Structures de données et algorithmes[1]

Irène Charon

Olivier Hudry

[1] Les chapitres VI à X, XII et XIII de ce polycopié sont extraits du livre « Méthodes d'optimisation combinatoire » d'I. Charon, A. Germa et O. Hudry, Masson, Paris, 1996.

Table des matières

I. Notions d'algorithme et de complexité

I.1 Algorithme

Un *algorithme* est une suite finie d'instructions non ambiguës pouvant être exécutées de façon automatique. Un algorithme prend en entrée des données et produit un résultat.

Nous verrons de nombreux exemples d'algorithmes. On peut dire qu'une recette de cuisine est un algorithme, l'entrée étant les ingrédients et la sortie le plat cuisiné. Nous décrirons aussi, par exemple, des algorithmes pour trier des éléments d'un ensemble totalement ordonné.

On peut distinguer plusieurs façons de décrire un algorithme.

- Une première méthode consiste à en donner le principe.

  Exemple : pour détecter si un certain élément figure dans une liste non triée, le principe d'un algorithme de recherche séquentielle consiste à comparer successivement tous les éléments de la liste avec l'élément recherché jusqu'à rencontrer l'élément recherché ou bien atteindre la fin de la liste.

- Une deuxième méthode consiste à utiliser du pseudo-code ; il s'agit d'une façon d'exprimer l'algorithme en précisant la façon de ranger les données, les variables, les initialisations et en séparant les différentes instructions ; on utilise un certain formalisme pour exprimer les affectations, les instructions conditionnelles, les boucles, etc. Il y a différents formalismes possibles mais, après en avoir choisi un, il est très préférable de s'y tenir.

  Exemple : le même algorithme que dans l'exemple précédent peut être décrit comme ci-dessous.

Recherche séquentielle dans un tableau

Il y a n données.
Les données se trouvent dans un tableau T à partir de l'indice 1.
La donnée à rechercher est notée *clé*.
On utilise une variable booléenne notée *trouvé* et un entier i.

- $i \leftarrow 1$;
- *trouvé* $\leftarrow$ faux ;
- tant que ((non *trouvé*) et ($i \leq n$))
  début boucle tant que
  - si ($T[i]$ = *clé*), alors *trouvé* $\leftarrow$ vrai ;
    sinon $i \leftarrow i + 1$;

  fin boucle tant que
- renvoyer *trouvé*.

Entre autres choix, le formalisme ci-dessus utilise le symbole « = » pour le test d'égalité et le symbole « ← » pour l'affectation ; cette dernière signifie que la valeur du membre de droite est transmise à la variable du membre de gauche.

On peut éventuellement supprimer les indications telles que « début boucle tant que » et « fin boucle tant que » en s'appuyant sur l'indentation, à condition que celle-ci soit suffisamment claire. C'est ce que nous ferons par la suite.

- Une troisième méthode consiste à traduire l'algorithme dans un langage de programmation.

Les trois méthodes décrites ci-dessus sont de plus en plus précises, de moins en moins ambiguës. Malheureusement, elles sont aussi de moins en moins faciles à lire et à comprendre. Si on veut programmer un algorithme, il sera généralement pertinent d'enchaîner les trois méthodes : d'abord en donner le principe, puis écrire le pseudo-code, avant de passer à la programmation.

Remarquons que, lorsqu'on propose un algorithme, il est généralement nécessaire de le prouver, c'est-à-dire de prouver qu'il fait bien, et dans tous les cas, ce qu'on attend de lui. Il existe des techniques pour effectuer des preuves d'algorithme, techniques que nous ne développerons pas dans ce polycopié.

Implémenter un algorithme signifie le traduire dans un langage de programmation pour pouvoir l'exécuter à l'aide d'un ordinateur. Pour effectuer l'implémentation, il faut choisir la façon de représenter les données, c'est-à-dire choisir la structure des données ; nous développerons différentes possibilités de structures de données dans les chapitres suivants. Pour résoudre un problème, il y a souvent de nombreux algorithmes possibles et, pour chaque algorithme, plusieurs choix de structures de données. Un algorithme assorti d'une structure des données pour son implémentation peut avoir plusieurs caractéristiques :

- *Sa simplicité* : un algorithme simple est facile à comprendre, à implémenter et généralement à prouver.
- *La qualité du résultat obtenu* : par exemple, une recette pour faire une mousse au chocolat peut donner un résultat plus ou moins agréable à déguster.
- *Le temps pris pour effectuer l'algorithme* : là encore, cette caractéristique dépend à la fois de l'algorithme et des structures de données retenues. Il n'est pas matériellement possible de résoudre un problème à l'aide d'un algorithme nécessitant quelques milliards d'années pour s'exécuter. On peut aussi préférer un algorithme qui s'exécutera en une seconde plutôt qu'en une heure. De plus, la ressource temps d'un ordinateur est souvent précieuse et on cherche alors à l'économiser. On verra que, d'un algorithme à l'autre, et d'une structure de données à l'autre, les temps d'exécution peuvent être différents. Cette caractéristique de temps d'exécution s'exprime avec ce qu'on appelle la *complexité* (en temps) de l'algorithme.
- *La place prise en mémoire* (complexité en place) : cette place va dépendre à la fois de l'algorithme et des structures de données retenues ; il ne faut pas dépasser les capacités de la mémoire de la machine sur laquelle l'algorithme s'exécutera et, si la place prise en mémoire est importante, on peut être conduit à limiter la taille des problèmes considérés.

Dans la suite, quand nous parlerons de complexité, il s'agira toujours de complexité en temps.

Les relations entre algorithmes et structures de données sont variables ; la plupart des algorithmes, dès la formulation de leur principe, s'appuient sur une certaine structure de données. Parmi les exemples que nous verrons plus tard, le tri rapide s'appuie sur une structure linéaire des données, le tri par arbre binaire de recherche sur une structure d'arbre binaire de recherche, le tri tas sur une structure de tas, les algorithmes de graphes sur une structure selon des graphes… Il ne reste souvent qu'à préciser la structure de données pour passer de l'algorithme à son implémentation.

Il arrivera fréquemment que, entre deux structures de données, l'une soit meilleure en ce qui concerne la place prise en mémoire alors que l'autre est meilleure en ce qui concerne la complexité.

I.2 Complexité

Étant donné un algorithme, on appellera *opérations élémentaires* :

- un accès en mémoire pour lire ou écrire la valeur d'une variable ou d'une case d'un tableau ;
- une opération arithmétique entre entiers ou entre réels : addition, soustraction, multiplication, division, calcul du reste dans une division entière ;
- une comparaison entre deux entiers ou deux réels.

Exemple : si on considère l'instruction $c \leftarrow a + b$, on peut compter quatre opérations élémentaires :

- l'accès en mémoire pour lire la valeur de a,
- l'accès en mémoire pour lire la valeur de b,
- l'addition de a et b,
- l'accès en mémoire pour écrire la nouvelle valeur de c.

Pour un problème donné, on appelle *instance* tout jeu de données de ce problème. Par exemple, pour un problème de tri, on obtient une instance en spécifiant la valeur numérique des nombres à trier.

Soient f et g deux fonctions positives d'une même variable entière n (resp. de deux mêmes variables entières n et m, ou plus). La fonction f est dite avoir un ordre de grandeur au plus égal à celui de la fonction g s'il existe un entier strictement positif k et un entier N (resp. deux entiers N et M) tels que, pour tout $n \geq N$ (resp. $n \geq N$ et $m \geq M$), on ait $f(n) \leq k\, g(n)$ (resp. $f(n, m) \leq k\, g(n, m)$) ; on écrira $f = \mathrm{O}(g)$ (notation de Landau). Les deux fonctions sont dites avoir même ordre de grandeur si l'ordre de grandeur de l'une est au moins égal à l'ordre de grandeur de l'autre et réciproquement ; on pourra écrire : $f = \Theta(g)$. Par exemple, les fonctions $f(n) = 3\, n^2 - 5\, n + 4$ et $g(n) = n^2$ ont même ordre de grandeur. On dira aussi que g est un ordre de grandeur de f.

On considère un algorithme $\mathcal{A}$. On appelle *complexité de* $\mathcal{A}$ tout ordre de grandeur du nombre d'opérations élémentaires effectuées pendant le déroulement de l'algorithme. On exprime ce nombre d'opérations en fonction de paramètres associés aux instances à traiter ; on pourra par exemple exprimer la complexité d'un tri en fonction du nombre de données à trier. Néanmoins, il se peut qu'avec deux jeux de données différents correspondant aux mêmes paramètres, la complexité ne soit pas la même. Par exemple, un algorithme de tri

pourra être plus rapide s'il s'agit de trier des données déjà triées que s'il s'agit de données très désordonnées. On peut alors s'intéresser à :

- la complexité *dans le pire des cas* : les paramètres avec lesquels on exprime la complexité étant fixés, on considère le plus grand nombre d'opérations élémentaires effectuées sur l'ensemble des instances correspondant à ces paramètres ; on cherche ainsi un majorant de la complexité qui puisse être atteint dans certains cas ; c'est ce qu'on fait le plus généralement ;
- la complexité *dans le meilleur des cas* : les paramètres avec lesquels on exprime la complexité étant fixés, on considère le plus petit nombre d'opérations élémentaires effectuées sur l'ensemble des instances correspondant à ces paramètres ; cette complexité peut venir compléter la précédente mais ne sera jamais suffisante pour l'utilisateur ;
- la complexité *en moyenne* : les paramètres avec lesquels on exprime la complexité étant fixés, il faut alors faire la moyenne des nombres d'opérations élémentaires effectuées, moyenne portant sur tous les jeux de données correspondant à ces paramètres. Ce calcul est généralement difficile et souvent même délicat à formuler car il faut connaître la probabilité de chacun des jeux de données pour effectuer un calcul pertinent de cette moyenne.

Dans certains cas, on peut être amené à ne calculer qu'un majorant (qu'on essaie de rendre le plus petit possible) de la complexité d'un algorithme.

Prenons le cas de la recherche d'un élément dans une liste non triée de n éléments avec l'algorithme développé dans la section 1.1. Le pire des cas est celui où la donnée ne figure pas, où la complexité est de l'ordre de n. Le meilleur des cas est celui où l'élément recherché est le premier de la liste, cas où la complexité est de l'ordre de 1. Si maintenant on veut faire une moyenne sur tous les jeux de données à n éléments et tout élément recherché, on peut supposer que, dans le cas où l'élément recherché figure dans la liste, toutes les places sont équiprobables ; mais quelle est la probabilité que l'élément recherché figure ? On peut la poser comme étant égal à 0,5, mais c'est arbitraire. Néanmoins, pour l'exemple, très simple, quelle que soit la façon de calculer la moyenne, on obtient un ordre de n ; on dira que cet algorithme est *linéaire* en moyenne.

II. Premières structures de données

II.1 Introduction

Les *structures de données* spécifient la façon de représenter les données du problème considéré ; cela est nécessaire en particulier pour un traitement par un algorithme à l'aide d'un ordinateur.

Deux préoccupations principales interviendront dans le choix d'une telle structure : la place qu'elle consomme dans la mémoire de l'ordinateur et la facilité qu'elle offre quand on cherche à accéder à une certaine donnée. Les structures de données sont définies indépendamment du langage de programmation qui interviendra dans l'écriture finale du programme qui les manipulera ; on supposera néanmoins que ce langage offre les outils nécessaires (par exemple les pointeurs) pour définir et manipuler ces structures de données. La plupart des langages de programmation permettent d'attribuer et d'utiliser la mémoire disponible de différentes façons :

- sous forme de *variables* isolées les unes des autres dans la mémoire de la machine ; elles peuvent être d'un type prédéfini par le langage (le type entier, le type réel, etc.) ou d'un type défini par l'utilisateur à partir des types prédéfinis précédents, par exemple des types conçus pour décrire des variables comprenant plusieurs données hétérogènes, type nommé *structure* dans le langage C ou *enregistrement* dans d'autres langages ; nous appellerons *structure* une variable appartenant à un type comportant plusieurs composantes (hétérogènes ou non) et nous appellerons *champs* les composantes d'une structure ;
- sous forme de *tableaux*, c'est-à-dire d'une suite de variables de même type associées à des « cases » consécutives dans la mémoire ; cette « consécutivité » permet à l'ordinateur de savoir où sont rangés les éléments du tableau, ce qui permet d'avoir accès directement à une variable du tableau à partir d'un indice donné.
- à l'aide de *pointeurs* ou *adresses* ou *références*, indiquant la localisation dans la mémoire de l'objet auquel on s'intéresse.

Dans certains cas (variables isolées, tableaux statiques), la place en mémoire est attribuée par le compilateur et ne peut donc pas être modifiée au cours du programme. Dans d'autres cas (tableaux dynamiques, listes chaînées), l'attribution de la mémoire nécessaire est effectuée pendant le déroulement du programme et peut donc varier pendant celui-ci.

Les structures de données classiques appartiennent le plus souvent aux familles suivantes :

- les *structures linéaires* : il s'agit essentiellement des structures représentables par des *listes linéaires*, c'est-à-dire des tableaux ou des listes chaînées unidimensionnelles ; on y trouve en particulier les *piles*, pour lesquelles les données sont ajoutées ou supprimées à

partir d'une même extrémité, et les *files*, pour lesquelles les données sont ajoutées à une extrémité tandis qu'elles sont supprimées à l'autre ;

- les *structures arborescentes*, avec la sous-famille importante des *arbres binaires* ;
- les *structures relationnelles* : celles-ci prennent en compte des relations existant ou n'existant pas entre les entités qu'elles décrivent ; les relations binaires sont représentables sous forme de *graphes* (orientés ou non).

II.2 Structures linéaires

Les structures linéaires tirent leur nom du fait que les données y sont organisées sous forme d'une liste dans laquelle elles sont mises les unes derrière les autres. On peut représenter une telle liste à l'aide d'un tableau unidimensionnel ou sous la forme d'une liste chaînée. Selon la nature des opérations autorisées, on obtient différents types de listes, en particulier les piles et les files.

II.2.1 Listes

Une liste est une suite ordonnée d'éléments d'un type donné ; une liste peut contenir zéro, un ou plusieurs éléments.

Exemples :

- (3, 7, 2, 8, 10, 4) est une liste d'entiers ;
- (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche) est la liste des jours de la semaine.

La *longueur* d'une liste est son nombre d'éléments. Une liste *vide* est une liste de longueur nulle. La *tête* de la liste est son premier élément. La définition de la *queue* d'une liste n'est pas universelle ; certains la définissent comme étant le dernier élément de la liste et d'autres comme la liste obtenue en enlevant la tête.

On s'intéresse souvent à des listes d'éléments appartenant à un ensemble sur lequel existe une relation d'ordre total (entiers, chaînes de caractères, objets identifiés par un entier…). C'est le cas quand on cherche à trier les éléments de la liste, c'est-à-dire à permuter les éléments de la liste pour qu'ils deviennent placés par ordre croissant (ou décroissant).

Opérations sur les listes

Les opérations habituelles sur les listes sont les suivantes.

- *Insertion* d'un élément au début, à la fin, ou entre deux éléments de la liste. En particulier, on pourra chercher à insérer un élément dans une liste triée à une bonne place, c'est-à-dire en conservant le fait que la liste est triée.
- *Suppression* d'un élément quelconque.
- *Recherche* : cette opération peut renvoyer *vrai* ou *faux* selon qu'un élément donné figure ou non dans la liste ; elle peut aussi renvoyer l'adresse de l'élément recherché sous forme selon les cas d'un indice d'un tableau ou d'un pointeur.

- *Concaténation* de deux listes *L1* et *L2* contenant des éléments d'un même type : cette opération donne la liste obtenue en mettant d'abord les éléments de *L1* puis ceux de *L2*.

On pourrait ajouter l'opération de *tri*, mais nous consacrerons un chapitre entier à ce sujet, que nous laissons pour l'instant de côté.

Implémentation d'une liste par un tableau

On peut représenter une liste par un tableau. L'indice du début de la liste sera alors fixé (en général, ce sera l'indice 0 ou l'indice 1). Il faudra connaître et actualiser le cas échéant le nombre *n* d'éléments de la liste pour savoir d'où à où les éléments se trouvent dans le tableau.

Si on veut insérer un nouvel élément dans la liste :

- s'il s'agit d'insérer un élément à sa place dans une liste triée, il faudra décaler tous les éléments suivants d'une case, auquel cas la complexité dans le pire des cas est de l'ordre de la longueur de la liste ;
- si l'ordre des éléments n'importe pas, on insérera en général l'élément nouveau à la fin du tableau, la complexité étant alors de l'ordre de 1.

Si on veut supprimer un élément de la liste :

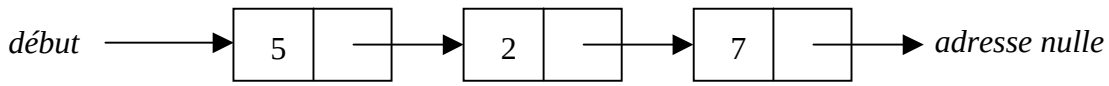
- s'il s'agit de supprimer un élément d'une liste triée, il faudra décaler tous les éléments suivants, d'où une complexité au pire de l'ordre de la longueur de la liste ;
- si l'ordre des éléments n'importe pas, le plus rapide sera de mettre à la place de l'élément à supprimer le dernier élément de la liste.

Si maintenant on veut concaténer deux listes *L1* et *L2*, soit on recopie *L1* puis *L2* dans un nouveau tableau, soit on recopie les éléments de *L2* à la suite de ceux de *L1* dans le tableau qui contenait *L1* ; dans les deux cas, il faut évidemment que les tableaux soient suffisamment grands. La complexité est soit de l'ordre du nombre total d'éléments de *L1* et *L2*, soit de l'ordre du nombre d'éléments de *L2*.

Un inconvénient d'un tableau, par rapport à la liste chaînée que nous développons ci-dessous, est qu'il faut prendre garde à ne pas déborder du tableau. Quand le tableau est plein, il faut soit refuser les insertions, soit prendre un tableau plus grand et y recopier la liste.

Implémentation d'une liste par une liste chaînée

Une liste chaînée est constituée de « maillons » ; les maillons peuvent contenir des données et l'adresse d'un autre maillon. Les données contenues par les maillons sont du même type d'un maillon à l'autre. Une liste chaînée composée des entiers 5, 2 et 7 pourra être représentée comme ci-dessous :



où *début* est l'adresse d'un maillon.

Le maillon qui se trouve à l'adresse *début* contient l'élément qui constitue la tête de la liste.

Pour accéder à l'ensemble des éléments de la liste, on part de l'adresse *début.* On peut avec cette adresse accéder au premier maillon. On connaît ainsi le premier élément de la liste et on lit l'adresse contenue dans ce premier maillon, qui est celle du deuxième maillon ; on peut alors accéder au deuxième maillon pour connaître le deuxième élément de la liste et ainsi de suite. Il est nécessaire qu'une adresse spéciale soit mise dans le dernier maillon de la liste pour qu'on puisse savoir qu'il s'agit du dernier maillon. Cette adresse spéciale est prévue par les langages de programmation qui permettent de manipuler les listes chaînées ; elle s'appelle en général `nil`, ou `NULL`, ou `null` ; c'est toujours en quelque sorte l'adresse qui vaut zéro, notée *adresse nulle* sur notre schéma.

À condition de connaître certaines adresses appropriées, on constate que l'insertion et la suppression d'un élément, ou bien la concaténation de deux listes se fait avec une complexité de l'ordre de 1. En revanche, nous verrons dans le chapitre suivant que la recherche d'un élément dans une liste triée de n éléments peut se faire avec une complexité de l'ordre de $\ln(n)$ lorsqu'on utilise un tableau, alors que la complexité est toujours dans le pire des cas et en moyenne de l'ordre de n lorsque la liste est codée avec une liste chaînée.

La gestion informatique d'une liste chaînée se fait généralement en employant une allocation dynamique de mémoire chaque fois qu'un nouveau maillon est nécessaire ; les adresses sont alors des adresses dans la mémoire principale de l'ordinateur. Néanmoins, on peut gérer une structure de liste chaînée avec un tableau de maillons, les adresses de maillons devenant alors des indices du tableau ; avec cette dernière façon de faire, on retrouve l'inconvénient de la limitation du nombre de données au nombre de cases du tableau de maillons. Dans la suite, nous supposerons implicitement qu'on utilise une allocation dynamique de mémoire pour chaque nouveau maillon.

Dans le pseudo-code plus bas, on notera *rien* l'adresse nulle ; si on considère un maillon d'adresse *p*, on notera (comme en langage Java) *p.donnée* la partie du maillon qui contient la donnée et *p.suivant* la partie du maillon qui contient l'adresse du maillon suivant (en langage C, on note cela $p \rightarrow donnée$ et $p \rightarrow suivant$).

II.2.2 Piles

Une *pile* (*stack* en anglais) est une liste dans laquelle l'insertion ou la suppression d'un élément s'effectue toujours à partir de la même extrémité de la liste, extrémité appelée le *début* de la pile. L'action consistant à ajouter un nouvel élément au début de la pile s'appelle *empiler* ; celle consistant à retirer l'élément situé au début de la pile s'appelle *dépiler*. Une pile permet de modéliser un système régi par la discipline « dernier arrivé - premier sorti » ; on dit souvent qu'il s'agit d'un traitement LIFO (*last in, first out*). La *hauteur* de la pile est son nombre d'éléments.

Les fonctions souvent définies dans l'implémentation d'une pile sont (les fonctions 1, 3 et 4 font partie de toute implémentation d'une pile) :

1. répondre à la question : la pile est-elle vide ?
2. éventuellement, selon l'implémentation, répondre à la question : la pile est-elle pleine ?
3. empiler un élément, en traitant éventuellement le cas où la pile serait pleine (fonction souvent appelée *push* en anglais),
4. dépiler un élément, en traitant le cas où la pile serait vide (fonction souvent appelée *pop* en anglais),

5. éventuellement, donner la valeur de l'élément qui se trouve au début de la pile, sans dépiler celui-ci,
6. éventuellement, vider la pile.

Implémentation d'une pile par un tableau

Si la pile est représentée par un tableau *T*, on doit gérer une variable pour la hauteur de la pile, variable que nous notons *hauteur*.

Supposons que les éléments de la pile soient stockés à partir de l'indice 0 (resp. 1) du tableau ; l'empilement d'un nouvel élément *x* consiste alors à introduire *x* dans la case *T*(*hauteur*) (resp. *T*(*hauteur* + 1)) et à remplacer *hauteur* par *hauteur* + 1 ; le dépilement consiste au contraire à supprimer l'élément *T*(*hauteur* – 1) (resp. *T*(*hauteur*)) de *T* puis à remplacer *hauteur* par *hauteur* – 1.

La pile est vide si la variable *hauteur* vaut zéro.

La pile est pleine si la hauteur est égale à la dimension du tableau.

Pseudo-code

On suppose qu'on utilise un tableau à *N* cases indicées de 1 à *N*.

On utilise une variable *hauteur*. La pile est vide si et seulement si *hauteur* vaut 0, la pile est pleine si et seulement si la *hauteur* vaut *N*. La variable *hauteur* est initialisée à 0.

Pour la fonction *empiler*, on choisit ci-dessous de ne rien faire si on appelle la fonction *empiler* alors que la pile est pleine. Dans un code plus élaboré, ce cas devrait être traité.

empiler(clé)

Si *hauteur* < *N*, alors

- *hauteur* ← *hauteur* + 1;
- *T*[*hauteur*] ← *clé*;

Pour la fonction *dépiler*, on choisit ci-dessous de ne rien faire si on appelle la fonction *dépiler* alors que la pile est vide. Dans un code plus élaboré, ce cas devrait être traité.

dépiler

Si *hauteur* > 0, alors

- *clé* ← *T*[*hauteur*];
- *hauteur* ← *hauteur* – 1;
- renvoyer *clé*;

sinon // à préciser

Implémentation d'une pile par une liste chaînée

Si la pile est représentée par une liste chaînée à laquelle on accède par un pointeur *début*, on ajoute un nouvel élément *clé* de la manière suivante. On crée un nouveau maillon *M* destiné à contenir *clé* et on y insère *clé* dans le champ de *M* prévu à cet effet. Dans le champ de *M* destiné à contenir l'adresse du maillon suivant de la liste après insertion de *clé*, on met

la valeur du pointeur *début*. Enfin, pour accéder à la nouvelle liste à l'aide du pointeur *début*, on attribue à *début* l'adresse de *M*. De la même façon, pour dépiler un élément, on extrait l'élément contenu dans le premier maillon de *L* (celui auquel on accède grâce à *début*) puis on décale *début* en le faisant pointer sur le maillon qui suit le premier maillon ; on peut alors en général libérer l'espace en mémoire alloué au maillon contenant *clé*.

La pile est vide si l'adresse *début* est égale à l'adresse nulle (voir plus haut).

La pile n'est jamais pleine (il faut néanmoins ne pas dépasser la capacité en mémoire de la machine sur laquelle on travaille, ce qui peut se produire en particulier si on empile et qu'on dépile fréquemment et qu'on oublie de désallouer l'espace en mémoire quand on dépile un élément).

Pseudo-code

On utilise une variable *début*. La pile est vide si et seulement si *début* vaut *rien*. La variable *début* est initialisée à *rien*. Une fonction nommée *nouvelle_cellule* permet de créer une cellule et retourne l'adresse de celle-ci.

empiler(clé)

- *p* ← *nouvelle_cellule*;
- *p.donnée* ← *clé*;
- *p.suivant* ← *début*;
- *début* ← *p*;

Pour la fonction *dépiler*, on choisit ci-dessous de ne rien faire si on appelle la fonction *dépiler* alors que la pile est vide. Dans un code plus élaboré, ce cas devrait être traité.

dépiler

Si *début* ≠ *rien*, alors

- *clé* ← *début.donnée*;
- *début* ← *début.suivant*;
- renvoyer *clé*;

sinon // à préciser

Dans cette dernière fonction, il peut convenir, selon le langage de programmation, de libérer l'espace-mémoire qui était alloué à la cellule contenant la donnée qui a été dépilée.

II.2.3 Files

Une *file* est une liste dans laquelle toutes les insertions de nouveaux éléments s'effectuent d'un même côté de la liste appelé *fin* et toutes les suppressions d'éléments s'effectuent toujours à partir de l'autre extrémité, appelée *début*. L'action consistant à ajouter un nouvel élément à la fin de la file s'appelle *enfiler* ; celle consistant à retirer l'élément situé au début de la file s'appelle *défiler*. Une file permet de modéliser un système régi par la discipline « premier arrivé - premier sorti » ; on dit souvent qu'il s'agit d'un traitement FIFO (*first in, first out*).

Implémentation d'une file par un tableau

Si la file est représentée par un tableau *T*, on doit gérer deux indices *début* et *fin* indiquant les cases de *T* entre lesquelles se trouvent toutes les données ; la donnée devant sortir en premier figure dans la case d'indice *début*, tandis que la première case libre, c'est-à-dire la case dans laquelle on placera la prochaine donnée enfilée, est d'indice *fin* + 1. Enfiler un nouvel élément *x* consiste donc à introduire *x* dans la case *T*(*fin* + 1) et à remplacer *fin* par *fin* + 1. Défiler un élément consiste d'autre part à remplacer *début* par *début* + 1, après avoir traité l'élément *T*(*début*). En pratique, la dimension de *T* est une constante *dim* : le nombre d'éléments présents simultanément dans *T* ne peut donc pas excéder *dim*. Si d'autre part le nombre d'entrées et de sorties est grand devant *dim*, on a intérêt à gérer *T* de manière cyclique pour éviter un débordement : quand il n'y a plus de place à la fin du tableau pour ajouter un élément, on recommence à remplir *T* à partir de ses premières cases (dans ce cas, *fin* devient plus petit que *début*).

