Frédéric Meunier

INTRODUCTION À LA RECHERCHE OPÉRATIONNELLE

Frédéric Meunier
Université Paris Est, CERMICS, Ecole des Ponts Paristech, 6-8 avenue Blaise Pascal, 77455 Marne-la-Vallée Cedex.
E-mail : frederic.meunier@cermics.enpc.fr

INTRODUCTION À LA RECHERCHE OPÉRATIONNELLE

Frédéric Meunier

TABLE DES MATIÈRES

CHAPITRE 1

GÉNÉRALITÉS

Présentation

La Recherche Opérationnelle (RO) est la discipline des mathématiques appliquées qui traite des questions d'utilisation optimale des ressources dans l'industrie et dans le secteur public. Depuis une dizaine d'années, le champ d'application de la RO s'est élargi à des domaines comme l'économie, la finance, le marketing et la planification d'entreprise. Plus récemment, la RO a été utilisée pour la gestion des systèmes de santé et d'éducation, pour la résolution de problèmes environnementaux et dans d'autres domaines d'intérêt public.

Exemples d'application. — Planifier la tournée d'un véhicule de livraison qui doit passer par des points fixés à l'avance puis revenir à son point de départ en cherchant à minimiser la distance parcourue est un problème typique de Recherche Opérationnelle. On appelle ce problème le *problème du voyageur de commerce* (étudié plus en détail au Chapitre 11).

Remplir un conteneur avec des objets de tailles et de valeurs variables. Si le conteneur a une capacité finie, on va chercher à maximiser la valeur placée dans le conteneur. On appelle ce problème le *problème du sac-à-dos* (étudié plus en détail au Chapitre 8).

Ordonnancer les tâches sur un chantier. Pour chaque tâche T, on connaît sa durée. De plus, on connaît les autres tâches dont T dépend directement et combien de temps avant ou après le début de chacune d'elles T doit démarrer. On désire minimiser la durée totale du chantier. On dit que ce problème est un *problème d'ordonnancement* (étudié plus en détail au Chapitre 10).

Chacun de ces problèmes peut bien sûr être compliqué à l'envie. Dans ce cours, on restera relativement simple – quelques contraintes de plus suffisent en effet à faire de ces problèmes de véritables sujets de thèse (par exemple pour le remplissage de conteneur un sujet de thèse peut consister en : plusieurs types de conteneurs, plusieurs produits à stocker, des incompatibilités).

Histoire

La Recherche Opérationnelle est née pendant la Seconde Guerre mondiale des efforts conjugués d'éminents mathématiciens (dont von Neumann, Dantzig, Blackett) à qui il avait été demandé de fournir des techniques d'optimisations des ressources militaires. Le premier succès de cette approche a été obtenue en 1940 par le Prix Nobel de physique Patrick Blackett qui résolut un problème d'implantation optimale de radars de surveillance. Le qualificatif « opérationnelle » vient du fait que les premières applications de cette discipline avait trait aux opérations militaires. La dénomination est restée par la suite, même si le domaine militaire n'est plus le principal champ d'application de cette discipline, le mot « opérationnelle » prenant alors plutôt

le sens d'« effectif ». Ce sont donc ces mathématiciens qui ont créé une nouvelle méthodologie caractérisée par les mots-clés *modélisation* et *optimisation*.

A partir des années 50, la Recherche Opérationnelle fait son entrée dans les entreprises. En France, des entreprises comme EDF, Air France, la SNCF créent à cette époque des services de recherche opérationnelle (qui existent toujours). La discipline commence à être enseignée dans les universités et les grandes écoles. Puis, au milieu des années 70, sans doute à cause d'un excès d'enthousiasme au départ et à l'inadéquation des moyens informatiques à l'application des méthodes de la RO, la discipline s'essouffle. A partir du mileu des années 90, on assiste à un retour en force la RO, les outils informatiques étant maintenant à la hauteur des méthodes proposées par la Recherche Opérationnelle. On assiste depuis à une explosion du nombre de logiciels commerciaux et l'apparition de nombreuses boîtes de conseil. Pour la France, notons Ilog (65 millions d'euros de CA), Eurodécision (2,8 millions d'euros de CA), Artelys (1,6 millions d'euros de CA) à l'étranger Dash-Optimization (racheté début 2008 pour 32 millions de dollars par Fair Isaac), IBM Optimization et beaucoup d'autres (le site de INFORMS Institute of Operation Research and Management Science en liste près de 240).

Les racines. — Si l'on cherche à trouver des précurseurs à la Recherche Opérationnelle, on peut penser à Alcuin ou à Euler qui se sont tous deux intéressés à des problèmes du type RO, bien qu'aucune application n'ait motivé leur travail.

Alcuin est le moine irlandais chargé par Charlemagne de construire l'école palatine et qui inventa le problème du loup, de la chèvre et du chou devant traverser une rivière dans une barque où au plus un élément peut prendre place.

> Un homme devait transporter de l'autre côté d'un fleuve un loup, une chèvre et un panier de choux. Or le seul bateau qu'il put trouver ne permettait de transporter que deux d'entre eux. Il lui a donc fallu trouver le moyen de tout transporter de l'autre côté sans aucun dommage. Dise qui peut comment il a réussi à traverser en conservant intacts le loup, la chèvre et les choux[(1)].

Euler est le mathématicien allemand à qui les notables de Königsberg demandèrent s'il était possible de parcourir les ponts de la ville en passant sur chacun des 7 ponts exactement une fois (voir Figure 15). Ce genre de problème se rencontre maintenant très souvent dans les problèmes de tournées du type facteur ou ramassage d'ordure, dans lesquels il faut parcourir les rues d'une ville de façon optimale. Euler trouva la solution en 1736 – un tel parcours est impossible – en procédant à une modélisation subtile par des mots. La solution actuelle, beaucoup plus simple, utilise une modélisation par un *graphe* (voir Chaître 2). Le graphe est un concept mathématique simple – tellement simple d'ailleurs que beaucoup de mathématiciens en nient l'intérêt – qui sera présenté dans ce premier chapitre, avec la solution du problème des ponts, et amplement utilisé dans la suite du cours. On voit sur cet exemple qu'une bonne modélisation peut simplifier de manière drastique la résolution d'un problème.

Le premier problème de Recherche Opérationnelle à visée pratique a été étudié par Monge en 1781 sous le nom du problème des déblais et remblais. Considérons n tas de sable, devant servir à combler m trous. Notons a_i la masse du $i$ème tas de sable et b_j la masse de sable nécessaire pour combler le $j$ème trou. Quel plan de transport minimise la distance totale parcourue par le sable ?

[(1)] Homo quidam debebat ultra fluvium transferre lupum, capram, et fasciculum cauli. Et non potuit aliam navem invenire nisi quae duos tantum ex ipsis ferre valebat. Praeceptum itaque ei fuerat ut omnia haec ultra illaesa omnino transferret. Dicat, qui potest, quomodo eis illaesis transire potuit.

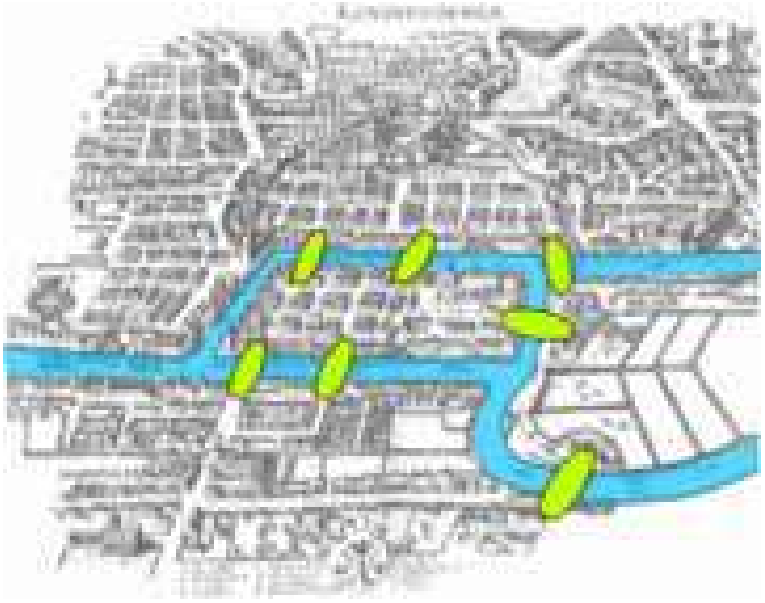


FIG. 1. Königsberg et ses 7 ponts

La solution que proposa Monge est intéressante et procède par une modélisation dans un espace continu dans lequel on cherche une géodésique – malheureusement, elle n'est pas correcte. La solution correcte pour trouver l'optimum est connue depuis les années 40 et utilise la programmation linéaire (que nous verrons au Chapitre 4), ou mieux, la théorie des flots (que nous verrons au Chapitre 5).

Modélisation et optimisation

> [Wikipedia] Un modèle mathématique est une traduction de la réalité pour pouvoir lui appliquer les outils, les techniques et les théories mathématiques, puis généralement, en sens inverse, la traduction des résultats mathématiques obtenus en prédictions ou opérations dans le monde réel.

Les problèmes d'organisation rencontrés dans une entreprise ne sont pas mathématiques dans leur nature. Mais les mathématiques peuvent permettre de résoudre ces problèmes. Pour cela, il faut traduire le problème dans un cadre mathématique, cadre dans lequel les techniques de la recherche opérationnelle pourront s'appliquer. Cette traduction est le modèle du problème.

Cette phase essentielle s'appelle la *modélisation*. La résolution d'un problème dépend crucialement du modèle choisi. En effet, pour un même problème, différentes modélisations sont possibles et il n'est pas rare que le problème semble insoluble dans une modélisation et trivial dans une autre.

D'autre part, tous les éléments d'un problème ne doivent pas être modélisés. Par exemple, lorsqu'on souhaite planifier une tournée, la couleur du véhicule n'a pas d'intérêt. Le statut du conducteur, la nature du véhicule ou du produit transporté peuvent, eux, en avoir, et seule une compréhension de l'objectif de l'optimisation de la tournée peut permettre de trancher. Souvent, la phase de modélisation est accompagnée ou précédée de nombreuses discussions avec le commanditaire (lequel n'a d'ailleurs pas toujours une idée claire de ce qu'il cherche à obtenir – ces discussions lui permettent alors également de préciser ses objectifs).

Une des vrais difficultés de départ est de savoir quels éléments doivent être modélisés et quels sont ceux qui n'ont pas besoin de l'être. Il faut parvenir à trouver le juste équilibre entre un modèle simple, donc plus facilement soluble, et un modèle compliqué, plus réaliste, mais plus difficile à résoudre. Cela dit, pour commencer, un modèle simple est toujours préférable et permet souvent de capturer l'essence du problème[(2)], de se construire une intuition et de proposer des solutions faciles à implémenter.

Ce qui est demandé au chercheur opérationnel, c'est de proposer une meilleure utilisation des ressources, voire une utilisation optimale. Les bonnes questions à se poser, face à un problème du type Recherche Opérationnelle, sont les suivantes :

- Quelles sont les *variables de décision* ? C'est-à-dire quels sont les éléments de mon modèle que j'ai le droit de faire varier pour proposer d'autres solutions ?
- Quelles sont les *contraintes* ? Une fois identifiées les variables de décision, quelles sont les valeurs autorisées pour ces variables ?
- Quel est l'*objectif* ou le *critère* ? Quelle est la quantité que l'on veut maximiser ou minimiser ?

Rappelons immédiatement que, en toute rigueur, on n'optimise qu'une seule quantité à la fois. On ne peut pas demander d'optimiser à la fois la longueur d'un trajet et son temps de parcours : le trajet le plus court peut très bien passer par un chemin vicinal et le trajet le plus rapide être très long mais sur autoroute. L'optimum par rapport à un critère n'a pas de raison de coïncider avec l'optimum par rapport à l'autre critère. Il existe bien ce qu'on appelle l'*optimisation multi-objectif* ou *multi-critère*, mais ce n'est jamais directement une optimisation. C'est une méthode qui consiste à hiérarchiser les objectifs, ou leur donner une certaine pondération, ce qui revient in fine à un vrai problème d'optimisation ; ou alors de proposer toute une famille de solutions dite Pareto-optimale.

Une fois le modèle écrit, le chercheur opérationnel va proposer un algorithme de résolution qui tiendra compte de l'objectif qui lui a été fixé. Comme nous le verrons à de nombreuses reprises dans ce cours, pour un même modèle, un grand nombre d'algorithmes peut être proposé. Ces algorithmes se différencient par la qualité de la solution qu'ils fournissent, le temps d'exécution, la simplicité d'implémentation. Dans certains cas, il peut être cruciale de pouvoir fournir une solution en 1 ms, avec une certaine tolérance sur la qualité de la solution. Dans d'autres cas, 1 semaine de calcul peut être acceptable mais en revanche on souhaite trouver l'optimum. En général, on se situe entre ces deux extrêmes.

La Recherche Opérationnelle dispose d'outils théoriques qui permettent a priori d'apprécier ces points (rapidité de l'algorithme, qualité de la solution,...) sans avoir à expérimenter. On parle de *validation théorique*. Ensuite, il faut réaliser un prototype de l'algorithme (on parle souvent de "code académique") qui permet de démontrer sa réalisabilité pratique – on parle de *validation pratique*. Enfin, si ces étapes sont validées, on passe au *déploiement de la solution*, qui consiste à produire un code robuste, programmer une interface, discuter les formats des fichiers d'input, de discuter la question de la maintenance du code, etc. mais là on s'éloigne du cœur du métier du chercheur opérationnel.

En résumé, la méthodologie de la recherche opérationnelle suit en général le schéma suivant.

1. Objectifs, contraintes, variables de décision.
2. Modélisation.

[(2)] En physique ou en économie, beaucoup de modèles simples, comme le modèle d'Ising ou le modèle de concurrence parfaite, sont très fructueux pour expliquer le réel.

3. Proposition d'un algorithme, validité théorique de l'algorithme (temps d'exécution pour trouver la solution, qualité de la solution fournie).
4. Implémentation, validation pratique de la solution.
5. Déploiement de la solution.

Objectif de ce cours

La Recherche Opérationnelle occupe une place grandissante dans l'industrie, la logistique et les transports. Pour un ingénieur souhaitant faire un travail technique dans ces disciplines, elle est quasi-incontournable. L'objectif de ce cours est de donner les bases de recherche opérationnelle : la méthodologie, les problèmes et les modèles typiques, les principales techniques de résolution. Un étudiant maîtrisant les exercices de ce cours est capable de proposer une modélisation d'une grande part des problèmes de recherche opérationnelle rencontrés dans l'industrie, de proposer des approches de résolution et d'en discuter les qualités respectives. Le cours se focalisera principalement sur les étapes 2. et 3. ci-dessus.

Première PARTIE I

FONDEMENTS

CHAPITRE 2

BASES

2.1. Graphes

Une brique de base dans la modélisation est le *graphe*. Les graphes apparaissent naturellement lorsqu'on est confronté à un réseau (réseau de transport, réseau informatique, réseau d'eau ou de gaz, etc.). Comment parcourir le réseau de manière optimale ? C'est la question du voyageur de commerce par exemple. Comment concevoir un réseau informatique robuste, tout en minimisant le nombre de connexions ? C'est le thème de la conception de réseau (Network Design), abordé au Chapitre 13.

Mais les graphes apparaissent également de manière plus subtile dans certaines modélisations de problèmes où la structure du graphe n'est pas physique. Un exemple classique est le problème de l'affectation optimale d'employés à des tâches qui sera étudié aux Chapitres 5 et 6. Les sommets représentent des tâches et des employés, et les arêtes les appariements possibles entre tâches et employés. Le problème des déblais et remblais étudié par Monge peut se modéliser par un graphe (biparti).

2.1.1. Graphe non-orienté. — Un *graphe* $G = (V, E)$ est la donnée d'un ensemble fini V de *sommets* et d'une famille finie E d'*arêtes* définies comme paires de sommets. On écrira indifféremment $e = uv$ ou $e = vu$, où e est une arête et où u et v sont des sommets. On parle de famille d'arêtes pour souligner le fait que les répétitions sont autorisées. En cas de répétition, on parle d'*arêtes parallèles*. La paire définissant une arête peut également être la répétition d'un sommet, auquel cas on parle de *boucle*. Un graphe est souvent représenté dans le plan par des points – ce sont les sommets – reliés par des lignes – ce sont les arêtes. Voir la Figure 1. Deux sommets appartenant à la même arête sont dits *voisins*.

Un graphe est dit *complet* si toute paire de sommets est reliée par une arête. Voir exemple Figure 2. Un graphe est dit biparti si l'ensemble des sommets peut être partitionné en deux parties n'induisant chacune aucune arête. Voir exemple Figure 3.

Le *degré* d'un sommet v est le nombre d'arêtes incidentes à v, c'est-à-dire admettant v comme extrémité. Précisons qu'une boucle doit être comptée deux fois. Un sommet de degré égal à 0 est dit *isolé*.

Un *sous-graphe (induit)* H d'un graphe G a pour ensemble de sommets un sous-ensemble $V' \subseteq V$. Les arêtes de H sont alors **toutes** celles de G ayant leurs deux extrémités dans V'[(1)].

[(1)] Dans la terminologie anglaise, un tel sous-graphe est appelé *induced subgraph*. Ces arêtes sont notées $E[V']$. On parle aussi en français de *graphe partiel* de G, qui a le même ensemble de sommets que G, mais n'a qu'une partie de ses arêtes, et de *sous-graphe partiel*, qui a comme ensemble de sommets une partie des sommets de G et comme ensemble d'arêtes une partie de celles de E – en anglais, on parle alors de *subgraph*.

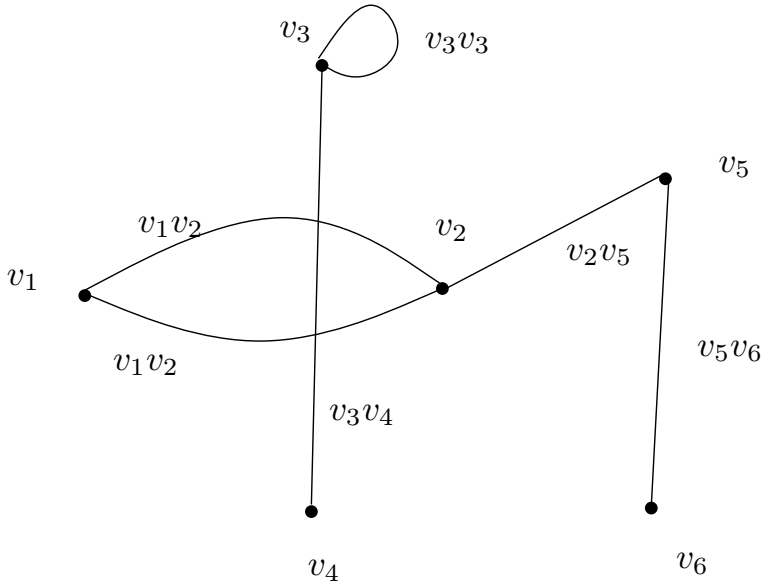


FIG. 1. Exemple de graphe. Ce graphe est non connexe, possède 6 sommets, 6 arêtes, dont deux arêtes parallèles et une boucle.

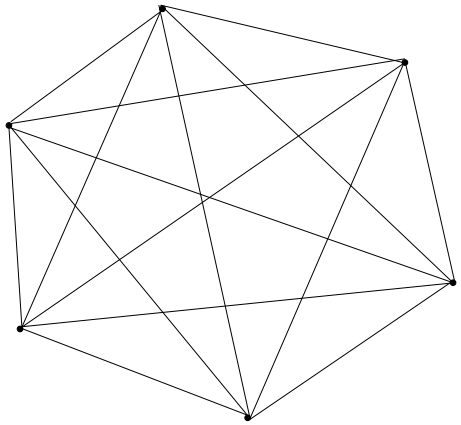


FIG. 2. Exemple de graphe complet.

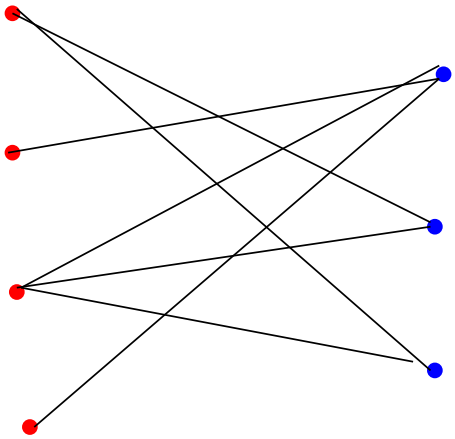


FIG. 3. Exemple de graphe biparti.

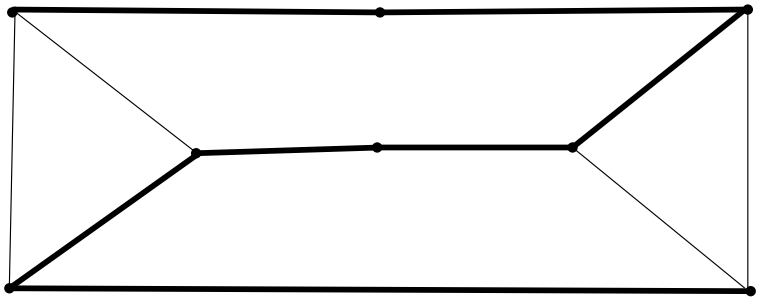


FIG. 4. Graphe possédant une chaîne hamiltonienne. La chaîne hamiltonienne est mise en gras.

2.1.1.1. Chaîne, cycle, connexité. — Une *chaîne* est une suite de la forme

$$v_0, e_1, v_1, \ldots, e_k, v_k$$

où k est une entier ≥ 0, $v_i \in V$ pour $i = 0, \ldots, k$ et $e_j \in E$ pour $j = 1, \ldots, k$ avec $e_j = v_{j-1}v_j$. L'entier k est la *longueur* de la chaîne. En d'autres termes, une chaîne est un trajet possible dans la représentation d'un graphe lorsqu'on suit les arêtes sans rebrousser chemin sur une arête. Si les e_i sont distincts deux à deux, la chaîne est dite *simple*. Si de plus la chaîne ne passe jamais plus d'une fois sur un sommet, elle est dite *élémentaire*. Une chaîne simple passant par toutes les arêtes d'un graphe est dite *eulérienne*. Une chaîne élémentaire passant par tous les sommets du graphe est dite *hamiltonienne* (voir un exemple Figure 4).

Un *cycle* est une chaîne de longueur ≥ 1, simple et fermée. C'est donc une suite de la forme

$$v_0, e_1, v_1, \ldots, e_k, v_0$$

avec $e_j = v_{j-1}v_j$ pour $j < k$, et $e_k = v_0v_{k-1}$. Un cycle est *élémentaire* si les v_i pour $i = 0, \ldots, k-1$ sont distincts deux à deux. Un cycle passant par toutes les arêtes d'un graphe est dit *eulérien*. Un cycle élémentaire passant par tous les sommets du graphe est dit *hamiltonien*.

Un graphe est dit *connexe* si entre toute paire de sommet il existe une chaîne.

2.1.1.2. Couplage. — Un *couplage* dans un graphe $G = (V, E)$ est un sous-ensemble d'arêtes $M \subseteq E$ telles que deux arêtes quelconques de M soient disjointes. Une *couverture par les sommets* est un sous-ensemble de sommets $C \subseteq V$ tel que toute arête de E touche au moins un sommet de C. On a la propriété très utile suivante, car elle permet par exemple d'évaluer la qualité d'un couplage.

Proposition 2.1.1. — *Soit $G = (V, E)$ un graphe. Soit M un couplage et C une couverture par les sommets. Alors*

$$|M| \leq |C|.$$

La preuve est laissée en exercice.

2.1.1.3. Coloration. — Une notion fructueuse en théorie des graphes et très utile en Recherche Opérationnelle est la notion de *coloration*. En dehors de ce chapitre, cette notion ne sera pas utilisée. A la fin du chapitre, la coloration sera utilisée pour illustrer la puissance de la modélisation sur des problèmes de natures très différentes – l'affectation de fréquence en téléphonie mobile et l'organisation de formations pour des employés.

Une *coloration* d'un graphe est une application $c : V \to \mathbb{N}$. Les entiers $\mathbb{N}$ sont alors appelés *couleurs*. Une *coloration propre* est telle que pour toute paire de voisins u, v on ait $c(u) \neq c(v)$: deux sommets voisins ont des couleurs différentes. Une question que l'on peut se poser, étant

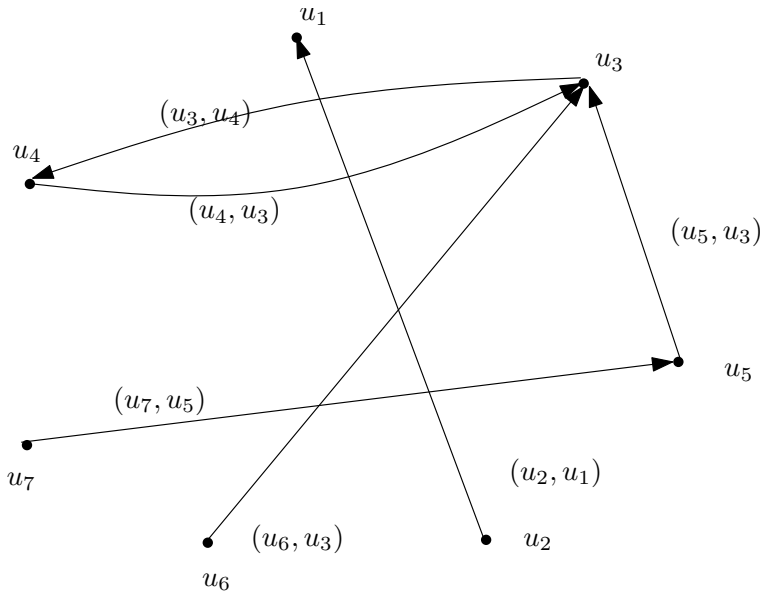


FIG. 5. Exemple de graphe orienté. Ce graphe possède 7 sommets et 5 arcs.

donné un graphe, est le nombre minimum de couleurs possible pour une coloration propre. Ce nombre, noté $\chi(G)$, s'appelle le *nombre chromatique* de G.

On a l'inégalité suivante : cardinalité du plus grand sous-graphe complet $\leq \chi(G)$ (démonstration laissée en exercice – mais c'est presque trivial !).

2.1.2. Graphe orienté. — Un *graphe orienté* $D = (V, A)$ est la donnée d'un ensemble fini V de *sommets* et d'une famille finie A d'*arcs* définies comme couples[(2)] de sommets. On écrira $a = (u, v)$, où a est un arc et où u et v sont des sommets. Comme pour les graphes non orientés, les répétitions sont autorisées et, en cas de répétition, on parle d'*arcs parallèles*. Le couple définissant un arc peut également être la répétition d'un sommet, auquel cas on parle de *boucle*. Lorsqu'on représente un graphe orienté, les lignes deviennent des flèches (voir Figure 5). Pour un arc (u, v), le sommet u est appelé *antécédent* de v et v est le *successeur* de u.

Le *degré sortant* (resp. *degré entrant*) d'un sommet u et noté $\deg_{out}(u)$ (resp. $\deg_{in}(u)$) est le nombre de successeurs (resp. antécédents) de v.

Un *chemin* est une suite de la forme

$$v_0, a_1, v_1, \ldots, a_k, v_k$$

où k est une entier ≥ 0, $v_i \in V$ pour $i = 0, \ldots, k$ et $a_j \in A$ pour $j = 1, \ldots, k$ avec $a_j = (v_{j-1}, v_j)$. L'entier k est la *longueur* du chemin. En d'autres termes, un chemin est un trajet possible dans la représentation d'un graphe lorsqu'on suit les arcs dans le sens des flèches. Si les a_i sont distincts deux à deux, le chemin est dite *simple*. Si de plus le chemin ne passe jamais plus d'une fois sur un sommet, elle est dite *élémentaire*. Un chemin simple passant par tous les arcs d'un graphe orienté est dit *eulérien*. Un chemin élémentaire passant par tous les sommets du graphe est dit *hamiltonien*.

Un *circuit* est un chemin de longueur ≥ 1, simple et fermée. C'est donc une suite de la forme

$$v_0, a_1, v_1, \ldots, a_k, v_0$$

[(2)]Rappelons que les deux éléments d'un couple sont ordonnés, ceux d'une paire ne le sont pas.

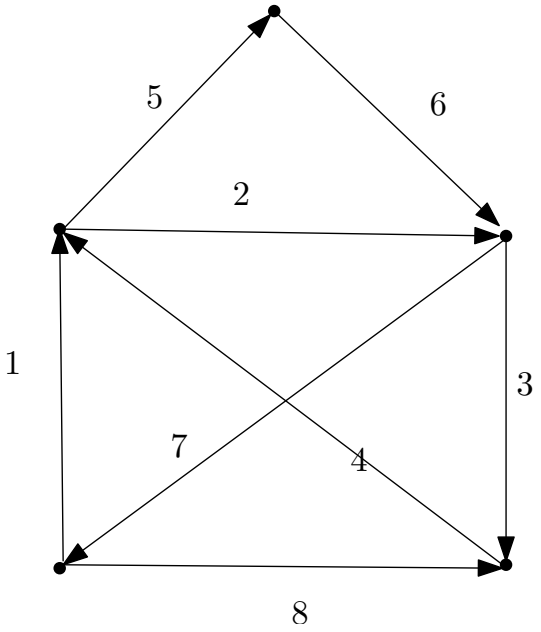


FIG. 6. Graphe orienté possédant un circuit eulérien. L'ordre des arcs dans un tel circuit est indiqué.

avec $a_j = (v_{j-1}, v_j)$ pour $j < k$, et $a_k = (v_0, v_{k-1})$. Un circuit est *élémentaire* si les v_i pour $i = 0, \ldots, k-1$ sont distincts deux à deux. Un circuit passant par tous les arcs d'un graphe orienté est dit *eulérien*. Un circuit élémentaire passant par tous les sommets du graphe est dit *hamiltonien*. Un exemple de circuit eulérien est donné Figure 6.

Un graphe orienté est dit *faiblement connexe* si le graphe non orienté obtenu en "oubliant" les orientations des arcs est connexe. Il est dit *fortement connexe* si pour tout couple (u, v) de sommet, il existe un chemin allant de u à v.

Une dernière remarque : Si rien n'est précisé par ailleurs, n représentera $|V|$ le nombre de sommets et m représentera $|E|$ (ou $|A|$) le nombre d'arêtes (ou le nombre d'arcs).

2.2. Retour sur les ponts et sur le voyageur

Le problème des ponts de Königsberg auquel il a été fait allusion ci-dessus se modélise de la manière suivante : chacune des îles et chacune des berges sont représentées par des sommets. Il y a donc 4 sommets. Chaque pont est représenté par une arête – voir Figure 7. On cherche donc à savoir s'il y a une chaîne eulérienne dans ce graphe.

La condition nécessaire et suffisante d'existence d'une chaîne eulérienne dans un graphe est simple. La preuve est laissée en exercice.

Théorème 2.2.1. — *Un graphe sans sommet isolé admet une chaîne eulérienne si et seulement si il est connexe et possède au plus deux sommets de degré impair.*

Un graphe sans sommet isolé admet un cycle eulérien si et seulement si il est connexe et n'a pas de sommet de degré impair.

. — □

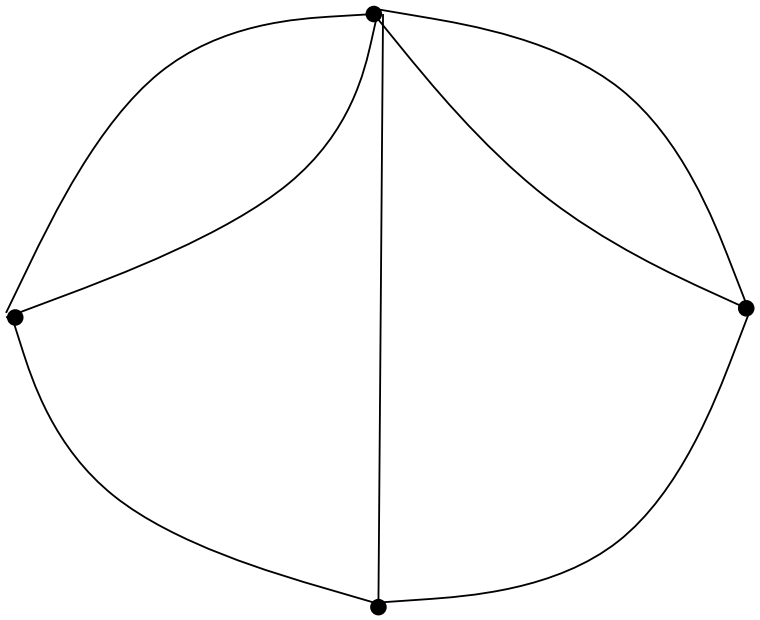


FIG. 7. Modélisation du problème des ponts de Königsberg. Ce graphe possède-t-il une chaîne eulérienne ?

On peut également tenter une **modélisation du problème du voyageur de commerce** dont on a parlé au début de ce chapitre. Le point à livrer est représenté par un sommet du graphe, la route la plus courte entre deux points est représentée par une arête. On appelle ce graphe $K_n = (V, E)$, où n est le nombre de sommets. Ce graphe – simple – s'appelle le *graphe complet* puisque toute paire de sommets correspond à une arête. On ajoute encore une fonction $d : E \to \mathbb{R}_+$ qui indique la longueur des routes représentées par les arêtes. On cherche donc le **cycle hamiltonien de K_n le plus court pour la fonction de poids d**. Le Chapitre 11 de ce cours traitera des méthodes pour résoudre le problème.

Dans le cas orienté, la condition nécessaire et suffisante d'existence de chemin ou circuit eulérien est semblable à celle du Théorème 2.2.1.

Théorème 2.2.2. — *Un graphe sans sommet isolé admet un chemin eulérien si et seulement si il est faiblement connexe et s'il existe deux sommet v et w tels que* $\deg_{out}(v) = \deg_{in}(v) + 1$, $\deg_{out}(w) + 1 = \deg_{in}(w)$ *et* $\deg_{out}(u) = \deg_{in}(u)$ *pour tout sommet* $u \neq v, w$.

Un graphe admet un cycle eulérien si et seulement si il est faiblement connexe et $\deg_{out}(u) = \deg_{in}(u)$ *pour tout sommet* u.

Notons qu'un graphe admettant un cycle eulérien est automatiquement fortement connexe.

2.3. Programmation mathématique

2.3.1. Définition. — Le terme *programmation mathématique* est un terme plutôt vague pour désigner le problème mathématique qui consiste à chercher un élément dans un ensemble X qui minimise ou maximise un *critère*, ou *coût*, ou *fonction de coût*. Puisqu'en général la Recherche Opérationnelle a pour objectif l'optimisation, la phase de modélisation se termine souvent par l'écriture formelle d'un programme mathématique.

Un programme mathématique s'écrit sous la forme suivante (on supposera que l'on cherche à minimiser une certaine quantité).

$$\begin{array}{ll} \text{Min} & f(x) \\ \text{s.c.} & x \in X. \end{array}$$

f est le critère. « s.c. » signifie « sous contraintes » X est l'ensemble des solutions possibles ou *réalisables*, et « $x \in X$ » est la ou les contraintes du programme. On cherche parmi ces solutions réalisables une *solution optimale* x^*, i.e. ici une solution réalisable qui minimise le critère. $\inf\{f(x) : x \in X\}$ est appelé *valeur du programme*. Si $X = \emptyset$, cette valeur est définie comme étant égale à $+\infty$.

En plus de fournir un cadre compact pour formuler un problème, cette notation à l'avantage de rappeler les questions essentielles à se poser lorsqu'on résout un problème. Veut-on minimiser ou maximiser ? Que veut-on optimiser ? Quelles sont les contraintes de notre système ? Quelles sont les variables sur lesquelles on peut jouer ?

Une remarque fondamentale et très utile est que tout problème de minimisation peut se réécrire comme un problème de maximisation : il suffit de prendre comme critère $-f(x)$. Et réciproquement.

Dans ce cours, nous verrons la *programmation dynamique* (Chapitre 3), la *programmation linéaire* et la *programmation linéaire en nombres entiers* (Chapitre 4). Ces trois types de programmation sont très fréquents en Recherche Opérationnelle.

2.3.2. Approximation et borne. — Chercher un optimum peut être difficile. On peut alors chercher une *α-approximation*, c'est-à-dire une solution réalisable $x \in X$ telle que

$$f(x) \leq \alpha f(x^*).$$

Si l'on est capable de trouver un *minorant de f*, c'est-à-dire une fonction g telle que pour tout $x \in X$, on ait $g(x) \leq f(x)$, une façon de prouver que l'on a une α-approximation consiste à montrer que l'on a $f(x) \leq \alpha g(x^*)$. On voit dans ce cadre l'**importance de la qualité du minorant de f**. Dans les chapitres à venir, nous verrons souvent l'importance que peuvent avoir ces minorants. De façon générale, avoir un minorant g permet de majorer la qualité de la solution x proposée par rapport à la solution optimale x^* : on est à moins de

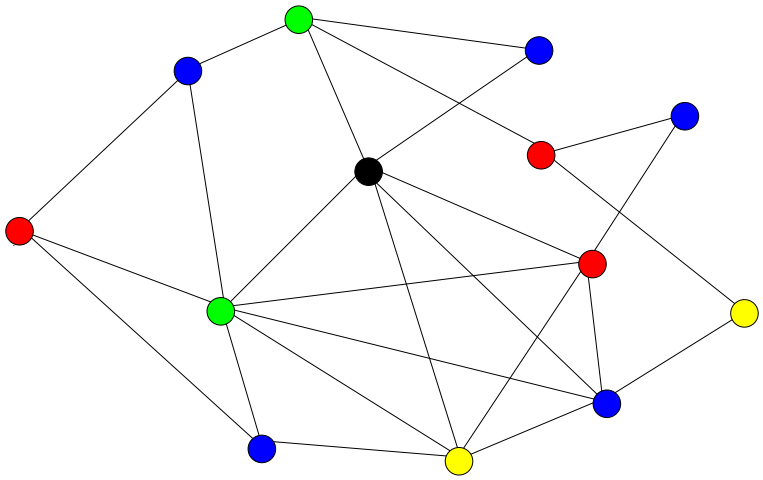
$$\frac{f(x) - g(x^*)}{g(x^*)}\%$$

de l'optimum. Il est toujours utile de vérifier si l'on n'a pas un bon minorant, facile à calculer. On peut alors éviter de chercher à améliorer une solution en faisant tourner longuement un programme informatique, alors qu'une borne inférieure nous fournit la preuve que l'on est à moins de 0,1% de l'optimum.

En particulier, un bon minorant permet de montrer l'optimalité d'une solution. Voir Figures 8 et 9.

2.3.3. Optimisation continue vs optimisation discrète. — Si X est une partie d'un espace vectoriel et si f satisfait certaines conditions de régularité, la résolution de ce programme relève de l'*optimisation continue*. Un outil fréquent est alors la dérivation. Notre seule rencontre avec l'optimisation continue dans ce cours se fera par l'intermédiaire de la programmation linéaire, qui, elle, ne peut pas se résoudre par une simple dérivation.

Dans les problèmes industriels, l'ensemble X est souvent de nature *discrète*, i.e. qu'il peut être mis en bijection avec une partie de $\mathbb{N}$. Par exemple, le problème du voyageur de commerce



Graphe coloré avec 5 couleurs: rouge, bleu, vert, jaune et noir.

FIG. 8. Peut-on colorer proprement ce graphe avec moins de 5 couleurs ?

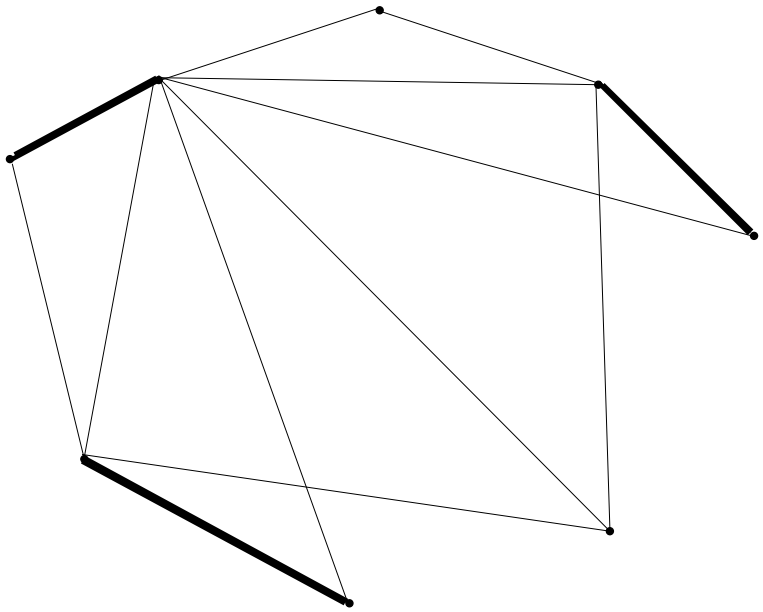


FIG. 9. Peut-on trouver un couplage de plus de 3 arêtes ?

qui consiste à trouver le cycle le plus court passant par des villes fixées à l'avance est de nature discrète. X est ici l'ensemble des cycles hamiltoniens. Tout cycle de ce type est une solution réalisable. Le critère est la distance parcourue. Il y a un nombre fini de cycles. On est bien dans le cadre de l'optimisation discrète (et l'on voit bien qu'ici la plupart des notions de l'optimisation continue sont sans intérêt : que dériver ici ?).

2.3.4. Programmation linéaire, programmation convexe. — Considérons le programme

$$\begin{array}{ll} \text{Min} & f(x) \\ \text{s.c.} & g(x) = 0, \\ & h(x) \leq 0 \\ & x \in X, \end{array} \quad \text{(P)}$$

avec $X \neq \emptyset$, $f : X \to \mathbb{R}$, $g : X \to \mathbb{R}^p$ et $h : X \to \mathbb{R}^q$. On dit que (P) est un *programme linéaire* si f est une fonction linéaire, et si g et h sont toutes deux affines (cf. Chapitre 4). On dit que (P) est un *programme convexe* si f, et h sont toutes deux convexes, et si g est affine. La programmation linéaire est bien entendu un cas particulier de programmation convexe. Les techniques pour aborder cette dernières ne seront que très sommairement évoquées dans ce cours.

2.3.5. Lagrangien et dualité faible. — On définit le *lagrangien* associé au problème (P)

$$\mathcal{L}(x, \boldsymbol{\lambda}, \boldsymbol{\mu}) := f(x) + \boldsymbol{\lambda}^T g(x) + \boldsymbol{\mu}^T h(x),$$

où $\boldsymbol{\lambda} \in \mathbb{R}^p$ et $\boldsymbol{\mu} \in \mathbb{R}^q_+$. On vérifie facilement que

$$\sup_{\boldsymbol{\lambda} \in \mathbb{R}^p, \boldsymbol{\mu} \in \mathbb{R}^q_+} \mathcal{L}(x, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \begin{cases} f(x) & \text{si } g(x) = 0 \text{ et } h(x) \leq 0, \\ +\infty & \text{sinon.} \end{cases}$$

On peut donc réécrire l'équation (P) de la manière suivante

$$\min_{x \in X} \sup_{\boldsymbol{\lambda} \in \mathbb{R}^p, \boldsymbol{\mu} \in \mathbb{R}^q_+} \mathcal{L}(x, \boldsymbol{\lambda}, \boldsymbol{\mu}).$$

On a toujours

$$\min_{x \in X} \sup_{\boldsymbol{\lambda} \in \mathbb{R}^p, \boldsymbol{\mu} \in \mathbb{R}^q_+} \mathcal{L}(x, \boldsymbol{\lambda}, \boldsymbol{\mu}) \geq \sup_{\boldsymbol{\lambda} \in \mathbb{R}^p, \boldsymbol{\mu} \in \mathbb{R}^q_+} \min_{x \in X} \mathcal{L}(x, \boldsymbol{\lambda}, \boldsymbol{\mu}). \quad (1)$$

En définissant

$$d(\boldsymbol{\lambda}, \boldsymbol{\mu}) := \min_{x \in X} \mathcal{L}(x, \boldsymbol{\lambda}, \boldsymbol{\mu}),$$

on peut regarder le programme mathématique appelé *dual* de (P)

$$\begin{array}{ll} \text{Max} & d(\boldsymbol{\lambda}, \boldsymbol{\mu}) \\ \text{s.c.} & \boldsymbol{\lambda} \in \mathbb{R}^p \\ & \boldsymbol{\mu} \in \mathbb{R}^q_+. \end{array} \quad \text{(D)}$$

(P) est alors appelé programme *primal*.

Notant v_P (resp. v_D) la valeur de (P) (resp. (D)), l'inégalité (1) fournit l'inégalité suivante, appelée *inégalité de dualité faible*

$$v_P \geq v_D.$$

Cette inégalité est extrêmement utile et constitue un moyen "automatique" de générer des bornes inférieures à un problème. Ces notions seront en particulier approfondies dans le Chapitre 4 sur le programmation linéaire et dans le Chapitre 7. En programmation linéaire, nous verrons qu'on dispose d'une relation plus forte, appelée *dualité forte*.

Remarque. — d est une fonction concave car infimum de fonctions affines.

2.4. Problème

Une autre notion commode dans la modélisation est la notion de *problème*. Un problème se décompose en deux parties : une partie **Données** et une partie **Tâche** ou **Question**. Une telle formalisation permet d'écrire clairement quelles sont les éléments dont on dispose au départ et quelle est précisément la tâche que l'on veut résoudre. L'objectif à atteindre devient donc clair et dans un tel contexte, il est plus facile de discuter des performances de telle ou telle méthode.

Par exemple, le problème de l'existence d'une chaîne eulérienne peut s'écrire :

Problème de la chaîne eulérienne

Données : Un graphe $G = (V, E)$.

Question : G a-t-il une chaîne eulérienne ?

Celui du voyageur de commerce :

Problème du voyageur de commerce

Données : Un graphe complet $K_n = (V, E)$ et une fonction de distance $d : E \to \mathbb{R}_+$.

Tâche : Trouver le cycle hamiltonien $C = v_0, e_1, \ldots, e_n, v_0$ minimisant $\sum_{j=1}^{n} d(e_j)$.

Le premier problème est un type particulier de problème, qui joue un rôle important en informatique théorique, et s'appelle un *problème de décision*, puisque la tâche à réaliser est de répondre à une question fermée, i.e. dont la réponse est soit « oui », soit « non ». Le second problème n'en est pas un (pour celui-là, on pourra parler de *problème d'optimisation*).

2.5. Algorithme et complexité

2.5.1. Algoritme, algorithme exact, algorithme d'approximation. — Une fois que l'on a modélisé notre problème concret en un problème formalisé ou en un programme mathématique, on doit se demander comment le résoudre. Dans ce cours, nous verrons différents problèmes, différents programmes, et quelles sont les méthodes pour les résoudre. De nouveaux problèmes sont proposés tout le temps par les chercheurs travaillant en Recherche Opérationnelle ; dans ce cours, nous nous limiterons bien entendu aux plus classiques.

On se fixe un ensemble d'opérations que l'on considère "facile à faire". Par exemple, comparaison d'entiers, lire une adresse mémoire, etc. Une suite d'opérations élémentaires permettant de résoudre un programme d'optimisation ou un problème s'appelle un *algorithme*. La résolution d'un problème de recherche opérationnelle passe toujours par l'application d'un algorithme, qui est ensuite implémenté. Si le problème que l'on tente de résoudre est un programme d'optimisation, on parlera d'*algorithme exact* si l'algorithme est sûr de ce terminer avec l'optimum du programme, et d'*algorithme d'approximation* s'il existe un α tel que pour tout jeu de données, l'algorithme est sûr de se terminer avec une α-approximation. On dit alors que l'agorithme est *α-approximatif*.

La question qui se pose également est celle de l'efficacité de l'algorithme, i.e. du temps qu'il va mettre pour résoudre le problème (une fois admis que l'algorithme est correct et résout bien le problème).

2.5.2. Question de l'efficacité. — Une méthode peut être de tester l'algorithme sur de grandes quantités de données. Mais ces tests là peuvent être très, très longs, et de toute façon, à chaque fois que l'on va penser à un algorithme possible, on ne va pas systématiquement procéder à son implémentation et à des tests. La *théorie de la complexité*, un des fondement théorique de la recherche opérationnelle, a pour but de mesurer a priori l'efficacité d'un algorithme. La petite histoire qui va suivre va montrer l'intérêt de cette théorie, et est librement adapté du livre [**10**].

2.5.3. De l'intérêt de la théorie pour sauver son boulot. — Supposez qu'un jour votre patron vous appelle dans son bureau et vous annonce que l'entreprise est sur le point d'entrer sur le marché compétitif du schmilblick. Il faut donc trouver une bonne méthode qui permette de dire si une collection de spécifications peuvent être satisfaites ou non, et si oui, qui donne la façon de construire ce schmilblick. On peut imaginer qu'un schmilblick est décrit par des variables booléennes C_i qui indiquent si le composant i est présent ou non. De plus, on a des règles du type si C_1 et C_5 sont vraies et C_3 est faux, alors C_7 doit être faux. Etc.

Comme vous travaillez au département de Recherche Opérationnelle, c'est à vous de trouver un algorithme efficace qui fasse cela.

Après avoir discuté avec le département de production des schmilblicks afin de déterminer quel est exactement le problème à résoudre, vous retournez dans votre bureau et commencez avec enthousiasme à travailler d'arrache-pied. Malheureusement, quelques semaines plus tard, vous êtes obligé de l'admettre : vous n'avez pas réussi à trouver un algorithme qui fasse substantiellement mieux que d'essayer toutes les possibilités – ce qui ne réjouira certainement pas votre patron puisque cela demandera des années et des années pour tester juste un ensemble de spécifications. Vous ne souhaitez certainement pas retourner dans le bureau de votre patron et lui dire : « Je ne parviens pas à trouver d'algorithme efficace. Je pense que je suis trop bête. »

Ce qui serait bien, c'est que vous soyez capable de démontrer que le problème du schmilblick est intrinséquement intractable[(3)]. Dans ce cas, vous pourriez sereinement aller voir votre patron et lui dire : « Je ne parviens pas à trouver d'algorithme efficace car un tel algorithme ne peut exister. »

Malheureusement, personne à ce jour n'a été capable de montrer qu'il existe un problème intrinsèquement intractable. En revanche, les théoriciens de l'informatique ont développé un concept qui est presqu'aussi bon, celui de problème **NP**-*complet*. La théorie de la **NP**-complétude fournit des techniques variées pour prouver qu'un problème est aussi difficile qu'une grande quantité d'autres problèmes, pour lesquels aucune méthode efficace n'a été trouvée à ce jour, malgré les efforts répétés des plus grands experts de la planète. Avec ces techniques, vous pourrez peut-être prouver que votre problème est **NP**-complet et donc équivalent à tous ces problèmes difficiles. Vous pourriez alors entrer dans le bureau de votre patron et annoncer : « Je ne parviens pas à trouver d'algorithme efficace, mais aucune de stars de la RO, aucune star de l'informatique théorique, aucune star de l'optimisation discrète ne peut le faire. » Au pire, cela l'informera qu'il ne sert à rien de vous virer pour embaucher un autre expert à votre place.

Prouver qu'un problème est **NP**-complet ne signifie pas la fin de l'histoire ; au contraire, c'est le début du vrai travail. Le fait de savoir qu'un problème est **NP**-complet fournit une information sur l'approche la plus productive. Cela montre qu'il ne faut pas se focaliser sur la recherche d'un algorithme exact et efficace et qu'il faut avoir des approches moins ambitieuses. Par exemple, on peut chercher des algorithmes efficaces résolvant divers cas particuliers. On peut chercher des algorithmes sans garantie sur le temps d'exécution, mais qui en général semblent

[(3)] anglicisme signifiant dans ce contexte qu'il n'existe pas d'algorithme efficace résolvant le problème considéré.

Fonction de complexité	Taille n					
	10	20	30	40	50	60
n	$0,01$ µs	$0,02$ µs	$0,03$ µs	$0,04$ µs	$0,05$ µs	$0,06$ µs
n^2	$0,1$ µs	$0,4$ µs	$0,9$ µs	$1,6$ µs	$2,5$ µs	$3,6$ µs
n^3	1 µs	8 µs	27 µs	64 µs	125 µs	216 µs
n^5	$0,1$ ms	$3,2$ ms	$24,3$ ms	$102,4$ ms	$312,5$ ms	$777,6$ ms
2^n	~ 1 µs	~ 1 ms	~ 1 s	~ 18 min20 s	~ 13 jours	~ 36 années et 6 mois

TAB. 1. Comparaison de diverses fonction de complexité pour un ordinateur effectuant 1 milliard d'opérations par seconde.

être rapides. Ou alors, on peut « relaxer » le problème et chercher un algorithme rapide qui trouve des schmilblicks satisfaisant presque toutes les spécifications. Dans les chapitres suivants, nous verrons des exemples concrets illustrant de telles approches.

2.5.4. Notions de base en théorie de la complexité ou comment gagner 1 million de $. — Pour évaluer théoriquement l'efficacité d'un algorithme résolvant un problème $\mathcal{P}$, on compte le nombre d'opérations élémentaires que cet algorithme effectue pour le résoudre. Comme la taille des données de $\mathcal{P}$ peut varier, on va avoir une *fonction de complexité* f qui à n, la taille des données, va associer le nombre $f(n)$ d'opérations élémentaires que l'algorithme va effectuer pour trouver une solution. La taille des données, c'est le nombre de bits qu'il faut pour coder les données.

Rappelons qu'en mathématique, dire qu'une fonction f est $O(g(n))$ (lire « grand 'o' de $g(n)$ »), c'est dire qu'il existe une constante B telle que $f(n) \leq Bg(n)$ pour tout n. Si la fonction de complexité d'un algorithme est un $O(p(n))$, où p est un polynôme, alors l'algorithme est dit *polynomial*. Sinon, il est dit *exponentiel*. Un algorithme polynomial est en général perçu comme efficace, un algorithme exponentiel est en général mauvais. Le Tableau 1 montre le temps qu'il faut à des algorithmes de fonction de complexité variable pour résoudre un problème pour différentes tailles de données.

On entend souvent dire : « Ces problèmes ne se poseront plus lorsque la puissance des ordinateurs aura augmenté ». C'est faux, comme le montre le Tableau 2. Supposons par exemple que l'on ait un algorithme résolvant le problème du schmilblick 2^n opérations élémentaires, où n est le nombre de spécifications (avec donc un ordinateur effectuant bien plus d'1 milliard d'opérations par seconde). Et supposons que l'on soit capable de résoudre des problèmes de schmilblick en 1 heure jusqu'à $n = 438$. L'utilisation d'une machine 1000 fois plus rapide ne permettra que d'aller jusqu'à $n = 448$ en 1 heure !

Un problème $\mathcal{P}$ est dite *polynomial* ou *dans* **P** s'il existe un algorithme polynomial qui le résout.

Un problème de décision $\mathcal{P}$ est *dans* **NP** si, lorsque la réponse est « oui », il existe un **certificat** (c'est-à-dire une information supplémentaire) et un algorithme polynomial qui permet de vérifier que la réponse est « oui », sans pour autant être capable de trouver cette réponse (voir des exemples ci-dessous). Le sigle **NP** ne signifie pas « non polynomial » mais « non déterministiquement polynomial ». Le non-déterminisme ici fait référence aux machines de Turing non-déterministe, et dépasse largement le cadre de ce cours.

Un problème de décision est dit **NP**-*complet* si l'existence d'un algorithme polynomial le résolvant implique l'existence d'un algorithme polynomial pour tout problème **NP**. A ce jour, on ne connaît pas d'algorithme polynomial résolvant un problème **NP**-complet. En revanche,

Taille de l'instance la plus large que l'on peut résoudre en 1 heure

Fonction de complexité	Avec un ordinateur actuel	Avec un ordinateur 100 fois plus rapide	Avec un ordinateur 1000 fois plus rapide
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

TAB. 2. Comparaison de diverses fonctions de complexité.

on connaît beaucoup de problèmes **NP**-complets. Le premier a été trouvé en 1970, par un informaticien appelé Cook.

Considérons le problème suivant.

Problème du cycle eulérien

Données : Un graphe $G = (V, E)$.

Question : G a-t-il un cycle eulérien ?

Ce problème de décision est dans **P**. En effet, le Théorème 2.2.1 ci-dessus permet facilement de répondre à la question en temps polynomial : si m est le nombre d'arêtes de G et n son nombre de sommets, tester le fait que tous les sommets sont de degré pair prend $O(m)$ et que le graphe est connexe prend $O(n + m)$, au total $O(n + m)$. Il est donc également dans **NP** : le certificat est le graphe lui-même.

De même, le problème suivant est dans **P**.

Problème de la chaîne eulérienne

Données : Un graphe $G = (V, E)$ et deux sommets $s, t \in V$.

Question : G a-t-il une s–t chaîne eulérienne ?

Considérons le maintenant problème suivant.

Problème du cycle hamiltonien

Données : Un graphe $G = (V, E)$.

Question : G a-t-il un cycle hamiltonien ?

A ce jour, nul n'a pu démontrer que ce problème est dans **P**, ni qu'il n'y était pas. En revanche, il est facile de voir que ce problème est dans **NP** car si la réponse est positive, alors l'algorithme qui consiste à suivre le cycle hamiltonien permet de prouver que la réponse est bien positive. Ici, le cycle hamiltonien joue le rôle de certificat. Il a été également démontré que ce problème est **NP**-complet. Cela signifie que si vous rencontrez quelqu'un vous disant qu'il connaît un algorithme efficace pour répondre à cette question, il est très probable qu'il se trompe car tous les problèmes **NP**-complets pourraient alors être résolus par un algorithme polynomial. Or personne à ce jour n'a pu en trouver, et quantité de gens très brillants ont cherché un tel algorithme. S'il ne se trompe pas, c'est une découverte fondamentale, qui aurait un impact énorme tant dans

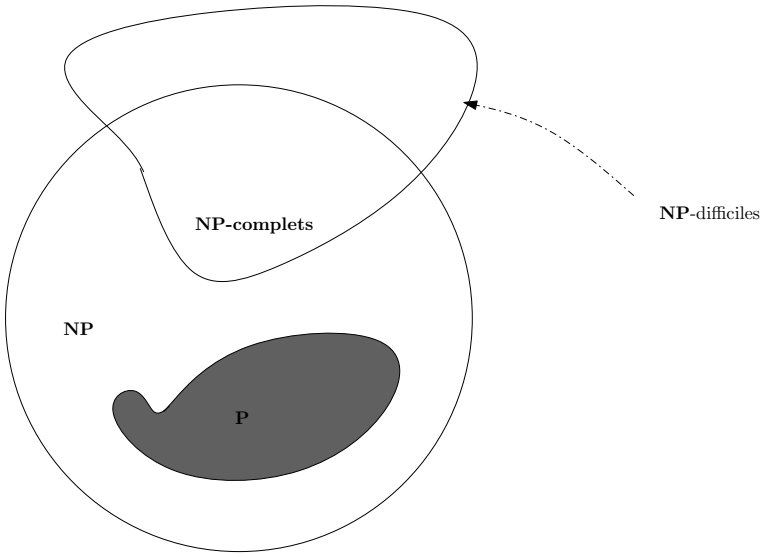


FIG. 10. Les problèmes **NP**

le monde de l'informatique théorique, que dans la recherche opérationnelle appliquée. De plus, cette personne percevrait le prix d'1 millions de dollars offert par la Fondation Clay pour la résolution de la question ouverte $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

De même le problème suivant est **NP**-complet.

Problème de la chaîne hamiltonienne

Données : Un graphe $G = (V, E)$, deux sommets $s, t \in V$.

Question : G a-t-il une s–t chaîne hamiltonienne ?

On peut définir également les mêmes problèmes pour les graphes orientés, les résultats seront semblables. En résumé, on a

Résultats de complexité pour des parcours.	
Problèmes	Complexité
Graphe non-orienté : Cycle eulérien	**P**
Graphe non-orienté : Chaîne eulérienne	**P**
Graphe non-orienté : Cycle hamiltonien	**NP**-complet
Graphe non-orienté : Chaîne hamiltonienne	**NP**-complet
Graphe orienté : Circuit eulérien	**P**
Graphe orienté : Chemin eulérien	**P**
Graphe orienté : Circuit hamiltonien	**NP**-complet
Graphe orienté : Chemin hamiltonien	**NP**-complet

La Figure 10 illustre les relations entre problèmes **P**, **NP**, **NP**-complets, si $\mathbf{P} \neq \mathbf{NP}$ (ce qui est le plus probable).

La notion de problème **NP** et a fortiori de problème **NP**-complet est définie pour les problèmes de décision, dont la réponse est « oui » ou « non ». Mais les problèmes que l'on rencontre en RO sont rarement de ce type. Il existe une notion qui permet de caractériser des problèmes difficiles : c'est celle de problème **NP**-difficile.

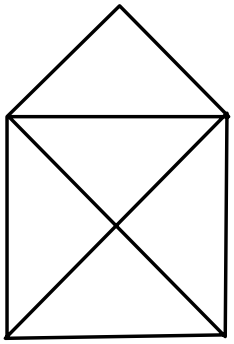


FIG. 11. Peut-on tracer cette figure sans lever le crayon ?

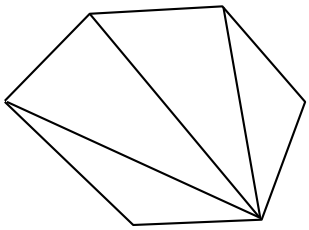


FIG. 12. Et celle-là ?

Un problème est **NP**-*difficile* si savoir le résoudre en temps polynomial impliquerait que l'on sait résoudre un problème (de décision) **NP**-complet en temps polynomial. Les problèmes **NP**-difficiles sont donc dans un sens plus dur que les problèmes **NP**-complets. Par exemple :

Théorème 2.5.1. — *Le Problème du voyageur de commerce est* **NP**-*difficile.*

La preuve de ce résultat est laissée en exercice.

2.6. Exercices

2.6.1. Les Ponts de Königsberg. — En 1736, les notables de Königsberg demandèrent à Euler s'il était possible de parcourir les ponts de la ville en passant sur chacun des 7 ponts exactement une fois (voir Figure 15). C'est le premier problème de Recherche Opérationnelle (à visée non pratique). Ce genre de problème se rencontre maintenant très souvent dans les problèmes de tournées du type facteur ou ramassage de déchets ménagers, dans lesquels il faut parcourir les rues d'une ville de façon optimale. Euler trouva la solution...

Un tel parcours existe-t-il ?

2.6.2. Dessiner sans lever le crayon. — En utilisant le résultat de l'exercice précédent, indiquez pour les Figures 11 et 12 quand il est possible de dessiner la figure sans lever le crayon (et sans repasser sur un trait déjà dessiné) et quand il ne l'est pas.

2.6.3. Le voyageur de commerce. — Un camion doit effectuer des livraisons en un certain nombre de points d'un réseau consitué de routes. On connaît le temps qu'il faut pour parcourir chaque portion de route. On veut minimiser le temps qu'il faut au camion pour effectuer ses livraisons et revenir à son point de départ. Modéliser ce problème comme un problème de cycle hamiltonien de poids minimum.

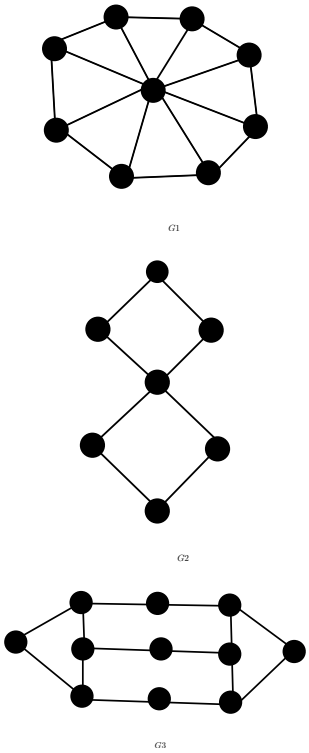


FIG. 13. Lequel de ces graphes contient une chaîne hamiltonienne ? Lequel contient un cycle hamiltonien ?

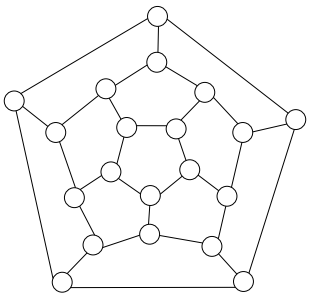


FIG. 14. Le jeu Icosian.

