matrice de coûts positifs. Si pour tout $i \in I$, $C(i, \sigma(i)) = 0$), alors σ est optimale.

Les propositions ci-dessus donnent le principe et la motivation d'une méthode de réduction de la matrice de coûts. En effet, on a le droit d'ajouter ou de soustraire une même valeur aux lignes et aux colonnes de C sans changer l'affectation optimale. De plus, si la matrice reste positive, l'affectations de n zéros dans la matrice est optimale. On est donc en droit de chercher des opérations sur les lignes et les colonnes qui feront apparaître autant de zéros que possible dans la matrice de coût. Une solution étant à portée si l'on est capable d'en choisir n avec exactement un zéro par ligne et par colonne.

De façon à faire apparaître au moins un zéro dans chaque ligne et chaque colonne tout en conservant la positivité des éléments de la matrice, on soustrait à chaque ligne et à chaque colonne son élément minimum. Cela est décrit par le pseudo-code suivant.

pour $i \in I$ **faire**
  Soit $r = \min_{j \in J} C_{i,j}$;
  Soustraire r à tous les éléments de la ligne i de C;
pour $j \in J$ **faire**
  Soit $r = \min_{i \in I} C_{i,j}$;
  Soustraire r à tous les éléments de la colonne j de C;

Algorithme 14 : Réduction de la matrice de coûts

Une fois la matrice de coûts réduite, il est naturel de se demander le nombre de zéros que l'on peut sélectionner dans la matrice de coûts sans excéder un zéro par ligne ou par colonne. On note $\bar{C}$ la matrice de coût réduite. On considère ensuite le graphe d'affectation $G = [S, A]$ où

- $S = \{s, t\} \cup I \cup J$,
- $A = \{(s, i) : i \in I\} \cup \{(j, t) : j \in J\} \cup \{(i, j) : \bar{C}_{i,j} = 0\}$,
- tous les arcs ont une capacité égale à 1.

Exemple 9. *On peut regarder la matrice de coût*

$$C = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 4 & 4 \\ 1 & 3 & 5 \end{pmatrix}$$

Après réduction par les lignes et par les colonnes, on obtient

$$\bar{C} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$

À partir de cette matrice, on construit le graphe d'affectation suivant.

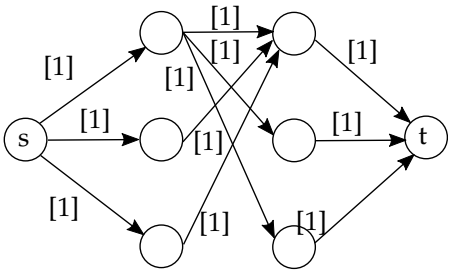


Figure 17: Graphe d'affectation

Proposition 3.14.

La valeur du flot $s - t$ maximum dans le graphe d'affectation est égale au nombres maximum de zéros que l'on peut affecter dans la matrice de coûts réduite $\bar{C}$. Les zéros affectés sont positionnés sur les éléments (i, j) correspondant aux arcs (i, j) du graphe où le flot max. est non nul (donc égal à 1).

Démonstration : *Tout chemin de s à t du graphe d'affectation est constitué de trois arcs (s,i), (i,j) et (j,t) où (i,j) correspond à un élément nul de la matrice réduite $\bar{C}$ ($\bar{C}_{i,j}=0$). Donc un flot de 1 sur un chemin de s à t correspond à la selection d'un zéro de la matrice réduite. Sachant que pour $i \in I$, il n'y a qu'un arc entrant dans i et qu'il est de capacité 1, le flot traversant i est au plus égal à 1. De même, le flot traversant $j \in J$ est au plus égal à 1. Ainsi, il ne peut y avoir qu'un zéro sélectionné pour chaque ligne et pour chaque colonne. Sachant par ailleurs que le graphe est acyclique, le théorème de décomposition assure que tout flot est la somme de tels flots. Ainsi, un flot max. de valeur F correspond à l'affectation de F zéros tel qu'au plus* ***un*** *zéro est sélectionné sur chaque ligne et chaque colonne.*

Nous sommes désormais prêts pour la description de l'algorithme Hongrois, découvert par le mathématicien Harold Kuhn.