Pseudo-code

On suppose qu'on utilise un tableau à *N* cases indicées de 1 à *N*.

On utilise deux variables *début* et *fin*. On peut envisager un traitement cyclique de la file mais ce n'est pas ce qui est fait ici. On fait en sorte que la file soit vide si et seulement si *fin* < *début*. La variable *début* est initialisée à 1 et la variable *fin* est initialisée à 0. Quand la file n'est pas vide, les données se trouvent entre l'indice *début* compris et l'indice *fin* compris.

Pour la fonction *enfiler*, on choisit ci-dessous de ne rien faire si on appelle la fonction *enfiler* alors que *fin* vaut *N*. Dans un code plus élaboré, ce cas devrait être traité.

enfiler(clé)

Si *fin* < *N*, alors

- *fin* ← *fin* + 1;
- *T*[*fin*] ← *clé*;

Pour la fonction *défiler*, on choisit ci-dessous de ne rien faire si on appelle la fonction *défiler* alors que la file est vide. Dans un code plus élaboré, ce cas devrait être traité.

défiler

Si *début* ≤ *fin*, alors

- *clé* ← *T*[*début*];
- *début* ← *début* + 1;
- renvoyer *clé*;

sinon // à préciser

Implémentation d'une file par une liste chaînée

Si la file est représentée par une liste chaînée, il est commode de maintenir à jour deux pointeurs, *début* et *fin*, la liste étant chaînée de *début* vers *fin*. Pour défiler, on décale *début* en le faisant pointer sur le maillon qui suit celui sur lequel il pointe avant cette opération, après avoir traité l'élément du maillon supprimé. Pour enfiler un nouvel élément *x*, on crée un nouveau maillon *M* dans lequel on place *x* et dans lequel l'adresse du maillon suivant est

l'adresse nulle. Puis on ajoute *M* à la fin de la liste à l'aide du pointeur *fin* que l'on décale pour le faire pointer sur *M*.

Pseudo-code

On utilise une variable *début* et une variable *fin*. On fait en sorte que la file soit vide si et seulement si *fin* vaut *rien*. Les variables *début* et *fin* sont initialisées à *rien*. On constatera dans le pseudo-code ci-dessous que *fin* vaut *rien* si et seulement si *début* vaut *rien*. Une fonction nommée *nouvelle_cellule* permet de créer une cellule et retourne l'adresse de celle-ci.

enfiler(clé)

- $p \leftarrow$ *nouvelle_cellule*;
- *p.donnée* $\leftarrow$ *clé*;
- *p.suivant* $\leftarrow$ *rien*;
- Si *fin* $\neq$ *rien*, alors *fin.suivant* $\leftarrow$ *p*;

  sinon *début* $\leftarrow$ *p*;
- *fin* $\leftarrow$ *p*;

Pour la fonction *défiler*, on choisit ci-dessous de ne rien faire si on appelle la fonction *défiler* alors que la file est vide. Dans un code plus élaboré, ce cas devrait être traité.

défiler

Si *début* $\neq$ *rien*, alors

- *clé* $\leftarrow$ *début.donnée*;
- *début* $\leftarrow$ *début.suivant*;
- si *début* = *rien* alors *fin* $\leftarrow$ *rien*;
- renvoyer *clé*;

sinon // à préciser

Dans cette dernière fonction, il peut convenir, selon le langage de programmation, de libérer l'espace-mémoire qui était alloué à la cellule contenant la donnée qui a été défilée.

II.3. Arbres

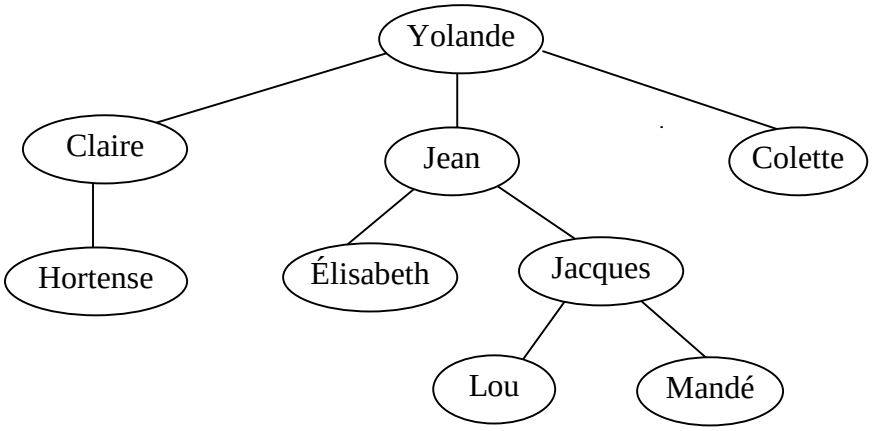
II.3.1 Arbres généraux

La définition que nous donnons d'un arbre est récursive[2].

Un arbre est une structure composée d'éléments appelés *nœuds* ; il est constitué d'un nœud particulier appelé *racine* et d'une suite ordonnée éventuellement vide A_1, A_2, …, A_p d'arbres disjoints appelés *sous-arbres* de la racine. Un arbre contient donc au moins un nœud : sa racine.

[2] Notons que les arbres dont il s'agit ici ne correspondent pas exactement aux arbres de la théorie des graphes, même si certaines propriétés leur sont communes.

Exemple : la figure suivante représente l'arbre d'une descendance ; la racine de cet arbre contient Yolande ; il est constitué de trois sous-arbres ($p = 3$) : A_1, de racine Claire, contient Claire et Hortense ; A_2, de racine Jean, contient Jean, Élisabeth, Jacques, Lou et Mandé ; A_3, de racine Colette, est réduit à Colette. L'arbre est ici dessiné de haut en bas (la racine est en haut) et les sous-arbres de gauche à droite ; d'autres représentations sont possibles.



Plusieurs définitions sont à retenir ; nous illustrons les définitions avec l'exemple ci-dessus en identifiant un nœud avec la chaîne de caractères qu'il contient :

- les fils d'un nœud sont les racines de ses sous-arbres ; sur l'exemple, les fils de Jean sont Élisabeth et Jacques ;
- le père d'un nœud *x* autre que la racine est l'unique nœud dont *x* est un fils ; sur l'exemple, Jacques est le père de Lou et de Mandé ; la racine d'un arbre n'a pas de père ;
- un *nœud interne* est un nœud qui a au moins un fils ; sur l'exemple, Claire est un nœud interne ;
- une *feuille* d'un arbre est un nœud sans fils ; sur l'exemple, Mandé est une feuille ;
- les *ancêtres* d'un nœud *a* sont les nœuds qui sont sur le chemin entre la racine (incluse) et *a* (inclus) ; les ancêtres de *a* différents de *a* sont les ancêtres propres de *a* ; sur l'exemple, les ancêtres de Jacques sont Jacques, Jean et Yolande ;
- les *descendants* d'un nœud *a* sont les nœuds qui appartiennent au sous-arbre de racine *a* ; les descendants de *a* différents de *a* sont les descendants propres de *a* ; sur l'exemple, les descendants de Jean sont Jean, Élisabeth, Jacques, Lou et Mandé.
- la *profondeur* d'un nœud est la longueur du chemin allant de la racine à ce nœud ; la profondeur de la racine est donc nulle et la profondeur d'un nœud autre que la racine vaut un de plus que la profondeur de son père ; la profondeur d'un nœud peut aussi être appelée *niveau* ou *hauteur* du nœud ; sur l'exemple, la profondeur d'Élisabeth vaut 2 ;
- la *hauteur* d'un arbre est la profondeur maximum de ses nœuds ; la hauteur de l'arbre donné en exemple vaut 3.

Pour connaître un arbre, on procède généralement en mémorisant la racine (ou plus précisément son adresse) ; il suffit alors que chaque nœud connaisse ses nœuds fils pour pouvoir retrouver tout l'arbre.

II.3.2 Parcours d'un arbre général

Parcourir un arbre, c'est examiner une fois et une seule chacun de ses nœuds, l'ordre des examens dépendant d'une certaine règle. On distingue les deux types suivants. Diverses opérations peuvent être effectuées quand on examine un nœud (par exemple, lui attribuer un numéro).

Le parcours en préordre

Parcourir un arbre en *préordre*, c'est :

- examiner la racine ;
- parcourir en préordre le premier sous-arbre ;
- …
- parcourir en préordre le dernier sous-arbre.

Ce parcours est aussi dit *parcours en ordre préfix*e.

On peut se convaincre que la complexité de ce parcours est du même ordre que le nombre de nœuds de l'arbre.

Exemple : le parcours en préordre de l'arbre donné en exemple considère successivement les nœuds Yolande, Claire, Hortense, Jean, Élisabeth, Jacques, Lou, Mandé, Colette.

Le parcours en postordre

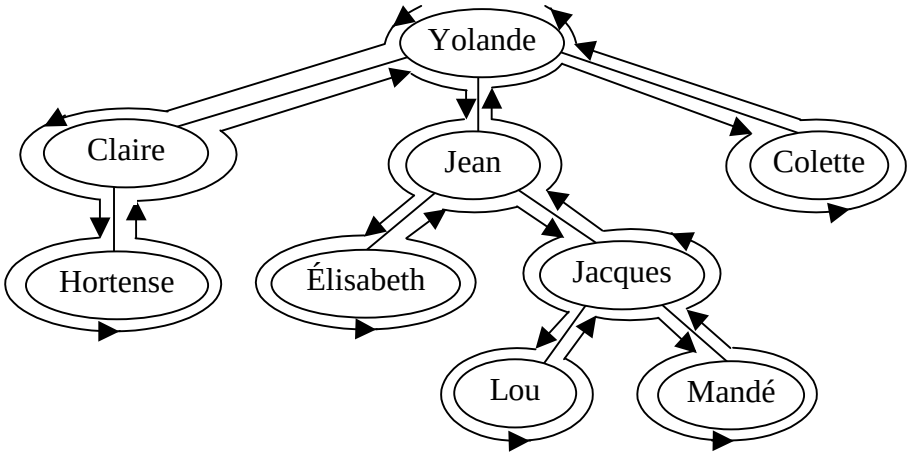
Parcourir un arbre en *postordre*, c'est :

- parcourir en postordre le premier sous-arbre ;
- …
- parcourir en postordre le dernier sous-arbre ;
- examiner la racine.

Ce parcours est aussi dit parcours en *ordre postfixe* ou *suffixe*. Le parcours en ordre postfixe a clairement la même complexité que le parcours en préordre, c'est-à-dire de l'ordre du nombre de nœuds de l'arbre.

Exemple : le parcours en postordre de l'arbre donné en exemple considère successivement les nœuds Hortense, Claire, Élisabeth, Lou, Mandé, Jacques, Jean, Colette, Yolande.

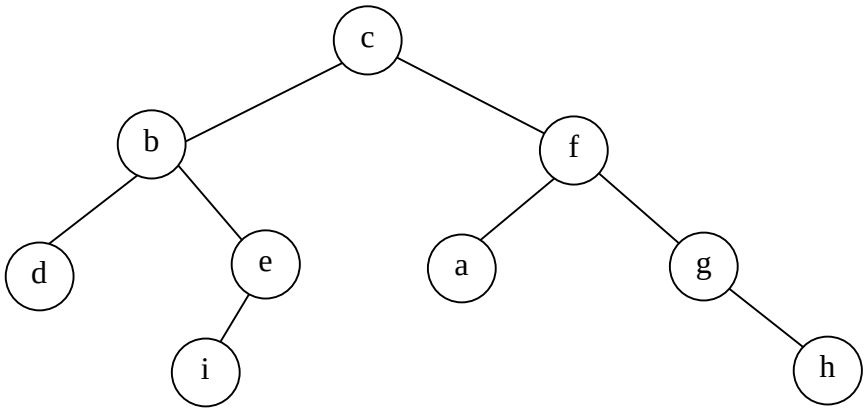
Remarque : on peut voir les ordres obtenus par les parcours en préordre et en postordre sur l'ensemble des noeuds d'une autre façon en considérant ce qu'on appelle « un parcours à main gauche » de l'arbre. On part de la racine et on suit le tracé de l'arbre comme l'indique la figure ci-dessous. On s'aperçoit que l'examen en préordre est obtenu en examinant les nœuds à la première rencontre alors que l'examen en postordre est obtenu en examinant les nœuds à la dernière rencontre.



II.3.3 Arbres binaires

Un *arbre binaire* est soit vide, soit constitué d'un nœud particulier appelé racine et de deux sous-arbres binaires disjoints appelés sous-arbre gauche et sous-arbre droit. On notera donc que, formellement, un arbre binaire n'est pas un arbre, au sens donné ci-dessus (un arbre n'est pas vide et n'utilise pas la notion de droite et de gauche). Cependant, la forte parenté qui existe entre ces deux concepts fait que l'on est souvent amené à étendre aux arbres binaires certaines définitions valables en principe seulement pour les arbres généraux. Ce sera par exemple le cas plus bas pour les parcours en préordre ou en postordre.

Exemple d'arbre binaire :

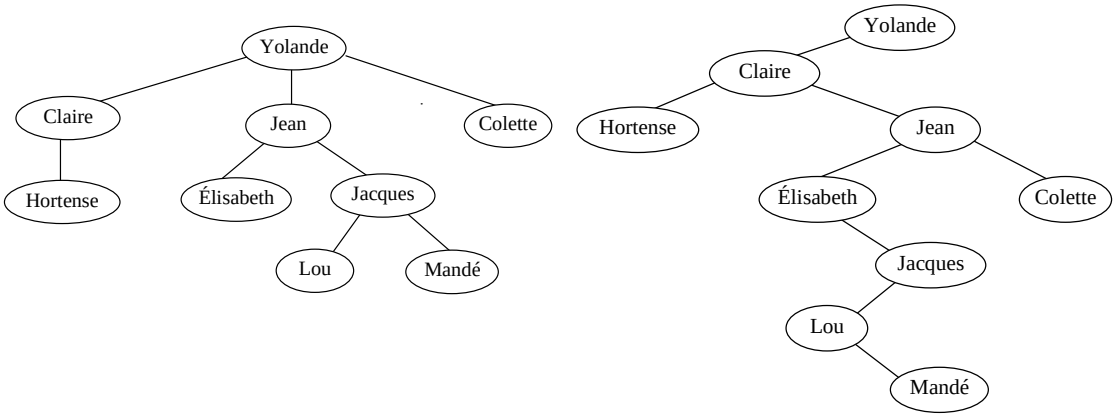


La terminologie est la même que celle des arbres généraux ; néanmoins, si on considère les fils d'un nœud, on parlera, s'ils existent, du fils gauche et du fils droit. Sur l'exemple, on remarque que certains nœuds peuvent avoir seulement un fils droit ou seulement un fils gauche ; par exemple, le nœud e (*i.e.* contenant la donnée e) admet uniquement un fils gauche, le nœud i, et pas de fils droit ; si on considérait l'exemple comme un arbre général, on dirait que e a un seul fils, le nœud i, et le lien entre e et i serait dessiné verticalement.

Pour coder un arbre binaire, on fait souvent correspondre à chaque nœud une structure contenant la donnée et deux adresses, une adresse pour chacun des deux nœuds fils, avec la convention qu'une adresse nulle indique un arbre binaire vide. Il suffit alors de mémoriser l'adresse de la racine pour pouvoir reconstituer tout l'arbre.

Remarque : on code souvent un arbre général en utilisant un arbre binaire ; pour cela on associe à chaque nœud de l'arbre initial un nœud de l'arbre binaire ; le fils gauche d'un nœud

dans l'arbre binaire correspond au premier fils du nœud correspondant de l'arbre initial (on met une adresse nulle s'il n'y a aucun fils) alors que le fils droit de ce nœud correspond au premier « frère cadet », c'est-à-dire au nœud de l'arbre initial qui a même père que le nœud considéré et qui se trouve immédiatement à sa droite (on met une adresse nulle s'il n'y a aucun frère cadet). Ainsi, l'arbre général donné en exemple précédemment et retracé ci-dessous à gauche peut être codé avec l'arbre binaire représenté à droite, en changeant bien entendu l'interprétation des liens.



Parcours ; le parcours en ordre symétrique

Les deux parcours définis pour les arbres généraux sont aussi définis pour les arbres binaires. Pour l'exemple ci-dessus, le parcours en préordre donne l'ordre : c, b, d, e, i, f, a, g, h et le parcours en postordre donne l'ordre : d, i, e, b, a, h, g, f, c.

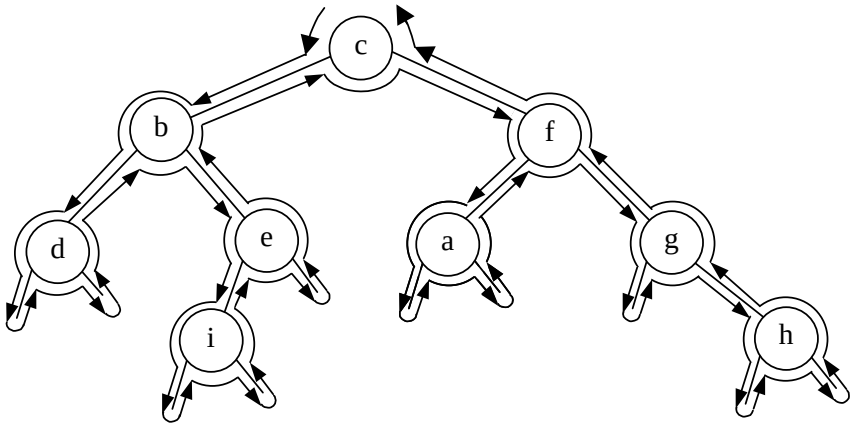
On considère pour les arbres binaires un troisième ordre : l'*ordre symétrique*. Parcourir un arbre binaire en ordre symétrique, c'est :

- si l'arbre binaire est non vide, alors
  - parcourir en ordre symétrique le sous-arbre gauche ;
  - examiner la racine ;
  - parcourir en ordre symétrique le sous-arbre droit.

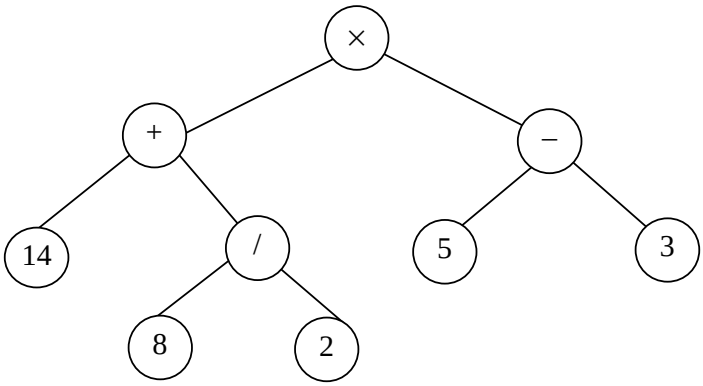
Ce parcours est aussi dit parcours en *ordre infixe* ou *milieu*. La complexité du parcours en ordre symétrique est, de même que pour les autres parcours, de l'ordre du nombre de nœuds de l'arbre.

Exemple : le parcours en ordre symétrique de l'arbre binaire donné en exemple conduit à l'ordre : d, b, i, e , c, a, f, g, h.

Remarque : on peut illustrer les trois parcours des arbres binaires à l'aide du parcours à main gauche ; il est plus agréable pour cela de faire figurer sur le tracé de l'arbre les sous-arbres vides comme il est fait ci-dessous ; le préordre correspond à la première fois que l'on rencontre les nœuds, l'ordre symétrique à la deuxième fois et le postordre à la troisième fois.



Exemple d'application : on peut utiliser un arbre binaire pour représenter une expression arithmétique composée à partir des entiers et des quatre opérateurs binaires : +, –, ×, /. L'arbre ci-dessous représente l'expression : $(14 + (8 / 2)) \times (5 – 3)$



Par construction, dans cet arbre, tout nœud interne a un fils gauche et un fils droit.

Les trois parcours de cet arbre donnent :

- parcours en préordre : × + 14 / 8 2 – 5 3 ; on reconnaît la forme polonaise de l'expression ;
- parcours en ordre infixe : 14 + 8 / 2 × 5 – 3 ; il faudrait ajouter les parenthèses pour retrouver l'expression initiale ;
- parcours en postordre : 14 8 2 / + 5 3 – × ; on reconnaît la forme polonaise inverse de l'expression.

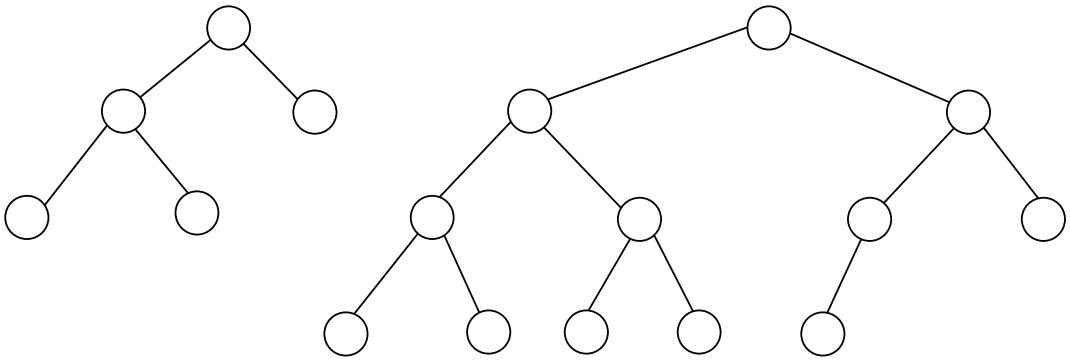
Quelques arbres binaires particuliers

Un arbre binaire est dit *complet* si tout nœud interne a exactement deux fils[3].

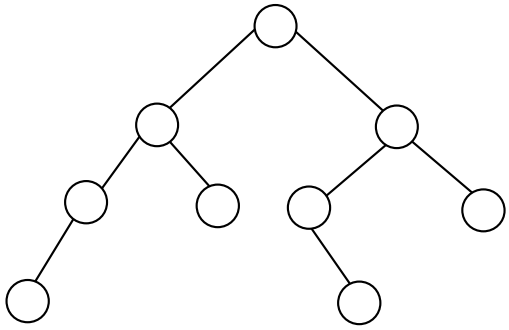
Un arbre binaire est dit *parfait* si, en appelant h la hauteur de l'arbre, les niveaux de profondeur 0, 1, …, $h – 1$ sont complètement remplis alors que le niveau de profondeur h est rempli en partant de la gauche ; la structure d'un arbre binaire parfait est complètement

[3] Certains auteurs adoptent une définition plus contraignante pour les arbres complets, en imposant de plus à toutes les feuilles d'un tel arbre d'avoir la même profondeur.

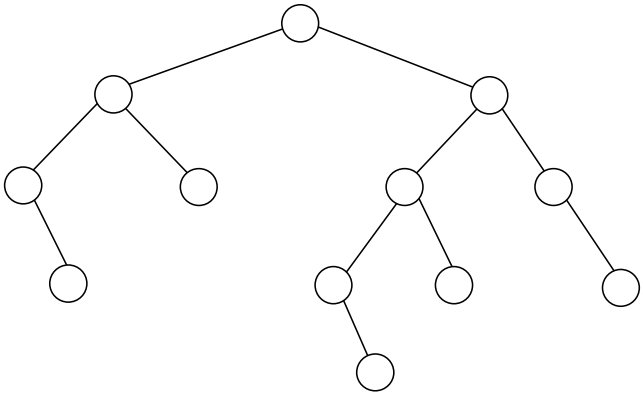
déterminée par son nombre de nœuds ; nous donnons ci-après les arbres binaires parfaits à 5 nœuds et à 12 nœuds. On rencontre aussi quelquefois le qualificatif de *presque-complet* pour un tel arbre.



En revanche, l'arbre ci-contre n'est pas parfait.



Un arbre binaire est dit *équilibré* si, pour tout nœud de l'arbre, les sous-arbres gauche et droit ont des hauteurs qui diffèrent au plus de 1. L'arbre ci-dessous est équilibré.



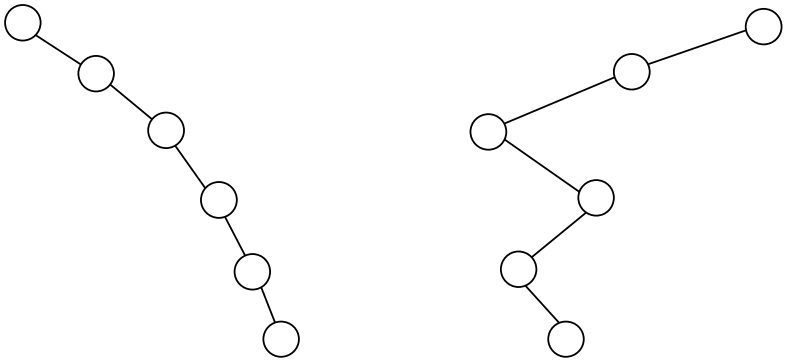
Un arbre binaire parfait est équilibré.

On pourrait démontrer que la hauteur h d'un arbre binaire équilibré ayant n nœuds vérifie :

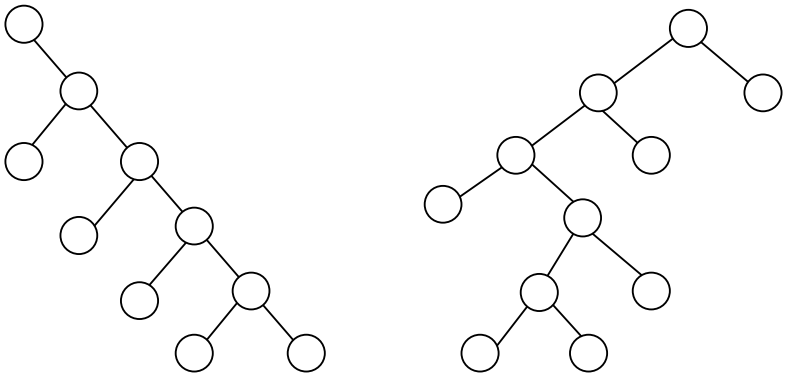
$$\log_2(n + 1) \leq h \leq 1{,}44 \log_2(n + 2),$$

d'où on peut déduire facilement que la profondeur moyenne des nœuds d'un arbre binaire équilibré est du même ordre que ln n.

En ce qui concerne un arbre binaire quelconque de n nœuds, on établirait facilement que la profondeur ainsi que la hauteur moyenne des feuilles ont toujours un ordre de grandeur compris entre ln n et n. La hauteur de l'ordre de n est atteinte pour des arbres qui sont « tout en hauteur » comme on peut en voir ci-après ; ces arbres seront dits *dégénérés*.



Des arbres binaires complets peuvent aussi avoir une hauteur de l'ordre du nombre de nœuds comme l'illustrent les deux arbres ci-dessous, qualifiés aussi de *dégénérés*.



III. Recherche et tri

III.1. Introduction

On considère des données appartenant à un ensemble totalement ordonné. Les trier consiste alors à les ranger en ordre croissant ou décroissant. Quant à rechercher une donnée, c'est soit simplement déterminer si la donnée figure ou non, soit, dans le cas où la donnée figure, trouver l'endroit où cette donnée est stockée.

Les opérations de recherche et de tri consomment un pourcentage important du temps total de calcul des ordinateurs. C'est pourquoi les gains sur la complexité des algorithmes de tri et de recherche ont une importance considérable : aussi ont-ils été très étudiés et perfectionnés.

Dans tout ce chapitre, les données que nous considérerons seront des entiers. Néanmoins, remarquons qu'il est rare de trier simplement des entiers ; on rencontre plus fréquemment des tris de structures selon un champ de cette structure. On peut trier des structures comportant un nom d'élève et une note de cet élève par liste alphabétique des noms des élèves ou par notes croissantes ; on peut trier des factures par dates croissantes ; on peut trier des cartes à jouer par couleurs et à l'intérieur de chaque couleur par niveaux et ainsi de suite. Il faudra considérer que le type des données considérées est un type quelconque sur lequel est définie une relation d'ordre total notée avec les symboles habituels $<$, $>$, $\leq$, $\geq$.