2.6.4. Parcours hamiltonien. — Pour chacune des trois graphes suivants, indiquez pour chacun d'eux s'il possède une chaîne hamiltonienne. Même question avec pour le cycle hamiltonien.

Justifiez votre réponse.

2.6.5. Le jeu Icosian. — La Figure 14 représente le jeu Icosian inventé en 1859 par le mathématicien Hamilton. Peut-on mettre les chiffres de 1 à 20 dans les cercles de manière à ce que toute paire de nombres consécutifs soit adjacente, 1 et 20 étant aussi considérés comme adjacents ?

2.6.6. Affectation de fréquences. — Considérons des relais de téléphonie mobile. Chaque relais reçoit et émet des signaux à une certaine fréquence. Pour éviter les interférences, on veut que deux relais à moins de 200 m fonctionnent avec des fréquences différentes. Le coût

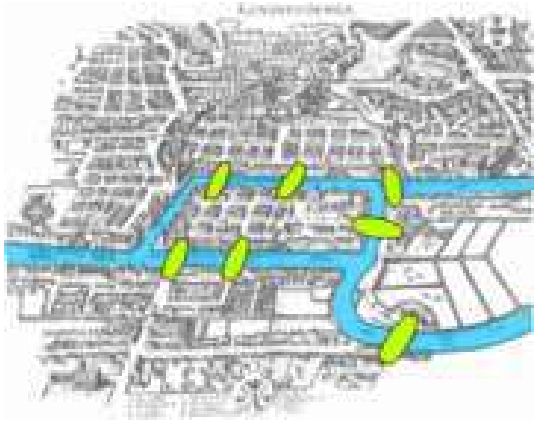


FIG. 15. Königsberg et ses 7 ponts

de gestion du parc de relais est une fonction croissante du nombre de fréquences utilisées. On souhaite minimiser ce coût.

Modéliser ce problème avec les outils du cours.

2.6.7. Organisation de formations. — Considérons maintenant un DRH qui doit assurer un certain nombre de formations à ses employés. Les formations sont $F_1, \ldots, F_n$. Et les employés sont $E_1, \ldots, E_m$. Le DRH sait pour chaque employé E_i quelles sont les formations qu'il doit suivre. Il veut organiser une session de formations – chacune des formation ne peut être dispensée qu'une fois et un employé peut suivre au plus une formation par jour. Le DRH souhaite organiser la session la plus courte possible[(4)].

Modéliser ce problème comme un problème de coloration.

2.6.8. Borne inférieure du nombre chromatique. — Montrer l'optimalité de la coloration de la Figure 8, et proposer une borne inférieure générale pour les nombre chromatique.

2.6.9. Borne supérieure pour les couplages maximum. — Montrer l'optimalité du couplage de la Figure 9, et proposer une borne supérieure générale pour les couplages maximum.

2.6.10. NP-complétude de la chaîne hamiltonienne. — Montrer que le problème de l'existence de la chaîne hamiltonnienne est **NP**-complet, sachant que celui du cycle hamiltonien l'est. Réciproquement, montrer que le problème de l'existence du cycle hamiltonnien est **NP**-complet, sachant que celui de la chaîne hamiltonienne l'est.

2.7. NP-difficulté du problème du voyageur de commerce

Montrer que le problème du voyageur de commerce est **NP**-difficile en utilisant la **NP**-complétude du problème du cycle hamiltonien. On utilisera le résultat de l'exercice **??**

[(4)] Source : Nadia Brauner.

2.7.1. Problème de transport. — Le premier problème de Recherche Opérationnelle à visée pratique a été étudié par Monge en 1781 sous le nom du problème des déblais et remblais. Considérons n tas de sable, devant servir à combler m trous. Notons a_i la masse du $i$ème tas de sable et b_j la masse de sable nécessaire pour combler le $j$ème trou. Pour chaque couple (i, j) on connaît la distance d_{ij} du $i$ème tas au $j$ème trou. Le coût d'un déplacement est proportionnel à la distance parcourue et à la masse déplacée. On souhaite savoir quel est le plan de transport qui permet de boucher les trous au coût minimum.

Modéliser ce problème sous la forme d'un programme mathématiques. Quelle est sa particularité ?

CHAPITRE 3

PLUS COURTS CHEMINS ET PROGRAMMATION DYNAMIQUE

Les problèmes de plus courts chemins apparaissent naturellement dans des contextes variés. Ils peuvent apparaître comme modélisation de problèmes opérationnels (trajet le plus rapide, ou le moins cher, gestion optimal d'un stock, plan d'investissement optimal, etc.), ils sont aussi fréquemment des sous-problèmes d'autres problèmes d'optimisation (flot de coût minimum – Chapitre 5 – ou en théorie de l'ordonnancement – Chapitre 10). Sous cette appellation anodine de plus court chemin, nous verrons qu'il se cache une réalité extrêmement variée, tant du point de vue de la complexité que des types d'algorithmes mis en œuvre. Le graphe peut être orienté ou non, les arêtes ou les arcs avoir des poids positifs ou quelconque,... Un cas particulier important est la *programmation dynamique* qui traite des situations où des décisions doivent être prises de manière séquentielle, chaque décision faisant passer le système d'un état à un autre. Ce passage d'un état à un autre s'appelle une *transition*. On suppose qu'un coût est associé à chaque transition. L'objectif de la programmation dynamique est de minimiser le coût de la suite des transitions, suite qu'on appelle *trajectoire*.

Dans ce qui suit, le nombre de sommets d'un graphe sera noté n et son nombre d'arêtes ou d'arcs sera noté m.

3.1. Cas du graphe orienté et programmation dynamique

3.1.1. Les poids sont positifs : algorithme de Dijkstra. — Le cas le plus naturel est le cas du graphe orienté avec des poids strictement positif sur les arcs. En effet, le problème du trajet le plus court dans un réseau (routier, de transport, informatique, etc.) rentre dans ce cadre. Le temps de parcours d'un arc, ou sa longueur, sont dans ces contextes strictement positifs.

En 1959, Dijkstra proposa le premier algorithme efficace. On suppose que l'on a un graphe orienté $D = (V, A)$, avec une fonction de poids $w : A \to \mathbb{R}_+$. On se fixe un sommet s de départ et un sommet t d'arrivée. On veut le chemin de plus petit poids (le poids d'un chemin étant la somme des poids de ses arcs), ou autrement dit le plus court chemin allant de s à t. Noter que dans un graphe dont tous les poids sont positifs, il existe toujours un plus court chemin qui soit un chemin élémentaire (avec répétition de sommets interdite) car si le plus court chemin passe par un circuit, on peut supprimer ce circuit sans détériorer le poids du chemin. Si tous les poids sont strictement positifs, les notions de plus court chemin, plus court chemin simple (avec répétition d'arcs interdite) et plus court chemin élémentaire (avec répétition de sommets interdite) coïncident puisqu'alors suivre un circuit détériore strictement le poids du chemin.

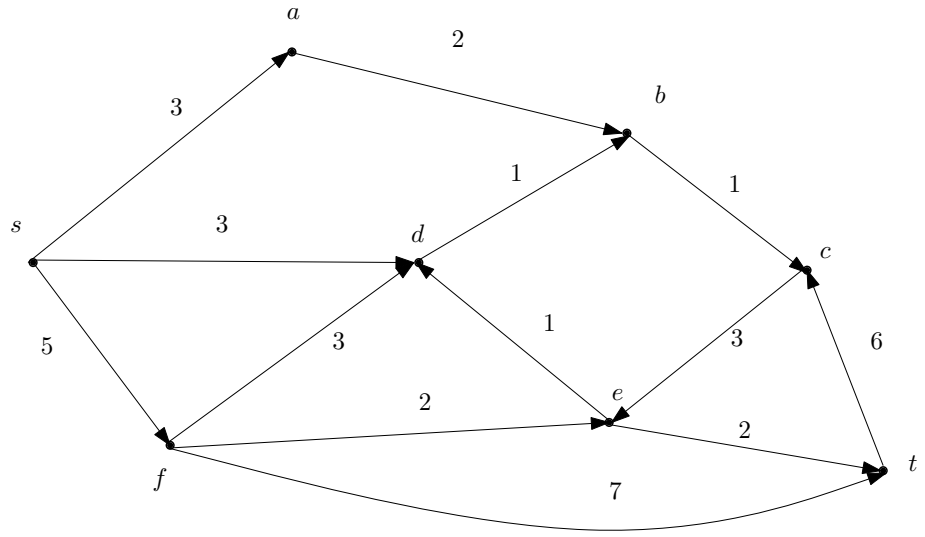


FIG. 1. Exemple de graphe orienté avec poids positifs sur les arcs.

L'algorithme de Dijkstra va en fait retourner les plus courts chemins démarrant en s pour tous les sommets d'arrivée possibles, donc en particulier pour t.

Au cours de l'algorithme, on garde un ensemble $U \subseteq V$ et une fonction $\lambda : V \to \mathbb{R}_+$, on dit que $\lambda(v)$ est le *label* courant de v. Ce label est la meilleure évaluation courante du poids du plus court chemin de s à v. On commence avec $U := V$ et $\lambda(s) := 0$ et $\lambda(v) := +\infty$ pour $v \neq s$. Ensuite, on répète la chose suivante de manière itérative :

> Choisir u minimisant $\lambda(u)$ pour $u \in U$. Pour chaque $a = (u, v) \in A$ pour lequel $\lambda(v) > \lambda(u) + w(a)$, redéfinir $\lambda(v) := \lambda(u) + w(a)$. Redéfinir $U := U \setminus \{u\}$.

On s'arrête lorsque $\lambda(u) = +\infty$ pour tout $u \in U$. La fonction λ donne les distances de s à tout sommet que l'on peut atteindre en partant de s. De plus, si pour chaque $v \neq s$, on garde en mémoire le dernier arc $a = (u, v)$ pour lequel on a redéfini $\lambda(v) := \lambda(u) + w(a)$, on peut reconstruire l'ensemble des plus courts chemins partant de s.

Il est clair que le nombre d'itérations est au plus $n = |V|$, et chaque itération prend un temps $O(n)$. L'algorithme a donc une fonction de complexité qui est en $O(n^2)$. D'où le théorème :

Théorème 3.1.1. — *Etant donné un graphe orienté $D = (V, A)$ $s \in V$ et une fonction de poids $w : A \to \mathbb{R}_+$, les chemins de plus petit poids de s à tout sommet v peuvent être trouvés en $O(n^2)$.*

Eléments de preuve. — Pour démontrer le théorème, il faut simplement voir que lorsque u est retiré de U dans l'itération au-dessus, $\lambda(u)$ est la distance de s à u (la preuve de cette assertion est laissée en exercice). □

Remarque : En utilisant de bonnes structures de données, il est possible d'obtenir une complexité de $O(m + n \log n)$.

Exemple : En appliquant l'algorithme de Dijkstra sur la Figure 1, on obtient le tableau suivant indiquant les valeurs successives de λ pour chaque sommet.

s	a	b	c	d	e	f	t
(0)	(∞)	(∞)	(∞)	(∞)	(∞)	(∞)	(∞)
0	(3)	(∞)	(∞)	(3)	(∞)	(5)	(∞)
0	3	(5)	(∞)	(3)	(∞)	(5)	(∞)
0	3	(4)	(∞)	3	(∞)	(5)	(∞)
0	3	4	(5)	3	(∞)	(5)	(∞)
0	3	4	5	3	(8)	(5)	(∞)
0	3	4	5	3	(7)	5	(12)
0	3	4	5	3	7	5	(9)
0	3	4	5	3	7	5	9

3.1.2. Les poids sont quelconques mais le graphe est sans circuit, la programmation dynamique. — On considère maintenant le cas où les poids peuvent prendre des valeurs négatives, mais où le graphe est sans circuit, ou *acircuitique*. Lorsqu'on voit le problème du plus court chemin comme la modélisation d'un plus court chemin dans un réseau physique, cela peut paraître surprenant puisque les temps de parcours ou les distances de portion de réseau sont toujours positifs. Mais dans de nombreux cas le graphe dont on cherche un plus court chemin est issu d'une modélisation et les poids ne correspondent pas nécessairement à des distances ou des temps qui s'écoulent.

Noter que dans ce cas, les notions de plus court chemin, plus court chemin simple et plus court chemin élémentaire coïncident, puisqu'il n'y a pas de circuit. Donc de même qu'avant, il existe toujours un plus court chemin qui soit élémentaire.

3.1.2.1. Algorithme. — Considérons donc maintenant le cas du graphe orienté $D = (V, A)$ acircuitique, avec une fonction de poids $w : A \to \mathbb{R}$, qui, comme on vient de le voir, contient le cas de la programmation dynamique.

L'algorithme de Dijkstra ne fonctionne plus lorsque les poids peuvent être négatifs. En effet, l'invariant de boucle qui fait fonctionner l'algorithme de Dijkstra est le fait que le minimum de $\lambda(u)$ pour u dans le U courant est la vraie distance de s à u. Ce qui justifie que l'on retire u de U. Or si les poids peuvent être négatifs, il facile de voir que cette propriété n'est plus vraie.

A la fin des années 50, des mathématiciens comme Ford, Bellman, Moore, et d'autres, remarquèrent que dans ce contexte, un algorithme très simple permet de trouver le plus court chemin.

Pour décrire cet algorithme, définissons pour $v \in V$:

$$\lambda(v) := \text{ poids minimum d'un } s\text{–}v \text{ chemin},$$

posant $\lambda(v) := \infty$ si un tel chemin n'existe pas.

L'algorithme est une traduction de l'*équation de Bellman* qui dit

$$\lambda(v) = \min_{(u,v)\in A} (\lambda(u) + w(u, v)) \tag{2}$$

pour tout $v \in V$. Cette équation peut être lue de la manière suivante :

> Le plus court chemin de s à v est le plus court chemin de s à u, où u est un antécédent de v, auquel on a ajouté le dernier arc (u, v).

En fait, de façon un peu plus compacte, on a le principe suivant qui implique ce qui précède

> [Principe d'optimalité de Bellman] La sous-trajectoire d'une trajectoire optimale est encore optimale.

L'équation (2) se traduit facilement en un algorithme. On commence par poser $\lambda(s) := 0$ et $\lambda(u) := \infty$ pour tout u tel qu'il n'existe pas de s–u chemin. Ensuite, on repète la boucle suivante

> Chercher un sommet v dont on connaît λ pour tous les prédécesseurs (on peut démontrer que, puisque le graphe est acircuitique, un tel sommet existe toujours). Avec l'équation (2) on peut alors calculer $\lambda(v)$.

On s'arrête lorsque λ aura été calculé pour tous les sommets.

Remarquons qu'il n'est pas difficile de reconstruire le plus court chemin : en même temps que $\lambda(v)$, on peut stocker $p(v)$ qui est le sommet u antécédent de v qui a permis d'obtenir le poids $\lambda(v)$. Comme pour l'algorithme de Dijkstra, on récupère ainsi un plus court chemin de s à v, pour tout $v \in V$.

Théorème 3.1.2. — *Etant donné un graphe acircuitique $D = (V, A)$, et une fonction de poids $w : A \to \mathbb{R}$, des plus courts chemins de s à tout sommet $v \in V$ peut être calculé en $O(m)$.*

Éléments de preuve. — Pour prouver ce théorème, il faut montrer que l'on peut trouver en $O(m)$ un ordre $v_1, v_2, \ldots, v_n$ sur les sommets de D tel que tous les antécédents de v_i ont un indice $< i$, ce qui assure que l'on peut toujours calculer $\lambda(v_i)$ une fois calculés les $\lambda(v_j)$ pour $j < i$. Or, ceci est une résultat classique de l'algorithmique de graphe : on dit que l'on cherche un « ordre topologique » (expliqué ci-dessous). □

Ordre topologique. — Un *ordre topologique* des sommets d'un graphe orienté D est une indexation $v_1, v_2, \ldots, v_n$ des sommets de D telle que pour tout i, j tels que (v_i, v_j) on ait $i < j$. Un graphe orienté admet un ordre topologique si et seulement si il est acircuitique. On peut alors trouver un ordre topologique en $O(m)$ avec l'algorithme suivant.

Scanner un sommet consiste à "ouvrir" le sommet, scanner ses successeurs puis "fermer" le sommet (définition récursive, consistante puisque le graphe est sans circuit). On choisit un sommet s sans prédécesseur. On scanne s. L'ordre dans lequel on "ferme" les sommets est un ordre topologique.

En effet, s'il y a un arc de u à v, alors v sera "fermé" avant u et son indice sera plus grand que celui de u.

Remarque : Plus long chemin Dans le cas acircuitique, on sait en particulier trouver un plus long chemin d'une origine s à une destination t : en effet, en définissant des poids $w'(a) := -w(a)$, on voit que le même algorithme fonctionne (en gardant les mêmes poids, il suffit de changer les min en max).

3.1.2.2. Programmation dynamique. — Un cas particulier important est celui de la *programmation dynamique* qui traite de situations où les décisions peuvent ou doivent être prises de manière séquentielle. Ce qui caractérise la programmation dynamique[(1)] est la présence

1. un système dynamique à temps discret caractérisé par un ensemble X d'*états* possibles et
2. une fonction de coût additive dans le temps.

Le système dynamique décrit l'évolution de l'état du processus au fil de N périodes et a la forme

$$x_{k+1} = f_k(x_k, u_k), \quad k = 1, 2, \ldots, N$$

[(1)] On se limite au cas déterministe discret.

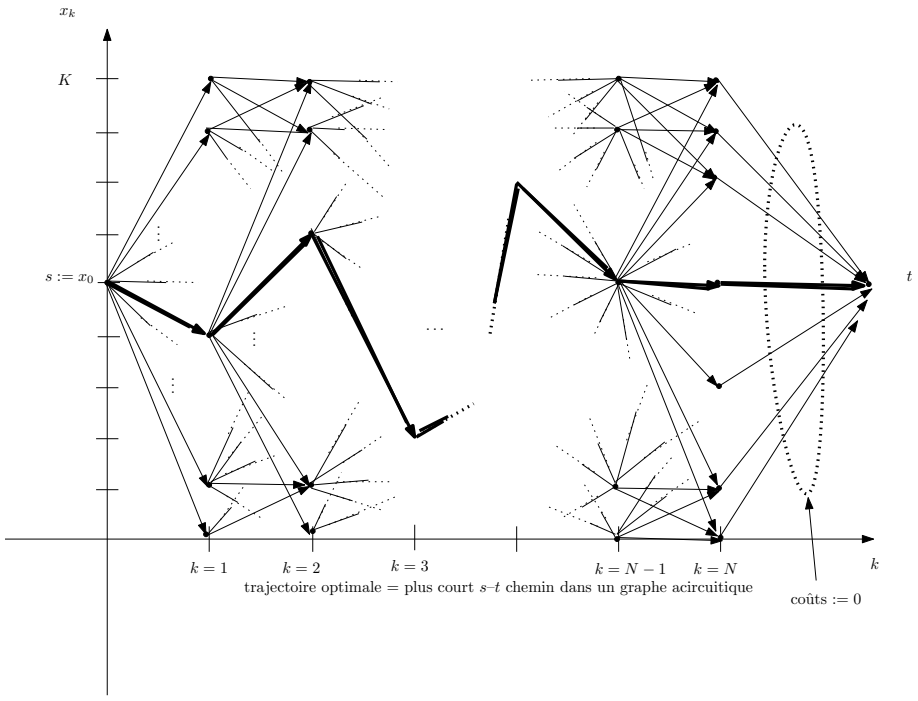


FIG. 2. La programmation comme problème de plus court chemin.

où k numérote les périodes et où x_k est l'*état* du système au début de la période k. De plus, on a un coût $c(x_k, x_{k+1})$ pour toute *transition* de l'état x_k à l'état x_{k+1}. Ces coûts s'additionnent au fil des périodes et pour la *trajectoire* complète $x_1, \ldots, x_N$, on a un coût total de

$$\sum_{k=1}^{N} c(x_k, x_{k+1}).$$

Le problème de la programmation dynamique consiste à trouver la trajectoire optimale, i.e. minimisant le coût total.

Ce problème de la programmation dynamique se modélise très bien comme celui du plus court chemin dans un graphe orienté, pondéré et sans circuit. En effet, pour chaque période k, on représente les états possibles par des sommets. Les transitions possibles sont des arcs. Le coût de la transition $x_k \to x_{k+1}$ est le poids de l'arc (x_k, x_{k+1}). Noter que ce graphe est sans circuit (ou *acircuitique*) : en effet, un arc va toujours d'un état de période k à un état de période $k+1$. En supposant que les états de départ s et d'arrivée t sont fixés, on va chercher le chemin de s à t qui a le plus petit poids. Voir Figure 2.

Si on définit

$\lambda(k, x) :=$ coût minimum de la trajectoire allant de l'état de départ x_1 à l'état x en k étapes,

posant $\lambda(k, x) := +\infty$ s'il n'est pas possible d'amener le système dans l'état x en k étapes en partant de l'état x_1, l'équation de Bellman (2) se réécrit ainsi :

$$\lambda(k, x) = \min_{y \in X}(\lambda(k-1, y) + c(y, x)). \tag{3}$$

Noter que dans le cas d'un graphe de programmation dynamique, l'orde topologique se construit simplement en utilisant le découpage en périodes de temps.

Remarquons enfin que tout graphe orienté acircuitique n'est pas issu d'une modélisation d'un système dynamique. En effet, un graphe modélisant un processus de programmation dynamique

a tous ses chemins de s à t ayant le même nombre d'arcs, à savoir N, ce qui n'est pas le cas du graphe de la Figure 3.4.2.

Venons-en maintenant à quelques exemples de modélisation par la programmation dynamique.

Exemple : Gestion de stock

Considérons le problème de la gestion d'un stock pour N périodes consécutives.

Un stock suit une dynamique de la forme $x_{k+1} = x_k - d_k + u_k$, où d_k désigne la demande pour la période k, supposée connue, x_k désigne le nombre d'unités disponibles en début de $k$ième période et u_k désigne le nombre d'unités commandées (et reçues immédiatement) en début de $k$ième période. On prendra $x_k \in \mathbb{Z}$, une quantité négative signifiant une pénurie. De plus, on a en général l'existence d'une capacité maximale de stockage K qui implique que $x_k \leq K$. On supposera x_0 fixé.

Les coûts de gestion de stock se décomposent pour chaque période k en

- un coût de réapprovisionnement $c(u_k)$, c'est-à-dire le coût d'achat des u_k unités, et
- un coût de gestion $g(x_{k+1})$, qui comprend le coût de stockage des unités en excès en fin de période et le coût de pénurie si la demande ne peut être totalement satisfaite sur la période.

Typiquement, on a $c(u) = b + au$, où a est un coût unitaire d'achat et b un coût fixe et

$$g(x) = \begin{cases} sx & \text{si } x \geq 0 \\ px & \text{si } x \leq 0, \end{cases}$$

où s est le coût unitaire de stockage et p le coût unitaire de défaillance. Si on veut interdire les stocks négatifs, on pose $p = +\infty$.

On veut minimiser

$$\sum_{k=0}^{N-1} c(u_k) + g(x_{k+1}).$$

Dans le vocabulaire de la programmation dynamique, les états sont les valeurs possibles de niveau de stock, les transitions possibles sont tous les $x_k \to x_{k+1}$ tels que x_{k+1} satisfasse simultanément $x_{k+1} \in \mathbb{Z}$, $x_{k+1} \leq K$ et $x_{k+1} \geq x_k - d_k$. Le coût de la transition $x_k \to x_{k+1}$ est $c(x_{k+1} - x_k + d_k) + g(x_{k+1})$. On souhaite donc trouver la trajectoire optimale pour ce modèle de programmation dynamique.

On applique l'algorithme de plus court chemin dans le graphe acircuitique suivant. Pour chaque période k, on aura un sommet pour chaque niveau de stock possible (inférieur à K et supérieur à $x_0 - d_0 - d_1 - \ldots - d_{k-1}$) en début de période. On notera un tel sommet (x, k), où x est le niveau de stock, et k la période. Entre un sommet de période k et un sommet de période $k+1$, on a un arc $((x, k), (x', k+1))$ seulement si $x' \geq x - d_k$. Le poids de l'arc $((x, k), (x', k+1))$ est égal à $c(x' - x + d_k) + g(x')$. On cherche un plus court chemin de $(x_0, 0)$ à un sommet de la forme (x, N). Le poids de ce plus court chemin sera le coût optimal de la gestion du stock.

Exemple : Remplissage d'un conteneur ou problème du sac-à-dos

Considérons un assortiment de N objets $j = 1, \ldots, N$, l'objet j ayant une valeur $v_j \geq 0$ et un poids $w_j > 0$, et supposons que l'on ait à stocker ces objets dans un conteneur de capacité en poids finie W. Si $\sum_{j=1}^{N} w_j > W$, on va devoir sélectionner des objets afin de maximiser la valeur des objets stockés. Ce problème est connu sous le nom du PROBLÈME DU SAC-À-DOS. On note $x(k, W')$ la valeur maximale qui peut être transportée dans un sac de capacité W', en ne se servant que des k premiers objets.

On a l'équation de Bellman

$$x(k, W') = \max(x(k-1, W'), x(k-1, W' - w_k) + v_k)$$

qui se lit : le remplissage optimal $x(k, W')$ d'un sac de capacité W', avec les objets de 1 à k,

– soit n'utilise pas le $k$ème objet, et dans ce cas $x(k, W') = x(k-1, W')$

– soit utilise le $k$ème objet, et dans ce cas $x(k, W') = x(k-1, W' - w_k) + v_k$.

La complexité de l'algorithme est : $O(nW)$. En effet, il faut calculer tous les $x(k, W')$ qui sont au nombre de nW, et une fois calculés $x(k-1, W')$ et $x(k-1, W' - w_k)$, la quantité $x(k, W')$ se calcule en temps constant. Ce n'est pas polynomial car W nécessite $\log_2(W)$ bits pour être codés. On dit que c'est un algorithme *pseudo-polynomial*.

3.1.3. Les poids sont quelconques, mais le graphe est sans circuit absorbant : encore la programmation dynamique. — Supposons maintenant que l'on se demande comment calculer un plus court chemin dans un graphe orienté pondéré, non nécessairement acircuitique. C'est encore possible si le graphe ne possède pas de *circuit absorbant*, i.e. de circuit dont la somme des poids est < 0. Ce cas contient les deux cas précédents (graphe avec poids ≥ 0 et graphe acircuitique). Noter qu'une fois encore, il existe toujours un plus court chemin qui soit un chemin élémentaire car si le plus court chemin passe par un circuit, on peut supprimer ce circuit sans détériorer le poids du chemin.

En fait, pour calculer un plus court chemin dans un graphe sans circuit absorbant, on peut appliquer la méthode de la programmation dynamique vue ci-dessus.

3.1.3.1. Résolution par programmation dynamique. — On définit les états comme étant les sommets du graphe, et on voit chaque période comme la traversée d'un arc. Avec les notations de la section précédente, on a

$$\lambda(k, v) := \text{ poids minimum d'un } s\text{–}v \text{ chemin traversant exactement } k \text{ arcs},$$

posant $\lambda(k, v) := \infty$ si un tel chemin n'existe pas.

L'équation de Bellman dit alors

$$\lambda(k+1, v) = \min_{(u,v)\in A} (\lambda(k, u) + w(u, v)) \tag{4}$$

pour tout $v \in V$. Cette équation peut être lue de la manière suivante :

> Le plus court chemin de s à v utilisant exactement $k+1$ arcs est un plus court chemin de s à u utilisant exactement k arcs, pour un certain u antécédent de v, auquel on a ajoute le dernier arc (u, v).

3.1.3.2. Algorithme. — L'équation (4) se traduit facilement en un algorithme. On commence par poser $\lambda(0, s) := 0$ et $\lambda(0, v) := \infty$ si $v \neq s$. Ensuite, on calcule l'ensemble des valeurs de $\lambda(1, \cdot)$ avec l'équation (4). Une fois que cela est fait, il est possible, toujours en utilisant l'équation (4), de déduire l'ensemble des valeurs de $\lambda(2, \cdot)$. On peut continuer ainsi jusqu'à obtenir $\lambda(n-1, t)$. La longueur du plus court chemin est alors $\min_{k=0,\dots,n-1} \lambda(k, t)$. En effet, s'il n'y a pas de circuit absorbant et s'il y a un chemin de s à t, il y a alors un plus court chemin de s à t qui est élémentaire : s'il y a un circuit sur le plus court chemin, il ne peut pas être de poids > 0, sinon, ce ne serait pas un plus court chemin, et il n'est pas de poids < 0 car un tel circuit est absorbant, il est dont de poids $= 0$ et on peut le supprimer du chemin. Comme ce chemin est élémentaire, il a au plus $n-1$ arcs.

Remarquons qu'il n'est pas difficile de reconstruire le plus court chemin : en même temps que $\lambda(k, v)$, on peut stocker $p_k(v)$ qui est le sommet u antécédent de v qui a permis d'obtenir le poids $\lambda(k, v)$. Comme ci-dessus, on récupère ainsi un plus court chemin de s à v, pour tout $v \in V$.

Théorème 3.1.3. — *Etant donné un graphe $D = (V, A)$, et une fonction de poids $w : A \to \mathbb{R}$ telle qu'il n'y ait pas circuit absorbant, des plus courts chemins de s à tout sommet $v \in V$ peuvent être calculés en $O(nm)$.*

Le $O(nm)$ vient du fait que le calcul de $\lambda(k, \cdot)$ prend $O(m)$ (on regarde tous les arcs du graphe), et que l'on va jusqu'à $\lambda(n-1, \cdot)$.

Remarque : En utilisant des techniques totalement différentes [**12**], on peut obtenir une complexité de $O(n^{1/2}m \log C)$ où C est la valeur abolue du plus petit poids négatif (en supposant que $C \geq 2$).

3.1.4. Les poids sont quelconques. — Dans le cas général, il n'y a pas d'algorithme polynomial qui trouve un plus court chemin élémentaire. Rappelons qu'un chemin élémentaire est un chemin dans lequel chaque sommet apparaît au plus une fois. Si l'on ne se restreint pas au chemin élémentaire, le problème peut ne pas avoir de solution : en effet, en faisant passer un chemin par un circuit absorbant, on peut répéter un nombre arbitraire de fois le passage sur le circuit absorbant et décroître le coût autant qu'on le souhaite.

Considérons le problème

PROBLÈME DU PLUS COURT CHEMIN ÉLÉMENTAIRE

Données : Un graphe orienté $D = (V, A)$, une fonction de poids $w : A \to \mathbb{R}$, deux sommets $s \in V$ et $t \in V$.

Tâche : Trouver un plus court chemin élémentaire de s à t.

On a le théorème suivant dont la preuve est laissée en exercice.

Théorème 3.1.4. — *Le PROBLÈME DU PLUS COURT CHEMIN ÉLÉMENTAIRE avec des poids quelconques, dans un graphe orienté quelconque, est* **NP***-difficile.*

3.2. Cas du graphe non orienté

3.2.1. Les poids sont positifs : toujours l'algorithme de Dijkstra. — Considérons un graphe (non-orienté) $G = (V, E)$, avec une fonction de poids $w : E \to \mathbb{R}_+$. Supposons fixés deux sommets s et t. On veut trouver le plus court chemin de s à t. Avec la terminologie des graphes introduite dans le Chapitre 2, on cherche une chaîne de poids minimum. Remarquons que le raisonnement fait ci-dessus s'applique encore : on sait qu'il existe une chaîne élémentaire de même poids que la chaîne de plus petit poids puisqu'on peut supprimer les cycles d'une chaîne de plus petit poids (ces cyles sont alors forcéments de poids nul).

L'algorithme de Dijkstra permet encore de résoudre le problème : en effet, il suffit de remplacer chaque arête uv du graphe par deux arcs (u, v) et (v, u) et de mettre sur chacun d'eux le poids de l'arête uv. Voir Figure 3. Un plus court chemin dans ce graphe orienté induit alors une plus courte chaîne dans le graphe de départ et réciproquement.

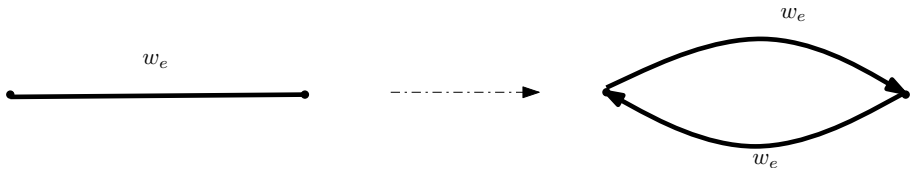


FIG. 3. Une transformation qui fait apparaître des circuits.

3.2.2. Les poids sont quelconques, mais sans cycle absorbant. — Il peut être tentant de vouloir refaire la même transformation que dans la sous-section précédente, à savoir de remplacer chaque arête par deux arcs de sens opposés, chacun recevant le poids de l'arête dont ils sont issus. Le problème, c'est que dans cette transformation, on fait apparaître quantité de circuits absorbants : toute arête de poids < 0 induit un circuit absorbant à deux arcs. Les algorithmes de style programmation dynamique ne peuvent pas appliquer. Il n'y a pas de raison particulière de penser que ce problème soit **NP**-difficile car le problème qui est **NP**-difficile est celui où l'on accepte des circuits absorbants avec un nombre d'arcs non borné. Ici nos circuits absorbants ont 2 arcs seulement.

En fait, il y a bien une méthode mais curieusement elle provient d'un tout autre domaine de l'optimisation discrète et nous en reparlerons dans le chapitre sur les tournées (Chapitre 11). L'outil dont on a besoin est le *T-joint* qui apparaît naturellement dans les problèmes de tournée du style « postier chinois ». Avec cet outil, on peut trouver une plus courte chaîne élémentaire en $O(n^3)$.

3.2.3. Les poids sont quelconques. — Dans le cas général, le problème est à nouveau **NP**-difficile.

Considérons le problème

PROBLÈME DE LA PLUS COURTE CHAÎNE ÉLÉMENTAIRE

Données : Un graphe $G = (V, E)$, une fonction de poids $w : E \to \mathbb{R}$, deux sommets $s \in V$ et $t \in V$.

Tâche : Trouver une plus courte chaîne élémentaire de s à t.

On a le théorème suivant.

Théorème 3.2.1. — *Le PROBLÈME DE LA PLUS COURTE CHAÎNE ÉLÉMENTAIRE avec des poids quelconques, dans un graphe quelconque, est* **NP***-difficile.*

Démonstration. — Même preuve que dans le cas orienté. □

3.3. Résumé

Rappelons que n représente le nombre de sommets du graphe, et m son nombre d'arêtes ou d'arcs.

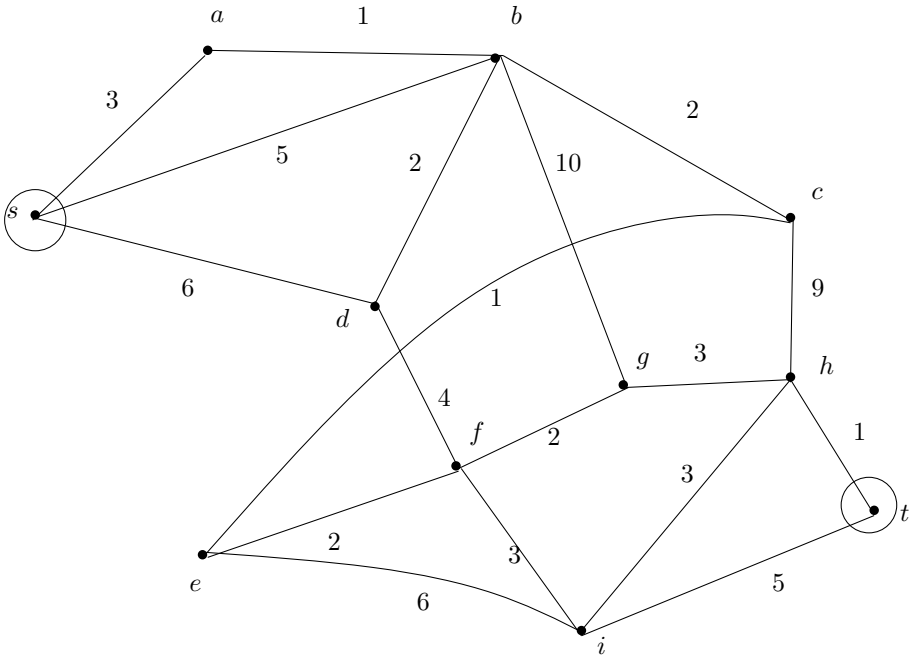


FIG. 4. On cherche la plus courte chaîne de s à t.

	Complexité
Graphe orienté, poids positifs	$O(n^2)$ (Dijkstra)[1]
Graphe orienté, poids quelconques, acircuitique	$O(m)$ (Programmation dynamique)
Graphe orienté, poids quelconques, sans circuit absorbant	$O(nm)$ (Programmation dynamique, algorithme de Ford-Bellman)[2]
Graphe orienté, poids quelconques	**NP**-difficile
Graphe non-orienté, poids positifs	$O(n^2)$ (Dijkstra)[1]
Graphe non-orienté, poids quelconques, pas de cycle absorbant	$O(n^3)$ (voir Chapitre 11)
Graphe non-orienté, poids quelconques	**NP**-difficile

3.4. Exercices

3.4.1. Plus court chemin. — Trouver la plus courte chaîne de s à t dans le graphe de la Figure 3.4.1 et le plus court chemin de s à t dans le graphe de la Figure 3.4.2.

3.4.2. Problème du goulot. — Soit N un réseau (de gaz par exemple) modélisé par un graphe orienté. Une origine o et une destination d sont données. Pour chaque arc, on dispose d'une capacité $c : A \to \mathbb{R}_+$. La *capacité* d'un chemin est le minimum des capacités de ses arcs. Proposer un algorithme efficace qui trouve la capacité maximale d'un chemin de o à d.

[1] On peut atteindre, avec les bonnes structures de données, $O(m + n \log n)$.
[2] On peut atteindre avec des techniques différentes $O(n^{1/2} m \log C)$.

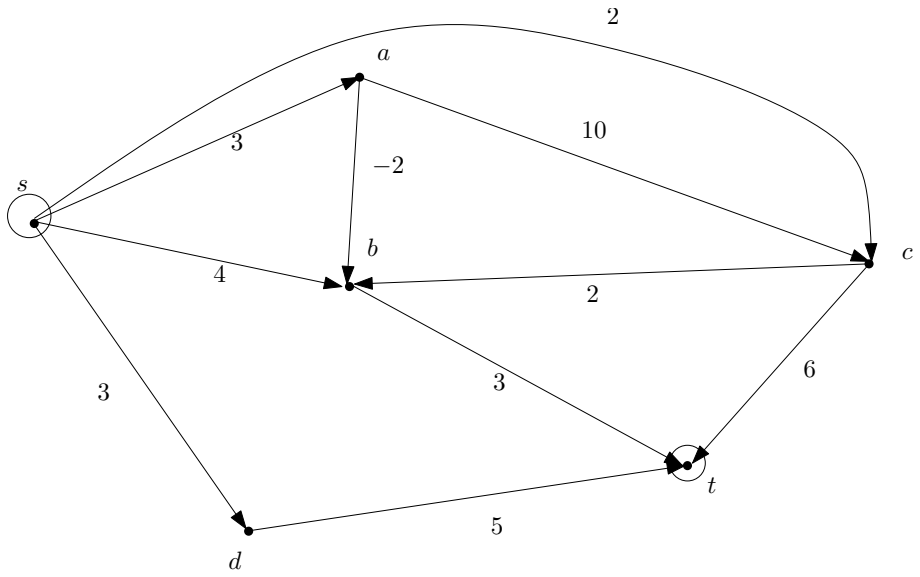


FIG. 5. On cherche le plus court chemin de s à t.

3.4.3. Gestion de stock. — Un stock suit une dynamique $x_{k+1} = x_k - d_k + u_k$, avec $x_1 = 2$ et

k	1	2	3
d_k	3	2	4

La quantité d_k est la demande pour la période k (supposée connue). $x_k \in \mathbb{N}$ est nombre d'unités disponibles en début de période k. u_k est le nombre d'unités commandées (et reçues immédiatement) en début de période k . On a une capacité maximale de stockage de 6, i.e que l'on doit toujours avoir $x_k + u_k \leq 6$ (ou de manière équivalente $x_{k+1} + d_k \leq 6$).

Le coût de gestion de stock pour la période k se décompose en un coût d'approvisionnement et un coût de stockage et s'écrit $4u_k + 3 + x_{k+1}$ si $u_k \neq 0$ et x_{k+1} sinon.

Trouver la stratégie qui minimise le coût total de la gestion du stock.

3.4.4. Problème du sac-à-dos. — Un bâteau peut transporter une cargaison formée à partir de N containers. Le poids total ne doit pas dépasser W. Le $k$ième container pèse w_k et son transport rapporte c_k. On veut déterminer un chargement qui maximise le bénéfice. Montrer que ce problème est un problème de sac-à-dos. Le résoudre sur les données $W = 6$ et

	Containers		
	1	2	3
Bénéfice c_k	9	2	8
Poids w_k	5	3	2

Même question avec le tableau suivant, $W = 13$ et avec la possibilité de prendre autant de container de type k que l'on veut.

	Containers		
	1	2	3
Bénéfice c_k	12	25	50
Poids w_k	4	5,5	6,5

3.4.5. Remplacement de machine. — Une entreprise vient de faire l'acquisition d'une machine neuve, dont elle a besoin pour assurer son bon fonctionnement pour les cinq années à venir. Une telle machine a une durée de vie de trois ans. A la fin de chaque année, l'entreprise peut

décider de vendre sa machine et d'en racheter une neuve pour 100 000 euros. Le bénéfice annuel assuré par la machine est noté b_k, son coût de maintenance c_k et son prix de revente p_k. Ces quantités dépendent de l'âge k de la machine et sont indiquées dans le tableau ci-dessous.

	Age k de la machine en début d'année		
k	0	1	2
b_k	80 000 euros	60 000 euros	30 000 euros
c_k	10 000 euros	14 000 euros	22 000 euros
p_k	50 000 euros	24 000 euros	10 000 euros

Proposer la stratégie de remplacement de la machine maximisant le profit de l'entreprise, sachant qu'à la fin de ces cinq années elle n'aura plus besoin de la machine.

3.4.6. Problème d'investissement - d'après Hêche, Liebling et de Werra [6]. — Une entreprise dispose d'un budget de 500 000 euros pour l'amélioration de ses usines pendant l'année à venir. L'entreprise possède trois sites de production dont les bénéfices annuels espérés, en fonction de la somme investie pour leur amélioration, sont données dans le tableau suivant.

	Montants investis (en centaines de milliers d'euros)					
Sites	0	1	2	3	4	5
	Bénéfices annuels					
1	1	3	4	5	5,5	6
2	0,5	2,5	4	5	6	6,5
3	2	4	5,5	6,5	7	7,5

3.4.7. Un problème d'intérimaires. — Une entreprise a une tâche à réaliser, qui va prendre 5 mois. Pour chaque mois, elle a besoin des nombres suivants d'intérimaires en plus de ses salariés.

Mois	Nombre d'intérimaires
1	10
2	7
3	9
4	8
5	11

Embaucher un intérimaire coûte 800 euros (embauche + formation). La fin d'une embauche coûte 1200 euros. Enfin, le coût mensuel d'un des ces intérimaires est de 1600 euros (salaire + charges). Trouver la stratégie de recrutement qui minimise les coûts.

3.4.8. Livraisons avec fenêtres de temps : cas linéaire. — Supposons qu'un camion ait à livrer des points $x_1, x_2, \ldots, x_n$, tous situés sur une droite. Pour chaque point x_i, on connaît l'instant le plus tard d_i où la livraison peut se faire. Le camion se situe au départ en un point x_0, et veut terminer sa tournée au plus tôt. Le camion n'a pas à retourner à son point de départ. De plus, on suppose que la livraison en chaque point se fait de manière instantanée, et que le temps pour aller de x_i à x_j est proportionnel à la distance les séparant. Montrer que l'on peut trouver la tournée optimale en $O(n^2)$ par un algorithme de type programmation dynamique (c'est un résultat de Tsitsiklis).

3.4.9. Biologie : plus long sous-mot commun. — Un *mot* w sur un alphabet Σ est une suite finie $w_1 w_2 \ldots w_n$ d'élément de Σ. La quantité n est appelé *longueur* du mot w. On dit qu'un mot $w' = w'_1, \ldots w'_{n'}$ de longueur n' est un *sous-mot* de w s'il existe une application f strictement

croissante de $\{1, 2, \ldots, n'\}$ dans $\{1, 2, \ldots, n\}$ telle que $w'_i = w_{f(i)}$ pour tout $i \in \{1, 2, \ldots, n'\}$. En d'autres termes, en effaçant des lettres de w, on peut obtenir w'. Par exemple : ABC est un sous-mot de $AEBBAC$, mais n'est pas un sous-mot de $AECAB$.

On se donne deux mots w_1 et w_2 sur un alphabet Σ. On cherche leur sous-mot commun w' le plus long. Montrer que c'est un problème de type programmation dynamique, et qu'il peut se résoudre en $O(n_1 n_2)$, où n_1 et n_2 sont les longueurs de w_1 et de w_2.

Ce problème a des applications en biologie, lorsque on veut évaluer la "distance" séparant deux brins d'ADN.

3.4.10. Plus courte chaîne élementaire. — Soit $G = (V, E)$ un graphe avec une fonction de poids $w : E \to \mathbb{R}$ et deux sommets particuliers s et t. Montrer que trouver la plus courte chaîne élémentaire (la chaîne élémentaire de plus petit poids) est **NP**-difficile.

3.4.11. Intervalles disjoints et locations rentables. — On suppose donnée une collection finie $\mathcal{C}$ d'intervalles fermés bornés de la droite réelle. On dispose d'une application poids $w : \mathcal{C} \to \mathbb{R}_+$.

1. Montrer que l'on sait trouver le sous-ensemble d'intervalles de $\mathcal{C}$ deux à deux disjoints de poids maximal en temps polynomial (indication : construire un graphe orienté acircuitique pondéré).

2. Application : Vous êtes chargé de la location d'un appartement. Vous connaissez un ensemble de demandes, chacune d'entre elles étant caractérisée par un instant de début, un instant de fin et le gain qu'apporterait la satisfaction de cette demande. Montrez que grâce à la question 1., on sait proposer une stratégie rapide maximisant le gain total de la location de l'appartement.

CHAPITRE 4

PROGRAMMATION LINÉAIRE

4.1. Définition et exemples

Un programme linéaire est un programme qui peut s'écrire sous la forme (*forme inéquationnelle*)

$$\begin{array}{ll} \text{Min} & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.c.} & A\boldsymbol{x} \leq \boldsymbol{b}, \end{array} \tag{5}$$

où $\boldsymbol{x}, \boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$ et où A est une $m \times n$ matrice réelle.

La programmation linéaire est peut-être le cas particulier de programmation mathématique, i.e. d'optimisation d'une fonction sous contrainte, le plus fréquemment rencontré dans l'industrie. Beaucoup de problèmes se modélisent comme des programmes linéaires, bien plus de problèmes encore font appel à la programmation linéaire pour résoudre des sous-problèmes qui mis bout-à-bout permettent la résolution du problème de départ.

Exemple. — Commençons par un exemple de problème de production dans l'agro-alimentaire[(1)]. Supposons que l'on dispose d'une grande surface cultivable sur laquelle il est possible de faire pousser des navets ou des courgettes. Le coût des semences est considéré comme négligeable. On dispose de deux types d'engrais X et Y, ainsi que d'un anti-parasite AP. Le besoin en engrais et en anti-parasite pour les courgettes et pour les navets est synthétisé dans le tableau suivant.

	Engrais X	Engrais Y	Anti-parasite AP
Courgettes	$2\,\text{l}\,\text{m}^{-2}$	$1\,\text{l}\,\text{m}^{-2}$	$0\,\text{l}\,\text{m}^{-2}$
Navets	$1\,\text{l}\,\text{m}^{-2}$	$2\,\text{l}\,\text{m}^{-2}$	$1\,\text{l}\,\text{m}^{-2}$

On dispose comme ressource de 8 l d'engrais X, de 7 l d'engrais Y et de 3 l d'anti-parasite AP.

On peut s'attendre à une productivité de $4\,\text{kg}.\text{m}^{-2}$ pour les courgettes et de $5\,\text{kg}.\text{m}^{-2}$ pour les navets, et à un gain de 1 €/kg tant pour les courgettes que pour les navets.

Quel est le gain maximum qui peut être fait compte tenu des ressources disponibles ?

On modélise ce problème comme un programme linéaire.

Variables de décision : x : surface de courgettes.
y : surface de navets.

Fonction objectif : Max $4x + 5y$

(1) Source : N. Brauner.

Contraintes	$2x + y \leq 8$	(ressources d'engrais A)
	$x + 2y \leq 7$	(ressources d'engrais B)
	$y \leq 3$	(ressources d'anti-parasite AB)
	$x \geq 0$ et $y \geq 0$.	

On veut donc résoudre le programme suivant.

$$\begin{array}{ll} \text{Max} & 4x + 5y \\ \text{s.c.} & 2x + y \leq 8, \\ & x + 2y \leq 7, \\ & y \leq 3 \\ & x \geq 0, y \geq 0 \end{array}$$

Ce programme est bien un programme linéaire comme défini par l'équation (5), avec

$$\boldsymbol{c} = \begin{pmatrix} 4 \\ 5 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 8 \\ 7 \\ 3 \\ 0 \\ 0 \end{pmatrix} \quad A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Comme nous l'avons déjà noté lors du premier cours, le fait qu'il y ait Max dans notre problème au lieu d'un Min comme dans l'équation (5) ne pose pas de problème : maximiser une quantité revient à minimiser son opposé.

Comme on n'a que deux variables, on peut procéder à une représentation graphique (voir Figure 1).

Problème de production. — De manière générale, le problème de production est un programme linéaire

$$\begin{array}{lll} \text{Max} & \sum_{j=1}^{n} c_j x_j & \\ \text{s.c.} & \sum_{j=1}^{n} a_{i,j} x_j \leq b_i, & \text{pour } i = 1, \ldots, n, \\ & x_j \geq 0, & \text{pour } j = 1, \ldots, m. \end{array} \tag{6}$$

qui s'interprète de la manière suivante :

- n types de produits,
- m types de ressources disponibles,
- x_j : quantité de produit j,
- c_j : profit unitaire du le produit j,
- $a_{i,j}$: consommation en ressource i pour produire une unité de j,
- b_i : quantité de ressource i en stock.

C'est un exemple classique et très fréquent de la programmation linéaire. L'aspect « allocation optimale des ressources » de la RO est particulièrement bien illustré par le problème de production.

Dans l'exemple des navets et des courgettes, nous avons pu résoudre le problème assez facilement car il n'y avait que 2 variables, et donc une représentation graphique était possible. Mais la plupart des problèmes réels – un problème de production par exemple – le nombre de variables et le nombre de contraintes peuvent dépasser le millier, et ni l'intuition, ni le dessin peuvent alors nous tirer d'affaire. Heureusement, depuis la fin des années 40, de nombreux chercheurs ont travaillé sur la programmation linéaire, et c'est maintenant un problème qui est bien résolu tant sur le plan pratique que sur le plan théorique.

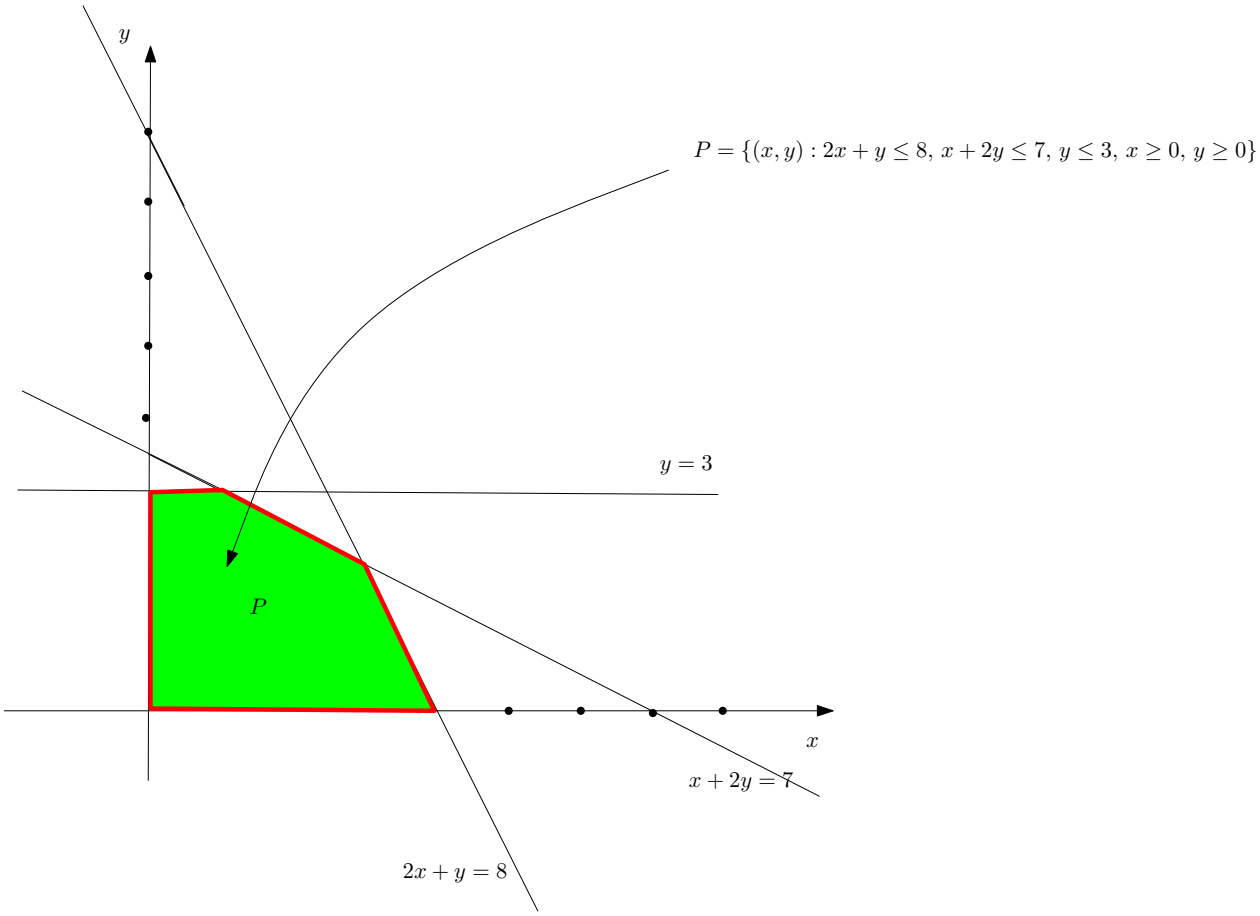


FIG. 1. Représentation graphique des contraintes du problème des navets et des courgettes.

Plan pratique : De nombreux logiciels (libres et commerciaux) résolvent des programmes linéaires de grande taille. Voir la liste en annexe. Il y a principalement deux algorithmes qui sont utilisés dans ces codes : l'*algorithme du simplexe* et l'*algorithme des points intérieurs*. On devrait plutôt parler de familles d'algorithmes car tant l'algorithme du simplexe que l'algorithme des points intérieurs existent sous de nombreuses variantes.

Plan théorique : La programmation linéaire est dans **P**. Cela signifie qu'il existe des algorithmes polynomiaux qui résolvent la programmation linéaire. L'algorithme des points intérieurs est un de ceux-là. L'algorithme du simplexe, bien que très rapide en pratique, ne possède pas pour le moment de version polynomiale : pour chacune de ses versions, on connaît des instances qui nécessitent un nombre exponentiel d'étapes.

Dans ce cours, nous allons donner quelques propriétés théoriques de la programmation linéaire (PL), qui ont un grand intérêt pratique (polyèdres, dualité,...). Nous verrons ensuite les différents algorithmes qui résolvent la programmation linéaire. Enfin, nous parlerons d'un sujet important, la programmation linéaire en nombres entiers (PLNE), qui demande que tout ou partie des variables soient entières. Dans de nombreuses applications, les quantités produites sont nécessairement entières (production de voitures, de chaînes Hi-fi), ce qui implique que les variables x_j dans l'équation (6) doivent en plus être des éléments de $\mathbb{N}$. En fait, la plupart des problèmes d'optimisation discrète peuvent se mettre sous cette forme, ce qui implique entre autre que la PLNE est **NP**-difficile.

4.2. Quelques éléments théoriques

4.2.1. Forme standard. — L'algorithme du simplexe et l'algorithme des points intérieurs utilisent une écriture du programme linéaire différente de celle de l'équation (5). Ils utilisent la *forme standard*[(2)] qui consiste à écrire le programme linéaire de la façon suivante :

$$\begin{array}{ll} \text{Min} & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.c.} & A\boldsymbol{x} = \boldsymbol{b}, \\ & \boldsymbol{x} \geq \mathbf{0}. \end{array} \tag{8}$$

Comme d'habitude, $\boldsymbol{x}$ est le vecteur des variables, A est une matrice $m \times n$ donnée, $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$ sont deux vecteurs donnés et $\mathbf{0}$ est le vecteur 0, avec n composantes.

Passage de la forme standard à la forme inéquationnelle et réciproquement. — Pour passer de la forme standard à la forme inéquationnelle, on pose $A' := \begin{pmatrix} A \\ -A \\ -I_n \end{pmatrix}$, $\boldsymbol{b}' := \begin{pmatrix} \boldsymbol{b} \\ -\boldsymbol{b} \\ \mathbf{0} \end{pmatrix}$, $\boldsymbol{c}' := \boldsymbol{c}$. Il est alors clair que les solutions réalisables et les solutions optimales de (8) et du programme linéaire sous forme inéquationnelle :

$$\begin{array}{ll} \text{Min} & \boldsymbol{c}'^T\boldsymbol{x} \\ \text{s.c.} & A'\boldsymbol{x} \leq \boldsymbol{b}', \end{array}$$

sont identiques.

La transformation réciproque est un peu plus subtile. On a alors le programme linéaire sous forme standard suivant :

$$\begin{array}{ll} \text{Min} & \boldsymbol{c}^T\boldsymbol{x}' \\ \text{s.c.} & A'\boldsymbol{x} = \boldsymbol{b}', \boldsymbol{x}' \geq \mathbf{0}, \end{array} \tag{9}$$

avec $A' := \begin{pmatrix} A & -A & I_m \end{pmatrix}$, $\boldsymbol{b}' := \boldsymbol{b}$ et $\boldsymbol{c}' := (\boldsymbol{c}, -\boldsymbol{c}, \mathbf{0})$. Il faut montrer que toute solution optimale de l'équation (5) donne une solution optimale de ce programme et réciproquement. Pour ce faire, nous allons montrer que toute solution réalisable de l'équation (5) donne une solution réalisable de même coût pour ce programme.

Prenons une solution réalisable $\boldsymbol{x}$ du programme (5). On construit les variables $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ et $\boldsymbol{y} \in \mathbb{R}^m$ de la manière suivante

$$y_i := b_i - (A\boldsymbol{x})_i \quad \text{pour } j = 1, \ldots, m$$

et

$$u_j := \begin{cases} x_j & \text{si } x_j \geq 0 \\ 0 & \text{sinon,} \end{cases} \quad \text{et } v_j := \begin{cases} x_j & \text{si } x_j \leq 0 \\ 0 & \text{sinon,} \end{cases} \quad \text{pour } j = 1, \ldots, n.$$

Il est alors aisé de vérifier que $\boldsymbol{x}' := (\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{y})$ est une solution réalisable du programme (9), et que l'on a bien $\boldsymbol{c}^T\boldsymbol{x} = \boldsymbol{c}^T\boldsymbol{u} - \boldsymbol{c}^T\boldsymbol{v}$.

Réciproquement, soient $\boldsymbol{x}' = (\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{y})$ une solution réalisable du programme (9). Posons $\boldsymbol{x} := \boldsymbol{u} - \boldsymbol{v}$. Le vecteur $\boldsymbol{x}$ est alors une solution réalisable du programme (8), et l'on a bien $\boldsymbol{c}^T\boldsymbol{x} = \boldsymbol{c}^T\boldsymbol{u} - \boldsymbol{c}^T\boldsymbol{v}$.

---

(2) En fait, on distingue classiquement deux formes pour la programmation linéaire : la *forme standard*, décrite ici, et la *forme canonique* qui consiste à écrire le programme linéaire de la façon suivante :

$$\begin{array}{ll} \text{Min} & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.c.} & A\boldsymbol{x} \leq \boldsymbol{b}, \\ & \boldsymbol{x} \geq \mathbf{0}. \end{array} \tag{7}$$

L'utilisation de la *forme inéquationnelle* ainsi que son appellation vient du livre **[22]**.

4.2.2. Préliminaires. — Dans la suite, on supposera sans perte de généralité, que la matrice A de la forme standard (8) est de plein rang, c'est-à-dire que les m lignes sont linéairement indépendantes. En effet, on peut tester l'existence d'une solution réalisable en résolvant l'équation $A\boldsymbol{x} = \boldsymbol{b}$ (par des pivots de Gauss par exemple). S'il y a une solution réalisable et si les lignes de A ne sont pas linéairement indépendantes, alors forcément l'équation correspondante est redondante et on peut la supprimer. D'où

Hypothèse : Nos programmes linéaires sous forme standard seront tels que A est de plein rang.

4.2.3. Solutions basiques réalisables. — On suppose toujours que A est une matrice $m \times n$, de plein rang. Pour introduire la notion de solution basique, nous introduisons la notation suivante : pour une partie $B \subseteq \{1, 2, \ldots, n\}$, on note A_B la sous-matrice de A réduite aux colonnes indicées par les éléments de B.

Par exemple, si

$$A = \begin{pmatrix} 1 & 2 & 0 & 3 & 3 & 1 \\ 0 & 8 & 9 & 3 & 4 & 4 \\ 5 & 6 & 0 & 7 & 1 & 1 \end{pmatrix},$$

et si $B = \{1, 4, 5\}$ alors

$$A_B = \begin{pmatrix} 1 & 3 & 3 \\ 0 & 3 & 4 \\ 5 & 7 & 1 \end{pmatrix}.$$

Une notation semblable sera utilisée pour les vecteurs : si $\boldsymbol{x} = (5, 4, 5, 1, 1, 2)$ et $B = \{1, 4, 5\}$, alors $\boldsymbol{x}_B = (5, 1, 1)$.

Remarquons que pour une partie B fixée, on peut réécrire l'égalité $A\boldsymbol{x} = \boldsymbol{b}$ sous la forme

$$A_B \boldsymbol{x}_B + A_N \boldsymbol{x}_N = \boldsymbol{b} \tag{10}$$

Une partie $B \in \{1, 2, \ldots, n\}$ à m éléments est appelée *base* si la matrice A_B est inversible. Dans ce cas, on peut *réécrire le système $A\boldsymbol{x} = \boldsymbol{b}$ dans la base B* en transformant l'écriture (10) en

$$\boldsymbol{x}_B + A_B^{-1} A_N \boldsymbol{x}_N = A_B^{-1} \boldsymbol{b}.$$

La solution $\tilde{\boldsymbol{x}}$ du système $A\boldsymbol{x} = \boldsymbol{b}$ obtenue en posant $\tilde{\boldsymbol{x}}_B := A_B^{-1}\boldsymbol{b}$ et $\tilde{\boldsymbol{x}}_N := 0$ est une *solution basique*. Si cette solution basique est réalisable (par rapport au programme linéaire (8)), i.e. si $A_B^{-1}\boldsymbol{b} \geq 0$, alors on dit que $\tilde{\boldsymbol{x}}$ est une *solution basique réalisable*, et que B est une base réalisable.

On a alors le théorème important suivant qui constitue la base de l'algorithme du simplexe.

Théorème 4.2.1. — *Considérons un programme linéaire sous forme standard (forme (8)). Alors*

- *si l'ensemble des solutions réalisables est non-vide et si la fonction objectif est bornée inférieurement sur cet ensemble, alors il existe une solution optimale.*
- *s'il existe une solution optimale, alors il existe une solution optimale qui soit basique réalisable.*

Ce théorème conduit à un algorithme naïf pour résoudre un programme linéaire : comme il n'y a qu'un nombre fini de bases (au plus $\binom{n}{m}$), on essaye chaque base l'une après l'autre et on garde celle réalisable qui fournit la plus petite valeur pour $\boldsymbol{c}^T\boldsymbol{x}$. Il faudrait encore vérifier que le fonction objectif est bornée inférieurement, mais dans tous les cas, un tel algorithme n'a aucun intérêt pratique. En effet, le nombre d'étapes est exponentiel[(3)].