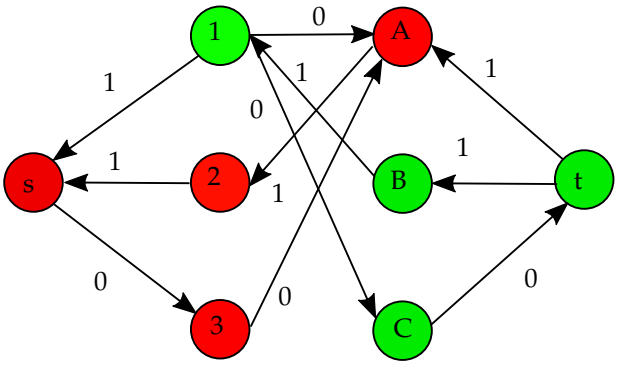
1 Initialiser $\bar{C}$ par réduction de la matrice de coûts C;
2 **tant que** *l'on n'a pas trouvé une affectation de n zéros dans $\bar{C}$* **faire**
3 Réduire la matrice de coûts $\bar{C}$`// facultatif pour la convergence`
4 $f \leftarrow$ flot max dans $G_{\bar{C}}$;
`// si l'on n'a pas affecté suffisamment d'éléments, on modifie` $\bar{C}$
5 **si** *moins de n éléments de I sont affectés* **alors**
6 Soit (S_1, S_2) la coupe min. déduite du graphe d'écart $G_{\bar{C}}(f)$;
7 Soit $r = \min_{i \in I \cap S_1, j \in J \cap S_2} \bar{C}_{i,j}$;
8 Soustraire r aux lignes $i \in S_1$ de $\bar{C}$ `//` $\bar{C}_{i,j} \geq 0, \forall i \in S_1, j \in S_2$`;`
9 Ajouter r aux colonnes $j \in S_1$ de $\bar{C}$ `//` $\bar{C}_{i,j} \geq 0, \forall i \in S_1, j \in S_1$`;`
10 Pour tout $(i,j) \in G_{\bar{C}}$ tel que $f(i,j)=1$, affecter $\sigma(i)=j$;
11 Retourner σ et son coût $\sum_{i \in I} C_{i,\sigma(i)}$`// on évalue bien le coût avec la matrice initiale`

Algorithme 15 : Algorithme Hongrois

Avant de passer à la preuve de l'algorithme, on revient sur les étapes 6 à 9. Si le flot max. dans le graphe d'affectation a une valeur strictement inférieure à n, on sait qu'il n'est pas possible de sélectionner n zéros de la matrice réduite sans en prendre plus d'un par ligne ou par colonne. Dans ce cas, l'algorithme va tâcher de modifier la matrice réduite pour y faire apparaître de nouveaux zéros, susceptibles d'être sélectionnés. Cette modification s'appuie sur une coupe min. dans le graphe d'affectation. Cette coupe (S_1, S_2) est celle déjà introduite lors de la présentation de Ford-Fulkerson : S_1 est l'ensemble des sommets accessibles depuis s dans le graphe d'écart final et S_2 contient les autres sommets.

Pour donner du sens à cette coupe, on peut d'abord remarquer que s'il n'y a pas de zéro sélectionné dans une ligne i, le flot sur (s,i) est nul donc (s,i) est encore dans le graphe d'écart de $G_{\bar{C}}$. Ainsi S_1 contient au moins tous les sommets des lignes sans zéro sélectionné. Ensuite, si le zéro de l'élément (i,j) est sélectionné, l'arc (j,i) est dans le graphe d'écart : c'est donc le seul arc entrant dans i (car il y a un flot 1 sur (s,i) et 0 sur les autres arcs issus de i). On déduit donc que i et j sont tous les deux dans S_1 ou tous les deux dans S_2. Il vient que pour les éléments (i,j) sélectionnés, on a $\{i,j\} \subset S_1$ ou $\{i,j\} \subset S_2$. Ainsi, les modifications de la matrice réduite conservent les zéros sélectionnés. En revanche, elles font apparaître au moins un nouveau zéro dans un élément (i,j) tel que $i \in S_1$ et $j \in S_2$ (et elles en font potentiellement disparaître d'autres dans des éléments $i \in S_2$ et $j \in S_1$).