Nous ne traitons pas tous les algorithmes de tri et de recherche dans ce chapitre, ni dans l'ensemble du polycopié, mais seulement une partie de ce qui peut être dit sur le sujet. Nous laissons pour les chapitres suivants les algorithmes de recherche et de tri utilisant une structure d'arbre binaire de recherche ainsi que l'algorithme de tri utilisant une structure de tas.

III.2 Recherche dichotomique dans un tableau trié

On suppose ici que l'on dispose d'un tableau trié par ordre croissant et qu'on souhaite savoir si un élément fixé (i.e. une donnée) figure ou non dans la liste.

On peut bien sûr utiliser une recherche séquentielle qui consiste à regarder successivement, en partant de la gauche du tableau, tous les éléments, selon l'exemple vu en début de ce polycopié. Néanmoins, il sera plus astucieux d'utiliser une *recherche dichotomique*.

La recherche dichotomique consiste à comparer l'élément, ou un des deux éléments, qui se trouve au centre de la liste avec l'élément recherché ; si cet élément central est égal à l'élément recherché, c'est terminé ; s'il est supérieur à l'élément recherché, on abandonne la moitié droite de la liste (correspondant aux éléments plus grands que l'élément central) et on recommence avec la liste formée par la moitié gauche (correspondant aux éléments plus petits que l'élément central) de la liste initiale ; s'il est inférieur à l'élément recherché, on

abandonne la moitié gauche de la liste (correspondant aux éléments plus petits que l'élément central) et on recommence avec la liste formée par la moitié droite (correspondant aux éléments plus grands que l'élément central) de la liste initiale. Et ainsi de suite jusqu'à ce que la liste restante soit vide (recherche infructueuse) ou bien avoir trouvé l'élément cherché (recherche fructueuse).

Écrivons le pseudo-code de cet algorithme qui renvoie *vrai* ou *faux* selon que l'élément recherché est trouvé ou non. Si on recherche l'indice de l'élément dans le cas où il figure, il faut adapter l'algorithme.

Recherche dichotomique dans un tableau trié

Les données sont rangées dans un tableau *T* entre les indices 1 et *n*.
On utilise trois variables entières notées *gauche*, *droite* et *milieu*. On utilise aussi une variable booléeenne notée *trouvé*. L'élément recherché s'appelle *clé*. La relation d'ordre entre les éléments s'exprime avec le signe '<'. La division utilisée est la division entière.

- *gauche* ← 1 ;
- *droite* ← *n* ;
- *trouvé* ← *faux* ;
- tant que ((*non trouvé*) et (*gauche* ≤ *droite*))
  - *milieu* ← (*gauche* + *droite*)/2 ;
  - si (*clé* < *T*[*milieu*]), alors *droite* ← *milieu* – 1 ;
    sinon si (*clé* > *T*[*milieu*]), alors *gauche* ← *milieu* + 1 ;
    sinon *trouvé* ← *vrai* ;
- renvoyer *trouvé*.

Nous allons établir la complexité de cet algorithme. Pour chaque passage dans la boucle tant que, on fait un nombre d'opérations élémentaires majoré par une constante ; par ailleurs, à chaque passage dans la boucle, la longueur de la liste est divisée par 2 (même un peu plus), ce qui fait que si on note *p* le nombre total de passages dans la boucle, au bout de *p* passages, la longueur de la liste est majorée par $n / 2^p$, et au moins égale à $1 : n / 2^p \geq 1$, et donc : $2^p \leq n$; d'où $p \leq \log_2(n)$. La complexité est au pire de l'ordre de $\log_2(n)$ ou, ce qui revient au même, de $\ln(n)$.

III.3. Complexité d'un algorithme de tri

Comme pour tout algorithme, la complexité d'un algorithme de recherche ou de tri peut s'évaluer soit dans le « pire des cas », c'est-à-dire pour un ordre initial des données spécialement défavorable pour cet algorithme, soit dans le « meilleur des cas », soit « en moyenne ». Par « en moyenne », on entend ici la moyenne arithmétique des complexités sur toutes les permutations possibles des données, que l'on suppose, pour ce calcul, deux à deux distinctes.

Si un algorithme de tri doit souvent être exécuté mais qu'il n'est pas impératif d'avoir un temps d'exécution maximum très petit, on pourra utiliser un algorithme de bonne complexité en moyenne mais dont la complexité au pire peut être médiocre. C'est le cas d'un certain

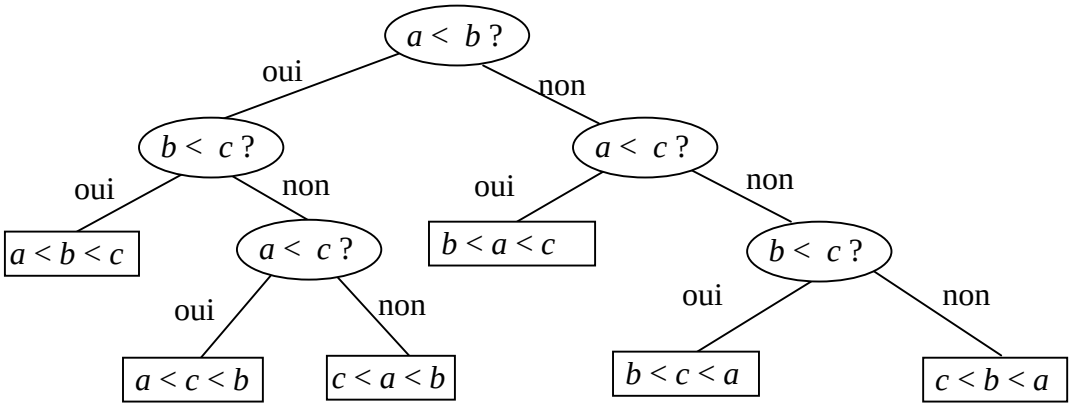
nombre de transactions que l'on peut exécuter en temps différé ; le fait que, dans le « pire des cas », de tels algorithmes ne soient pas très performants n'est pas pénalisant, le pire étant peu probable en principe. En revanche, lorsqu'on travaille en temps réel, on s'intéresse souvent à améliorer la complexité dans le pire des cas. En effet, même si elles sont rares, des attentes très longues peuvent être inacceptables.

La plupart des algorithmes de tri sont fondés sur des comparaisons successives entre les données pour déterminer la permutation correspondant à l'ordre croissant des données. Nous appellerons *tri comparatif* un tel tri. La complexité de l'algorithme a alors le même ordre de grandeur que le nombre de comparaisons entre les données faites par l'algorithme. Nous ne verrons dans ce polycopié que des tris comparatifs.

Signalons néanmoins le *tri baquet* ou le *tri par paquets*, qui sont deux algorithmes de tri non comparatifs. Une forme dégénérée du tri baquet suit le principe ci-dessous. Supposons qu'il s'agisse de trier des entiers compris entre 0 et N ; on utilise un tableau supplémentaire T (autre que celui qui contient éventuellement les données) indicé de 0 à N ; la case d'indice p de T contiendra après l'algorithme le nombre de fois que l'entier p figure dans la liste des entiers. On initialise à 0 toutes les cases du tableau T ; on considère successivement tous les entiers de la liste à trier ; si l'entier considéré vaut p, on incrémente $T[p]$ de 1 ; après avoir traité toutes les données, il suffit de relire le tableau T en partant de l'indice 0 pour connaître la liste triée (si par exemple $T[0]$ vaut 1, $T[1]$ et $T[2]$ valent 0, $T[3]$ vaut 2 et $T[4]$ vaut 1, la liste triée commence par 0, 3, 3, 4). Dans ce tri, on ne compare jamais les données entre elles. En notant n le nombre de données triées, ce tri a pour complexité $n + N$ ou, ce qui revient au même, $\max(n, N)$.

De façon à évaluer la complexité théorique des tris comparatifs, nous allons nous intéresser au nombre de comparaisons faites par un tel algorithme de tri. Nous allons employer une approche intuitive.

Considérons un algorithme de tri de n données ; la réponse cherchée par le tri est l'une des $n!$ permutations possibles des données. Prenons un exemple ; on considère un algorithme pour trier trois données a, b et c décrit par l'arbre ci-dessous ; cet algorithme correspond au fonctionnement du tri insertion que nous verrons plus loin :



Cet arbre signifie : la première comparaison faite est « $a < b$? ». Si la réponse est oui, la comparaison suivante est « $b < c$? », si la réponse est non c'est « $a < c$? ». Lorsqu'une permutation est déterminée, on est dans une feuille de l'arbre. On voit qu'on peut avoir plus ou moins de chance ; pour deux des permutations, on fait deux comparaisons, pour les quatre

autres, on fait trois comparaisons. Le plus grand nombre de comparaisons est 3, le meilleur est 2 et le nombre moyen de comparaisons est : $(2 \times 2 + 4 \times 3) / 6 \approx 2{,}67$.

Pour tout algorithme de tri comparatif, la première comparaison faite consiste à prendre deux des données a et b de la liste et à poser la question : « a-t-on $a < b$? », ce qui divise les $n!$ permutations envisageables en deux parties égales ; les comparaisons suivantes ne garderont pas une telle symétrie : on aura de la chance quand il y aura moins de la moitié des permutations restant possibles qui correspondent à la réponse obtenue et de la malchance sinon. Si on s'intéresse à la complexité dans le pire des cas d'un algorithme, il faut supposer que la permutation considérée correspond toujours à un cas où on n'a jamais de chance, c'est-à-dire où chaque comparaison élimine au plus la moitié des permutations encore envisageables. Dans ce cas, après k comparaisons, il restera au moins $n! / 2^k$ permutations envisageables ; il faut avoir $n! / 2^k < 2$ pour avoir une permutation unique ; donc on aura effectué au moins de l'ordre de $\log_2(n!)$ comparaisons. Comme $\log_2(n!)$ est équivalent à $n\log_2(n)$, on a :

Résultat : tout algorithme comparatif fait dans le pire des cas au moins de l'ordre de $n\ln(n)$ comparaisons.

En ce qui concerne le nombre moyen de comparaisons, nous admettons que la hauteur moyenne des feuilles d'un arbre binaire ayant p feuilles est au moins égale à $\log_2(p)$. Le nombre moyen de comparaisons d'un algorithme de tri est la hauteur moyenne des feuilles de l'arbre tracé de la même façon que sur notre exemple. Comme $\log_2(n!)$ est équivalent à $n\log_2(n)$, on a :

Résultat : tout algorithme comparatif fait en moyenne au moins de l'ordre de $n\ln(n)$ comparaisons.

Nous allons voir plus loin le *tri sélection* et le *tri insertion* de complexité en $O(n^2)$, qui seront simples à écrire mais peu efficaces, ; en effet, nous verrons aussi le *tri rapide* et le *tri par arbre binaire de recherche* qui sont en moyenne en $O(n\ln n)$, mais dans le pire des cas en $O(n^2)$; nous verrons enfin *le tri tas* qui est dans le pire des cas en $O(n\ln n)$.

III.4. Tri sélection

Le tri sélection consiste à chercher la plus petite donnée de la liste, à mettre celle-ci en première position, puis à chercher la plus petite donnée parmi les données autres que la première, la mettre en deuxième position, et ainsi de suite.

Dans le pseudo-code donné ci-dessous, on suppose qu'on dispose d'une fonction *échanger* telle que, si T est un tableau et i et j deux indices de ce tableau, *échanger*(T, i, j) échange les données du tableau T qui se trouvent aux indices i et j.

Tri sélection

Les données sont rangées dans un tableau *T* entre les indices 1 et *n*.

On utilise trois variables entières notées *i*, *j* et *indicePetit*, et une variable *min* du même type que les données de la liste. Pour chaque itération de la boucle portant sur *i*, on cherche le plus petit élément de la liste qui se trouve entre les indices *i* et *n*.

- pour *i* qui varie de 1 à $n - 1$;
  - *indicePetit* ← *i* ;
  - *min* ← *T*[*i*] ;
  - pour *j* qui varie de $i + 1$ à *n*, faire
    - si *T*[*j*] < *min*
      - *indicePetit* ← *j* ;
      - *min* ← *T*[*j*] ;
  - *échanger*(*T*, *i*, *indicePetit*).

La complexité de ce tri est aussi bien dans le pire des cas que dans le meilleur des cas (et donc qu'en moyenne) en $O(n^2)$. Ce n'est donc pas un tri très efficace, mais en revanche c'est un tri très simple.

III.5. Tri insertion

Le principe du tri par insertion est le suivant. On souhaite trier une liste de *n* données ; on procède par étape ; à la i^e étape (*i* variant de 1 à $n - 1$), on suppose que les *i* premières données sont déjà triées ; on considère alors la $(i + 1)^e$ donnée que l'on appelle *clé* ; on la compare successivement aux données précédentes, en commençant par la i^e puis en remontant dans la liste jusqu'à trouver la bonne place de *clé* (c'est-à-dire entre deux données successives, l'une étant plus petite et l'autre plus grande que *clé*), ou bien en tout premier si *clé* est plus petite que les *i* premières données ; au fur et à mesure, on décale « d'une case vers la droite » les données plus grandes que *clé* de façon à anticiper la place de ces données après insertion de *clé* ; on met *clé* à la bonne place ; à l'issue de cette étape, les $i + 1$ premières données sont donc triées.

Tri insertion

Les données sont rangées dans un tableau *T* entre les indices 1 et *n*.

On utilise deux variables entières notées *i*, *j* et une variable *clé* du même type que les données de la liste.

- pour *i* qui varie de 2 à *n*
  - *j* ← *i* ;
  - *clé* ← *T*[*j*] ;
  - tant que $j \geq 2$ et *T*[$j - 1$] > *clé*, faire
    - *T*[*j*] ← *T*[$j - 1$] ;
    - *j* ← $j - 1$;
  - *T*[*j*] ← *clé*.

Ce tri est en $O(n^2)$ dans le pire des cas (cas où la liste est triée dans l'ordre inverse) et en $O(n)$ dans le meilleur des cas (cas où la liste serait déjà triée avant application de l'algorithme) ; si on considère que toutes les permutations des données sont équiprobables, on vérifie facilement que le tri est en moyenne en $O(n^2)$. C'est donc encore un tri peu efficace mais qui a aussi le mérite de la simplicité.

Le tri insertion a une complexité en moyenne du même ordre que celle du tri sélection ; néanmoins, si les données à trier sont souvent presque triées (dans le sens souhaité), il vaudra mieux choisir le tri insertion.

III.6. Tri rapide

Le principe du tri rapide est le suivant. On utilise une fonction nommée *partition* qui s'applique à une liste ; cette fonction extrait une donnée, nommée *clé*, de la liste, par exemple la première (c'est ce que nous choisirons plus bas) puis elle réorganise la liste de sorte qu'on trouve d'abord les données ne dépassant pas la valeur de *clé* (on appellera sous-liste gauche cette partie de la liste), puis la donnée *clé*, puis les données supérieures à *clé* (on appellera sous-liste droite cette partie de la liste). Le tri rapide peut être défini récursivement. Pour appliquer le tri rapide à une liste ayant au moins deux données, on commence par appliquer la fonction *partition*, puis on applique le tri rapide à la sous-liste gauche, puis on applique le tri rapide à la sous-liste droite.

La fonction *partition* peut être implémentée de différentes façons ; il importe simplement que sa complexité pour une liste de longueur n soit en $O(n)$, pour que le tri rapide ait une bonne complexité (voir plus bas). Nous supposons que les données sont dans un tableau T et que ce tableau contient une case supplémentaire à la droite des données, qui contient une donnée strictement supérieure à toutes les données de la liste ; cette condition est nécessaire pour que, dans le pseudo-code donné ci-dessous, la variable i ne risque pas de croître « indéfiniment » ; il ne faudra pas l'oublier au moment d'une programmation (ou bien il faudra modifier légèrement le pseudo-code ci-dessous pour éviter le débordement sur la droite du tableau). Dans le pseudo-code suivant, *partition* renvoie l'indice du tableau où *clé* est rangée. La procédure *échanger* est définie comme plus haut.

Fonction *partition*(*g*, *d*)

On s'intéresse aux données qui sont dans le tableau *T* entre les indices *g* et *d* inclus.

On utilise deux variables entières notées *i*, *j* et une variable *clé* du même type que les données de la liste.

- *clé* ← *T*[*g*] ;
- *i* ← *g* + 1 ;
- *j* ← *d* ;
- tant que *i* ≤ *j*, faire
  - tant que *T*[*i*] ≤ *clé*, faire *i* ← *i* + 1 ;
  - tant que *T*[*j*] > *clé*, faire *j* ← *j* – 1 ;
  - si *i* < *j*, *alors*
    - *échanger*(*T*, *i*, *j*) ;
    - *i* ← *i* + 1 ;
    - *j* ← *j* – 1 ;
- *échanger*(*T*, *g*, *j*) ;
- renvoyer *j*.

Prenons un exemple. Les données sont les valeurs qui figurent dans la seconde ligne du tableau *T* suivant, la première précisant les indices des cases de *T* :

1	2	3	4	5	6	7	8	9	10	11	12
9	4	18	11	15	5	17	1	10	7	12	6

On suppose qu'on applique la fonction partition avec *g* = 1 et *d* = 12. La variable *clé* vaut T[1] = 9. Au premier passage dans la boucle externe, *i* s'arrête à 3, *j* s'arrête à 12. On procède à l'échange et on obtient :

1	2	3	4	5	6	7	8	9	10	11	12
9	4	6	11	15	5	17	1	10	7	12	18

Après quoi *i* passe 4 et *j* à 11.

Au deuxième passage dans la boucle, *i* s'arrête à 4 et *j* s'arrête à 10. On procède à l'échange et on obtient :

1	2	3	4	5	6	7	8	9	10	11	12
9	4	6	7	15	5	17	1	10	11	12	18

avec *i* = 5 et *j* = 9.

Au troisième passage dans la boucle, *i* s'arrête à 5 et *j* s'arrête à 8. On procède à l'échange et on obtient :

1	2	3	4	5	6	7	8	9	10	11	12
9	4	6	7	1	5	17	15	10	11	12	18

avec *i* = 6 et *j* = 7.

Au quatrième passage dans la boucle, i s'arrête à 7 et j s'arrête à 6. Le test « $i < j$? » est négatif car on a $i \geq j$. On sort de la boucle externe « tant que $i \leq j$ » car on a $i > j$; on échange $T[1]$ avec $T[j]$, *i.e.* $T[6]$; on obtient :

1	2	3	4	5	6	7	8	9	10	11	12
5	4	6	7	1	9	17	15	10	11	12	18

On retourne l'indice 6. On remarque que la donnée de valeur 9 est à sa place définitive, qu'avant elle se trouvent des données inférieures à 9 et, après, des données supérieures à 9.

On peut maintenant écrire le pseudo-code du tri rapide pour un tableau compris entre les indices g et d. On appellera *tri_rapide*(1, n) pour trier des données se trouvant entre les indices 1 et n d'un tableau.

Procédure tri_rapide(*g*, *d*)

On utilise une variable entière notée j.

- si $g < d$, alors
  - $j \leftarrow$ *partition*(g, d) ;
  - *tri_rapide*(g, $j - 1$) ;
  - *tri_rapide*($j + 1$, d).

Sur notre exemple, il reste à appliquer le tri rapide entre les indices 1 et 5 puis entre les indices 7 et 12.

L'application de *tri_rapide*(1, 5) appelle *partition*(1, 5) qui conduit au tableau :

1	2	3	4	5	6	7	8	9	10	11	12
1	4	5	7	6	9	17	15	10	11	12	18

et renvoie la valeur 3. On applique alors *tri_rapide*(1, 2), qui appelle *partition*(1, 2) qui ne change pas le tableau et retourne la valeur 1, puis *tri_rapide* (1, 0) qui ne fait rien, puis *tri_rapide* (4, 5) qui appelle *partition*(4, 5) qui conduit au tableau :

1	2	3	4	5	6	7	8	9	10	11	12
1	4	5	6	7	9	17	15	10	11	12	18

et retourne l'indice 4. Puis, *tri_rapide*(4, 3) ne fait rien ainsi que *tri_rapide* (5, 5). Le tri rapide entre les indices 1 et 5 est terminé.

L'application de *tri_rapide*(7, 12) fait appel à *partition*(7, 12) qui conduit au tableau :

1	2	3	4	5	6	7	8	9	10	11	12
1	4	5	6	7	9	12	15	10	11	17	18

et retourne l'indice 11. Alors, *tri_rapide*(7, 10) appelle *partition*(7, 10) qui conduit à :

1	2	3	4	5	6	7	8	9	10	11	12
1	4	5	6	7	9	10	11	12	15	17	18

et retourne l'indice 9. Puis, *tri_rapide*(7, 8) fait appel à *partition*(7, 8) qui ne change pas le tableau et retourne l'indice 7, puis *tri_rapide*(7, 6) ainsi que *tri_rapide*(8, 8) ne font rien. On « remonte » avec *tri_rapide*(12, 12) qui ne fait rien. L'algorithme est terminé.

On remarque que la complexité de la fonction *partition* est bien linéaire en la longueur de la liste considérée.

Le « gros » du travail consiste en l'application des fonctions *partition*.

Le pire des cas pour cet algorithme est celui où la clé vient toujours à une extrémité du sous-tableau allant de l'indice *g* à l'indice *d* pour chaque appel à la fonction *partition*. Ce sera par exemple le cas si on trie un tableau déjà trié. Il sera alors fait appel à la fonction *partition* pour des listes de longueur *n*, puis $n - 1$, puis $n - 2$, …, puis 2 ; la somme des complexités de ces fonctions donne un algorithme en $O(n^2)$.

Le meilleur des cas est celui où la liste est toujours partagée en deux sous-listes de même longueur. On aura en tout :

- un appel à la fonction *partition* sur une liste de longueur *n*,
- deux appels à la fonction *partition* sur des listes de longueurs environ $n/2$,
- quatre appels à la fonction *partition* sur des listes de longueurs environ $n/4$,
- …

D'où une complexité de l'ordre de *n* pour la liste de longueur *n*, une complexité totale de l'ordre de *n* pour les deux sous-listes de longueur $n/2$, une complexité totale de l'ordre de *n* pour les quatre sous-listes de longueur $n/4$, … Comme la longueur est divisée par deux quand on passe de *n* à $n/2$, puis de $n/2$ à $n/4$, on a en tout une complexité de l'ordre de $n \times \ln n$.

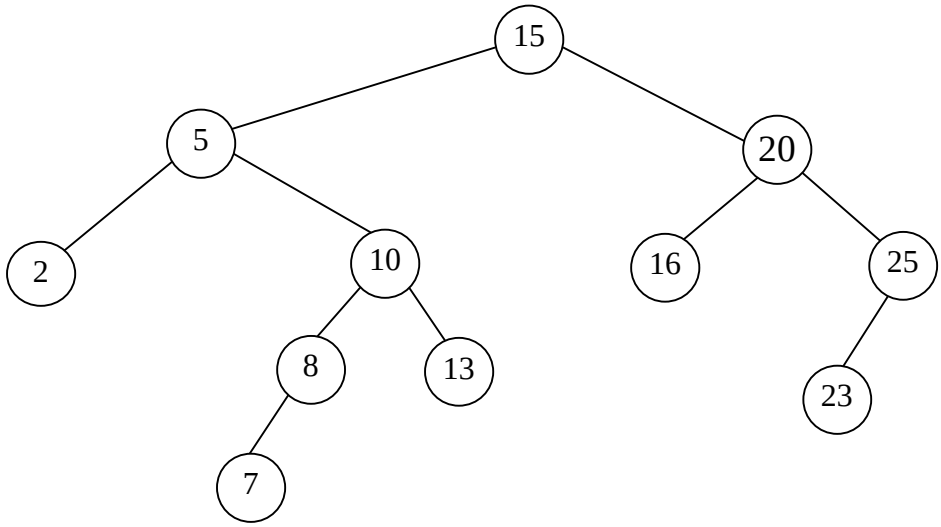
On peut montrer sans trop de difficulté, par une formule de récurrence, que la complexité en moyenne est aussi de l'ordre de $n \times \ln n$; nous ne prouvons pas ici ce résultat.

Remarquons que pour éviter d'avoir une complexité en n^2 lorsqu'on trie une liste déjà triée, on prend souvent, au tout début de la fonction *partition*, une donnée au hasard dans la liste traitée et on l'échange avec la donnée qui se trouve en première position, puis on continue de la façon indiquée précédemment.

III.7. Arbre binaire de recherche

Un *arbre binaire de recherche* est un arbre dont les nœuds appartiennent à un ensemble totalement ordonné et tel que, pour **tout** nœud interne *a*, les données contenues dans le sous-arbre gauche de *a* sont inférieures ou égales à la donnée contenue dans le nœud *a*, qui est elle-même inférieure ou égale aux données contenues dans le sous-arbre droit de *a*.

Exemple



Par définition d'un arbre binaire de recherche, un parcours en ordre symétrique de l'arbre donne la liste triée des données que contient l'arbre.

Ci-dessous, on supposera que la racine de l'arbre ainsi que les fils gauche et droit d'un nœud sont des adresses de nœuds. Si on a l'adresse *adr* d'un nœud, *adr.donnée* sera la donnée contenue par le nœud (cela serait noté *adr→donnée* s'il s'agissait d'un codage en langage C), *adr.fils_gauche* sera l'adresse du fils gauche et *adr.fils_droit* sera l'adresse du fils droit.

Pour insérer une donnée nommée *clé* dans un arbre binaire de recherche, on compare *clé* à la donnée contenue dans la racine ; si *clé* est plus petite, on l'insère dans le sous-arbre gauche ; si elle est plus grande, on l'insère dans le sous-arbre droit. Ceci est décrit plus précisément par la fonction récursive *insérer* dont on donne ci-dessous le pseudo-code. Cette fonction retourne l'adresse de la racine de l'arbre obtenu après insertion dans un arbre binaire de recherche dont la racine a pour adresse *racine* d'un nouveau nœud contenant la donnée *clé*.

Fonction *insérer*(*racine*, *clé*)

- Si *racine* vaut *adresse_nulle*, faire
  - créer un nouveau nœud nommé d'adresse nommée *nouveau* ;
  - *nouveau.donnée* ← *clé* ;
  - *nouveau.fils_gauche* ← *adresse_nulle* ;
  - *nouveau.fils_droit* ← *adresse_nulle* ;
  - retourner *nouveau* ;

  sinon
  - si *clé* ≤ *racine.donnée*, faire
    - *racine.fils_gauche* ← *insérer*(*racine.fils_gauche*, *clé*) ;

    sinon faire
    - *racine.fils_droit* ← *insérer*(*racine.fils_droit*, *clé*). ;
  - retourner *racine*.

On pourrait écrire d'une manière analogue un algorithme de *recherche* qui retourne vrai ou faux selon qu'une clé reçue en paramètre appartient ou non à l'arbre binaire de recherche.

La complexité d'une insertion ou d'une recherche dans un arbre binaire de recherche est :

- dans le pire des cas égale à la profondeur de l'arbre binaire dans lequel se fait l'insertion ;

- en moyenne égale à la profondeur moyenne des feuilles de l'arbre binaire.

Considérons un arbre binaire de recherche obtenu par insertions successives de n données formant une permutation quelconque, toutes les permutations étant équiprobables. On pourrait montrer qu'en moyenne la hauteur ainsi que la profondeur moyenne des feuilles de cet arbre sont de l'ordre de ln(n) ; le pire néanmoins correspond à un arbre dégénéré, où on obtient une hauteur de l'arbre et une profondeur moyenne des feuilles de l'ordre de n.