[(3)] Par exemple, pour $n = 2m$, on a $\binom{n}{m} \simeq 4^m$.

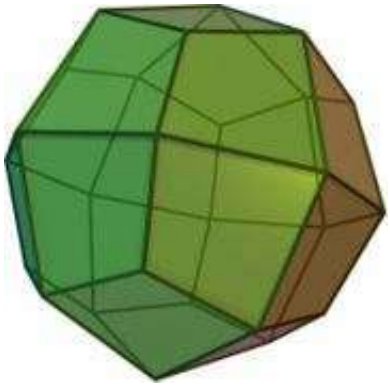


FIG. 2. Un polyèdre.

L'algorithme du simplexe va également énumérer des bases réalisables, mais dans un ordre intelligent, ce qui va réduire considérablement (en général) le nombre de bases réalisables à considérer. Pour décrire cet ordre (voir ci-dessous), on a besoin de la notion de *base voisine*. B' est une base voisine de la base B si c'est une base telle que $B' \setminus B$ ne contient qu'un élément (et puisque les bases ont toute même cardinalité, elle est également telle que $B \setminus B'$ ne contient qu'un élément).

Il se peut aussi que l'ensemble des solutions réalisables soit non-vide, mais que la fonction objectif ne soit pas bornée inférieurement sur cet ensemble. Dans ce cas, on a un *rayon infini*.

Théorème 4.2.2. — *Considérons un programme linéaire sous forme standard (forme (8)). Si l'ensemble des solutions réalisables est non-vide et si la fonction objectif n'est pas bornée inférieurement sur cet ensemble, alors il existe* $\boldsymbol{x}_0$ *et* $\boldsymbol{q}$ *tels que les points* $\boldsymbol{x}(t) = \boldsymbol{x}_0 - t\boldsymbol{q}$ *soient tous réalisables pour* $t \geq 0$*, et tels que* $\lim_{t\to+\infty} \boldsymbol{c}^T\boldsymbol{x}(t) = -\infty$.

L'ensemble des points $\boldsymbol{x}(t) = \boldsymbol{x}_0 - t\boldsymbol{q}$ pour $t \geq 0$ est appelé *rayon infini*. La variable t est le *paramètre* de ce rayon, $\boldsymbol{x}_0$ son *origine* et $\boldsymbol{q}$ sa *direction*.

4.2.4. Polyèdres. — On peut également essayer de « voir » ce que signifie le programme (8), un peu comme on l'a fait dans la résolution de l'exemple introductif des navets et des courgettes. Les contraintes $A\boldsymbol{x} = \boldsymbol{b}$ et $\boldsymbol{x} \geq \boldsymbol{0}$ délimitent ce qu'on appelle un *polyèdre*, c'est-à-dire une partie de $\mathbb{R}^n$ obtenue comme intersection d'un nombre fini de demi-espaces fermés délimités par des hyperplans.

On vérifie facilement qu'un polyèdre est *convexe*, c'est-à-dire que pour toute paire de points $\boldsymbol{x}$ et $\boldsymbol{y}$ d'un polyèdre P, l'ensemble des points du segment $[\boldsymbol{x}, \boldsymbol{y}]$ est inclus dans P.

Un polyèdre possède en général des *points extrêmes* ou *sommets*, qui sont les "pointes", ou les "piques" du polyèdre (voir Figure 2). Pour éviter la confusion avec les sommets des graphes, nous utiliserons plutôt l'appellation "points extrêmes" dans ce cours. On définit mathématiquement un point extrême de la manière suivante. $\boldsymbol{v} \in P$ est un point extrême de P si pour tout $\boldsymbol{x}, \boldsymbol{y} \in P$, on a l'implication

$$(\text{le segment } [\boldsymbol{x}, \boldsymbol{y}] \text{ contient } \boldsymbol{v}) \Rightarrow (\boldsymbol{v} = \boldsymbol{x} \text{ ou } v = \boldsymbol{y}).$$

Autrement dit, on ne peut avoir un segment inclus dans P et non réduit à un point qui contienne $\boldsymbol{v}$.

En réalité, comme le montre le théorème suivant, les points extrêmes et les solutions basiques réalisables recouvrent le même concept.

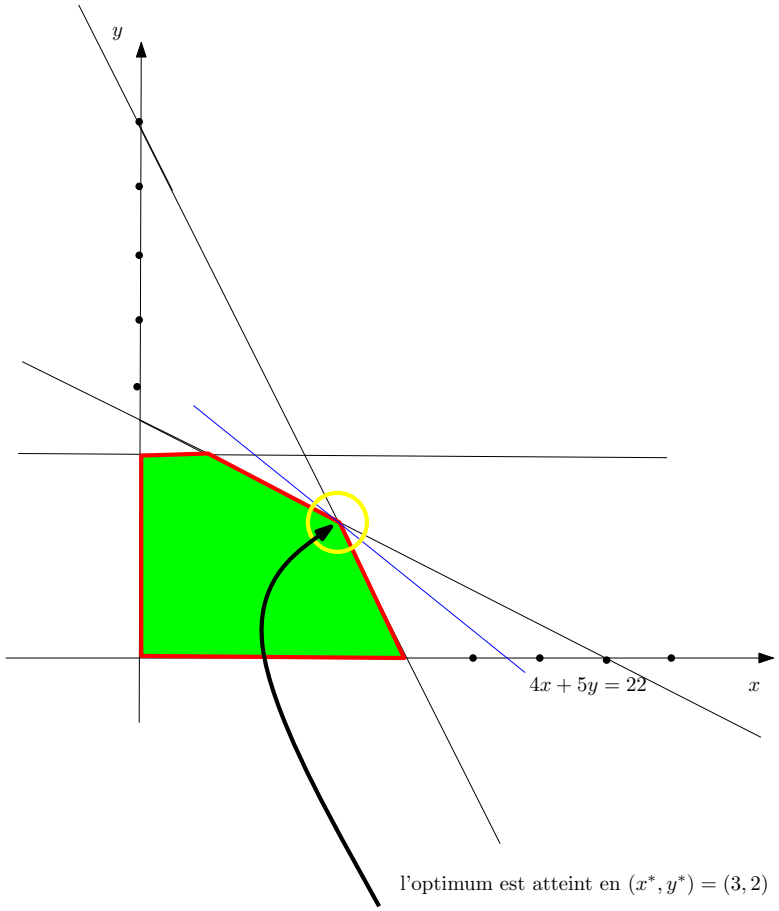


FIG. 3. La solution optimale au problème des navets et des courgettes est un sommet du polyèdre.

Théorème 4.2.3. — *Soit P l'ensemble des solutions réalisables du programme linéaire (8) (P est donc un polyèdre). Soit $\boldsymbol{v} \in P$. Les deux assertions sont équivalentes :*

- *$\boldsymbol{v}$ est une solution basique réalisable de (8).*
- *$\boldsymbol{v}$ est un point extrême de P.*

On peut voir une illustration sur la Figure 3, qui montre la solution optimale au problème des navets et des courgettes du début du chapitre.

4.3. Algorithmes

Comme il a déjà été indiqué, il y a trois (familles d') algorithmes :

- le simplexe.
- les ellipsoïdes.
- les points intérieurs.

Nous allons décrire brièvement l'idée de l'algorithme du simplexe et encore plus brièvement celle des points intérieurs. Il fut un temps où les cours de recherche opérationnelle consistaient en grande partie à décrire et pratiquer l'algorithme du simplexe. Nous considérons que l'intérêt d'un tel enseignement pour un ingénieur ou un chercheur est assez limité et la compréhension des grandes idées largement suffisante.

Pour la description exacte de ces algorithmes, il existe de nombreux ouvrages. Un excellent ouvrage qui couvre l'ensemble de la programmation linéaire et de ces algorithmes est celui de Matousek et Gärtner [**22**]. Sur l'algorithme du simplexe, le meilleur livre est certainement celui de Chvátal [**4**]. Sur l'algorithme des ellipsoïdes, le livre de Grötchel, Lovász et Schrijver est une excellente référence et un grand classique [**14**]. Enfin, sur les points intérieurs, l'article de Terlaky est une bonne introduction aux points intérieurs [**25**].

Sur le plan pratique, l'algorithme des ellipsoïdes n'a pas encore fait ses preuves, ce qui explique qu'il ne soit pas décrit dans ce cours. Son intérêt est (pour le moment ?) théorique et historique puisqu'il constitue le premier algorithme polynomial proposé pour la programmation linéaire (en 1979, par Khachyan). Avant 1979, la question, alors ouverte, de savoir s'il était possible de résoudre la programmation linéaire en temps polynomial passionnait beaucoup de chercheurs.

4.3.1. Le simplexe. — L'idée de l'algorithme du simplexe est de passer de base réalisable en base réalisable en améliorant à chaque fois la valeur du critère. Comme le nombre de bases est fini, on obtient l'optimum ou la preuve que le programme est non borné en un nombre fini d'étapes.

On suppose que l'on connaît au départ une base réalisable B. L'algorithme du simplexe commence par réécrire le programme (8) sous la forme équivalente

$$\begin{array}{ll} \text{Min} & \boldsymbol{c}_B^T A_B^{-1}\boldsymbol{b} + \left(\boldsymbol{c}_N - \boldsymbol{c}_B^T A_B^{-1} A_N\right)\boldsymbol{x}_N \\ \text{s.c.} & \boldsymbol{x}_B + A_B^{-1} A_N \boldsymbol{x}_N = A_B^{-1}\boldsymbol{b}, \\ & \boldsymbol{x} \geq \boldsymbol{0}. \end{array} \tag{11}$$

Il faut bien noter que ce programme et le programme (8) sont complètement équivalents (comme on l'a déjà vu précédemment). L'avantage de cette forme, c'est qu'on peut directement lire la valeur de la solution basique réalisable associée à B : c'est le terme constant $\boldsymbol{c}_B^T A_B^{-1}\boldsymbol{b}$ de la fonction objectif. En effet, la solution basique réalisable associée à B est telle que $\boldsymbol{x}_N = 0$. On peut également lire cette solution basique réalisable : c'est $\tilde{x}_B = A_B^{-1}\boldsymbol{b}$ et $\tilde{x}_N = 0$.

Ensuite, l'algorithme cherche une base réalisable voisine B' qui améliore le critère (c'est l'opération de *pivot*). Il est en effet assez simple de construire une telle base B' à partir de B, simplement à la lecture des entrées de $\boldsymbol{c}_N - \boldsymbol{c}_B^T A_B^{-1} A_N$ et de $A_B^{-1} A_N$. On sélectionne d'abord l'élément $j \notin B$ qui va entrer dans B' (*la variable entrante* x_j), à la lecture des coefficients de $\boldsymbol{c}_N - \boldsymbol{c}_B^T A_B^{-1} A_N$. Ensuite, à la lecture des coefficients de la colonne correspondante de $A_B^{-1} A_N$, on trouve l'élément i qui va quitter B (*la variable sortante* x_i).

Variable entrante : Tout coefficient négatif de $\boldsymbol{c}_N^T - \boldsymbol{c}_B^T A_B^{-1} A_N$ peut faire office de variable entrante.

Variable sortante : Soit $\boldsymbol{q}$ la colonne de $A_B^{-1} A_N$ correspondant à la variable entrante et soit $\boldsymbol{p} := A_B^{-1}\boldsymbol{b}$. On regarde les coefficients strictement positifs de $\boldsymbol{q}$ et parmi ceux-là, on choisit la variable sortante x_i telle que p_i/q_i soit le plus petit possible.

On réécrit ensuite le programme sous la forme équivalente

$$\begin{array}{ll} \text{Min} & \boldsymbol{c}_{B'}^T A_{B'}^{-1}\boldsymbol{b} + \left(\boldsymbol{c}_{N'}^T - \boldsymbol{c}_{B'}^T A_{B'}^{-1} A_{N'}\right)\boldsymbol{x}_{N'} \\ \text{s.c.} & \boldsymbol{x}_{B'} + A_{B'}^{-1} A_{N'} \boldsymbol{x}_{N'} = A_{B'}^{-1}\boldsymbol{b}, \\ & \boldsymbol{x} \geq \boldsymbol{0}. \end{array} \tag{12}$$

pour laquelle on est à nouveau capable de lire la valeur du critère pour la solution basique réalisable associée à B' (c'est $\boldsymbol{c}_{B'}^T A_{B'}^{-1}\boldsymbol{b}$) et la solution basique réalisable elle-même ($\tilde{x}_{B'} = A_{B'}^{-1}\boldsymbol{b}$ et $\tilde{x}_{N'} = 0$).

Et on recommence.

L'optimalité est atteinte quand tous les coefficients de $\boldsymbol{c}_{N'}^T - \boldsymbol{c}_{B'}^T A_{B'}^{-1} A_{N'}$ sont ≥ 0. En effet, quelque soit alors le choix de $\boldsymbol{x}_{N'}$, qui est à coefficients positifs ou nuls, le critère verra sa valeur augmenter.

De deux choses l'une, soit on termine sur une base optimale, soit à la lecture des coefficients, on voit que le programme est non bornée (i.e $-\infty$ est la valeur optimale) : cela arrive lorsque toutes les composantes de la colonne $\boldsymbol{q}$ correspondante à la variable entrante sont négatives ou nulles, on a alors un rayon infini d'origine $\tilde{x}_{B'}$ et de direction $\boldsymbol{q}$.

On a supposé que l'on disposait d'une base réalisable de départ. C'est en effet souvent le cas. Par exemple, si le programme est sous forme canonique, l'ajout des variables d'écart fait apparaître une sous-matrice identité, dont les colonnes sont une base réalisable naturelle de départ. Sinon, on dispose de technique standard pour construire une telle base réalisable (en général, on résout un autre programme linéaire dont l'optimum est atteint sur une base réalisable du programme qui nous intéresse - voir l'exercice sur cette question en fin de chapitre). Il se peut aussi bien entendu que notre programme soit tout simplement non réalisable.

Interprétation géométrique : L'algorithme du simplexe s'interprète facilement géométriquement. En effet, deux bases voisines fournissent des solutions basiques réalisables qui sont des sommets confondus ou voisins (au sens des arêtes du polyèdre) du polyèdre des solutions réalisables. L'opération de pivot consiste à "glisser" le long d'une arête du polyèdre[(4)] pour aller vers un point extrême voisin, de meilleur valeur par la fonction objectif.

Par conséquent, la suite des solutions basiques réalisables et des pivots générés par l'algorithme du simplexe fournit une trajectoire qui passe par des sommets et des arêtes du polyèdre et qui cherche à se diriger dans le sens de $\boldsymbol{c}$.

Efficacité de l'algorithme du simplexe : On considère qu'en général, pour un programme à m contraintes, l'algorithme du simplexe va faire entre $2m$ et $3m$ opérations de pivots avant d'arriver à l'optimum. La réécriture du système lors d'un pivot nécéssite $O(m^2)$ opérations. On a donc en général $O(m^3)$ opérations pour résoudre un programme linéaire par l'algorithme du simplexe, ce qui est tout à fait raisonnable. Sur des instance réelles, le simplexe fonctionne bien.

Cela dit, il existe des instances pour lesquelles le nombre de pivots est exponentiel. La question de savoir s'il existe une règle de pivot conduisant à un comportement polynomial est une (importante) question ouverte.

4.3.2. Les points intérieurs. — A l'opposé de l'algorithme du simplexe, l'algorithme des points intérieurs cherche à éviter le bord du polyèdre et rester le plus possible à l'intérieur. C'est seulement à la dernière étape que l'algorithme atteint le bord, précisément sur une solution optimale.

Concrètement, cela consiste à considérer le coût pénalisé

$$f_\mu(\boldsymbol{x}) := \boldsymbol{c}^T \boldsymbol{x} - \mu \dot{\sum_{j=1}^{n}} \log x_j$$

et à remplacer le programme (8) par le programme suivant

$$\begin{array}{ll} \text{Min} & f_\mu(\boldsymbol{x}) \\ \text{s.c.} & A\boldsymbol{x} = \boldsymbol{b}, \\ & \boldsymbol{x} > \boldsymbol{0}, \end{array} \tag{13}$$

où $\boldsymbol{x} > \boldsymbol{0}$ signifie que $x_j > 0$ pour $j = 1, \ldots, n$.

(4) ou à rester sur le même sommet ; un pivot ne parvient pas toujours à améliorer directement la valeur du critère.

L'idée de l'algorithme est de faire tendre progressivement μ vers 0, et de chercher la solution optimale $\boldsymbol{x}^*(\mu)$ du programme (13) pour chaque valeur de μ successive. La suite des $\boldsymbol{x}^*(\mu)$ tend alors vers l'optimum du programme (8) (à quelques conditions techniques près). La recherche de la solution optimale $\boldsymbol{x}^*(\mu)$ du programme (13) se fait en fait approximativement et utilise la solution optimale approximative du programme pour la valeur de μ précédente – cette étape revient alors à une simple résolution d'un système linéaire.

Efficacité de l'algorithme des points intérieurs : La complexité de l'algorithme est $O(n^3L)$ où L est le nombre maximum de bits nécessaires pour coder un coefficient du programme, ce qui en fait bien un algorithme polynomial. Le nombre d'itérations (nombre de μ distincts pour lequel $\boldsymbol{x}^*(\mu)$ est approximativement calculé) est $O(\sqrt{n}L)$ mais en pratique cela semble plutôt être un $O(\log n)$, ce qui en fait un algorithme réellement efficace.

4.4. Dualité

4.4.1. Dualité faible, dualité forte. — Nous abordons maintenant la propriété théorique fondamentale de la programmation linéaire, dont les applications sont innombrables. Il s'agit de la propriété de dualité.

En suivant la méthode présentée en Sous-Section 2.3.5, à tout programme linéaire, on peut associer un autre programme linéaire, son dual. Le premier programme est le programme primal. Lors de la transformation, les variables du primal deviennent les contraintes du dual et les contraintes du primal deviennent les variables du dual.

Considérons par exemple, le programme linéaire suivant, sous forme standard :

$$\begin{array}{ll} \text{Min} & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.c.} & A\boldsymbol{x} = \boldsymbol{b}, \\ & \boldsymbol{x} \geq \boldsymbol{0}. \end{array} \tag{14}$$

Son dual se définit alors

$$\begin{array}{ll} \text{Max} & \boldsymbol{b}^T\boldsymbol{y} \\ \text{s.c.} & A^T\boldsymbol{y} \leq \boldsymbol{c}. \end{array} \tag{15}$$

Cela se vérifie aisément en suivant la méthode présentée en 2.3.5.

Noter que chaque y_i multiplie la $i$ème ligne (la contrainte) de A, pour $i = 1, \ldots, m$, et que chaque colonne de A, associée chacune à une des n variables du programme (14), devient une contrainte pour le programme (15).

Prenons une solution réalisable $\boldsymbol{x}$ du primal et une solution réalisable $\boldsymbol{y}$ du dual.

Multiplions à gauche les contraintes du primal par $\boldsymbol{y}$. On obtient

$$\boldsymbol{y}^T A\boldsymbol{x} = \boldsymbol{y}^T\boldsymbol{b}.$$

En utilisant les contraintes du dual et le fait que les composantes de $\boldsymbol{x}$ sont positives, on obtient $\boldsymbol{c}^T\boldsymbol{x} \geq \boldsymbol{y}^T\boldsymbol{b}$ ou plus lisiblement

$$\boldsymbol{c}^T\boldsymbol{x} \geq \boldsymbol{b}^T\boldsymbol{y}.$$

On retrouve la *dualité faible*. En particulier, la valeur optimale du dual minore la valeur optimale du primal (si optimum il y a). En réalité, on a beaucoup plus : *en général, pour la programmation linéaire, les valeurs optimales des programmes primal et dual coïncident*. Plus précisément :

***Théorème 4.4.1* (Théorème fort de la dualité pour la programmation linéaire)**
Pour les deux programmes linéaires suivants

$$\begin{array}{ll} \text{Min} & \boldsymbol{c}^T\boldsymbol{x} \\ \textit{s.c.} & A\boldsymbol{x} = \boldsymbol{b}, \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{array} \quad \text{(P)}$$

et

$$\begin{array}{ll} \text{Max} & \boldsymbol{b}^T\boldsymbol{y} \\ \textit{s.c.} & A^T\boldsymbol{y} \leq \boldsymbol{c}, \end{array} \quad \text{(D)}$$

une et une seule des possibilités suivantes se réalise

1. *Ni (P), ni (D) n'ont de solution réalisable.*
2. *(P) est non borné et (D) n'a pas de solution réalisable.*
3. *(D) est non borné et (P) n'a pas de solution réalisable.*
4. *(P) et (D) sont tous deux réalisables. Ils ont alors tous deux des solutions optimales, respectivement $\boldsymbol{x}^*$ et $\boldsymbol{y}^*$, et l'on a*
$$\boldsymbol{c}^T\boldsymbol{x}^* = \boldsymbol{b}^T\boldsymbol{y}^*.$$
Autrement dit, le minimum de *(P)* est égal au maximum de *(D)*.

La preuve est omise, mais ce théorème peut se prouver soit en utilisant un lemme classique en géométrie – le lemme de Farkas – soit en utilisant l'écriture de la base optimale dans l'algorithme du simplexe (voir exercice en fin de chapitre).

4.4.2. Dualité sous toutes ses formes. — Dans le tableau suivant, on donne toutes les situations possibles. Un bon exercice consiste à redémontrer ces programmes en utilisant la méthode de la Sous-Section 2.3.5.

	Programme linéaire primal		Programme linéaire dual	
Variables	$x_1, x_2, \ldots, x_n$		$y_1, y_2, \ldots, y_m$	
Matrice	A		A^T	
Membre de droite	b		c	
Fonction objectif	Min $c^T x$		Max $b^T y$	
Contraintes	$i$ème contrainte a	$\leq$	$y_i \leq 0$	
		$\geq$	$y_i \geq 0$	
		$=$	$y_i \in \mathbb{R}$	
	$x_j \leq 0$		$j$ème contrainte a	$\geq$
	$x_j \geq 0$			$\leq$
	$x_j \in \mathbb{R}$			$=$

Une propriété important de la dualité en programmation linéaire, qui se vérifie directement sur le tableau ci-dessus mais aussi à partir de la définition de la Sous-Section 2.3.5, est que

Proposition 4.4.2. — *Le dual du dual est équivalent au primal.*

On a une autre propriété importante, appelée *propriété des écarts complémentaires.*
Soit un programme sous forme canonique :

$$\begin{array}{lll} \text{Min} & \sum_{j=1}^n c_j x_j & \\ \text{s.c.} & \sum_{j=1}^n a_{ij} x_j \geq b_i & \text{pour tout } i \in \{1, \ldots, m\} \\ & x_j \geq 0 & \text{pour tout } j \in \{1, \ldots, n\} \end{array}$$

et son dual

$$\begin{array}{lll} \text{Max} & \sum_{i=1}^m b_i y_i & \\ \text{s.c.} & \sum_{i=1}^m a_{ij} y_i \leq c_j & \text{pour tout } j \in \{1,\ldots,n\} \\ & y_i \geq 0 & \text{pour tout } i \in \{1,\ldots,m\}. \end{array}$$

Alors

***Proposition 4.4.3* (Ecarts complémentaires).** — *Soient $\boldsymbol{x}^*$ et $\boldsymbol{y}^*$ des solutions optimales du primal et du dual (en supposant qu'elles existent). Alors pour tout $j \in \{1,\ldots,n\}$, si $x_j^* > 0$ alors $\sum_{i=1}^m a_{ij} y_i^* = c_j$.*

La preuve est laissée en exercice (voir fin de ce chapitre).

4.4.3. Interprétation économique de la dualité. — La dualité a des interprétations économiques intéressantes. Nous les illustrons sur deux types de programmes linéaires.

4.4.3.1. Le primal est un problème de maximisation. — Soit le problème de production pour une entreprise E :

$$\begin{array}{lll} \text{Max} & \sum_{j=1}^n c_j x_j & \\ \text{s.c.} & \sum_{j=1}^n a_{ij} x_j \leq b_i, & \text{pour } i = 1,\ldots,m, \\ & x_j \geq 0, & \text{pour } j = 1,\ldots,n. \end{array}$$

Soit une firme F qui veut racheter les ressources de l'entreprise. A quels prix doit-elle le faire pour tout recheter et maximiser son profit ?

En notant y_i le prix de la ressource i, on écrit

$$\begin{array}{lll} \text{Min} & \sum_{i=1}^m b_i y_i & \\ \text{s.c.} & \sum_{i=1}^m a_{ij} y_i \geq c_j, & \text{pour } j = 1,\ldots,n, \\ & y_i \geq 0, & \text{pour } i = 1,\ldots,m. \end{array}$$

La fonction objectif traduit le souci de F de maximiser son revenu. La contrainte $\sum_{i=1}^m a_{ij} y_i \geq c_j$ impose que l'entreprise E n'a pas intérêt à produire une unité de produit j, elle gagnera plus à vendre les ressources correspondantes à F.

C'est précisément le dual de notre problème.

4.4.3.2. Le primal est un problème de minimisation. — Considérons le problème suivant.

$$\text{Min} \sum_{j=1}^n c_j x_j; \quad \text{s.c.} \sum_{j=1}^n a_{ij} x_j \geq b_i \text{ pour tout } i = 1,\ldots,m$$

Dans un contexte industriel, cela peut correspondre à la situation d'une demande à satisfaire au coût minimum. Quelles quantités x_j des ressources j acheter pour satisfaire la demande à coût minimum ?

Le point de vue dual : à quel prix proposer les produits pour maximiser le profit tout en restant compétitif ?

$$\text{Max} \sum_{i=1}^m b_i y_i; \quad \text{s.c.} \sum_{j=1}^n a_{ij} y_i \leq c_j \text{ pour tout } j = 1,\ldots,n$$

4.5. Une application de la dualité : jeux à somme nulle

Soit $A = ((a_{ij}))$ une matrice m lignes n colonnes d'un jeu à somme nulle. Joueur C choisit une colonne j, et Joueur L choisit une ligne i. Chacun reçoit a_{ij}. Le joueur C cherche à minimiser, L cherche à maximiser.

Ce qu'on cherche en théorie des jeux, ce sont les *équilibres de Nash*, i.e. des situations où aucun des joueurs n'a intérêt à changer de choix. Ici, lorsque le choix est restreint au choix d'une ligne et d'une colonne, on parle d'un équilibre en *stratégies pures*. En général, il n'y a pas d'équilibre en stratégie pure. En revanche, comme l'avait remarqué von Neumann, si les joueurs choisissent des distributions de probabilité, il y a un équilibre en *stratégies mixtes*, le gain des joueurs étant alors l'espérance de gain.

Notons Δ^k l'ensemble $\{(x_1, \ldots, x_k) \in \mathbb{R}_+^k : \sum_{i=1}^k x_i = 1\}$. C'est l'ensemble des probabilités sur l'univers $\{1, \ldots, k\}$.

Si C choisit une distribution $\boldsymbol{x} \in \Delta^n$ et si L choisit une distribution $\boldsymbol{y} \in \Delta^m$, l'espérance du gain est $\boldsymbol{y}^T A \boldsymbol{x}$.

***Théorème 4.5.1* (von Neumann).** — *Il existe $\tilde{\boldsymbol{x}} \in \Delta^n$, $\tilde{\boldsymbol{y}} \in \Delta^m$ et $v \in \mathbb{R}$ tels que*

$$\boldsymbol{y}'^T A \tilde{\boldsymbol{x}} \leq v \quad \textit{pour tout } \boldsymbol{y}' \in \Delta^m$$

et

$$\tilde{\boldsymbol{y}}^T A \boldsymbol{x}' \geq v \quad \textit{pour tout } \boldsymbol{x}' \in \Delta^n.$$

Noter que $v = \tilde{\boldsymbol{y}}^T A \tilde{\boldsymbol{x}}$. Le théorème signifie que le joueur C a toujours intérêt à jouer la distribution $\tilde{\boldsymbol{x}}$ et L la distribution $\tilde{\boldsymbol{y}}$. Tout autre choix serait pénalisant.

Démonstration du théorème 4.5.1. — On peut supposer

$$A >> 0$$

(en effet, ajouter une même constante sur chaque entrée de la matrice ne change rien).

Considérer le programme linéaire

$$\begin{array}{ll} \text{Max} & \sum_{j=1}^n x_j \\ \text{s.c.} & A\boldsymbol{x} \leq \mathbf{1} \\ & \boldsymbol{x} \geq \mathbf{0}. \end{array}$$

Le dual est

$$\begin{array}{ll} \text{Min} & \sum_{i=1}^m y_i \\ \text{s.c.} & A^T\boldsymbol{y} \geq \mathbf{1} \\ & \boldsymbol{y} \geq \mathbf{0}. \end{array}$$

Soit $\boldsymbol{x}^*$ la solution optimale du primal et $\boldsymbol{y}^*$ celle du dual. Ils ont même valeur d'après le Théorème fort de la dualité $w := \sum_{j=1}^n x_j^* = \sum_{i=1}^m y_i^*$, finie car $A >> 0$. Les solutions $\tilde{\boldsymbol{x}} := \frac{\boldsymbol{x}^*}{w}$, $\tilde{\boldsymbol{y}} := \frac{\boldsymbol{y}^*}{w}$ et $v := \frac{1}{w}$ satisfont les relations demandées. □

Exercices

4.5.1. Production de boissons. — Une entreprise fabrique trois types de boissons : XXX, YYY et ZZZ qui rapportent respectivement $1,55$€, $3,05$€ et $4,80$€ le litre. Chaque boisson a un niveau minimal de production hebdomadaire qui sont respectivement de 1020, 1450 et 750 litres. Un litre de chaque boisson requiert un temps en heures de *fabrication*, un temps de *mélange* et un temps d'*embouteillage*.

	XXX	YYY	ZZZ
fabrication	2	3	6
mélange	4	5,5	7,5
embouteillage	2	1	3,5

Pendant la semaine à venir, l'entreprise aura 180 heures disponibles en fabrication, 210 en mélange et 65 en embouteillage. Formuler un modèle donnant un plan de production qui maximise le profit.

4.5.2. Un problème de raffinerie - d'après Hêche, Liebling et de Werra [6]. — Une raffinerie doit fournir chaque jour deux qualités A et B d'essence à partir de constituants 1, 2 et 3. On note $Q_{\max}$ la quantité maximale disponible quotidiennement :

constituant	$Q_{\max}$	coût unitaire
1	3000	3
2	2000	6
3	4000	4

essence	spécification	prix de vente unitaire
A	$\leq 30\%$ de 1	5,5
	$\geq 40\%$ de 2	
	$\leq 50\%$ de 3	
B	$\leq 50\%$ de 1	4,5
	$\geq 10\%$ de 2	

Donner un modèle qui permet de maximiser la recette, sachant que toute la production pourra être écoulée.

4.5.3. Formes standard, canonique, inéquationnelle. — Montrer qu'un programme écrit sous une de ces trois formes peut toujours s'écrire sous les deux autres formes.

4.5.4. Dual du dual. — En utilisant les recettes de dualisation du cours, montrer que le dual du dual est équivalent au primal.

4.5.5. Théorème fort de la dualité. — En s'appuyant sur la caractérisation de l'optimalité de la solution retournée par l'algorithme du simplexe, prouver le théorème fort de la dualité (Théorème 4.4.1).

4.5.6. Ecarts complémentaires. — En utilsant le théorème fort de la dualité, prouver le théorème des écarts complémentaires Proposition 4.4.3.

4.5.7. Trouver une solution réalisable. — Soit le programme linéaire suivant, donné sous forme standard

$$\begin{array}{ll} \text{Min} & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.c.} & A\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{array} \tag{16}$$

où $\boldsymbol{x}, \boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$ et où A est une $m \times n$ matrice réelle.

1. Montrer qu'on peut supposer que $\boldsymbol{b} \geq \boldsymbol{0}$.

Considérons maintenant le programme

$$\begin{array}{ll} \text{Min} & \sum_{i=1}^{m} y_i \\ \text{s.c.} & A\boldsymbol{x} + \boldsymbol{y} = \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \\ & \boldsymbol{y} \geq \boldsymbol{0} \end{array} \tag{17}$$

2. Montrer que l'équation (16) a une solution réalisable si et seulement si l'équation (17) a une valeur optimale $= 0$.

Par conséquent, l'algorithme du simplexe est une méthode pour résoudre un système d'équations et d'inéquations linéaires.

3. Réciproquement, montrer avec la théorie de la dualité que si on a une méthode pour trouver une solution réalisable de n'importe quel système d'équations et d'inéquations linéaires, alors on peut utiliser cette méthode pour résoudre les programmes linéaires.

4.5.8. Meilleure droite approximante. — Soit $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ des points dans le plan. On cherche la meilleure droite approximante pour la norme sup, i.e. la droite du plan d'équation $ax + b = y$ telle que $\sup_{i=1,\ldots,n} |ax_i + b - y_i|$ soit le plus petit possible. Montrer que cela revient à résoudre un programme linéaire. Même question en cherchant à minimiser $\sum_{i=1}^{n} |ax_i + b - y_i|$.

4.5.9. Interprétation économique de la dualité. — Un diététitien utilise six produits comme source de vitamines A et C. Il a une certaine demande en vitamine A et en vitamine C. Il veut la satisfaire au moindre coût.

	Produits (unités/kg)						Demande
vitamine A	1	0	2	2	1	2	9
vitamine C	0	1	3	1	3	2	19
Prix par kg	35	30	60	50	27	22	

1. Modéliser ce problème comme un programme linéaire.

2. Ecrire le programme dual. Proposer une interpréation.

4.5.10. Programmation linéaire en nombres entiers. — Un programme linéaire du type

$$\begin{array}{ll} \text{Min} & \boldsymbol{c}^T \boldsymbol{x} \\ \text{s.c.} & A\boldsymbol{x} = \boldsymbol{b} \\ & x_j \in \mathbb{N} \text{ pour } j = 1, \ldots, n \end{array} \tag{18}$$

est appelé programme linéaire en nombres entiers.

Utiliser le fait que le problème du sac-à-dos est **NP**-difficile pour montrer que le programme (18) est **NP**-difficile.

Dans le cas où les variables x_j peuvent prendre des valeurs réelles, la programmation linéaire est polynomial (avec les ellipsoïdes ou les points intérieurs). Si certaines de ces variables sont contraintes à être entières, on perd la polynomialité.

CHAPITRE 5

FLOTS ET MULTIFLOTS

Les flots et les multiflots sont des notions naturelles : étant donné un réseau de transport (train, tuyau, cables électriques), avec des capacités sur les arcs, quelle quantité de biens peut-on faire transiter au maximum ? Ou alors, en supposant qu'à chaque arc est attaché un coût unitaire, à quel coût minimum peut-on faire transiter un volume de biens donné ?

On parle de flot lorsqu'il n'y a qu'un seul bien qui circule. Sinon, on parle de multiflot.

Algorithmiquement, les flots sont plus simples à traiter que les multiflots. Nous commencerons donc par les flots.

5.1. Flots

5.1.1. Généralité. —

5.1.1.1. Définition. — La notion de flot ne peut exister que par rapport à un réseau. Avant de défnir ce qu'est un flot, il nous faut d'abord nous fixer un réseau. Soit donc un réseau modélisé par un graphe orienté $D = (V, A)$, muni de capacité $u \to \mathbb{R}_+$ et de deux sommets particuliers s (la *source*) et t (le *puits*). Une fonction $f : A \to \mathbb{R}_+$ est un *s–t–flot* si $f(a) \leq u(a)$ pour tout arc $a \in A$ et si pour tout sommet $v \neq s, t$, la loi de conservation

$$\sum_{a \in \delta^-(v)} f(a) = \sum_{a \in \delta^+(v)} f(a)$$

est respectée.

En introduisant la notation

$$\underset{f}{\text{excess}}(v) := \sum_{a \in \delta^-(v)} f(a) - \sum_{a \in \delta^+(v)} f(a),$$

cela revient à demander que $\text{excess}_f(v) = 0$ pour tout sommet $v \neq s, t$.

On définit la *valeur* du s–t flot f par $\text{value}(f) := -\text{excess}_f(s)$. On peut interpréter la valeur d'un flot comme la quantité de matière quittant la source s. Puisqu'il y a conservation de la matière aux sommets $\neq s, t$, il est intuitivement (et mathématiquement ? : exercice !) clair que l'on a aussi $\text{value}(f) = \text{excess}_f(t)$

5.1.1.2. Quelques propriétés. — Soit $X \subseteq V$ une partie des sommets telle que $s \in X$ et $t \notin X$. L'ensemble d'arcs $\delta^+(X)$ est appelée *s–t coupe*. Il est intuitivement clair qu'un flot devant passer par n'importe quelle s–t coupe aura toujours une valeur inférieure à la *capacité* de n'importe quelle s–t coupe, où la capacité d'une s–t coupe $\delta^+(X)$ est égale à $\sum_{a \in \delta^+(X)} u(a)$. Le lemme suivant formalise entre autres cette propriété.

Lemme 5.1.1. — *Soit $X \subseteq V$ une partie des sommets telle que $s \in X$ et $t \notin X$, et soit f un s–t flot quelconque. Alors*

$$\text{value}(f) = \sum_{a \in \delta^+(X)} f(a) - \sum_{a \in \delta^-(X)} f(a).$$

En particulier,

$$\text{value}(f) \leq \sum_{a \in \delta^+(X)} u(a). \tag{19}$$

La preuve de ce lemme est laissée en exercice.

On a également une propriété fondamentale qui donne corps à l'intuition : tout flot peut se décomposer en combinaison conique de s–t chemins élémentaires et circuits élémentaires.

Proposition 5.1.2. — *Soit f un s–t flot. Pour $B \subseteq A$, on note χ^B la fonction qui à $a \in A$ renvoie 1 si $a \in B$ et 0 sinon. Soient enfin $\mathcal{P}$ l'ensemble des s–t chemins élémentaires et $\mathcal{C}$ l'ensemble des s–t circuits élémentaires. Il existe des coefficients réels positifs $(\lambda_P)_{P \in \mathcal{P}}$ et $(\mu_C)_{C \in \mathcal{C}}$ tels que*

$$f = \sum_{P \in \mathcal{P}} \lambda_P \chi^P + \sum_{C \in \mathcal{C}} \mu_C \chi^C.$$

5.1.1.3. Graphe résiduel et chemins f–augmentants. — Cette sous-section vise à définir deux objets un peu techniques, le graphe résiduel et les chemins f–augmentants, dont l'intérêt est avant tout algorithmique. Il est intéressant de lire cette sous-section surtout si l'on cherche à comprendre les algorithmes ci-dessous.

Informellement, le graphe résiduel est le graphe qui indique les arcs où l'on peut augmenter la quantité de flot sans violer la loi de conservation et les bornes de capacités (voir exemple Figure 1).

Pour un arc $a = (u, v)$, on note $a^{-1} := (v, u)$. On définit $A^{-1} := \{a^{-1} : a \in A\}$. Soit f un s–t flot. On définit

$$A_f := \{a : a \in A \text{ et } f(a) < u(a)\} \cup \{a^{-1} : a \in A \text{ et } f(a) > 0\}.$$

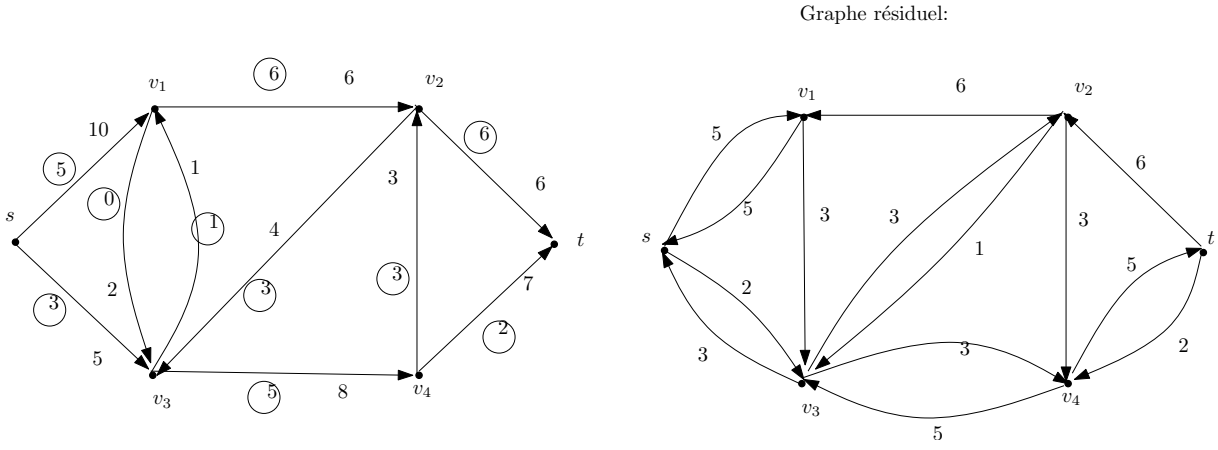


FIG. 1. Le graphe résiduel.

Le *graphe résiduel* est le graphe $D_f := (V, A_f)$. Pour $a \in A_f$, on associe une *capacité résiduelle* $u_f(a) := u(a) - f(a)$ si $a \in A$ et $u_f(a) := f(a)$ si $a^{-1} \in A$. Tout s–t–chemin dans D_f est appelé *chemin f-augmentant.*

5.1.2. Flot maximum et coupe minimum. —

5.1.2.1. Le problème. — La question naturelle qui suit la description d'un flot est celle de la valeur maximum d'un flot étant donné un réseau avec des capacités. Informellement, étant donné un réseau avec des capacités sur des arcs, la question est de savoir quelle quantité maximum de matière on peut faire passer de la source s à la destination t. Cette question se modélise par

Problème du flot maximum

Donnée : Un graphe $D = (V, A)$ deux sommets particuliers s et t, et des capacités $u : A \to \mathbb{R}_+$.

Tâche : Trouver un s–t–flot de valeur maximum.

En fait, ce problème est polynomial car il peut se modéliser comme un programme linéaire

$$\begin{array}{lll} \text{Max} & \displaystyle\sum_{a\in\delta^+(s)} x_a - \sum_{a\in\delta^-(s)} x_a & \\ \text{s.c.} & \displaystyle\sum_{a\in\delta^-(v)} x_a - \sum_{a\in\delta^+(v)} x_a = 0 & \text{pour } v \in V \setminus \{s,t\} \\ & x_a \le u(a) & \text{pour } a \in A \\ & x_a \ge 0 & \text{pour } a \in A, \end{array}$$

et peut être résolu avec l'algorithme des points intérieurs. L'algorithme du simplexe résout également très bien ce problème (bien que la solution ne se fasse pas alors forcément en temps polynomial). Mais il faut bien convenir que la structure des contraintes et de la fonction objectif est très particulière (entre autres, il n'y a que des coefficients ± 1), puisqu'elle traduit une structure en réseau. On peut (et doit) se demander s'il n'existe pas des algorithmes plus rapides et/ou plus simples qui résolvent ce problème en exploitant la structure de réseau. La réponse est 'oui', et il existe même une grande quantité de tels algorithmes. Nous allons décrire le plus simple (et le premier à avoir été polynomial), celui d'Edmonds et Karp [**8**].

5.1.2.2. Algorithme d'Edmonds et Karp. — On commence par poser $f(a) := 0$ pour tout $a \in A$. Ensuite on répète

> Trouver un chemin f–augmentant P dans le graphe résiduel D_f. S'il n'y en a pas, alors f est optimal. Sinon, on calcule μ le plus grand possible qui permette d'"augmenter" f le long de P et on "augmente" f de la quantité μ.

Le calcul de μ se formalise par $\mu := \min_{a\in P} u_f(a)$. Augmenter f le long de P se fait de la manière suivante : si $a \in P$ est dans le même sens dans A (i.e. $a \in A$), alors on pose $f(a) := f(a) + \mu$; sinon (i.e. $a^{-1} \in A$), alors on pose $f(a) := f(a) - \mu$.

L'algorithme décrit ci-dessus n'est polynomial que si on choisit toujours le chemin f-augmentant ayant le moins d'arcs possible, ce qui se fait par un algorithme de plus court chemin avec comme coût 1 sur les arcs[(1)].

[(1)]Dans ce cas particulier, coût = 1, il n'est pas nécessaire d'appliquer l'algorithme de Dijkstra. Une simple recherche en largeur d'abord donne la solution.

D'où

Théorème 5.1.3. — *On peut trouver un flot maximum en $O(nm^2)$.*

Remarque : Goldberg et Tarjan ont montré qu'il était possible de trouver un flot maximum en $O(nm \log n^2/m)$.

5.1.2.3. Théorème max flot–min coupe. — Une application de l'algorithme et du théorème ci-dessus est la célèbre propriété de max flot–min coupe. En effet, lorsque l'algorithme se termine, on a obtenu un flot maximum. Le critère qui a permis de s'arrêter est l'absence de chemin f-augmentant dans le graphe résiduel, i.e. qu'on ne peut atteindre t depuis s dans D_f. Posant X l'ensemble des sommets que l'on peut atteindre depuis s dans D_f, cela signifie que pour les arcs a quittant X (formellement $a \in \delta^+(X)$), on a $f(a) = u(a)$ et pour ceux entrant dans X (formellement $a \in \delta^-(X)$), on a $f(a) = 0$. Donc $\sum_{a\in\delta^+(X)} f(a) - \sum_{a\in\delta^-(X)} f(a) = \sum_{a\in\delta^+(X)} u(a)$. Ce qui signifie d'après le Lemme 5.1.1

$$\text{value}(f) = \sum_{a\in\delta^+(X)} u(a).$$

Avec l'équation (19), on obtient donc

Théorème 5.1.4. — *La valeur maximale d'un s–t flot est égale à la capacité minimale d'une s–t coupe.*

Et on peut donc trouver une s–t coupe de capacité minimale en $O(nm^2)$. En effet, après avoir appliqué l'algorithme d'Edmonds et Karp, l'ensemble X des sommets que l'on peut atteindre dans D_f depuis s est tel que $\delta^+(X)$ est une coupe de capacité minimale.

Une autre conséquence importante de l'algorithme est la suivante

Théorème 5.1.5. — *Si toutes les capacités sont entières, alors il existe un s–t–flot maximum entier, et l'algorithme d'Edmonds et Karp le trouve.*

Dire que le flot f est entier signifie que $f(a)$ est entier pour tout $a \in A$. En fait, on a aussi l'existence de la décomposition de la Proposition 5.1.2 avec des coefficients λ_P et μ_C entiers positifs.

5.1.2.4. Application : Contrôle routier. — Pour améliorer l'utilisation des flottes de camion (pour diminuer les émissions de CO2 par exemple), on souhaite effectuer des contrôles de poids lourds partant d'une région P et allant vers une autre région Q par le biais d'un réseau autoroutier R. Les contrôles sont effectués sur des tronçons d'autoroute. Faire un contrôle sur le tronçon t coûte c_t. Le problème consiste à trouver le sous-ensemble de tronçons sur lesquels les contrôles vont être effectués tel que tout camion allant de P à Q passe par un contrôle, et ce, à coût minimum.

On modélise le réseau routier par un graphe orienté $D = (V, A)$. On ajoute un sommet p rélié à tous les points d'entrée du réseau en P, et un sommet q auquel tout point de sortie du réseau en Q est relié. Les autres arcs a modélisent les tronçons t. Sur un arc a correspondant à un tronçon t, on met un capacité $u(a)$ égale à c_t. Sur les arcs quittant p, ou atteignant q, on met une capacité infinie, modélisant ainsi le fait que l'on ne veut (ou ne peut) pas effectuer des contrôles sur ces arcs. On cherche alors une coupe de capacité minimale, ce qui peut se faire par l'algorithme d'Edmonds et Karp. En effet, lorsque l'algorithme s'arrête, l'ensemble X des sommets que l'on peut atteindre dans D_f depuis p est tel que $\delta^+(X)$ est une coupe de capacité minimale.

5.1.2.5. Application : Problème de l'affectation de tâches. — On suppose que l'on a différentes tâches à accomplir dont on connaît les durées d'exécution, et que l'on dispose d'une ressource de main d'œuvre dont on connaît les compétences. Les tâches sont supposées telles que les employés peuvent y travailler en parallèle. On souhaite minimiser le temps nécessaire pour réaliser l'ensemble des tâches.

Problèmes de l'affectation des tâches

Données : n tâches et leurs durées $t_1, \dots, t_n \in \mathbb{R}_+$; m employés et des sous-ensembles $S_i \subseteq \{1, \dots, m\}$ qui correspondent aux employés compétents pour la tâche i.

Tâche : Trouver des réels $x_{ij} \in \mathbb{R}_+$ pour tous $i \in \{1, \dots, n\}$ et $j \in \{1, \dots, m\}$ (signifiant le temps consacré par l'employé j à la tâche i) tels que toutes les tâches soient finies, i.e. tels que $\sum_{j \in S_i} x_{ij} = t_i$ pour $i = 1, \dots, n$. Minimiser le temps qu'il faut pour terminer toutes les tâches, i.e. la quantité $T(x) := \max_{j \in \{1,\dots,m\}} \sum_{i:\, j \in S_i} x_{ij}$.

Ce problème se modélise comme un problème de flot de la manière suivante. On construit un graphe biparti avec d'un côté des sommets v_i représentant les tâches et de l'autre des sommets w_j représentant des employés. On met un arc (v_i, w_j) si l'employé j est compétent pour effectuer la tâche i, en d'autres termes, si $j \in S_i$. On ajoute deux autres sommets s et t, et on construit tous les arcs (s, v_i) et tous les arcs (w_j, t). On dénomme ce graphe D. Fixons un temps T et on va se demander s'il est possible d'effectuer toutes les tâches en un temps plus petit que T.

On munit D de capacités $u(s, v_i) := t_i$, et $u(a) := T$.

Prenons maintenant un jeu de réels x_{ij} satisfaisant les contraintes du problème, et réalisant les tâches en un temps total $\leq T$. Associons à chaque arc (v_i, w_j) la quantité x_{ij}. A chaque arc (s, v_i) on associe la quantité $\sum_{j \in S_i} x_{ij}$ et à chaque arc (w_j, t) la quantité $\sum_{i:\, j \in S_i} x_{ij}$. On obtient ainsi un flot réalisable de valeur $\sum_{i=1}^n t_i$.

Réciproquement, supposons que l'on trouve un flot réalisable f de valeur $\sum_{i=1}^n t_i$. Alors, les réels $x_{ij} := f(v_i, w_j)$ constituent une solution réalisable du problème de l'affectation de tâches, avec un temps de réalisation total $\leq T$.

Par conséquent pour résoudre le problème de l'affectation de tâches (trouver une solution optimale), on cherche un flot maximum sur D. On sait qu'il y a un flot maximum de valeur $\sum_{i=1}^n t_i$ si on pose $T := \sum_{i=1}^n t_i$. On pose alors $T := \dfrac{\sum_{i=1}^n t_i}{2}$ et $T' := \sum_{i=1}^n t_i$. On répète

> Si on trouve un flot de valeur $\sum_{i=1}^n t_i$, on sait qu'on peut réaliser les tâches en un temps $\leq T$. On recommence alors avec $T := T/2$ et $T' := T$. Sinon, c'est que T était trop petit, et on pose $T := \dfrac{T + T'}{2}$ et on laisse $T' := T'$.

La solution optimale x^* est telle $T \leq T(x^*) \leq T'$. On peut s'arrêter dès que $T' - T$ est plus petit qu'une quantité fixée au préalable.

On a résolu un problème d'optimisation par une recherche binaire, chaque étape étant un problème de décision.

Cette façon de procéder est très efficace en pratique, plus que l'autre solution consistant à voir le problème de l'affectation de tâches comme un programme linéaire (Exercice).

Question : si les t_i sont entiers, a-t-on nécessairement une solution optimale entière ? (je ne sais pas) L'intérêt pourrait être de montrer que la recherche binaire termine alors en un nombre fini d'étapes.

5.1.3. Flot de coût minimum. —

5.1.3.1. Le problème. — Supposons maintenant que nous ajoutions un coût sur chaque arc. De plus, pour des commodités de modélisation, on va autoriser plusieurs sources et plusieurs puits.

Soit un graphe orienté $D = (V, A)$, muni de capacité $l \leq u : A \to \mathbb{R}_+$ et des nombres $b : V \to \mathbb{R}$ tels que $\sum_{v \in V} b(v) = 0$. Une fonction $f : A \to \mathbb{R}_+$ est un *b-flot* si pour tout arc $a \in A$, on a $l(a) \leq f(a) \leq u(a)$ et si pour tout sommet $v \in V$, la loi de conservation

$$\sum_{a \in \delta^+(v)} f(a) - \sum_{a \in \delta^-(v)} f(a) = b(v)$$

est respectée.

Les sommets v tels que $b(v) > 0$ sont appelés *sources* et les sommets v tels que $b(v) < 0$ sont appelés *puits*. Pour une source, $b(v)$ est l'*offre* ; pour un puits, $b(v)$ est la *demande*.

Dans le cas particulier où tous les $b(v)$ sont nuls, on parle de *circulation*.

On suppose que l'on a une fonction de coût $c : A \to \mathbb{R}$.

On peut alors se demander, parmi tous les b-flots, lequel est de plus petit coût, où le coût d'un b-flot f est défini par

$$\sum_{a \in A} c(a) f(a).$$

Les applications des flots de coût minimum sont innombrables. Nous verrons ce que cela peut signifier pour le problème de l'affectation de travaux. Tout comme le problème de flot maximum, le problème du b-flot de coût minimum se formule comme un programme linéaire et peut donc être résolu avec ma méthode des points intérieurs (en temps polynomial donc) ou par l'algorithme du simplexe. En effet, on peut écrire ce problème :

$$\begin{array}{lll} \text{Min} & \displaystyle\sum_{a \in A} c(a) x_a & \\ \text{s.c.} \quad b(v) + \displaystyle\sum_{a \in \delta^-(v)} x_a = \sum_{a \in \delta^+(v)} x_a & & \text{pour } v \in V \\ x_a \leq u(a) & & \text{pour } a \in A \\ x_a \geq l(a) & & \text{pour } a \in A, \end{array}$$

En 1985, Goldberg et Tarjan [**13**] ont montré qu'il existait un algorithme particulièrement efficace pour résoudre le problème flot de coût minimum. L'algorithme travaille encore sur le graphe résiduel, en définissant $c(a^{-1}) := -c(a)$ pour $a \in D$ et

$$A_f := \{a : a \in A \text{ et } f(a) < u(a)\} \cup \{a^{-1} : a \in A \text{ et } f(a) > l(a)\}.$$

Le graphe résiduel D_f est donc maintenant muni de coûts.

5.1.3.2. Algorithme de Goldberg et Tarjan. — On commence par fixer un b-flot (voir ci-dessous : comment trouver un b–flot réalisable). Le *coût moyen* d'un circuit C est défini par

$$\frac{\sum_{a \in C} c(a)}{|C|},$$

somme des coûts des arcs du circuit C divisée par le nombre d'arcs du circuit C.

On repète

> Repérer un circuit C de coût moyen minimum dans D_f. S'il n'y en a pas, alors f est optimal. Sinon, on calcule μ le plus grand possible qui permette d'"augmenter" f le long de C et on "augmente" f de la quantité μ.

Le calcul de μ se formalise par $\mu := \min_{a \in C} u_f(a)$. Augmenter f le long de C se fait de la manière suivante : si $a \in C$ est dans le même sens dans A (i.e. $a \in A$), alors on pose $f(a) := f(a) + \mu$; sinon (i.e. $a^{-1} \in A$), alors on pose $f(a) := f(a) - \mu$.

La difficulté de cet algorithme est de trouver un circuit de coût moyen minimum dans le graphe résiduel D_f. En fait, pour cela, on peut faire appel à un algorithme de Karp [**20**] qui trouve un tel circuit en $O(nm)$.

Théorème 5.1.6. — *On peut trouver un b-flot de coût minimum en $O(n^2 m^3 \log n)$.*

Comme pour le problème du flot maximum, si les capacités sont entières, il existe une solution optimale entière, que l'on trouve sans plus de difficulté.

Théorème 5.1.7. — *Si toutes les capacités sont entières et tous les $b(v)$ sont entiers, alors il existe un b-flot de coût minimum entier, et l'algorithme de Goldberg et Tarjan le trouve.*

Si l'on accepte d'avoir une complexité qui dépend de la taille des nombres de l'input (on parle alors d'algorithme *faiblement polynomial*), on peut avoir d'autres bonnes complexités. Par exemple, l'algorithme classique d'Edmonds et Karp [**8**] tourne en $O(m(m + n \log n)L)$, où L est le nombre maximum de bits pour coder un coefficient du problème.

5.1.3.3. Comment trouver un b-flot réalisable. — Remarquons d'abord que le cas où $l(a) = 0$ pour tout $a \in A$ est facile. En effet, en ajoutant une source fictive s, un puits fictif t, les arcs (s, v) avec capacité $b(v)$ pour les v tels que $b(v) > 0$ et les arcs (v, t) avec capacité $-b(v)$ pour les v tels que $b(v) < 0$, l'existence d'un b-flot se resout par un algorithme de flot maximum, par exemple avec l'algorithme d'Edmonds et Karp ci-dessus.

Nous allons voir maintenant comment nous ramener au cas $l(a) = 0$ pour tout arc a. On fait la même transformation que ci-dessus, et on ajoute un arc (t, s) de capacité infinie. On cherche donc une circulation $f(a)$ dans ce graphe. Montrons donc comment trouver une circulation dans un graphe dont les arcs sont munis de bornes inférieures $l(a)$ et de bornes supérieures $u(a)$.

Une telle circulation par définition satisfait les contraintes suivantes

$$\sum_{a \in \delta^+(v)} f(a) = \sum_{a \in \delta^-(v)} f(a) \quad \text{pour tout } v \in V,$$

$$l(a) \leq f(a) \leq u(a) \quad \text{pour tout } a \in A.$$

Posons $f'(a) = f(a) - l(a)$. On voit que le problème de l'existence de la circulation $f(a)$ revient à celui de l'existence d'un flot f' satisfaisant

$$\sum_{a \in \delta^+(v)} l(a) + \sum_{a \in \delta^+(v)} f'(a) = \sum_{a \in \delta^-(v)} l(a) + \sum_{a \in \delta^-(v)} f'(a) \quad \text{pour tout } v \in V,$$

$$0 \leq f'(a) \leq u(a) - l(a) \quad \text{pour tout } a \in A.$$

Ce dernier est un problème d'existence de b'-flot avec borne inférieure $= 0$, où

$$b'(v) := \sum_{a \in \delta^-(v)} l(a) - \sum_{a \in \delta^+(v)} l(a),$$

lequel est résolu par un problème de flot maximum, comme indiqué au début de cette discussion.

5.1.3.4. Application : Conception d'un réseau de transport à moindre coût, d'après [**6**]. — Supposons que l'on souhaite calibrer un réseau de transport de façon à pouvoir assurer des livraisons depuis des sources jusqu'à des destinations. Pour chaque tronçon direct (u, v), on connaît le coût $c(u, v)$ d'établissement d'une liaison de capacité unitaire. Comment construire le réseau dont l'établissement soit de coût minimum ? On suppose que l'on connaît pour les sources, l'offre et pour les destinations, la demande.

Ce problème se modélise comme la recherche d'un b-flot de coût minimum. En effet, on cherche le b-flot de coût minimum, où les coûts sont donnés par les $c(u, v)$ et $b(v)$ est égal à l'offre si v est une source, et est égal à $-$ demande si v est une destination. Noter que d'après le Théorème 5.1.7, on peut trouver une solution entière à ce problème, ce qui est commode si chaque tronçon ne peut avoir que des capacités unitaires.

5.1.3.5. Application : Problème de l'affectation de tâches avec considération salariale. — Soit le problème

Problème de l'affectation de tâche avec salaires

Données : n tâches et leurs durées $t_1, \dots, t_n \in \mathbb{R}_+$; m employés et des sous-ensembles $S_i \subseteq \{1, \dots, m\}$ qui correspondent aux employés compétents pour la tâche i ; un salaire horaire c_{ij} pour l'employé j lorsqu'il effectue la tâche i ; une durée limite T pour la réalisation de l'ensemble des tâches.

Tâche : Trouver des réels $x_{ij} \in \mathbb{R}_+$ pour tous $i \in \{1, \dots, n\}$ et $j \in \{1, \dots, m\}$ tels que toutes les tâches soient finies en un temps inférieur à T, i.e. tels que $\sum_{j \in S_i} x_{ij} = t_i$ pour $i = 1, \dots, n$ et tels $\max_{j \in \{1,\dots,m\}} \sum_{i:\, j \in S_i} x_{ij} \leq T$. Minimiser le coût qu'il faut pour terminer toutes les tâches, i.e. la quantité $C(x) := \sum_{j \in \{1,\dots,m\}, i \in \{1,\dots,n\}} c_{ij} x_{ij}$.

On peut répéter la construction du graphe D comme dans la sous-section précédente, avec les mêmes capacités. Les coûts sont $c(v_i, w_j) := c_{ij}$, $c(s, v_i) := 0$ et $c(w_j, t) := 0$. Une simple application d'un algorithme de flot de coût minimum donne la solution.

5.2. Multiflots

Nous n'avons jusqu'à maintenant envisagé qu'un seul bien traversant le réseau. Dans de nombreuses applications (en particulier dans le transport), il y a plusieurs biens distincts qui utilisent le même réseau. On parle alors de multiflots.

Un multiflot se définit pour une paire de graphes orientés (D, H), où $D = (V, A)$ est le *graphe d'offre* et $H = (T, R)$ avec $T \subseteq V$ le *graphe de demande*. Le graphe D est muni de *capacités* $u : A \to \mathbb{R}_+$. Un *multiflot* f est une collection $(f_r)_{r \in R}$, où chaque f_r est s-t-flot pour chaque paire $r = (s, t)$ dans R, telle que

$$\sum_{r \in R} f_r(a) \leq u(a) \quad \text{pour tout } a \in A.$$

Différents objectifs peuvent être cherchés. On peut chercher à maximiser la quantité totale de biens transitant dans le réseau, i.e. la quantité $\sum_{r \in R} \text{value}\, f_r$. On peut fixer une *demande* $d : R \to \mathbb{R}_+$ et se demander si le réseau permet de la satisfaire, i.e. s'il existe un multiflot f tel que pour tout r on ait $\text{value}\, f_r = d(r)$. Parfois, on met en plus des coûts $c : A \to \mathbb{R}$ sur les arcs et on cherche à satisfaire la demande au moindre coût, i.e. que l'on cherche le multiflot f tel que $\text{value}\, f_r = d(r)$ pour tout $r \in R$ avec $\sum_{a \in A} \left(c(a) \sum_{r \in R} f_r(a) \right)$ minimum.

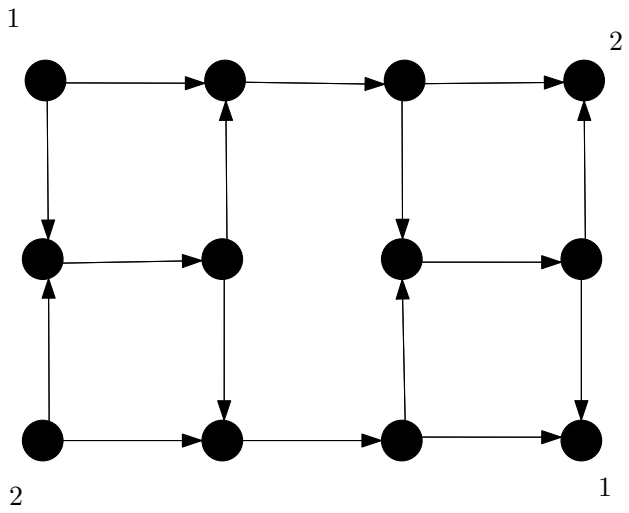


FIG. 2. Contre-exemple à la propriété d'intégrité des multi-flots

Toutes ces questions peuvent être résolus en temps polynomial. En effet, elles se formulent comme des programmes linéaires.

L'intérêt de formuler un problème linéaire comme un problème de multiflot réside dans le fait qu'il existe des algorithmes plus efficaces pour le résoudre. Ces algorithmes dépassent largement la cadre de ce cours. Il faut simplement savoir que si un programme linéaire n'a que des $0, \pm 1$ comme coefficients dans ses contraintes, alors il existe des algorithmes plus efficaces que le simplexe ou les points intérieurs. Par exemple, Tardos a proposé tel un algorithme [**24**] qui est *fortement polynomial*, i.e. dont le nombre d'itérations ne dépend pas des valeurs prises par c ou d. En pratique, il existe des algorithmes encore plus efficaces, et polynomiaux, qui résolvent approximativement le problème (voir par exemple [**11**]).

Un autre intérêt réside dans la possibilité de faire de la *génération de colonne* facilement. La génération de colonne dépasse aussi le cadre de ce cours.