Exemple 10. *À la fin de l'exécution de Ford-Fulkerson sur le graphe d'affectation de la figure 17, on trouve le graphe d'écart ci-dessous. Les indices des lignes sont notés avec des chiffres et ceux des colonnes avec des lettres. Les valeurs qui apparaissent au-dessus des arcs sont les flots sur le graphe d'affectation. Les sommets en rouge sont les sommets de S_1 et ceux en vert sont dans S_2.*



Ainsi, dans S_1, il y a deux lignes (2 et 3) et une colonne (A). Lors de la modification, on va soustraire $r = 1$ des éléments $(2, B)$, $(2, C)$, $(3, B)$ et $(3, C)$, et ajouter r à l'élément $(1, A)$. On obtient :

$$\bar{C} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Deux zéros supplémentaires sont apparus et un a disparu.

Preuve de l'algorithme : On se place à une itération quelconque de la boucle principale de l'algorithme de Kuhn. Si la recherche de flot max. permet d'affecter tous les éléments de I, l'algorithme a bien trouvé la solution optimale d'après les propositions démontrées plus haut.

On suppose maintenant que le flot max. ne permet pas d'affecter tous les éléments de I. La suite de l'algorithme modifie la matrice réduite par les étapes 6 à 9. Par la discussion qui précède la preuve, on fait les observations suivantes :

i Ces étapes ne modifient pas les arcs du graphe d'affectation sur lesquels le flot max. est égal à 1 (c'est-à-dire les zéros de $\bar{C}$ sélectionnés). La valeur du flot max. ne peut donc pas diminuer suite à ces opérations.

ii Chaque nouveau zéro qui apparaît suite à ces modifications est sur un élément (i, j) où $i \in S_1$ et $j \in S_2$.

iii Si des zéros disparaissent, ils sont sur des éléments (i, j) tels que $i \in S_2$ et $j \in S_1$.

Par la seconde observation, on obtient que si l'on répercute les modifications de $\bar{C}$ sur le graphe d'écart, les ajouts d'arcs permettent d'accéder à de nouvelles colonnes depuis s, et les arcs supprimés n'empêchent pas d'accéder à des sommets de S_1. En conséquence, l'ensemble S_1 grandit strictement suite aux étapes 6 à 9.

À l'itération suivante, lors du nouvel appel à Ford-Fulkerson, on peut repartir de ce graphe d'écart. Deux alternatives existent : soit il existe un nouveau chemin de s à t dans ce graphe d'écart pour faire augmenter le flot max., soit le graphe d'écart correspond à une solution optimale donc S_1 contient au moins un élément de plus qu'à l'itération précédente. Sachant que S_1 contient au plus $2n + 1$ sommet, on sait que le flot max. augmente de 1 au plus toutes les $2n$ itérations. Cela garantit que l'algorithme finit par produire un flot max. de valeur n correspondant à une affectation de tous les éléments de I après au plus $2n^2$ itérations. Faite en cours.

Complexité standard : À chaque itération de la boucle principale, on réduit la matrice de coût en $\mathcal{O}(n^2)$ opérations, on met à jour G_C en $\mathcal{O}(n^2)$ opérations et l'on effectue l'affectation en $\mathcal{O}(n^2)$ opérations. Ensuite, pour la modification de $\bar{C}$, il faut déterminer la coupe min. On a déjà vu que cela se fait par un parcours dans le graphe d'écart, donc en $\mathcal{O}(n^2)$ opérations ici. Le calcul de r, puis les additions et soustractions de cette valeur se font aussi en $\mathcal{O}(n^2)$ opérations. Par ailleurs, on a déjà vu dans la preuve de l'algorithme qu'il convergeait en au plus $2n^2$ itérations. Ainsi, ces étapes de l'algorithme s'exécutent en $\mathcal{O}(n^4)$ au total.

Il reste maintenant les recherches de flot max. Comme discuté dans la preuve de l'algorithme, à une itération donnée, on peut repartir du flot max. trouvé à l'itération précédente. Cela signifie qu'au total, on a besoin de faire au plus n itérations de Ford-Fulkerson donnant lieu à une augmentation de la valeur du flot. De plus, à chaque itération, il faut aussi faire le parcours du graphe d'écart qui montre qu'il n'y plus de chemin de s à t. On sait qu'une itération de Ford-Fulkerson est en $\mathcal{O}(m)$ où m est le nombre d'arcs et qu'il en va de même pour le parcours du graphe d'écart. Ce sont donc les parcours effectués à chaque itération qui sont prépondérants dans le temps d'exécution. Ici $m = n^2$, donc la complexité est en $\mathcal{O}(n^4)$ sur l'ensemble de l'algorithme.

Au final, l'algorithme Hongrois s'exécute en $\mathcal{O}(n^4)$. (démonstration faite en cours).