La recherche d'une donnée dans un arbre binaire de recherche contenant n données est donc au pire en O(n) et en moyenne (sur les arbres binaires possibles et sur les données de ces arbres) en O(ln n).

Si on veut trier n des données en utilisant un arbre binaire de recherche, il faut :

- construire l'arbre binaire de recherche en partant de l'arbre vide et en insérant les données les unes après les autres ; la complexité pour l'ensemble des insertions est de l'ordre de la somme des profondeurs des nœuds de l'arbre obtenu ; celle-ci a un ordre de grandeur compris entre $n \ln n$ et n^2, et est en moyenne sur les jeux de données possibles de l'ordre de $n \ln n$;
- on lit en ordre symétrique l'arbre binaire obtenu, opération dont la complexité est du même ordre que le nombre de nœuds de l'arbre, et donc inférieure à celle de la construction de l'arbre.

Le tri par arbre binaire de recherche est donc au pire en O(n^2) et en moyenne en O($n \ln n$).

La somme des profondeurs des nœuds d'un arbre binaire de recherche équilibré est de l'ordre de $n \ln n$; on peut écrire un algorithme d'insertion qui, partant d'un arbre binaire de recherche équilibré, fait l'insertion comme indiqué ci-dessus suivi d'une transformation à temps constant qui transforme l'arbre binaire de recherche obtenu en un arbre binaire de recherche équilibré. Nous ne verrons pas ici cette technique. Néanmoins, cette amélioration permet d'obtenir un tri en O($n \ln n$) dans le pire des cas et de construire un arbre binaire dans lequel la recherche se fait certainement en temps logarithmique.

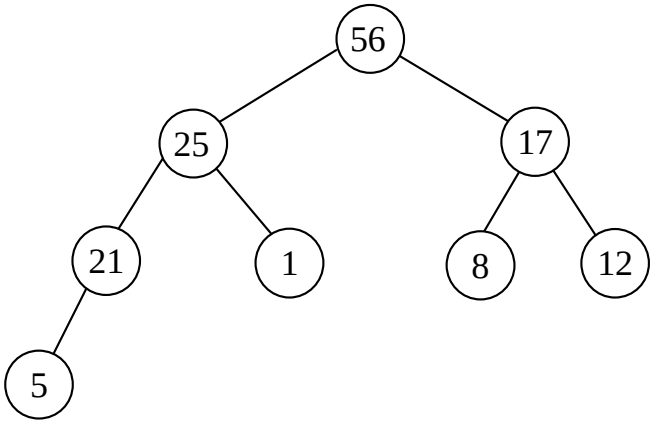
Exercice : comment peut-on supprimer une donnée dans un arbre binaire de recherche ? On s'aidera d'exemples pour concevoir le principe d'un algorithme.

III.8. Structure de tas et tri tas

III.8.1. Définition

On appelle *tas* un arbre binaire parfait, c'est-à-dire où tous les niveaux sont remplis sauf éventuellement le dernier, celui-ci étant rempli de la gauche vers la droite, et dans les sommets duquel figurent des éléments d'un ensemble totalement ordonné, l'élément stocké en un nœud quelconque étant plus grand que les éléments stockés dans ses deux nœuds fils s'il a deux fils, dans son fils s'il a un seul fils. Il n'y a pas d'autre relation d'ordre : un « neveu » peut être plus grand que son « oncle ».

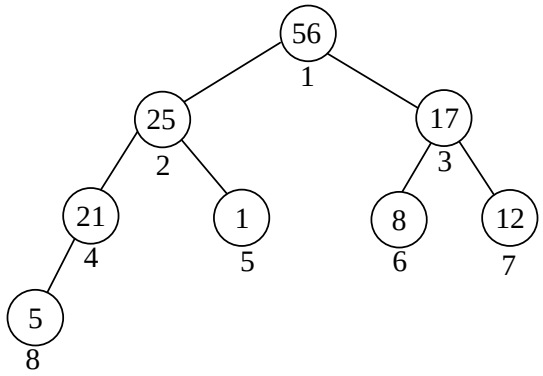
Exemple : ci-dessous est représenté un tas.



III.8.2. Intérêt de la structure d'arbre parfait

Il existe une numérotation canonique des nœuds d'un tas. Si on numérote les nœuds ligne par ligne et de gauche à droite, en donnant le numéro 1 à la racine, on vérifie par récurrence que les numéros des fils du nœud de numéro i, s'ils existent, sont les nœuds de numéros $2i$ et $2i + 1$, de sorte que le numéro du père d'un nœud de numéro i ($i \geq 2$) est le nœud de numéro égal à la partie entière du résultat de la division de i par 2. Si donc on dispose d'un tableau de m cases, indicées de 1 à m, pour stocker un tas ayant au plus m nœuds, on peut y stocker l'élément contenu dans le nœud de numéro i dans la case d'indice i. Les fils du nœud de numéro i, s'ils existent, sont stockés dans les cases d'indices $2i$ et $2i + 1$ et son père, s'il existe (c'est-à-dire si le nœud considéré n'est pas la racine), se trouve dans la case d'indice $\lfloor i/2 \rfloor$. C'est ce codage avec un tableau qui sera en général retenu dans le traitement informatique d'un tas. Nous appellerons nœud d'indice i le nœud qui est numéroté par i, et donc qui est rangé dans la i^e case du tableau.

On donne ci-dessous la représentation du tas considéré précédemment en exemple par un tableau, les nœuds du tas étant numérotés comme on l'a indiqué plus haut.



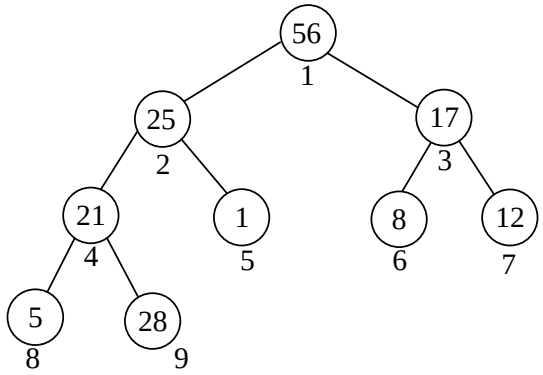
1	2	3	4	5	6	7	8
56	25	17	21	1	8	12	5

Nous allons tout d'abord voir comment structurer une liste donnée de valeurs en tas. À partir d'un tas vide, on insère les éléments les uns après les autres en veillant à conserver la structure de tas. Pour respecter la structure d'arbre binaire parfait on insère le nouvel élément à la première place disponible dans la dernière rangée si celle-ci n'est pas encore pleine ; si la dernière rangée est pleine, on en crée une nouvelle et on insère le nouvel élément à la première place de la nouvelle rangée. Dans les deux cas, on a inséré le nouvel élément dans la

première case non utilisée du tableau. Ensuite, on « fait remonter cet élément » dans l'arbre, en l'échangeant avec son père, tant qu'il est plus grand que son père.

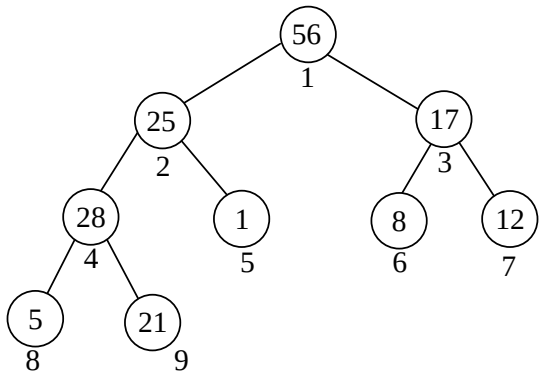
III.8.3. Exemple d'insertion d'un nouvel élément dans un tas

Imaginons que, partant du tas représenté ci-dessus, on veuille maintenant introduire la valeur 28. On place la valeur 28 à droite de la dernière feuille introduite, c'est-à-dire dans la case d'indice 9 du tableau.



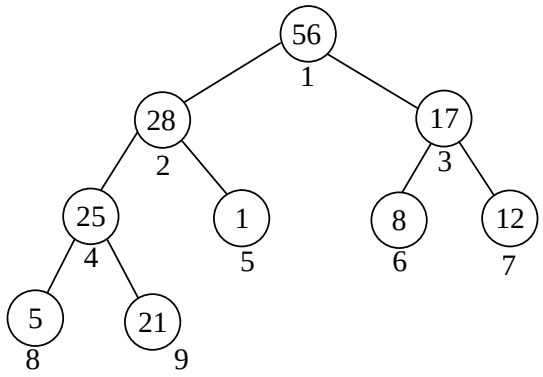
1	2	3	4	5	6	7	8	9
56	25	17	21	1	8	12	5	28

On compare 28, la nouvelle donnée insérée, avec la donnée contenue dans le nœud père, autrement dit on compare la donnée de la case d'indice 9 du tableau avec la donnée de la case d'indice $\lfloor 9/2 \rfloor = 4$. Puisque 28 est plus grand que 21, on échange le 21 et le 28. En observant cet échange, on remarque qu'en toute généralité la seule relation définissant un tas qui peut ne pas être vérifiée est mainenant celle qui existe entre la donnée (28) contenue par le nœud d'indice 4 et la donnée (25) contenue par le nœud père de celui-ci, d'indice $\lfloor 4/2 \rfloor = 2$.



1	2	3	4	5	6	7	8	9
56	25	17	28	1	8	12	5	21

On compare la donnée 28 du nœud d'indice 4 avec la donnée 25 contenue dans son nœud père, d'indice 2. Puisque 28 est plus grand que 25, on échange le 25 et le 28. Comme précédemment, on remarque que la seule relation définissant un tas qui peut ne pas être vérifiée est mainenant celle qui existe entre la donnée (28) contenue par le nœud d'indice 2 et la donnée (56) contenue par le nœud d'indice $\lfloor 2/2 \rfloor = 1$.



1	2	3	4	5	6	7	8	9
56	28	17	25	1	8	12	5	21

On compare la donnée 28 de la case d'indice 2 avec la donnée contenue dans le nœud père d'indice 1. Puisque 28 est plus petit que 56, l'insertion est terminée.

On a bien ainsi restaurée la structure de tas.

III.8.4. Pseudo-code de l'insertion d'une donnée dans un tas

On suppose qu'on dispose d'un tas de $p - 1$ nœuds et qu'on ajoute un nœud d'indice p ; il s'agit de décrire en pseudo-code ce qui a été développé ci-dessus, pour obtenir un tas de p nœuds ; nous nommons *montée* cette procédure. On utilise une variable nommée *clé* qui mémorise la donnée à insérer de façon à éviter des échanges inutiles. Le tableau représentant le tas est noté *T*. On peut remarquer que cette procédure est très proche de l'insertion séquentielle d'une nouvelle donnée dans un tableau trié ; on se déplace de père en père au lieu de se déplacer d'une case vers celle qui lui est adjacente à gauche.

Procédure *montée*(*p*)

On utilise un indice entier *i* et une variable *clé* du même type que les données du tas.

- $i \leftarrow p$;
- $clé \leftarrow T[p]$;
- tant que $i \geq 2$ et que $clé > T[\lfloor i/2 \rfloor]$, faire
  - $T[i] \leftarrow T[\lfloor i/2 \rfloor]$;
  - $i \leftarrow \lfloor i/2 \rfloor$;
- $T[i] \leftarrow clé$;

On peut utiliser cette même procédure si, dans un tas ayant n données, la donnée contenue par le nœud d'indice p ($p \leq n$) a été augmentée. On verra plus loin comment traiter le cas de la diminution d'une donnée du tas.

III.8.5. Complexité d'une insertion dans un tas

Lorsqu'on insère une p^{e} donnée dans un tas, celle-ci est insérée initialement en dernier dans le tableau, c'est-à-dire dans le niveau de profondeur $h = \lfloor \log_2 p \rfloor$; on fait alors au plus h comparaisons et échanges d'éléments.

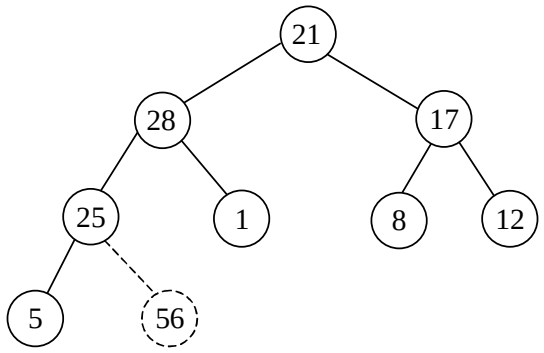
III.8.6. Principe du tri tas

Dans un tas, le plus grand élément est à la racine. Nous allons voir comment tirer parti de cette indication pour trier des données de façon très efficace.

On échange la donnée contenue par la racine avec la donnée contenue dans la dernière feuille du tas, c'est-à-dire avec la donnée contenue dans la dernière case du tableau représentant le tas. La donnée contenue dans la racine avant l'échange étant la plus grande, après l'échange la donnée contenue dans la dernière case du tableau est la plus grande et est donc à sa place dans un tri par ordre croissant ; on ne va plus y toucher. On va alors reconstruire un tas avec les données d'indices compris entre 1 et $n - 1$ (en notant n le nombre initial de données) ; la seule condition définissant un tas qui peut être violée est la supériorité de la donnée de la racine par rapport à ses fils. Pour cela, dans une démarche symétrique de celle de la construction du tas, on « fait descendre » la donnée mise à la racine : tant qu'on n'a pas atteint une feuille du tas et que la donnée qui descend est plus petite que la plus grande des données de ses deux fils, on l'échange avec cette plus grande donnée. Nous illustrons cette suppression de la racine dans le paragraphe ci-dessous.

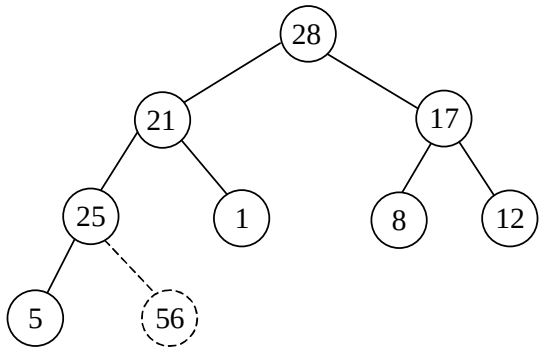
III.8.7. Exemple de suppression de la racine

On échange la donnée de la racine avec la dernière donnée du tas.



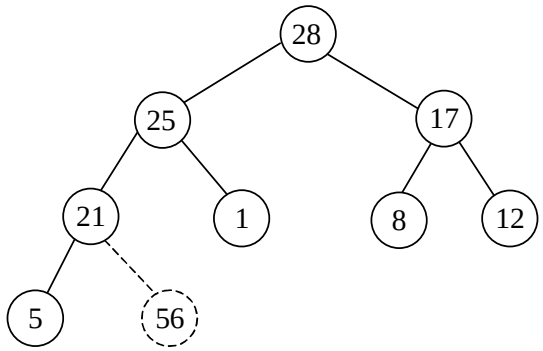
1	2	3	4	5	6	7	8	9
21	28	17	25	1	8	12	5	56

On cherche à reconstituer un tas sur les 8 premières cases du tableau. Pour cela, on cherche la plus grande des deux données contenues dans les fils de la racine, c'est-à-dire les nœuds d'indice $2 \times 1 = 2$ et $2 \times 1 + 1 = 3$; la plus grande de ces données vaut 28 ; on échange donc le contenu de la case 1 avec le contenu de la case 2.



1	2	3	4	5	6	7	8	9
28	21	17	25	1	8	12	5	56

On cherche la plus grande des données entres les données contenues par les nœuds d'indice $2 \times 2 = 4$ et $2 \times 2 + 1 = 5$; la plus grande de ces données vaut 25 et est plus grande que 21 ; on échange donc le contenu de la case 2 avec le contenu de la case 4.



1	2	3	4	5	6	7	8	9
28	25	17	21	1	8	12	5	56

Comme on ne considère qu'un tas sur les 8 premières cases, le nœud d'indice 4 a un seul fils dans le tas, qui contient la donnée 5. Comme 21 est plus grand que 5, la descente de la donnée insérée à la racine est terminée.

III.8.8. Pseudo-code de la descente d'une donnée

Nous allons écrire une procédure un peu plus générale que ce qui a été utilisé plus haut. On suppose qu'on dispose d'un tas de *p* données et que la donnée qui se trouve dans le nœud d'indice *q* est diminuée. On souhaite alors réarranger les données pour récupérer la structure de tas sur les *p* données ; il faut pour cela faire « descendre » la donnée diminuée jusqu'à ce qu'elle soit à la bonne place. Le principe de la procédure correspond à ce qui a été fait plus haut. On utilise une variable nommée *clé* qui mémorise la donnée qui descend, de façon à éviter des échanges inutiles. Le tableau représentant le tas est noté *T*. Cette procédure est très proche de l'insertion séquentielle d'une nouvelle donnée dans un tableau trié. On se déplace de fils en « plus grand fils » au lieu de se déplacer d'une case vers celle qui lui est adjacente à gauche.

Procédure *descente*(*q*, *p*)

On utilise deux indices entiers *i* et *indice_grand* et deux variables, *clé* et *grand*, du même type que les données du tas, et une variable booléenne *trouvé*.

- *trouvé* ← faux ;
- *i* ← *q* ;
- *clé* ← *T*[*q*] ;
- tant que (non *trouvé* et 2*i* ≤ *p*) faire
  - si 2*i* = *p*, alors *indice_grand* ← *p* ;
    sinon
    - si *T*[2*i*] ≥ *T*[2*i* + 1], alors *indice_grand* ←2*i* ;
      sinon *indice_grand* ←2*i* + 1 ;
  - si *clé* < *T*[*indice_grand*], alors
    - *T*[*i*] ← *T*[*indice_grand*] ;
    - *i* ← *indice_grand* ;

    sinon *trouvé* ← vrai ; (* instruction qui sert à sortir de la boucle *)
- *T*[*i*] ← *clé*.

III.8.9. Complexité de la descente de la donnée de la racine

Si on veut utiliser la descente d'une donnée à partir de la racine, la complexité est au pire de l'ordre de la hauteur de l'arbre binaire, puisque, pour chaque niveau de l'arbre, on fait un nombre d'opérations majoré par une constante (au plus deux comparaisons et quelques affectations) ; la descente de la donnée de la racine est donc en O(ln *n*) pour un tas de *n* données.

III.8.10. Pseudo-code du tri tas

On suppose qu'on dispose d'un tableau *T* contenant *n* données dans le désordre, dans les cases d'indices compris entre 1 et *n*, et qu'on veut trier ces données. On peut utiliser la procédure *tri_tas* décrite ci-dessous en pseudo-code.

Procédure *tri_tas*

On utilise un indice entier *p*.

- Pour *p* qui varie de 2 à *n*, faire *montée*(*p*) ;
- Pour *p* qui varie de *n* à 2, faire
  - *échanger*(*T*, 1, *p*) ;
  - *descente*(1, *p* – 1).

III.8.11. Complexité du tri tas

La complexité se calcule directement à partir de celle des procédures *montée* et *descente* ; elle est en $O\left(\sum_{p=1}^{n} \ln p\right)$, c'est-à-dire en $O(n \ln n)$. On constate donc que ce tri a la complexité optimum d'un tri comparatif.

III.8.12. Exercice

Si vous connaissez ou lorsque vous connaîtrez l'algorithme de Dijkstra, vous pourrez utiliser la structure de tas pour modifier l'algorithme afin d'avoir une complexité en $m \ln n$, en notant n le nombre de sommets du graphe et m son nombre d'arcs.

IV. Le hachage

IV.1. Principe général

Le hachage peut être utilisé par un analyseur syntaxique pour reconnaître les mots réservés du langage de programmation dans le fichier source d'un programme. Si on ne veut pas faire appel à des structures dynamiques, on doit disposer d'un tableau de taille *m*, *m* étant plus grand que le nombre de mots réservés, tableau que nous appellerons *table de hachage*. On définit par ailleurs sur l'ensemble des chaînes de caractères une fonction *f* à valeurs dans l'ensemble des entiers {0, ..., *m* – 1} dite *fonction de hachage* ou *de codage*. On indique plus bas des exemples de telles fonctions. On code les mots réservés à l'aide de la fonction *f*. Si un mot a pour code *k*, on le range en position *k* dans la table. Hélas, même si la table est nettement plus grande que le nombre de mots réservés, on peut avoir à faire face à des *collisions*. On dit qu'il y a *collision* si la case du tableau correspondant au code du mot qu'on veut ranger est déjà occupée. Selon la façon dont on traite les collisions, on obtient une méthode de hachage différente ; néanmoins, toutes les méthodes font en sorte qu'on ne déplace pas un mot une fois qu'il occupe une position donnée dans le tableau. Pour simplifier, supposons momentanément qu'on n'ait pas rencontré de collision et qu'on ait donc rangé dans la table tous les mots réservés aux positions attendues. Par la suite, lorsque l'analyseur rencontre un mot du programme, il le code ; si la case correspondante est vide ou contient un mot différent du mot considéré, on sait, en au plus une comparaison, qu'il ne s'agit pas d'un mot réservé ; dans le cas contraire, la réponse est positive.

Nous allons maintenant spécifier trois façons de gérer les collisions et, par suite, la recherche.

Mais illustrons tout d'abord, à l'aide d'un exemple, le fait que les collisions sont « presque » inévitables.

IV.2. Exemple pour illustrer le risque de collisions

Dans une classe de 23 élèves, il y a un peu plus d'une chance sur deux pour que deux élèves aient la même date anniversaire ! Ou encore, dit dans le langage de ce chapitre, si on dispose de 365 cases pour coder 23 mots, avec une fonction de codage qui distribue uniformément les mots (la probabilité qu'un mot ait *k* pour code est égale à 1/365 pour tout *k* compris entre 0 et 364), il y a un peu plus d'une chance sur deux pour qu'on rencontre le phénomène de collision.

Preuve

Nous allons calculer la probabilité de l'événement inverse : les 23 codes sont différents. Pour calculer cette probabilité, imaginons qu'on a rangé les mots dans une suite et qu'on les code dans l'ordre de cette suite. Il y a 365^{23} suites différentes, équiprobables (avec l'hypothèse « idéale » de l'uniformité de la répartition des codes). Parmi ces suites, il y en a $365(365 - 1)(365 - 2) \ldots (365 - 22)$ qui correspondent à des codages différents. La probabilité pour qu'il n'y ait pas de collision est le rapport de ces deux nombres :

$$R = \prod_{i=0}^{22} \frac{365 - i}{365} = \prod_{i=1}^{22} \left(1 - \frac{i}{365}\right).$$

D'où : $\ln R = \sum_{i=1}^{22} \ln\left(1 - \frac{i}{365}\right) \approx -\frac{1}{365} \sum_{i=1}^{22} i = -\frac{1}{365} \frac{22 \times 23}{2} \approx -0{,}6932$.

Or on a $\exp(-0{,}6932) \approx 0{,}4999$, ce qui donne une approximation de la probabilité qu'il n'y ait pas de collision. Celle qu'il y ait des collisions vaut donc un peu plus de 0,5.

IV.3. Exemples de fonctions de hachage

Les fonctions de hachage que nous donnons en exemple sont définies sur l'ensemble des chaînes de caractères. On peut aussi définir des fonctions de hachage sur toute sorte d'ensembles, par exemple sur l'ensemble des entiers.

Fonction h_1 : cette fonction ne convient que pour les chaînes écrites en minuscules avec les 26 caractères de l'alphabet. On code chaque lettre de l'alphabet par son rang dans l'alphabet, 'a' ayant ainsi le code 1. On associe à chaque mot tout d'abord la somme s des codes de ses lettres, puis le reste modulo m de s.

Exemple : si $m = 20$, pour la chaîne "lou", on obtient :
h_1("lou") = (12 + 15 + 21) modulo 20 = 8.

Les valeurs de cette fonction seront mal réparties sur l'ensemble $\{0, ..., m - 1\}$ si m est grand et que les chaînes sont courtes.

Fonction h_2 : on code chaque lettre de la chaîne par son code ASCII ; on multiplie ce code ASCII par la position de la lettre dans la chaîne considérée ; on additionne les valeurs obtenues pour chacune des lettres puis on prend le reste modulo m.

Exemple : si $m = 100$, pour la chaîne "lou", on obtient :
h_2("lou") = $(108 \times 1 + 111 \times 2 + 117 \times 3)$ modulo 100 = 81.

Ce code a l'avantage de tenir compte de la position des lettres dans le mot.

Fonction h_3 : on peut faire beaucoup plus sophistiqué ; la fonction h_3 utilise différents principes qui ne sont souvent que partiellement utilisés par d'autres fonctions de hachage. On choisit un entier t qui ne doit pas être trop grand de façon à éviter ci-dessous la manipulation de trop grands entiers (voir ci-dessous l'utilisation de t) ; on choisit un réel θ vérifiant $0 < \theta < 1$; un bon choix de θ peut être $\theta = \left(\sqrt{5} - 1\right)/2 \approx 0{,}6180339887$ ou bien $\theta = 1 - \left(\sqrt{5} - 1\right)/2 \approx 0{,}3819660113$. Si x est un nombre réel, on note ci-dessous x mod 1 la partie décimale de x, c'est-à-dire la valeur obtenue en remplaçant par 0 la partie qui se trouve « avant la virgule ».

Pour obtenir la valeur de $h_3(s)$, où s est une chaîne de caractères, on effectue les opérations suivantes :

1. on code chaque lettre de la chaîne *s* par son code ASCII ;
2. on écrit en binaires les valeurs obtenues (le chiffre de gauche est un 1) ;
3. on concatène les chaînes binaires de ces écritures ;
4. on complète éventuellement la chaîne par des 0 à droite pour avoir une chaîne dont la longueur soit multiple de *t* ;
5. on découpe la chaîne concaténée en tranches de longueur *t* ;
6. on effectue un « ou exclusif » bit à bit sur l'ensemble de ces morceaux de chaînes ;
7. on appelle *v* la valeur, écrite en décimale, du résultat du « ou exclusif » ;
8. on pose $h_3(s) = \lfloor ((v \times \theta) \bmod 1) \times m \rfloor$.

On remarque qu'à l'issue du point 4, deux chaînes différentes donneront des résultats différents.

Exemple : pour $t = 8$, $\theta = (\sqrt{5} - 1)/2$ et $m = 100$, pour la chaîne "lou", on obtient successivement:

1. '1' → 108 ; 'o' → 111 ; 'u' → 117 ;
2. 108 → 1101100 ; 111 → 1101111 ; 117 → 1110101 ;
3. 110110011011111110101
4. 11011001101111111010100 0
5. 11011001, 10111111, 10101000
6. 11001110
7. $v = 206$; $v \times \theta \approx 127{,}315$
8. h_3("lou") = 31.

IV.4. Le hachage linéaire

Si, lors de la construction de la table, on veut ranger un nouveau mot et que la case correspondant au code de ce mot est déjà occupée, on se déplace dans la table, à partir de cet indice « cycliquement vers la droite » jusqu'à ce qu'on trouve une place vide. On y place alors l'élément. Dire qu'on se déplace « cycliquement vers la droite » signifie qu'on se déplace vers la droite mais que lorsqu'on arrive à la position $m - 1$ et que celle-ci est occupée, on repart de la position 0.

On dit qu'on a une *collision primaire* lorsque deux mots ont le même code, et donc se voient initialement attribuer la même place dans le tableau. Remarquons qu'ici on peut rencontrer des *collisions secondaires*, c'est-à-dire le fait que deux mots se disputent une même case, sans avoir cependant le même code, par exemple parce que le premier à être entré dans la table avait dû être déplacé vers la droite, la place lui revenant étant déjà occupée.