En revanche, dans le cas des multiflots, des capacités entières n'impliquent plus forcément l'existence d'un flot de même valeur entier. L'exemple de la Figure 5.2 permet de s'en convaincre (exercice) : les capacités sont supposées être égales à 1 sur tous les arcs, et la valeur de chaque flot est égale à 1.

Ce dont il faut se souvenir, c'est que si un problème peut être modélisé comme un problème de multiflot, il peut être bon de le souligner, et de chercher des méthodes qui exploitent cette propriété.

5.3. Exercices

5.3.1. Valeur d'un s–t flot. — Prouver le Lemme 5.1.1.

5.3.2. Combat sur un réseau. — Un centre de commandement est situé en un sommet p d'un réseau non-orienté. On connaît la positions des subordonnés modélisée par un sous-ensemble S des sommets du réseau. On souhaite détruire un nombre minimum de liens afin d'empêcher toute communication entre le centre de commandement et les subordonnés. Comment résoudre ce problème en temps polynomial ?

5.3.3. Le problème des représentants – d'après Ahula, Magnanti et Orlin [1]. — Une ville à n citoyens, c clubs et p partis politiques. Chaque citoyen appartient à au moins un club, et à au plus un parti.

Le jour du renouvellement du conseil de la ville approche... Le nombre de conseillers du parti P_k ne doit pas excéder u_k. Or, chaque club doit nommer un représentant au conseil de la ville, et certains de ces représentants sont membre d'un parti politique... Proposer un algorithme polynomial permettant de décider si ces contraintes peuvent être satisfaites.

5.3.4. Théorème de Menger. — Démontrer le théorème suivant.

Soit $D = (V, A)$ une graphe orienté, et s et t deux sommets particuliers. Le nombre maximum de s-t chemins arc-disjoints est égale à la cardinalité minimale d'un sous-ensemble d'arcs intersectant tout s-t chemin.

5.3.5. Problème de transport de Monge. — Le premier problème de Recherche Opérationnelle à visée pratique a été étudié par Monge en 1781 sous le nom du problème des déblais et remblais. Considérons n tas de sable, devant servir à combler m trous. Notons a_i la masse du $i$ème tas de sable et b_j la masse de sable nécessaire pour combler le $j$ème trou. Quel plan de transport minimise la distance totale parcourue par le sable ?

Montrer que ce problème se modélise comme un problème de flot de coût minimum.

5.3.6. Flotte d'avions. — Une compagnie aérienne a p vols à satisfaire. Pour chacun de ces vols $i = 1, \ldots, p$, elle connaît son lieu o_i et son heure de départ h_i, la durée du vol t_i et le lieu d'arrivée d_i. De plus, le temps pour se rendre de d_j à o_i est connu pour chaque couple (i, j). La compagnie souhaite minimiser le nombre d'avions tout en satisfaisant la demande en vols. Modéliser ce problème comme un problème de flot.

5.3.7. Flotte d'avions bis. — Une compagnie aérienne a la possibilité de satisfaire p vols. Pour chacun de ces vols $i = 1, \ldots, p$, elle connaît son lieu o_i et son heure de départ h_i, la durée du vol t_i, le lieu d'arrivée d_i et le produit de la vente des billets. De plus, le temps pour se rendre de d_j à o_i est connu pour chaque couple (i, j). Enfin, pour chaque vol (u, v), le coût $c(u, v)$ du vol est connu (frais, salaire, carburant). La compagnie souhaite maximiser son gain, sachant qu'elle dispose de K avions.

Modéliser ce problème comme un problème de flot.

5.3.8. Remplissage d'un avion. — Un avion capable d'embarquer au plus B passagers doit partir de l'aéroport 1 à destination des aéroports $2, 3, \ldots, n$ successivement. Le nombre de passagers voulant voyager de l'aéroport i à l'aéroport j (avec $i < j$) est $d(i, j)$ et le prix de ce trajet est $p(i, j)$. Combien de passagers faut-il prendre à chaque aéroport pour maximiser les recettes totales ? Modéliser ce problème comme un problème de flot.

5.3.9. Gestion dynamique de stock, cas à 1 seul bien. — On souhaite satisfaire une demande prescrite d_t pour chacune des T périodes $t = 1, 2, \ldots, T$. Pour satisfaire la demande d_t sur la période t, on peut produire une quantité $x_t \in \mathbb{R}_+$ sur la période t et/ou retirer une certaine quantité du stock $y_{t-1} \in \mathbb{R}_+$ de la période $t-1$ (on suppose $y_0 = 0$). On suppose de plus que la production sur la période t ne peut pas excéder P_t.

1. Justifier la dynamique du stock

$$y_t = y_{t-1} + x_t - d_t, \tag{20}$$

pour $t = 1, 2, \ldots, T$.

Sur la période t, le coût unitaire de stockage est $s_t \geq 0$ et le coût unitaire de production est $p_t \geq 0$. On veut gérer le stock au coût minimum.

2. Montrer que ce problème se modélise comme un programme linéaire.

3. *Application numérique :* On considère les données du tableau suivant

$t =$	1	2	3	4
d_t	5	4	1	3
s_t	1	1	1	xxx
p_t	2	3	3	4
P_t	9	5	5	5

Indiquer le coût minimal de gestion de stock, ainsi que les niveaux de productions (en p.4 , des programmes linéaires sont donnés).

4. Montrer que ce problème peut également se modéliser comme un problème de b-flot de coût minimum. (Indication : introduire un sommet "source" v avec $b(v) = \sum_{t=1}^{T} d_t$ et des sommets "puits" w_t avec $b(w_t) = -d_t$; à vous d'indiquer les arcs, les coûts sur les arcs, les capacités sur les arcs, etc.) Justifier la modélisation.

5. Quel peut être l'intérêt d'une telle modélisation alors qu'on sait résoudre le problème avec un solveur de programmation linéaire ?

5.3.10. Extraction de mine à ciel ouvert. — Un domaine d'application de la recherche opérationnelle est l'exploitation des mines à ciel ouvert. Un problème important consiste à déterminer les *blocs* à extraire. Dans toute la suite, la mine sera assimilée à une collection de n blocs numérotés de 1 à n. L'extraction du bloc i rapporte la quantité c_i ; cette quantité peut être positive ou négative (en fonction de la quantité de minerai présent dans le bloc).

Les blocs ne peuvent pas être extraits dans n'importe quel ordre : pour chaque bloc i, on connaît l'ensemble Y_i des blocs qu'il faut avoir extraits pour pouvoir extraire le bloc i. L'objectif est de proposer un sous-ensemble de blocs à extraire de manière à maximiser le profit total.

Dans un graphe orienté, on dit qu'un ensemble S de sommets est *fermé* si tout successeur d'un sommet de S est encore dans S. Considérons le problème suivant.

Le problème du fermé maximum.

Données. Un graphe $D = (V, A)$ orienté, des réels c_v pour tout $v \in V$.
Tâche. Trouver un sous-ensemble fermé $S \subseteq V$ tel que $\sum_{v \in S} c_v$ soit maximal.

1. Expliquez pourquoi le problème peut se modéliser sous la forme d'un problème de fermé maximum.

L'objectif va maintenant être de modéliser le problème du fermé maximum comme un problème de s-t coupe de capacité minimale.

On construit un nouveau graphe $\tilde{D} = (\tilde{V}, \tilde{A})$ en ajoutant un sommet source s et un sommet puits t à V. On note $V^+ = \{v \in V : c_v \geq 0\}$ et $V^- = \{v \in V : c_v < 0\}$. L'ensemble $\tilde{A}$ est alors A auquel on ajoute les arcs $\{(s, v) : v \in V^+\}$ et les arcs $\{(v, t) : v \in V^-\}$. On met une capacité $u_{(s,v)} = c_v$ pour $v \in V^+$ et $u_{(v,t)} = -c_v$ pour $v \in V^-$.

2. Quelles capacités u_a mettre sur les arcs $a \in A$ de manière à ce que tout $X \subseteq V$ tel que $\delta^+_{\tilde{D}}(X \cup \{s\})$ soit une s-t coupe de $\tilde{D}$ de capacité minimale soit un ensemble fermé de D ?

3. Soit $X \subseteq V$ un ensemble fermé de D. Montrez que

$$\sum_{a \in \delta^+_{\tilde{D}}(X \cup \{s\})} u_a = \sum_{v \in V^+} c_v - \sum_{v \in X} c_v.$$

4. En déduire que le problème du fermé maximum se modélise comme un problème de coupe minimale. Conclure sur la résolution pratique de problème.

CHAPITRE 6

GRAPHES BIPARTIS : PROBLÈME D'AFFECTATION, PROBLÈME DE TRANSPORT, MARIAGES STABLES

L'objectif de ce cours est d'étudier un objet central de l'optimisation discrète et d'en voir quelques applications les plus importantes. Il s'agit des graphes bipartis.

6.1. L'objet

Un *graphe biparti* est un graphe dont l'ensemble des sommets peut être partitionné en deux parties A et B telles que toute arête à l'une de ces extrémités dans A et l'autre dans B. Voir la Figure 1. Notons en passant la Proposition suivante, très utile, dont la démonstration est laissée en exercice.

Proposition 6.1.1. — *Un graphe est biparti si et seulement si il ne contient pas de circuit de taille impaire.*

6.2. Problème du couplage optimal

Rappelons qu'un couplage dans un graphe $G = (V, E)$ est un sous-ensemble d'arêtes $M \subseteq E$ disjointes : quelle que soient e et f dans M, les arêtes e et f n'ont pas de sommet en commun. On note $\nu(G)$ la cardinalité maximale d'un couplage d'un graphe.

Problème du couplage de poids maximum :

Données : Un graphe $G = (V, E)$ biparti avec des poids $w : E \to \mathbb{R}$ sur les arêtes.

Demande : Un couplage de poids maximal.

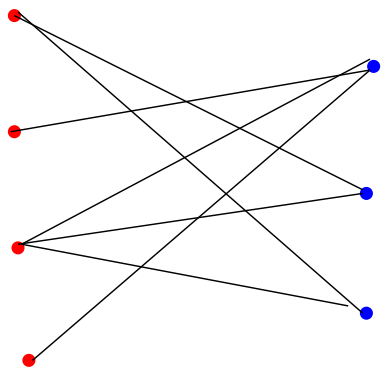


FIG. 1. Un exemple de graphe biparti.

Le problème du couplage de poids maximum se résout en temps polynomial. En effet, ce problème peut se modéliser comme un problème de flot de coût maximum (qui est un problème de flot de coût minimum avec des coûts opposés) : on oriente toutes les arêtes de A vers B, on ajoute un sommet s et un sommet t, on ajoute des arcs (s, v) pour $v \in A$ et (v, t) pour $v \in B$; enfin, on ajoute un arc (s, t), et on pose $b(s) = |A| = -b(t)$ et $b(v) = 0$ ailleurs. Enfin, les capacités des arcs sont partout $= +\infty$ sauf sur les arcs (s, v) et (v, t) où elles sont égales à 1, et les coûts sont partout nuls sauf sur les arcs de A à B où ils sont égaux aux poids w. C'est un problème de flot de coût maximum : en effet, tout flot entier induit un couplage de même poids ; et tout couplage induit un flot entier de même coût.

Tout algorithme trouvant des flots de coût optimal entier trouve donc la solution (en particulier celui vu au chapitre précédent). En fait, il existe un algorithme plus efficace, appelé *algorithme hongrois*, découvert par Kühn en 1955 [**21**].

Soit $G = (V, E)$ notre graphe biparti, avec les ensembles A et B partitionnant V et n'induisant chacun aucune arête. On a de plus une fonction w de poids sur E. L'algorithme hongrois commence avec un couplage $M = \emptyset$. Ensuite, on répète

> Créer le graphe orienté D_M de la façon suivante :
> – Orienter chaque arête e de M de B vers A, et définir $l(e) := w(e)$.
> – Orienter chaque arête e de $E \setminus M$ de A vers B et définir $l(e) := -w(e)$.
> Soit A_M (resp. B_M) les sommets de A (resp. B) qui ne sont pas couverts par une arête de M. S'il y a un chemin de A_M à B_M, en chercher un plus court (pour la fonction l), que l'on note P. Remplacer le M courant par $M \triangle E(P)$ (où $E(P)$ est l'ensemble des arêtes de P, et où $X \triangle Y$ représente les éléments présents dans exactement l'un des ensembles X ou Y[(1)]).

L'algorithme s'arête lorsqu'on ne peut pas trouver de chemin de A_M à B_M dans D_M. On peut démontrer que M est alors un couplage de poids maximum.

Un lecteur attentif remarquera qu'il faut calculer un plus court chemin dans D_M qui est un graphe orienté avec des poids quelconques. Ce problème est **NP**-dur en général. Heureusement ici, on peut montrer que l'algorithme est tel que D_M ne contient jamais de circuit absorbant et que l'on peut donc calculer un tel plus court chemin avec la méthode Bellman-Ford vue au Chapitre 3.

Avec quelque subtilité d'implémentation, l'algorithme hongrois peut tourner en $O(n(m + n \log n))$. On a donc le théorème suivant (donné ici sans preuve).

Théorème 6.2.1. — *Le problème du couplage de poids maximal peut être résolu en* $O(n(m + n \log n))$.

Avec un algorithme de flot, on est en $O(n^2 m^3 \log n)$.

Le problème du couplage maximum peut modéliser des situations où l'on veut attribuer des tâches à des personnes, affecter des personnes à des tâches, créer des binômes, etc.

Si on veut simplement trouver le couplage le plus grand (en nombre d'arêtes), on peut aller encore plus vite.

Théorème 6.2.2. — *Dans un graphe biparti, on peut construire un couplage de cardinalité maximale en* $O(\sqrt{n}m)$.

(1) On a donc $X \triangle Y = (X \cup Y) \setminus (X \cap Y)$; on appelle cette opération la *différence symétrique* de X et de Y.

Dans ce cas, on gardant la même modélisation par un graphe orienté (et en oubliant les coûts), on cherche le s-t flot de valeur maximale, ce qui donne une complexité de $O(nm^2)$. Par une série d'astuces, non détaillées ici, et sans utiliser la modélisation par les flots, on arrive à la complexité donnée dans le théorème ci-dessus.

Le couplage de cardinalité maximale peut modéliser la situation suivante. Les sommets du graphe biparti peuvent être des employés d'une part et des tâches à effectuer de l'autre. Les arêtes modélisent les compétences. On veut maximiser le nombre de tâches effectuées sachant que tout employé en fait au plus une seule.

Le théorème max flot–min coupe permet également de montrer le théorème suivant, dû à König (la preuve est laissée en exercice). On rappelle qu'une *couverture par les sommets* dans un graphe $G = (V, E)$ est un sous-ensemble de sommets $C \subseteq V$ tel que toute arête e de G soit incident à au moins un sommet de C et qu'on note $\tau(G)$ la cardinalité minimale d'une couverture par les sommets.

Théorème 6.2.3. — *Dans un graphe biparti, on a*

$$\nu(G) = \tau(G).$$

Rappelons que l'inégalité $\nu(G) \leq \tau(G)$ est triviale. L'algorithme de Ford-Fulkerson permet par la même construction que celle utilisée dans la preuve de calculer une telle couverture optimale. Cela dit, il existe des algorithmes plus rapides.

6.3. Problème de b-couplages simples optimaux

Soit $b : V \to \mathbb{N}$. On appelle *b-couplage simple* dans un graphe un ensemble d'arêtes $F \subseteq E$ tel que $\deg_F(v) \leq b(v)$ pour tout $v \in V$ ($\deg_F(v)$ signifie qu'on ne compte que les arêtes de F incidentes à v). Illustration Figure 2. On peut alors considérer le problème suivant.

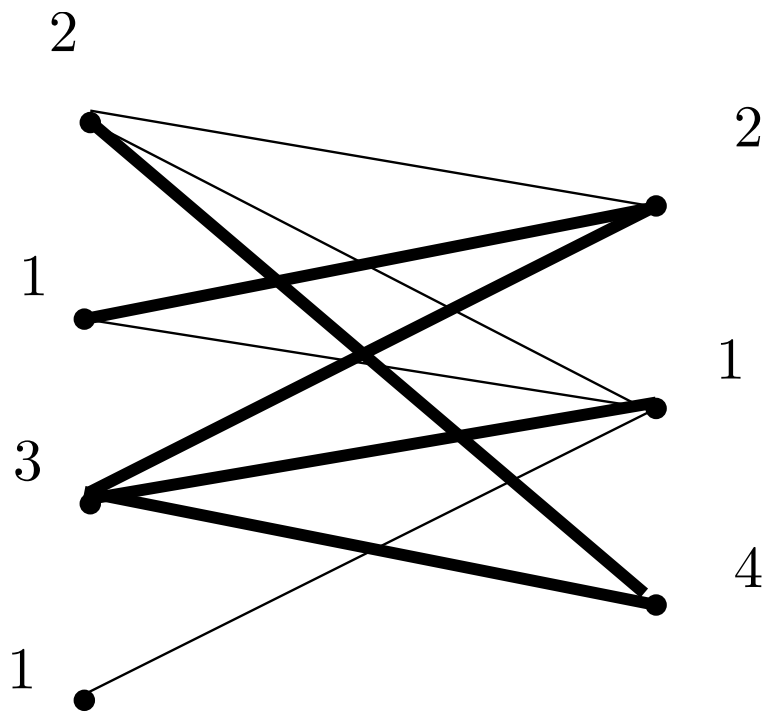


FIG. 2. Un exemple de b-couplage, avec les valeurs de b indiquées à côté des sommets.

Problème du b-couplage simple de poids maximum :

Données : Un graphe $G = (V, E)$ biparti avec des poids $w : E \to \mathbb{R}$ sur les arêtes.

Demande : Un b-couplage simple de poids maximal.

Il se résout en temps polynomial par un algorithme de flot de coût minimum, avec une construction similaire à celle de la section précédente. Noter que si on ne veut que maximiser la cardinalité, on peut utiliser un algorithme de flot maximum (plus rapide), toujours avec la même construction. Nous llons voir maintenant une application importante.

Exemple d'application : C'est un exemple inspiré d'un cas concret rencontré au centre de tri d'ADP (Aéroport de Paris). En début de journée, on a n palettes $1, \ldots, n$ à trier. La palette i contient a_i marchandises $m_{i,1}, \ldots, m_{i,a_i}$. Chaque marchandise est caractérisée par une heure limite de tri $t_{i,j}$, avec $j \in \{1, \ldots, a_i\}$. Si on dépasse l'heure limite, on "paye" une amende de $c_{i,j}$. Supposons que l'on dispose de k personnes pour trier, que chaque personne met une heure à trier et que tout se passe sur T créneaux d'une heure. Trouver la stratégie qui minimise l'amende totale.

C'est un problème de b-couplage simple de poids maximal. En effet, on considère le graphe biparti suivant. D'un côté, on a les palettes $1, 2, \ldots$, de l'autre les horaires $1, 2, \ldots, T$. L'arête it modélise le fait qu'on trie la palette i sur le créneau t. On met comme poids sur it l'opposé de l'amende totale payée si on trie la palette i sur le créneau t (somme des amendes des marchandises qui auraient dues être triées avant t). On définit $b(i) = 1$ pour les palettes $i = 1, \ldots, n$, signifiant qu'il faut trier chaque palette au plus une fois (on ne gagne rien à la trier deux fois). Et $b(t) = k$ pour le créneau t, signifiant qu'on ne peut pas trier plus de k palettes sur un créneau horaire.

6.4. Problème de l'affectation optimale

Une situation classique où apparaît un graphe biparti est le problème de l'affectation. On dit qu'un couplage est *parfait* lorsque tout sommet de G est incident à une arête de M.

Problème de l'affectation :

Données : Un graphe $G = (V, E)$ biparti avec des poids $w : E \to \mathbb{R}$ sur les arêtes.

Demande : Un couplage parfait de poids minimal.

Ce problème apparaît dans quantité de situations. Comme précédemment, les sommets du graphe biparti peuvent être des employés d'une part et des tâches à effectuer de l'autre (on suppose qu'il y a autant d'employés que de tâches). Les poids peuvent modéliser le coût de réalisation de la tâche par un employé donné. On veut réaliser toutes les tâches avec un coût minimal, en supposant que tout employé peut réaliser n'importe quelle tâche.

Théorème 6.4.1. — *On sait résoudre le problème de l'affectation en $O(n(m + n \log n))$.*

Il y a de nombreux algorithmes qui résolvent ce problème avec cette complexité. On peut également appliquer l'algorithme hongrois auquel il a été fait mention ci-dessus. En effet, l'algorithme hongrois fait en fait mieux que trouver un couplage de poids maximum : il trouve, pour un k fixé, le couplage de cardinalité k de plus grand poids. Il suffit alors de multiplier par -1 les poids et de chercher le couplage de cardinalité $n/2$.

6.5. Mariages stables

Nous avons vu ci-dessus des problèmes d'affectation ou de couplage où l'on maximise une quantité globale. Dans de nombreuses situations, par exemple la création de binômes, on veut maximiser dans un certain sens une quantité locale (la satisfaction de chaque binôme). On sent bien qu'en général, on ne peut jamais satisfaire de manière optimale tout le monde, mais on a des notions qui s'en approchent, issues de l'économie ou de la théorie des jeux.

Nous présentons le plus célèbre des résultats de cette famille. Bien des généralisations sont possibles, à l'origine d'une littérature spécialisée abondante, mais elles ne seront pas évoquées ici.

Imaginons la situation suivante. On a m filles et n garçons. Chaque fille a un ordre de préférence (total) entre les différents garçons qu'elle connaît, et qu'il en est de même pour les garçons.

Un ensemble de mariages est représenté par un couplage, il est dit *stable* si lorsqu'une fille, disons Alice, et un garçon, disons Bob, ne sont pas mariés entre eux, alors Alice est mariée à un garçon qu'elle préfère à Bob ou Bob est marié à une fille qu'il préfère à Alice.

Un théorème de Gale et Shapley [**9**] dit alors

Théorème 6.5.1. — *Il existe toujours un couplage stable. De plus, un tel couplage se trouve en* $O(nm)$.

Ce qui est remarquable, c'est que la preuve est simple, algorithmique et peut se raconter sous la forme d'une histoire.

Démonstration. — Chaque matin, chaque garçon invite à dîner la fille qu'il préfère parmi celles qui ne lui ont pas déjà refusé une invitation. Chaque fille qui a été invitée dîne le soir avec le garçon qu'elle préfère parmi ceux qui l'ont invitée ce jour là. L'algorithme se poursuit tant qu'au moins une invitation change. Les couples qui ont dîné entre eux le dernier soir sont mariés.

Pour se convaincre que cet algorithme fournit un mariage stable, il faut d'abord remarquer que :

toute fille qui est invitée un soir à dîner est sûre de dîner le lendemain avec un garçon qui lui plaise au moins autant.

En effet, si une fille, disons Alice, est invitée à dîner par un garçon, disons Bob, c'est que Bob préfère Alice à toutes celles qui ne l'ont pas déja refusé. Alice est donc sûre d'être réinvitée le lendemain par Bob, mais elle sera peut être aussi invitée par de nouveaux garçons qui viennent d'essuyer des refus et pour qui elle est désormais le meilleur choix restant. Peut être que Charlie, l'un de ces nouveaux garçons, lui plait plus que Bob, dans ce cas elle dînera avec Charlie, en congédiant Bob. Dans tous les cas, Alice dîne le lendemain avec un garçon au moins aussi plaisant.

Ensuite, on remarque que

l'algorithme se termine.

En effet, une invitation qui a été refusée ne sera jamais répétée. À chaque étape précédant la fin de l'algorithme, au moins une invitation est refusée. Le nombre d'étapes avant que la liste des invitations ne se stabilise est donc borné par le nombre d'invitations possibles, i.e. par le nombre de filles fois le nombre de garçons.

Enfin, lorsque l'algorithme se termine,

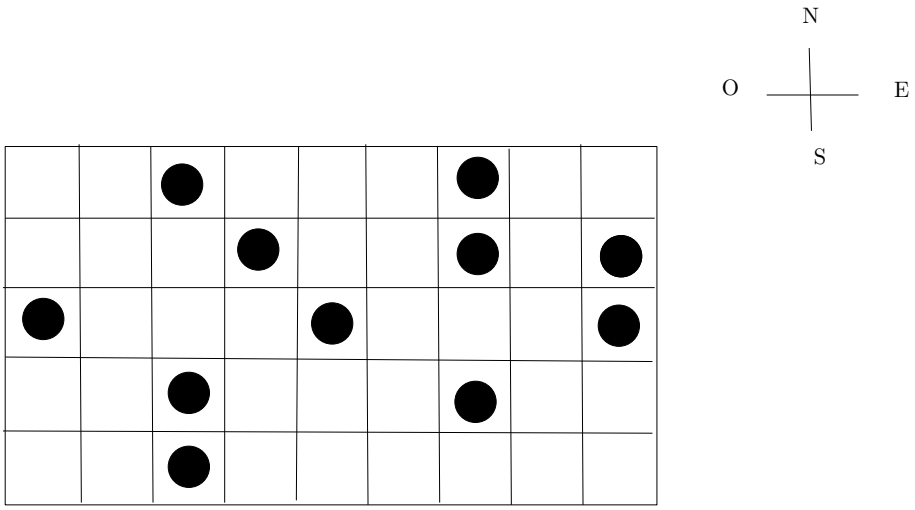


FIG. 3. Des postes d'observation possibles.

les mariages fournissent un couplage stable.

En effet, considérons le mariage obtenu à l'issue de l'algorithme. Si une fille, disons Alice, n'est pas mariée à un garçon, disons Bob, une possibilité est qu'Alice ait un mari qu'elle préfére à Bob. Dans le cas contraire, Bob ne l'a jamais invitée, car en vertu de la question précédente, Alice serait mariée à un garçon au moins aussi plaisant que Bob, contredisant notre hypothèse. Mais si Bob ne l'a jamais invitée, c'est qu'il a une épouse, Corinne, qu'il préfère à Alice. Sinon, le dernier soir, il auraît dû inviter Alice, ou peut être une autre fille, Delphine, qui lui plaît encore plus qu'Alice, mais sûrement pas Corinne, contredisant le fait qu'il est finalement marié avec Corinne. Ainsi, le mariage est stable.

□

Cette modélisation par des mariages stables est utilisée dans de nombreuses situations. Aux Etats-Unis, depuis plus de 50 ans, elle sert dans l'affectation des internes dans les services hospitaliers. En France, depuis peu, certaines écoles préparatoires traitent ainsi les demandes des élèves pour accéder à telle ou telle prépa. De nombreux autres exemples sont connus.

6.6. Exercices

6.6.1. Une caractérisation des graphes bipartis. — Prouver la Proposition 6.1.1.

6.6.2. Théorème de König. — Prouver le Théorème 6.2.3.

6.6.3. Positionnement de postes d'observation. — Considérer le domaine de la Figure 3. Chaque point indique une position possible pour un poste d'observation. On veut que chaque poste d'observation ait la vue complètement dégagée dans les 4 directions cardinales (nord, sud, est, ouest) – deux postes d'observation ouverts ne peuvent donc ni se trouver sur la même ligne, ni se trouver sur la même colonne. On veut en ouvrir un nombre maximum. Proposer un nombre maximum dans le cas particulier de la Figure 3. Prouver la qualité de votre solution. Proposer la méthode générale (est-ce polynomial ?).

CHAPITRE 7

QUE FAIRE FACE À UN PROBLÈME DIFFICILE ?

7.1. Introduction

Jusqu'à présent nous avons vu un certain nombre de problèmes, certains **NP**-durs, d'autres polynomiaux. Pour les problèmes polynomiaux, nous avons discuté les algorithmes possibles pour les résoudre ; pour les autres, rien n'a été indiqué, ce qui pourrait laisser penser qu'on ne sait pas les résoudre. Bien entendu, c'est faux, et heureusement, car de nombreux (la plupart ?) problèmes industriels sont **NP**-durs.

L'objet de ce chapitre est de présenter quelques méthodes générales, indépendantes du problème, que l'on peut employer lorsqu'on est confronté à un problème d'optimisation **NP**-dur, que l'on écrit sous la forme

$$\begin{array}{ll} \text{Min} & f(x) \\ \text{s.c.} & x \in X. \end{array} \tag{21}$$

On suppose X fini, mais très grand.

Un problème **NP**-dur est tel qu'il n'est pas possible (sauf si $\mathbf{P} = \mathbf{NP}$) de trouver en temps polynomial la solution exacte. On peut donc soit relâcher la condition "solution exacte" et vouloir garder un temps polynomial d'exécution, soit relâcher la condition "temps polynomial" et garder le désir d'avoir une solution exacte. La première option conduit aux algorithmes d'approximation, définis dans le premier chapitre. Nous verrons quelques exemples dans les chapitres suivants, mais il n'existe pas de schéma général qui permettent de dériver de tels algorithmes. L'existence même de ces derniers dépend fortement de la nature des problèmes. La seconde option conduit par exemple aux algorithmes de *séparation et évaluation*, ou *branch-and-bound*, dont nous allons présenter le principe dans la Section 7.2. Plusieurs exemples seront donnés aux cours des séances suivantes. Un branch-and-bound suppose l'existence de bonnes bornes sur la solution (borne inférieure si on minimise, borne supérieure si on maximise). Si l'on ne dispose pas de telles bornes, il faut alors recourir à d'autres techniques.

- soit des *heuristiques*, qui sont des algorithmes *ad hoc* pour lesquels le bon sens assure leur bon fonctionnement en général, mais pour lesquels il est difficile de garantir en toute généralité le temps de calcul ou la qualité de la solution. Par définition, une heuristique n'a de sens que pour un problème donné, nous ne verrons pas d'exmple dans ce cours.
- soit des *métaheuristiques*, qui sont des méthodes très générales, en quelques sortes qui fonctionnent toujours, mais pour lesquelles il y a rarement de garanties d'exactitude ou de temps d'exécution. Ce sera l'objet de la Section 7.3.

En général, pour les problèmes industriels, ce sont soit les algorithmes exacts qui sont utilisés, soit les métaheuristiques. Les algorithmes d'approximation sont plutôt développés dans des cadres théoriques. Un algorithme exact sera plutôt utilisé pour des questions stratégiques, une métaheuristique pour des questions tactiques ou opérationnelles.

Si le problème se modélise d'emblée sous la forme d'un programme linéaire simple, éventuellement avec des variables entières, un solver peut être utilisé. Il en existe des libres (LPSOLVE, GLPK, SOPLEX,...) ou des payants, plus performants (CPLEX, XPRESS,...) – voir la liste en fin d'ouvrage – qui sont capables de résoudre rapidement des grands problèmes. On dispose alors d'un algorithme exact pas trop difficile à développer. Sinon, il faut développer une technologie mathématique supérieure à celle des métaheuristiques, comme le branch-and-cut ou la relaxation lagrangienne (qui seront vus plusieurs fois dans la suite de l'ouvrage) ou la génération de colonnes (non abordée dans cet ouvrage). Le niveau mathématique des développeurs, ou l'énergie que l'on souhaite mettre dans le développement, constitue donc également des critères à prendre en compte dans le choix d'une ou de l'autre approche.

7.2. Séparation et évaluation ou branch-and-bound

7.2.1. Description. — On suppose que l'on dispose d'une fonction $\lambda : \mathcal{P}(X) \to \mathbb{R}$ qui a toute partie Y de X, associe $\lambda(Y) \leq \text{ Min }_{x \in Y} f(x)$. La quantité $\lambda(Y)$ est donc une borne inférieure de f sur Y. On supposera de plus que λ se calcule "facilement" (par exemple, en temps polynomial).

L'algorithme maintient

- une collection $\mathcal{Y}$ de parties de X telle que $\bigcup \mathcal{Y}$ contient un minimum de $f(x)$ sur X
- la meilleure solution courante $\tilde{x}$.

Une itération est alors

> Choisir une partie Y de $\mathcal{Y}$.
> Partitionner Y en parties $Y_1, Y_2, \ldots, Y_s$.
> Supprimer Y de $\mathcal{Y}$.
> Faire : pour $i = 1, \ldots, s$, si $\lambda(Y_i) < f(\tilde{x})$, poser $\mathcal{Y} := \mathcal{Y} \cup \{Y_i\}$.

L'idée principale du branch-and-bound réside dans cette dernière étape : il ne sert à rien de conserver la partie Y_i si $\lambda(Y_i) \geq f(\tilde{x})$. En effet, comme $\lambda(Y_i) \leq \text{ Min }_{x \in Y_i} f(x)$, on n'est sûr que sur Y_i on ne parviendra pas à améliorer strictement la valeur de la fonction objectif. Il ne sert donc à rien d'explorer cette partie.

On représente souvent l'exploration de l'espace des solutions par une arborescence, dont les noeuds sont les Y, et les arêtes codent la partition (voir exemple ci-dessous).

On comprend que un algorithme de branch-and-bound marchera d'autant mieux que la borne λ sera bonne. La *branchement*, i.e. l'opération de partition de Y est également importante. Souvent, la structure du problème impose assez naturellement les façons de brancher. Enfin, le choix de la partie Y peut également influencer la qualité de l'algorithme. Deux solutions classiques sont les suivantes :

- *En profondeur d'abord* : brancher toujours sur la dernière partie Y_i ajoutée à $\mathcal{Y}$. On va descendre très vite dans l'arborescence. Cela peut être utile par exemple si on n'a pas de bonne solution réalisable.
- *En largeur d'abord* : brancher toujours sur la partie Y telle que $Y = \text{argmin}_{Y \in \mathcal{Y}} \lambda(Y)$. C'est intéressant si λ est une bonne évaluation par défaut de f.

Nous allons maintenant voir deux exemples très classiques de calcul de borne, et nous appliquerons ensuite l'algorithme sur un exemple très simple, en guise d'illustration.

7.2.2. Quelques exemples de calcul de bornes. — Un grand champ d'application du branch-and-bound est la programmation linéaire en nombres entiers. Nous allons présenter deux techniques de bornes dans ce cadre.

7.2.2.1. Programmation linéaire en nombres entiers. — Contrairement à la programmation linéaire où les variables prennent leurs valeurs dans $\mathbb{R}$, la programmation linéaire en nombres entiers, où les variables prennent leurs valeurs dans $\mathbb{Z}$, n'est pas polynomiale. De façon générale, la programmation linéaire en nombre entiers se modélise, avec A une matrice $m \times n$ à coefficients rationnels, et $\boldsymbol{b} \in \mathbb{Q}^m$ et $\boldsymbol{c} \in \mathbb{Q}^n$

$$\begin{array}{ll} \text{Min} & \boldsymbol{c} \cdot \boldsymbol{x} \\ \text{s.c.} & A\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \in \mathbb{Z}^n. \end{array} \tag{22}$$

Même le cas suivant, où il n'y a qu'une seule contrainte, est **NP**-dur.

$$\begin{array}{lll} \text{Max} & \sum_{i=1}^n c_i x_i & \\ \text{s.c.} & \sum_{i=1}^n w_i x_i \leq W & \\ & x_i \in \{0,1\} & \text{pour } i = 1, \ldots, n, \end{array}$$

avec les w_i et W entiers.

Il s'agit en effet du problème du sac-à-dos, vu au chapitre programmation dynamique.

7.2.2.2. Relaxation continue. — Une borne inférieure à la valeur optimale v_{plne} de (22) s'obtient en relâchant la contrainte d'intégrité, i.e. en ne demandant plus à ce que $\boldsymbol{x}$ soit à coordonnées entières. En effet, considérons

$$\begin{array}{ll} \text{Min} & \boldsymbol{c} \cdot \boldsymbol{x} \\ \text{s.c.} & A\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \in \mathbb{R}^n, \end{array} \tag{23}$$

et notons v_{pl} sa valeur optimale. Toute solution réalisable du programme (22) étant solution réalisable de (23), on a $v_{\text{plne}} \geq v_{\text{pl}}$[(1)].

Exemple : On considère le programme

$$\begin{array}{ll} \text{Max} & x_1 + 5_2 \\ \text{s.c.} & -4x_1 + 2x_2 \leq 3 \\ & 2x_1 + 8x_2 \leq 39 \\ & x_1, x_2 \in \mathbb{Z}. \end{array} \tag{24}$$

L'arborescence est donnée Figure 1. Remarquer que par deux fois, on évite l'exploration d'une sous-arborescence en utilisant la borne : une fois sur le noeud $x_1 \leq 1$, et l'autre fois sur le noeud $x_1 \geq 4$, $x_2 \leq 3$.

7.2.2.3. Relaxation lagrangienne. — Considérons le problème

$$\begin{array}{lll} \text{Min} & f(x) & \\ \text{s.c.} & x \in X & \\ & g_i(x) = 0 & i = 1, \ldots, p \\ & g_i(x) \leq 0 & i = p+1, \ldots, p+q \end{array} \tag{25}$$

[(1)] Rappelons qu'une solution réalisable d'un programme linéaire est un vecteur satisfaisant les contraintes.

où $f, g_1, \ldots, g_{p+q}$ sont des fonctions de $\mathbb{R}^n$ dans $\mathbb{R}$, et X un sous-ensemble non vide de $\mathbb{R}^n$. On suppose que si les fonctions g_i n'étaient pas présentes, le problème serait facile à résoudre. L'idée consiste à dualiser les contraintes g_i en écrivant le lagrangien (voir Sous-section 2.3.5)

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) := f(\boldsymbol{x}) + \sum_{i=1}^{p+q} \lambda_i g_i(\boldsymbol{x}),$$

avec $\boldsymbol{\lambda} \in \Lambda := \mathbb{R}^p \times (\mathbb{R}_+)^q$.

Si on note v la valeur optimale de (25), on a toujours

$$v = \inf_{\boldsymbol{x} \in X} \sup_{\boldsymbol{\lambda} \in \Lambda} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) \geq \sup_{\boldsymbol{\lambda} \in \Lambda} \inf_{\boldsymbol{x} \in X} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = \sup_{\boldsymbol{\lambda} \in \Lambda} \mathcal{G}(\boldsymbol{\lambda}), \tag{26}$$

où

$$\begin{array}{rccc} \mathcal{G} : & \Lambda & \to & \mathbb{R} \cup \{-\infty\} \\ & \mathcal{G}(\boldsymbol{\lambda}) & \mapsto & \inf_{x \in X} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}), \end{array}$$

est la fonction duale. Elle est concave et affine par morceaux (car infimum de fonctions affines). L'idée de la *relaxation lagrangienne* est d'utiliser $\sup_{\boldsymbol{\lambda} \in \Lambda} \mathcal{G}(\boldsymbol{\lambda})$ comme borne inférieure. La question est donc de savoir calculer cette quantité. Pour cela on peut utiliser des techniques de l'optimisation non-différentiable, une méthode de sur-gradient par exemple [(2)], qui nécessite

[(2)] ou alors par *la méthode des faisceaux*, plus efficace mais qui dépasse le cadre de ce cours, voir **[5]**.

FIG. 1. Un arbre de branch-and-bound.

principalement de savoir calculer un sur-gradient en tout point $\boldsymbol{\lambda}$. Cette méthode est décrite à la fin de cette section.

Rappelons qu'un *sur-gradient* $\boldsymbol{p}$ de $\mathcal{G}$ au point $\boldsymbol{\lambda}$ est tel que

$$\mathcal{G}(\boldsymbol{\mu}) - \mathcal{G}(\boldsymbol{\lambda}) \leq \boldsymbol{p}^T(\boldsymbol{\mu} - \boldsymbol{\lambda}) \quad \text{pour tout } \boldsymbol{\mu} \in \Lambda.$$

En particulier, si $\mathcal{G}$ est dérivable, le sur-gradient coïncide avec son gradient.

Heureusement, cela se fait facilement, grâce à la proposition suivante.

Proposition 7.2.1. — *Si X est fini, alors $(g_i(\boldsymbol{x}))_{i=1,\dots,p+q}$ est un sur-gradient de $\mathcal{G}$ au point $\boldsymbol{\lambda}$ dès que $\boldsymbol{x}$ réalise* $\inf_{x\in X} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda})$.

Démonstration. — Soit un tel $\boldsymbol{x}$. Pour tout $\boldsymbol{\mu}$, on a $\mathcal{G}(\boldsymbol{\mu}) - \mathcal{G}(\boldsymbol{\lambda}) \leq \mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}) - \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = \sum_{i=1}^{p+q} g_i(\boldsymbol{x})(\mu_i - \lambda_i)$. □

Un des intérêts de la relaxation lagrangienne est contenu dans le théorème suivant (preuve omise).

Théorème 7.2.2. — *Pour un programme linéaire en nombres entiers, la borne obtenue par relaxation lagrangienne est toujours au moins aussi bonne que celle obtenue par relaxation linéaire.*

Cela dit, il existe beaucoup de cas où ces deux bornes coincident. Cela n'empêche pas la relaxation lagrangienne d'être intéressante (comme dans l'exemple ci-dessous), car le calcul de $\sup_{\boldsymbol{\lambda}\in\Lambda} \mathcal{G}(\boldsymbol{\lambda})$ peut-être plus rapide que la résolution du programme linéaire.

Exemple : Un exemple très classique est celui du plus court chemin avec contrainte de temps.

Soit $D = (V, A)$ un graphe orienté, avec deux sommets s et t. A chaque arc a est associé un coût $c_a \geq 0$ et un temps $t_a \geq 0$. On veut trouver le s–t chemin le moins coûteux qui mette un temps inférieur à T. Ce problème se résout bien avec un branch-and-bound, utilisant des bornes fournies par la relaxation lagrangienne. X est alors l'ensemble des s–t chemins, $f(\boldsymbol{x}) = \sum_{a\in A} c_a x_a$ et l'on a une seule contrainte du type g_i, c'est $g(\boldsymbol{x}) := \sum_{a\in A} t_a x_a \leq T$.

Ecrivons $\mathcal{G}(\lambda)$, défini pour $\lambda \geq 0$.

$$\mathcal{L}(\boldsymbol{x}, \lambda) = \sum_{a\in A} c_a x_a + \lambda \left(-T + \sum_{a\in A} t_a x_a \right).$$

Donc

$$\mathcal{G}(\lambda) = -T\lambda + \min_{x\in X} \sum_{a\in A} (c_a + t_a\lambda) x_a.$$

Le calcul de $\mathcal{G}(\lambda)$ se fait donc par un calcul de plus court chemin où les coûts des arcs sont non plus les c_a mais les $c_a + \lambda t_a$. Ce calcul peut se faire par l'algorithme de Dijkstra car les poids sont positifs ou nuls, et la solution $\boldsymbol{x}$, injectée dans g donne la valeur $-T + \sum_{a\in A} t_a x_a$, cette dernière quantité étant alors le sur-gradient de $\mathcal{G}$ en λ (d'après la Proposition 7.2.1). On sait donc calculer les sur-gradients de $\mathcal{G}$, donc on sait maximiser $\mathcal{G}$ sur $\Lambda = \mathbb{R}_+$, donc on sait calculer de bonnes bornes inférieures pour notre problème, et on peut donc faire un branch-and-bound.

Méthode de sur-gradient. — Posons $\Lambda = \{(\lambda_1, \ldots, \lambda_p, \lambda_{p+1}, \ldots, \lambda_{p+q}) \in \mathbb{R}^{p+q} : \lambda_i \geq 0 \text{ pour } i \geq p+1\}$. Soit P_Λ la projection dans Λ définie par $P_\Lambda(\boldsymbol{\lambda}) = (\lambda_1, \ldots, \lambda_p, \lambda_{p+1}^+, \ldots, \lambda_{p+q}^+)$, avec la notation $t^+ = \max(0, t)$. Supposons que l'on veuille maximiser $\mathcal{G}(\boldsymbol{\lambda})$ sur Λ, avec $\mathcal{G}$ concave. L'algorithme de sur-gradient consiste à construire la suite

$$\boldsymbol{\lambda}_{k+1} = P_\Lambda\left(\boldsymbol{\lambda}_k + \frac{\rho_k}{||\boldsymbol{p}_k||}\boldsymbol{p}_k\right),$$

où $\boldsymbol{\lambda}_0$ est choisi arbitrairement dans Λ, où $\boldsymbol{p}_k$ est un sur-gradient de $\mathcal{G}$ au point $\boldsymbol{\lambda}_k$, et où (ρ_k) est une suite prédéterminée de réels strictement positifs telle que

$$\lim_{k \to +\infty} \rho_k = 0 \text{ et } \sum_{k=0}^{+\infty} \rho_k = +\infty.$$

Lorsque $\boldsymbol{p}_k$ vaut 0, l'algorithme s'arrête, $\boldsymbol{\lambda}_k$ est alors le maximum de $\mathcal{G}$.

On peut montrer que l'algorithme converge vers le maximum de $\mathcal{G}$, mais la convergence est lente (le pas de convergence devant nécessairement tendre vers 0).

7.3. Métaheuristiques

Nous allons décrire quelques métaheuristiques, qui sont des méthodes générales d'exploration de l'espace des solutions réalisables, qui peuvent être décrites indépendamment du problème. Les métaheuristiques peuvent parfois donner de très bons résultats ou constituer la seule arme pour attaquer un problème ; il est donc nécessaire de les connaître lorsqu'on fait de la recherche opérationnelle. Cela dit, leur implémentation dépend d'un certain nombre de paramètres dont les valeurs nécessitent beaucoup d'expérimentations. De plus, ces algorithmes n'ont pas toujours des garanties sur la durée de l'exécution ou sur la qualité de la solution trouvée.

7.3.1. Recherche locale. —

7.3.1.1. Le principe. — On cherche toujours à résoudre (21), cette fois avec X supposé fini.

L'idée de la *recherche locale* est la suivante. On suppose connue une bonne solution réalisable de départ, trouvée par une heuristique quelconque. Ensuite, on essaie d'améliorer la solution courante en procédant à des "modifications" locales.

Plus précisément,

1. il faut définir sur X un graphe implicite (ce graphe ne sera jamais explicitement codé en machine), une solutions réalisable étant connectée à une autre solution réalisable si on passe de la première à la seconde par une modification autorisée. Pour une solution réalisable x, on note $\Gamma(x)$ l'ensemble de ses voisins dans le graphe implicite, i.e. l'ensemble des solutions réalisables que l'on peut atteindre de x par une modification locale. Une condition nécessaire pour pouvoir trouver l'optimum global est que la solution réalisable de départ est dans la même composante connexe du graphe implicite que l'optimum cherché.
2. Il faut aussi définir à quelle condition est-ce qu'on modifie la solution courante. La solution consistant à modifier la solution courante si la modification a amélioré la valeur de f est naïve : en effet, un tel algorithme va "plonger" vers un optimum local, qu'il ne va plus quitter.

L'implémentation de la version naïve est donc : On part d'une solution réalisable x. Ensuite on répète

Si $\{y \in \Gamma(x) : f(y) < f(x)\}$ est non vide, choisir $y \in \Gamma(x)$ tel que $f(y) < f(x)$ et poser $x := y$.

Puisqu'on a supposé X fini, l'algorithme se termine sur un $x \in X$, qui sera un minimum local de f.

Pour éviter de se retrouver bloqué sur n'importe quel minimum local, deux stratégies classiques sont employées, la *méthode tabou* et le *recuit simulé*, détaillées plus bas.

7.3.1.2. Un exemple : la coloration de graphe. — Considérons le problème de la coloration de graphe (voir section 2.1.1.3). On veut colorer les sommets d'un graphe $G = (V, E)$ de manière à ce que deux sommets voisins ont des couleurs distinctes, et en utilisant un nombre minimum de couleurs. C'est un problème **NP**-difficile.

Une recherche locale pour ce problème consiste à définir l'ensemble X des solutions réalisables comme l'ensemble des colorations propres et à considérer deux colorations propres comme voisines si elles diffèrent uniquement par la couleur d'un sommet. La fonction objectif est le nombre de couleurs utilisées. Cette approche a un inconvénient : la plupart des modifications n'implique pas un changement de la valeur de la fonction objectif. Avec la définition stricte de l'algorithme naïf ci-dessus, on se retrouve très vite bloqué sur des minima locaux. Même en acceptant des modifications à valeur constante de la fonction objectif, ou des modifications détériorant cette valeur (comme dans les métaheuristiques décrites ci-dessous), cela reste problématique car la fonction objectif donne peut d'information sur la direction à prendre.

Une façon plus efficace de résoudre le problème de la coloration ([**16**]) est de fixer un entier k et de résoudre le problème consistant à minimiser $\sum_{i=1}^{k} |E[V_i]|$ sur l'ensemble des partitions $V_1 \cup \ldots \cup V_k$ de V. Rappelons que $E[V_i]$ est l'ensemble des arêtes de G ayant leurs deux extrémités dans V_i. Si l'on parvient à atteindre 0, on aura trouvé une coloration en au plus k couleurs : chaque partie V_i peut être vue comme l'ensemble des sommets de couleur i ; comme alors $E[V_i] = \emptyset$, on est sûr de ne pas avoir deux sommets de même couleur adjacent. La recherche locale résout extrêmement bien cette tâche : l'ensemble X des solutions réalisables est alors l'ensemble des partitions $V_1 \cup \ldots \cup V_k$ de V. La fonction objectif est $\sum_{i=1}^{k} |E[V_i]|$. On accepte de passer de passer d'une solution à une autre si le changement consiste modifier la couleur (l'appartenance à l'un des V_i) d'un sommet appartenant à une arête dont les deux extrémités sont de couleur identique. Une dichotomie sur k permet de trouver *in fine* le nombre chromatique de G.

7.3.1.3. Méthode tabou. — La méthode tabou consiste à maintenir une liste L de solutions réalisables "tabous", dont la taille, constante, est un paramètre de l'algorithme. Si l est cette taille[(3)], on commence par une solution réalisable x tabou. On choisit une liste L qui contient l autres solutions réalisables et on pose $x^* := x$ (x^* est la meilleure solution courante). Ensuite, on répète

- choisir y minimisant $f(y)$ sur $\Gamma(x) \setminus L$
- enlever le premier élément de L, et ajouter x à la fin de L,
- poser $x := y$,
- si $f(x) < f(x^*)$, poser $x^* := x$.

Puisqu'on a supposé X fini, l'algorithme se termine avec $x^* \in X$, qui sera un minimum local de f. Rien n'assure qu'on se retrouve sur un optimum global.

[(3)] par exemple $l = 7$.

7.3.1.4. Recuit simulé. — Le recuit simulé imite la pratique du recuit en métallurgie qui consiste à alterner de lents refroidissements et de réchauffages (recuits) pour minimiser l'énergie d'un matériau. L'algorithme maintient une température T qui va tendre vers 0, et les transitions d'une solution réalisable à une solution réalisable voisine auront lieu avec une probabilité dépendant de la différence de la valeur prise par f et de la température. Plus précisément, si on a choisi y voisin de x, la probabilité de passer effecivement de x à $y \in \Gamma(x)$ sera, par analogie avec la physique[(4)]

$$\min(1, e^{-\frac{f(y)-f(x)}{T}}).$$

Une fois fixée au préalable la suite T_k des temptératures (telle que $T_k \to 0$ quand $k \to \infty$), on peut écrire l'algorithme. On part d'une solution réalisable x. Ensuite, on répète

> Tirer un voisin y uniformément dans $\Gamma(x)$. Faire $x := y$ avec la probabilité $\min(1, e^{-\frac{f(y)-f(x)}{T_k}})$. Poser $k := k + 1$.

Plusieurs auteurs ont donné des conditions suffisantes pour que x tende en probabilité vers l'optimum global. Par exemple, Hajek [**15**] a donné une condition nécessaire et suffisante assez légère pour cette convergence en probabilité avec T_k de la forme $\frac{\kappa}{\log 2+k}$.Cette condition met en jeu les notions d'irréductibilité et de faible réversibilité des chaînes de Markov (attention, elles sont ici non homogènes !).

En pratique, on aime bien avoir T_k de la forme

$$T_k := T_0 \beta^k$$

[(5)] ou de la forme

$$T_k := \frac{1}{\alpha + sk}$$

avec α et s des paramètres bien choisis (il faut pas mal expérimenter).

7.3.2. Algorithmes évolutionnaires, colonie de fourmi, etc.— Il y a bien d'autres exemples de métaheuristiques. L'idée des *algorithmes évolutionnaires* est d'identifier les solutions réalisables à des individus et de coder ces solutions réalisables sur les chromosomes des individus. On autorise les individus à se croiser avec une probabilité d'autant plus grande que les individus sont de qualité. De plus ces individus peuvent muter. On part alors d'une population de taille fixée d'individus, et on laisse évoluer le tout. Ce qui est crucial ici, c'est le codage d'une solution réalisable sur les chromosomes et la nature du croisement, qui doit maintenir les avantages compétitifs de ses parents.

L'*algorithme de colonie de fourmis*, tout comme le recuit simulé ou les algorithmes évolutionnaires, cherche à imiter la nature dans la façon dont elle résout ses problèmes d'optimisation. Les fourmis pour trouver leur nourriture explorent au hasard l'espace des solutions, en ayant une toute petite vue locale, mais grâce à des échanges d'information (phéromones), elles parviennent à trouver la source de nourriture et à trouver le trajet le plus court. En effet, en marquant leur chemin vers la nourriture, les plus courts chemins seront plus attractifs que les longs et se trouveront renforcés. L'algorithme va donc simuler l'exploration de l'espace des solutions par des fourmis, qui marqueront leur passage. Plus une transition sur X est choisie, plus forte sera la probabilité qu'elle soit choisie par une fourmi.

[(4)] Pour un système fermé, à la température T, la probabilité d'être dans un état d'énergie E est de la forme $\frac{e^{\frac{E}{k_B T}}}{Z}$ où Z est la fonction de partition, k_B la constante de Boltzmann.

[(5)] avec par exemple $\beta = 0,93$.

7.4. Exercices

7.4.1. Borne inférieure par la relaxation lagrangienne. — Prouver l'inégalité (26).

Gestion dynamique de stock, cas à plusieurs biens. — Cet exercice fait suite à l'exercice 5.3.9.

Supposons maintenant qu'il n'y ait pas un seul bien, mais K biens, indicés par $k = 1, 2, \ldots, K$. Pour chaque bien k, on a une demande d_{kt} sur chaque période t. On satisfait la demande en produisant x_{kt} de bien k sur la période t et/ou en prélevant une certaine quantité sur le srock $y_{k(t-1)}$ de la période $t-1$. On suppose que la production en période t pour le bien k ne peut pas excéder P_{kt}. De plus, on ne possède qu'une machine, et donc sur une période on ne peut produire qu'un seul type de bien. Le coût de production unitaire du bien k en période t est noté p_{kt}, et celui de stockage unitaire s_{kt}.

1. Ecrire la dynamique du stock dans ce cas-là.

2. En introduisant une variable z_{kt} qui indique si le bien k est produit sur la période t, proposer une contrainte linéaire (indicée par t) qui empêche la production de plusieurs biens sur la période t.

3. Ecrire une contrainte (indicée par k et t) qui limite la production du bien k en période t, en tenant compte du fait que si un autre bien $k' \neq k$ est produit, alors la production du bien k doit être nulle.

4. Montrer que ce problème peut se modéliser comme un programme linéaire mixte en nombres entiers (mixte signifie que toutes les variables ne sont pas contraintes à être entières).

Dans une approche par branch-and-bound, on va chercher des bornes inférieures. Une solution est de procéder par relaxation lagrangienne.

5. Montrer qu'en relaxant les bonnes contraintes, le calcul des bornes inférieures par la relaxation lagrangienne se ramène à des calculs de gestion de stock à un seul bien, et expliquer comment calculer ces bornes dans le cas où les P_{kt} sont suffisament grands.

6. Si les P_{kt} ne sont pas suffisament grands, le problème à un seul bien est **NP**-dur. Proposer une solution pour le calcul de la borne inférieure par la programmation dynamique qui garde une complexité raisonnable.

7.4.2. Extraction de mine à ciel ouvert – cas dynamique. — On se remet dans le contexte de l'Exercice 5.3.10. On essaie cette fois de prendre en compte l'aspect dynamique. On cherche à déterminer la séquence d'extraction des *blocs* dans les mines à ciel ouvert. La mine est assimilée à une collection de n blocs numérotés de 1 à n, le bloc i ayant une masse m_i. On se place dans un contexte à horizon de temps fini. Le temps est discrétisé est assimilé à des années $\tau = 1, \ldots, T$. Un bloc i extrait l'année τ entraîne un profit de $c_{i,\tau}$ dollars. Cette quantité $c_{i,\tau}$ peut éventuellement être négative si l'extraction du bloc i coûte plus qu'il ne rapporte. La dépendance en temps permet de tenir compte du taux d'actualisation, des tendances des cours boursiers, etc.

A l'année τ donnée, on peut extraire plusieurs blocs, mais pas plus d'une masse totale M_τ.

Les blocs ne peuvent pas être extraits dans n'importe quel ordre : pour chaque bloc i, on connaît l'ensemble Y_i des blocs qu'il faut avoir extraits pour pouvoir extraire le bloc i. Si le bloc i est extrait au cours de l'année τ, tout bloc de Y_i doit être extrait au cours de l'année τ ou

avant. L'objectif est de proposer un sous-ensemble de blocs à extraire et les années auxquelles extraire ces blocs de manière à maximiser le profit total (on appelle cela un *plan d'extraction*).

On suppose maintenant que les $c_{i,\tau}$ dépendent bien du temps. On parle alors du *cas dynamique*. Etant donné un plan d'extraction, on note $x_{i,\tau} = 0$ si le bloc i est extrait strictement après l'année τ et $x_{i,\tau} = 1$ sinon. On tient à nouveau compte de la contrainte de masse.

1. Proposez un programme linéaire en nombres entiers qui modélise le cas dynamique, en utilisant les variables $x_{i,\tau}$. On posera de plus $x_{i,0} = 0$ pour tout i. Justifiez votre réponse.

En pratique, ces programmes linéaires peuvent avoir un grand nombre de variables (le produit NT peut être grand). On va chercher dans la suite des méthodes pour améliorer les temps de calculs lorsqu'on donne un tel programme linéaire à un solver.

2. Soit i^* et τ^* tels que $m_{i^*} + \sum_{k \in Y_{i^*}} m_k > \sum_{\tau'=1}^{\tau^*} M_{\tau'}$. On dit alors que (i^*, τ^*) est une *bonne paire*. Expliquez pourquoi toute solution réalisable du programme linéaire en nombres entiers de 1. satisfait $x_{i^*,\tau^*} = 0$ lorsque (i^*, τ^*) est une bonne paire.

3. Considérons deux blocs i^* et j^* et une année τ^* tels que $\sum_{k \in Y_{i^*} \cup Y_{j^*} \cup \{i^*\} \cup \{j^*\}} m_k > \sum_{\tau'=1}^{\tau^*} M_{\tau'}$. On dit alors que (i^*, j^*, τ^*) est un *bon triplet*. Expliquez pourquoi toute solution réalisable du programme linéaire en nombres entiers de 1. satisfait $x_{i^*,\tau^*} + x_{j^*,\tau^*} \leq 1$ lorsque (i^*, j^*, τ^*) est un bon triplet (on dira que cette contrainte est induite par le bon triplet).

4. Supposons que le solver fonctionne par un branch-and-bound qui utilise les bornes de la relaxation continue (ou linéaire). On fixe à 0 les variables indicées par une bonne paire. On ajoute au programme linéaire en nombres entiers de 1. plusieurs contraintes induites par des bons triplets. Expliquez pourquoi cela ne change pas la solution optimale du programme linéaire en nombres entiers, et en quoi cela va améliorer les temps de calculs (au moins pour les grandes instances).

Deuxième PARTIE II

PROBLÉMATIQUES

CHAPITRE 8

REMPLISSAGE DE CONTENEURS

Le thème de ce chapitre est le suivant : on a des objets et des conteneurs, comment remplir au mieux ? Ecrit comme cela, le problème est assez imprécis. Nous allons nous focaliser sur deux problèmes particuliers qui rentrent dans la catégorie des problèmes de remplissage, le problème du *sac-à-dos* et celui du *bin-packing*. Le problème du sac-à-dos a déjà été vu au Chapitre 3, mais allons discuter d'autres aspects de ce problème.

Le problème du sac-à-dos dans sa version la plus simple peut se décrire informellement de la manière suivante : on a des objets de poids et de valeur variable ; on dispose d'un seul conteneur (le sac-à-dos) qui est muni d'une contrainte de poids ; remplir le conteneur de manière à maximiser la valeur des objets stockés.

Le problème du bin-packing dans sa version la plus simple peut se décrire informellement de la manière suivante : on a des objets de taille variée et un seul type de conteneur ; trouver le nombre minimum de conteneur permettant de tout stocker.

Ces problèmes ont des applications directes dans le domaine de la logistique : stocker des produits, remplir des camions, etc.

8.1. Sac-à-dos

8.1.1. Le problème. — De façon formelle, le problème du sac-à-dos s'écrit

Problème du sac-à-dos

Données : des entiers n, $w_1, \ldots, w_n$ et W, et des réels $c_1, \ldots, c_n$.

Tâche : trouver un sous-ensemble $S \subseteq \{1, \ldots, n\}$ tel que $\sum_{j \in S} w_j \leq W$ et $\sum_{j \in S} c_j$ est maximum.

Cela peut s'interpréter de la manière suivante : W est la charge maximale du conteneur, w_i est le poids de l'objet i, et c_i sa valeur. On a n objets, mettre dans le conteneur un sous-ensemble S d'objets de valeur maximale, tout en respectant la contrainte de poids.

Les applications en logistique sont évidentes. Mais il existe bien d'autres domaines où ce problème se retrouve. Par exemple, en finance : on a un budget fini W, on a des produits financiers i coûtant chacun w_i et rapportant c_i sur l'année à venir ; maximiser le profit.

Comme d'habitude, commençons par cerner la complexité du problème.

Théorème 8.1.1. — *Le problème du sac-à-dos est* **NP***-difficile.*

Nous avons déjà vu au Chapitre 3 qu'une façon commode de résoudre ce problème, si les w_i sont entiers (ce à quoi on peut toujours se ramener en changeant l'unité de mesure) et si W n'est pas trop grand, passe par l'utilisation de la programmation dynamique, laquelle fournit un algorithme pseudo-polynomial en $O(nW)$. Nous allons voir aussi une autre approche exacte par branch-and-bound, qui elle devient utile quand W est grand. Enfin, nous allons exhiber un algorithme polynomial simple de 1/2-approximation.

8.1.2. Formulation sous forme d'un programme linéaire et branch-and-bound. — On peut modéliser le problème du sac-à-dos sous forme d'un programme linéaire en nombres entiers.

$$\begin{array}{ll} \max & \sum_{j=1}^n c_j x_j \\ \text{s.c.} & \sum_{j=1}^n w_j x_j \leq W \\ & x_j \in \{0,1\} \qquad \text{pour tout } j \in \{1,\ldots,n\} \end{array}$$

Remarquons que dans un sens ce programme linéaire en nombres entiers est le plus simple possible : chaque variable ne peut prendre que deux valeurs, et il n'y a qu'une seule contrainte. Cela montre en passant que la programmation linéaire en nombres entiers est **NP**-difficile, même dans ses versions les plus simples.

La relaxation continue fournit une borne supérieure naturelle à la solution du programme précédent

$$\begin{array}{ll} \max & \sum_{j=1}^n c_j x_j \\ \text{s.c.} & \sum_{j=1}^n w_j x_j \leq W \\ & 0 \leq x_j \leq 1 \qquad \text{pour tout } j \in \{1,\ldots,n\} \end{array}$$

Pour la calculer (pour faire du branch-and-bound par exemple – voir ci-dessous), on pourrait bien sûr faire appel à l'algorithme du simplexe ou à l'algorithme des points intérieurs. Mais il existe un algorithme très simple, glouton, qui calcule la solution optimale du relâché continu.

On suppose que $\sum_{j=1}^n w_j > W$, sinon, le problème du asc-à-dos est trivial. Ensuite on fait

Classer les objets de façon à ce que
$$\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \ldots \geq \frac{c_n}{w_n}.$$
Poser $x_1 = x_2 = \ldots = x_{\bar{j}} := 1$ avec $\bar{j}$ le plus grand entier tel que $\sum_{j=1}^{\bar{j}} w_j \leq W$.
Poser $x_{\bar{j}+1} := \frac{W - \sum_{j=1}^{\bar{j}} w_j}{w_{\bar{j}+1}}$.
Poser $x_{\bar{j}+2} = \ldots = x_n := 0$.