L'algorithme Hongrois s'exécute en $\mathcal{O}(\mathbf{n^4})$.

Exercice 25. *On veut affecter 5 projets à 5 ingénieurs de façon à minimiser la somme des durées des projets. Le temps que chaque ingénieur passerait sur chaque projet est donné par le tableau suivant :*

	projet 1	*projet 2*	*projet 3*	*projet 4*	*projet 5*
ingénieur 1	*15*	*40*	*5*	*20*	*15*
ingénieur 2	*22*	*33*	*9*	*16*	*15*
ingénieur 3	*40*	*6*	*28*	*0*	*21*
ingénieur 4	*9*	*1*	*8*	*26*	*45*
ingénieur 5	*10*	*10*	*60*	*15*	*0*

Trouver l'affectation optimale et son coût en appliquant l'algorithme Hongrois.

Exercice 26. *Appliquer l'algorithme Hongrois (algorithme de Kuhn) pour résoudre les problèmes d'affectations correspondant aux deux matrices de coût suivantes :*

$$\begin{pmatrix} +\infty & 8 & 7 \\ 7 & 6 & 4 \\ 3 & 3 & 3 \end{pmatrix} \qquad \begin{pmatrix} 17 & 15 & 9 & 5 & 12 \\ 16 & 16 & 10 & 5 & 10 \\ 12 & 15 & 14 & 11 & 5 \\ 4 & 8 & 14 & 17 & 13 \\ 13 & 9 & 8 & 12 & 17 \end{pmatrix}$$

Vous exécuterez d'abord l'algorithme en recherchant des affectations nulles dans les matrices réduites de façon heuristique (commencer par réduire sur les lignes pour moins d'itérations).

Dans un second temps, pour la plus petite des deux matrices, vous exécuterez à nouveau l'algorithme en recherchant itérativement un flot maximum (par Ford-Fulkerson) dans un graphe que vous décrirez. La mise à jour de ce graphe se fera en identifiant une coupe minimum dans le graphe.

4 Programmation linéaire en nombre entiers

Les programmes linéaires en nombre entiers (ou PLNE, en anglais, Integer (Linear) Programs ou IPs) sont une classe importante de problèmes d'optimisation combinatoire.

Définition 4.1. Programme en nombre entiers

Un programme linéaire en nombre entiers est un problème de minimisation d'une forme linéaire sur l'intersection entre $\mathbb{Z}$ et un polyèdre convexe. Dit autrement, un PLNE est un problème d'optimisation combinatoire qui peut être formulé comme suit :

$$IP : \begin{cases} \text{min. } c^T x \\ \qquad Ax \leq b, \\ \qquad x \in \mathbb{Z}^n_+, \end{cases}$$

où $c \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}$.

4.1 Familles particulières de PLNE et extensions

La famille la plus importante des PLNE est celle des programmes binaires. Dans un programme binaire, les contraintes incluent $-x \leq 0$ et $x \leq 1$. Cette famille comprend la plupart des problèmes qui peuvent être formulés avec des graphes. Les programmes binaires sont simplement formulés avec $x \in \{0,1\}^n$ au lieu de $x \in \mathbb{Z}^n$.

Une extension naturelle de la programmation en nombre entiers est la programmation linéaire mixte (mixed integer programming), dans laquelle on impose à une partie des variables d'être des entiers. Formellement,

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \text{ avec } x_1 \in \mathbb{Z}^{n_1}, x_2 \in \mathbb{R}^{n-n_1}.$$

Dans tous ces cas, il s'agit d'optimisation non-différentiable, donc les méthodes de gradient étudiées dans le cours d'optimisation ne peuvent pas fonctionner.

4.2 Pourquoi étudier cette classe de problèmes ?

La première raison est pratique : un grand nombre d'applications réelles peut être modélisée par des PNLE, dans des domaines variés[1] :

- L'industrie : production, télécommunications, transport, informatique
- Les services publics : hôpitaux, transports publics, etc.
- La défense: logistique, planification, etc.
- La finance: gestion de portefeuille

De plus, la programmation en nombre entiers est un champ d'optimisation mature, dans lequel des algorithmes efficaces ont été développés, et beaucoup de recherche est encore en cours. Les méthodes spécifiques et les résultats théoriques présentés dans ce cours s'appuient sur des techniques de programmation

[1] voir notamment les sites `http://www.artelys.com/`, `http://www.eurodecision.fr/` et `http://www.giro.ca/`.

mathématique pour lesquelles la convergence est garantie. Pour cela, il est généralement nécessaire d'énumérer une partie suffisamment large de l'ensemble des solutions, et la difficulté est de réduire autant que possible l'ensemble à énumérer.