Exemple : supposons que les chaînes "lou", "david", "sophie", "marie", "jean", "gaspard" doivent être entrées dans une table de hachage ayant $m = 7$ cases avec :

h("lou") = 5, h("david") = 4, h("sophie") = 2, h("marie") = 4, h("jean") = 0, h("gaspard") = 6.

Les chaînes "lou", "david" et "sophie" vont dans les cases d'indice correspondant à leurs codes ; on a une collision primaire pour la chaîne "marie" qui devrait aller dans la case d'indice 4 qui est occupée, on essaie alors pour "marie" la case d'indice 5, mais cette case est occupée par "lou"; on essaie enfin la case d'indice 6 qui est libre, on y met "marie" ; on place

"jean" sans problème ; lorsqu'on veut placer "gaspard", on a une collision secondaire sur la case 6 avec "marie" ; on essaie alors la case d'indice 0, qui est occupée par "jean" et on place "gaspard" dans la case d'indice 1. On a donc obtenu :

0	1	2	3	4	5	6
"jean"	"gaspard"	"sophie"		"david"	"lou"	"marie"

Comment s'effectue alors la recherche d'un élément dans la table ? On code l'élément, on se place dans la table à la position d'indice égal au code de cet élément, on compare l'élément avec celui qui est contenu à cette place, puis on se déplace cycliquement vers la droite dans la table, jusqu'à ce qu'on trouve l'élément cherché (ceci pouvant avoir lieu dès la première comparaison) ou que l'on trouve une place non occupée ou, si le tableau est complètement plein, jusqu'à ce qu'on soit revenu au point de départ (ces deux derniers cas sont les cas d'échec de la recherche).

IV.5. Le hachage avec chaînage interne

Comme dans le hachage linéaire, on place tous les éléments dans un tableau. Ici, l'idée est, d'une part, de prévoir dans le tableau, en plus des m cases, une zone de débordement dans laquelle on rangera les éléments dont la place légitime était occupée lorsqu'on a voulu les ranger et, d'autre part, de « chaîner » entre eux les éléments qui entrent en collision primaire. On note p la taille de la zone de débordement ; cette zone se situe entre les indices m et $m + p - 1$ du tableau. Lors de la construction de la table, on prévoit pour chaque indice deux champs : un champ pour l'élément à ranger et un champ pour indiquer éventuellement un indice du tableau ; cette seconde information servira à faire un chaînage interne des éléments dépendant d'une même case ; une valeur de –1 dans le second champ indiquera qu'il n'y a pas d'« élément suivant ». Supposons que l'élément en cours de rangement ait pour code k ;

- si la case d'indice k est libre, on met l'élément à la place k et la valeur –1 dans le second champ ;
- si la case d'indice k est occupée, on range alors l'élément à la première case libre **à partir de la droite** du tableau, case qui est dans la zone de débordement si celle-ci n'est pas déjà pleine et la valeur –1 dans le second champ de cette même case. On note i l'indice de la case où on vient de ranger le nouvel élément. Si le second champ de la case d'indice k vaut –1, on remplace cette valeur par la valeur i ; si ce second champ ne valait pas –1, c'est qu'il y a déjà eu une collision sur la case k ; on parcourt alors la chaîne interne issue de la case k en suivant les indices successifs donnés par les seconds champs jusqu'à rencontrer un second champ de valeur –1, valeur qu'on remplace alors par la valeur i : tous les éléments de code k sont chaînés dans le tableau à partir de la case d'indice k.

Si la zone de débordement est pleine, on utilise la table initiale comme zone de débordement en occupant la première place libre à partir de la droite du tableau. Ceci ralentit la fabrication initiale de la table, car cela introduit des collisions secondaires.

Exemple : on choisit $m = 7$ et $p = 2$; supposons que l'on ait à placer des éléments e_1, e_2, e_3, e_4, e_5, e_6 et e_7 avec $h(e_1) = 4$, $h(e_2) = 6$, $h(e_3) = 2$, $h(e_4) = 4$, $h(e_5) = 2$, $h(e_6) = 2$, $h(e_7) = 5$. On place d'abord e_1, e_2 et e_3 ; on obtient :

0	1	2	3	4	5	6	7	8
		e_3		e_1		e_2		
		–1		–1		–1		

La place de e_4 n'est pas libre ; on a une collision primaire et le tableau devient après insertion de e_4 :

0	1	2	3	4	5	6	7	8
		e_3		e_1		e_2		e_4
		–1		8		–1		–1

La place de e_5 n'est pas libre ; on a une collision primaire et le tableau devient après insertion de e_5 :

0	1	2	3	4	5	6	7	8
		e_3		e_1		e_2	e_5	e_4
		7		8		–1	–1	–1

La place de e_6 n'est pas libre ; on a une collision primaire, la zone de débordement est pleine et la première place libre en partant de la droite a pour indice 5 ; le tableau devient après insertion de e_6 :

0	1	2	3	4	5	6	7	8
		e_3		e_1	e_6	e_2	e_5	e_4
		7		8	–1	–1	5	–1

Les éléments de code 2 sont chaînés et occupent successivement les places d'indices 2, 7 et 5.

La place de e_7, d'indice 5, n'est pas libre et elle est occupée par un élément dont le code ne vaut pas 5 ; on a une collision secondaire ; le tableau devient après insertion de e_7 :

0	1	2	3	4	5	6	7	8
		e_3	e_7	e_1	e_6	e_2	e_5	e_4
		7	–1	8	3	–1	5	–1

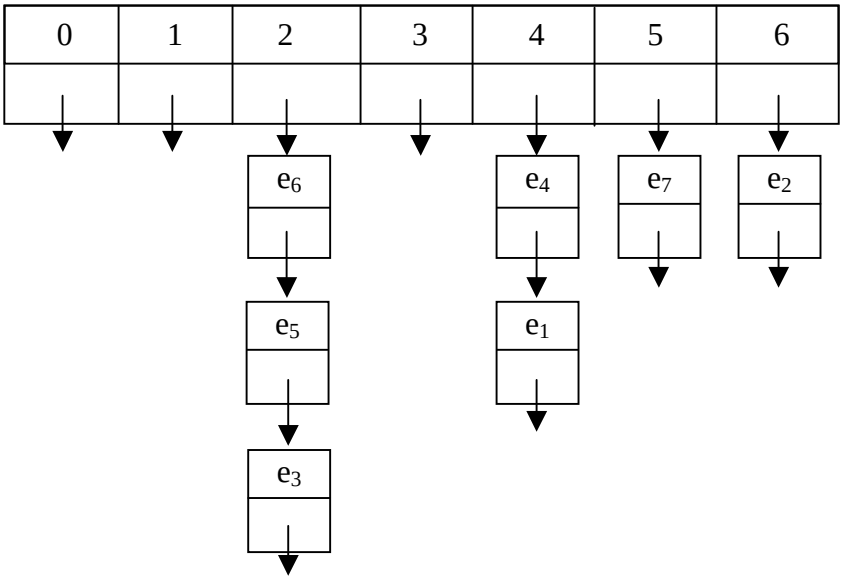
La recherche d'un élément *x* se déroule comme suit dans le cas du hachage avec chaînage interne. On code *x* ; appelons *k* le code de *x*. Si la case d'indice *k* ne contient pas d'élément, on conclut à l'échec ; sinon on compare successivement *x* à tous les éléments qui sont chaînés à partir de la case d'indice *k* jusqu'à avoir trouvé l'élément (situation de succès de la recherche) ou bien avoir atteint la fin de cette chaîne (situation de l'échec de la recherche).

IV.6. Le hachage avec chaînage externe

Lorsqu'on peut utiliser l'allocation dynamique de mémoire, on pourra préférer le chaînage externe pour gérer les collisions.

La méthode est simple : chaque case du tableau est le début d'une liste chaînée qui contient les éléments de la table dont le code est égal à l'indice de cette case ; les cases du tableau ne contiennent donc ici que les adresses en mémoire des débuts des listes chaînées ou bien l'adresse nulle si aucun élément n'a l'indice de la case comme code. Lors de l'utilisation de la table, pour voir si un « mot » est ou non dans la table, on code ce mot, puis on parcourt la liste des éléments chaînés dans la case d'indice égal au code trouvé, jusqu'à ce qu'on trouve l'élément (situation de succès de la recherche) ou que l'on arrive à la fin de la liste sans l'avoir trouvé (situation de l'échec de la recherche).

Rappelons l'exemple du paragraphe précédent. Les éléments sont : e_1, e_2, e_3, e_4, e_5, e_6 et e_7 avec $h(e_1) = 4$, $h(e_2) = 6$, $h(e_3) = 2$, $h(e_4) = 4$, $h(e_5) = 2$, $h(e_6) = 2$, $h(e_7) = 5$. On utilise un tableau de longueur $m = 7$. On obtient le tableau suivant :



IV.7. Nombre de comparaisons

Nous ne calculons pas ici le nombre de comparaisons effectuées en moyenne pour une insertion ou une recherche dans une table de hachage et nous nous contentons de quelques indications.

Signalons que la technique du hachage linéaire est moins performante que les deux autres techniques et que, dans ce cas, il faut prendre un tableau largement plus grand que le nombre de données à ranger ; en effet, on peut remarquer que, lorsqu'il y des plages de cases consécutives occupées, alors, plus la plage est large, plus la probabilité qu'une prochaine donnée à insérer ait son code dans cette plage est grande ; les plages les plus grandes ont tendance à s'agrandir davantage.

Les deux techniques avec chaînages ont des performances équivalentes si la zone de débordement du hachage avec chaînage externe est prévue suffisamment grande. Le hachage externe est marginalement meilleur quand aucune zone de débordement n'est prévue.

On dit qu'une recherche est positive si la donnée cherchée figure dans la table et qu'elle est négative dans le cas contraire. Le nombre moyen d'accès au tableau pour rechercher une donnée est toujours un peu plus grand pour une recherche négative que pour une recherche positive. Quand on compte le nombre moyen d'accès au tableau, on inclut l'accès éventuel

qui indique que la chaîne est terminée. Les évaluations ci-dessous restent correctes même si la zone de débordement du hachage avec chaînage interne a une taille nulle.

Si le nombre de cases du tableau est deux fois plus grand que le nombre de données à ranger, le nombre moyen d'accès au tableau pour rechercher une donnée est compris entre 1,2 et 1,4 pour une recherche positive et entre 1,5 et 1,7 pour une recherche négative dans le cas des hachages avec chaînage. Pour le hachage linéaire, il est de l'ordre de 1,5 pour une recherche positive et de 3 pour une recherche négative.

Si le nombre de cases du tableau est égal à 1,5 fois le nombre de données à ranger, le nombre moyen d'accès au tableau pour rechercher une donnée est compris entre 1,3 et 1,5 pour une recherche positive et entre 1,6 et 1,9 pour une recherche négative dans le cas des hachages avec chaînage. Pour le hachage linéaire, il est de l'ordre de 2,5 pour une recherche positive et de 6 pour une recherche négative.

Si le nombre de cases du tableau est égal à 1,1 fois le nombre de données à ranger, le nombre moyen d'accès au tableau pour rechercher une donnée est compris entre 1,4 et 1,7 pour une recherche positive et entre 1,9 et 2,2 pour une recherche négative dans le cas des hachages avec chaînage. Le hachage linéaire devient impraticable.

V. L’algorithme de Huffman

V.1. Présentation du problème

L’algorithme de Huffman est utilisé en « codage de source ». Il s’agit de coder les caractères d’un texte à l’aide de suites de 0 et de 1. On appellera *mot de code* le codage d’un des caractères ; un mot de code est donc une chaîne binaire écrite avec des 0 et des 1.

Une première possibilité est d’utiliser un codage où tous les mots de code ont la même longueur ; on dira alors qu’il s’agit d’un *codage de longueur fixe*. Si le texte est écrit avec n caractères différents, il faudra avoir des mots de code de longueur $p = \lceil \log_2 n \rceil$. Prenons un exemple ; supposons que le texte soit écrit uniquement avec les caractères a, b, c, d, e, f, g et h et contiennent 5852 caractères. Étant donné qu’il n’y a que 8 caractères à coder, on peut les coder avec des chaînes binaires distinctes de longueur 3 ; le texte codé sera alors pour longueur (comptée en nombre de 0 et de 1) égale à $3 \times 5852 = 17556$. Le texte codé est facile à décoder puisqu’il suffit de découper le texte codé en mots de longueur p.

On désire, pour diverses raisons (place de stockage, durée de transmission, risque d’erreur de transmission…), avoir un texte codé le plus court possible. Pour cela, on abandonne l’idée du codage de longueur fixe et on va faire en sorte que les caractères les plus fréquents aient des codages plus courts que les caractères plus rares. Cela nécessite d’avoir :

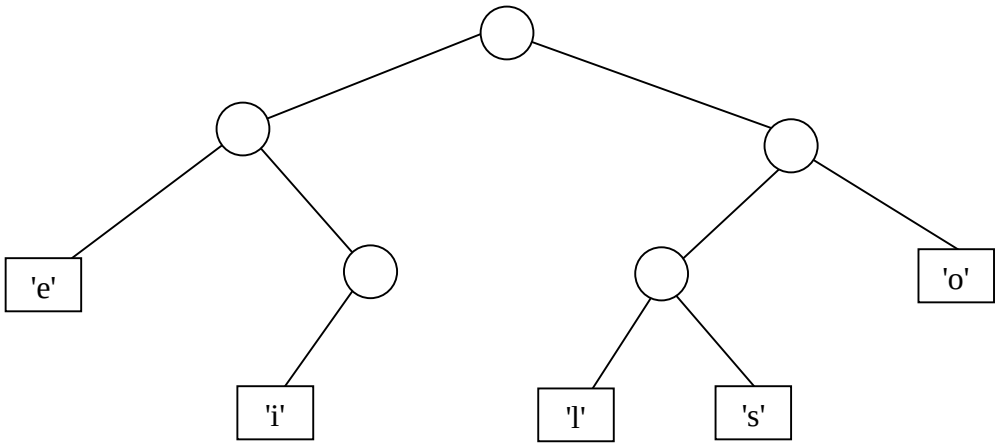
- soit une estimation de la fréquence des caractères dans le type de texte qu’on doit coder ; on pourrait par exemple faire des statistiques sur la fréquences des différents caractères dans l’ensemble des textes écrits en français et utiliser ces statistiques pour le codage d’un texte écrit en français ;
- soit calculer précisément les nombres d’occurrences de chacun des caractères dans le texte que l’on souhaite coder.

Il s’agit donc maintenant, muni de ces informations, de définir un codage de longueur variable. Pour cela, il faut se préoccuper du décodage ; il faudra savoir où s’arrête chacun des mots de code. On peut utiliser un séparateur, mais il est possible de ne pas y recourir dans le cas où le codage respecte la *règle du préfixe* : un ensemble de mots de code vérifie la règle du préfixe si un mot de code n’est jamais préfixe d’un autre mot de code. Par exemple, si on considère les mots de code 0100110 et 010, le second est préfixe du premier ; ces deux mots ne pourront donc pas faire partie simultanément d’un codage respectant la règle du préfixe. Un codage qui suit la règle du préfixe est appelé *codage préfixe*. Montrons que si un codage est préfixe, alors le décodage est possible.

On place les mots de code dans un arbre binaire construit selon l’exemple suivant. On suppose qu’on a cinq caractères 'e', 'i', 'l', 'o' et 's' et que le code est donné par :

code('e') = 00, code('i') = 010, code('l') = 100, code('o') = 11, code('s') = 101.

On vérifie facilement qu’il s’agit d’un codage préfixe ; on construit alors l’arbre binaire suivant :



Les caractères sont les feuilles de l'arbre binaire ; un caractère peut être retrouvé dans l'arbre binaire en partant de la racine et en suivant « le chemin indiqué par le code de ce caractère » : un 0 indique de descendre à gauche et un 1 de descendre à droite. Voyons comment on peut décoder le message : 1011110000010100 en utilisant cet arbre. On part de la racine de l'arbre et on lit le premier chiffre du message qui est un 1 : on part à droite dans l'arbre ; le chiffre suivant est un 0, on continue la descente en partant à gauche, le chiffre suivant est un 1, on part à droite et on atteint le caractère 's', qui est ainsi le premier caractère du texte décodé ; on repart de la racine et on lit le chiffre suivant dans le message, c'est-à-dire le quatrième, il s'agit d'un 1, on part à droite, puis encore un 1, on part à droite et on lit le caractère 'o' ; le message commence par "so". Nous laissons le lecteur terminer le décodage et découvrir le message.

Pour un codage non préfixe, on peut encore tracer un arbre analogue, mais certains caractères se trouvent alors dans des nœuds internes ; si, pendant le décodage, on rencontre un tel caractère, on ne peut pas savoir si on a obtenu ainsi un caractère du texte ou bien s'il faut passer son chemin pour aller voir bas.

On peut montrer que, si on se restreint aux codages d'un texte par le codage des caractères (et non pas par exemple de sous-chaînes), le codage de taille minimum est obtenu par un codage préfixe. Le codage de Huffman que nous allons définir plus bas est un codage préfixe qui permet d'obtenir le texte codé de taille minimum.

Si la règle du préfixe est vérifiée, chaque caractère est associé à une feuille de l'arbre, et la longueur (en nombre de bits) de la chaîne codant le caractère est la distance de la racine à cette feuille. Notre but est d'associer à la suite des caractères à coder un arbre binaire tel que, toute feuille étant associée à un caractère c_i de fréquence f_i et étant à une distance l_i de la racine, la somme des produits $l_i f_i$ prise sur toutes les feuilles soit minimum. Cet arbre sera dit *arbre optimum*.

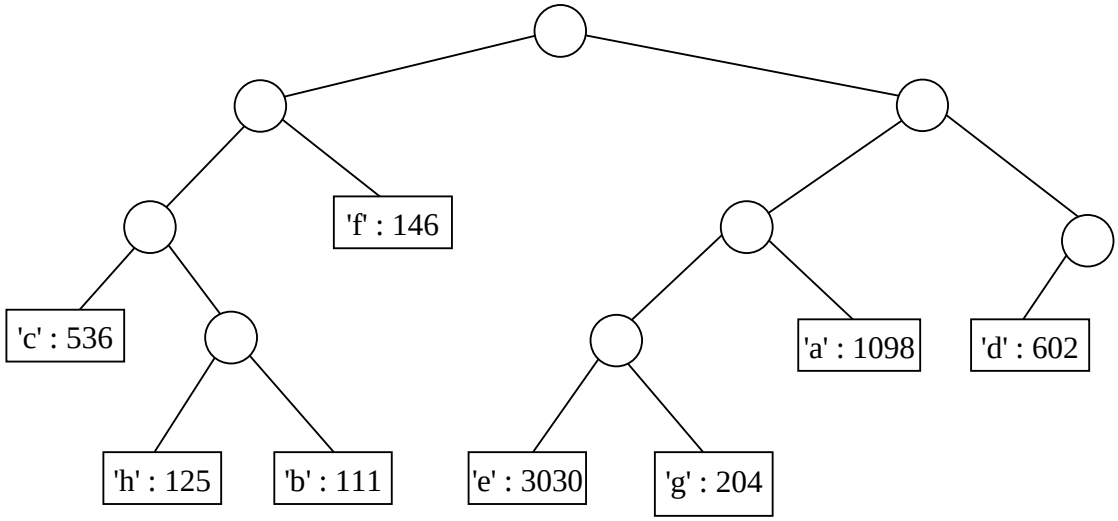
V.2. Quelques remarques préliminaires

On peut d'abord remarquer que l'opération définie dans le paragraphe précédent qui associe à un codage préfixe un arbre binaire permet de définir une bijection entre les codages préfixes d'un ensemble de caractères et les arbres binaires dont les feuilles contiennent exactement ces caractères.

Considérons un texte composé des caractères distincts c_1, c_2, …, c_n, le caractère c_i apparaissant avec un nombre d'occurrences noté $occ(c_i)$. On pourrait considérer les fréquences en divisant les nombres d'occurrences par la longueur du texte à coder, cela reviendrait au même. On considère un codage préfixe des caractères c_1, c_2, …, c_n ; on complète l'arbre binaire correspondant à ce codage en ajoutant à chaque feuille, en plus du caractère x correspondant à cette feuille, le nombre d'occurrences de x ; on note A l'arbre obtenu ; on note $p(x)$ la longueur du code du caractère x ou, ce qui revient au même la profondeur du caractère x dans l'arbre A. Reprenons notre premier exemple du texte de longueur 5852 écrit uniquement avec les caractères 'a', 'b', 'c', 'd', 'e', 'f', 'g' et 'h'. On suppose que l'on a :

occ('a') = 1098, *occ*('b') = 111, *occ*('c') = 536, *occ*('d') = 602,
occ('e') = 3030, *occ*('f') = 146, *occ*('g') = 204, *occ*('h') = 125.

Ces données sont des données issues d'un texte réel sauf que ce texte contient aussi d'autres caractères. On peut alors considérer le codage défini par l'arbre A ci-dessous :



Ce codage correspond à :

code('a') = 101, code('b') = 0011, code('c') = 000, code('d') = 110,
code('e') = 1000, code('f') = 01, code('g') = 1001, code('h') = 0010.

La longueur du texte codé sera alors égale à :

$1098 \times 3 + 111 \times 4 + 536 \times 3 + 602 \times 3 + 3030 \times 4 + 146 \times 2 + 125 \times 4 + 204 \times 4 = 20080$.

Si f est une feuille de A, on note $occ(f)$ le nombre d'occurrences contenu par cette feuille. Posons :

$$L(A) = \sum_{f, \text{ feuille de } A} occ(f) \times p(f) .$$

Donc $L(A)$ donne la longueur du texte codé dans le codage correspondant à A.

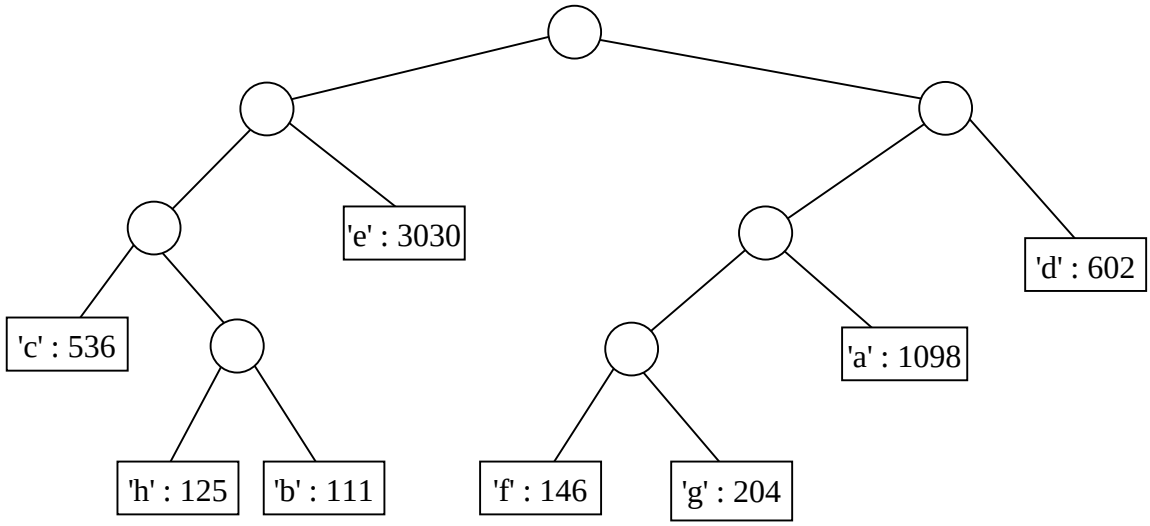
Le problème devient : déterminer un arbre A qui minimise $L(A)$. Un tel arbre sera dit optimum.

Examinons l'arbre de notre exemple. Peut-il être optimum ? On peut faire les deux remarques suivantes :

on gagnerait en remontant la feuille contenant 'd' et en la mettant à la place de son père ; on gagnerait exactement 602 ;

on gagnerait en échangeant la feuille contenant 'f' avec la feuille contenant 'a', on gagnerait exactement $(3030 - 146) \times 2 = 5768$.

En appliquant ces deux transformations, on obtient l'arbre ci-dessous pour lequel L vaut 13710 :



On peut généraliser ces remarques.

Lemme 1 : dans un arbre optimum, tout nœud interne a deux fils.

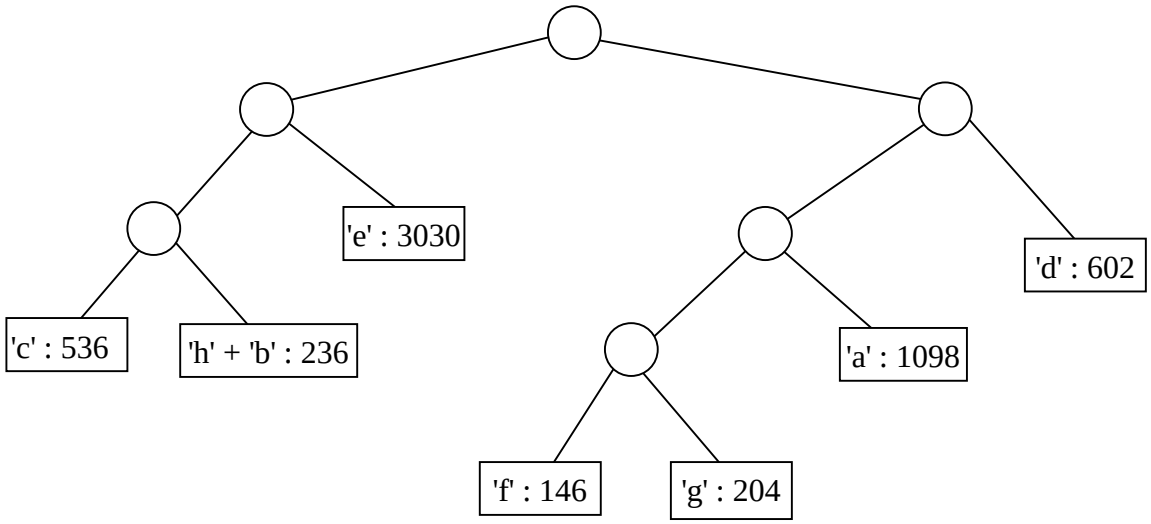
Lemme 2 : dans un arbre optimum, les deux plus petites occurrences se trouvent à la profondeur maximum de l'arbre.

Les preuves de ces deux lemmes sont immédiates. Le second lemme utilise le premier qui montre que, dans un arbre optimum, il y a au moins deux nœuds à la profondeur maximum.

Lemme 3 : il existe un arbre optimum dans lequel les deux plus petites occurrences se trouvent dans des feuilles « frères » qui se trouvent à la profondeur maximum.

En effet, considérons un arbre optimum qui ne vérifie pas la propriété du lemme 3 ; le lemme 2 indique que les deux plus petites occurrences se trouvent à la profondeur maximum de l'arbre et le lemme 1 que toute feuille a une feuille « frère ». Un simple échange permet d'obtenir un arbre qui vérifie la propriété du lemme 3 avec une valeur inchangée de la fonction L.

On considère la transformation suivante de l'arbre A. On considère deux feuilles « frères » f_1 et f_2. On note x le caractère contenu par f_1 et y le caractère contenu par f_2. On supprime ces deux feuilles et on transforme leur père en une feuille contenant la chaîne symbolique $x + y$ assorti d'un nombre d'occurrences égal à $occ(x) + occ(y)$; on note A' l'arbre ainsi obtenu. En appliquant cette transformation aux feuilles contenant les caractères 'b' et 'h' du dernier arbre tracé, on obtient l'arbre A' ci-dessous :



On remarque facilement sur l'exemple : $L(A') = L(A) - 236$. Ce résultat se généralise immédiatement à : $L(A') = L(A) - (occ(x) + occ(y))$.