Lemme 8.1.2. — *Un tel $\boldsymbol{x}$ est solution optimale du relâché continu.*

Eléments de preuve. — On prouve d'abord que si $\boldsymbol{x}$ est optimal avec $x_i < 1$, $x_k > 0$ et $i < k$, alors $c_i/w_i = c_k/w_k$. Par conséquent, il existe une solution optimale avec un indice $\bar{j}$ tel que $x_i = 1$ pour tout $i < \bar{j}$ (si $\bar{j} \geq 2$) et tel que $x_i = 0$ pour tout $i > \bar{j}$ (si $\bar{j} \leq n-1$). La conclusion est alors claire. □

On sait donc calculer une borne d'assez bonne qualité (relâché continu) en $O(n \log(n))$. Cette borne permet de mettre en place un branch-and-bound. Pour le branchement, c'est la technique usuelle pour les programmes linéaires en $\{0,1\}$: fixer certaines variables à 0 et d'autres à 1.

Cette borne permet aussi de décrire un algorithme d'approximation, comme nous allons le voir.

8.1.3. Algorithme d'approximation. — La borne précédente permet de décrire un algoritme 1/2-approximatif.

En effet, reprenons le $\bar{j}$ calculé par l'algorithme qui résout le relâché continu (ci-dessus). Supposons que pour tout j on a $w_j \leq W$ (sans perte de généralité). Alors $\sum_{j=1}^{\bar{j}+1} c_j$ est une borne supérieure sur la solution optimale. $\{1, \ldots, \bar{j}\}$ et $\{\bar{j}+1\}$ sont deux solutions réalisables, l'une des deux étant par conséquent au moins de valeur $\geq \frac{1}{2} \sum_{j=1}^{\bar{j}+1} c_j$, i.e. de valeur $\geq \frac{1}{2} OPT$.

En réalité, on peut faire beaucoup mieux : Ibarra et Kim [**17**] ont montré en 1975 que

Théorème 8.1.3. — *Pour tout $\epsilon > 0$ fixé, il existe un algorithme polynomial en $O(n^{2\frac{1}{\epsilon}})$ qui donne une solution $\frac{1}{1+\epsilon}$-approximatif.*

8.2. Bin-packing

8.2.1. Le problème. — Le problème du bin-packing traite du cas où l'on a des objets de tailles variables et un seul type de conteneurs, et où l'on se demande comment utiliser un nombre minimum de conteneurs pour ranger tous les objets. En toute généralité, on peut aussi prendre en compte la forme des objets (on parle alors de bin-packing 2D ou de bin-packing 3D), des incompatibilités, etc.

Ce problème peut être parfois appelé aussi *cutting-stock*. En effet, les problèmes de découpes (de pièces de textile, métal, etc.), où l'on cherche à minimiser les pertes, se modélisent de façon similaire.

Ici, on se limite au cas le plus simple, 1D. C'est un cas déjà très utile en pratique, puisqu'il peut fournir des bornes, être utiliser en sous-routines, ou être appliquer tel quel (nombre min de CD-rom pour stocker le contenu d'un disque dur, découper des planches de longueurs variables dans des grandes planches de longueur fixée, découpe dans des bandes de tissu, etc.).

Le problème s'écrit alors formellement

Problème du bin-packing

Données : des nombres positifs ou nuls $a_1, \ldots, a_n \leq 1$.

Tâche : trouver un entier naturel k et une affectation $f : \{1, \ldots, n\} \to \{1, \ldots, k\}$ avec $\sum_{i:\, f(i)=j} a_i \leq 1$ pour tout $j \in \{1, \ldots, k\}$ tels que k soit minimum.

Théorème 8.2.1. — *Le problème du bin-packing est* **NP***-difficile.*

Non seulement, il est **NP**-difficile, mais tout comme pour les problèmes d'ordonnancement d'atelier, il existe encore de nombreuses instances de "petite" taille non résolue, ce qui justifie et motive largement des travaux de recherche dans ces domaines. Par exemple, considérons l'instance suivante, dont on ne connaît pas à ce jour la solution optimale[(1)].

taille de la boîte : 150
nombre d'objets : 120
taille des objets : 100 22 25 51 95 58 97 30 79 23 53 80 20 65 64 21 26 100 81 98 70 85 92 97 86 71 91 29 63 34 67 23 33 89 94 47 100 37 40 58 73 39 49 79 54 57 98 69 67 49 38 34 96 27 92

[(1)]source : la page web du professeur Eric Taillard http ://mistic.heig-vd.ch/taillard/

82 69 45 69 20 75 97 51 70 29 91 98 77 48 45 43 61 36 82 89 94 26 35 58 58 57 46 44 91 49 52 65 42 33 60 37 57 91 52 95 84 72 75 89 81 67 74 87 60 32 76 85 59 62 39 64 52 88 45 29 88 85 54 40 57

meilleure solution connue : 51 boîtes.

Le défi est de trouver une solution en moins de 51 boîtes, ou alors de parvenir à montrer que 51 boîtes est la solution optimale. Nous allons décrire maintenant quelques algorithmes d'approximation classique (NEXT-FIT, FIRST-FIT, FIRST-FIT DEACRISING), puis nous verrons les approches branch-and-bound.

8.2.2. Algorithmes d'approximation. — On a la borne suivante, qui sera utile pour les algorithmes d'approximation

$$\left\lceil \sum_{i=1}^{n} a_i \right\rceil \leq OPT, \tag{27}$$

où OPT désigne la solution optimale au problème du bin-packing.

Cette équation se montre en numérotant les boîtes de 1 à OPT dans la solution optimale. Ensuite, on note $o(j)$ l'ensemble des indices i d'objet tels que l'objet i soit dans la boîte j. Les $o(j)$ forment une partition de $\{1, \ldots, n\}$. On a donc

$$\sum_{j=1}^{OPT} 1 \geq \sum_{j=1}^{OPT} \sum_{i \in o(j)} a_i = \sum_{i=1}^{n} a_i.$$

Le fait que OPT soit entier permet de conclure (ajout des parties entières).

8.2.2.1. NEXT-FIT. — Je prends les objets les uns après les autres. Dès que l'objet i ne peut pas entrer dans la boîte courante, je passe à une nouvelle boîte.

De façon plus formelle :

Commencer par $k := 1$, $i := 1$, et $S := 0$. Répéter

> Si $S + a_i > 1$, faire $k := k + 1$ and $S := 0$. Sinon, faire $f(i) := k$, $S := S + a_i$ et $i := i + 1$.

On a

Théorème 8.2.2. — *NEXT-FIT fournit une solution SOL telle que*

$$SOL \leq 2OPT - 1.$$

Démonstration. — On veut prouver que le k fournit par NEXT-FIT est tel que $k \leq 2OPT - 1$.

On utilise la borne (27) $\lceil \sum_{i=1}^{n} a_i \rceil \leq OPT$. On va donc montrer que $k \leq 2 \lceil \sum_{i=1}^{n} a_i \rceil - 1$.

Pour $j = 1, \ldots, \lfloor \frac{k}{2} \rfloor$ on a

$$\sum_{i:\, f(i) \in \{2j-1, 2j\}} a_i > 1,$$

par définition de l'algorithme NEXT-FIT.

En sommant :

$$\left\lfloor \frac{k}{2} \right\rfloor < \sum_{i=1}^{n} a_i,$$

qui peut se réécrire

$$\frac{k-1}{2} \leq \left\lceil \sum_{i=1}^{n} a_i \right\rceil - 1,$$

ce qu'on voulait montrer. □

8.2.2.2. FIRST-FIT. — Je prends les objets les uns après les autres. Je mets l'objet i dans la boîte de plus petit rang où il peut entrer.

De façon plus formelle,

poser $i := 1$. Répéter :

$$\text{poser } f(i) := \min\left\{ j \in \mathbb{N} : \textstyle\sum_{h<i:\, f(h)=j} a_h + a_i \leq 1 \right\}; \text{ poser } i := i+1.$$

Poser $k := \max_{i \in \{1,\dots,n\}} f(i)$.

Théorème 8.2.3. — *FIRST-FIT fournit une solution SOL telle que*

$$SOL \leq \left\lceil \frac{17}{10} OPT \right\rceil.$$

8.2.2.3. FIRST-FIT DECREASING. — Je trie d'abord les objets par a_i décroissant. Puis j'applique FIRST-FIT.

Théorème 8.2.4. — *FIRST-FIT DECREASING fournit une solution SOL telle que*

$$SOL \leq \frac{3}{2} OPT.$$

L'avantage des deux premiers algorithmes sur ce dernier est qu'ils peuvent fonctionner *on-line*, ce qui signifie qu'on peut les faire tourner lorsque les objets arrivent les uns après les autres et qu'on ne connaît pas la taille des objets futurs.

8.2.3. Branch-and-bound. —

8.2.3.1. Formulation PLNE. — Pour l'approche branch-and-bound, la formulation sous forme d'un programme linéaire va s'avérer utile.

On suppose que l'on a K boîtes disponibles. Le problème du bin-packing peut alors s'écrire

$$\begin{array}{lll} \min & \sum_{j=1}^{K} z_j & \\ \text{s.c.} & \sum_{j=1}^{K} y_{ji} = 1 & \text{pour } i = 1,\dots,n \\ & \sum_{i=1}^{n} a_i y_{ji} \leq z_j & \text{pour } j = 1,\dots,K \\ & y_{ji}, z_i \in \{0,1\} & \text{pour } i = 1,\dots,n;\ j = 1,\dots,K \end{array} \tag{28}$$

où $z_j = 1$ si la boîte j est utilisée et $y_{ji} = 1$ si l'objet i est mis dans la boîte j.

8.2.3.2. Relaxation continue. — Comme d'habitude, la borne la plus naturelle s'obtient par relaxation continue :

$$\begin{array}{lll} \min & \sum_{j=1}^{K} z_j & \\ \text{s.c.} & \sum_{j=1}^{K} y_{ji} = 1 & \text{pour } i = 1,\dots,n \\ & \sum_{i=1}^{n} a_i y_{ji} \leq z_j & \text{pour } j = 1,\dots,K \\ & y_{ji}, z_i \geq 0 & \text{pour } i = 1,\dots,n;\ j = 1,\dots,K \\ & y_{ji}, z_i \leq 1 & \text{pour } i = 1,\dots,n;\ j = 1,\dots,K \end{array}$$

C'est un programme linéaire avec un nombre linéaire de contraintes, il n'y a donc pas de problème pour calculer la borne obtenue par relaxation continue.

8.2.3.3. Relaxation lagrangienne. — On peut faire la relaxation lagrangienne en "oubliant" les contraintes $\sum_{j=1}^{K} y_{ji} = 1$. On écrit le lagrangien

$$\mathcal{L}(y, z, \lambda) := \sum_{j=1}^{K} z_j + \sum_{i=1}^{n} \lambda_i (\sum_{j=1}^{K} y_{ji} - 1).$$

Si on note v la valeur optimale du programme (28), c'est-à-dire la solution optimale de notre problème de bin-packing, on a toujours

$$v = \inf_{\boldsymbol{x} \in X} \sup_{\boldsymbol{\lambda} \in \Lambda} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) \geq \sup_{\boldsymbol{\lambda} \in \Lambda} \inf_{\boldsymbol{x} \in X} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = \sup_{\boldsymbol{\lambda} \in \Lambda} \mathcal{G}(\boldsymbol{\lambda}),$$

où

$$\begin{array}{rccc} \mathcal{G} : & \Lambda & \to & \mathbb{R} \cup \{-\infty\} \\ & \mathcal{G}(\boldsymbol{\lambda}) & \mapsto & \inf_{x \in X} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}), \end{array}$$

est la fonction duale.

Rappelons que cette dernière est concave et affine par morceaux et que l'idée de la relaxation lagrangienne est d'utiliser $\sup_{\boldsymbol{\lambda} \in \Lambda} \mathcal{G}(\boldsymbol{\lambda})$ comme borne inférieure (ou même n'importe quel $\mathcal{G}(\boldsymbol{\lambda})$ proche de l'optimum).

Ici,

$$\begin{array}{rll} \mathcal{G}(\boldsymbol{\lambda}) := & \min & \sum_{j=1}^{K} z_j + \sum_{i=1}^{n} \lambda_i (\sum_{j=1}^{K} y_{ji} - 1) \\ & \text{s.c.} & \left| \begin{array}{ll} \sum_{i=1}^{n} a_i y_{ji} \leq z_j & \text{pour } j = 1, \ldots, K \\ y_{ji}, z_i \in \{0, 1\} & \text{pour } i = 1, \ldots, n;\ j = 1, \ldots, K \end{array} \right. \end{array}$$

se réécrit

$$\mathcal{G}(\boldsymbol{\lambda}) := \sum_{i=1}^{n} \lambda_i + \sum_{j=1}^{K} S_j$$

avec $S_j := \min_{y, z \in \{0,1\}} \{z + \sum_{i=1}^{n} \lambda_i y_i : \sum_{i=1}^{n} a_i y_i \leq z\}$.

On doit donc résoudre K problèmes de sac-à-dos (quitte à multiplier par une constante pour avoir des entiers) : $S_j := \min_{\boldsymbol{y}, z \in \{0,1\}} \{z + \sum_{i=1}^{n} \lambda_i y_i : \sum_{i=1}^{n} a_i y_i \leq z\}$.

On sait donc calculer $\mathcal{G}(\boldsymbol{\lambda})$ (assez) facilement et trouver les $\boldsymbol{y}, z$ le réalisant. Et par conséquent, on sait calculer les sur-gradients de la fonction concave $\mathcal{G}(\boldsymbol{\lambda})$, et donc la maximiser.

Exercices

8.2.4. Inégalités valides pour le sac-à-dos. — On considère un problème de sac-à-dos avec $a_1, \ldots, a_n, b$ des réels positifs. On note S l'ensemble des solutions réalisables

$$S := \left\{ \boldsymbol{x} \in \{0, 1\}^n : \sum_{i=1}^{n} a_i x_i \leq b \right\}.$$

On appelle *ensemble dépendant* tout ensemble $C \subseteq \{1, \ldots, n\}$ tel que $\sum_{i \in C} a_i > b$.

1. Montrer que tout $\boldsymbol{x} \in S$ satisfait l'inégalité

$$\sum_{i \in E(C)} x_i \leq |C| - 1,$$

où $E(C) = C \cup \{j : a_j \geq \max_{i \in C} a_i\}$.

2. Expliquer l'intérêt de ce type d'inégalités dans une résolution d'un problème de sac-à-dos dans une approche branch-and-bound.

8.2.5. Inégalités valides pour le bin-packing. — On rappelle la modélisation PLNE du problème du bin-packing. On suppose que l'on dispose d'un stock de K boîtes de taille 1.

$$\begin{array}{lll} \min & \sum_{j=1}^{K} z_j & \\ \text{s.c.} & \sum_{j=1}^{K} y_{ji} = 1 & \text{pour } i = 1, \ldots, n \\ & \sum_{i=1}^{n} a_i y_{ji} \leq z_j & \text{pour } j = 1, \ldots, K \\ & y_{ji}, z_i \in \{0, 1\} & \text{pour } i = 1, \ldots, n; \\ & & j = 1, \ldots, K \end{array}$$

où $z_j = 1$ si la boîte j est utilisée et $y_{ji} = 1$ si l'objet i est mis dans la boîte j.

1. Montrer qu'en ajoutant les contraintes $z_j \geq z_{j+1}$, pour $j = 1, \ldots, K-1$, on continue à modéliser le problème du bin-packing.
2. Même question avec la contrainte $\sum_{j=1}^{K} z_j \geq \lceil \sum_{i=1}^{n} a_i \rceil$.
3. Expliquer l'intérêt de ce type d'inégalité dans une résolution d'un problème de bin-packing dans une approche branch-and-bound.

8.2.6. Gros objets. — Supposons qu'un instance $a_1, \ldots, a_n$ du bin-packing soit telle que $a_i > \frac{1}{3}$ pour tout $i = 1, \ldots, n$. On suppose toujours que les conteneurs sont de taille 1.

1. Montrer comment modéliser ce problème de bin-packing comme un problème de couplage de cardinalité maximale dans un graphe.

2. Montrer comment en réalité on peut résoudre ce problème en $O(n \log(n))$.

8.2.7. Calcul glouton de relâché continu de problème de sac-à-dos. — Compléter la preuve du Lemme 8.1.2.

CHAPITRE 9

POSITIONNEMENT D'ENTREPÔTS

De nombreuses décisions économiques mettent en jeu la sélection ou le positionnement de dépôts, d'usines, de relais, d'hôpitaux, etc. afin de répondre de manière optimale à la demande.

9.1. Formalisation

La version la plus simple du problème de positionnement d'entrepôts est la suivante.

Problème du positionnement d'entrepôts :

Données : Un ensemble fini de clients $\mathcal{D}$, un ensemble fini d'entrepôts potentiels $\mathcal{F}$, un coût fixe $f_i \in \mathbb{R}_+$ d'ouverture pour chaque entrepôt $i \in \mathcal{F}$ et un coût de service $c_{ij} \in \mathbb{R}_+$ pour chaque $i \in \mathcal{F}$ et $j \in \mathcal{D}$.

Demande : Trouver un sous-ensemble $X \subseteq \mathcal{F}$ (dits *entrepôts ouverts*) et une affectation $\sigma : \mathcal{D} \to X$ des clients aux entrepôts ouverts, de façon à ce que la quantité

$$\sum_{i \in X} f_i + \sum_{j \in \mathcal{D}} c_{\sigma(j)j}$$

soit minimale.

Un exemple d'input et d'output pour ce problème est donné Figures 1 et 2.

On dit qu'on est dans le le cas *métrique* si

$$c_{ij} + c_{i'j} + c_{i'j'} \geq c_{ij'} \quad \text{pour tout } i, i' \in \mathcal{F} \text{ et } j, j' \in \mathcal{D}.$$

C'est en particulier le cas lorsque les coûts de services c_{ij} sont proportionnels à la distance géométrique. Cette inégalité contient plus de termes que l'inégalité triangulaire classique, à laquelle elle ressemble. C'est dû au fait que les coûts ne sont pas définis entre entrepôts ou entre clients.

Commençons par la question de la complexité.

Proposition 9.1.1. — *Le problème du positionnement d'entrepôts est* **NP***-difficile, même dans le cas métrique.*

On va voir qu'il existe un algorithme d'approximation (découvert en 1997), de bons schémas de branch-and-bound, et enfin des formulations de recherche locale très naturelles.

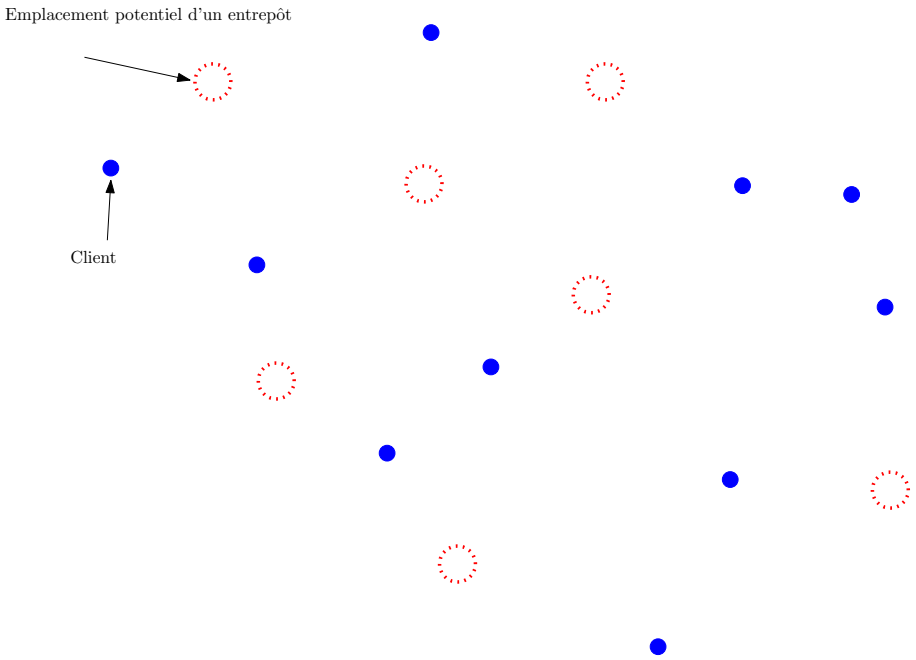


FIG. 1. Les données d'un problème de positionnement d'entrepôts.

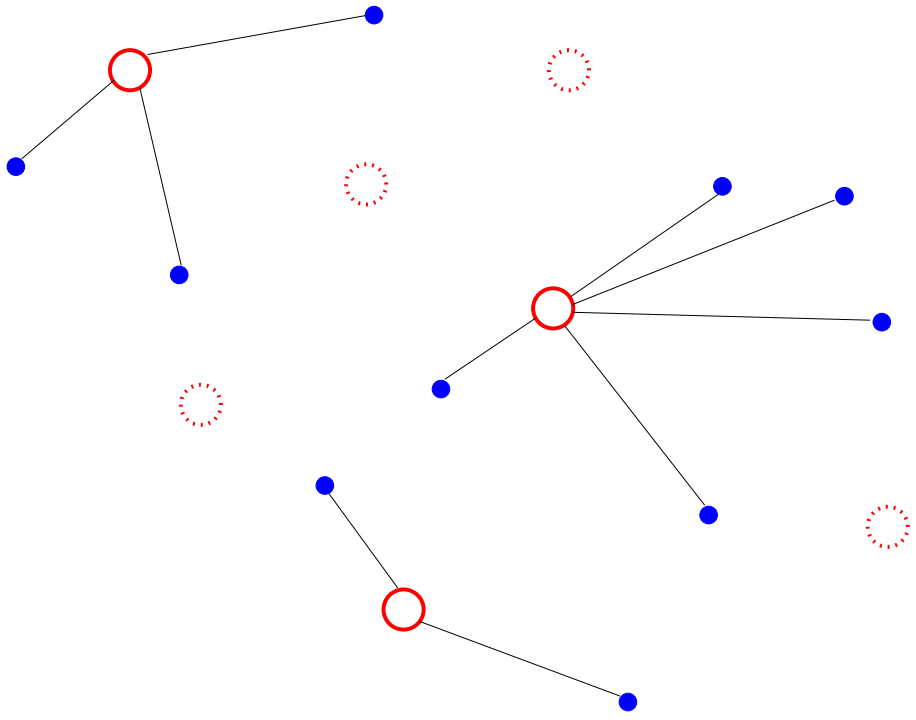


FIG. 2. Une solution au problème du positionnement d'entrepôts.

9.2. Branch-and-bound

9.2.1. Programme linéaire en nombres entiers. — Une bonne façon de rechercher un schéma de branch-and-bound est de commencer par la modélisation sous forme d'un programme linéaire en nombres entiers. Ici, le problème s'écrit facilement sous cette forme.

$$
\begin{array}{lll}
\text{Min} & \sum_{i\in\mathcal{F}} f_i y_i + \sum_{i\in\mathcal{F}}\sum_{j\in\mathcal{D}} c_{ij}x_{ij} & \\
\text{s.c.} & x_{ij} \le y_i & i\in\mathcal{F},\, j\in\mathcal{D} \quad (1)\\
& \sum_{i\in\mathcal{F}} x_{ij} = 1 & j\in\mathcal{D} \quad (2)\\
& x_{ij}\in\{0,1\} & i\in\mathcal{F},\, j\in\mathcal{D} \quad (3)\\
& y_i\in\{0,1\} & i\in\mathcal{F}. \quad (4)
\end{array}
$$

On est en présence d'un programme linéaire en nombres entiers, à valeur dans $\{0,1\}$, avec un nombre polynomial de contraintes.

9.2.2. Relaxation linéaire. — On peut appliquer la technique du branch-and-bound dans sa version la plus classique. Le branchement se fait sur les choix de fixer à 0 ou à 1 certaines variables x_{ij} et y_i. Les bornes sont obtenues par le relâché continu :

$$
\begin{array}{lll}
\text{Min} & \sum_{i\in\mathcal{F}} f_i y_i + \sum_{i\in\mathcal{F}}\sum_{j\in\mathcal{D}} c_{ij}x_{ij} & \\
\text{s.c.} & x_{ij} \le y_i & i\in\mathcal{F},\, j\in\mathcal{D} \quad (1)\\
& \sum_{i\in\mathcal{F}} x_{ij} = 1 & j\in\mathcal{D} \quad (2)\\
& 0 \le x_{ij} & i\in\mathcal{F},\, j\in\mathcal{D} \quad (3)\\
& 0 \le y_i & i\in\mathcal{F}. \quad (4)
\end{array} \tag{29}
$$

9.2.3. Relaxation lagrangienne. — On peut également obtenir une borne inférieure par relaxation lagrangienne. Relâchons la contrainte (2)

$$\sum_{i\in\mathcal{F}} x_{ij} = 1 \text{ pour } j\in\mathcal{F}.$$

On écrit le lagrangien

$$\mathcal{L}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{\lambda}) = \sum_{i\in\mathcal{F}} f_i y_i + \sum_{i\in\mathcal{F}}\sum_{j\in\mathcal{D}} c_{ij}x_{ij} + \sum_{j\in\mathcal{D}} \lambda_j \left(1 - \sum_{i\in\mathcal{F}} x_{ij}\right).$$

Notre problème consiste à résoudre

$$\min_{\boldsymbol{x}\in\{0,1\}^{\mathcal{F}\times\mathcal{D}},\boldsymbol{y}\in\{0,1\}^{\mathcal{F}}:\, x_{ij}\le y_i} \max_{\lambda_j\in\mathbb{R}} \mathcal{L}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{\lambda}).$$

Une borne inférieure à notre problème est donc fournie par

$$\max_{\lambda_j\in\mathbb{R}} \min_{\boldsymbol{x}\in\{0,1\}^{\mathcal{F}\times\mathcal{D}},y\in\{0,1\}^{\mathcal{F}}:\, x_{ij}\le y_i} \mathcal{L}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{\lambda}),$$

c'est-à-dire, en posant

$$\mathcal{G}(\boldsymbol{\lambda}) := \min_{\boldsymbol{x}\in\{0,1\}^{\mathcal{F}\times\mathcal{D}},\boldsymbol{y}\in\{0,1\}^{\mathcal{F}}:\, x_{ij}\le y_i} \mathcal{L}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{\lambda}),$$

on a $\mathcal{G}(\boldsymbol{\lambda})$ borne inférieure pour tout $\boldsymbol{\lambda}\in\mathbb{R}^{\mathcal{F}}$, la meilleure borne étant

$$\max_{\boldsymbol{\lambda}\in\mathbb{R}^{\mathcal{F}}} \mathcal{G}(\boldsymbol{\lambda})$$

On veut donc être capable de calculer $\mathcal{G}(\boldsymbol{\lambda})$, i.e. de résoudre à $\boldsymbol{\lambda}$ fixé

$$\min_{\boldsymbol{x}\in\{0,1\}^{\mathcal{F}\times\mathcal{D}},\boldsymbol{y}\in\{0,1\}^{\mathcal{F}}:\, x_{ij}\le y_i} \mathcal{L}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{\lambda}).$$

Il est facile de résoudre le programme : $\mathcal{G}(\lambda) := \min_{\boldsymbol{x}\in\{0,1\}^{\mathcal{F}\times\mathcal{D}},\boldsymbol{y}\in\{0,1\}^{\mathcal{F}}:\, x_{ij}\le y_i} \mathcal{L}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{\lambda})$.

En effet on peut écrire

$$\mathcal{G}(\boldsymbol{\lambda}) = \sum_{j\in\mathcal{D}} \lambda_j + \sum_{i\in\mathcal{F}} d_i(\boldsymbol{\lambda})$$

avec

$$d_i(\boldsymbol{\lambda}) = \min_{x_i.\in\{0,1\}^{\mathcal{D}},y_i\in\{0,1\},x_{ij}\le y_i} f_i y_i + \sum_{j\in\mathcal{D}} (c_{ij}-\lambda_j)x_{ij}.$$

Chacun des d_i se calcule facilement. Il suffit de comparer la valeur obtenue en posant $y_i = 0$ qui impose $d_i = 0$ et celle obtenue en posant $y_i = 1$ qui impose $x_{ij} = 0$ si et seulement si $c_{ij} \geq \lambda_j$.

9.3. Algorithme d'approximation

Reprenons la relaxation linéaire (29) et le programme dual

$$\begin{array}{lll} \text{Max} & \sum_{j\in\mathcal{D}} v_j & \\ \text{s.c.} & v_j - w_{ij} \leq c_{ij} & i \in \mathcal{F},\, j \in \mathcal{D} \quad (1) \\ & \sum_{j\in\mathcal{D}} w_{ij} \leq f_i & i \in \mathcal{F} \quad (2) \\ & w_{ij} \geq 0 & i \in \mathcal{F},\, j \in \mathcal{D} \quad (3) \end{array}$$

et notons $(\boldsymbol{x}^*, \boldsymbol{y}^*)$ les solutions optimales du primal et $(\boldsymbol{v}^*, \boldsymbol{w}^*)$ les solutions optimales du dual.

L'algorithme Shmoys-Tardos-Aardal [**23**] (du nom de ses inventeurs[(1)]) prend ces solutions optimales $(\boldsymbol{x}^*, \boldsymbol{y}^*)$ et $(\boldsymbol{v}^*, \boldsymbol{w}^*)$. A partir d'elles, il construit une solution réalisable du problème du positionnement d'entrepôts de la manière suivante.

Choisir le client j_1 tel que $v^*_{j_1}$ soit minimum. Choisir l'entrepôt i_1 tel que $x^*_{i_1 j_1} > 0$ et f_{i_1} minimum. Tous les clients j pour lesquels il existe i tels que $x^*_{ij_1} > 0$ et $x^*_{ij} > 0$ sont affectés à l'entrepôt i_1.

On efface tous les clients qui ont été affectés, et on recommence. On définit alors j_2, i_2, etc.

L'algorithme est illustré Figures 3 et 4.

Théorème 9.3.1. — *L'algorithme Shmoys-Tardos-Aardal retourne une solution qui vaut au plus* 4 *fois l'optimum pour le problème de positionnement d'entrepôt dans le cas métrique.*

Cet algorithme a été trouvé en 1997. Auparavant, l'existence d'un algorithme d'approximation était une question ouverte.

Démonstration. — Par la propriété des écarts complémentaires (écarts complémentaires - voir Chapitre 4), on a

$$x^*_{ij} > 0 \quad \Rightarrow \quad c_{ij} \leq v^*_j.$$

Par conséquent le coût de service du client j est au plus

$$c_{i_k j} \leq c_{ij} + c_{ij_k} + c_{i_k j_k} \leq v^*_j + 2v^*_{j_k} \leq 3v^*_j$$

où i est tel que $x^*_{ij} > 0$ et $x^*_{ij_k} > 0$.

Le coût d'ouverture de l'entrepôt i_k peut être majoré par

$$f_{i_k} \leq \sum_{i\in\mathcal{F}:\, x^*_{ij_k}>0} x^*_{ij_k} f_i \leq \sum_{i\in\mathcal{F}:\, x^*_{ij_k}>0} y^*_i f_i.$$

Comme $x^*_{ij_k} > 0$ implique que $x^*_{ij_{k'}} = 0$ pour tout $k' \neq k$, le coût total d'ouverture des entrepôts est majoré par $\sum_{i\in\mathcal{F}} y^*_i f_i$.

Le total est majoré par $3\sum_{j\in\mathcal{D}} v^*_j + \sum_{i\in\mathcal{F}} y^*_i f_i$, qui est plus petit que 4 fois la valeur optimale des programmes linéaires. □

(1) L'ambiguité du terme « inventeur » – rappelons que l'on parle de l'*inventeur* d'un trésor et non de son « découvreur » permet d'éviter de se mettre à dos tant les partisants que les détracteurs de la préexistence des objets mathématiques.

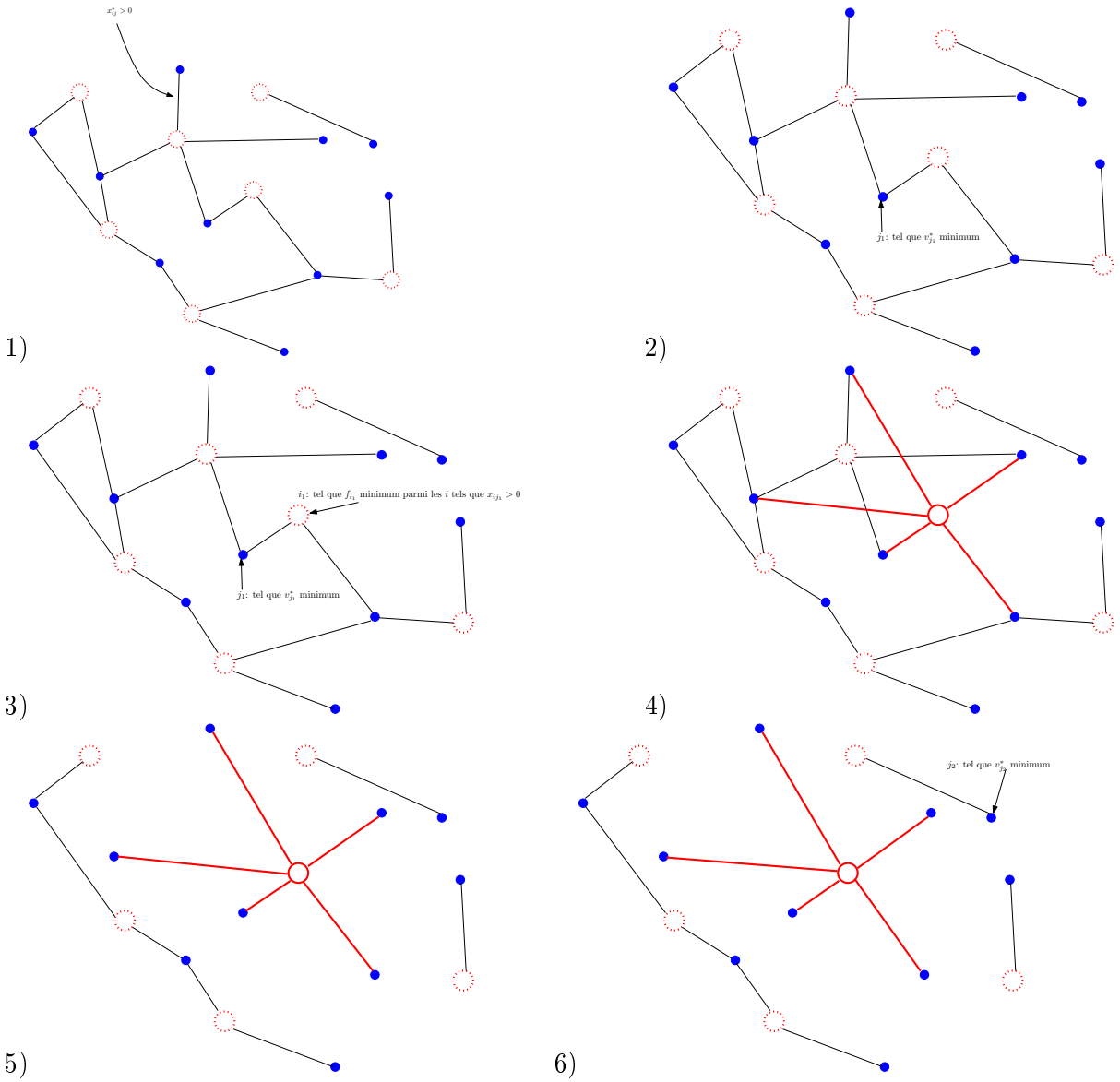


FIG. 3. Illustration de l'algorithme de Schmoys-Tardos-Aardal.

9.4. Recherche locale

Rappelons que la recherche locale consiste à mettre une notion de voisinage sur l'espace des solutions réalisables. Une fois ce voisinage défini, on peut facilement implémenter des métaheuristiques du type tabou, récuit simulé, etc...

Dans le cas du positionnement d'entrepôts, il est très facile de définir un voisinage sur l'espace des solutions.

Remarquons d'abord que l'espace des solutions peut être identifié à l'ensemble des parties X de $\mathcal{F}$. En effet, une fois X fixé, on a toujours intérêt à affecter le client j à l'entrepôt ouvert $i \in X$ minimisant c_{ij}.

Ensuite, on dit que X' est *voisin* de X si l'on est dans une des ces situations

1. $X' := X \setminus \{x\}$ pour un $x \in X$, $(drop)$
2. $X' := X \cup \{x'\}$ pour un $x' \in \mathcal{F} \setminus X$ (add) ou

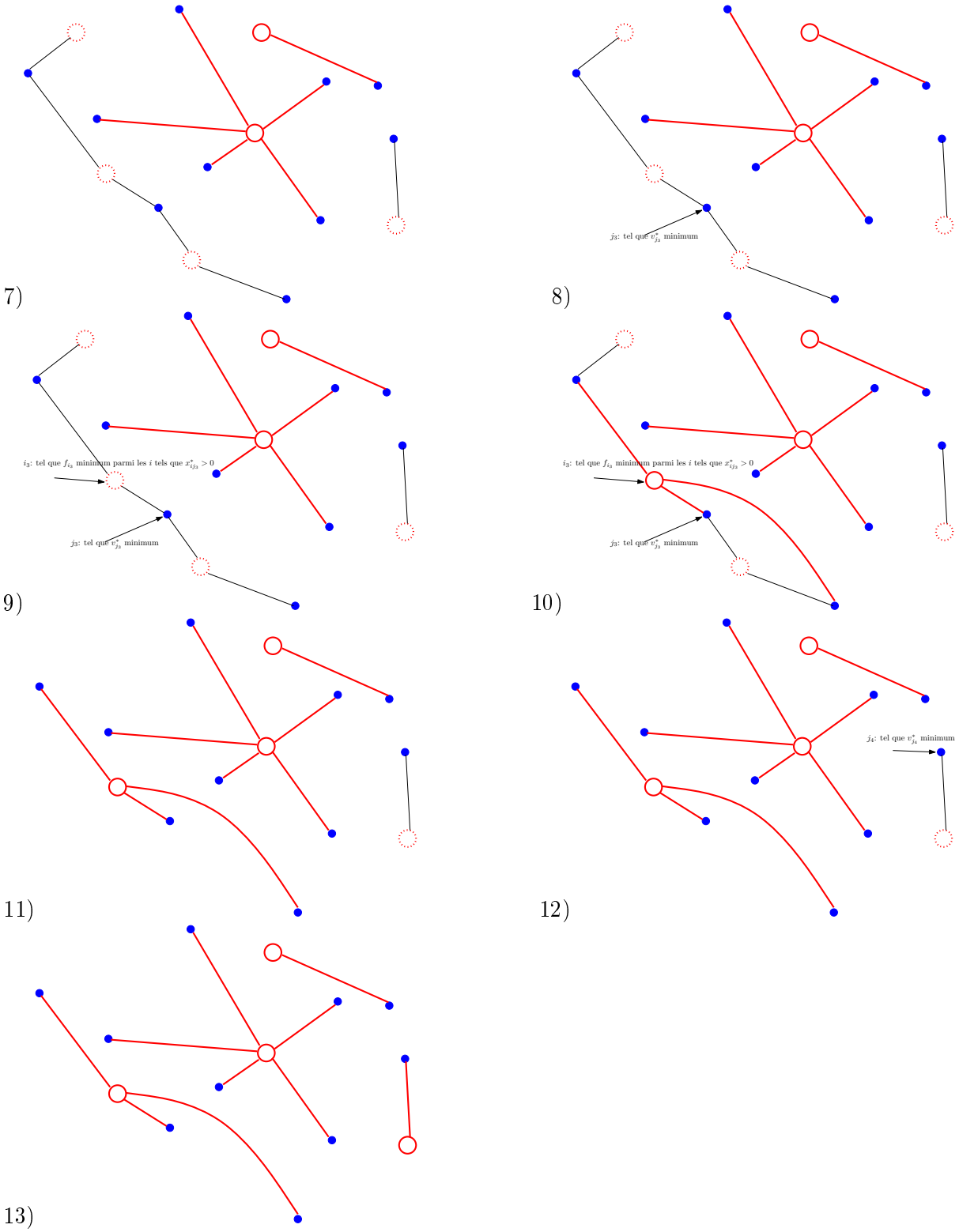


FIG. 4. Illustration de l'algorithme de Schmoys-Tardos-Aardal (suite).

3. $X' := X \setminus \{x\} \cup \{x'\}$ (*swap*).

Dans le cas du problème de positionnement d'entrepôts, on a réalisé dans les années 2000 que la recherche locale était extrêmement efficace (plus que pour les autres problèmes de RO).

On a par exemple le théorème suivant [**2**]

Théorème 9.4.1. — *Si X est un minimum local pour le voisinage défini précédemment, alors X vaut au plus* 3 *fois l'optimum.*

En revanche, on n'a pas a priori d'évaluation du temps de calcul d'un optimum local.

9.5. Exercices

9.5.1. Positionnement d'entrepôts avec capacité. — On reprend le problème de positionnement d'entrepôts vu en cours, mais avec cette fois une contrainte supplémentaire : chaque entrepôt ne peut desservir qu'un nombre limité de clients. Modéliser le problème suivant comme un problème linéaire en nombres entiers.

Données : Un ensemble fini de clients $\mathcal{D}$, un ensemble fini d'entrepôts potentiels $\mathcal{F}$, un coût fixe $f_i \in \mathbb{R}_+$ d'ouverture et une capacité K_i pour chaque entrepôt $i \in \mathcal{F}$, et un coût de service $c_{ij} \in \mathbb{R}_+$ pour chaque $i \in \mathcal{F}$ et $j \in \mathcal{D}$.

Demande : Trouver un sous-ensemble $X \subseteq \mathcal{F}$ (dits *entrepôts ouverts*) et une affectation $\sigma : \mathcal{D} \to X$ des clients aux entrepôts ouverts, tel que pour tout i, l'entrepôt i ne desserve pas plus de K_i clients, de façon à ce que la quantité

$$\sum_{i \in X} f_i + \sum_{j \in \mathcal{D}} c_{\sigma(j)j}$$

soit minimale.

9.5.2. Relaxation lagrangienne du problème de localisation d'entrepôt. — On reprend la modélisation PLNE du problème de la localisation d'entrepôt.

$$\begin{array}{lll}
\text{Min} & \sum_{i\in\mathcal{F}} f_i y_i + \sum_{i\in\mathcal{F}}\sum_{j\in\mathcal{D}} c_{ij}x_{ij} & \\
\text{s.c.} & x_{ij} \le y_i & i \in \mathcal{F},\, j \in \mathcal{D} \quad (1)\\
& \sum_{i\in\mathcal{F}} x_{ij} = 1 & j \in \mathcal{F} \quad (2)\\
& x_{ij} \in \{0,1\} & i \in \mathcal{F},\, j \in \mathcal{D} \quad (3)\\
& y_i \in \{0,1\} & i \in \mathcal{F}. \quad (4)
\end{array}$$

Dans le cours, on a relâché la contrainte (2). Relâcher maintenant la contrainte (1) à la place de la contrainte (2) et montrer que la fonction duale se calcule encore très simplement.

9.5.3. Positionnement d'entrepôts, forward logistique et reverse logistique. — La logistique traditionnellement prise en compte est la logistique forward, qui concerne l'approvisionnement de clients à partir d'entrepôts, que nous appellerons dans cet exercice *entrepôts forward*. La logistique reverse concerne les retours (biens non consommés par le client) vers des *entrepôts reverse* chargés de traiter ces retours.

Pour de nombreuses raisons (environnementales, durée de vie plus courte des produits, vente à distance), la reverse logistique prend de l'importance dans le management de la supply chain. L'objectif de cet exercice est d'étudier l'implication que cela peut avoir dans la modélisation des problèmes de positionnement d'entrepôts.

On suppose donc que l'on dispose d'un ensemble I de positions potentielles d'entrepôts, et d'un ensemble J de clients. Les clients souhaitent s'approvisionner en un certain bien, dont la quantité est mesurée par un nombre réel.

Le coût d'ouverture d'un entrepôt forward en la position i est noté $f_i \in \mathbb{R}_+$ et celui d'un entrepôt reverse $r_i \in \mathbb{R}_+$. La capacité d'un entrepôt forward en position i (ie la demande totale qu'il peut satisfaire) est notée b_i et la capacité d'un entrepôt reverse en position i (ie le retour total qu'il peut accepter) est notée e_i. Chaque client j a une demande $h_j \in \mathbb{R}$. De plus, chaque client j a un taux de retour $\alpha_j \in [0, 1]$, ce qui signifie qu'il renvoie une quantité $\alpha_j h_j$ du bien. Le coût de tranport d'une unité de bien d'un entrepôt i à un client j, ou d'un client j à l'entrepôt i, est $c_{ij} \in \mathbb{R}_+$.

Le bien étant supposé parfaitement fractionnable, un client peut être approvisionné par plusieurs entrepôts forward ; et de même, plusieurs entrepôts reverse peuvent être destination des retours d'un client. De plus, il est possible d'ouvrir au même endroit i en entrepôt forward et un entrepôt reverse.

1. Proposer une modélisation par la programmation linéaire mixte (avec des variables entières et des variables réelles) du problème de minimisation des coûts de conception d'un tel système, pouvant traiter toute la demande et tous les retours. Noter que ce problème se décompose en deux sous-problèmes indépendants : un pour le forward et l'autre pour le reverse.

On suppose maintenant que si on parvient à ouvrir un entrepôt forward et un entrepôt reverse au même endroit i, cela diminue le coût total d'ouverture des deux entrepôts d'une quantité s_i.

2. Modifier la modélisation précédente pour tenir compte de cette nouvelle possibilité, tout en restant dans la programmation linéaire mixte. Noter que les deux sous-problèmes ne sont plus indépendants.

3. Proposer une définition de l'espace des solutions et d'un voisinage et expliquer comment pourrait être construite une métaheuristique de type recherche locale à partir de cette définition.

4. Toujours dans le cas d'une métaheuristique de type recherche locale, comment adapter la méthode au cas où le bien n'est pas parfaitement fractionnable (les quantités seront donc des entiers) et que les $\alpha_j h_j$, les h_j, les b_i et les e_i sont entiers pour tout i et j ?

9.5.4. Positionnement d'ambulances. — Pour diminuer le temps mis par les ambulances pour atteindre les victimes d'accidents, certains hôpitaux envisagent de les positionner de manière optimisée sur le département dont ils ont la charge. Considérons donc un certain hôpital ayant K ambulances. L'ensemble des communes est noté C. On dispose d'un ensemble S de points de stationnement potentiel et on suppose que $|S| \geq K$. Pour $c \in C$ et $s \in S$, on note $t_{s,c}$ le temps nécessaire à une ambulance positionné en s pour atteindre c. Si $S' \subseteq S$ est l'ensemble des points où sont positionnées les ambulances (avec bien sûr $|S'| \leq K$), dans le pire des cas, une ambulance atteindra la commune où a eu lieu l'appel en un temps égal à $\max_{c \in C} \min_{s \in S'} t_{s,c}$. C'est ce temps maximum que l'hôpital souhaite rendre le plus petit possible.

1. Montrez que ce problème est NP-difficile. Indication : soit $G = (V, E)$ un graphe non-orienté ; on appelle *dominant* un sous-ensemble $Y \subseteq V$ tel que tout $v \in V$ est soit dans Y, soit a un voisin dans Y ; décider s'il existe un dominant de taille $\leq K$ est NP-complet.

2. Démontrez que le programme linéaire suivant modélise le problème. Pour cela, procédez en deux temps : montrez que toute solution optimale du problème donne une solution de même

valeur au programme linéaire ; puis montrez que toute solution optimale du programme linéaire donne une solution de même valeur au problème.

$$
\begin{array}{lrcll}
\min & h & & & \\
\text{s.c.} & \displaystyle\sum_{s\in S} y_s & \leq & K & \text{(i)} \\
& \displaystyle\sum_{c\in C} x_{s,c} & \leq & |C|y_s \quad \text{pour tout } s \in S & \text{(ii)} \\
& \displaystyle\sum_{s\in S} x_{s,c} & = & 1 \quad \text{pour tout } c \in C & \text{(iii)} \\
& t_{s,c}x_{s,c} & \leq & h \quad \text{pour tout } s \in S \text{ et } c \in C & \text{(iv)} \\
& x_{s,c} & \in & \{0,1\} \quad \text{pour tout } s \in S \text{ et } c \in C & \text{(v)} \\
& y_s & \in & \{0,1\} \quad \text{pour tout } s \in S & \text{(vi)} \\
& h & \in & \mathbb{R}_+ & \text{(vii)}
\end{array}
$$

Un tel programme linéaire se résout par branch-and-bound. La borne dont on se sert alors peut être celle obtenue en relâchant les contraintes d'intégrité de $\boldsymbol{x}$ et de $\boldsymbol{y}$. On se propose de voir si l'on peut également obtenir des bornes par relaxation lagrangienne.

3. Ecrire le programme d'optimisation obtenu lorsqu'on procède à la relaxation lagrangienne de la contrainte (ii).

4. Montrez que ce programme se ramène à deux programmes distincts et indépendants, l'un ne mettant en jeu que les variables $\boldsymbol{y}$, l'autre ne mettant en jeu que les variables $\boldsymbol{x}$ et h.

5. Montrez que chacun de ces deux programmes se résout en temps polynomial.

6. Expliquez pourquoi la polynomialité de ces deux programmes permet d'espérer un calcul efficace d'une borne inférieure au programme linéaire de la question 2.

CHAPITRE 10

ORDONNANCEMENT INDUSTRIEL

Dans un contexte de tâches ou d'opérations à effectuer sous des contraintes de temps et de ressource, un problème d'*ordonnancement* consiste à donner les instants auxquels commencer les tâches, en vue d'optimiser un certain critère. Dans l'industrie, dans le bâtiment, sur les gros chantiers, les problèmes d'ordonnancement sont souvent très présents, et de leur solution peut dépendre la réussite du projet. Nous verrons que pour une certaine famille de problèmes d'ordonnancement (les problèmes d'*atelier*), les problèmes sont non seulement **NP**-difficiles, mais même mal résolu en pratique, contrairement aux problèmes de tournées.

Dans ce cours, on se limitera au cas de ressources *renouvelables* : quand l'opération a terminé d'utiliser une ressource, cette dernière est à nouveau disponible (équipe d'employés, machine, etc.).

10.1. Préliminaires

De manière un peu générique, un problème d'ordonnancement se formule avec des *tâches*, qui elles-mêmes se découpent éventuellement en *opérations*. Etant donnée des *contraintes*, on appellera *ordonnancement* la donnée des instants auxquels commencent les différentes tâches ou opérations.

Les *contraintes de temps* peuvent être de deux natures :

- Durées des tâches
- Précédence

Les *contraintes de ressource* peuvent également être de deux natures :

- *Disjonctive* "si les tâches i et j sont effectuées sur la machine k, elles doivent être faites dans tel ordre"
- *Cumulative* "la tâche i consomme a_{ik} de la ressource k. La ressource k a une capacité A_k. A tout instant, la somme des consommations sur la machine k doit être inférieure à sa capacité."

Les critères à optimiser peuvent être variés. Exemples :

- **Coût** on connaît le coût d'affectation d'une tâche i sur une machine k, minimiser le coût.
- **Makespan** Temps pour effectuer l'ensemble des tâches.
- **Temps moyen** Temps moyen mis par une tâche pour être traitée. Son intérêt peut être varié. Par exemple si le problème d'ordonnancement s'insère dans un processus plus grand, ce genre de critère peut être intéressant.

Une façon commode de représenter un ordonnancement est le *digramme de Gantt* : en abscisse, on met le temps, et on ordonnées, les différentes tâches. Sur chaque ligne repérée par une tâche,

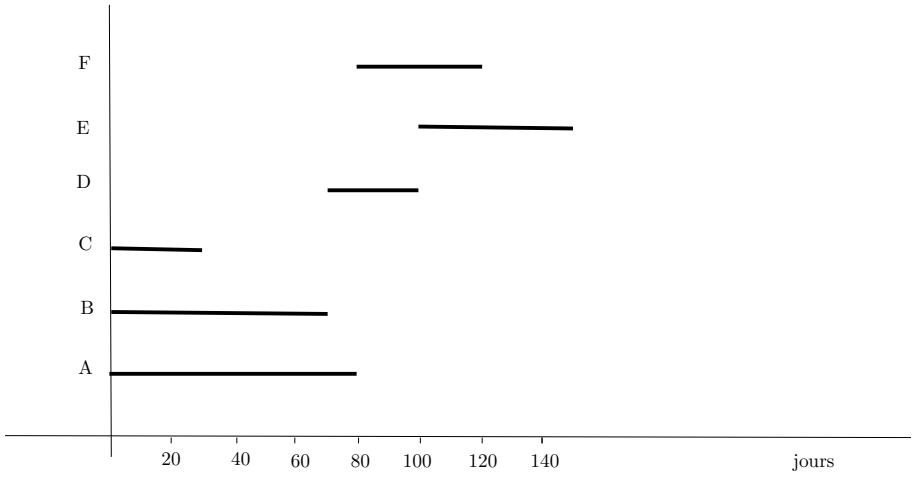


FIG. 1. Le diagramme de Gantt qui correspond à l'exemple de la section qui suit.

on met des intervalles qui occupent les instants pendant lesquels la tâche est traitée. Voir Figure 1 pour voir un exemple.

10.2. Management de projet

Lorsqu'on a un projet à conduire et différentes tâches à effectuer, connaissant la durée des tâches et les relations de précédence, une question naturelle est celle de la durée minimale du projet. Différentes méthodes peuvent être utilisées, certaines informelles, en jouant avec le diagramme de Gantt, d'autres exactes, comme la méthode PERT ou la méthode potentiel-tâches. C'est cette dernière que nous allons maintenant présenter.

On se donne une collection I de tâches, pour chaque tâche i, on se donne sa durée d_i, et des contraintes par rapport aux autres tâches de cette forme : "la tâche i ne peut commencer que lorsque $r\%$ de la tâche j au moins a été effectué".

On veut trouver

- la durée minimale du projet.
- *les tâches critiques* : les tâches pour lesquelles une modification de l'instant de début rallonge la durée du projet.
- *les marges* : la marge d'une tâche est l'intervalle de temps sur lequel on peut faire démarrer la tâche sans rallonger la durée du projet. En particulier, une marge nulle signifie une tâche critique.

On modélise ce problème sous la forme d'un graphe dont les sommets sont les débuts des tâches et les arcs sont les relations de précédence. On ajoute deux tâches fictives : **Début** et **Fin**. De plus, chaque arc (u, v) est muni d'une longueur égale à la durée minimale séparant le début de u de celui de v.

La première remarque que l'on peut faire est la suivante

Si ce projet a un sens, le graphe est acircuitique.

En effet, un cycle traduirait l'obligation de remonter dans le temps.

Considérons l'exemple suivant.

Tâches	Description	Durée (j)	Contraintes
A	Construction voie ferrée	80	Début
B	Forage des puits	70	Début
C	Construction logements provisoires	30	Début
D	Pompage de l'eau (Fonds de mine)	30	B
E	Aménagement fonds + ascenseurs	50	A, D
F	Construction logements définitifs	40	A, C

On fait la modélisation décrite ci-dessus : voir Figure 2.

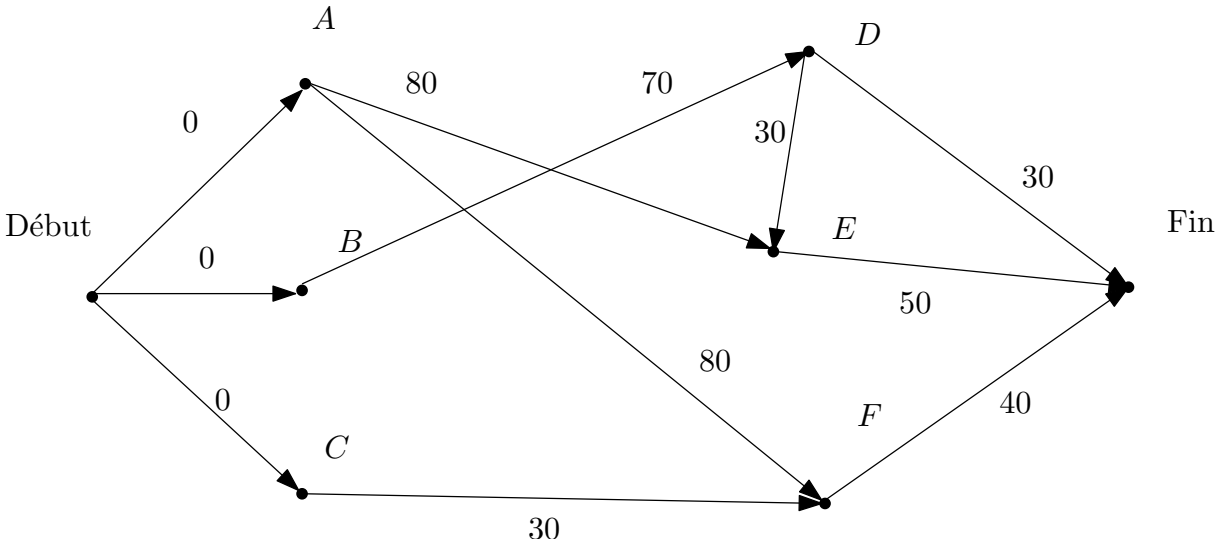


FIG. 2. Le graphe de la méthode potentiel tâche.

Notons t_I et t'_I les instants au plus tôt et au plus tard auxquels commencer la tâche I.

Proposition 10.2.1. — *t_I est la longueur du plus long chemin de* **Début** *à I. t'_I, c'est t_{Fin} moins la longueur du plus long chemin de I à* **Fin**.

Démonstration. — En effet,
- pour chaque tâche I, on ne peut pas commencer avant la somme des longueurs des arcs du plus long chemin de **Début** à I.
- on faisant débuter chaque tâche I précisément à la somme des longueurs des arcs du plus long chemin de **Début** à I, on peut commencer chaque tâche à l'instant t_I.

De même pour t'_I. □

Pour trouver la durée minimale du projet, les tâches critiques, etc. il faut donc savoir calculer les plus longs chemins dans un graphe acircuitique $D = (V, A)$, muni d'une longueur $l : A \to \mathbb{R}_+$. Cela se fait par la programmation dynamique, comme nous l'avons vu au Chapitre 3. Si on note η_I la longueur du plus long chemin de **Début** à I, on peut écrire (équation de Bellman)

$$\eta_I = \max_{(J,I)\in A} (\eta_J + l(J, I)).$$

On peut calculer tous les η_I en $O(m)$ où m est le nombre d'arcs. On peut calculer de même les π_I, plus longs chemins de I à **Fin**. On pose alors $t_I := \eta_I$, $t'_I := t_{\textbf{Fin}} - \pi_I$ et la marge $m_I := t'_I - t_I$. Les *chemins critiques* sont les plus longs chemins de **Début** à **Fin** ; toutes les tâches d'un chemin critique sont caractérisées par une marge nulle.

10.3. Ordonnancement d'atelier

10.3.1. Introduction. —

- On a des *tâches*, chacune consistant en des *opérations*.
- Chaque opération doit être faite sur une machine particulière. Une machine ne peut faire au plus qu'une opération à la fois.
- Objectif : minimiser le temps total, ou la somme des instants de fins des tâches (par exemple).

Il y a trois modèles de base pour les problèmes d'ordonnancement d'atelier.

- *flow-shop* : chaque tâche est constituée du même ensemble d'opérations, et pour chaque tâche, l'ordre (total) dans lequel doit être effectué les opérations est le même.
- *job-shop* : pour chaque tâche, les opérations sont totalement ordonnées.
- *open-shop* : pas de contrainte de précédence sur les opérations.

Remarque Le flow-shop est un cas particulier du job-shop.

On se place dans un premier temps dans le cas *non-préemptif* : une opération, une fois commencée, ne peut être interrompue.

En formalisant un peu plus : On a des tâches $J_1, \dots, J_n$ et des machines $M_1, \dots, M_m$. La tâche J_j est constituée de m_j opérations $O_{j1}, \dots, O_{jm_j}$. L'opération O_{jk} a un temps d'exécution de p_{jk}. La machine correspondant à l'opération O_{jk} est notée m_{jk}.

Trouver l'ordonnancement, c'est fixer les instants t_{jk} de début des opérations O_{jk}.

L'instant de fin de la tâche j dans le cas du job shop s'écrit $C_j := t_{jm} + p_{jm}$. Dans le contexte de l'ordonnancement d'atelier, on a deux objectifs classiques (parmi beaucoup d'autres) :

- minimiser "la durée moyenne" : $\sum_j C_j$,
- minimiser le *makespan* : $C_{\max} = \max_j C_j$.

Ce sont des problèmes très difficiles.

Théorème 10.3.1. —
- *Le problème du flow-shop à trois machines et minimisation du makespan $C_{\max}$ est* **NP***-difficile.*
- *Le problème du flow-shop à deux machines et minimisation du $\sum_j C_j$ est* **NP***-difficile.*
- *Le problème de l'open-shop à trois machines et minimisation du makespan $C_{\max}$ est* **NP***-difficile.*
- *Le problème de l'open shop à deux machines et minimisation du $\sum_j C_j$ est* **NP***-difficile.*

Mais en plus d'être **NP**-difficile, on a du mal en pratique à résoudre de manière optimale ces problèmes, contrairement au problème du voyageur de commerce par exemple. Même avec des métaheuristiques ou des techniques de branch-and-bound, des instances job-shop à 15 machines et 15 tâches sont encore non résolues à ce jour !

Par exemple, on ne connaît pas l'ordonnancement optimal pour l'instance suivante de flow-shop à 20 tâches et 5 machines[(1)]

54	83	15	71	77	36	53	38	27	87	76	91	14	29	12	77	32	87	68	94
79	3	11	99	56	70	99	60	5	56	3	61	73	75	47	14	21	86	5	77
16	89	49	15	89	45	60	23	57	64	7	1	63	41	63	47	26	75	77	40
66	58	31	68	78	91	13	59	49	85	85	9	39	41	56	40	54	77	51	31
58	56	20	85	53	35	53	41	69	13	86	72	8	49	47	87	58	18	68	28

[(1)]source : la page web du professeur Eric Taillard http ://mistic.heig-vd.ch/taillard/

Ce tableau se lit de la manière suivante : il faut effectuer 20 tâches. Chaque tâche se décompose en 5 opérations $1, 2, 3, 4$ et 5, identifiées avec 5 machines. Les opérations doivent être faites dans cet ordre (flow-shop), le temps pour faire la $k$ème opération de la tâche j se lit sur l'entrée qui est à l'intersection de la $k$ème ligne et de la $j$ème colonne.

Nous allons voir maintenant
- des cas particuliers polynomiaux,
- un cas *préemptif* : l'open-shop avec un nombre quelconque de machines, qui est encore un problème polynomial.
- un schéma de branch-and-bound pour le job-shop.
- un schéma de recherche locale pour le job-shop.

10.3.2. Quelques cas particuliers polynomiaux. — Les cas particuliers polynomiaux les plus célèbres sont les suivants.

1. Le problème du flow-shop à deux machines et minimisation du makespan $C_{\max}$: *algorithme de Johnson.*
2. Le problème du job-shop à deux machines et minimisation du makespan $C_{\max}$: *algorithme de Jackson.*
3. Le problème du job-shop à deux tâches et minimisation du makespan $C_{\max}$: *algorithme de Brucker.*
4. Le problème de l'open-shop à deux machines et minimisation du makespan $C_{\max}$: *algorithme de Gonzalez et Sahni*

10.3.2.1. Flow-shop non-préemptif à deux machines et minimisation du makespan $C_{\max}$. — C'est résolu par l'algorithme de Johnson [**19**]. On a n tâches $J_1, J_2, \ldots, J_n$. Chaque tâche J_j a une durée p_{j1} sur la machine 1 et p_{j2} sur la machine 2. On doit d'abord passer par la machine 1, puis par la machine 2.

L'algorithme produit une séquence $j_1, j_2, \ldots, j_n$: ce sera l'ordre dans lequel effectuer les tâches sur les deux machines. En effet, comme nous le verrons ci-dessous, il y a toujours une solution optimale pour laquelle les tâches sont réalisées dans le même ordre sur les deux machines.

On définit pour chaque tâche j la durée $q_j = \min(p_{j1}, p_{j2})$. Classer les tâches par q_j croissant. Quitte à renuméroter, on a $q_1 \leq q_2 \leq \ldots \leq q_n$. Commencer à $j = 1$. Fixer $r, s = 0$. Répéter jusqu'à $j = n$:

> si q_j est atteint sur la première machine, définir $j_{r+1} := j$ et $r := r + 1$;
> si q_j est atteint sur la seconde machine, définir $j'_{s+1} := j$ et $s := s + 1$.
> Faire $j := j + 1$.

Retourner l'ordre $j_1, \ldots, j_r, j'_s, \ldots, j'_1$.