Une autre approche consiste à se centrer sur des ensembles beaucoup plus petits en exploitant l'intuition, ou par mimétisme avec des processus naturels. Ces méthodes appartiennent à la famille des algorithmes *meta-heuristiques*. Ils ne seront pas présentés dans ce cours, mais ils sont intéressants à étudier quand on s'intéresse à des applications de la vie réelle pour lesquelles des "bonnes" solutions peuvent avoir presque autant de valeur que des solutions optimales.

Pour certains problèmes, les algorithmes de programmation sous contrainte peuvent être plus efficaces que les algorithmes de programmation mathématique pour trouver la solution optimale. Ces méthodes sont aussi basées sur des techniques d'énumération, mais elles se centrent sur la propagation de l'information quand on explore l'arbre d'énumération.

4.3 Modélisation par un programme en nombre entiers

On rappelle d'abord la définition d'un programme linéaire.

Définition 4.2. Programme linéaire

Un *programme linéaire* (PL) est un problème d'optimisation convexe qui peut être formulé sous la forme :

$$PL : \begin{cases} \min. \ c^T x \\ \quad Ax \leq b, \\ \quad x \geq 0, \end{cases}$$

où $c \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}$.

Lorsque c'est possible, c'est généralement intéressant de modéliser un problème sous forme de PL. Pourtant, le cadre de la programmation linéaire peut être étroit. Même si les programmes linéaires en nombre entiers ont des fonctions objectifs et des contraintes linéaires, ils élargissement considérablement la capacité de modélisation de la programmation linéaire.

Remarque 13. *Il est essentiel de comprendre que le terme "linéaire" fait référence aux variables x. Ceci implique que*

- *les coefficients multiplicatifs devant toutes les variables sont constants,*
- *on n'autorise pas de produits de variables,*
- *l'ensemble des solutions admissibles est continu, convexe et fermé.*

Tout d'abord, la programmation en nombre entiers permet de modéliser des décisions discrètes.

- Combien ? $\Rightarrow x \in \mathbb{N}$
- Oui ou non ? $\Rightarrow x \in \{0, 1\}$
- Choisir un élément dans un ensemble

$$x \in \{v_1, v_2, \ldots, v_N\}$$

$$\downarrow$$

$$y_1, y_2, \ldots, y_N \in \{0, 1\}$$
$$x = v_1 y_1 + v_2 y_2 + \cdots + v_N y_N$$
$$y_1 + y_2 + \cdots + y_N = 1$$

Exercice 27. *Un artisan est spécialisé dans la production de deux biens A et B, et il veut investir dans une machine pour accélérer sa production. Il a le choix entre deux machines 1 et 2 dont les coûts annuels sont respectivement c_1 et c_2, et il estime qu'il sera capable de produire a_1 produits A et b_1 produits B avec la machine 1 et a_2 et b_2 produits avec la machine 2. Le prix de vente unitaire de A est p_A et celui de B est p_B.*

i Modéliser le problème que doit résoudre l'artisan pour choisir la machine qui maximisera ses bénéfices.

ii En sachant qu'il a fait 15000 euros de bénéfice l'an dernier, modifier le modèle pour s'assurer que l'investissement est profitable.

Exercice 28. *Gestion de portefeuille (tiré en grande partie de : L. A. Wolsey, Integer Programming, John Wiley and Sons, 1998)*

Un riche client est intéressé par sept investissements $\{1, \ldots, 7\}$. Modéliser les contraintes suivantes par des inégalités linéaires impliquant des variables binaires :

i Le client ne peut pas choisir tous les investissements à la fois.

ii Il doit choisir au moins l'un des investissements.

iii L'investissement 1 ne peut pas être choisi si l'investissement 3 est choisi.

iv L'investissement 4 ne peut être choisi que si l'investissement 2 est également choisi.

v Il doit investir à la fois dans 1 et 5 ou n'investir dans aucun des deux.

vi Il doit choisir au moins l'un des trois investissements 1, 2, 3 ou au moins deux des investissements 2, 4, 5, 6.

vii S'il choisit l'investissement 1 alors il doit aussi choisir les investissements 3, 5 et 7.