Lemme 4 : On considère un arbre A où les deux plus petites occurrences sont « frères » à la profondeur maximum ; on note x et y les caractères contenus par ces deux feuilles. On effectue l'opération qui transforme A en A' en utilisant ces deux feuilles. L'arbre A est optimum si et seulement si l'arbre A' est optimum.

Preuve du lemme 4

Supposons que A ne soit pas optimum. Considérons un autre arbre binaire B sur le même jeu de données que A vérifiant $L(B) < L(A)$ et où les feuilles contenant x et y sont « frères » à la profondeur maximum (ce qui est possible d'après le lemme 3). On construit l'arbre B' à partir de B. On a :

$L(A') = L(A) - (occ(x) + occ(y))$
$L(B') = L(B) - (occ(x) + occ(y))$

et donc $L(B') < L(A')$; l'arbre A' n'est donc pas optimum.

Supposons que A' ne soit pas optimum. Considérons un autre arbre binaire B' sur le même jeu de données que A' vérifiant $L(B') < L(A')$. On remplace la feuille contenant la chaîne $x + y$ en effectuant la transformant inverse de celle considérée plus haut : cette feuille devient un nœud interne ayant pour fils deux feuilles contenant x et y avec leurs occurrences respectives. On obtient un arbre B vérifiant : $L(B) = L(B') + (occ(x) + occ(y))$. De la relation $L(A) = L(A') + (occ(x) + occ(y))$, on déduit que $L(B) < L(A)$; l'arbre A n'est donc pas optimum.

Ce résultat est à la base de l'algorithme décrit ci-dessous.

V.3. Description de l'algorithme de Huffman

On construit un arbre réduit à un nœud pour chacun des caractères, ce nœud contenant le caractère et son nombre d'occurrences. On range dans un tableau A (indicé à partir de 1) les nœuds par ordre décroissant de nombres d'occurrences. On considère les nœuds situés en $A[n-1]$ et $A[n]$; ils contiennent les plus petits nombres d'occurrences ; on note x et y les caractères contenus par ces noeuds ; on construit un nouveau nœud noté N qui contient la chaîne $x + y$ et la somme des nombres d'occurrences de ces deux caractères ; on donne à N

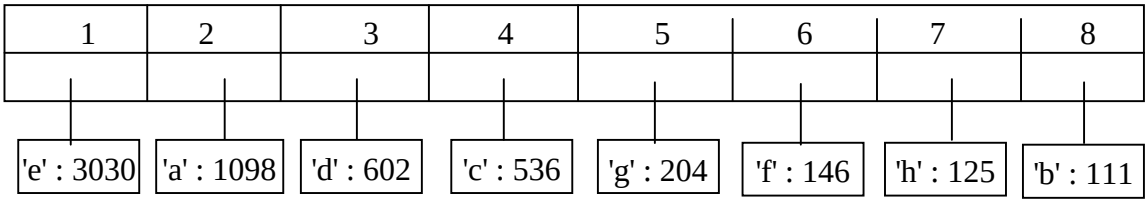
comme fils gauche $A[n-1]$ et comme fils droit $A[n]$ puis on remplace $A[n-1]$ par N. On considère qu'il ne reste plus que $n-1$ arbres dans le tableau A rangés de l'indice 1 à l'indice $n-1$; on remanie le tableau A pour qu'il soit à nouveau trié par ordre décroissant des nombres d'occurrences, ce qui revient à insérer le nœud N dans la sous-liste triée qui le précède. On procède ainsi jusqu'à ce qu'il n'y ait plus qu'un arbre dans la liste. On obtient alors un codage optimal, comme l'énonce le théorème suivant.

Théorème : L'algorithme de Huffman permet de construire un codage préfixe optimal.

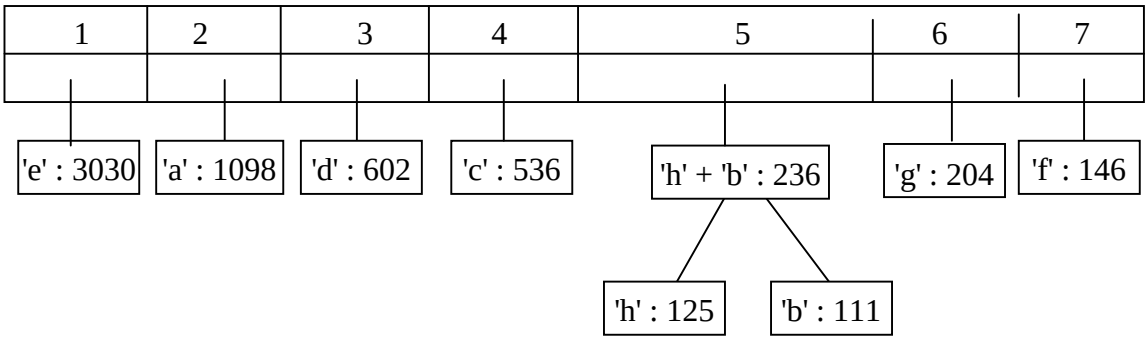
La preuve de ce théorème (et donc de l'algorithme de Huffman) est laissée en exercice et découle des lemmes précédents.

V.4. Exemple d'application

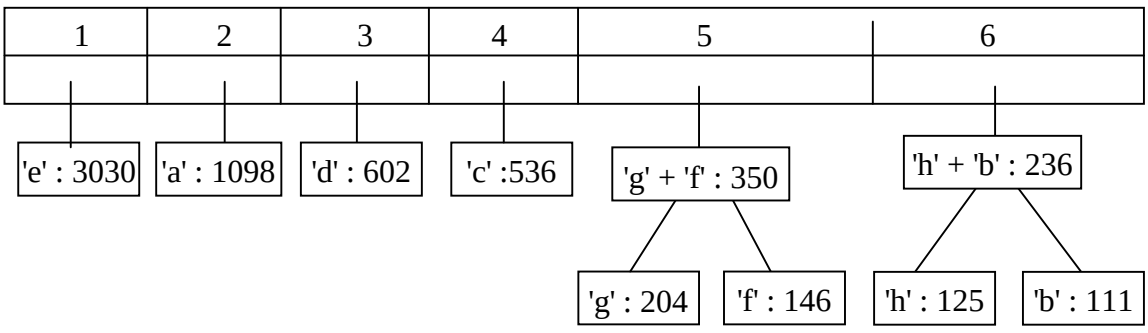
On reprend l'exemple précédent à ses débuts en classant les nombres d'occurrences par ordre décroissant ; on obtient :



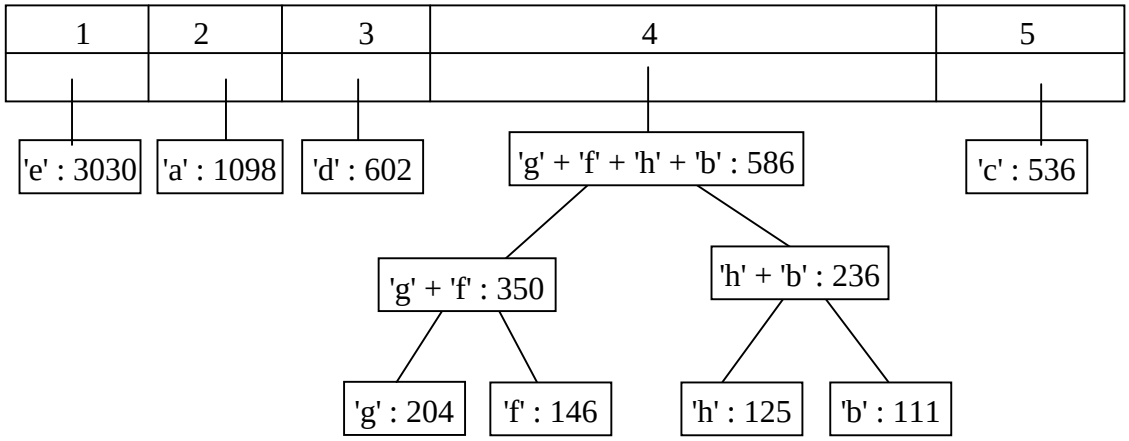
On effectue la première étape de l'algorithme :



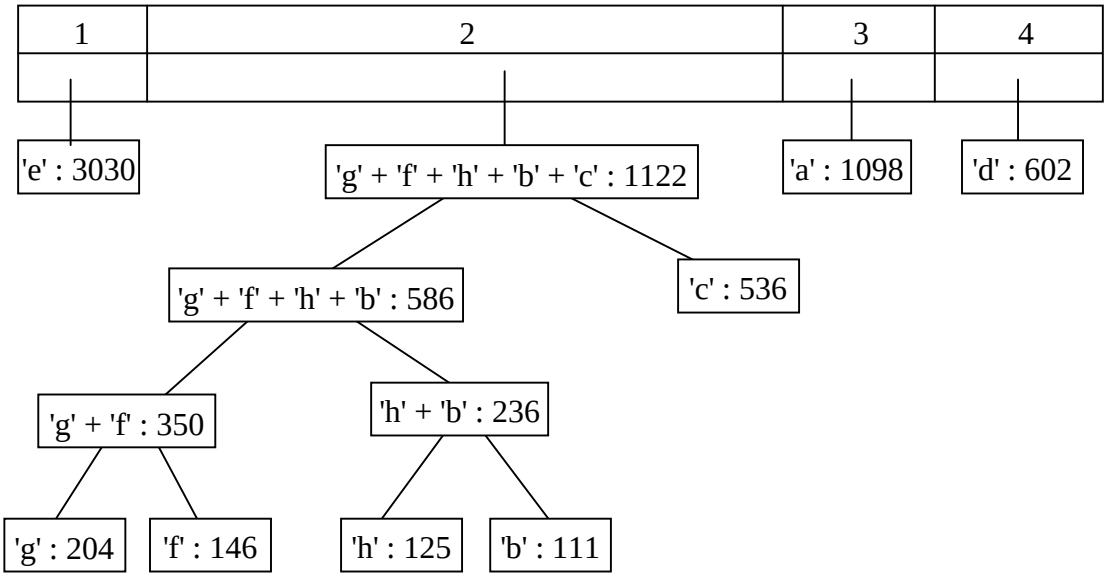
puis l'étape suivante :



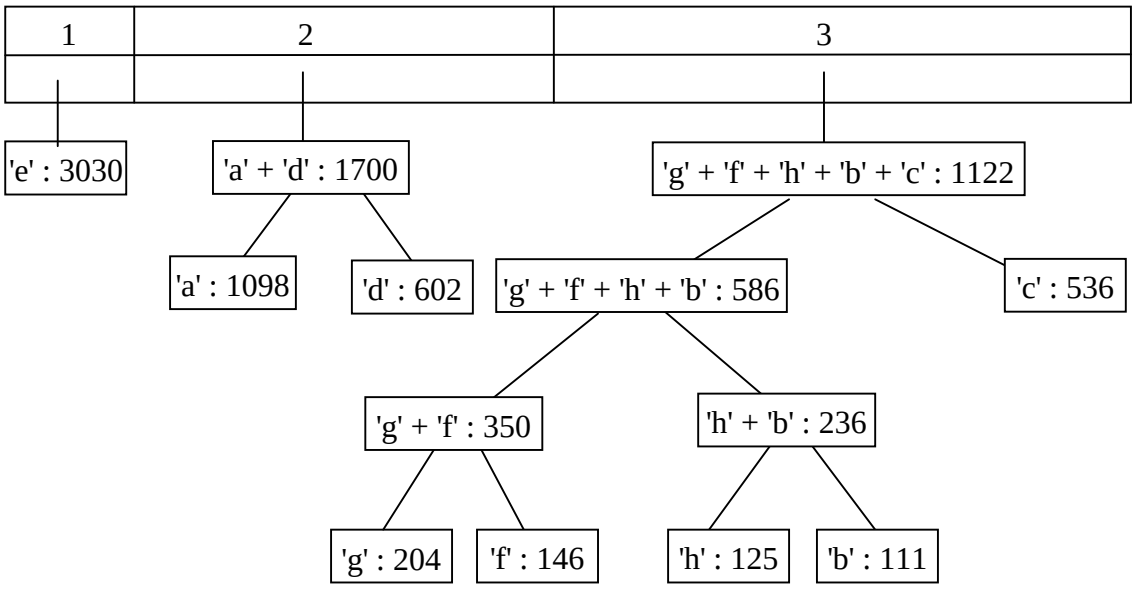
puis :



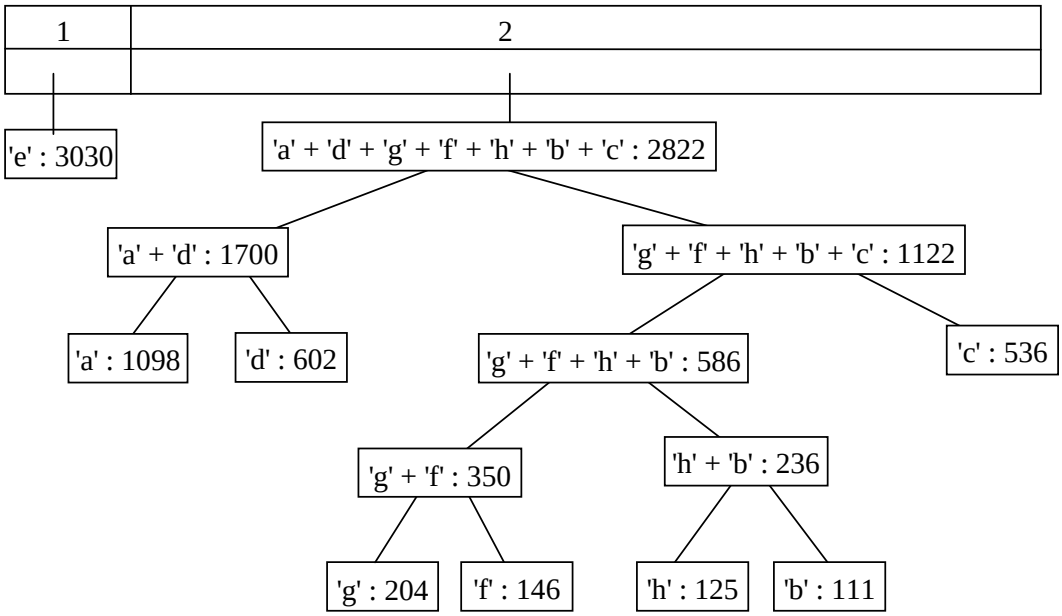
puis :



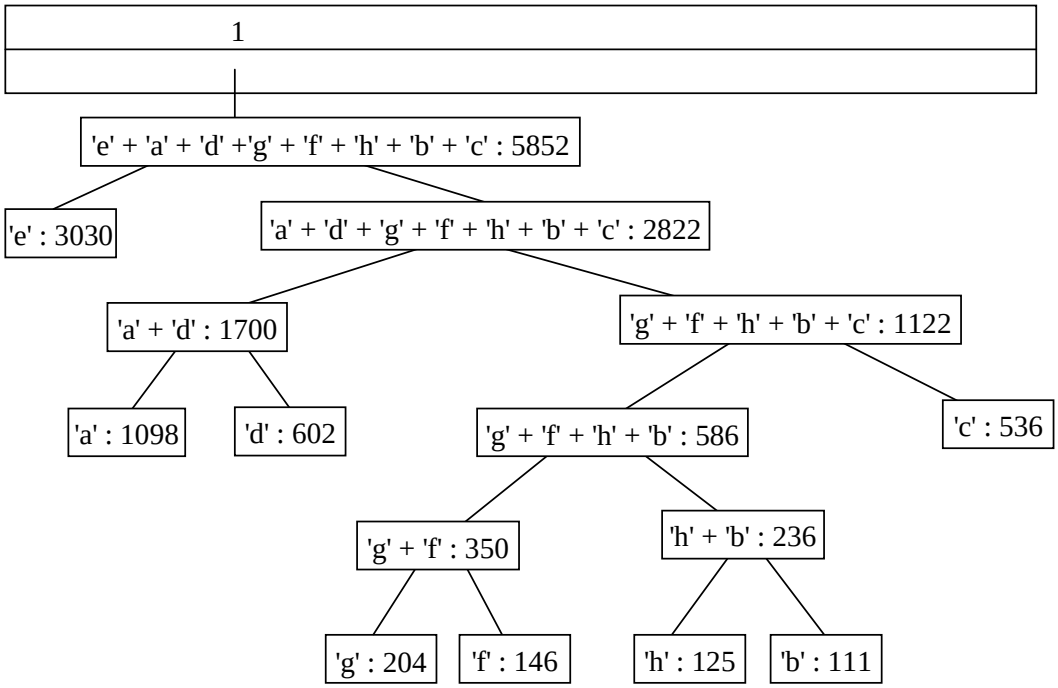
puis :



puis :



et enfin :



Cet arbre donne un code préfixe optimum :

code(a) = 100, code(b) = 11011, code(c) = 111, code(d) = 101,
code(e) = 0, code(f) = 11001, code(g) = 11000, code(h) = 11010.

Avec ce code, le texte codé a pour longueur :

$$(204 + 146 + 125 + 111) \times 5 + (1098 + 602 + 536) \times 3 + 3030 \times 1 = 12668.$$

Tout autre codage des caractères du texte donnera une longueur au moins aussi grande.

L'utilisation d'un tas permet d'obtenir une complexité en $O(n \ln n)$, s'il y a n caractères à code

VI. Généralités sur les graphes. Arbre couvrant de poids minimum

VI.1. Introduction à la théorie des graphes et définitions

Les graphes, orientés ou non orientés, interviennent dans de nombreux problèmes ; ils permettent par exemple de modéliser des réseaux, qu'ils soient de communication ou de commutateurs, ou de représenter des relations, comme nous le comprendrons très vite...

Un *graphe (simple) non orienté* $G = (X, E)$ est défini par deux ensembles : l'ensemble X des *sommets* et l'ensemble E des *arêtes*, un élément e de E étant défini par une paire de sommets distincts x et y de X (on considérera qu'une arête donnée n'apparaît pas plusieurs fois dans E). On dit que x et y sont *incidents* à e, que x et y sont les *extrémités* de e, que x et y sont *adjacents* et on écrit souvent $e = \{x, y\}$.

Un *graphe (simple) orienté* $G = (X, U)$ est défini par l'ensemble X de ses sommets et l'ensemble U de ses *arcs* ; un arc u de U est défini par un couple de sommets distincts x et y (comme pour le cas non orienté, on considérera qu'un arc donné n'apparaît pas plusieurs fois dans U, ce qui n'empêche pas d'avoir simultanément les arcs (x, y) et (y, x), puisque ces arcs sont différents l'un de l'autre) ; on dit que u admet x comme *origine* (ou aussi *extrémité initiale*), y comme *extrémité finale* ou *terminale*, et on note $u = (x, y)$.

Dans ce chapitre et les suivants, nous ne considérons que des graphes *finis*, c'est-à-dire des graphes tels que le cardinal de X, appelé *ordre* du graphe, et celui de E ou de U, appelé *taille* du graphe, sont finis ; on pose $n = |X|$ et $m = |E|$ ou $m = |U|$.

Le *graphe complet* (ou *clique*) à n sommets, noté K_n, est le graphe non orienté d'ordre n dont deux sommets quelconques sont adjacents. Il est donc de taille

$$\frac{n(n-1)}{2}.$$

On appelle *graphe partiel* de $G = (X, E)$ un graphe ayant même ensemble de sommets X que G et ayant pour ensemble d'arêtes une partie de E. Étant donnée une partie Y de X, on appelle *sous-graphe* F de G *engendré* par Y le graphe ayant pour ensemble de sommets Y, une arête (respectivement un arc) de G donnant naissance à une arête (respectivement un arc) de F si et seulement si les deux extrémités de cette arête (respectivement de cet arc) sont dans Y.

Étant donné un sommet x d'un graphe G non orienté, on appelle *degré* de x et on note $d(x)$ le nombre d'arêtes incidentes à x ; les autres extrémités de ces arêtes constituent l'ensemble des *voisins* de x. Si x est un sommet d'un graphe G orienté, on appelle *degré sortant* de x, ou *demi-degré extérieur* de x, noté $d^+(x)$, le nombre d'arcs d'origine x ; on définit similairement le *degré entrant*, ou *demi-degré intérieur* de x, noté $d^-(x)$, comme étant le nombre d'arcs admettant x comme extrémité terminale. On dit que y est un *prédécesseur* ou un *antécédent* (respectivement un *successeur*) de x si l'arc (y, x) (respectivement l'arc (x, y)) existe ; quand

le prédécesseur de x est unique, on l'appelle aussi *père* de x et on dit que x est un *fils* de ce sommet ; on dit que y est un voisin de x si y est un prédécesseur ou un successeur de x.

Soit $G = (X, E)$ un graphe non orienté. On appelle *chaîne* une suite $x_1e_1x_2...$ $x_{k-1}e_{k-1}x_k$ avec $x_i \in X$ pour $1 \leq i \leq k$, $e_j \in E$ et $e_j = \{x_j, x_{j+1}\}$ pour $1 \leq j \leq k-1$. Une chaîne est dite *élémentaire* si tous les sommets la constituant sont deux à deux distincts.

Un *cycle* est une chaîne dont les deux extrémités coïncident. Un cycle $x_1e_1...x_{k-1}e_{k-1}x_1$ est dit *élémentaire* si tous les sommets x_i $(1 \leq i \leq k-1)$ sont deux à deux distincts.

Un graphe est dit *connexe* si, pour toute paire de sommets, il existe une chaîne les joignant. Étant donné un graphe $G = (X, E)$, on définit une relation d'équivalence sur X en disant que deux sommets x et y sont équivalents si et seulement s'ils sont extrémités d'une même chaîne dans G. Les classes d'équivalence de X pour cette relation s'appellent *composantes connexes* de G.

Un *arbre* est un graphe connexe et sans cycle.

Théorème. *Pour un graphe G d'ordre n et de taille m, il y a équivalence entre*

(a) G est un arbre ;
(b) G est connexe, et on a m = n – 1 ;
(c) G est sans cycle, et on a m = n – 1 ;
(d) G est connexe et la suppression d'une arête quelconque le déconnecte ;
(e) G est sans cycle et l'ajout d'une arête quelconque crée un cycle ;
(f) entre deux sommets quelconques, il existe une chaîne élémentaire unique.

Preuve.— La preuve est laissée en exercice.

VI.2. Représentation des graphes en machine

Nous supposerons les sommets de G numérotés par les entiers de 1 à n, chaque sommet étant repéré par son numéro.

Ce qui suit concerne les graphes non orientés ; les modifications à apporter lorsqu'on désire représenter un graphe orienté sont tout à fait élémentaires et laissées au lecteur. Les deux structures de données les plus utilisées pour représenter un graphe sont les suivantes.

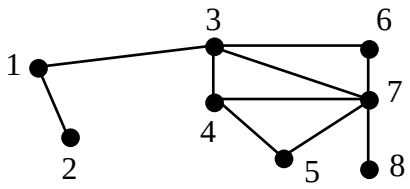
VI.2.1. Matrice d'adjacence (sommets-sommets)

On suppose donnés (lus par exemple sur un fichier) l'ordre n du graphe et l'ensemble des arêtes $\{i, j\}$ du graphe.

On représente le graphe par une matrice d'adjacence, c'est-à-dire une matrice carrée Adj à n lignes et n colonnes, telle que

- $Adj[i, j] = 1 = Adj[j, i]$ si i et j sont adjacents,
- $Adj[i, j] = 0$ sinon ;
- $Adj[k, k]$ sera pris égal à 0 ou 1 suivant le problème.

EXEMPLE

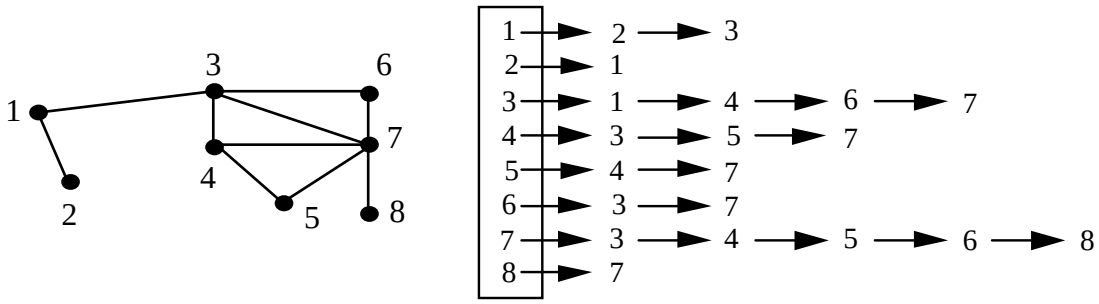


Adj	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	0	0	0	0	0	0
3	1	0	0	1	0	1	1	0
4	0	0	1	0	1	0	1	0
5	0	0	0	1	0	0	1	0
6	0	0	1	0	0	0	1	0
7	0	0	1	1	1	1	0	1
8	0	0	0	0	0	0	1	0

VI.2.2. Tableau de listes d'adjacence

Les données étant les mêmes que précédemment, on définit un tableau de pointeurs de taille *n*, les *n* cases de ce tableau étant en bijection avec les sommets 1, 2, ..., *n* du graphe. Le pointeur en position *i* est la tête d'une liste chaînée qui contient les voisins du sommet *i* : un voisin *j* de *i* doit appartenir à la liste correspondant au sommet *i* et *i* à la liste de *j*.

EXEMPLE



Le choix éventuel entre ces deux structures dépend :

- (comme toujours) de l'algorithme que l'on désire implémenter ;
- de la taille du graphe : si cette taille est « petite », c'est-à-dire de l'ordre de *n* ou de $n.\log_2 n$, on choisira plutôt la structure par listes d'adjacence, sinon on choisira plutôt la représentation matricielle.

REMARQUES

1. En fonction de l'algorithme que l'on désire programmer, on peut être amené à imaginer d'autres structures de données, comme le montre l'exemple de l'algorithme de Kruskal dans la partie I.4.

2. Les structures précédentes peuvent facilement être généralisées à des graphes pondérés, c'est-à-dire dont les arêtes ou les arcs sont munis d'une valuation appelée, selon les cas, « coût », « poids », « longueur », etc. Le coût (ou poids, ou longueur…) d'un graphe partiel est alors la somme des coûts des arêtes ou des arcs le constituant.

VI.3. Complexité d'un algorithme

Nous parlerons beaucoup, tout au long de cet ouvrage, d'*algorithmes*. Un algorithme est une méthode de résolution constituée d'une suite finie d'instructions qui, pour toute instance du problème à traiter, fait passer des données initiales au résultat cherché. Définir un algorithme consiste à décrire pas à pas les instructions que cet algorithme doit exécuter.

L'étude des performances d'un algorithme conduit à définir la *complexité* de celui-ci, qui constitue un paramètre important pour mesurer l'efficacité de l'algorithme. De façon informelle, la complexité d'un algorithme évalue un majorant du nombre d'opérations élémentaires qu'on doit effectuer, dans le pire des cas, pour obtenir le résultat cherché. Par *opération élémentaire*, on entend des opérations comme la comparaison, l'affectation, les opérations arithmétiques, etc., appliquées à des types simples comme des entiers ou des réels ; les vecteurs ou les matrices par exemple ne seront pas considérés comme des types simples : l'addition de deux vecteurs à p composantes sera considérée comme p opérations (additions) élémentaires. Ce nombre d'opérations élémentaires, et donc la complexité, est exprimé à l'aide de la *taille des données*, c'est-à-dire en fonction du nombre de bits nécessaires pour coder les données en machine ; par exemple, coder la matrice d'adjacence d'un graphe à n sommets nécessite n^2 bits, et le codage d'un entier K nécessitera environ $\log_2 K$ bits.