Formulations alternatives :

1. Faire deux paquets : un paquet A avec les j tels que $p_{j1} \leq p_{j2}$, un paquet B avec les j tels que $p_{j1} > p_{j2}$. Ordonner A selon les p_{j1} croissants, B selon les p_{j2} décroissants. Concaténer la suite des j de A à celle de B dans cet ordre.
2. **Choisir** la tâche J_j ayant le plus petit p_{j1} ou p_{j2} parmi tous les p_{ji}.

   Appliquer récursivement l'algorithme sur les tâches $\neq j$: cela donne une séquence $j_1, j_2, \ldots, j_r, j'_s, \ldots, j'_1$, avec $r + s = n - 1$.

Les tâches $j_1, \ldots, j_r$ sont choisies de façon à ce que le temps sur la machine 1 soit plus court que sur la machine 2. Les tâches $j'_s, \ldots, j'_1$ sont choisies de façon à ce que le temps sur la machine 2 soit strictement plus court que sur la machine 1.

Si $p_{j1} \leq p_{j2}$, **retourner** $j, j_1, j_2, \ldots, j_r, j'_s, \ldots, j'_1$.

Si $p_{j1} > p_{j2}$, **retourner** $j_1, j_2, \ldots, j_r, j'_s, \ldots, j'_1, j$.

Considérons l'exemple suivant.

Tâches	temps sur la machine 1	temps sur la machine 2
A	12	3
B	14	11
C	25	13
D	9	17
E	6	15

L'algorithme va successivement fixer :

$$\ldots, A$$
$$E, \ldots, A$$
$$E, D, \ldots, A$$
$$E, D, \ldots, B, A$$
$$E, D, C, B, A.$$

Théorème 10.3.2. — *L'algorithme de Johnson résout correctement le problème du flow-shop à deux machines et minimisation du makespan* $C_{\max}$.

Démonstration. — Déroulons maintenant l'algorithme et plaçons nous à l'itération j. On peut supposer sans perte de généralité que q_j est atteint sur la première machine. $j-1$ tâches ont déjà été fixées : pour fixer les notations, supposons que les tâches fixées sont celles d'indices $j_1, \ldots, j_r$ et $j'_1, \ldots, j'_s$. On suppose donc que $j = r + s + 1$. On peut mettre les autres dans un ordre quelconque, pas forcément le même sur la première et la seconde machine. On suppose que l'on a les instants de démarrage de chacune des opérations et que cet ordonnancement est réalisable. On a donc un ordre de tâches sur la première machine $j_1, \ldots, j_r, i_1^{(1)}, \ldots, i_{n-r-s}^{(1)}, j'_s, \ldots, j'_1$ et sur la seconde $j_1, \ldots, j_r, i_1^{(2)}, \ldots, i_{n-r-s}^{(2)}, j'_s, \ldots, j'_1$.

On fait maintenant commencer j sur la machine 1 à l'instant auquel commençait $i_1^{(1)}$, et sur la machine 2 à l'instant auquel commençait $i_1^{(2)}$. On décale vers la droite les tâches qui était à gauche de j sur la machine 1 et sur la machine 2 en conséquence. Cette opération est illustrée sur le Figure 3.

Montrons que ce nouvel ordonnancement est encore réalisable et garde le même makespan. Une fois l'itération j finie, les tâches décalées ont vu leur temps de démarrage augmenter de p_{j1} sur la machine 1, et de p_{j2} sur la machine 2. Comme $p_{j1} < p_{j2}$, on reste réalisable, en tout cas pour les tâches différentes de j. Il faut encore vérifier que la tâche j démarre sur la machine 2 lorsqu'elle a été terminée sur la machine 1. Mais c'est évident car j commence sur la machine 1 au plus tard à l'instant auquel commençait $i_1^{(2)}$ sur la machine 1 et $p_{i_1^{(2)}1} \geq p_{j1}$.

En conclusion, partant d'un ordonnancement quelconque, on peut toujours obtenir un ordonnancement de même makespan tel que les tâches soient classées de la même façon que par l'algorithme de Johsonn.

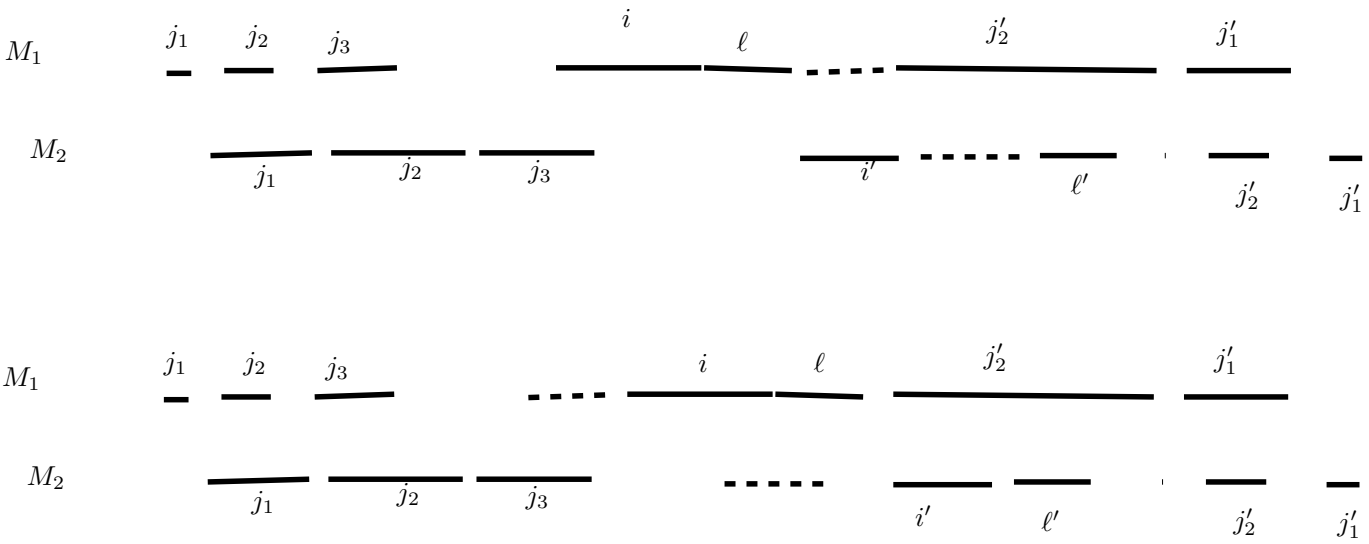


FIG. 3. Illustration d'une itération de l'algorithme de Johnson : changer la position relative de la tâche en pointillés par rapport à i et ℓ ne détériore pas la durée de l'ordonnancement

□

10.3.2.2. Job-shop non-préemptif à deux machines et minimisation du makespan $C_{\max}$. — C'est résolu par l'algorithme de Jackson [**18**], qui est une simple adaptation de l'algorithme de Johnson. On a n tâches $J_1, J_2, \ldots, J_n$. Chaque tâche J_j a une durée p_{j1} sur la machine 1 et p_{j2} sur la machine 2. Pour chaque tâche j, l'ordre de passage sur les machines 1 et 2 est fixé.

L'algorithme consiste à définir

$$\mathcal{J}_{12} := \{J_j : \ J_j \text{ doit d'abord être effectué sur 1 puis sur 2}\}$$

$$\mathcal{J}_{21} := \{J_j : \ J_j \text{ doit d'abord être effectué sur 2 puis sur 1}\}$$

et à faire les opérations suivantes :
– sur la machine 1 : $\mathcal{J}_{12}\mathcal{J}_{21}$,
– sur la machine 2 : $\mathcal{J}_{21}\mathcal{J}_{12}$,
où $\mathcal{J}_{12}$ et $\mathcal{J}_{21}$ sont ordonnés selon l'algorithme de Johnson.

Théorème 10.3.3. — *L'algorithme de Jackson résout correctement le problème du job-shop à deux machines, au plus deux opérations par tâche, et minimisation du makespan $C_{\max}$.*

Démonstration. — Similaire à celle du Théorème 10.3.2. □

10.3.2.3. Job-shop non-préemptif à deux tâches et minimisation du makespan $C_{\max}$. — On a deux tâches et M machines numérotées de 1 à m. Chaque tâche $j = 1, 2$ est décrite par une suite $(O_{ji})_{i=1,\ldots,m}$ d'opérations. On note p_{ji} le temps pour effectuer la $i$ème opération de la tâche j et m_{ji} la machine correspondante.

Pour résoudre ce problème, on peut appliquer l'algorithme de Brucker [**3**], qui est en fait un algorithme de plus court chemin dans un certain graphe.

On se place dans le quart de plan positif. Pour toute paire i, j telle que $O_{1i} = O_{2j}$ (ie la $i$ème opération de la tâche 1 utilise la même machine que la $j$ème opération de la tâche 2 : $m_{1i} = m_{2j}$),

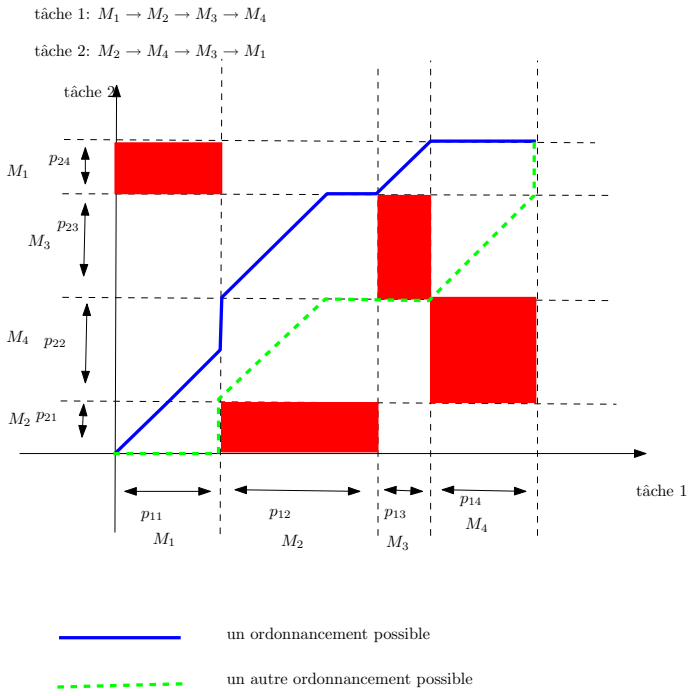


FIG. 4. Illustration de l'algorithme de Brucker qui résout le job-shop à deux tâches et minimisation du makespan.

on dessine un obstacle rectangulaire de coordonnées sud-ouest $\left(\sum_{k=1}^{i-1} p_{1k}, \sum_{k=1}^{j-1} p_{2k}\right)$ et de côtés p_{1i} et p_{2j}.

Un ordonnancement réalisable induit un chemin partant de $(0,0)$ et allant jusqu'à

$$\left(\sum_{k=1}^{m} p_{1k}, \sum_{k=1}^{m} p_{2k}\right).$$

Ce chemin se déplace soit de gauche à droite, soit de haut en bas, soir vers le nord-est. On cherche donc le plus court chemin : c'est un problème de plus court chemin dans un graphe acircuitique – le poids d'un arc étant la durée correspondant à la transition représentée par l'arc – qui conduit à un algorithme en $O(m^2 \log(m))$ (le plus court chemin lui-même met $O(m^2)$, mais la construction du graphe est un peu plus coûteuse).

En toute rigueur, le nombre de chemins respectant ce genre de déplacement est infinis : si la tâche 1 met en jeu la machine 1 sur une certaine plage de temps, et si la tâche 2 doit pendant ce temps passer sur la machine 2, on peut faire débuter **à n'importe quel moment** de la plage horaire l'opération sur la machine 2. La démonstration du Théorème 10.3.4 ci-dessous, omise ici, consiste dans un premier temps à montrer qu'en se limitant à un nombre fini de mouvements possibles (voir Figure 10.3.2.3), on ne diminue pas la longueur du plus court chemin.

D'après cette discussion, on a

Théorème 10.3.4. — *L'algorithme de Brucker résout correctement le problème du job-shop à deux tâches et minimisation du makespan.*

. — □

L'algorithme est illustré sur les Figures 4 et 10.3.2.3.

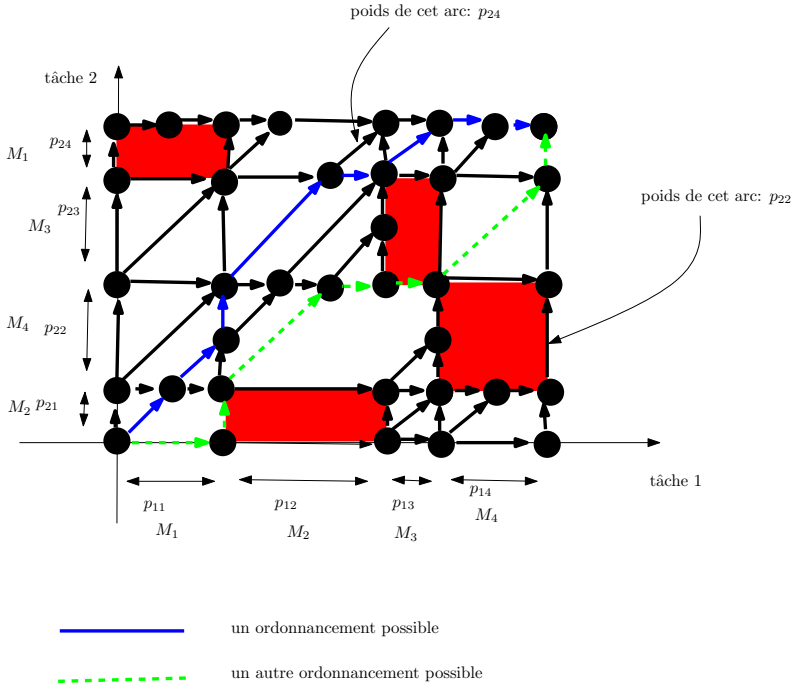


FIG. 5. Graphe acircuitique associé à l'algorithme de Brucker.

10.3.2.3.1. Remarque. — Cet algorithme peut se généraliser au problème à n tâches, mais la complexité devient alors $O(nm^n \log(m))$, ce qui n'est pas polynomial...

10.3.2.4. Open-shop non-préemptif à deux machines et minimisation du makespan $C_{\max}$. — On a n tâches et deux machines. Chaque tâche est constituée de deux opérations, identifiées avec les machines. On note p_{ji} le temps que prend la tâche j sur la machine i pour $i = 1, 2$. Comme on est en open-shop, il n'y a pas de contrainte sur l'ordre dans lequel doivent être faites les opérations d'une tâche. On a alors le résultat suivant.

Théorème 10.3.5. — *Pour ce cas, on a à l'optimum*

$$C_{\max} = \max\left(\max_j(p_{j1} + p_{j2}), \sum_{j=1}^{n} p_{j1}, \sum_{j=1}^{n} p_{j2}\right).$$

Un tel ordonnancement peut se trouver en temps polynomial.

Démonstration. — Il est clair que l'on ne peut pas faire mieux que la valeur T ci-dessus. Montrons qu'on peut l'atteindre. Quitte à ajouter des tâches fictives, on peut supposer que $\sum_{j=1}^{n} p_{j1} = \sum_{j=1}^{n} p_{j2} = T$.

On peut définir j_1 le plus grand indice j tel que

$$(p_{11} + p_{12}) + (p_{21} + p_{22}) + \ldots + (p_{j1} + p_{j2}) \leq T.$$

On considère que cela donne une seule grande tâche J_1^*, avec $p_{11}^* := \sum_{j=1}^{j_1} p_{j1}$ et $p_{12}^* := \sum_{j=1}^{j_1} p_{j2}$.

De même, on peut définir j_2 le plus grand indice j tel que

$$(p_{(j_1+1)1} + p_{(j_1+1)2}) + (p_{(j_1+2)1} + p_{(j_1+2)2}) + \ldots + (p_{j1} + p_{j2}) \leq T.$$

On considère que cela donne une seule grande tâche J_2^*, avec $p_{21}^* := \sum_{j=j_1+1}^{j_2} p_{j1}$ et $p_{22}^* := \sum_{j=j_1+1}^{j_2} p_{j2}$.

Enfin, on définit de même j_3 (on ne peut pas dépasser j_3(on ne peut pas dépasser j_3[(2)]). On considère que cela donne une seule grande tâche J_3^*, avec $p_{31}^* := \sum_{j=j_2+1}^n p_{j1}$ et $p_{32}^* := \sum_{j=j_2+1}^n p_{j2}$.

On se retrouve donc avec l'open shop à trois tâches J_1^*, J_2^*, J_3^*, avec $p_{11}^* + p_{21}^* + p_{31}^* = T$ et $p_{12}^* + p_{22}^* + p_{32}^* = T$, et $p_{j1}^* + p_{j2}^* \leq T$ pour tout $j = 1, 2, 3$.

En renumérotant, on peut supposer que $p_{11}^* \geq p_{22}^*$ et $p_{12}^* \geq p_{31}^*$ (exercice). On vérifie ensuite que l'on peut effectuer J_1^*, J_2^*, J_3^* sur la machine 1 dans cet ordre, et J_2^*, J_3^*, J_1^* sur la machine 2 dans cet ordre. C'est un ordonnancement de durée T (exercice). Pour chaque J_s^*, on effectue les J_j dont il est composé, consécutivement selon les indices. □

10.3.2.5. Open-shop préemptif. — On a n tâches et m machines, chaque tâche j est décrite par un vecteur $\boldsymbol{p}_j = (p_{j1}, \ldots, p_{jm})$, le nombre p_{jk} indiquant le temps que la tâche j doit passer sur la machine k. Chaque tâche peut passer dans un ordre quelconque sur les machines (open-shop). De plus, chaque opération (passage sur une machine) peut être interrompue et reprise plus tard, et ce, autant de fois que nécessaire (cas préemptif). L'important est qu'au total, chaque tâche j ait passé un temps p_{jk} sur la machine k, et ce pour tout $k \in \{1, \ldots, m\}$.

On a le résultat suivant.

Théorème 10.3.6. — *A l'optimum, on a*

$$C_{\max} = \max\left(\max_j\left(\sum_{k=1}^m p_{jk}\right), \max_k\left(\sum_{j=1}^n p_{jk}\right)\right).$$

De plus, un tel ordonnancement se trouve en temps polynomial.

En fait, c'est une application directe du théorème de Birkhoff-von Neumann dont la preuve est laissée en exercice.

***Théorème 10.3.7* (Théorème de Birkhoff-von Neumann)**. — *Soit $A = ((a_{ij}))$ une* matrice carrée bistochastique *(ie $a_{ij} \geq 0$ et $\sum_{i=1}^n a_{ij} = 1$ pour tout j et $\sum_{j=1}^n a_{ij} = 1$ pour tout i).*

Alors A est combinaison convexe de matrices de permutation *(matrices $0-1$ ayant exactement un 1 par ligne et par colonne).*

Preuve du Théorème 10.3.6. — Considérons la matrice $P := ((p_{jk}))$. Définissons

$$A := \begin{pmatrix} \ddots & P \\ P^T & \ddots \end{pmatrix}.$$

C'est une matrice $(n+m) \times (n+m)$. Quitte à ajouter des quantités positives sur la diagonale de A, on peut supposer que A a toutes ses sommes par ligne et par colonne égale à

$$T := \max\left(\max_j\left(\sum_{k=1}^m p_{jk}\right), \max_k\left(\sum_{j=1}^n p_{jk}\right)\right).$$

On applique Birkhoff-von Neumann à A. On obtient que P peut s'écrire comme $T\sum_i \lambda_i Q_i$ où les Q_i sont des matrices ayant au plus un 1 par ligne et par colonne, et où $\lambda_i > 0$ pour tout i et $\sum_i \lambda_i = 1$.

[(2)] En effet, $p_{11}^* + p_{12}^* + p_{(j_1+1)1} + p_{(j_1+1)2} > T$ et $\sum_{j=1}^n p_{j1} + \sum_{j=1}^n p_{j2} = 2T$, donc $\sum_{j=j_1+2}^n p_{j1} + \sum_{j=j_1+2}^n p_{j2} < T$

Cela donne un ordonnancement : sur les $\lambda_1 T$ premiers instants, on effectue les opérations O_{jk} telles que (j,k) soit une entrée non nulle de la matrice Q_1, sur les $\lambda_2 T$ instants suivants, on effectue les opérations O_{jk} telles que (j,k) soit une entrée non nulle de la matrice Q_2, etc. □

10.3.3. Cas non-polynomiaux. — Comme il a été indiqué dans l'introduction de ce chapitre, les problèmes d'ordonnancement sont mal résolus en pratique, contrairement au problème du voyageur de commerce où l'on sait résoudre des instances à plusieurs dizaine de milliers de villes. Pour le job-shop par exemple, il existe à ce jours un grand nombre d'instances à 15 tâches et 15 opérations par tâches (soit des instances occupant l'équivalent mémoire de 225 nombres réels) dont on est très loin de connaître l'optimum.

Les techniques employées en pratique pour résoudre les problèmes d'ordonnancement sont donc en général très sophistiquées et dépassent largement le cadre d'un tel cours. Nous n'allons donc qu'effleurer certaines notions utiles à la conception d'algorithmes branch-and-bound pour le problème job-shop.

10.3.3.1. Graphe disjonctif. — Un outil particulièrement commode pour les cas difficiles, et qui peut servir tant pour les algorithmes de recherche locale que les méthodes exactes du type branch-and-bound est le *graphe disjonctif*.

Le graphe disjonctif est un graphe *mixte* (à la fois des arcs et des arêtes) dont les sommets sont les opérations, les arcs les relations conjonctives (un arc est mis entre deux opérations consécutives de chaque tâche), les arêtes les relations disjonctives (une arête est mise entre toute paire d'opérations utilisant la même machine). La longueur d'un arc est la durée de l'opération dont il est issu.

Considérons par exemple le job-shop suivant, à trois tâches, et trois machines.

J_1	$M_1 : 2$	$M_2 : 5$	$M_3 : 4$
J_2	$M_2 : 1$	$M_1 : 6$	$M_3 : 7$
J_3	$M_3 : 6$	$M_2 : 8$	$M_1 : 3$

Le graphe disjonctif correspondant est donné sur la Figure 6.

La remarque fondamentale qui motive l'introduction du graphe disjonctif est la suivante

Trouver un ordonnancement, c'est orienter les arêtes du graphe disjonctif de manière acircuitique.

En mettant sur chaque arc obtenu par orientation d'une arête la durée de l'opération dont elle est issue, il est facile de calculer le makespan, c'est-à-dire la durée totale de l'ordonnancement. On se retrouve en effet dans la même situation que dans la méthode potentiel-tâche (Section 10.2) où les durées et les précédences sont entièrement fixées.

10.3.3.2. Branch-and-bound. — On définit alors comme dans la Section 10.2, pour toute opération O, la quantité η_O qui est la longueur du plus long chemin de **Début** à O, et de même π_O (en ne gardant que les arcs). Les calculs sont décrits Section 10.2.

On définit $\eta(\mathcal{O}) := \min_{O\in\mathcal{O}} \eta_O$, $\pi(\mathcal{O}) := \min_{O\in\mathcal{O}} \pi_O$ et $p(\mathcal{O}) := \sum_{O\in\mathcal{O}} p(O)$.

Cela permet de définir un schéma de branch-and-bound. On a en effet alors la borne suivante sur la durée des ordonnancements :

$$\max_{k=1,\dots,m} \left(\max_{\mathcal{O}\subseteq\mathcal{O}_k} \eta(\mathcal{O}) + \pi(\mathcal{O}) + p(\mathcal{O}) \right),$$

où $\mathcal{O}_k$ est l'ensemble des opérations devant être effectuées sur la machine k. Carlier a montré qu'elle se calculait en temps polynomial. Le branchement le plus naturel consiste à fixer progressivement les arcs depuis l'origine **Début**.

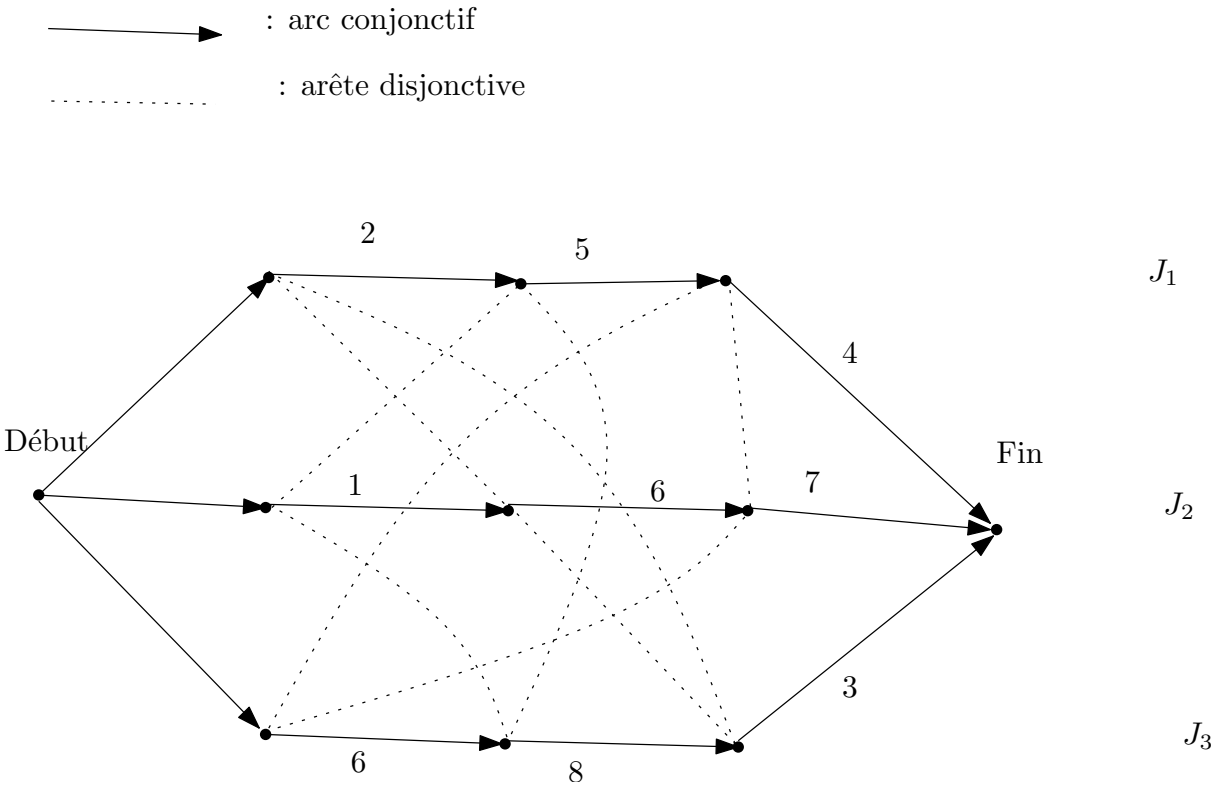


FIG. 6. Un exemple de graphe disjonctif.

10.3.3.3. Recherche locale. — Pour faire de la recherche locale (recuit simulé, tabou, etc.), il faut définir une notion de voisinage sur les ordonnancements réalisables, ces derniers étant identifiés avec les orientations acicuitique du graphe disjonctif. Par exemple, on peut dire que deux ordonnancements réalisables sont *voisins* si on passe de l'un à l'autre par

- retournement d'arc
- retournement des arcs disjonctifs d'un chemin critique.

10.4. Exercices

10.4.1. Management de projet. — Considérons le projet de construction d'immeuble suivant, découpé en différentes tâches. Pour la ligne d'une tâche T donnée, la colonne contraintes indique quelles tâches doivent être terminées pour pouvoir commencer T. Trouver la durée minimale de ce projet. Justifier la réponse.

Tâches	Description	Durée (j)	Contraintes
A	préparation, fondations	6	Début
B	construction des murs	10	A
C	plomberie extérieure	4	B
D	plomberie intérieure	5	A
E	électricité	7	A
F	toit	6	B
G	peinture et finitions extérieures	16	B,C,F
H	lambris	8	D,E
I	sol	4	D,E
J	peinture et finitions intérieures	11	H,I

10.4.2. Ordonnancement de travaux sur un chantier, d'après de Werra, Liebling, Hêche. — Trouver le temps minimum nécessaire pour réaliser les tâches suivantes. Trouver également les chemins critiques.

Tâches	Durées	Contraintes
A Pose de cloisons	10	Début
B menuiserie	12	A, G
C vitrerie	14	B
D construction de l'ossature	10	au plus tôt 2 jours après Début
E plomberie	9	D
F charpente	11	au plus tôt lorsque les 4/5 de D sont terminés et au plus tard 30 jours avant le début B
G pose de la couverture	13	F
H installation des radiateurs	15	E, I
I revêtement des murs et des cloisons	16	C

10.4.3. Cas à une seule machine. — On considère les problèmes d'ordonnancement d'atelier dans le cas à une seule machine. On se met dans le cas non préemptif, chaque tâche est faite d'une seule opération (flow-shop, job-shop et open-shop identiques). Résoudre le minimum makespan, minimum $\sum_j C_j$, minimum $\sum_j w_j C_j$ (où chaque tâche j est munie d'un poids j).

10.4.4. Retour sur le problème de l'affectation de tâche (cours sur les flots). — On revient sur l'exercice d'affectation des tâches, mais cette fois on ne suppose plus que les employés puissent travailler en parallèle.

On suppose que l'on a différentes tâches à accomplir dont on connaît les durées d'exécution, et que l'on dispose d'une ressource de main d'œuvre dont on connaît les compétences. A un instant donné, un employé ne peut se consacrer qu'à une seule tâche à la fois. Enfin, on se met dans un cadre préemptif, c'est-à-dire qu'un employé peut interrompre son travail sur une tâche, et le reprendre plus tard.

On veut trouver un ordonnancement, c'est-à-dire à tout instant la donnée pour chaque employé i de la tâche j sur lequel il travaille. On souhaite minimiser le temps nécessaire pour réaliser l'ensemble des tâches (makespan).

Problème de l'affectation de tâches

Données : n tâches et leurs durées $t_1, \ldots, t_n \in \mathbb{R}_+$; m employés et des sous-ensembles $S_i \subseteq \{1, \ldots, m\}$ qui correspondent aux employés compétents pour la tâche i.

Tâche : un ordonnancement, c'est-à-dire à tout instant la donnée pour chaque employé i de la tâche j sur lequel il travaille, minimisant le temps nécessaire pour réaliser l'ensemble des tâches (makespan).

Montrer que ce problème peut se résoudre en temps polynomial.

10.4.5. Preuve du théorème de Birkoff-von Neumann. — Nous allons montrer le théorème de Birkhoff-von Neumann, qui dit que toute matrice *bistochastique* A (matrice carrée à coefficients positifs, dont la somme des termes de chaque ligne fait 1, et la somme des termes de chaque colonne fait 1) est combinaison convexe de *matrices de permutation* (matrice carrée à coefficients dans $\{0, 1\}$ ayant exactement un 1 par ligne et par colonne).

1. Avec l'aide du théorème de König (qui dit que la cardinalité minimale d'une couverture par les sommets est égale à la cardinalité maximale d'un couplage), montrer le théorème de Hall suivant : *Soit G un graphe biparti, de classes U et V. Supposons que pour tout $X \subseteq U$, on a $|N(X)| \geq |X|$, où $N(X)$ est le voisinage de X dans G. Alors il existe un couplage couvrant U.*

2. En déduire le théorème de Birkhoff-von Neumann (indication : le faire par récurrence sur le nombre d'entrées non-nulles de A, et construire un graphe biparti G dont les classes sont les lignes d'une part, et les colonnes d'autre part.)

10.4.6. Job-shop à deux machines. — Considérons l'instance job-shop non-préemptif suivante, la flèche indique l'ordre des tâches.

	M_1		M_2
J_1	6	$\rightarrow$	3
J_2	5	$\rightarrow$	4
J_3	7		$\emptyset$
J_4	$\emptyset$		3
J_5	2	$\leftarrow$	6
J_6	4	$\leftarrow$	3

Proposer un ordonnancement optimal.

10.4.7. Open-shop à deux machines. — Considérons l'instance open-shop non-préemptif suivante.

	M_1	M_2
J_1	7	0
J_2	3	7
J_3	9	14
J_4	7	3
J_5	7	3
J_6	7	0

Proposer un ordonnancement optimal.

10.4.8. Déchargement et chargement d'une flotte de camions. — Un site d'une entreprise logistique est constitué de deux entrepôts : l'un de déchargement (entrepôt D) et l'autre de chargement (entrepôt C). Le matin, à 6h, 15 camions se présentent, chacun contenant un nombre de caisses connu, indiqué dans le tableau ci-dessous. Chacun de ces camions doit être intégralement vidé dans l'entrepôt D avant d'accéder à l'entrepôt C, où il sera à nouveau chargé avec un nombre de caisses connu, indiqué également dans le tableau ci-dessous. Pour des raisons liées aux effectifs en personnel et aux contraintes de manutention, un seul camion peut être traité à la fois par chaque entrepôt, et lorsqu'on commence à traiter un camion, on ne peut interrompre cette opération. On souhaite finir au plus tôt le traitement des 15 camions, sachant que le temps de chargement et de déchargement est exactement proportionnel au nombre de caisses : 1 minute par caisse. On néglige le temps pour se rendre d'un entrepôt à l'autre. A quelle heure peut-on avoir fini au plus tôt ? Justifier votre réponse.

Camions	Nbre de caisses à décharger	Nbre de caisses à charger
1	2	6
2	1	2
3	20	11
4	2	3
5	7	6
6	6	6
7	2	3
8	1	4
9	8	7
10	6	4
11	12	11
12	5	8
13	7	3
14	6	9
15	5	1

CHAPITRE 11

TOURNÉES

Les problèmes de tournées sont parmi les problèmes les plus naturels en Recherche Opérationnelle et en Logistique. Le thème général est le suivant : un réseau étant donné, il s'agit de le visiter entièrement en minimisant le coût. Les variations sont alors infinies et peuvent concerner la nature du réseau, la notion de visite, la définition du coût. Des contraintes peuvent également être ajoutées à l'envie : fenêtre de temps, retour au point de départ, etc.

Dans cette séance, nous nous intéresserons aux deux problèmes les plus simples à formuler dans cette famille, tous deux d'un grand intérêt pratique : le *problème du voyageur de commerce* et le *problème du postier chinois*. De plus, nous verrons qu'ils illustrent les deux aspects de la recherche opérationnelle car, bien que très semblables dans leur formulation, ils diffèrent radicalement quant à leur résolution puisque l'un d'eux peut être résolu en temps polynomial par un algorithme simple à implanter, et l'autre au contraire est **NP**-difficile et nécessite pour être résolu un mélange de techniques avancées et d'heuristiques.

11.1. Problème du voyageur de commerce

11.1.1. Version non orientée. —

11.1.1.1. Formulation du problème. — Le problème du voyageur de commerce (Traveling Salesman Problem ou TSP en anglais) se formule de la manière suivante.

Problème du voyageur de commerce :

Données : Un graphe $G = (V, E)$ et une fonction de coût sur les arêtes $c : E \to \mathbb{R}_+$.

Demande : Une chaîne fermée C passant par tous les sommets telle que $\sum_{e \in C} c(e)$ soit minimum.

Rappelons que c'est un problème difficile. La preuve a été vue au Chapitre 2.

Théorème 11.1.1. — *Le problème du voyageur de commerce est* **NP***-difficile.*

Il faut donc mettre en œuvre des stratégies de résolutions plus intelligentes, et donc se tourner vers les techniques de la recherche opérationnelle. Nous allons présenter trois approches : algorithme d'approximation, branch-and-bound, recherche locale. Les grands progrès de ces dernières années dans la résolution du problème du voyageur de commerce résident principalement dans les techniques de branch-and-bound, et plus précisément dans le calcul des bornes. On sait de nos jours résoudre des instances à 100000 villes, alors qu'il y a une vingtaine d'année, on ne

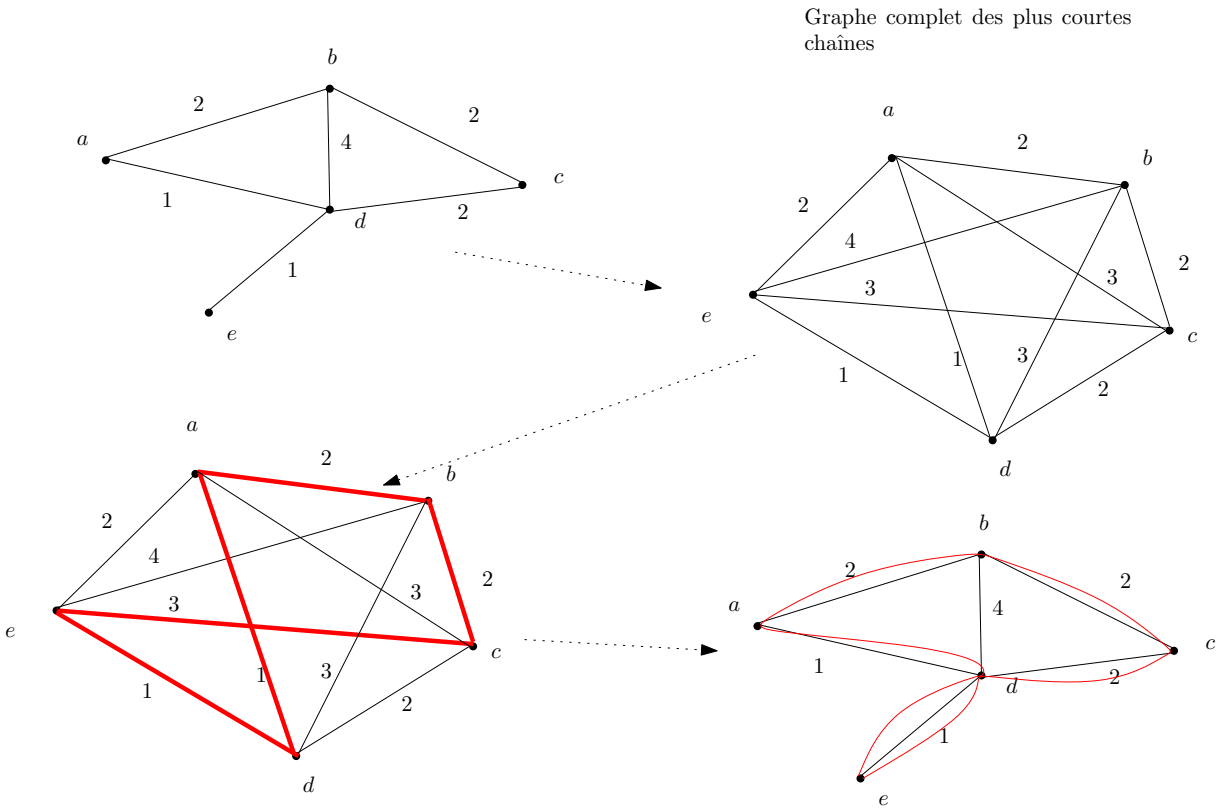


FIG. 1. Le problème du voyageur de commerce peut se reformuler comme un problème de cycle hamiltonien dans un graphe complet.

dépassait pas quelques centaines de villes (la puissance de calcul joue un rôle très secondaire dans ces performances – voir la discussion sur la complexité au début du cours).

Les approches que nous allons décrire s'énoncent plus facilement sous la reformulation suivante sur le graphe complet K_n. Elle s'obtient en prenant le même ensemble de sommets et en mettant comme coût $c(xy)$ le coût du plus court chemin dans G de x à y. En effet, prenons une chaîne fermée de G passant par tous les sommets. La succession des premiers passages en chaque sommet plus le sommet de départ induit un cycle hamiltonien de coût inférieur dans K_n. Réciproquement, un cycle hamiltonien de coût minimum dans K_n induit une chaîne fermée de même coût dans G. Cela permet de conclure que le cycle hamiltonien de plus petit coût dans K_n induit une tournée de coût minimum dans G.

Voir l'illustration de cette reformulation sur la Figure 1.

Problème du voyageur de commerce (reformulation) :

Données : Le graphe complet $K_n = (V, E)$ avec n sommets et une fonction de coût sur les arêtes $c : E \to \mathbb{R}_+$ avec $c(xy) + c(yz) \geq c(xz)$ pour tout $x, y, z \in V$.

Demande : Un cycle hamiltonien de coût minimum.

11.1.1.2. Heuristique de Christofidès. — Le plus célèbre algorithme d'approximation du voyageur de commerce dans le cas non-orienté est l'*heuristique de Christofidès* qui fournit une 3/2-approximation. Selon la terminologie contemporaine, on devrait plutôt utiliser le terme d'algorithme d'approximation, mais la tradition a imposé l'utilisation du terme heuristique.

L'algorithme est très simple (voir l'illustration Figure 2).

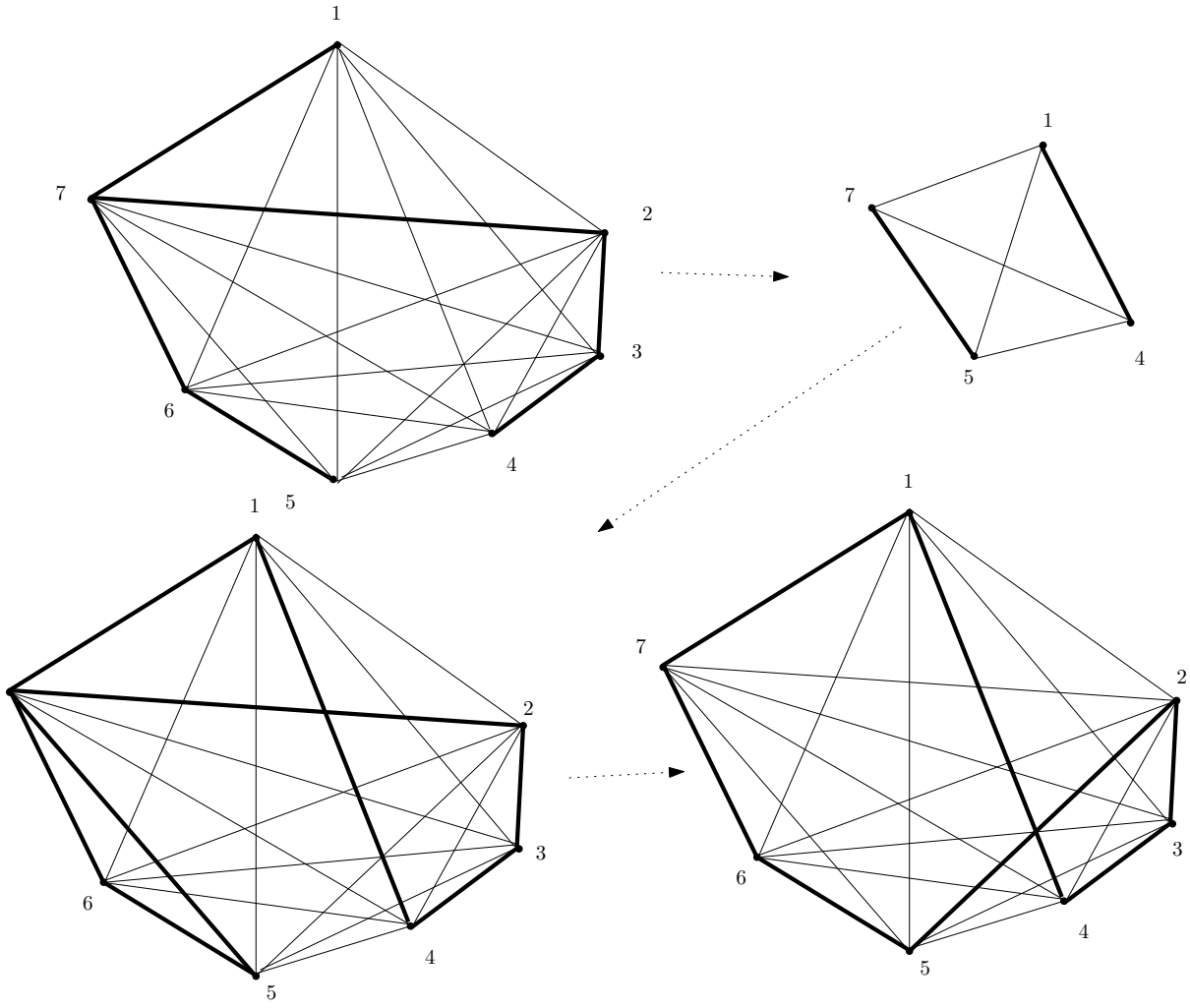


FIG. 2. L'heuristique de Christofidès.

Trouver un arbre couvrant T de poids min. Trouver un couplage M parfait de poids min sur J, où J est l'ensemble des sommets de T de degré impair. $M \cup T$ est un graphe eulérien contenant tous les sommets de K_n, suivre un cycle eulérien, prendre des "raccourcis" quand nécessaire.

Cet algorithme utilise deux routines : celle de l'arbre couvrant de poids minimum, qui est décrite au chapitre sur la conception de réseau (Chapitre 13) ; et celle du couplage parfait de poids minimum, non décrit dans ce cours. Il suffit de savoir qu'un *couplage parfait* est un couplage qui couvre tous les sommets d'un graphe. Si le graphe est pondéré, et s'il existe un tel couplage, on sait trouver un couplage parfait de poids minimum avec un algorithme d'Edmonds.

Théorème 11.1.2. — *L'algorithme décrit ci-dessus fournit une* $3/2$*-approximation au problème du voyageur de commerce.*

Démonstration. — Soit H le cycle hamiltonien obtenu à la fin. Il faut prouver que l'on obtient bien

$$\sum_{e \in H} c_e \leq \frac{3}{2} OPT.$$

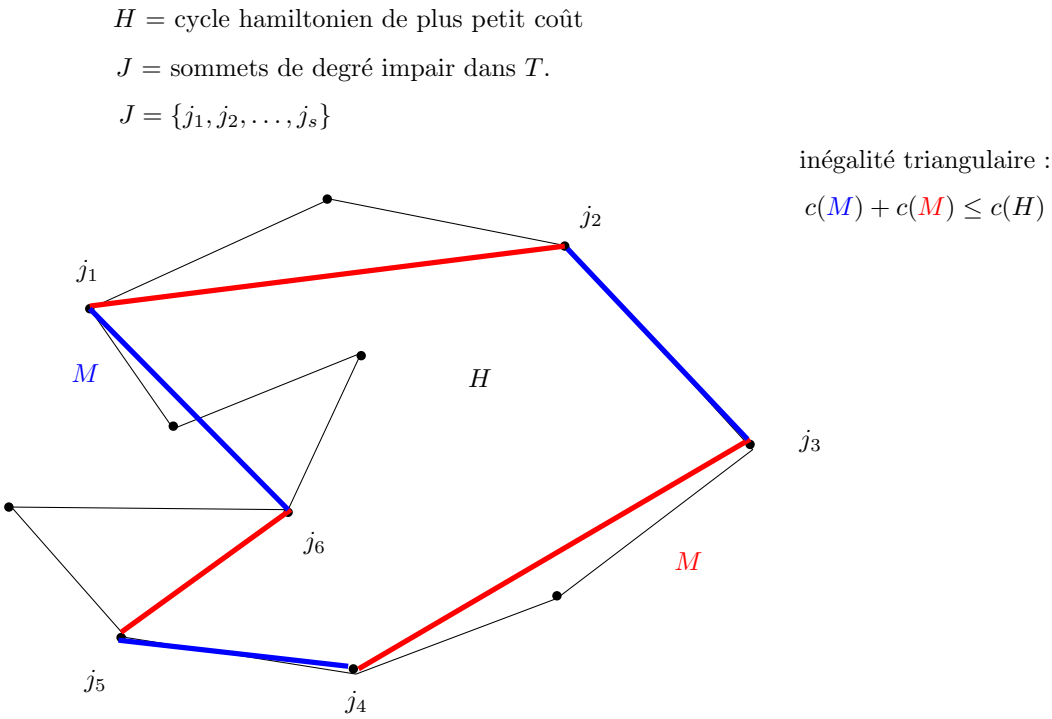


FIG. 3. Illustration de la preuve du facteur d'approximation de l'heuristique de Christofidès.

Pour cela, il faut prouver que le poids de M est $\leq \frac{1}{2}OPT$.

Soit $\tilde{H}$ un cycle hamiltonien optimal. $J = \{j_1, \ldots, j_s\}$ sont les sommets de degré impair de T, numéroté dans l'ordre de parcours de $\tilde{H}$. Considérons le couplage $j_i j_{i+1}$ pour $i = 1, 3, \ldots$, ainsi que son "complémentaire" $j_i j_{i+1}$ pour $i = 2, 4, \ldots$. La somme de leur poids est $\leq OPT$, par inégalité triangulaire. Par conséquent, l'un des deux a un poids $\leq \frac{1}{2}OPT$. A fortiori, c'est le cas pour M (car c'est le couplage de plus petit poids sur J). (La preuve est illustrée Figure 3.)

□

11.1.1.3. Formulation sous forme d'un programme linéaire en nombres entiers.— Rappelons qu'un **vecteur d'incidence** d'une partie A d'un ensemble B est le vecteur $\boldsymbol{x} \in \{0,1\}^B$ où

$$x_e = \begin{cases} 1 & \text{si } e \in A \\ 0 & \text{sinon.} \end{cases}$$

On peut modéliser le problème du voyageur de commerce de façon très compacte sous forme d'un programme linéaire en nombres entiers.

Proposition 11.1.3. — *Les vecteurs d'incidences des cycles hamiltoniens sont exactement les vecteurs entiers du système*

$$\begin{array}{ll} 0 \leq x_e \leq 1 & \textit{pour tout } e \in E, \\ \sum_{e \in \delta(v)} x_e = 2 & \textit{pour tout } v \in V, \\ \sum_{e \in \delta(X)} x_e \geq 2 & \textit{pour tout } X \subseteq V,\ X \neq \emptyset, V. \end{array}$$

Démonstration. — Soit $\boldsymbol{x}$ le vecteur d'incidence d'un cycle hamiltonien. Il est clair que ce vecteur satisfait les contraintes ci-dessus. Réciproquement, soit $\boldsymbol{x}$ un vecteur satisfaisant les contraintes

ci-dessus. Comme

$$\begin{array}{ll} 0 \leq x_e \leq 1 & \text{pour tout } e \in E, \\ \sum_{e \in \delta(v)} x_e = 2 & \text{pour tout } v \in V, \end{array}$$

les arêtes e telles que $x_e = 1$ forment une collection disjointe de cycles élémentaires couvrant tous les sommets du graphe. Il suffit donc de montrer qu'il n'y a qu'un seul cycle dans la collection. C'est une conséquence des inégalités $\sum_{e \in \delta(X)} x_e \geq 2$ pour tout $X \subseteq V$, $X \neq \emptyset, V$. En effet, s'il y a un cycle qui ne couvre pas tous les sommets, en posant X l'ensemble des sommets couverts par ce cycle, on obtient une contradiction. □

Résoudre le problème du voyageur de commerce, c'est donc résoudre le programme

$$\begin{array}{lll} \min & \sum_{e \in E} c(e) x_e & \\ & 0 \leq x_e \leq 1 & \text{pour tout } e \in E, \\ & x_e \in \mathbb{Z} & \text{pour tout } e \in E, \\ & \sum_{e \in \delta(v)} x_e = 2 & \text{pour tout } v \in V, \\ & \sum_{e \in \delta(X)} x_e \geq 2 & \text{pour tout } X \subseteq V,\ X \neq \emptyset, V. \end{array} \tag{30}$$

11.1.1.4. Branch-and-cut. — On peut essayer d'en tirer une borne pour construire une méthode de branch-and-bound. Le problème, c'est que les contraintes sont en nombre exponentiel. On peut en prendre moins de contraintes, de manière à obtenir des bornes calculables en temps raisonnable, mais la qualité des bornes est moindre.

$$\begin{array}{lll} \min & \sum_{e \in E} c(e) x_e & \\ 0 \leq x_e \leq 1 & \text{pour tout } e \in E, & \\ & \sum_{e \in \delta(v)} x_e = 2 & \text{pour tout } v \in V, \\ & \sum_{e \in \delta(X)} x_e \geq 2 & \text{pour } \textbf{certains } X \subseteq V,\ X \neq \emptyset, V. \end{array} \tag{31}$$

Dans les branch-and-bound, la qualité de la borne est cruciale. Une technique efficace, appelée *branch-and-cut*, permet d'améliorer la qualité de la borne. Elle consiste à suivre le schéma d'un branch-and-bound en agrandissant le programme linéaire qui fournit les bornes au fur et à mesure de l'exploration de l'arbre de branchement.

On résout le programme (31). Supposons que l'on ait une solution $\boldsymbol{x}^*$ du programme linéaire. Il y a **3 possibilités**

1. $\boldsymbol{x}^*$ **est le vecteur d'incidence d'un cycle hamiltonien.** On a trouvé une solution optimale pour le nœud courant de l'arbre de branchement.
2. **On trouve une inégalité du type** $\sum_{e \in \delta(X)} x_e \geq 2$ **qui soit violée.** Cela se fait en temps polynomial : c'est un problème de coupe minimum (voir le Chapitre sur les flots). On ajoute la contrainte violée au programme linéaire, et on résout le nouveau programme linéaire.
3. **On ne trouve aucune inégalité de ce type qui soit violée, mais** x^* **n'est pas entier.** On a une borne pour le nœud de l'arbre de branchement. Il faut brancher sur ce nœud et recommencer pour les fils.

11.1.1.5. Relaxation lagrangienne. — Nous avons vu que dans certains cas, la **relaxation lagrangienne** pouvait fournir de bonnes bornes, si une partie des contraintes est "facile".

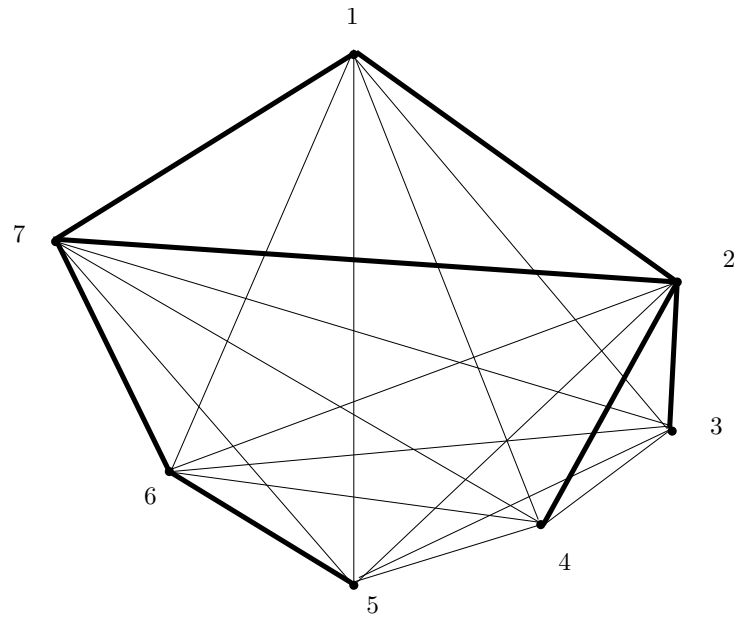


FIG. 4. Un 1-arbre.

C'est le cas ici (Held et Karp 1970). On peut réécrire le système linéaire (30)

$$\begin{array}{ll} 0 \le x_e \le 1 & \text{pour tout } e \in E, \\ x_e \in \mathbb{Z} & \text{pour tout } e \in E, \\ \sum_{e \in \delta(v)} x_e = 2 & \text{pour tout } v \in V, \\ \sum_{e \in E[X]} x_e \le |X| - 1 & \text{pour tout } X \subseteq V,\ X \neq \emptyset, V. \end{array}$$

où $E[X]$ désigne l'ensemble des arêtes induites par X, i.e. l'ensemble des arêtes ayant leurs deux extrémités dans X.

En effet $2\sum_{e \in E[X]} x_e + \sum_{e \in \delta(X)} x_e = \sum_{v \in X} \sum_{e \in \delta(v)} x_e$. Donc, si $\sum_{e \in \delta(v)} x_e = 2$ pour tout $v \in V$, on a $2\sum_{e \in E[X]} x_e + \sum_{e \in \delta(X)} x_e = 2|X|$, d'où l'équivalence des deux formulations. En faisant jouer un rôle particulier au sommet 1, on peut réécrire le système sous la forme

$$\begin{array}{ll} 0 \le x_e \le 1 & \text{pour tout } e \in E, \\ x_e \in \mathbb{Z} & \text{pour tout } e \in E, \\ \sum_{e \in \delta(1)} x_e = 2 & \\ \sum_{e \in E[X]} x_e \le |X| - 1 & \text{pour tout } X \subseteq \{2, \ldots, n\},\ X \neq \emptyset, \\ \sum_{e \in E} x_e = n & \\ \sum_{e \in \delta(v)} x_e = 2 & \text{pour tout } v \in \{2, \ldots, n\}, \end{array}$$

En "oubliant" les contraintes $\sum_{e \in \delta(v)} x_e = 2$ pour $v \in \{2, \ldots, n\}$, on obtient un système linéaire qui décrit les 1-*arbres*. Un 1-arbre est un arbre couvrant $\{2, \ldots, n\}$, avec deux arêtes supplémentaires quittant 1 (voir la Figure 4) - se calcule en temps linéaire (la routine de calcul des arbres couvrants de poids minimum est décrite au chapitre sur la conception de réseau).

On a tout ce qu'il faut pour calculer une borne par relaxation lagrangienne. Le lagrangien est

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) := \sum_{e \in E} c_e x_e + \sum_{i=2}^{n} \lambda_i \left(\sum_{e \in \delta(i)} x_e - 2 \right).$$

La borne inférieure sur le cycle hamiltonien est donnée par

$$\max_{\lambda \in \mathbb{R}^{n-1}} \mathcal{G}(\boldsymbol{\lambda})$$

où

$$\mathcal{G}(\boldsymbol{\lambda}) := -2\sum_{i \in V} \lambda_i + \min_{x \text{ vecteur d'incidence d'un 1-arbre}} \mathcal{L}(x, \lambda)$$

en posant $\lambda_1 := 0$, soit encore

$$\mathcal{G}(\boldsymbol{\lambda}) = -2\sum_{i \in V} \lambda_i + \min_{T \text{ est un 1-arbre}} \sum_{ij \in T} (c_{ij} + \lambda_i + \lambda_j).$$

On sait calculer $\mathcal{G}(\boldsymbol{\lambda})$ en temps polynomial, pour tout $\boldsymbol{\lambda}$, cela suffit (voir Chapitre 7) pour pouvoir calculer $\max_{\boldsymbol{\lambda} \in \mathbb{R}^{n-1}} \mathcal{G}(\boldsymbol{\lambda})$ par une méthode de sur-gradient.

Cette borne est très bonne :

Théorème 11.1.4 **(Held Karp 1970)**. — $\max_{\boldsymbol{\lambda} \in \mathbb{R}^{n-1}} \mathcal{G}(\boldsymbol{\lambda})$ *est égal à la solution de la relaxation linéaire (30).*

La façon classique de brancher pour le voyageur de commerce est de choisir une arête e, et d'écrire l'ensemble des solution $X = X_e \cup (X \setminus X_e)$ où X_e est l'ensemble des solutions empruntant l'arête e. Par conséquent, tout noeud de l'arbre de branch-and-bound est de la forme

$$\mathcal{S}_{A,B} = \{S \in \mathcal{S} : \ A \subseteq S,\, B \cap S = \emptyset\} \quad \text{avec } A, B \subseteq E.$$

Résoudre le problème au niveau du noeud $\mathcal{S}_{A,B}$ consiste donc à chercher le cycle hamiltonien de plus petit coût empruntant toutes les arêtes de A, et aucune de B. Il reste à décrire la façon de calculer des bornes lorsqu'on branche. On pourrait se dire qu'il suffit de calculer des 1-arbres avec des arêtes prescrites (celles de A) et des arêtes interdites (celles de B), mais il n'y a pas d'algorithme simple qui effectue cette tâche. Une petite astuce permet de pallier cette difficulté.

On niveau d'un nœud de l'arbre de branchement, on a donc un problème de voyageur de commerce avec contraintes (ensemble d'arêtes prescrites et interdites). Une petite astuce permet d'écrire ce problème de voyageur de commerce "avec contraintes", comme un problème de voyageur de commerce sans contrainte, simplement en modifiant la fonction de coût.

Si on pose

$$c'(e) := \begin{cases} c(e) \text{si } e \in A \\ c(e) + C & \text{si } e \notin A \cup B \\ c(e) + 2C & \text{si } e \in B, \end{cases}$$

où $C := \sum_{e \in E} c(e) + 1$, on a la propriété (exercice)

Les cycles hamiltoniens de $\mathcal{S}_{A,B}$ sont exactement ceux dont le nouveau coût est $\leq (n+1-|A|)C$. De plus, les cycles hamiltoniens dans $\mathcal{S}_{A,B}$ ont un coût qui diffère de l'ancien coût d'exactement $(n - |A|)C$.

Du coup, on peut calculer des bornes par relaxation lagrangienne pour un problème de voyageur de commerce non contraint, avec la nouvelle fonction de coût, pour obtenir des bornes inférieures pour le problème avec contraintes. D'ailleurs, la borne sur $\mathcal{S}_{A,B}$ s'obtient avec n'importe quelle borne sur le coût optimal d'un cycle hamiltonien, avec les coût $c'(e)$, à la quelle on retranche $(n - |A|)C$.

En pratique, la méthode par relaxation lagrangienne est l'une des plus efficaces.

11.1.1.6. Recherche locale. — On peut aussi se demander si l'on peut attaquer le problème du voyageur de commerce par des méta-heuristiques. IL faut alors définir la notion de **voisinage**. Le **voisinage 2-OPT** est le plus classique. On dit que deux cycles hamiltoniens C, C' sont **voisins** s'ils diffèrent d'exactement deux arêtes $|C \setminus C'| = |C' \setminus C| = 2$. On peut alors intégrer ce voisinage dans n'importe quel schéma de métaheuristique du type "recherche locale" (méthode tabou, recuit simulé,...). Cela a l'avantage d'être simple à programmer, mais les résultats sont très moyens. De plus, on construit facilement des minimums locaux pour ce voisinage.

11.1.2. Version orientée. —

11.1.2.1. Formulation. — On a aussi une version orientée de ce problème :

Problème du voyageur de commerce, version orientée :

Données : Un graphe $D = (V, A)$ avec n sommets et une fonction de coût sur les arêtes $w : A \to \mathbb{R}_+$.

Demande : Un chemin fermé de coût minimum passant par tous les sommets.

De même que dans le cas non-orienté, on peut reformuler la problème sur le graphe complet orienté (toute paire de sommets est reliée par deux arcs de sens opposé).

Problème du voyageur de commerce, version orientée (reformulation) :

Données : Un graphe $K_n = (V, A)$, avec un arc (i, j) pour tout $i \neq j \in V$, et une fonction de coût sur les arcs $c : A \to \mathbb{R}_+$.

Demande : Un circuit hamiltonien C passant par tous les sommets tel que $\sum_{a \in C} c(a)$ soit minimum.

Comme dans le cas non-orienté, on a :

Théorème 11.1.5. — *Le problème du voyageur de commerce dans sa version orientée est* **NP***-difficile.*

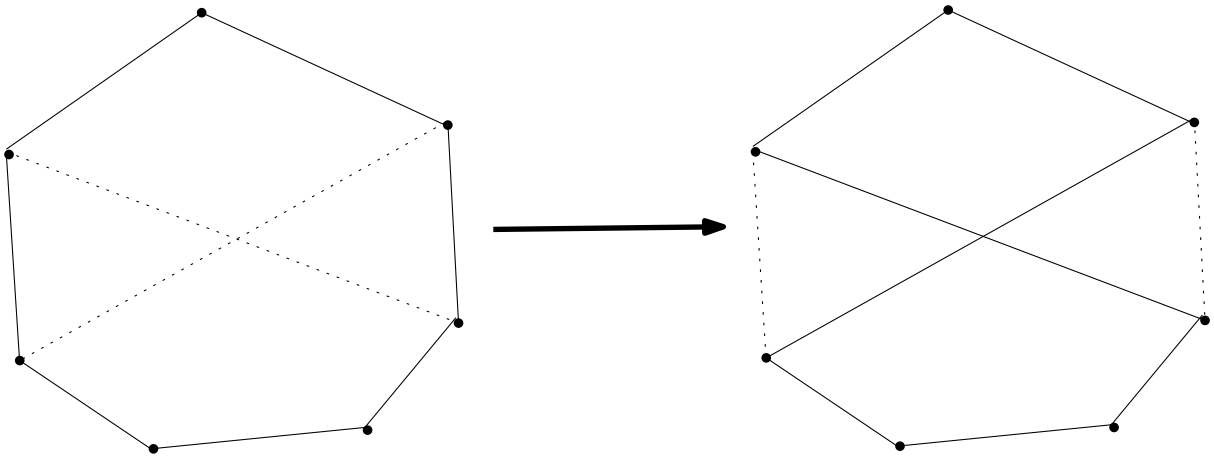


FIG. 5. Illustration du voisinage 2-OPT.