Plus formellement, soit un algorithme A permettant de résoudre un problème P. Soit I une *instance* de P, c'est-à-dire une spécification des données définissant le cas à traiter. Soit $f(A, I)$ le nombre d'opérations élémentaires effectuées pour passer de I au résultat voulu à l'aide de A. La complexité c_A de A est alors définie comme la fonction qui, sur l'ensemble des instances I de taille fixée, considère le maximum de $f(A, I)$:

$$c_A(n) = \max\{ f(A, I) \text{ pour toute instance } I \text{ telle que } |I| = n \}$$

où $|I|$ représente la taille de I.

Il est souvent difficile de déterminer de façon précise l'expression de c_A. Pour cette raison, on ne s'intéresse qu'à un majorant asymptotique de $c_A(n)$ que l'on note $O(g(n))$, en utilisant les notations de Landau, ce qui signifie qu'il existe une constante K et une valeur N_K telles qu'on ait, pour tout n vérifiant $n \geq N_K$:

$$|c_A(n)| \leq K.|g(n)|.$$

Les algorithmes dont la complexité est majorée par un polynôme n^k de la taille n des données sont dits *polynomiaux*, en $O(n^k)$; les autres sont dits *exponentiels*.

La complexité donne des indications importantes sur le temps de calcul. Par exemple, si on utilise un algorithme linéaire (donc en $O(n)$ si n représente la taille des données), on peut prévoir que le traitement de données deux fois plus grosses pourra prendre, *grosso modo*, deux fois plus de temps, tandis que l'utilisation d'un algorithme en $O(n^3)$ pourrait prendre huit fois plus de temps. C'est la raison pour laquelle on préférera généralement des algorithmes de faible complexité.

VI.4. Le problème de l’arbre couvrant de poids minimum

VI.4.1. Définition du problème

Étant donné un graphe non orienté dont les arêtes sont munies d’une valuation que nous appellerons indifféremment « coût » ou « poids » ou « longueur », on cherche un graphe partiel de ce graphe qui soit un arbre et qui soit de coût minimum. Puisqu’il s’agit d’un graphe partiel, cet arbre a pour ensemble de sommets l’ensemble de tous les sommets du graphe initial ; aussi dit-on qu’il s’agit d’un *arbre couvrant*. Dans la terminologie anglaise, ce problème est connu sous le nom de *connector problem* ou encore *minimum spanning tree*.

Il est clair que le problème a une (ou des) solution(s) si et seulement si le graphe est connexe ; nous supposerons dans toute la suite qu’il en est ainsi.

Remarquons que, si tous les coûts sont positifs, cet arbre couvrant de coût minimum est solution du problème suivant : trouver un graphe partiel connexe de G, qui soit de coût minimum.

VI.4.2. Une application en réseaux

Ce problème peut se poser lors de l’établissement d’un réseau, qu’il soit de communication ou d’interconnexion, si, ayant estimé le coût des liaisons directes entre toutes les paires d’objets à relier, on cherche à réaliser un réseau connexe de coût minimum. Il est clair qu’une solution optimale à ce problème est un graphe sans cycle puisque, tous les coûts étant positifs, si une solution comportait un cycle, on obtiendrait une solution plus économique en supprimant une arête de ce cycle. Or, on prouve aisément que la suppression d’une arête d’un cycle ne peut déconnecter un graphe connexe (autrement dit, une arête d’un cycle n’est jamais un *isthme*, un isthme étant toute arête dont la suppression augmente le nombre de composantes connexes).

VI.4.3. Une application en traitement d’images

Une image (en noir et blanc) est représentée sous forme de *pixels*, définis par les 512 lignes et 512 colonnes d’un réseau. Chaque *pixel* possède un certain niveau de gris et, en dehors des points situés sur les bords, admet huit voisins. On cherche à déterminer des régions dans cette image, c’est-à-dire des parties connexes de l’image constituées de points de niveaux de gris très voisins. On modélise la donnée par un graphe, dont presque tous les sommets sont de degré 8 : chaque sommet du graphe correspond à un des 262144 *pixels* de l’image, deux sommets sont adjacents si et seulement si les *pixels* qu’ils représentent sont voisins, et on value l’arête qui les joint par un nombre proportionnel à la différence de leurs niveaux de gris.

Moriss et Constantinidès ont proposé une *heuristique*, c’est-à-dire une méthode approchée, pour déterminer les régions d’une image : construire un arbre couvrant le graphe et de coût minimum puis supprimer de cet arbre couvrant les arêtes de coût supérieur à un seuil donné, fixé de façon à séparer deux *pixels* dont les niveaux de gris sont trop différents pour qu’ils appartiennent à la même zone. On obtient ainsi une *forêt* couvrante, dont les composantes

connexes, qui sont (par définition d'une forêt) des arbres, couvrent ce que l'on considérera comme les zones cherchées.

Nous allons voir ci-après deux algorithmes pour résoudre le problème de l'arbre couvrant de coût minimum. Le premier, appelé algorithme de Kruskal, est peut-être plus intuitif que le second, même si les preuves de ces deux algorithmes sont de difficultés tout à fait comparables. Le second, dit algorithme de Prim, a une très bonne complexité, si du moins on sacrifie un peu de place mémoire pour l'implémenter correctement. Ce phénomène se retrouvera, au chapitre des plus courts chemins, dans l'algorithme de Dijkstra, qui lui ressemble beaucoup.

VI.4.4. Algorithme de Kruskal

Principe de l'algorithme

Pour déterminer un arbre couvrant de poids minimum d'un graphe connexe à n sommets, on sélectionne les arêtes d'un graphe partiel initialement sans arête, en itérant $n - 1$ fois l'opération suivante (n étant l'ordre du graphe) : choisir une arête de poids minimum ne formant pas cycle avec les arêtes précédemment choisies.

Mise en œuvre de l'algorithme

L'algorithme de Kruskal se déroule en deux phases :

- première phase : trier les arêtes par ordre des poids croissants ;
- seconde phase : tant que l'on n'a pas retenu $n - 1$ arêtes, procéder aux opérations suivantes : considérer, dans l'ordre du tri, la première arête non examinée ; si elle forme un cycle avec les arêtes précédemment choisies, la rejeter, sinon la garder.

Preuve de l'algorithme

L'objet constitué par les $n - 1$ arêtes choisies est clairement un arbre couvrant T du graphe de départ G.

Supposons qu'il ne soit pas de poids minimum. Représentons alors T par la suite des arêtes le constituant, dans l'ordre où elles ont été sélectionnées : $T = (e_1, e_2, ..., e_{n-1})$. Parmi les arbres couvrants de poids minimum, choisissons-en un, T_0, coïncidant avec T le plus longtemps possible : $T_0 = (e_1, ..., e_k, f_{k+1}, ..., f_{n-1})$, où $f_{k+1} \neq e_{k+1}$, avec k le plus grand possible. L'arête e_{k+1} n'appartient pas à T_0, à cause du choix de T_0, donc $T_0 \cup \{e_{k+1}\}$ contient un cycle. Ce cycle n'est pas constitué uniquement d'arêtes de T puisque T est sans cycle. Soit f_i une arête de ce cycle n'appartenant pas à T ; $T_0 \cup \{e_{k+1}\} - \{f_i\}$ est à nouveau un arbre couvrant de G et, puisqu'il coïncide davantage avec T que T_0, c'est qu'il n'est pas de poids minimum : c'est donc que le poids de e_{k+1} est strictement supérieur au poids de f_i. Mais alors on aurait dû examiner f_i avant e_{k+1}, et ceci implique que l'on se soit trompé dans l'application de l'algorithme : en effet, les arêtes $e_1, ..., e_k, f_i$ étant dans l'arbre T_0, elles n'engendrent pas de cycle et donc l'examen de f_i avant celui de e_{k+1} aurait dû aboutir à la sélection de f_i, ce qui ne fut pas fait.

Indications pour l'implémentation de l'algorithme

Avant de choisir les structures de données nécessaires à l'implémentation, nous devons d'abord résoudre le problème suivant : comment déterminer si une arête choisie à l'étape *i* forme un cycle avec les arêtes précédemment choisies ?

Une solution pour résoudre ce problème est la suivante : pour qu'une arête $\{x, y\}$ ferme un cycle, il faut que précédemment ses extrémités aient été reliées par une chaîne, et donc aient été dans une même composante connexe. Nous allons donc gérer l'évolution des composantes connexes, au fur et à mesure du choix des arêtes. Initialement, lorsque le graphe partiel ne contient aucune arête, chaque sommet constitue une composante connexe et nous initialisons, pour chaque sommet *i*, un indice à cette valeur *i*. Chaque fois qu'une arête $\{x, y\}$ est candidate, on compare les valeurs des indices de *x* et *y*. Si elles sont égales, c'est que les deux sommets sont déjà dans la même composante, l'arête $\{x, y\}$ créerait donc un cycle et par conséquent on ne la retient pas ; sinon on garde l'arête $\{x, y\}$ et on donne comme valeur à l'indice associé à *y* celle de l'indice associé à *x*, ainsi qu'à tout sommet qui avait auparavant le même indice que *y* (on peut bien sûr intervertir ici le rôle de *x* et celui de *y*), ceci voulant dire que tous les sommets de la composante connexe de *x* et de la composante connexe de *y*, après le choix de l'arête $\{x, y\}$, forment désormais une seule composante et portent donc tous le même indice de composante connexe.

Une structure de données raisonnable semble être un tableau (appelé *CC* dans la présentation qui suit) d'entiers associés aux sommets numérotés de 1 à *n* et qui tiendra à jour les indices des composantes connexes associés à ces sommets. Par ailleurs, nous représentons le graphe, donnée du problème, par un tableau *A* contenant les arêtes, rangées par ordre de poids croissants. Enfin le résultat, c'est-à-dire l'arbre couvrant de poids minimum, sera représenté par le tableau *T* de ses $n - 1$ arêtes. L'algorithme de Kruskal peut alors être décrit de la façon suivante.

- Trier les arêtes par poids croissants et les ranger dans *A* selon cet ordre
- pour *i* qui varie de 1 à $n - 1$, faire $CC(i) \leftarrow i$
- $compteurT \leftarrow 0$
- $compteurA \leftarrow 1$
- tant que $compteurT < n - 1$, faire
    - soit $\{x, y\}$ l'arête $A(compteurA)$
    - $compteurA \leftarrow compteurA + 1$
    - si $CC(x) \neq CC(y)$, alors
        - $compteurT \leftarrow compteurT + 1$
        - $T(compteurT) \leftarrow \{x, y\}$
        - $auxiliaire \leftarrow CC(y)$
        - pour *i* qui varie de 1 à *n*, faire
            si $CC(i) = auxiliaire$, alors $CC(i) \leftarrow CC(x)$

Complexité de l'algorithme

On doit commencer par trier les arêtes du graphe par poids croissants, ce qui nécessite $O(m.\log_2 m)$ opérations élémentaires, où *m* est la taille du graphe. Les autres initialisations peuvent se faire à l'aide de $O(n)$ opérations élémentaires.

Chaque fois qu'on examine une arête candidate, on doit comparer les indices de composante connexe de ses extrémités. En cas d'égalité, on la refuse ; on fait alors O(1)

opérations élémentaires par arête rejetée. Au contraire, si les numéros sont différents, on adopte l'arête ; on doit alors mettre à jour les indices des sommets et pour cela effectuer O(n) opérations élémentaires par arête retenue ; or, nous devons retenir $n - 1$ arêtes. On doit donc effectuer O(n^2) opérations élémentaires pour les $n - 1$ arêtes retenues et au plus O(m) pour les arêtes rejetées, soit au total O(n^2) opérations élémentaires pour cette partie.

On voit donc, en tenant compte du tri initial, que la complexité de l'algorithme de Kruskal est en O($n^2 + m.\log_2 m$), ou encore en O($n^2.\log_2 n$) si l'on considère des graphes pour lesquels m est de l'ordre de n^2.

VI.4.5. Algorithme de Prim

Principe de l'algorithme

Nous allons maintenant décrire un autre algorithme, dû à Prim, qui permet également de trouver un arbre couvrant de poids minimum, en O(n^2) opérations, et cela sans qu'il soit nécessaire de trier les arêtes.

Le principe de l'algorithme de Prim est le suivant : on étend de proche en proche un arbre couvrant une partie des sommets du graphe, en atteignant un sommet supplémentaire à chaque étape, et en prenant pour cela, à chaque étape, l'arête la plus légère parmi celles qui joignent l'ensemble des sommets déjà couverts à l'ensemble des sommets non encore couverts.

Si on analyse cette version brute de l'algorithme, il est facile de se convaincre que le nombre d'opérations élémentaires à effectuer est de l'ordre de n^3, en tous cas lorsque le graphe sur lequel on travaille est le graphe complet. En effet, à l'étape k, lorsque nous avons déjà choisi $k - 1$ arêtes, le cardinal de l'ensemble des sommets couverts est k, celui des sommets non couverts est $n - k$, et le nombre d'arêtes entre ces deux ensembles de sommets est $k.(n - k)$. Déterminer l'arête de poids minimum entre les deux ensembles de sommets coûte par conséquent $k.(n - k) - 1$ comparaisons, d'où le résultat annoncé lorsque l'on fait la somme de ces quantités pour k variant de 1 à $n - 1$.

Nous allons voir qu'on peut ramener la complexité de l'algorithme à O(n^2).

Description de l'algorithme

On appelle S les sommets de l'arbre en cours de construction et on note $p(x, y)$ le poids de l'arête $\{x, y\}$. À chaque étape, on adjoint à S un sommet. À une étape donnée, appelons R l'ensemble des sommets non encore dans S. À chaque sommet x de R on associe *proche*(x), c'est-à-dire le sommet de S qui est tel que le poids de l'arête $\{x, \textit{proche}(x)\}$ soit minimum sur l'ensemble des sommets de S : ce poids est dit « distance » $d(x)$ de x à S. On choisit alors de faire entrer dans S le sommet x de R dont la distance $d(x)$ à S est minimum. Ce sommet sera le *pivot* de l'étape suivante. On met ensuite à jour les attributs des sommets z qui restent dans R et qui sont voisins de *pivot*, en comparant l'ancienne distance $d(z)$ de z à S à la nouvelle façon d'atteindre z à partir de *pivot* : si $p(\textit{pivot}, z)$ est plus petit que l'actuelle valeur de $d(z)$, alors *proche*(z) prend *pivot* comme valeur, et $d(z)$ reçoit $p(\textit{pivot}, z)$.

L'algorithme de Prim peut alors être décrit de la façon suivante.

- $S \leftarrow \{x_0\}$, où x_0 est un sommet quelconque
- $pivot \leftarrow \{x_0\}$
- pour tout sommet x autre que x_0, faire $d(x) \leftarrow +\infty$
- pour i qui varie de 1 à $n - 1$, faire
  - pour tous les sommets z voisins de *pivot* qui ne sont pas dans S faire
    - si $p(pivot, z) < d(z)$ alors
      $proche(z) \leftarrow pivot$
      $d(z) \leftarrow p(pivot, z)$
  - parmi les sommets qui ne sont pas dans S, déterminer un sommet *pivot* qui réalise le minimum des valeurs de d
  - ajouter *pivot* et l'arête $\{pivot, proche\ (pivot)\}$ à S

Preuve de l'algorithme

Elle est laissée en exercice, et nous suggérons une preuve par l'absurde, aussi proche que possible de la preuve de l'algorithme de Kruskal.

Indications pour l'implémentation de l'algorithme

Le choix de la représentation en machine du graphe G est lié à la densité (rapport entre la taille et l'ordre) de G. L'arbre résultat sera codé comme le tableau de ses arêtes. La fonction *proche* sera par exemple codée dans un tableau indicé par les noms des sommets du graphe, si le langage le permet.

Complexité de l'algorithme

Il est facile, à l'aide de la présentation de l'algorithme de Prim donnée plus haut, de montrer que l'application de cet algorithme peut se faire à l'aide de $O(n^2)$ opérations élémentaires.

En effet, les initialisations (les trois premières lignes) nécessitent $O(n)$ opérations élémentaires, la partie la plus longue étant ici l'initialisation de d. D'autre part, on passe exactement $n - 1$ fois dans la boucle qui commence par « pour i » ; il suffit donc de déterminer la complexité d'un passage pour obtenir, en multipliant celle-ci par n, la complexité de la boucle entière. Or, pour i fixé (et donc aussi *pivot* fixé), il y a au plus $n - 1$ sommets z voisins de *pivot* dont il faut mettre à jour les attributs ; cette mise à jour, pour z fixé, demande $O(1)$ opérations élémentaires, d'où $O(n)$ pour la boucle qui commence par « pour tous les sommets z » ; la détermination du sommet *pivot* suivant revenant au calcul d'un minimum parmi au plus n valeurs, cette partie peut se faire en $O(n)$; enfin la mise à jour de T nécessite $O(1)$ opérations élémentaires. Un passage dans la boucle dépendant de i a donc une complexité en $O(n)$, et de ce fait la complexité de la boucle complète est $O(n^2)$, ce qui donne aussi la complexité de l'algorithme tout entier.

VI.5. Exercices

Exercice 1

Prouver qu'un graphe G de degré minimum (c'est-à-dire le minimum des degrés) $\delta \geq 2$ contient au moins une chaîne élémentaire de longueur (nombre d'arêtes) supérieure ou égale à δ et un cycle élémentaire de longueur supérieure ou égale à $\delta + 1$.

Exercice 2

Montrer que dans un graphe non orienté le nombre de sommets de degré impair est pair. En déduire qu'il n'existe pas de graphe cubique (c'est-à-dire un graphe dans lequel tout sommet est de degré 3) d'ordre 5.

Exercice 3

Prouver que, pour un graphe G d'ordre n, il y a équivalence entre les six propositions suivantes :

(a) G est un arbre ;

(b) G est connexe et est de taille $n - 1$;

(c) G est sans cycle et est de taille $n - 1$;

(d) G est connexe et la suppression d'une arête quelconque le déconnecte ;

(e) G est sans cycle et l'ajout d'une arête quelconque crée un cycle ;

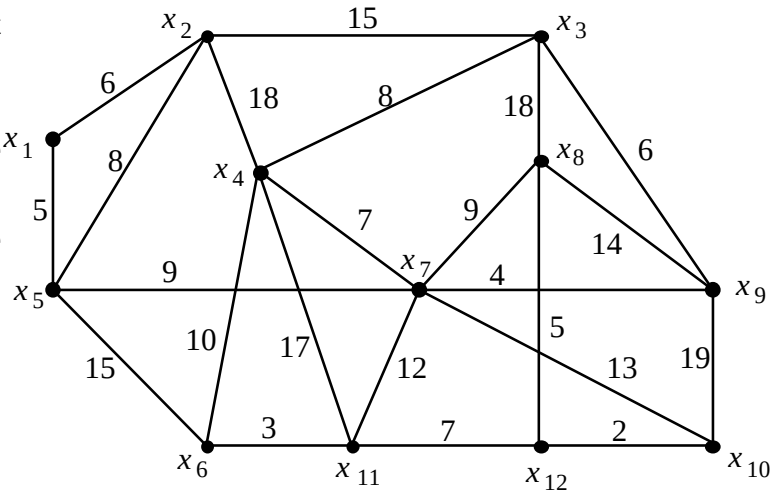
(f) entre deux sommets quelconques il existe une chaîne (élémentaire) unique.

Exercice 4

Déterminer un arbre couvrant de poids minimum du graphe ci-contre :

a) d'abord à l'aide de l'algorithme de Kruskal ;

b) puis à l'aide de celui de Prim.



Exercice 5

Prouver l'algorithme de Prim.

VII. Problèmes de plus courts chemins

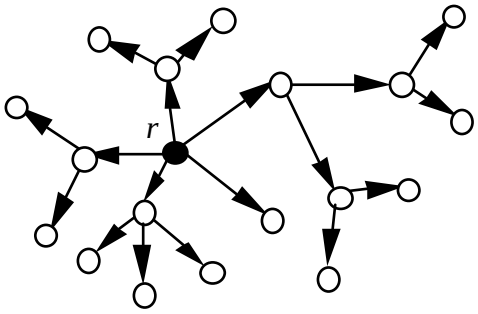
VII.1. Définition des différents problèmes

VII.1.1. Définitions nécessaires à ce chapitre

Soit $G = (X, U)$ un graphe orienté que nous supposons valué, c'est-à-dire qu'on a défini une application p de U dans R, appelée poids ou longueur ou coût ou... On appelle *chemin* une suite $x_1\ e_1\ x_2\ e_2\ ...\ e_{k-1}\ x_k$ de sommets x_i $(1 \leq i \leq k)$ et d'arcs e_j $(1 \leq i \leq k-1)$ telle que l'arc e_j ait pour origine x_j et pour extrémité x_{j+1}. On appelle poids du chemin (ou longueur du chemin ou...) la somme des poids des arcs du chemin. On appelle chemin *élémentaire* un chemin qui n'utilise pas deux fois le même sommet. Un *circuit* étant un chemin dont les extrémités coïncident, on appelle *circuit absorbant* un circuit tel que la somme des valuations de ses arcs soit strictement négative.

On appelle *racine* d'un graphe G orienté un sommet r tel qu'il existe un chemin de r à tout sommet de G. On appelle *arborescence* de racine r un graphe orienté tel que :

- le graphe non orienté sous-jacent (c'est-à-dire obtenu en oubliant les orientations) est un arbre ;
- pour tout sommet x, l'unique chaîne entre r et x du graphe non orienté sous-jacent correspond à un chemin de r vers x dans l'arborescence.



Exemple d'arborescence de racine r

Parmi les caractérisations possibles d'une arborescence figure la proposition suivante : un graphe orienté A est une arborescence de racine r si et seulement si le graphe non orienté sous-jacent est un arbre avec de plus $d^-(r) = 0$ et $d^-(x) = 1$ pour tout sommet x différent de r.

On connaît donc une arborescence lorsque l'on connaît, pour chaque sommet, son père dans l'arborescence.

REMARQUE LIMINAIRE

Dans tout ce chapitre, sauf indication contraire, nous ne nous posons pas de problème d'existence de chemins, d'un sommet vers un autre, ou d'un sommet vers tous les autres. Nous supposons qu'il en existe. Ceci n'est pas irréaliste car nous pouvons toujours considérer

que nous travaillons dans le graphe complet orienté (de tout sommet part un arc vers tout autre sommet). En effet, seuls les plus courts chemins nous intéressant, nous pouvons toujours, s'il n'existe pas d'arc de *a* vers *b*, en ajouter un de longueur (ou poids...) « infinie ». Il y a alors équivalence entre le fait qu'il n'existe pas de chemin de *x* vers *y* dans le graphe initial et le fait que, dans le graphe complété, le plus court chemin de *x* vers *y* soit de longueur « infinie ». Cette hypothèse simplifie considérablement les énoncés.

VII.1.2. Problèmes

On peut considérer les six problèmes suivants :

PROBLEME 1

Étant donnés deux sommets *x* et *y*, trouver un chemin de poids minimum, ou plus court chemin, de *x* à *y*.

PROBLEME 2

Étant donné un sommet *x*, trouver un plus court chemin de *x* à tous les sommets du graphe ou encore, de façon équivalente, déterminer une arborescence de racine *x*, couvrant les sommets du graphe et constituée de plus courts chemins de *x* aux autres sommets.

PROBLEME 3

Trouver, pour toute paire de sommets *x* et *y*, un plus court chemin de *x* à *y*.

Les problèmes 1-bis, 2-bis et 3-bis correspondent aux problèmes 1, 2 et 3 respectivement : il suffit de remplacer le terme « chemin » par le terme « chemin élémentaire » et nous allons tenter d'expliquer ce que cela induit sur les problèmes.

Si un graphe contient un circuit absorbant, les problèmes 1, 2 et 3 peuvent très bien ne pas avoir de solution finie, puisqu'on diminue la longueur d'un chemin chaque fois que l'on utilise un tel circuit.

Inversement, si un graphe ne contient pas de circuit absorbant, toute solution du problème 1-bis (respectivement 2-bis ou 3-bis) est solution du problème 1 (respectivement 2 ou 3). La preuve de ces faits est laissée en exercice (exercice 1), ainsi que les conditions d'existence de solutions, dont nous avons annoncé dans la remarque liminaire qu'elles n'étaient pas la préoccupation essentielle de ce chapitre.

Curieusement, même s'il existe un nombre infini de chemins dans un graphe, puisque rien dans la définition d'un chemin n'interdit de passer plusieurs fois par le même sommet ou le même arc, lorsqu'il existe des solutions aux problèmes 1, 2, 3, on peut les trouver facilement, à l'aide des algorithmes que nous allons expliquer par la suite. Par « facilement », il faut comprendre qu'il existe des algorithmes polynomiaux, que l'on qualifie aussi de « bons algorithmes », qui permettent de déterminer la solution cherchée. Au contraire, lorsque les problèmes 1, 2, 3 n'ont pas de solution, alors que les problèmes 1-bis, 2-bis, 3-bis, en ont (ceci étant dû à l'existence de circuits absorbants dans le graphe), on ne connaît pas de « bons algorithmes » pour résoudre ces problèmes ; en fait, dans ce cas, on ne sait pas résoudre le problème plus efficacement qu'en énumérant tous les chemins élémentaires et en conservant le plus court, ou peu s'en faut.

Enfin, dans un autre ordre d'idée, notons que l'on ne connaît pas, pour résoudre le problème 1, de solution bien meilleure que de résoudre le problème 2.

Pour résoudre le problème 3 on peut, soit résoudre le problème 2 à partir de tout sommet, c'est-à-dire *n* fois si le graphe est d'ordre *n*, soit utiliser un algorithme comme la méthode matricielle qui est expliquée plus loin.

On voit donc que le problème 2 est central dans ce chapitre, c'est de lui que nous traiterons presque exclusivement dans la suite, sauf dans la dernière partie consacrée au problème 3.

Il se trouve qu'il existe deux cas, fort importants en pratique, où l'on est sûr que le graphe ne contient pas de circuit absorbant :

- lorsque tous les poids sont positifs ou nuls et on dispose alors de l'algorithme de Dijkstra ;
- lorsque le graphe est sans circuit, et on dispose alors de l'algorithme de Bellman.

Dans le cas général où on ne peut faire une de ces deux hypothèses, nous ferons appel à un algorithme qui devra soit donner la solution s'il y en a une, soit mettre en évidence un circuit absorbant s'il en existe.

VII.2. Plus courts chemins d'un sommet à tous les autres : cas des valuations positives

VII.2.1. Motivation

Trouver un routage dans un réseau, c'est trouver, pour tout couple de sommets (x, y), un chemin de x vers y, chemin suivant lequel transiteront les messages ou les trames. Un des routages les plus simples, dit de plus courts chemins, consiste à choisir un « plus court chemin » de x vers y. La longueur d'un arc dépend de la modélisation considérée, mais il est fréquent d'obtenir des valuations positives, par exemple pour représenter un temps ou un coût de transmission.

VII.2.2. Algorithme de Dijkstra

Principe de l'algorithme

On étend une arborescence initialement réduite à r, en gagnant à chaque étape un sommet non encore couvert et un arc dont ce sommet est l'extrémité, l'origine étant déjà dans l'arborescence. L'ensemble des sommets couverts par l'arborescence à une étape donnée est noté A. À la fin de l'algorithme, cette arborescence donne, pour chaque sommet x du graphe, un plus court chemin de r à x et est appelée *arborescence des plus courts chemins de r à x.* En fait, on définit deux fonctions sur l'ensemble des sommets : une fonction π, à valeurs réelles, et une fonction *père*, à valeurs dans les noms de sommets. Si on appelle *distance* d'un sommet r à un sommet x la longueur d'un plus court chemin de r à x, à la fin de l'algorithme $\pi(x)$ donne la distance de r à x et *père*(x) le père de x dans l'arborescence des plus courts chemins de r à x (*père*(r) n'est en fait pas défini) ; à une itération quelconque, $\pi(x)$ donne la longueur d'un plus court chemin de r à x n'utilisant comme sommets intermédiaires que des sommets déjà dans A, et *père*(x) désigne le prédécesseur de x sur un tel chemin.