Cela peut se retrouver à partir du Théorème 11.1.1. En effet, en prenant un graphe non-orienté, on peut dédoubler les arêtes pour en faire deux arcs parallèles mais de sens opposés. On se ramène alors au problème du voyageur de commerce sur un graphe orienté, et si on avait un algorithme polynomial le résolvant, on en aurait également un dans la version non-orientée, ce qui contredit le Théorème 11.1.1.

11.1.2.2. Algorithmes. — Ici, la situation est pire que dans le cas du voyageur de commerce non orienté car dans ce cas-ci, l'existence d'un algorithme d'approximation reste une question ouverte.

Comparons avec le cas non-orienté.

- Complexité : pareil, c'est **NP**-difficile.
- Branch-and-Bound : pareil :
  - La formulation par programme linéaire semblable.
  - La relaxation lagrangienne semblable : il faut savoir trouver en temps polynomial un **1-arbre orienté**.
- Algorithme d'approximation : c'est une question ouverte. On ne connaît à ce jour aucun algorithme d'approximation pour le voyageur de commerce dans le cas orienté.
- Recherche locale : même voisinage.

Un *1-arbre orienté* est un ensemble F d'arcs tel que le

- F est un arbre sur $\{2, \dots, n\}$
- $\deg_{\text{in}}(v) = 1$ si $v \in \{2, \dots, n\}$
- $\deg_{\text{in}}(1) = \deg_{\text{out}}(1) = 1$.

On sait trouver un tel ensemble en temps polynomial (théorie des r-arborescences).

11.2. Problème du postier chinois

11.2.1. Version non orientée. —

11.2.1.1. Formulation. — Le problème du postier chinois (Chinese Postman Problem en anglais) se formule de la manière suivante :

Problème du postier chinois :

Données : Un graphe $G = (V, E)$ et une fonction de coût sur les arêtes $c : E \to \mathbb{R}_+$.

Demande : Une chaîne fermée de coût minimum passant par toutes les arêtes au moins une fois.

Ici, G peut être vu comme la modélisation d'une ville, les sommets étant les intersections de rues et les arêtes les portions de rues. Les poids peuvent être le temps de parcours des rues. Le problème est alors celui que se pose un facteur voulant déposer tout son courrier en un minimum de temps.

11.2.1.2. Algorithme. — Contrairement au problème du voyageur de commerce, il existe un algorithme polynomial simple qui trouve la tournée optimale. Remarquons déjà que dans le cas où le graphe est eulérien (voir la définition dans le chapitre d'introduction), la question est triviale.

Sinon, on a la propriété suivante

La chaîne fermée de coût minimum qui passe par toutes les arêtes au moins une fois passe au plus deux fois par chaque arête.

En effet, soit $G = (V, E)$ le graphe. Soit une chaîne C passant par toutes les arêtes au moins une fois. Pour chaque arête e, on construit des arêtes $e_1, \ldots, e_r$ correspondant à chacun des passages sur e. Notons G' ce nouveau (multi)graphe. G' est eulérien. On a donc créé des copies de certaines arêtes de manière à rendre G eulérien. Réciproquement, si on crée des copies d'arêtes pour rendre G eulérien, on a la possibilité de parcourir le multigraphe en passant par chaque arête du multigraphe une fois et une seule, et cela induit sur G une chaîne C passant par toutes les arêtes au moins une fois.

Soit un multigraphe G' eulérien, et soit une arête e présente un nombre s de fois $s \geq 3$. Alors, si on enlève $2p$ représentant de e, avec $p \in \mathbb{N}$ et $s > 2p$, alors G' reste eulérien.

Chercher la chaîne de G qui passe à moindre coût par toutes les arêtes consiste donc à dupliquer à moindre coût des arêtes de G afin de le rendre eulérien. Ici, le coût de la duplication est égal à la somme des coûts des arêtes dupliquées.

On peut donc reformuler le problème du postier chinois de la manière suivante.

Problème du postier chinois

Données : Un graphe $G = (V, E)$ et une fonction de coût sur les arêtes $c : E \to \mathbb{R}_+$.

Demande : Trouver $F \subseteq E$ tel que $\deg_F(v) = \deg_E(v) \bmod 2$ pour tout $v \in V$, et tel que $\sum_{e \in F} c(e)$ soit minimal.

Trouver $F \subseteq E$ tel que $\deg_F(v) = \deg_E(v) \bmod 2$ pour tout $v \in V$ et tel que $\sum_{e \in F} c(e)$ soit minimal : On peut vérifier que, si les $c(e) \geq 0$, un tel F est constitué de chaînes appariant les sommets de degré impairs de G (qui sont en nombre pair).

Un tel F est alors facile à trouver : soit J l'ensemble des sommets de degré impair de G. On considère le graphe complet K sur J, et on met un coût

$$c(jj') = \text{ le coût de la chaîne de plus petit coût entre } j \text{ et } j'$$

sur chaque arête jj' de K. Le couplage parfait de plus petit coût donne alors la solution (on sait trouver un tel couplage en temps polynomial, comme indiqué dans le paragraphe sur l'heuristique de Christofidès).

D'où

Théorème 11.2.1. — *Il existe un algorithme polynomial qui résout le problème du postier chinois.*

11.2.2. Version orientée. —

11.2.2.1. Formulation. — La version orientée existe également. Dans l'exemple de notre facteur, cela permet de prendre en compte l'existence de sens interdits.

Problème du postier chinois, version orientée :

Données : Un graphe $D = (V, A)$ et une fonction de coût sur les arcs $c : A \to \mathbb{R}_+$.

Demande : Un chemin fermé de coût minimum passant par tous les arcs au moins une fois.

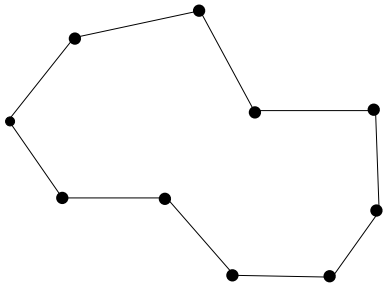


FIG. 6. Une tournée dans le plan.

11.2.2.2. Algorithme. — Et de même que pour la version non orientée, on a un algorithme de résolution exacte en temps polynomial, simple à implanter. En effet, un tel chemin est une *circulation* $\boldsymbol{x} \in \mathbb{R}_+^A$ de coût minimum, étudiée au Chapitre 5, pour des capacités inférieures $l_a = 1$ et des capacités supérieures $u_a = +\infty$ pour tout $a \in A$. (Rappelons qu'une circulation satisfait la loi de Kirchoff en tous les sommets – c'est un flot sans source ni puits). Réciproquement, si on a une circulation $\boldsymbol{x} \in \mathbb{R}_+^A$ entière satisfaisant ces contraintes, et si D est faiblement connexe, en vertu du Théorème 2.2.2, il existe un chemin fermé passant par chaque arc un nombre x_a de fois.

Théorème 11.2.2. — *Il existe un algorithme polynomial qui résout le problème du postier chinois, dans sa version orientée.*

11.3. Exercices

11.3.1. Nombre de sommets de degré pair. — Montrer que dans tout graphe, le nombre de sommets de degré impair est pair.

11.3.2. Système linéaire pour le voyageur de commerce. — Montrer que

$$\begin{array}{ll} x_e \in \{0,1\} & \text{pour tout } e \in E, \\ \sum_{e\in\delta(v)} x_e = 2 & \text{pour tout } v \in V, \\ \sum_{e\in\delta(X)} x_e \geq 2 & \text{pour tout } X \subseteq V,\ X \neq \emptyset, V. \end{array}$$

et

$$\begin{array}{ll} x_e \in \{0,1\} & \text{pour tout } e \in E, \\ \sum_{e\in\delta(v)} x_e = 2 & \text{pour tout } v \in V \\ \sum_{e\in E[X]} x_e \leq |X| - 1 & \text{pour tout } X \subseteq V,\ X \neq \emptyset, V. \end{array}$$

sont deux systèmes équivalents.

11.3.3. Voyageur de commerce dans le plan. — On considère la tournée de la Figure 6. Les distances sont les distances euclidiennes ("à vol d'oiseau"). Montrez que cette tournée est optimale en vous inspirant de la Figure 7.

11.3.4. Voyageur de commerce, cas orienté. —

- Ecrire la relaxation linéaire du problème du voyageur de commerce dans le cas orienté, comme cela a été fait pour le cas non-orienté dans le cours.
- Expliquer pourquoi une relaxation lagrangienne dans ce cas peut conduire au calcul des 1-arbres orientés.

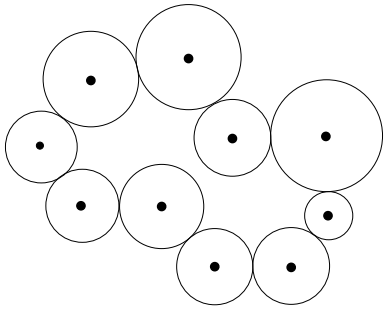


FIG. 7. Cela permet-il de prouver l'optimalité de la tournée de la Figure 6 ?

11.3.5. T-joins. — Soit G un graphe avec des coûts $c : E \to \mathbb{R}$ (contrairement au cours, on accepte des coûts négatifs). Soit T un ensemble de cardinalité paire. Notons E^- l'ensemble des arêtes de coût négatif, T^- l'ensemble des sommets incidents à un nombre impair d'arêtes de E^-, et enfin $d : E \to \mathbb{R}_+$ avec $d(e) := |c(e)|$. On dit que J est un T-*join* si $\deg_J(v)$ est impair si $v \in T$ et pair sinon. (rappel $A \triangle B = (A \setminus B) \cup (B \setminus A)$).

1. Montrer que $c(J) = d(J \triangle E^-) + c(E^-)$.
2. Montrer que J est un T-join si et seulement si $J \triangle E^-$ est un $(T \triangle T^-)$-join.
3. Conclure que J est un T-join de coût minimal pour le coût c si et seulement si $J \triangle E^-$ est un $(T \triangle T^-)$-join de coût minimal pour le coût d.
4. Conclure que l'on sait trouver en temps polynomial un T-join de coût minimum, quelque soit le coût.

11.3.6. Plus courte chaîne dans les graphes sans cycle absorbant. — Utiliser la question précédente pour montrer que l'on sait calculer en $O(n^3)$ une plus courte chaîne dans les graphes sans cycle absorbant (rappel : un cycle absorbant est un cycle de coût total < 0 ; trouver un couplage parfait de coût min se fait en $O(n^3)$).

11.3.7. Tournées avec contraintes de temps. — Un représentant de commerce doit visiter des clients situés en des villes différentes. Il a fixé un rendez-vous pour chacun de ses clients, i.e. pour tout client i, le représentant sait qu'il doit passer après l'instant a_i, mais avant l'instant b_i. Il souhaite optimiser sa tournée (et en passant vérifier que les contraintes ne sont pas contradictoires). On suppose qu'il sait le temps qu'il lui faut pour aller d'une ville à une autre.

Proposer une formulation sous la forme d'un programme linéaire mixte (variables entières et continues), qui, contrairement au problème du voyageur de commerce usuel, ne contient pas un nombre exponentiel de contraintes.

11.3.8. Voyageur de commerce et programmation dynamique. — Considérons l'instance suivante : un graphe complet $K_n = (V, E)$ à n sommets ($n \geq 3$), des coûts positifs $c(vw)$ définis pour toute paire v, w de sommets, et un sommet particulier $s \in V$. On cherche la chaîne hamiltonienne de plus petit coût dont une des extrémités est s.

1. En prenant comme états les couples (X, v) tels que $v \in X \subseteq V \setminus \{s\}$, montrer que l'on peut écrire une équation de programmation dynamique permettant le calcul de la chaîne optimale.

Pour les questions suivantes, on peut se servir des identités indiquées à la fin de l'examen.

2. Quel est le nombre d'états possibles ?

On prend comme opération élémentaire l'addition.

3. Estimez le nombre d'additions que ferait un algorithme exploitant cette équation de programmation dynamique. Comparez ce nombre au nombre d'additions que ferait un algorithme qui énumérerait toutes les solutions.

4. Si votre ordinateur est capable de faire 1 million d'opérations élémentaires par seconde, pour chacune des valeurs suivantes de n, indiquez (par un calcul "à la louche") si vous serez capable de résoudre le problème avec cet algorithme en 1 seconde, 1 heure, 1 jour, 1 semaine, 1 mois, 1 an, 1 siècle :

$$n = 15 \quad n = 30 \quad n = 45.$$

5. Comment utiliser l'algorithme pour résoudre le problème du voyageur de commerce (cycle hamiltonien de plus petit coût) sur K_n ?

CHAPITRE 12

TOURNÉES À PLUSIEURS VÉHICULES

Nous avons eu au Chapitre 11 une première approche des problèmes de tournées. Dans la réalité, le problème de tournée se pose souvent avec une flotte de véhicules, ce qui signifie que l'on peut diminuer la charge d'un véhicule en augentant celle d'un autre. Cette flexibilité ajoute une grosse dose de complexité au problème. Nous allons voir quelques exemples de problème dans cet esprit, sous des versions aussi simples que possible. En particulier, nous nous contenterons de mentionner les problèmes de parcours d'arcs ou d'arêtes, qui, contrairement au cas à un seul véhicule, semblent beaucoup plus durs que les problèmes de parcours de sommets.

12.1. Généralité

Soit $G = (V, E)$ un graphe, muni de coûts $c : E \to \mathbb{R}$. Les sommets sont numérotés $1, 2, \ldots, n$. On dispose d'une flotte de K véhicules, tous situés au sommet 1, aussi appelé *le dépôt*. Tout sommet $i \neq 1$ a une demande d_i. Chaque véhicule a une capacité de u.

On suppose que les véhicules ne reviennent au dépôt qu'à la fin de leur tournée, et que la demande de chaque client n'est satisfaite que par un seul camion

La question qui va nous intéresser est la suivante : Quelle est la tournée des véhicules satisfaisant la demande à coût minimum ?

Ce problème est souvent appelé le problème du *vehicle routing*.

On peut formaliser cette question sous la forme d'un problème, une fois effectuée au préalable la transformation qui ramène les problèmes de tournées à des problèmes de cycles élémentaires sur un graphe complet (voir Chapitre tournées).

Problème du vehicle routing :

Données : Un graphe complet $K_n = (V, E)$ (les sommets sont identifiés à $1, 2, \ldots, n$), une fonction de demande $d : \{2, \ldots, n\} \to \mathbb{R}_+$ sur les sommets, une fonction de coût sur les arêtes $c : E \to \mathbb{R}_+$ (satisfaisant l'inégalité triangulaire) et une capacité u.

Demande : K cycles élémentaires $C_1, C_2, \ldots, C_K$ telles que

- pour tout $k = 1, \ldots, K$ on a $1 \in C_k$.
- pour tout $i \neq 1$ il existe $k \in \{1, \ldots, K\}$ tel que C_k contient i.
- pour tout $k \in \{1, \ldots, K\}$, on a $\sum_{i \in V(C_k)} d(i) \leq u$.
- la quantité $\sum_{k=1}^{K} \sum_{e \in E(C_k)} c(e)$ est la plus petite possible.

La forme d'une solution est illustrée sur la Figure 1.

Ce problème contient le problème du voyageur de commerce (poser $K := 1$ et $u := n - 1$ et $d_i := 1$ par exemple).

Une solution réalisable a 3 véhicules.

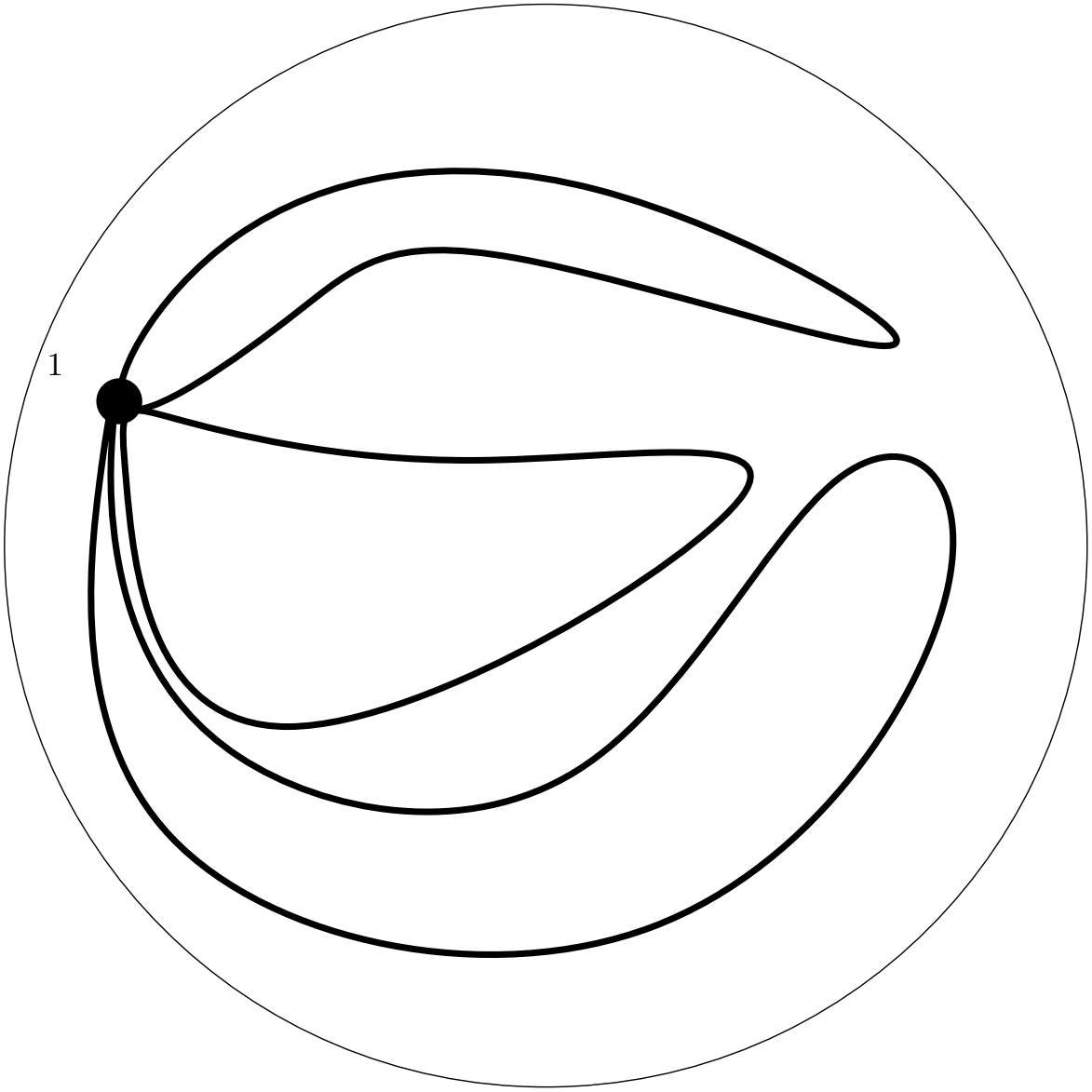


FIG. 1. Une solution du problème du vehicule routing, c'est K cycles élémentaires, passant tous par le sommet 1, et partitionnant le reste des sommets.

Donc, sans surprise

Proposition 12.1.1. — *Le problème du vehicle routing est* **NP***-difficile.*

12.2. Capacité infinie

Si les camions ont une capacité infinie, on peut modéliser le problème du vehicle routing comme un problème de voyageur de commerce.

En effet, il suffit de faire K copies du sommet 1 : $1_1, 1_2, \ldots, 1_K$, de les relier par des arêtes de coût 0, de résoudre le problème du voyageur de commerce sur ce nouveau graphe (à $n+K-1$ sommets), et d'affecter la portion du cycle hamiltonien comprise entre 1_k et 1_{k+1} au véhicule k.

On appelle cette variante le problème du *K-TSP*.

12.3. Branch-and-cut

On peut formuler le problème du Vehicle Routing comme un programme linéaire en nombres entiers

Proposition 12.3.1. — *Le problème du vehicle routing se formule*

$$
\begin{array}{lll}
\text{Min} & \sum_{e\in E} c_e x_e & \\
s.c. & \sum_{e\in\delta(1)} x_e = 2K & \\
& \sum_{e\in\delta(i)} x_e = 2 & \text{pour tout } i \in V\setminus\{1\} \\
& \sum_{e\in\delta(X)} x_e \geq 2\left\lceil \frac{\sum_{i\in X} d_i}{u}\right\rceil & \text{pour tout } \emptyset \neq X \subseteq V\setminus\{1\} \\
& 0 \leq x_e \leq 1 & \text{pour tout } e \notin \delta(1) \\
& 0 \leq x_e \leq 2 & \text{pour tout } e \in \delta(1) \\
& x_e \text{ entier} & \text{pour tout } e \in E.
\end{array} \tag{32}
$$

Démonstration. — Toute tournée réalisable satisfait les contraintes. Voir Figure 2 pour une illustration.

Réciproquement, toute solution réalisable du programme linéaire en nombres entiers (32) fournit une tournée réalisable.

A. il faut vérifier que chaque cycle passe par le dépôt $= 1$.

B. il faut vérifier que chaque cycle n'excède pas la capacité.

Vérification de A. Si C est un cycle dans le support de $\boldsymbol{x} \in \{0,1\}^E$ solution réalisable de (32) qui ne contient pas 1, alors en posant $X :=$ sommets de C, on viole la contrainte $\sum_{e\in\delta(X)} x_e \geq 2\left\lceil \frac{\sum_{i\in X} d_i}{u}\right\rceil$.

Vérification de B. Si C est un cycle dans le support de $\boldsymbol{x} \in \{0,1\}^E$ solution réalisable de (32), alors en posant $X :=$ sommets de C privé de 1, on obtient $2 \geq 2\left\lceil \frac{\sum_{i\in X} d_i}{u}\right\rceil$, i.e. $\sum_{i\in X} d_i \leq u$. □

La formulation (32) présente un certain nombre de difficultés :

1. C'est un programme en nombres entiers.
2. Le nombre de contraintes du type $\sum_{e\in\delta(X)} x_e \geq 2\left\lceil \frac{\sum_{i\in X} d_i}{u}\right\rceil$ est exponentiel.
3. Contrairement à ces contraintes dans le voyageur de commerce, on ne sait pas s'il est facile de trouver rapidement une telle contrainte violée (c'est une question ouverte) – on appelle cette opération *séparer*.

Si l'on met plutôt des contraintes plus faibles $\sum_{e\in\delta(X)} x_e \geq 2\frac{\sum_{i\in X} d_i}{u}$, on sait séparer par un algorithme de flot max (exercice), et donc faire du branch-and-cut comme pour le voyageur de commerce, mais les bornes seront moins bonnes. Rappelons que ce qui est essentiel dans les approches branch-and-bound, branch-and-cut, etc. c'est la qualité de la borne. Les meilleurs algorithmes actuels utilisant la formulation (32) préfèrent utiliser des heuristiques pour séparer les contraintes $\sum_{e\in\delta(X)} x_e \geq 2\left\lceil \frac{\sum_{i\in X} d_i}{u}\right\rceil$ plutôt que de séparer rapidement des contraintes moins bonnes.

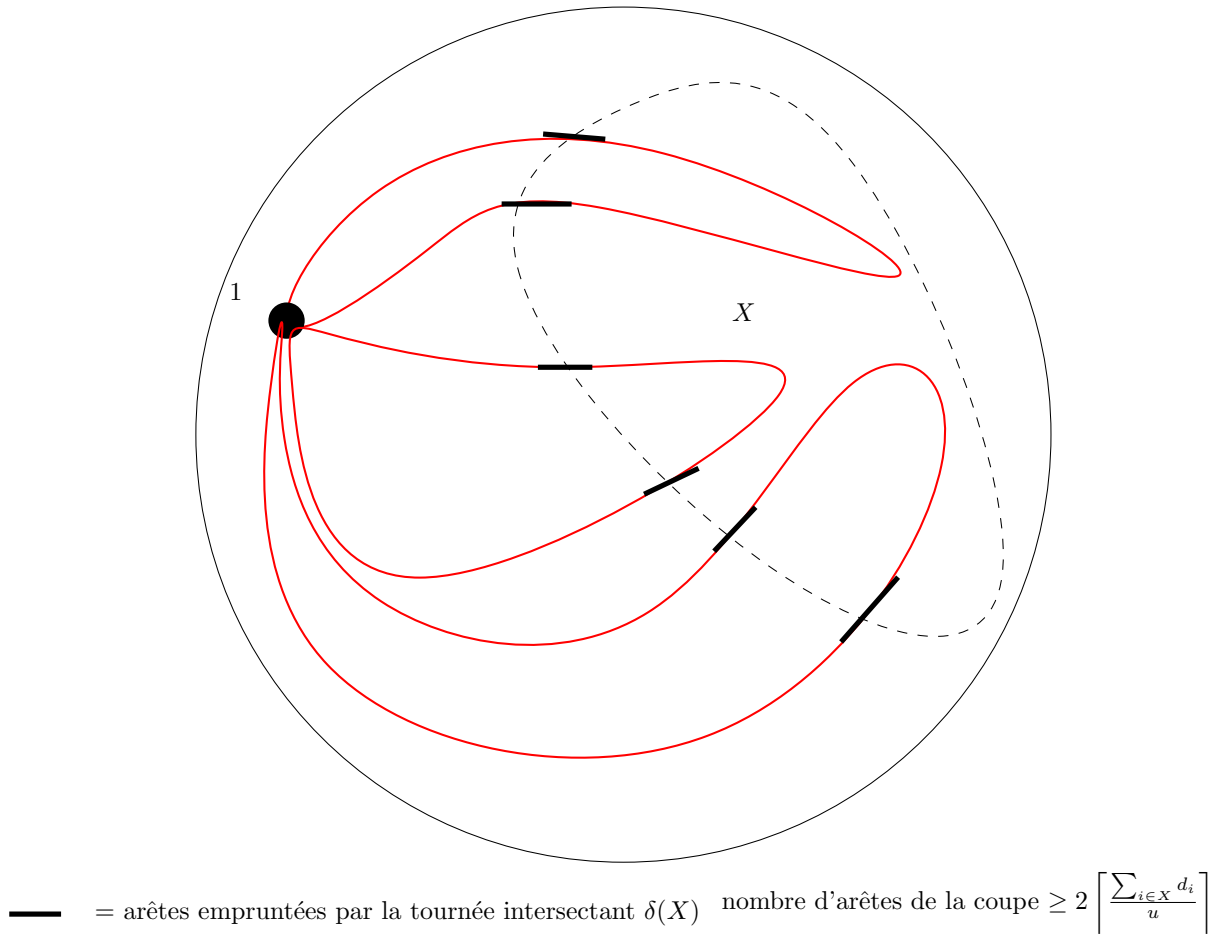


FIG. 2. Toute tournée réalisable satisfait la contrainte de coupe.

12.4. Parcours d'arêtes ou d'arcs à plusieurs véhicules

Si l'on veut résoudre un problème semblable, mais avec parcours de toutes les arêtes ou tous les arcs d'un graphe, les choses semblent moins bien résolues. L'application traditionnel de ces problèmes, que l'on appelle problèmes d'*edge routing* ou *arc routing*, est le ramassage d'ordures ménagères.

On présente le cas où le graphe est orienté. Le cas non-orienté est très semblable.

Problème du vehicle routing pour les arcs :

Données : Un graphe orienté $D = (V, A)$ (les sommets sont identifiés à $1, 2, \ldots, n$), une fonction de demande $d : A \to \mathbb{R}_+$ sur les arcs, deux fonctions de coût sur les arcs $c, c' : A \to \mathbb{R}_+$, l'une correspondant au coût de passage sur l'arc sans traitement de la demande, et l'autre correspondant au coût de passage sur l'arc avec traitement de la demande, et une capacité u.

Demande : K chemins fermés $C_1, C_2, \ldots, C_K$ telles que

- pour tout $k = 1, \ldots, K$ on a $1 \in C_k$.
- pour tout $a \in A$ il existe $k \in \{1, \ldots, K\}$ tel que $a \in C_k$.
- il existe une affectation $\sigma : A \to \{1, \ldots, K\}$ telle que pour tout $k \in \{1, \ldots, K\}$, on a $\sigma(a) \neq k$ si $a \notin C_k$ et $\sum_{a \in A : \sigma(a)=k} d(a) \leq u$.

• la quantité $\sum_{k=1}^{K}\left(\sum_{a\in A(C_k),\sigma(a)\neq k} c(a)+\sum_{a:\,\sigma(a)=k} c'(a)\right)$ soit minimale.

L'affectation σ indique pour chaque arc quel est le véhicule k qui traite cet arc.

Remarquons que si $u=+\infty$, c'est facile, c'est le problème du postier chinois orienté, vu au Chapitre 11. La réduction à un seul véhicule est immédiate. Sinon, le problème est **NP**-difficile.

Actuellement, il semble que les approches exactes (branch-and-cut) pour les problèmes d'edge ou d'arc routing ne parviennent pas à dépasser les instances à 30 arêtes. Les approches privilégiées dans la communauté de la recherche opérationnelle sont du type métaheuristiques.

On peut tout de même écrire ce problème sous forme d'une programme linéaire en nombres entiers, dont la preuve est laissée en exercice.

Rappelons que $A[X]$, pour un sous-ensemble X de sommets, représente l'ensemble des arcs *induits* par X, c'est-à-dire l'ensemble des arcs dont les deux extrémités sont dans X.

Proposition 12.4.1. — *Le problème de l'arc routing se formule sous la forme suivante*

$$
\begin{array}{lll}
\text{Min} & \sum_{k=1}^{K}\left(\sum_{a\in A} c_a x_{ka}+\sum_{a\in A} c'_a y_{ka}\right) & \\
s.c. & \sum_{a\in\delta^+(i)}(x_{ka}+y_{ka})=\sum_{a\in\delta^-(i)}(x_{ka}+y_{ka}) & \textit{pour tout } i\in V\\
& \sum_{k=1}^{K} y_{ka}=1 & \textit{pour tout } a\in A\\
& \sum_{a\in A} d_a y_{ka}\leq u & \textit{pour tout } k\in\{1,\dots,K\}\\
& \sum_{a'\in\delta^+(X)}(x_{ka'}+y_{ka'})\geq y_{ka} & \textit{pour tout } k\in\{1,\dots,K\},\ \textit{pour tout } X\subseteq V\setminus\{1\}\\
& & \textit{et pour tout } a\in A[X]\\
& x_{ka}\in\{0,1\} & \textit{pour tout } a\in A \textit{ et tout } k\in\{1,\dots,K\}\\
& y_{ka}\in\{0,1\} & \textit{pour tout } a\in A \textit{ et tout } k\in\{1,\dots,K\}
\end{array}
$$

12.5. Exercices

12.5.1. Séparation de la contrainte de capacité pour le problème du vehicle routing. — On rappelle la formulation du problème du vehicle routing sous la forme d'un programme linéaire en nombres entiers.

$$
\begin{array}{lll}
\text{Min} & \sum_{e\in E} c_e x_e & \\
\text{s.c.} & \sum_{e\in\delta(1)} x_e=2K & \\
& \sum_{e\in\delta(i)} x_e=2 & \text{pour tout } i\in V\setminus\{1\}\\
& \sum_{e\in\delta(X)} x_e\geq 2\left\lceil\frac{\sum_{i\in X} d_i}{u}\right\rceil & \text{pour tout } \emptyset\neq X\subseteq V\setminus\{1\}\\
& 0\leq x_e\leq 1 & \text{pour tout } e\in E[X]\\
& 0\leq x_e\leq 2 & \text{pour tout } e\in\delta(1)\\
& x_e \text{ entier} & \text{pour tout } e\in E.
\end{array}
$$

On ne sait pas s'il est facile de "séparer" les contraintes

$$\sum_{e\in\delta(X)} x_e\geq 2\left\lceil\frac{\sum_{i\in X} d_i}{u}\right\rceil,$$

i.e. trouver, pour un $\boldsymbol{x}\in\mathbb{R}^E$ fixé, un X tel que l'inégalité précédente soit violée, ou prouver qu'il n'existe pas de tel X.

En revanche, si l'on supprime la partie entière, on obtient des contraintes de la forme

$$\sum_{e\in\delta(X)} x_e\geq 2\frac{\sum_{i\in X} d_i}{u}$$

que l'on sait séparer avec un algorithme de flot maximum. Le prouver.

(Indication : transformer chaque arête en deux arcs opposés, chacun de capacité $\frac{1}{2}ux_e$, et essayer de faire passer un flot maximum de 1 aux autres sommets...)

12.5.2. Relaxations lagrangiennes pour une formulation du Vehicle routing problem dans le cas orienté. — Soit le programme linéaire en nombres entiers suivant :

$$\text{Min} \quad \sum_{k=1}^{K} \sum_{a \in A} c(a) x_a^k$$

$$\begin{array}{lll}
\text{s.c.} & x_a^k \in \{0,1\} & \text{pour tout } a \in A \text{ et tout } k = 1, \ldots, K, \quad (1) \\
& y_a \in \{0,1\} & \text{pour tout } a \in A, \quad (2) \\
& \sum_{k=1}^{K} x_a^k = y_a & \text{pour tout } a \in A, \quad (3) \\
& \sum_{a \in \delta^+(v)} y_a = 1 & \text{pour tout } v \in V \setminus \{1\}, \quad (4) \\
& \sum_{a \in \delta^-(v)} y_a = 1 & \text{pour tout } v \in V \setminus \{1\}, \quad (5) \\
& \sum_{a \in \delta^+(1)} y_a = K & (6) \\
& \sum_{a \in \delta^-(1)} y_a = K & (7) \\
& \sum_{i=2}^{n} \sum_{a \in \delta^+(i)} d_i x_a^k \leq u & \text{pour tout } k = 1, \ldots, K, \quad (8) \\
& \sum_{a \in A[X]} y_a \leq |X| - 1 & \text{pour tout } X \subseteq V \setminus \{1\} \quad (9) \\
& & X \neq \emptyset.
\end{array}$$

1. Montrer que les solutions réalisables sont les solutions réalisable du vehicle routing sur le graphe orienté.
2. Montrer que si l'on relâche les contraintes (3), (8) et (9), on retrouve un problème d'affectation. (écrire à chaque fois la fonction duale).
3. Montrer que si l'on relâche les contraintes (3), (4), (5), (6), (7) et (9), on obtient K problèmes de sac-à-dos.
4. Montrer que si l'on relâche les contraintes (3), (4), (5), (6), (7) et (8), on obtient un problème de forêt couvrante de poids minimum.

D'autres relaxations sont possibles. Toutes ces relaxations montrent la souplesse de l'approche "relaxation lagrangienne".

12.5.3. Formulation PLNE de l'arc routing. — Prouver la Proposition 12.4.1.

CHAPITRE 13

CONCEPTION DE RÉSEAUX

Les réseaux sont omniprésents. Qu'ils soient routiers, ferrés, informatiques, électriques ou gaziers, ils ont souvent des rôles stratégiques et économiques cruciaux, et leur robustesse (i.e. leur capacité à assurer leur mission même en cas de défaillance locale) est une qualité souvent recherchée.

L'objet de ce chapitre est de présenter quelques modèles et algorithmes de base dans le domaine de la conception de réseau robuste – champ essentiel de la recherche opérationnelle.

Nous étudierons deux problèmes :

- celui de l'arbre couvrant de poids minimal : étant donné un graphe pondéré, trouver un arbre inclus dans ce graphe, de poids minimal, tel que tous les sommets soient reliés par l'arbre.
- celui de l'arbre de Steiner de poids minimal, qui généralise le problème précédent : étant donné un graphe pondéré, trouver un arbre inclus dans ce graphe, de poids minimal, tel que tous les sommets d'un sous-ensemble prédéterminé soient reliés par l'arbre.

13.1. Quelques rappels

Un *arbre* est un graphe connexe sans cycle. On a la propriété suivante.

Proposition 13.1.1. — *Tout arbre à n sommets a $n-1$ arêtes.*

Démonstration. — Par récurrence sur n. Si $n = 1$, c'est évident. Pour $n \geq 2$, prendre un sommet de degré 1[(1)]. Retirer ce sommet de l'arbre. Le graphe restant reste un arbre. Par récurrence, il a $n-1$ sommets et $n-2$ arêtes. En remmettant le sommet que l'on vient de retirer, on ajoute un sommet et une arête. □

Un *arbre couvrant* un graphe $G = (V, E)$ est un arbre $T := (V, F)$ avec $F \subseteq E$. En d'autres termes, tous les sommets de G sont couverts par T, et les arêtes de T sont des arêtes de G. Voir une illustration sur la figure 1.

Un graphe sans cycle mais pas forcément connexe est appelé *forêt*.

---

(1) Si on veut être précis, il faut encore prouver qu'un tel sommet existe. Cela a l'air évident,... mais il faut parfois se méfier des évidences en théorie des graphes. Pour le prouver, on prend un sommet quelconque de l'arbre. On suit une chaîne partant de ce sommet en s'interdisant de repasser par un même sommet. Par finitude de l'arbre, la chaîne s'arrête en un sommet v. Si v est de degré ≥ 2, alors il existe une autre arête que par celle par laquelle on est arrivé dont l'autre extrémité est sur la chaîne. Or ceci est impossible, car il y aurait alors un cycle sur l'arbre. Donc le sommet v est de degré 1.

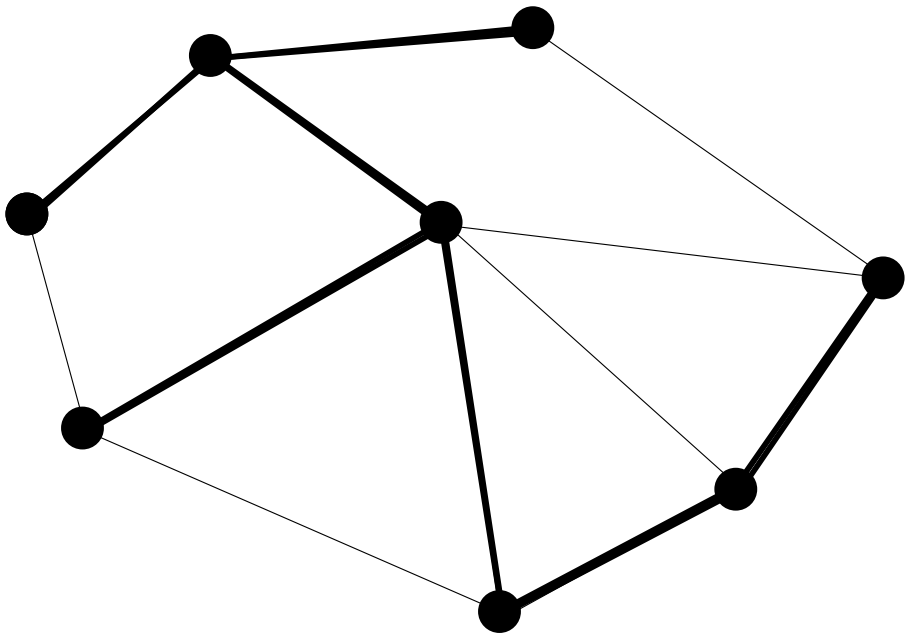


FIG. 1. Un exemple d'arbre couvrant.

13.2. Arbre couvrant de poids minimal

Considérons le problème suivant. On a n villages, les distances entre le village i et le village j sont données dans le tableau suivant ($n = 7$)

	A	B	C	D	E	F	G
A	0	3	8	9	10	5	5
B	3	0	9	9	12	5	4
C	8	9	0	2	10	9	9
D	9	9	2	0	11	9	8
E	10	12	10	11	0	11	11
F	5	5	9	9	11	0	1
G	5	4	9	8	11	1	0

On veut trouver le réseau routier de distance totale minimale qui relie tous les villages, sachant qu'un tronçon relie toujours deux villages (il n'y a pas d'embranchement sur les routes).

On peut modéliser le problème précédent par un graphe complet $K_n = (V, E)$, où V est l'ensemble des villages et E tous les tronçons possibles, ie toutes les paires ij avec $i \neq j$ des éléments de V. On a une fonction de poids $w : E \to \mathbb{R}_+$.

On cherche alors un sous-ensemble $F \subseteq E$ tel que $\sum_{e \in F} w(e)$ soit minimal et tel que le graphe (V, F) soit connexe.

Remarque : Le problème ci-dessus admet toujours une solution qui soit un arbre. En effet, s'il ne l'était pas, il y aurait nécessairement un cycle puisqu'il est connexe. On pourrait alors supprimer une arête de ce cycle sans faire disparaître la connexité et sans détériorer le poids total puisque la fonction de poids est à valeur ≥ 0.

On cherche donc à résoudre un problème d'arbre couvrant de poids minimal.

Une autre application a déjà été vue : les arbres couvrants apparaissent dans la relaxation lagrangienne du problème du voyageur de commerce (borne du 1-arbre), voir Chapitre 11.

Cela justifie la formulation sous forme de problème.

Problème de l'arbre couvrant de poids minimal

Données : Un graphe $G = (V, E)$, une fonction de poids $w : E \to \mathbb{R}$.

Tâche : Trouver un arbre couvrant de plus petit poids.

Pour ce problème, il y a un "petit miracle" puisque nous allons voir qu'il existe un algorithme polynomial glouton qui le résout : l'algorithme de Kruskal. Rappelons qu'un algorithme est dit *glouton* si à chaque itération on fait le meilleur choix local. Ces algorithmes sont rarement optimaux (penser aux algorithmes FIRST-FIT et NEXT-FIT du bin-packing : ils sont gloutons, mais non-optimaux ; de même pour l'algorithme glouton pour le sac-à-dos).

L'algorithme de Kruskal se décrit de la manière suivante.

- Trier les arêtes par poids croissant : $e_1, \ldots, e_m$.
- Poser $F := \{e_1\}$, $i := 1$.
- Répéter

  Faire $i := i + 1$. Si $F \cup \{e_i\}$ ne contient pas de cycle, faire $F := F \cup \{e_i\}$.

Théorème 13.2.1. — *L'algorithme de Kruskal résout le problème de l'arbre couvrant de poids minimal.*

La preuve s'appuie sur le lemme suivant, illustré Figure 2. On dit qu'une forêt $\mathcal{F} = (V, F)$ est *bonne* s'il existe un arbre couvrant $T = (V, F')$ de poids minimal avec $F \subseteq F'$

Lemme 13.2.2. — *Soit $\mathcal{F} = (V, F)$ une bonne forêt, soit une coupe $\delta(X)$ (avec $X \subseteq V$) disjointe de F et e une arête de poids minimal dans $\delta(X)$. Alors $(V, F \cup \{e\})$ est encore une bonne forêt.*

Démonstration. — Soit T un arbre couvrant de poids minimal contenant F. Soit P le chemin reliant dans T les extrémités de e. P intersecte $\delta(X)$ en au moins une arête f. Alors $T' := T \setminus \{f\} \cup \{e\}$ est encore un arbre couvrant. D'après la définition de e, on a $w(e) \leq w(f)$, et donc $w(T') \leq w(T)$. Par conséquent, T' est encore un arbre couvrant de poids minimal. Comme $F \cup \{e\}$ est contenu dans T', c'est encore une bonne forêt. □

Preuve du Théorème 13.2.1. — L'algorithme de Kruskal ajoute à chaque itération l'arête de plus petit poids dans la coupe $\delta(K)$, où K est l'une des deux composantes connexes de $\mathcal{F} = (V, F)$ incidentes à e_i. Comme $\delta(K)$ est disjointe de F, le Lemme 13.2.2 s'applique.

Tout au long de l'algorithme, $(V, F \cup \{e_i\})$ est donc une bonne forêt. En particulier à la dernière itération où la forêt se transforme en arbre. Par définition d'une bonne forêt, cet arbre est optimal. □

Remarque. Cet algoritme fonctionne quelque soit le signe des poids des arêtes. Dans l'exemple introductif, on supposait que les poids étaient tous positifs ; cela pour permettre de conclure que le problème du graphe connexe couvrant de plus petit poids pouvait se ramener au problème de l'arbre couvrant de plus petit poids. Ce dernier problème lui se résout par l'algorithme de Kruskal indépendamment du signe du poids des arêtes.

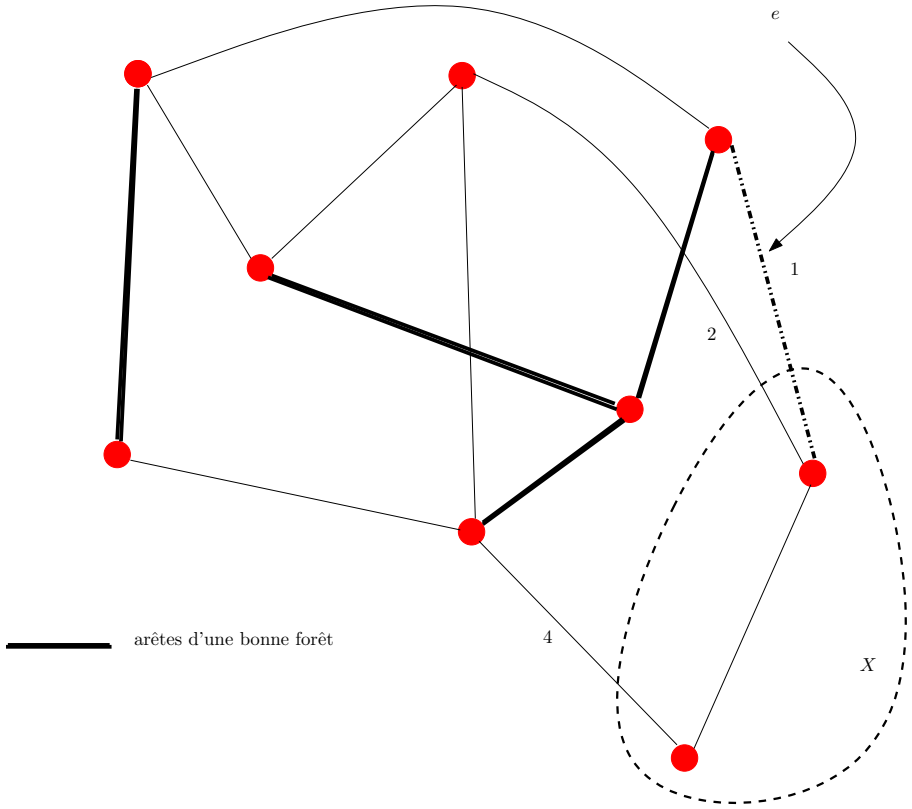


FIG. 2. Une bonne forêt, qui reste une bonne forêt lorsqu'on ajoute l'arête e (on suppose que e est de poids minimal dans $\delta(X)$).

13.3. Arbre de Steiner

13.3.1. Généralités. — Dans le problème de l'arbre de Steiner, c'est seulement un sous-ensemble des sommets du graphe que l'on souhaite voir relié par un arbre. Curieusement, alors que le problème de l'arbre couvrant de poids minimal est polynomial, celui de l'arbre de Steiner est **NP**-difficile.

Le problème se formalise comme suit.

Problème de l'arbre de Steiner

Données : Un graphe $G = (V, E)$, une fonction de poids $w : E \to \mathbb{R}_+$ et un ensemble de sommets $S \subseteq V$, appelés *terminaux*.

Tâche : Trouver un arbre T de plus petit poids couvrant S.

Les applications sont innombrables, et ce problème apparaît fréquemment dans la conception de réseau en tant que tel ou en tant que sous-routine.

Théorème 13.3.1. — *Le problème de l'arbre de Steiner est* **NP***-difficile, même si tous les poids sont égaux à 1.*

Comme d'habitude dans ce cas-là, il faut se poser la question de l'existence d'algorithme d'approximation, d'algorithmes exacts efficaces ou de cadre adapté à la recherche locale. C'est ce que nous allons voir maintenant.

13.3.2. Algorithme 2-approximatif. — Il existe un algorithme polynomial 2-approximatif qui se décrit très simplement.

On suppose $G = (V, E)$ connexe. On construit le graphe complet $K = (S, E')$ sur S et sur chaque arête $uv \in E'$ on met le poids $\bar{w}$ du plus court chemin de u à v dans G (même construction que pour le problème du voyageur de commerce – voir Chapitre 11).

On prend l'arbre T' de plus petit poids couvrant K pour le poids $\bar{w}$.

T' induit un graphe partiel H de G. On en prend un arbre couvrant de plus petit poids w.

L'algorithme est illustré Figure 13.3.2.

Théorème 13.3.2. — *L'algorithme décrit ci-dessus fournit une solution de poids au plus* 2 *fois plus grand que la solution optimale.*

Démonstration. — Il suffit de prouver que $\bar{w}(T')$ est au plus deux fois plus grand que celui de l'arbre de Steiner T de poids minimum $w(T)$.

Soit T l'arbre de Steiner de plus petit poids. Dupliquons chacune des arêtes de T. On a donc maintenant un graphe eulérien que l'on peut parcourir en passant exactement une fois sur chaque arête : cycle C. Cela induit un cycle hamiltonien C' dans K : on parcourt S dans l'ordre de leur première apparition dans C. On a finalement

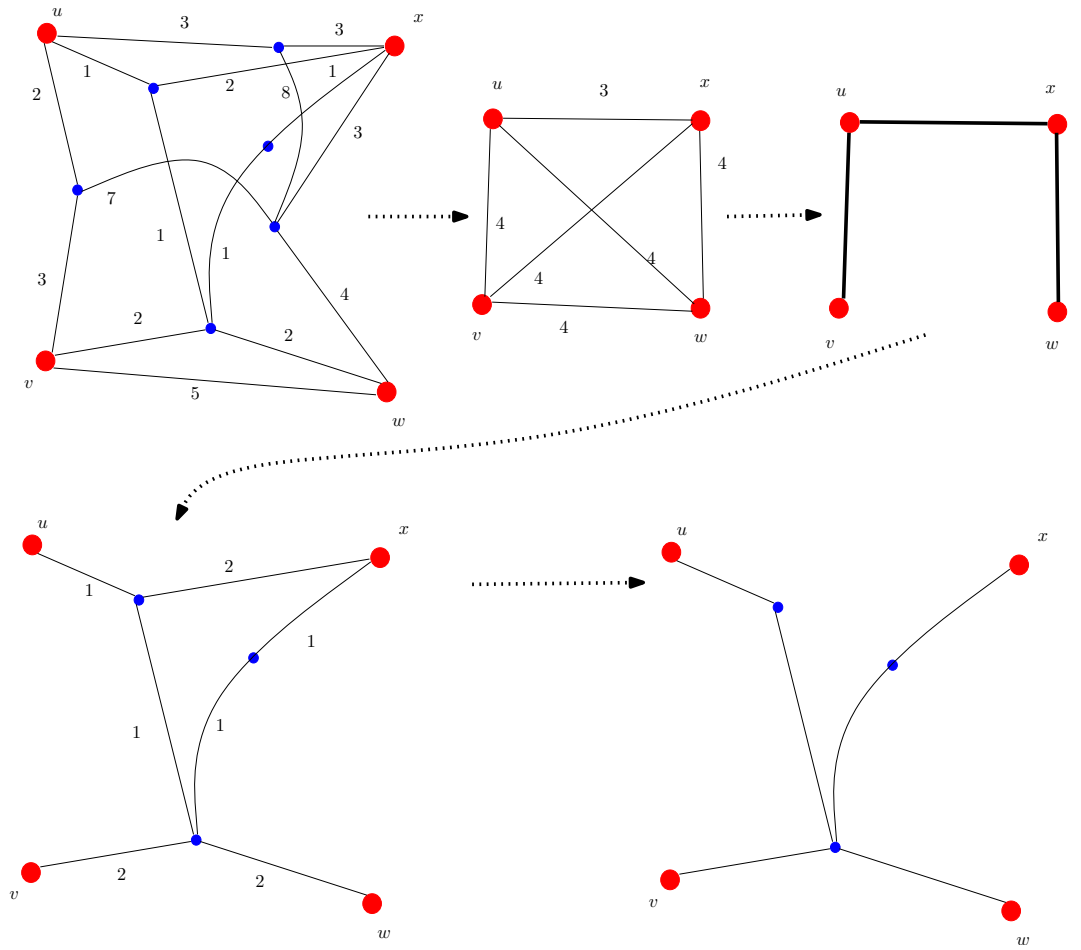


FIG. 3. Illustration de l'algorithme 2-approximatif résolvant le problème de l'arbre de Steiner.

$$\bar{w}(T') \leq \bar{w}(C') \leq w(C) = 2w(T).$$

□

13.3.3. Programmation dynamique. — Dreyfus et Wagner ont trouvé [**7**] une méthode assez astucieuse – basée sur la programmation dynamique (voir Chapitre 3) – qui permet de calculer un arbre de Steiner de poids minimum, quand le nombre d'éléments de S n'est pas trop grand.

En effet, si on pose pour $U \subseteq V$ et $x \notin U$

$$p(U) := \min\{w(T) : T \text{ est arbre de Steiner pour } U\},$$

on a l'équation de programmation dynamique suivante

$$p(U \cup \{x\} = \min_{y \in V, \emptyset \subsetneq U' \subsetneq U} (\mathbf{dist}(x,y) + p(U' \cup \{y\}) + p(U \setminus U' \cup \{y\})). \tag{33}$$

où $\mathbf{dist}(x,y)$ est la *distance* de x à y, i.e. la longueur d'une plus courte chaîne de x à y en prenant w comme longueur des arêtes (voir Chapitre 3 pour ces calculs de plus courts chemins).

Pour voir que l'équation (33) est vraie, supposons d'abord que x est une feuille. Dans ce cas, x est relié par une chaîne à un sommet y qui est tel que

– soit $y \in U$

– soit $y \notin U$ et y de degré au moins 2.

Pour le premier point, l'équation (33) est vraie en posant $U' := \{y\}$. Pour le second point, l'équation est vraie car alors y est le point de rencontre d'une chaîne partand de x, d'un arbre couvrant $U' \cup \{y\}$ et d'un arbre couvrant $U \setminus U' \cup \{y\}$.

Enfin, supposons que x ne soit pas une feuille. Dans ce cas l'équation est vraie pour $y = x$.

L'équation (33) calculées de proche en proche conduit à

Théorème 13.3.3. — *Le problème de l'arbre de Steiner de poids minimum se résout de manière exacte en $O(3^s ns + mn + n^2 \log n)$, où s est le nombre d'éléments de S.*

Elements de preuve. — La preuve de l'équation (33) a été esquissée ci-dessus. Il reste à expliquer la complexité.

Le terme $O(3^s n^2)$ vient du calcul de $p(U)$. Supposons que l'on ait calculé tous les $p(U)$ pour un cardinal $|U| = i$. Le calcul d'un $p(U)$ pour une cardinalité $|U| = i+1$ se fait en $n2^i$ (interprétation de l'équation (33)). La complexité totale est donc de l'ordre de $\sum_{i=0}^{s-1} n2^i \binom{s}{i+1} = O(3^s ns)$.

Le terme $O(mn + n^2 \log n)$ vient du calcul des plus courtes distances. □

Une approche par la programmation dynamique peut donc être pertinente si le nombre s de terminaux est petit.

13.3.4. Branch-and-cut. — L'approche branch-and-cut (voir le Chapitre 11 pour une première approche du branch-and-cut) repose sur la proposition suivante.

Proposition 13.3.4. — *Soit r un sommet arbitraire, $r \in S$. Résoudre le problème de l'arbre de Steiner de coût minimum revient à résoudre*

$$\begin{array}{lll} \min_{x \in \mathbb{R}^E} & \sum_{e \in E} w(e) x_e & \\ s.c. & \sum_{e \in \delta(X)} x_e \geq 1 & \textit{pour tout } X \subseteq V \textit{ tel que } r \notin X \\ & & \textit{et } X \cap S \neq \emptyset \\ & x_e \in \{0,1\} & \textit{pour tout } e \in E. \end{array}$$

Démonstration. — Si $\boldsymbol{x}$ est vecteur d'incidence d'un arbre de Steiner, il est clair qu'il satisfait les inégalités ci-dessus. Réciproquement, soit $\boldsymbol{x}^*$ une solution optimale du programme linéaire. Notons $T := \{e \in E : x_e = 1\}$. D'après l'inégalité $\sum_{e\in\delta(X)} x_e \geq 1$, l'ensemble T est connexe. Et si T contient un cycle C, on peut trouver une arête $e \in C$ telle que $w(e) = 0$, arête que l'on peut supprimer sans que les contraintes se retrouvent violées. □

Le programme linéaire en nombres entiers précédent a deux caractéristiques qui le rende difficile à résoudre.

- il est en nombres entiers.
- il a un nombre exponentiel de contraintes.

On est exactement dans la même situation que pour la formulation PLNE du problème du voyageur de commerce. La relaxation continue lève le premier problème : la solution de ce problème fournit une borne inférieure sur la solution optimale. Cela indique qu'une méthode de branch-and-cut peut être appropriée. Pour résoudre le programme linéaire précédent, on peut utiliser une approche branch-and-cut très semblable à celle suivie pour le voyageur de commerce.

Pour pouvoir appliquer un branch-and-cut, il faut pouvoir trouver s'il existe $X \subseteq V$ avec $r \notin X$ et $X \cap S \neq \emptyset$ tel que $\sum_{e\in\delta(X)} x_e < 1$. Cela se fait par une recherche de coupe minimum : on recherche le v–r flot maximum pour tout $v \in S$ (voir Chapitre 5).

On a donc tout ce qu'il faut pour faire du branch-and-cut. L'algorithme suit le même schéma que celui décrit au Chapitre 11.

13.3.5. Relaxation lagrangienne. — La ressemblance avec les problèmes d'arbres couvrants suggère la modélisation suivante

Proposition 13.3.5. — *Soit r un sommet arbitraire, $r \in S$. Soit v_0 un nouveau sommet, relié à tout sommet de $V \setminus S$ et à r. On étend w par $w(v_0v) := 0$. On appelle $G_0 = (V_0, E_0)$ ce nouveau graphe. Résoudre le problème de l'arbre de Steiner de poids minimum revient à résoudre*

$$
\begin{array}{lll}
\min_{\boldsymbol{x}\in\mathbb{R}^E} & \sum_{e\in E} w(e)x_e & \\
s.c. & x_e \in \{0,1\} & \textit{et est le vecteur indicateur} \\
 & & \textit{d'un arbre couvrant } G_0, \\
 & x_{v_0v} + x_e \leq 1 & \textit{pour tout } v \in V \setminus S \\
 & & \textit{et tout } e \in \delta_G(v)
\end{array}
\tag{34}
$$

Démonstration. — Si $\boldsymbol{x}$ est vecteur d'incidence d'un arbre de Steiner, il est clair qu'il satisfait les inégalités ci-dessus, quitte à ajouter des $x_{v_0v} := 1$, sans modifier le poids. Réciproquement, soit $\boldsymbol{x}^*$ une solution optimale du programme linéaire. Notons $T := \{e \in E : x_e = 1\}$. En ne gardant que les arêtes qui sont non incidentes à v_0, on obtient un arbre de Steiner de même coût. □

Comme d'habitude pour ce genre de problème, on peut avoir de meilleures bornes en utilisant la relaxation lagrangienne, ce qui peut être cruciale pour les problèmes de grande taille. Il faut donc se demander si la suppression de certaines contraintes du programme (34) rend le problème soluble.

Il est clair que la suppression des contraintes $x_{v_0v}+x_e \leq 1$ rend le problème polynomial, grâce à l'algorithme de Kruskal. On écrit donc le lagrangien.

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) := \sum_{e\in E} w(e)x_e + \sum_{v\in V\setminus S} \sum_{e\in\delta(v)} \lambda_{v,e}(x_{v_0v} + x_e - 1).$$

Le problème de l'arbre de Steiner consiste à trouver $\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda})$. On peut minorer cette quantité par $\max \mathcal{G}(\boldsymbol{\lambda})$, avec $\mathcal{G}(\boldsymbol{\lambda}) := \min_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda})$, ie :

$$\begin{array}{ll} \mathcal{G}(\boldsymbol{\lambda}) := & \\ \min & \sum_{e\in E} w(e)x_e + \sum_{v\in V\setminus S} \sum_{e\in\delta_G(v)} \lambda_{v,e}(x_{v_0v} + x_e - 1) \\ \text{s.c.} & x_e \text{ est le vecteur indicateur d'un arbre couvrant } G_0, \end{array}$$

qui peut se réécrire...

$$\mathcal{G}(\boldsymbol{\lambda}) := -\sum_{v\in V\setminus S} \sum_{e\in\delta(v)} \lambda_{v,e} + \text{poids minimal d'un arbre couvrant } G_0$$

où l'arbre couvrant G_0 est calculé avec les poids $w'_{uv} := w_{uv} + \lambda_{u,uv}\mathbf{1}(u \notin S) + \lambda_{v,uv}\mathbf{1}(v \notin S)$ si $uv \in E$ et $w'_{v_0v} := \sum_{e\in\delta(v)} \lambda_{v,e}$.

$\mathcal{G}(\boldsymbol{\lambda})$ se calcule donc aisément par l'algorithme de Kruskal, ainsi que ses sur-gradients. On sait donc maximiser $\mathcal{G}(\boldsymbol{\lambda})$, et on peut calculer de bonnes bornes inférieures.

Pour le branchement, c'est simple : les solutions réalisables associées à un nœud de l'arbre de branch-and-bound correspond à l'ensemble des arbres avec un sous-ensemble d'arêtes imposé. Il est en effet facile de voir que trouver l'arbre couvrant de poids minimum avec des arêtes imposées se fait encore avec l'algorithme de Kruskal. La situation est ici plus simple que dans le cas du voyageur de commerce où pour conserver la borne du 1-arbre en fixant des arêtes et en en interdisant d'autres il faut modifier la fonction de coût.

13.3.6. Recherche locale. — Classiquement, les algorithmes de recherche locale pour le problème du Steiner tree s'appuient sur la remarque suivante : Une fois fixés les sommets $V' \subseteq V$ de l'arbre de Steiner (on a bien sûr $T \subseteq V'$), les arêtes s'obtiennent par un simple calcul d'arbre couvrant de poids minimum.

Le voisinage se construit sur les parties V' de V : on dit que V' est *voisin* de V'' si l'on est dans une des ces situations

1. $V'' := V' \setminus \{v'\}$ pour un $v' \in V'$, (*drop*)
2. $V'' := V' \cup \{v''\}$ pour un $v'' \in V \setminus V'$ (*add*) ou
3. $V'' := V' \setminus \{v'\} \cup \{v''\}$ (*swap*).

13.4. Quelques remarques pour finir

Reprenons le problème des villages du début, mais maintenant autorisons-nous des embranchements. On veut donc convevoir le réseau routier de distance totale minimale tel que tous les villages soient reliés.

Ici, contrairement à la situation du début, la liste des tronçons possibles est potentiellement infinie, l'input est alors plutôt les coordonnées des n villages. Ce genre de problème justifie la formalisation du problème suivant.

Problème de l'arbre de Steiner euclidien

Données : t points de coordonnées $(x_1, y_1), \ldots, (x_t, y_t)$.

Tâche : L'arbre de plus petites longueur reliant tous les points.

Sans surprise, on a

Théorème 13.4.1. — *Le problème de l'arbre de Steiner euclidien est* **NP***-difficile.*

FIG. 4. A gauche, l'input d'un problème de Steiner euclidien, à droite l'output.

Remarquons que l'arbre de Steiner de plus petite longueur est le réseau de plus petite longueur reliant tous les points.

Les heuristiques de résolutions, les algorithmes, etc. s'appuient sur la propriété suivante. On appelle *point de Steiner* une extrémité de segment qui ne soit pas un des points de l'input.

Proposition 13.4.2. — *Les points de Steiner sont toujours de degré 3. On ne peut avoir deux segments se rencontrant en un angle de strictement plus de 120°.*

Pour une illustration, voir Figure 4.

Tout comme le problème de l'arbre de Steiner sur un graphe, le problème de l'arbre de Steiner euclidien est très étudié. On connaît également des algorithmes d'approximation (avec des facteurs aussi proches de 1 que l'on veut !(2)), des branch-and-bound efficaces ou des méta-heuristiques du style recherche locale.

Un autre problème très étudié – avec des résultats semblables – est le problème de l'arbre de Steiner "Manhattan", en ne s'autorisant que les segments horizontaux et verticaux. Application : conception de circuits intégrés.

Toujours dans le domaine des circuits intégrés, on s'intéresse beaucoup au problème des arbres de Steiner disjoints.

13.5. Exercices

13.5.1. Réseau connexe minimal. — On cherche le plus petit réseau reliant les points A, B, C, D, E, F, G. On n'a uniquement droit au lien direct (pas d'embranchement). Les distances sont indiquées dans le tableau suivant.

	A	B	C	D	E	F	G
A	0	10	5	6	7	9	3
B	5	0	8	7	6	2	1
C	3	3	0	11	4	1	8
D	2	4	7	0	9	7	10
E	2	3	8	6	0	3	1
F	6	7	9	2	3	0	4
G	5	5	7	5	1	2	0

Trouver ce réseau.

(2) on parle alors de *schéma d'approximation*

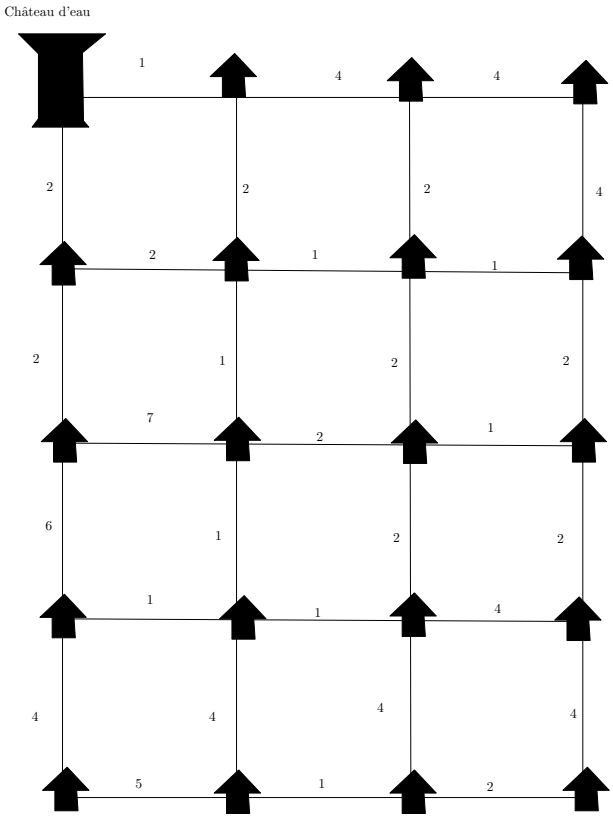


FIG. 5

13.5.2. Un réseau d'eau. — Un ensemble de pavillons doit être relié par un réseau de canalisation d'eau au moindre coût à un château d'eau. Voir Figure 5. Sur la figure, les liaisons possibles sont indiquées par la présence d'un lien entre deux pavillons. Le coût d'ouverture d'une liaison est indiqué à proximité de ce lien.

Mettez en évidence un réseau de moindre coût sur la figure, indiquez son coût et justifiez l'optimalité de la solution que vous proposez.

13.5.3. Plus grand poids le plus petit possible. — Considérons un graphe $G = (V, E)$ muni de poids $w : E \to \mathbb{R}$. On cherche l'arbre couvrant dont le poids maximal d'une arête est le plus petit possible. Comment peut-on le trouver ?

13.5.4. Produit des poids. — Considérons un graphe $G = (V, E)$ muni de poids $w : E \to \mathbb{R}_+^*$ strictement positifs. Peut-on trouver en temps polynomial l'arbre couvrant dont le produit des poids est minimal ?

13.5.5. Graphe partiel connexe couvrant. — Considérons un graphe $G = (V, E)$ muni de poids $w : E \to \mathbb{R}$. Peut-on trouver en temps polynomial le graphe partiel connexe couvrant de poids minimal ?

Rappelons qu'un *graphe partiel* H d'un graphe $G = (V, E)$ est tel que $H = (V', F)$ avec $V' \subseteq V$ et $F \subseteq E$. Si $V' = V$, on dit qu'il *couvre* G.

13.5.6. Transmission optimale de message, d'après Ahuja, Alimanti et Orlin [1]. — Un service d'espionnage a n agents dans un pays ennemi. Chaque agent connaît certains

des autres agents et peut orgainer une rencontre avec toute personne qu'il connaît. Pour toute rencontre entre l'agent i et l'agent j, la probabilité que le message tombe entre des mains hostiles est de p_{ij}. Comment transmettre le message à tous les agents en minimisant la probabilité d'interception ?

13.5.7. Matroïdes (pour les élèves ayant un goût pour les maths). — Soit E un ensemble fini et soit $\mathcal{I}$ une collection de parties de E. Si $\mathcal{I}$ satisfait

(i) si $I \in \mathcal{I}$ et $J \subseteq I$, alors $J \in \mathcal{I}$.

(ii) si $I, J \in \mathcal{I}$ et $|I| < |J|$, alors $I \cup \{z\} \in \mathcal{I}$ pour un certain $z \in J \setminus I$.

alors $(E, \mathcal{I})$ est appelé un *matroïde*.

1. Soit $G = (V, E)$ un graphe connexe. Montrer que $(E, \mathcal{F})$ où $\mathcal{F}$ est l'ensemble des forêts de G est un matroïde.
2. Soit $E = E_1 \cup E_2 \cup \ldots \cup E_k$ l'union de k ensemble disjoints, et soient $u_1, u_2, \ldots, u_k$ des entiers positifs. Montrer que $(E, \mathcal{I})$ où $\mathcal{I}$ est l'ensemble des parties I de E telles que $I \cap E_i \leq u_i$ est un matroïde (appelé *matroïde de partition*).
3. Soit M une matrice réelle. Soit E l'ensemble des colonnes de M et soit $\mathcal{I}$ les sous-ensembles de colonnes indépendantes (pour l'agèbre linéaire). Montrer que $(E, \mathcal{I})$ est un matroïde (appelé *matroïde matriciel*).
4. Soit $(E, \mathcal{I})$ un matroïde. Supposons donnée une fonction de poids $w : E \to \mathbb{R}$. Montrer que trouver l'indépendant de poids maximum peut se faire par l'algorithme glouton.

13.5.8. Design de réseau - un seul bien. — Supposons que l'on souhaite calibrer un réseau de transport de façon à pouvoir assurer des livraisons depuis des sources jusqu'à des destinations. Pour chaque tronçon direct (u, v), on connaît le coût $c(u, v)$ d'établissement d'une liaison de capacité unitaire. Comment construire le réseau dont l'établissement soit de coût minimum ? On suppose que l'on connaît pour les sources, l'offre et pour les destinations, la demande. Modéliser ce problème comme un problème de flot.

13.5.9. Design de réseau - plusieurs biens. — On dispose d'un réseau orienté $D = (V, A)$. Ce réseau a des sources $s_1, \ldots, s_k$ et des destinations $t_1, \ldots, t_k$. Le bien i circule de la source s_i à la destination t_i. Ce bien est modélisé par un flot $x^i : A \to \mathbb{R}_+$.

On doit sélectionner maintenant un sous-ensemble A' de A tel que chaque bien puisse circuler de sa source à sa destination.

Le coût de sélection de l'arc a se note f_a et le coût d'utilisation de l'arc a pour une unité de bien i se note c_a^i. On veut trouver A' tel que le coût total (sélection des arcs et transport du flot) soit le plus petit possible.

1. Modéliser ce problème sous forme d'un programme liénaire en nombres entiers, en les variables x_a^i, pour $a \in A$ et $i = 1, \ldots, k$, et y_a, pour $a \in A$, où y_a code la sélection de l'arc A dans la conception du réseau.

2. Proposer une relaxation lagrangienne qui permette de calculer de bonnes bornes inférieures.

CHAPITRE 14

OUVERTURE

En conclusion, signalons les outils de la recherche opérationnelle ainsi que des domaines à la frontière que nous avons simplement évoqués, par manque de temps et de place, mais qui sont extrêmement importants.

Ces outils sont les **méthodes de décomposition**, les **méthodes de coupes**, l'**optimisation convexe** et la **simulation**.

Ces domaines sont les **statistiques**, les **files d'attente** et la **théorie des jeux**.

14.1. Quelques outils absents de ce livre

14.1.1. Les méthodes de décomposition. — Souvent associé au nom de Bender qui initia ces techniques, les méthodes de décomposition traitent des cas où le nombre de variables est très grand, voire exponentiel en la taille du problème. Très schématiquement, dans le cas où le problème se formule sous la forme d'un programme linéaire (P), ces méthodes procèdent de la façon suivante (on parle alors de *génération de colonne* – le mot « colonne »étant alors synonyme de « variable »comme dans l'algorithme du simplexe) :

1. ne travailler que sur une partie des variables, on appelle ce problème (P') ;
2. résoudre le dual (D') de ce problème réduit ; de deux choses l'une
   - soit la solution de ce dual satisfait toutes les contraintes du dual (D) de (P) auquel cas, la solution primale optimale de (P') est également solution optimale de (P), et l'algorithme est fini,
   - soit la solution de ce dual ne satisfait pas toutes les contraintes de (D). On sélectionne alors une des contraintes violées (par exemple, la plus violée, ce qui nécessite souvent un autre problème d'optimisation), on ajoute la variable correspondante à (P') ce qui donne naissance à un nouveau programme réduit. On retourne en 1.

14.1.2. Les méthodes de coupes. — Les méthodes de coupes sont utilisées pour la programmation linéaire en nombres entiers. On peut écrire le problème de la programmation linéaire en nombres entiers sous la forme $\min(\boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{x} \in P \cap \mathbb{Z}^n)$, où P est un polyèdre.

Les méthodes de coupes consistent à ajouter des contraintes linéaires qui sont vérifiées par tous les points entiers de P, mais pas par tous les points de P. Comme on cherche une solution entière, on ne se prive pas de solution, et la relaxation continue est meilleure. Il existe des techniques de génération automatique de contraintes linéaires (coupes de Dantzig, Gomory, Chvatal,...), on peut aussi trouver de telle coupe par des méthodes ad hoc, adaptées au problème particulier auquel on s'intéresse. On a vu deux exemples dans les exercices du Chapitre 8.

14.1.3. Optimisation convexe. — La généralisation immédiate de la programmation linéaire est la programmation convexe, dont les contraintes et l'objectif sont convexes. Dans le domaine du transport de gaz ou dans les réseaux télécom par exemple, on rencontre souvent des problèmes du type : on se donne un graphe orienté, avec des contraintes de capacité, et des coûts $c_a : \mathbb{R} \to \mathbb{R}$ convexes, pour tout $a \in A$, et on demande de trouver le flot de coût minimum. On dispose de nombreux algorithmes efficaces (et très souvent polynomiaux) pour résoudre ces problèmes.

Il faut aussi signaler le cas particulier de la **programmation semi-définie positive**, résolue en temps polynomial par les points intérieurs. Elle concerne le cas où les vecteurs des problèmes de programmation linéaires, tant ceux qui définissent l'objectif ou les contraintes que les variables, sont remplacés par des *matrices semi-définie positive*, i.e. des matrices symétriques réelles dont toutes les valeurs propres sont positives ou nulles.

La programmation semi-définie positive ne modélise pas aussi naturellement que la programmation linéaire des problèmes réels, mais apparaît souvent comme relaxation efficace de certains problèmes combinatoires.

14.1.4. Simulation. — Simuler c'est reproduire à volonté un phénomène *original* à l'aide d'un *modèle* qui en abstrait les éléments essentiels, dans le but de tirer des conclusions sur ce phénomène.

L'intérêt est évident quand

- le phénomène original est difficile à reproduire à volonté (conditions particulières difficiles à remplir, coût, durée, etc.)
- la modélisation sous forme problème ne se résout pas facilement (par exemple, les problèmes biniveaux compliqués ; ou en physique : équations Navier-Stokes)

On distingue les simulations à *temps continu* et ceux à *événements discrets*.

La simulation peut être utile par exemple pour dimensionner une flotte de véhicule de transport à la demande, compte tenu de la fonction d'utilité des agents, de la congestion, etc. ; pour proposer un nouveau système logistique, avec des tarifications variables ; pour étudier une chaîne d'approvisionnement complexe ; etc.

La simulation est souvent couplée avec des techniques plus classiques de recherche opérationnelle, comme on le dévine aisément à la lecture des exemples ci-dessus.

14.2. Trois domaines à la frontière de la recherche opérationnelle

14.2.1. Statistiques. — Les statistiques, que l'on peut définir comme la science de la représentation simplifiée des grands ensembles de données, interviennent en amont de la partie **Données** d'un Problème : les données ne sont pas toujours faciles à obtenir, ou peuvent être entachées d'erreurs.

14.2.2. Files d'attente. — C'est un outil de modélisation (puissant) des systèmes logistiques et de communication. Il y a des *clients*, qui cherchent à accéder à des *ressources* limitées, afin d'obtenir un *service*. Ces demandes concurrentes engendrent des délais et donc des files d'attente de clients.

Les mesures de performance sont en général : **nombre moyen de clients en attente** et **temps moyen d'attente**.

Le modèle le plus simple est donnée Figure 1.

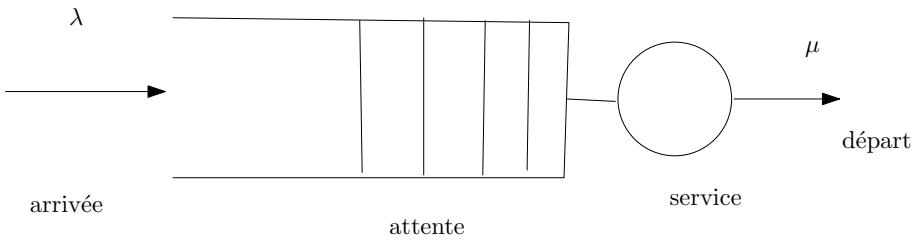


FIG. 1. Une file d'attente simple.

Les taux d'arrivée et de service, λ et μ sont donnés. Si le processus d'arrivée est poissonien et si la durée de service suit une loi exponentielle, le nombre de personnes dans la file est une chaîne de Markov en temps continu. Notons que pour avoir un régime stationnaire, il faut que $\lambda/\mu < 1$. En toute généralité, il n'y a pas de raison que la file d'attente puisse être modélisée par une chaîne de Markov, et leur étude peut devenir extrêmement difficile.

Les applications sont nombreuses : dimensionnement des caisses de supermarché, des services hospitaliers, des standard téléphoniques, serveurs informatiques, etc.

14.2.3. Théorie des jeux. — La théorie des jeux en même temps que la recherche opérationnelle, pendant la seconde guerre mondiale.

Dans ce cours, nous avons vu la théorie des jeux 2 fois : dans le Chapitre 4, la dualité forte nous a permis de démontrer le théorème de von Neumann (Théorème 4.5.1 ; dans le Chapitre 6, nous avons vu les mariages stables (théorème de Gale-Shapley), qui appartiennent au champ des jeux coopératifs où l'on s'intéresse à la stabilité des coalitions.

Il y a parfois des intéractions fortes entre la théorie des jeux et le recherche opérationnelle :

1. lorsque la fonction objectif ne veut pas optimiser mais "partager équitablement" : les problèmes académiques sont par exemple les mariages stables, *cake cutting*, *necklace splitting*. Dans ces deux derniers problèmes, on veut partager un objet (un gâteau ou un collier) et l'on veut que toutes les personnes perçoivent le partage comme équitable, même si elles n'ont pas la même fonction d'évaluation.
2. lorsque la fonction objectif contient un terme qui résulte d'un terme "réponse" économique. Par exemple, lorsqu'on veut concevoir un réseau routier, on est obligé de se demander par où va s'écouler le flot des usagers. Mais le flot est le résultat de ce que chaque personne décide de faire, d'un processus d'optimisation locale. Il faut donc évaluer et comparer les équilibres de Nash selon les choix qui sont faits. On parle alors de **problème biniveau**.

Dans ce second cas, il faut souligner que les choses sont difficiles, et peu intuitives comme en témoigne le célèbre paradoxe de Braess.

Considérons le réseau de la Figure 2. Un flot de valeur 1 veut aller de s à t. Chaque élément infinitésimal du flot choisit le chemin le moins coûteux. L'ajout d'une autoroute au milieu détériore tout.

Enfin, on peut mettre également à l'intersection de la recherche opérationnelle et de la théorie des jeux, la problématique de *Yield Management*, utilisé dans les transports et l'hôtellerie, et des *enchères*. La question traitée par le Yield Management concerne la trajectoire suivie par le tarif d'une prestation en fonction de la date où elle est achetée : on peut maximiser le bénéfice en suivant une trajectoire tenant compte des différentes fonctions d'utilité des clients potentiels. Dans les problèmes d'enchères, utilisées en particulier par les prestataires logistiques, on se

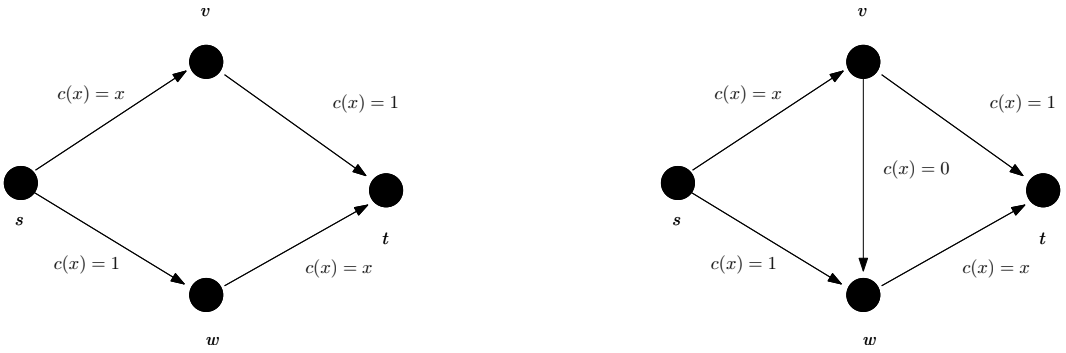


FIG. 2. Un réseau simple où apparaît le paradoxe de Braess

demande quelles règles mettre en place pour maximiser le prix auquel est vendu in fine la prestation.

ELÉMENTS DE CORRECTION

Exercice 2.6.3. — On suppose donc donné le réseau $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ de routes, et où $\mathcal{V}$ modélise les jonctions de portions de routes. Les points à visiter par le voyageur de commerce (dont son point de départ identifié avec son point d'arrivée) peut être vu comme un sous-ensemble $V \subseteq \mathcal{V}$.

On considère alors $G = (V, E)$ le graphe complet sur l'ensemble de sommets V. On munit chaque arête $e = uv$ d'un poids $w(e)$ égal à la longueur du plus court chemin dans $\mathcal{G}$ reliant u à v. Le problème du voyageur de commerce est alors exactement le problème du cycle hamiltonien de plus petit poids dans G.

En effet, prenons une tournée optimale dans $\mathcal{G}$. Elle induit un ordre de visite des éléments de V puis un retour au point de départ. Cela induit donc un cycle hamiltonien. Puisque la tournée est optimale, le voyageur de commerce choisit toujours un plus court chemin entre deux points visites successives de'éléments de V. Le coût de la tournée optimale est donc égale au coût du cycle hamiltonien qu'elle induit sur G.

Il reste à vérifier que le cycle hamiltonien optimal de G induit une tournée de même coût sur $\mathcal{G}$, mais c'est évident : le voyageur de commerce visite sur $\mathcal{G}$ les sommets de V dans l'ordre du cycle hamiltonien, et prends à chaque fois le plus court chemin.

Trouver un cycle hamiltonine de plus petit poids dans un graphe complet est un problème NP-difficile. En effet, supposons que l'on dispose d'un algorithme polynomial A résolvant ce problème. Prenons alors un graphe G quelconque à n sommets, et ajoutons toute arête non présente avec un poids égal à 2. Sur les arêtes présentes, on met un poids égal à 1. On obtient donc un graphe complet et appliquons lui l'algorithme A. Si le poids minimal d'un cycle hamiltonien est n, il y a un cycle hamiltonien dans G. Si ce poids est $> n$, il n'y a pas de cycle hamiltonien. A serait donc un algorithme d'un algorithme polynomial décidant si un graphe possède ou non un cycle hamiltonien. Or ce dernier problème est un problème NP-complet.

Exercice 2.6.7. — Considérons le graphe $G = (V, E)$ dont les sommets V sont les formations, et dans lequel une arête relie deux formations s'il existe un employé devant suivre ces deux formations. Enfin, les couleurs sont les jours. Il est clair que tout planning compatible avec les contraintes de la DRH induit une coloration propre du graphe, et dont le nombre de couleurs est le nombre de jours de la session. Réciproquement, toute coloration induit un planning compatible avec les contraintes de la DRH, avec un nombre de jours égal au nombre de couleurs.

Exercice 3.4.5. — Ce problème se résout par la programmation dynamique. Les états sont les âges possibles de la machine en début d'année. Les périodes sont les années. On s'intéresse

aux quantités

$\pi(t,k)$ = profit maximal possible en terminant la $t$ème année en possédant une machine d'âge k.

On souhaite trouver $\max_{k\in\{1,2,3\}}(\pi(5,k)+p_k)$.

$\pi(t,k)$ satisfait les relations suivantes :

$$\pi(1,k) = \begin{cases} 0 & \text{pour tout } k \in \{2,3\} \\ b_0 - c_0 & \text{pour } k = 0. \end{cases}$$

$$\pi(t+1,k+1) = \pi(t,k) + b_k - c_k \quad \text{si } k \in \{1,2\}$$

$$\pi(t+1,1) = \max_{k\in\{1,2,3\}} (\pi(t,k) + p_{k-1} - 100000 + b_0 - c_0) \text{ si } t \in \{1,2,3,4\}$$

On peut alors remplir le tableau des valeurs de π

t / k	1	2	3	4	5
1	70000	90000	110000	90000	150000
2	0	116000	136000	156000	136000
3	0	0	124000	144000	164000

Le profit maximal de l'entreprise est donc $\max(150000+50000, 136000+24000, 164000+10000) = 200000$. Une stratégie optimale consiste à garder la machine deux ans, puis la vendre et en racheter une nouvelle, encore une fois pour deux ans.

Exercice 3.4.11. — 1. On considère le graphe orienté D dont l'ensemble des sommets V est la collection d'intervalles $\mathcal{C}$ (on identifie alors V et $\mathcal{C}$) et dans lequel on a un arc (I,J) si l'extrémité supérieure de I est strictement inférieure à l'extrémité inférieure de J. C'est un graphe acircuitique, et les sommets des chemins élémentaires sont précisement les sous-ensembles d'intervalles deux à deux disjoints. En mettant comme poids sur tout arc (I,J) la quantité $w(I)$, et en ajoutant un dernier sommet X et tous les arcs (I,X) avec $I \in \mathcal{C}$ que l'on pondère avec $w(I)$, chercher le sous-ensemble d'intervalles de $\mathcal{C}$ deux à deux disjoints de poids maximal revient à chercher le plus long chemin de D pour la pondération w, ce qui se fait par un algorithme de programmation dynamique (le graphe étant acircuitique). Pour se ramener totalement au cas du cours, on peut également ajouter un sommet Y et tous les arcs (Y,I) avec $I \in \mathcal{C}$ que l'on pondère avec 0, et chercher le chemin de Y à X de plus grand poids.

2. Une demande de location se modélise par un intervalle de la droite réelle dont les extrémités sont les debut et fin de la location, pondéré par la gain qui serait obtenue en satisfaisant cette demande. Maximiser le gain revient alors à chercher le sous-ensemble d'intervalles disjoints (les locations ne pouvant se recouvrir) de plus grand poids.

Exercice 5.3.10. — 1. Considérons le graphe $D = (V,A)$ dont les sommets sont les blocs et pour lequel on a un arc (u,v) si $v \in Y_u$. Une solution du cas statique est un fermé de ce graphe et réciproquement tout fermé du graphe est une solution du cas statique.

2. Si on met $u_a := +\infty$ pour tout $a \in A$, on a la propriété recherchée. En effet, soit $X \subseteq V$ tel que $\delta^+_{\tilde{D}}(X \cup \{s\})$ soit une s-t coupe de $\tilde{D}$ de capacité minimale. Aucune arête de A ne peut être dans cette coupe, sinon sa capacité serait $= +\infty$. Donc, si $u \in X$, tout successeur de u est encore dans X. Ce qui est précisément la définition de "fermé".

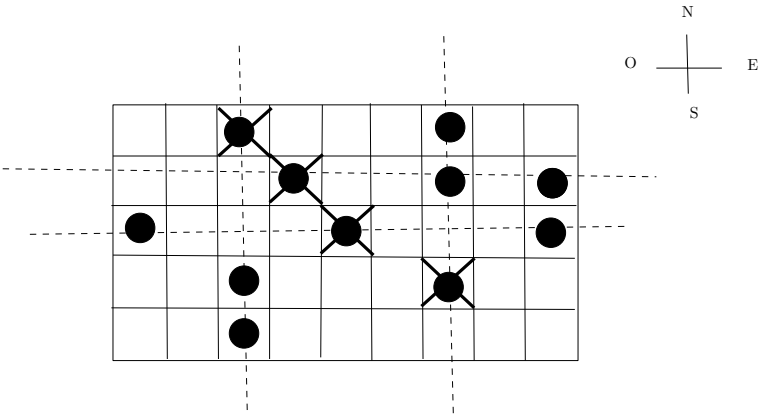


FIG. 3. Une solution optimale et une preuve d'optimalité.

3. On note $\bar{X} := V \setminus X$. Comme X est fermé, les seuls arcs qui vont jouer un rôle dans la coupe sont les arcs de la forme (s, v) avec $v \in \bar{X}$ et ceux de la forme (v, t) avec $v \in X$. On a donc

$$\sum_{a \in \delta^+_{\tilde{D}}(X \cup \{s\})} u_a = \sum_{v \in \bar{X} \cap V^+} c_v - \sum_{v \in X \cap V^-} c_v = \sum_{v \in V^+} c_v - \sum_{v \in X} c_v.$$

4. Posons $C := \sum_{v \in V^+} c_v$.

Soit X un fermé de D de valeur maximale η. D'après ce qui a été montré ci-dessus, on a alors une s-t coupe de $\tilde{D}$ de capacité $C - \eta$.

Réciproquement, prenons une s-t coupe de $\tilde{D}$ de capacité minimale κ. D'après ce qui a été montré, l'ensemble X induit sur D est alors fermé et de valeur $C - \kappa$.

On a donc : $C - \kappa = \eta$, et chercher une coupe minimale sur $\tilde{D}$ est équivalent à chercher un fermé maximum sur D. Comme chercher une coupe minimale se fait en temps polynomial, on sait résoudre le problème de manière efficace.

Exercice 6.6.3. — Dans le cas de la Figure, ce maximum est 4. Pour prouver son optimalité, il suffit de trouver 4 lignes ou colonnes qui contiennent tous les points. Notons ν la cardinalité maximale d'une famille P de points tels que deux points quelconques de P ne soient jamais dans la même ligne ou colonne, et notons τ la cardinalité minimale d'une famille R de lignes ou colonnes contenant tous les points. On a forcément $\nu \leq \tau$ puisque chaque élément de R contient au plus un élément de P.

Ces ensembles P et R de cardinalité 4 sont indiqués sur la Figure 3.

Le cas général est polynomial. En effet, ce problème se modélise sous la forme d'un graphe biparti de la façon suivante :

les sommets sont les lignes d'une part et les colonnes d'autre part, une arête relie la ligne l et la colonne c s'il y a un point noir à leur intersection. Un ensemble de points tels que deux points quelconques de l'ensemble soient toujours sur des lignes et des colonnes distinctes correspond à un couplage dans ce graphe. On recherche donc un couplage de cardinalité maximale dans ce graphe biparti, ce qui peut se faire en $O(\sqrt{n}m)$ où n est le nombre de sommets (ici le demi-périmètre du rectangle), et m le nombre d'arêtes (ici le nombre de points noirs).

Exercice 7.4.1. —

Gestion dynamique de stock, cas à 1 seul bien. — 1. Par définition du processus : ce qui reste dans le stock sur la période t, c'est ce qui était déjà dans le stock à la période $t-1$, plus ce qui a été produit, moins ce qui est parti à cause de la demande.

2. Aux contraintes décrivant la dynamique du stock, il faut ajouter les contraintes de capacité $x_t \leq P_t$ et les contraintes de positivité. La fonction objectif est simplement le coût total $\sum_{t=1}^{T} p_t x_t + \sum_{t=1}^{T-1} s_t y_t$. On s'arrête à $T-1$ pour le stock car comme on souhaite minimiser et que les coût de stockage sont positifs, on aura toujours $y_T = 0$.

Le programme linéaire s'écrit donc

$$\begin{array}{lll} \min & \sum_{t=1}^{T} p_t x_t + \sum_{t=1}^{T-1} s_t y_t & \\ \text{s.c.} & x_t - y_t + y_{t-1} = d_t & \text{pour tout } t = 1, \dots, T \\ & x_t \leq P_t & \text{pour tout } t = 1, \dots, T \\ & x_t \geq 0 & \text{pour tout } t = 1, \dots, T \\ & y_t \geq 0 & \text{pour tout } t = 1, \dots, T-1. \end{array}$$

3. Le coût minimal est 37. Le vecteur de production est $\boldsymbol{x} = (8, 1, 3, 1)$ ou $\boldsymbol{x} = (9, 0, 4, 0)$, ou n'importe quelle combinaison convexe des deux.

4. En dehors du puits et des sources proposées par l'énoncé, il n'y a pas d'autres sommets. Il y a deux types d'arcs :

- les arcs (v, w_t), avec une capacité supérieure $= P_t$, une capacité inférieure $= 0$ et un coût $= p_t$ et
- les arcs (w_{t-1}, w_t) avec une supérieure $= +\infty$, une capacité inférieure $= 0$ et un coût $= s_{t-1}$.

Le problème du b-flot de coût minimum est exactement le problème linéaire de la question 2. En particulier, la dynamique correspond à la loi de Kirchoff.

5. Les problèmes de flot de coût min sont des cas particuliers de la programmation linéaire. Ils possèdent des algorithmes dédiés qui peuvent être plus rapides que les solveurs généraux de programme linéaire (méthode des cycles de coût moyen minimum, vu en cours par exemple).

Gestion dynamique de stock, cas à plusieurs biens. — 6. On a $y_{kt} = y_{k(t-1)} + x_{kt} - d_{kt}$ pour tout k, t.

7. C'est $\sum_{k=1}^{K} z_{kt} \leq 1$ pour tout t.

8. C'est $x_{kt} \leq P_{kt} z_{kt}$ pour tout k, t.

9. En reprenant les contraintes déjà identifiées en première partie, et cette nouvelle contrainte, le problème que l'on cherche à résoudre s'écrit

$$\begin{array}{lll} \min & \sum_{k=1}^{K} \sum_{t=1}^{T} p_{kt} x_{kt} + \sum_{k=1}^{K} \sum_{t=1}^{T-1} s_{kt} y_{kt} & \\ \text{s.c.} & x_{kt} - y_{kt} + y_{k(t-1)} = d_{kt} & \text{pour tout } t = 1, \dots, T, \text{ pour tout } k = 1, \dots, K \\ & x_{kt} \leq P_{kt} z_{kt} & \text{pour tout } t = 1, \dots, T, \text{ pour tout } k = 1, \dots, K \\ & \sum_{k=1}^{K} z_{kt} \leq 1 & \text{pour tout } t = 1, \dots, T \\ & z_{kt} \in \{0, 1\} & \text{pour tout } t = 1, \dots, T, \text{ pour tout } k = 1, \dots, K \\ & x_{kt} \geq 0 & \text{pour tout } t = 1, \dots, T, \text{ pour tout } k = 1, \dots, K \\ & y_{kt} \geq 0 & \text{pour tout } t = 1, \dots, T-1, \text{ pour tout } k = 1, \dots, K. \end{array}$$

10. Relâchons les contraintes $\sum_{k=1}^{K} z_{kt} \leq 1$ et $x_{kt} \leq P_{kt} z_{kt}$. On pose $\boldsymbol{\lambda} \in \mathbb{R}_+^T$ le multiplicateur de Lagrange associé aux premières contraintes et $\boldsymbol{\mu} \in \mathbb{R}_+^{TK}$ celui associé aux secondes contraintes.

Avec les notations usuelles on a

$$\begin{array}{rcll}\mathcal{G}(\boldsymbol{\lambda},\boldsymbol{\mu}) & = & -\sum_{t=1}^{T}\lambda_t + & \min\left(\sum_{k=1}^{K}\sum_{t=1}^{T}p_{kt}x_{kt}+\sum_{k=1}^{K}\sum_{t=1}^{T-1}s_{kt}y_{kt}+\sum_{k=1}^{K}\sum_{t=1}^{T}(\lambda_t-\mu_{kt}P_{kt})z_{kt}\right)\\ & & \text{s.c.} & x_{kt}-y_{kt}+y_{k(t-1)}=d_{kt} \quad \text{pour tout } t=1,\dots,T, \text{ pour tout } k=1,\dots,K\\ & & & z_{kt}\in\{0,1\} \quad \text{pour tout } t=1,\dots,T, \text{ pour tout } k=1,\dots,K\\ & & & x_{kt}\geq 0 \quad \text{pour tout } t=1,\dots,T, \text{ pour tout } k=1,\dots,K\\ & & & y_{kt}\geq 0 \quad \text{pour tout } t=1,\dots,T-1, \text{ pour tout } k=1,\dots,K,\end{array}$$

qui peut se réécrire, puisque les biens ne sont plus couplés

$$\begin{array}{rcll}\mathcal{G}(\boldsymbol{\lambda},\boldsymbol{\mu}) & = & -\sum_{t=1}^{T}\lambda_t + \sum_{k=1}^{K} & \min\left(\sum_{t=1}^{T}p_{kt}x_{kt}+\sum_{t=1}^{T-1}s_{kt}y_{kt}+\sum_{t=1}^{T}(\lambda_t-\mu_{kt}P_{kt})z_{kt}\right)\\ & & \text{s.c.} & x_{kt}-y_{kt}+y_{k(t-1)}=d_{kt} \quad \text{pour tout } t=1,\dots,T, \text{ pour tout } k=1,\dots,K\\ & & & z_{kt}\in\{0,1\} \quad \text{pour tout } t=1,\dots,T, \text{ pour tout } k=1,\dots,K\\ & & & x_{kt}\geq 0 \quad \text{pour tout } t=1,\dots,T, \text{ pour tout } k=1,\dots,K\\ & & & y_{kt}\geq 0 \quad \text{pour tout } t=1,\dots,T-1, \text{ pour tout } k=1,\dots\end{array}$$

On veut donc résoudre, en oubliant l'indice k,

$$\begin{array}{ll}\min & \left(\sum_{t=1}^{T}p_tx_t+\sum_{t=1}^{T-1}s_ty_t+\sum_{t=1}^{T}(\lambda_t-\mu_tP_t)z_t\right)\\ \text{s.c.} & x_t-y_t+y_{t-1}=d_t \quad \text{pour tout } t=1,\dots,T\\ & z_t\in\{0,1\} \quad \text{pour tout } t=1,\dots,T\\ & x_t\geq 0 \quad \text{pour tout } t=1,\dots,T\\ & y_t\geq 0 \quad \text{pour tout } t=1,\dots,T-1.\end{array}$$

On remarque qu'à l'optimum on peut choisir $z_t=0$ si $\lambda_t-\mu_tP_t\geq 0$ et $z_t=1$ si $\lambda_t-\mu_tP_t<0$.

On veut résoudre finalement, puisque la valeur de z_t peut être fixée indépendamment de $\boldsymbol{x}$ et $\boldsymbol{y}$:

$$\begin{array}{ll}\min & \left(\sum_{t=1}^{T}p_tx_t+\sum_{t=1}^{T-1}s_ty_t\right)\\ \text{s.c.} & x_t-y_t+y_{t-1}=d_t \quad \text{pour tout } t=1,\dots,T\\ & x_t\geq 0 \quad \text{pour tout } t=1,\dots,T\\ & y_t\geq 0 \quad \text{pour tout } t=1,\dots,T-1.\end{array}$$

qui est précisément le problème de la première partie, sans les capacités de production. C'est bien un problème de b-flot.

On sait donc facilement trouver les solutions optimales $\boldsymbol{x}^*$, $\boldsymbol{y}^*$ et $\boldsymbol{z}^*$ qui résolvent $\mathcal{G}(\boldsymbol{\lambda},\boldsymbol{\mu})$. On sait calculer les sur-gradients de $\mathcal{G}$ (d'après le cours). On sait donc maximiser $\mathcal{G}$ et donc trouver de bonnes bornes inférieures à notre problème de départ. Tout est en place pour un branch-and-bound.

Exercice 7.4.2. — 1.

$$\begin{array}{lrcll}\min & \sum_{i=1}^{n}\sum_{\tau=1}^{T}c_{i,\tau}\left(x_{i,\tau}-x_{i,\tau-1}\right) & & & \\ \text{s.c.} & x_{i,\tau-1} & \leq & x_{i,\tau} & \text{pour tout } i\in\{1,\dots,n\} \text{ et tout } \tau\in\{1,\dots,T\}\\ & x_{i,\tau} & \leq & x_{j,\tau} & \text{pour tout } i\in\{1,\dots,n\}, \text{ pour tout } j\in Y_i \text{ et tout } \tau\in\{1,\dots\}\\ & \sum_{i=1}^{n}m_i\left(x_{i,\tau}-x_{i,\tau-1}\right) & \leq & M_\tau & \text{pour tout } \tau\in\{1,\dots,T\}\\ & x_{i,\tau} & \in & \{0,1\} & \text{pour tout } i\in\{1,\dots,n\} \text{ et tout } \tau\in\{0,1,\dots,T\}\\ & x_{i,0} & = & 0 & \text{pour tout } i\in\{1,\dots,n\}\end{array}$$

$x_{i,\tau}-x_{i,\tau-1}$ vaut 1 si le bloc i est extrait au cours de l'année τ et 0 sinon. La fonction objectif est donc bien celle attendue. Les contraintes (i) imposent que pour i fixé, la séquence des $x_{i,\tau}$ est de la forme $0,0,\dots,0,1,1,\dots,1$, comme attendu. Les contraintes (ii) imposent que si un bloc i est extrait au cours de l'année τ ou avant, alors tout bloc de Y_i est extrait au cours de l'année τ

ou avant, conformément à la définition de Y_i. Les contraintes (iii) imposent que la masse totale extraite au cours de l'année τ soit inférieure à M_τ, comme attendu.

2. $\sum_{\tau'=1}^{\tau^*} M_{\tau'}$ est la masse maximale qui a pu être extraite sur les années de 1 à τ^*. Si la masse de Y_{i^*} et du bloc i^* est strictement supérieure, forcément le bloc i^* ne peut être extrait au cours de l'année τ^* ou avant.

3. $\sum_{\tau'=1}^{\tau^*} M_{\tau'}$ est la masse maximale qui a pu être extraite au cours des années de 1 à τ^*. Si la masse de $Y_{i^*} \cup Y_{j^*}$ et des blocs i^* et j^* est strictement supérieure, forcément les blocs i^* et j^* ne peuvent être extraits tous deux au cours de l'année τ^* ou avant.

4. La suppression des variables indicées par des bonnes paires réduit le nombre de variables du programme linéaire. Cela réduit forcément sa complexité. L'ajout des contraintes induites par des bons triplets ne changent pas les solutions entières du programme linéaire, mais diminue la valeur de la relaxation continue, ce qui améliore la qualité de la borne utilisée dans le branch-and-bound – on est dans un problème de maximisation – et donc réduit potentiellement le nombre de branchements.

Exercice 8.2.6. — Question 2. Supposons que l'on ait une solution optimale S et notons p et g respectivement le plus petit et le plus gros objets qui soient placés. S'ils ne sont pas mis ensemble dans un conteneur, notons p' celui qui est avec g et g' celui qui est avec p. On a $a_p \leq a_{p'}$ et $a_{g'} \leq a_g$. Comme $a_{p'} + a_g \leq 1$, on a $a_{p'} + a_{g'} \leq 1$ et $a_p + a_g \leq 1$. Il existe donc toujours une solution optimale dans laquelle le plus petit et le plus gros objets placés sont ensemble dans un conteneur. Par induction, on peut montrer qu'en triant les a_i et en cherchant à mettre le plus petit objet courant avec le plus grand possible, on obtient une solution optimale en $O(n \log(n))$.

Exercice 9.5.3. — Nous proposons dans la suite une modélisation, toute en sachant que plusieurs autres sont possibles.

1. Notons x_{ij} (resp. y_{ij}) la part de la demande du client j desservie par l'entrepôt forward (resp. reverse) i. Notons u_i (resp. v_i) la variable binaire codant l'ouverture d'un entrepôt forward (resp. reverse) i.
Un programme linéaire possible est alors

$$\begin{array}{lll}
\min & \sum_{i\in I}(f_i u_i + r_i v_i + \sum_{j\in J} c_{ij} h_j (x_{ij} + \alpha_j y_{ij})) & \\
\text{s.c.} & \sum_{i\in I} x_{ij} = h_j & j \in J \\
& \sum_{i\in I} y_{ij} = \alpha_j h_j & j \in J \\
& \sum_{j\in J} x_{ij} \leq b_i u_i & i \in I \\
& \sum_{j\in J} y_{ij} \leq e_i v_i & i \in I \\
& u_i, v_i \in \{0,1\} & i \in I, j \in J, \\
& x_{ij}, y_{ij} \in \mathbb{R}_+ & i \in I, j \in J.
\end{array}$$

Ce programme linéaire est décomposable en deux sous-problèmes indépendants : l'un avec les x_{ij} et u_i et l'autre avec les y_{ij} et v_i.

2. En ajoutant la variable z_i qui indique la possibilité de fusionner les deux entrepôts lorsqu'il sont tous deux ouverts en i, on peut modéliser un gain à la mutualisation de la manière suivante :

$$
\begin{array}{lll}
\min & \sum_{i\in I}(f_i u_i + r_i v_i - s_i z_i + \sum_{j\in J} c_{ij} h_j (x_{ij} + \alpha_j y_{ij})) & \\
\text{s.c.} & \sum_{i\in I} x_{ij} = h_j & j \in J \\
& \sum_{i\in I} y_{ij} = \alpha_j h_j & j \in J \\
& z_i \leq u_i & i \in I \\
& z_i \leq v_i & i \in I \\
& \sum_{j\in J} x_{ij} \leq b_i u_i & i \in I \\
& \sum_{j\in J} y_{ij} \leq e_i v_i & i \in I \\
& u_i, v_i \in \{0,1\} & i \in I, j \in J, \\
& x_{ij}, y_{ij} \in \mathbb{R}_+ & i \in I, j \in J.
\end{array}
$$

En effet, si u_i et v_i sont tous deux égaux à 1, puisqu'on cherche à minimiser la fonction objectif, z_i sera automatiquement mis à 1. Sinon, z_i vaut 0, comme souhaité.

3. Même remarque que dans le cours : une fois les u_i et v_i fixés, toutes les autres variables (z_i, x_{ij}, y_{ij}) sont automatiquement fixées, puisqu'on cherche à minimiser la fonction objectif. Les x_{ij} et y_{ij} sont alors le résultat d'un problème de transport sur le graphe biparti complet avec I et J les deux classes de couleur.

Par conséquent, une façon compacte de coder les solutions consistent à ne considérer que les couples $(\boldsymbol{u}, \boldsymbol{v}) \in \{0,1\}^{2|I|}$. La valeur d'une solution $(\boldsymbol{u}, \boldsymbol{v})$ s'obtient en mettant $z_i = 1$ dès que $u_i = v_i = 1$ et à résoudre le problème de transport.

On dit que deux solutions $(\boldsymbol{u}, \boldsymbol{v})$ et $(\boldsymbol{u}', \boldsymbol{v}')$ sont voisines si

$$
\begin{cases}
\boldsymbol{u} = \boldsymbol{u}' \\
\text{on passe de } \boldsymbol{v} \text{ à } \boldsymbol{v}' \text{ par une opération } \textit{drop}, \textit{add} \text{ ou } \textit{swap} - \text{définies en cours,}
\end{cases}
$$

ou

$$
\begin{cases}
\text{on passe de } \boldsymbol{u} \text{ à } \boldsymbol{u}' \text{ par une opération } \textit{drop}, \textit{add} \text{ ou } \textit{swap} - \text{définies en cours,} \\
\boldsymbol{v} = \boldsymbol{v}'
\end{cases}
$$

Avec une telle définion de voisinage, l'espace des solutions est clairement connexe, ce qui assure qu'une recherche locale potentiellement pourra atteindre la meilleure solution.

Remarque : on peut également faire les opérations simultanément sur $\boldsymbol{u}$ et $\boldsymbol{v}$, mais alors il faut démontrer, ce qui n'est pas immédiat, que l'espace des solutions est bien connexe...

4. Cela ne change rien, car les $h_j, \alpha_j h_j, b_i, e_i$ étant entier, le problème de transport a forcément une solution entière.

Exercice 9.5.4. —

1. Prenons un graphe $G = (V, E)$ et construisons l'instance suivante du problème des ambulances : $S = C = V$, K ambulances, et $t_{u,v}$ = nombre minimum d'arêtes de u à v. Il y a un dominant à $\leq K$ sommets dans G si et seulement si le temps maximum pour le problème des ambulances associé est $= 1$. Si le problème des ambulances admettait un algorithme polynomial, on pourrait alors s'en servir pour tester l'existence en temps polynomial de dominant de taille $\leq K$ dans le graphe. Ce qui est impossible, ce dernier problème étant NP-complet.

2. Considérons une solution optimale au problème des ambulances. On construit alors de la manière suivante une solution au programme linéaire : $y_s = 1$ s'il y a au moins une ambulance située en s et $y_s = 0$ sinon ; $x_{s,c} = 1$ si l'ambulance la plus proche de c est située en s, et $x_{s,c} = 0$ sinon. Si plusieurs sites sont les plus proches, on en choisit arbitrairement un. Enfin, on pose $h =$ temps maximum pour qu'une commune soit atteinte par une ambulance. Cette solution est une solution réalisable du programme linéaire, et de même valeur du critère à optimiser.

Réciproquement, soit $\boldsymbol{x}, \boldsymbol{y}, h$ une solution optimale du programme linéaire. On positionne une (ou plusieurs) ambulance en s si $y_s = 1$. Comme on cherche à minimiser h, h est forcément égal à $\max_{c\in C} \min_{s\in S:\, x_{s,c}=1} t_{s,c}$ d'après l'inégalité (iv). Combiné avec (ii), cela indique que toute les communes peuvent être ralliées en un temps maximum h.

Le problème et le programme linéaire ont donc même valeur optimale et on peut passer directement d'une solution du problème à une solution du programme linéaire et vice-versa.

3. En notant λ_s le multiplicateur de Lagrange assiocié à la contrainte (ii), on obtient

$$\min \quad h + \sum_{s\in S}(-\lambda_s)|C|y_s + \sum_{s\in S}\sum_{c\in C}\lambda_s x_{s,c}$$

$$\begin{aligned}
\text{s.c.} \qquad \sum_{s\in S} y_s &\leq K && && \text{(i)}\\
\sum_{s\in S} x_{s,c} &= 1 && \text{pour tout } c\in C && \text{(iii)}\\
t_{s,c}x_{s,c} &\leq h && \text{pour tout } s\in S \text{ et } c\in C && \text{(iv)}\\
x_{s,c} &\in \{0,1\} && \text{pour tout } s\in S \text{ et } c\in C && \text{(v)}\\
y_s &\in \{0,1\} && \text{pour tout } s\in S && \text{(vi)}\\
h &\in \mathbb{R}_+ && && \text{(vii)}
\end{aligned}$$

4. Les variables $\boldsymbol{y}$ d'une part, et les variables $\boldsymbol{x}$ et h d'autre part sont indépendantes dans les contraintes du programme linéaire de la question précédente. Quelque soit le choix de $\boldsymbol{y}$, on peut fixer $\boldsymbol{x}$ et h indépendament. On optimise donc les deux programmes

$$\min \quad h + \sum_{s\in S}\sum_{c\in C}\lambda_s x_{s,c}$$

$$\begin{aligned}
\text{s.c.} \qquad \sum_{s\in S} x_{s,c} &= 1 && \text{pour tout } c\in C && \text{(iii)}\\
t_{s,c}x_{s,c} &\leq h && \text{pour tout } s\in S \text{ et } c\in C && \text{(iv)}\\
x_{s,c} &\in \{0,1\} && \text{pour tout } s\in S \text{ et } c\in C && \text{(v)}\\
h &\in \mathbb{R}_+ && && \text{(vii)}
\end{aligned}$$

et

$$\min \quad \sum_{s\in S}(-\lambda_s)y_s$$

$$\begin{aligned}
\text{s.c.} \qquad \sum_{s\in S} y_s &\leq K && && \text{(i)}\\
y_s &\in \{0,1\} && \text{pour tout } s\in S && \text{(vi)}
\end{aligned}$$

indépendament.

5. Optimiser le deuxième programme est simple : il suffit de trier les s par valeurs de λ_s décroissantes ; ensuite on met $y_s = 1$ pour les K premiers s, et $y_s = 0$ pour les suivants. Complexité $O(S \log S)$.

Optimiser le premier programme est presque aussi simple. Noter que les λ_s sont positifs. A l'optimum, (iv) est une égalité. On essaie successivement dans h toutes les valeurs de $t_{s,c}$, et pour chacune de ces valeurs, on fait la chose suivante : pour chaque c, on cherche parmi les $t_{s,c} \leq h$ celui tel que λ_s est le plus petit, et c'est ce couple (s, c) pour lequel $x_{s,c} = 1$. Pour chacune des SC valeurs de h essayées, on obtient une valeur de la fonction objectif ; on garde alors la plus petite. Complexité $O(S^2C^2)$.

6. Avec les notations de cours, pour tout $\boldsymbol{\lambda} \in \mathbb{R}^S$, on sait calculer $\mathcal{G}(\boldsymbol{\lambda})$ et ses surgradients en $O(S^2C^2)$ (donc rapidement), et donc optimiser par un algorithme de surgradient $\mathcal{G}(\boldsymbol{\lambda})$, fournissant ainsi des bornes au programme linéaire en nombres entiers.

Exercice 10.4.3. — Les deux premières questions sont triviales. Pour la dernière, notons p_j la durée de la tâche j. Sans perte de généralité, supposons que les tâches sont ordonnées de manière à ce que

$$\frac{w_1}{p_1} \geq \frac{w_2}{p_2} \geq \ldots \geq \frac{w_n}{p_n}.$$

Prenons maintenant un ordonnancement quelconque et supposons qu'il y ait une tâche j et suivie d'une tâche j' telles que $\frac{w_j}{p_j} \leq \frac{w_{j'}}{p_{j'}}$. En inversant l'ordre de ces deux tâches, on ajoute à la fonction objectif la quantité $w_j p_{j'} - w_{j'} p_j$ qui est négative ou nulle. Un ordonnancement optimal est donc obtenu en ordonnant les tâches dans l'ordre de leur indexation.

Exercice 10.4.8. — Si on voit les entrepôts D et C comme des machines, les camions comme des tâches, et le déchargement et le chargement comme des opérations, on est exactement dans le cas d'un problème de flow-shop à 2 machines, 15 tâches et minimisation du makespan, ce qui se résout en temps polynomial (et même à la main) par l'algorithme de Johnson. Les durées des opérations sont données par le temps qu'il faut pour décharger ou charger les boîtes.

Ici la solution est de 91 minutes. On terminera donc au plus tôt à 7h31.

Exercice 11.3.8. — Conformément à ce qui a été ajouté à l'oral, on suppose les coûts positifs.

1. Pour (X, v) tels que $v \in X \subseteq V \setminus \{s\}$, on note $\pi(X, v)$ le coût minimum d'une chaîne élémentaire dont les sommets sont $\{s\} \cup X$ et dont les extrémités sont s et v. On a alors

$$\pi(X, v) = \begin{cases} c(sv) & \text{si } |X| = 1 \\ \min_{u \in X \setminus \{v\}} (\pi(X \setminus \{v\}, u) + c(uv)) & \text{sinon.} \end{cases}$$

Pour trouver la meilleure chaîne hamiltonienne d'extrémité s, il suffit alors de comparer les valeurs de $\pi(V, v)$ pour tout $v \in V$ (les coûts étant positifs, on n'a pas intérêt à revenir en s).

2. C'est le nombre de couples (X, v) comme dans l'énoncé. Il y en a

$$\sum_{k=1}^{n-1} \binom{n-1}{k} k = (n-1)2^{n-2}.$$

3. Pour chaque état (X, v), le nombre d'additions est $|X| - 1$ (nombre de u possibles dans l'équation de programmation dynamique). D'où un nombre d'opérations

$$\sum_{k=1}^{n-1} \binom{n-1}{k} k(k-1) = (n-1)(n-2)2^{n-3}.$$

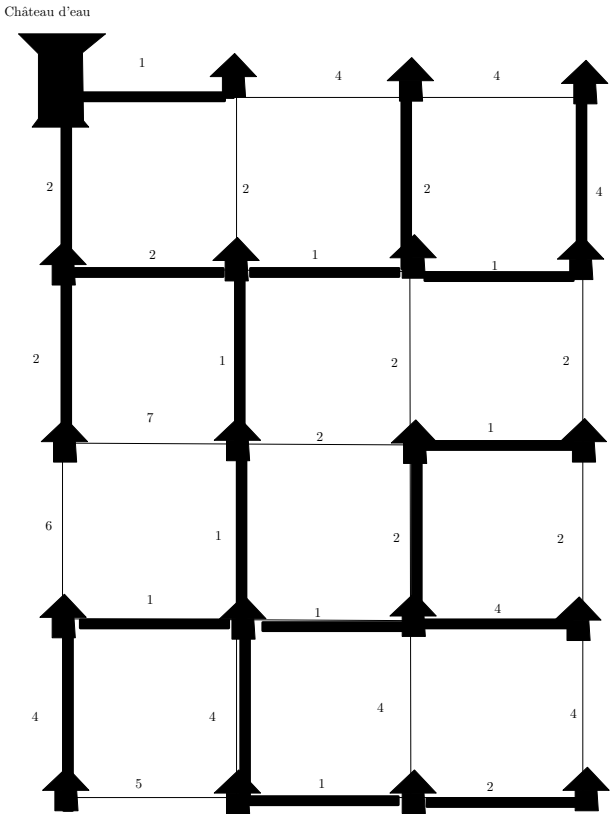


FIG. 4

Enumérer toutes les solutions donne un nombre d'additions $(n-1)!(n-1) \sim n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, ce qui est incomparablement plus grand.

4. Pour $n = 15$, on compte $745'472$ opérations, ce qui se fait en moins d'une seconde.

Pour $n = 30$, on compte 1.09×10^{11} opérations, ce qui se fait en 1.09×10^5 secondes, soit environ 30 heures. On ne peut résoudre cette instance en 1 jour, mais en une semaine oui.

Pour $n = 45$, on compte 8.32×10^{15} opérations, ce qui fait 8.32×10^9 secondes, soit environ 263 années. Même en un siècle on ne parvient à résoudre cette instance.

5. On sait calculer la plus courte chaîne hamiltonienne de s à v, pour tout $v \notin V$. Il suffit alors de comparer les valeurs de $\pi(V, v) + c(vs)$ pour tout v.

Exercice 13.5.2. — C'est une application directe de l'algorithme de Kruskal, qui détermine l'arbre couvrant de plus petit poids. Il y a plusieurs arbres possibles. L'un est donné Figure 4. Le coût optimal est 35.

BIBLIOGRAPHIE

[1] R.K. Ahuja, T.I. Magnanti, and J.B. Orlin, *Network flows*, Prentice Hall, 1993.

[2] V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Mungala, and V. Pandit, *Local search heuristics for k-median and facility location problems*, SIAM Journal on Computing **33** (2004), 544–562.

[3] P. Brucker, *An efficient algorithm for the job-shop problem with two jobs*, Computing **40** (1988), 353–359.

[4] V. Chvátal, *Linear programming*, W. H. Freeman, New York, 1983.

[5] R. Correa and C. Lemarechal, *Convergence of some algorithms for convex minimization*, Mathematical Programming **62** (1993), 261–275.

[6] D. de Werra, T. M. Liebling, and Hêche J.-F., *Recherche opérationnelle pour ingénieurs*, Presses polytechniques et universitaires romandes, 2003.

[7] S.E. Dreyfus and R.A. Wagner, *The steiner problem in graphs*, Networks (1972), 195–207.

[8] J. Edmonds and R.M. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, Journal of the ACM **19** (1972), 248–264.

[9] D. Gale and L.S. Shapley, *College admissions and the stability of marriage*, American Mathematical Monthly (1962), 9–15.

[10] M.R. Garey and D.S. Johson, *Computers and intractability : A guide to the theory of* **np***-completness.*

[11] N. Garg and J. Könemann, *Faster and simpler algorithms for multicommodity flow and other fractional packing problems*, Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science, 1998, pp. 300–309.

[12] A. V. Goldberg, *Scaling algorithms for the shortest paths problem*, SIAM Journal on Computing **24** (1995), 222–231.

[13] A.V. Goldberg and R.E. Tarjan, *Finding minimum-cost circulations by cancelling negative cycles*, Journal of the ACM **36** (1985), 873–886.

[14] M. Grötchel, L. Lovász, and L. Schrijver, *Geometric algorithms and combinatorial optimization*, 2nd edition, Springer, Heidelberg, 1994.

[15] B. Hajek, *Cooling schedules for optimal annealing*, Mathematics of Operations Research **13** (1991), 311–329.

[16] A. Hertz and D. De Werra, *Using tabu search techniques for graph coloring*, Computing **39** (1987), 345–351.

[17] O.H. Ibarra and C.E. Kim, *Fast approximation algorithms for the knapsack and sum of subset problem*, Journal of the ACM (1975), 463–468.

[18] J.R. Jackson, *Scheduling a production line to minimize maximum tardiness*, Tech. report, University of California, Los Angeles, 1955.

[19] S.M. Jonson, *Optimal two- and three-stage production schedules with setup times included*, Naval Res. Log. Quart **1** (1954), 61–68.

[20] R.M. Karp, *A characterization of the minimum cycle mean in a digraph*, Discrete Mathematics **23** (1978), 309–311.

[21] Kuhn, (1955).

[22] J. Matousek and B. Gärtner, *Understanding and using linear programming*, Sringer, Berlin Heidelberg New York, 2007.

[23] D.B. Shmoys, E. Tardos, and K. Aardal, *Approximation algorithms for facility location problems.*, Proceedings of the 29th Annual ACM Symposium on Theory of Computing, 1997, pp. 265–274.

[24] E. Tardos, *A strongly polynomial algorithm to solve combinatorial linear programs*, Operations Research **34** (1986), 250–256.

[25] T. Terlaky, *An easy way to teach interior-point methods*, European Journal of Operation Research **130** (2001), 1–19.

ANNEXE

QUELQUES OUTILS DE R.O.

Quelques sociétés

Roadef (www.roadef.org) : société francaise de recherche opérationnelle et d'aide à la décision.
Euro (www.euro-online.org) : associations européennes des sociétés de recherche opérationnelle.
Informs (www.informs.org) : Institute for Operations Research and the Management Sciences. C'est la plus grande société professionnelle dans le domaine de la recherche opérationnelle.

Ressources en ligne

- 24h Operations research.
- Une liste d'outils open-source logiciels pour la Recherche Opérationnelle : NEOS : Server for optimisation (http ://www-neos.mcs.anl.gov) Propose des serveurs de logiciel R.O utilisables en ligne via des formats de modélisation standard (AMPL, ...)
- Outils par ordre alphabétique COIN-OR (http ://www.coin-or.org/) COmputational INfrastructure for Operations Research Projet Open source, ambitieux, d'origine IBM (C++)
- COMET (http ://www.cs.brown.edu/people/pvh/comet/comet.html) COMET est une plateforme associant la recherche local et la programmation par contrainte. COMET s'appuit sur un langage objet et peut être étendu (création de contraintes...) en C++.
- GLPK (http ://www.gnu.org/software/glpk/) : La bibliothèque Glpk est livrée avec un solveur autonome (glpsol), capable de traiter des problèmes linéaires (continus ou en nombre entier) par des méthodes de simplexe ou de Points Intérieur. Ces problèmes peuvent être modélisés dans différents langages, dont l'excellent GMPL (clone de AMPL).
- Page RO de Google (http ://code.google.com/p/or-tools/) : Google a développé pour ses besoins propres un certain nombre d'outils de recherche opérationnelle. Sur ce site, ils sont proposés en open-source.
- GUSEK (http ://gusek.sourceforge.net/gusek.html) : le solver GLPK avec une interface ; pour Windows.
- Graphviz (http ://www.graphviz.org/) : Logiciel permettant d'afficher automatiquement un graphe à partir de formats de description textuels simples.
- LP_Solve (http ://groups.yahoo.com/group/lp_solve/)
- Operations Research - Java object (http ://OpsResearch.com/OR-Objects) Librairie JAVA pour (petits) problèmes de R.O. Utilisation gratuite, mais sources non accessibles. Ne semble plus maintenue, mais reste intéressante pour les API.
- QSopt-Exact Home Page (http ://www.dii.uchile.cl/ daespino/) Solveur linéaire avec calculs exacts (car rationnels), basé sur la librairie GMP (Gnu Multi Precision)
- TSP (http ://www.tsp.gatech.edu/index.html) : le site de référence sur le TSP. De la documentation, des solvers, interfaces, jeux, etc.

- SCILAB(http ://www.scilab.org/) : L'environnement de calcul scientifique open-source de référence.
- SCIP (http ://scip.zib.de/doc/html/index.html) : Peut-être un des meilleurs logiciels libres actuels. Il possède de plus une librairie permettant une implémentation simple des branch-and-cuts.

Quelques SSII spécialisées dans la RO

- Artelys (http ://www.artelys.com/) Société spécialisée en optimisation - formation. Par ailleurs, Artelys propose désormais Artelys Kalis un environnement (librairie C++ ou java) de Programmation par Contraintes exploitable avec le langage de modélisation Xpress-Mosel.
- Eurodécision (http ://www.eurodecision.com) réalise des applications d'aide à la décision par intégration de technologies d'optimisation et de simulation. Les principaux domaines d'application sont : la logistique, les ressources humaines, la production, le transport, les achats, les télécommunications.
- FuturMaster (http ://www.FuturMaster.com) FuturMaster accompagne ses clients dans leur démarche d'amélioration de leurs performances logistiques en leur offrant des solutions logicielles pertinentes, évolutives et réactives, pour suivre les changements internes et externes de l'entreprise.
- Rostudel (http ://www.rostudel.com/) C'est une toute jeune entreprise (fondée par un ancien élève de l'école des ponts) qui propose des prestations de conseil en recherche opérationnelle.

Quelques sociétés éditrices de logiciels de RO

- Amadeus (http ://www.amadeus.com) : propose des solutions d'optimisation dans l'industrie du voyage et du tourisme.
- Dash optimization (http ://www.dash.co.uk/) : propose l'outil Xpress-MP.
- Equitime (http ://www.equitime.fr/) : propose des logiciels d'optimisation d'emplois du temps.
- Gower Optimal Algorithms Ltd (http ://www.packyourcontainer.com/index.htm) : optimisation de la logistique.
- ILOG (http ://www.ilog.fr/products/optimization) Propose des outils de référence dans le domaine de l'optimisation comme ILOG/Cplex pour la programmation linéaire et quadratique et ILOG/CP pour la Programmation par contraintes.
- INRO (http ://www.inro.ca/fr/produits/) : L'INRO est spécialisée dans les logiciels de planification des transports : EMME/2 et Dynameq.
- KLS Optim (http ://www.klsoptim.com/) : spécialisée dans l'optimisation des plans de chargement (packing).
- Koalog (http ://www.koalog.com/php/jcs.php) : Cette société française propose entre autre un Solveur par contraintes écrit 100% en java, avec formation... Une application permettant le choix d'une configuration d' options d'automobile .
- LoadPlanner (http ://www.loadplanner.com/) : spécialisée dans l'optimisation des plans de chargement (packing).
- Opti-time (http ://www.opti-time.com) : spécialisée dans l'optimisation de tournées.
- Ortec (http ://fr.ortec.com/solutions/lb.aspx : spécialisée dans l'optimisation des plans de chargement (packing) et des tournées.

Entreprises françaises ayant des départements avec des compétences en Recherche Opérationnelle

EDF, SNCF, Bouygues, Renault, GDF-Suez, Saint-Gobain, Air Liquide, France Telecom, Air France, Véolia, la Poste, l'Armée de Terre, la Marine, ...

Quelques solveurs de programmation linéaire

Payant : CPLEX, XPRESS, MINOS

Libres : GLPK, LPSOLVE, CLP, SCIP, SOPLEX

En général, ces solveurs peuvent résoudre bien d'autres types de programmes mathématiques.

Quelques langages de modélisation

AMPL, APL, OPL, TOMLAB, ...