Pour à la fois faciliter la compréhension de l'algorithme de Dijkstra et en permettre la

preuve, nous insérons des assertions (écrites en italiques et entre parenthèses) dans la description de l'algorithme indiquée ci-dessous, assertions que nous prouverons par récurrence sur l'indice *j* d'étape.

- $A \leftarrow \{r\}$
- $pivot \leftarrow r$
- $\pi(r) \leftarrow 0$
- pour tout sommet x différent de r, faire $\pi(x) \leftarrow \infty$
- pour j variant de 1 à $n - 1$, faire
  - pour tout sommet y non encore dans A et successeur de *pivot*, faire
    - si $\pi(pivot) + p(pivot, y) < \pi(y)$, alors
      $\pi(y) \leftarrow \pi(pivot) + p(pivot, y)$
      $père(y) \leftarrow pivot$

  (assertion 1 : pour tout sommet y non dans A, s'il existe au moins un arc dont l'origine est dans A et dont l'extrémité est y, on a :

$$\pi(y) = \min_{x \in A \text{ et } (x,y) \text{ arc}} [\pi(x) + p(x,y)]$$

  et père(y) est un sommet de A qui atteint ce minimum)
  - chercher, parmi les sommets non dans A, un sommet y tel que $\pi(y)$ soit minimum
  - $pivot \leftarrow y$

  (*assertion 2 : π(pivot) est la longueur d'un plus court chemin de r à pivot et père(pivot) est le prédécesseur de pivot dans ce plus court chemin)*
  - $A \leftarrow A \cup (pivot)$

Preuve de l'algorithme

Nous appelons « étape j » l'ensemble des instructions faites pour la valeur j du paramètre.

Nous montrons, par récurrence sur le numéro d'étape, que les assertions sont vérifiées au moment où elles sont énoncées pour toutes les étapes 1, ..., $n - 1$.

Pour $j = 1$, l'assertion 1 est trivialement vérifiée. Pour prouver l'assertion 2, il suffit de remarquer que, tous les poids étant positifs, il faut, d'après le choix de *pivot*, parcourir une distance d'au moins $\pi(pivot)$ pour aller de r à un quelconque autre sommet.

Soit maintenant j vérifiant $2 \leq j \leq n - 1$; nous supposons les assertions 1 et 2 vérifiées pour les étapes de numéro strictement inférieur à j et on considère l'étape j. L'assertion 1 est trivialement vérifiée : l'actualisation qui la précède assure directement la propriété énoncée. Pour prouver l'assertion 2, remarquons qu'il découle de l'hypothèse de récurrence que, pour tout sommet x de A, $\pi(x)$ est la distance de r à x (*pivot* n'est pas encore dans A quand on rencontre l'assertion 2 à l'étape j). Si, partant de r, on se propose de sortir de l'ensemble A, on doit parcourir un chemin allant du sommet r à un sommet x de A (chemin de longueur au moins $\pi(x)$) suivi d'un arc (x, y), où y n'est pas dans A ; on aura alors parcouru, d'après l'assertion 1, une distance d'au moins $\pi(y)$, et donc *a fortiori*, d'après le choix de *pivot*, une distance d'au moins $\pi(pivot)$: sortir de A en partant de r coûte au moins $\pi(pivot)$. Ainsi, les longueurs étant positives, tout chemin ayant son origine en r et son extrémité non dans A aura une longueur d'au moins $\pi(pivot)$: la distance de r à *pivot* vaut donc au minimum $\pi(pivot)$. Or, le chemin de longueur minimum qui, dans A, joint r à $père(pivot)$ prolongé par l'arc $(père(pivot), pivot)$ est de longueur $\pi(pivot)$. D'où l'assertion 2.

Complexité de l'algorithme

À chaque étape, l'actualisation de la fonction π implique $O(d^+(pivot))$ calculs qui sont chacun en $O(1)$ (où $d^+(x)$ représente, rappelons-le, le nombre d'arcs ayant x comme origine). Le pivot décrivant successivement tous les sommets du graphe, on a au total pour cette partie un nombre d'opérations élémentaires en :

$$O\left(\sum_{x \in X} d^+(x)\right),$$

soit $O(m)$, m représentant le nombre d'arcs.

D'autre part, à chaque étape, on doit déterminer le minimum d'un ensemble de q valeurs, où q décroît de $n - 1$ à 1, si n est l'ordre du graphe, ceci pouvant se faire à l'aide de $O(q)$ opérations élémentaires. Par conséquent, on effectue pour cette partie

$$O\left(\sum_{q=1}^{n-1} q\right)$$

opérations élémentaires, soit $O(n^2)$.

L'algorithme de Dijkstra a donc finalement une complexité qui est en $O(n^2)$.

Compléments autour de cet algorithme

Voir les exercices 3 et 4.

VII.3. Plus courts chemins d'un sommet à tous les autres : cas des graphes sans circuit

VII.3.1. Motivation

Un problème d'ordonnancement de tâches peut se définir comme suit : supposons que pour réaliser un travail complexe on doive avoir mené à bien un certain nombre de tâches, dont les périodes d'exécution ne sont pas indépendantes ; on peut effectuer les tâches en parallèle, mais certaines d'entre elles doivent être achevées avant que d'autres puissent commencer. Le problème est alors de savoir quand on peut et quand on doit commencer les tâches pour achever l'ensemble du travail « au plus tôt ». On peut modéliser ce problème en utilisant un graphe orienté. Les sommets de ce graphe représentent les tâches. Un arc de *a* vers *b* indique que la tâche *a* doit être achevée avant que la tâche *b* ne commence. On porte sur l'arc (*a*, *b*) la durée de la tâche *a*. Enfin deux sommets, que nous appellerons *début* et *fin*, modélisent le début et la fin du chantier. Bien entendu il y a un arc entre *début* et toute tâche, de durée nulle ; de même qu'il y en a un de toute tâche vers *fin*, portant la durée de cette tâche. Étant donné un chemin de longueur maximum de *début* à *fin*, comme toutes les tâches qui figurent sur ce chemin doivent être achevées pour que le travail soit terminé, la durée minimum du chantier ne peut être inférieure à la longueur de ce chemin. Réciproquement, si on commence toute tâche dès que possible, la durée totale ne sera pas supérieure à cette longueur. Pour déterminer la durée minimum totale du chantier, nous devons donc chercher un chemin de

longueur maximum dans un graphe qui doit être sans circuit. L'existence d'un circuit implique en effet que le travail est irréalisable... ou que des contraintes inutiles et nuisibles ont été introduites.

Résoudre le problème de la recherche d'un chemin de longueur maximum revient à résoudre le problème de la recherche d'un chemin de longueur minimum dans le graphe obtenu en remplaçant toutes les longueurs par leurs opposées. Si le graphe est sans circuit, cette opération ne risque pas d'introduire de circuit absorbant. Le problème proposé aura donc des solutions, et un algorithme efficace pour le résoudre est l'algorithme de Bellman.

VII.3.2. Algorithme de Bellman

Principe de l'algorithme

On opère sur l'ensemble des sommets du graphe G une opération qu'on appelle un *tri topologique*, c'est-à-dire que l'on numérote les sommets 1, 2, ..., n de telle façon que tous les antécédents du sommet de numéro j aient des numéros inférieurs à j (nous verrons, dans l'exercice 5, que cela est possible si et seulement si le graphe est sans circuit). Une telle numérotation est appelée *numérotation topologique*. Nous noterons $num(x)$ le numéro topologique du sommet x.

On suppose que l'on cherche à résoudre le problème 2 à partir de r. D'après la définition de la numérotation topologique, le sommet de numéro topologique 1 n'admet aucun antécédent et n'est donc extrémité d'aucun chemin. Le sommet r étant par hypothèse racine du graphe, on a nécessairement $num(r) = 1$.

On opère comme dans l'algorithme de Dijkstra en $n - 1$ étapes, en grossissant, à chaque étape, d'un sommet et d'un arc une arborescence dont l'ensemble de sommets est initialement réduit à r : si y est un sommet différent de r, l'arc de l'arborescence d'extrémité y a pour origine le sommet noté $père(y)$.

- déterminer une numérotation topologique *num*
- $\pi(r) \leftarrow 0$
- Pour tout sommet x différent de r, faire $\pi(x) \leftarrow \infty$
- Pour i qui varie de 2 à n, faire
  - soit x le sommet de numéro topologique i : $num(x) = i$
  - $\pi(x) \leftarrow \min\{\pi(y) + p(y,x)$ pour y tel que $num(y) < i$ et (y, x) arc$\}$
  - Si le minimum ci-dessus est réalisé grâce au sommet y, alors $père(x) \leftarrow y$

Complexité de l'algorithme

La méthode expliquée dans l'exercice 5 pour déterminer une numérotation topologique est un algorithme dont le nombre d'opérations est majoré par $K.n^2$, où K est une certaine constante : il est en $O(n^2)$.

Le sommet de numéro topologique i ayant au plus $i - 1$ antécédents, la détermination de « son père dans l'arborescence » nécessite au plus $O(i)$ opérations élémentaires et l'ensemble des recherches, sur tous les sommets du graphe nécessite donc au plus $O\left(\sum_{i=1}^{n} i\right)$,

c'est-à-dire $O(n^2)$, opérations élémentaires.

Cet algorithme est donc lui aussi un algorithme polynomial, en $O(n^2)$.

Preuve de l'algorithme

Si l'on remarque que, pour tout chemin de r au sommet x de numéro topologique i, l'antécédent de x sur ce chemin à un numéro topologique strictement inférieur à i, la preuve devient immédiate en procédant par récurrence sur l'indice i d'étape.

VII.4. Plus courts chemins d'un sommet à tous les autres : cas général

VII.4.1. Motivation

Lorsque l'on modélise un problème de nature économique, la valuation des arcs correspondant à la notion de coût (ou symétriquement de profit) peut, dans un même énoncé, avoir des valeurs positives ou négatives suivant les arcs : nous en verrons un exemple dans l'exercice 6. La recherche de plus courts chemins dans un graphe avec des circuits et des arcs de valuation quelconque peut donc se poser directement.

VII.4.2. Algorithme de Ford

Le but de cet algorithme est de mettre en évidence, s'il en existe un, un circuit absorbant dont on puisse atteindre un sommet par un chemin d'origine r, et sinon de donner une arborescence de plus courts chemins de r à tous les sommets que l'on peut atteindre par un chemin à partir du sommet r.

Principe de l'algorithme

Cet algorithme fonctionne par étapes. À l'étape k on cherche, de r vers tout sommet x de G, un plus court chemin ayant au plus k arcs, et la longueur d'un tel plus court chemin. Dans un graphe d'ordre n, les chemins élémentaires ont au plus $n - 1$ arcs ; l'ensemble des longueurs des plus courts chemins trouvées à l'étape $n - 1$ doit donc être identique à celui de l'étape n, sauf si les chemins que l'on construit ne sont pas tous élémentaires, ce qui implique que le graphe contient un circuit absorbant (auquel cas le problème n'est pas borné inférieurement). Pour savoir si on est dans un tel cas, on utilise dans la description ci-dessous un booléen, *changement*, qui indique si le passage de $k - 1$ à k fait varier la longueur d'au moins un des chemins qu'on détermine. Si, quand k vaut au plus n, aucune distance n'a variée par rapport à l'itération précédente, alors le graphe ne contient pas de circuit absorbant et les dernières distances calculées donnent la longueur des plus courts chemins issus de r. Si au contraire on constate encore des changements bien que k vaille n, alors c'est qu'il existe un circuit absorbant et les distances calculées ne peuvent être considérées comme les longueurs des plus courts chemins.

Dans la présentation qui suit, on constatera que, pour chaque sommet x, on n'utilise que deux valeurs consécutives de la suite $dist^{(k)}(x)$. On peut donc modifier l'algorithme de façon

à consommer moins de place mémoire. On peut même s'arranger pour ne manipuler qu'un seul tableau *dist* en remplaçant, dans le calcul du minimum ci-dessous, $dist^{(k)}$ et $dist^{(k-1)}$ par *dist*, ce qui permet simultanément d'accélérer l'algorithme et d'économiser de la place mémoire.

- $k \leftarrow 0$
- $dist^{(0)}(r) \leftarrow 0$
- pour tout sommet x différent de r, faire $dist^{(0)}(x) \leftarrow \infty$
- répéter
    - *changement* ← faux
    - $k \leftarrow k + 1$
    - pour tout sommet x, faire
        - $dist^{(k)}(x) \leftarrow \text{Min}\left\{ dist^{(k-1)}(x), \underset{y \text{ prédécesseur de } x}{\text{Min}} \left[dist^{(k-1)}(y) + p(y,x) \right] \right\}$
        - si $dist^{(k-1)}(x) \neq dist^{(k)}(x)$, alors
            $père(x) \leftarrow y^*$, où y^* est un prédécesseur de x qui réalise le minimum précédent
            changement ← vrai
- jusqu'à $k = n$ ou *changement* = faux
- si *changement* = vrai, il existe un circuit absorbant accessible depuis r

Preuve et complexité de l'algorithme

La preuve de l'algorithme découle du principe énoncé plus haut et peut se faire par récurrence.

Pour le calcul de la complexité, constatons que la boucle « répéter » est exécutée au plus n fois. Or, un passage dans la boucle est en $O(m)$ si on dispose d'une structure de données appropriée (liste des prédécesseurs pour chaque sommet) : pour x fixé, la recherche du minimum qui définit $dist^{(k)}(x)$ peut se faire en $O(d^-(x))$, d'où le résultat puisque la somme des demi-degrés intérieurs vaut m. Au total, la complexité de l'algorithme de Ford est donc en $O(n.m)$.

VII.4.3. Algorithme général de Ford-Dantzig

Principe de l'algorithme

On part d'une arborescence A de racine r et d'une fonction π définie sur les sommets telles qu'on ait $\pi(r) = 0$ et, pour tout arc (s, t) de A, $\pi(t) = \pi(s) + p(s, t)$. Une telle arborescence et une telle fonction peuvent par exemple être déterminées par l'algorithme de Dijkstra.

On cherche alors s'il existe un arc (k, l) non dans l'arborescence, tel que $\pi(l) > \pi(k) + p(k, l)$. Tant qu'il existe un tel arc (k, l), on ajoute cet arc à A. Si on crée ainsi un circuit, ce circuit est absorbant, on arrête. Sinon, on supprime de l'arborescence l'arc qui précédemment entrait en l, on diminue, pour tous les descendants de l dans l'arborescence (y compris l) la valeur de π de :

$$\pi(l) - [\pi(k) + p(k, l)].$$

Preuve de l'algorithme

À tout moment, la valeur de $\pi(z)$ pour un sommet z quelconque du graphe représente la longueur du chemin de r à z sur l'arborescence.

À chaque étape, la somme des $\pi(z)$ sur tous les sommets décroît strictement : on ne peut donc retrouver deux fois la même arborescence, ce qui prouve entre autres la finitude de l'algorithme.

Montrons que si l'adjonction de l'arc (k, l) à A crée un circuit, ce circuit est absorbant :

- pour tout arc (x, y) de ce circuit déjà dans A, on a $\pi(y) - \pi(x) = p(x, y)$;
- pour l'arc (k, l), on a $\pi(l) - \pi(k) > p(k, l)$.

Sommant ces relations pour tous les arcs du circuit, on trouve $0 > p(\text{circuit})$.

Enfin, supposons qu'il n'existe plus aucun arc (k, l) non dans l'arborescence, tel que $\pi(l) > \pi(k) + (k, l)$ et que, néanmoins, il existe un chemin C de r à un sommet x de longueur strictement inférieure à $\pi(x)$; on ne peut pas avoir $\pi(r) < 0$, car sinon r aurait été descendant dans l'arborescence, ce qui ne peut se faire. Il existe donc un arc (u, v) de C tel que la longueur de la portion de C allant de r à u soit de longueur au moins égale à $\pi(u)$ alors que la longueur de la portion de C allant de r à v est de longueur strictement inférieure à $\pi(v)$; cela implique :

$$\pi(v) > \pi(u) + p(u, v).$$

L'arc (u, v) devrait entrer dans l'arborescence, d'où une contradiction.

En conséquence, si l'algorithme s'arrête sans qu'il y ait eu création de circuit, l'arborescence donne bien les plus courts chemins de r à tous les autres sommets.

On laisse au lecteur le soin de trouver un algorithme permettant de tester, étant donné une arborescence A et un arc u, si l'adjonction de u à A crée un circuit.

VII.5. Plus courts chemins de tout sommet à tout sommet : cas général

Donnons tout d'abord une définition utile pour ce problème. Un graphe G est dit *fortement connexe* si, pour toute paire $\{x, y\}$ de sommets, il existe un chemin de x vers y et un chemin de y vers x.

Pour terminer ce chapitre, nous décrivons maintenant l'algorithme de Dantzig qui permet de résoudre le problème 3 dans un graphe fortement connexe, autrement qu'en résolvant successivement le problème 2 à partir de tous les sommets du graphe. On fait l'hypothèse que le graphe G est sans circuit absorbant.

Principe de l'algorithme

Ayant numéroté 1, 2, ..., n les sommets du graphe, on construit successivement les matrices $D^{(k)}$ dont l'élément de la i^{e} ligne et j^{e} colonne ($1 \leq i \leq k$, $1 \leq j \leq k$) représente la plus courte distance de i à j dans le sous-graphe engendré par les k premiers sommets. On fait l'hypothèse

simplificatrice que tous les arcs existent, quitte à ajouter les arcs manquants en leur attribuant un poids infini.

On pose $D^{(1)}(1, 1) = 0$. La construction de la matrice $D^{(k+1)}$ à partir de la matrice $D^{(k)}$ utilise les formules suivantes ($1 \leq k \leq n - 1$) :

- pour $1 \leq i \leq k$: $D^{(k+1)}(i, k+1) = \underset{j \in \{1,\ldots,k\}}{\text{Min}} \{D^{(k)}(i, j) + p(j, k+1)\}$;
- pour $1 \leq i \leq k$: $D^{(k+1)}(k+1, i) = \underset{j \in \{1,\ldots,k\}}{\text{Min}} \{p(k+1, j) + D^{(k)}(j, i)\}$;
- $D^{(k+1)}(k + 1, k + 1) = 0$;
- pour $1 \leq i \leq k$ et $1 \leq j \leq k$:
$$D^{(k+1)}(i, j) = \text{Min}\{D^{(k)}(i, j),\ D^{(k+1)}(i, k + 1) + D^{(k+1)}(k + 1, j)\}.$$

La matrice $D^{(n)}$ donne les plus courtes distances entre tout couple de sommets.

La preuve et le calcul de la complexité de cet algorithme font l'objet de l'exercice 7. Nous laissons au lecteur le soin d'améliorer l'algorithme pour :

- obtenir les plus courts chemins en plus des plus courtes distances ;
- traiter le cas général en détectant, s'il en existe, un circuit absorbant.

VII.6. Exercices

Exercice 1

Prouver les théorèmes suivants :

Théorème 1. *Une condition nécessaire et suffisante pour que le problème 1 ait une solution est que l'ensemble Y des sommets qui sont à la fois des descendants de x (c'est-à-dire des sommets situés sur des chemins d'origine x ou x lui-même) et des ascendants de y (c'est-à-dire des sommets situés sur des chemins d'extrémité y ou y lui-même) soit non vide et que le sous-graphe induit par Y soit sans circuit absorbant. Si ces deux conditions sont satisfaites, les solutions du problème 1 ne diffèrent des solutions du problème 1-bis que par l'éventuelle adjonction à celles-ci de circuits de longueur nulle.*

Théorème 2. *Une condition nécessaire et suffisante pour que le problème 2 ait une solution est que x soit racine du graphe et que le graphe ne contienne pas de circuit absorbant. Si ces deux conditions sont satisfaites, les solutions du problème 2 ne diffèrent des solutions du problème 2-bis que par l'éventuelle adjonction à celles-ci de circuits de longueur nulle.*

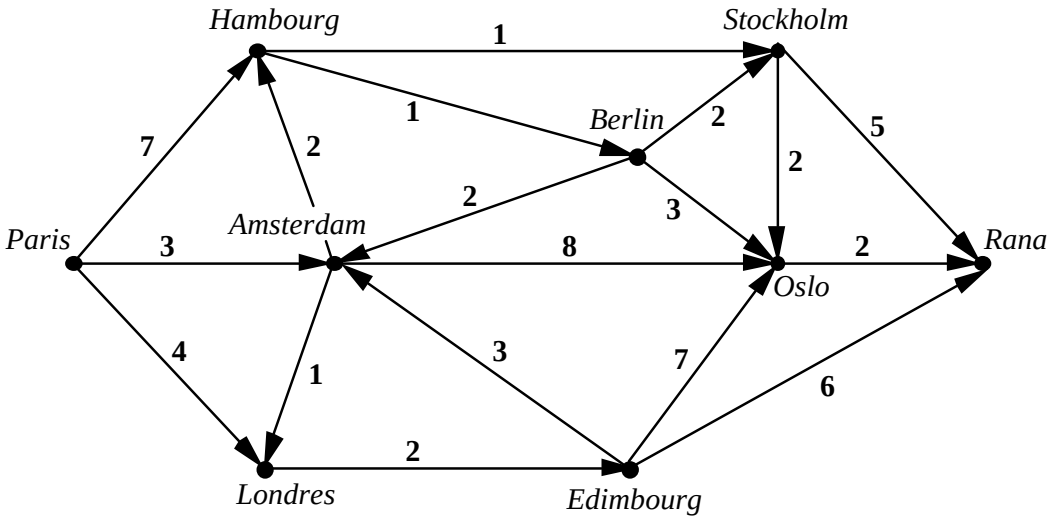
Théorème 3. *Une condition nécessaire et suffisante pour que le problème 3 ait une solution est que le graphe soit fortement connexe et que le graphe ne contienne pas de circuit absorbant. Si ces deux conditions sont satisfaites, les solutions du problème 3 ne diffèrent des solutions du problème 3-bis que par l'éventuelle adjonction à celles-ci de circuits de longueur nulle.*

Exercice 2

Un étudiant souhaite voir le soleil de minuit sur les fjords de Norvège. Il décide donc de se rendre à Rana, charmante petite ville située, comme chacun sait, à proximité du cercle polaire, sur la côte norvégienne. Après avoir fait le tour de quelques compagnies, il a recensé plusieurs connexions aériennes possibles lui permettant d'aller de Paris à Rana ; il les a représentées à l'aide du graphe suivant.

Les valuations portées sur les arcs correspondent au temps nécessaire (en heures) pour parcourir ces arcs, compte tenu des éventuels temps de transit pour les escales.

Notre voyageur ne possédant qu'un faible nombre de jours de vacances, nous allons l'aider à déterminer le chemin le plus rapide pour se rendre de Paris à Rana : que donne ici l'algorithme de Dijkstra ?



Exercice 3

Comment modifier l'algorithme de Dijkstra lorsque x n'est pas nécessairement racine du graphe pour trouver les plus courts chemins de x à tous les sommets que l'on peut atteindre par des chemins à partir de x ?

Exercice 4

Montrer, à l'aide d'un exemple, que, lorsque les poids peuvent être négatifs, l'algorithme de Dijkstra ne donne pas nécessairement les plus courts chemins. Que donne toujours l'application de cet algorithme ?

Exercice 5

a. Soit $G = (X, U)$ un graphe orienté sans circuit. Montrer que G possède un sommet de demi-degré intérieur nul.

b. On appelle numérotation topologique des sommets d'un graphe G une bijection num de X dans $\{1, 2, \ldots, |X|\}$ vérifiant : $\forall\ (x, y) \in U$, $num(x) < num(y)$. Donner une condition nécessaire et suffisante pour que G admette une numérotation topologique (pour montrer que la condition est suffisante, on établira un algorithme exhibant cette numérotation).

Exercice 6

L'histoire se passe dans un ensemble de sept pays imaginaires : PAYS-BAS, BELGIQUE, ALLEMAGNE, SUISSE, ITALIE, ESPAGNE et FRANCE. Dans chacun de ces pays, le beurre a un prix différent : certains de ces pays surproduisent, d'autres doivent importer. Le gouvernement de chaque pays décide de créer un organisme dont le rôle est d'aider les échanges. Pour cela, dans un premier temps, chaque pays passe un accord commercial avec chacun de ses voisins fixant des aides pour l'exportation vers des pays déficitaires et des taxes pour l'exportation vers des pays surproducteurs.

Ceci peut se résumer en deux tableaux, les coûts étant donnés en milliers de francs pour 20 tonnes transportées. Le tableau 1 est celui des taxes : T_{ij} représente la taxe payée pour passer du pays *i* au pays *j* ; le tableau 2 est celui des aides : le producteur reçoit A_{ij} s'il exporte de *i* vers *j*. S'il n'y a ni aide ni taxe, on supposera qu'il n'y a pas d'échange possible.

		P-B	B	A	S	I	F	E
Tableau des taxes	P-B							
	B	15		8			22	
	A	5			5		15	
	S						5	
	I				15		8	
	F							
	E						5	

		P-B	B	A	S	I	F	E
Tableau des aides	P-B		15	5				
	B							
	A		10					
	S			5		15		
	I							
	F		10	15	5	8		5
	E							

Par ailleurs le transport du beurre coûte lui-même un certain prix, précisé dans le tableau suivant et exprimé dans la même unité que pour les deux tableaux précédents.

Tableau des coûts		P-B	B	A	S	I	F	E
	P-B		2	4				
	B	2		3			2	
	A	4	3		3		4	
	S			3		2	5	
	I				2		6	
	F		2	4	5	6		6
	E						6	

a. Modéliser le problème sous forme d'un problème de plus court chemin dans un graphe orienté valué.

b. Résoudre le problème de l'exportation du beurre des PAYS-BAS vers l'ESPAGNE. Que remarque-t-on ?

c. Que se passe-t-il si une commission des 7 décide de réduire à 10 milliers de francs pour 20 tonnes l'aide à l'exportation du beurre de la FRANCE vers l'ALLEMAGNE ?

Exercice 7

Prouver l'algorithme de Dantzig et calculer sa complexité.

VIII. Parcours de graphes

Les parcours de graphe servent de base à bon nombre d'algorithmes. Ils n'ont pas une finalité intrinsèque, ce qui les distingue d'algorithmes déjà vus dans cet ouvrage, tels qu'un algorithme de recherche d'un arbre couvrant de poids minimum ; en revanche, ils doivent respecter certaines conditions, que nous verrons plus loin.

Le plus souvent, un parcours de graphe est un outil pour étudier une propriété globale du graphe : le graphe est-il connexe ? est-il biparti ? un graphe orienté est-il fortement connexe ? quels en sont les points d'articulation ? un certain flot est-il maximum ?...

Nous distinguerons deux types de parcours de graphes : les parcours « marquer-examiner » et les « parcours en profondeur ».

VIII.1. Définition d'un algorithme de « parcours de graphe »

Nous allons définir une sorte de « moule » appelé « parcours de graphe à partir d'un sommet donné ». C'est à partir de ce moule que l'on pourra insérer certaines instructions de façon à utiliser le parcours dans un objectif donné. Nous traiterons d'abord le cas orienté, puis le cas non orienté.

VIII.1.1. Cas orienté

Soit $G = (X, E)$ un graphe orienté et r un sommet de G.

Nous dirons que chaque sommet peut être ou non dans l'état « marqué ». Marquer un sommet, c'est passer son état de « non marqué » à « marqué ».

Nous utiliserons aussi l'expression « traverser un arc » : on ne traversera un arc (x, y) que lorsque x est marqué ; *traverser un arc (x, y)*, c'est regarder si son extrémité y est ou n'est pas marquée ; après avoir été traversé, un arc prend l'état « traversé ».

Un sommet r doit être choisi comme point de départ du parcours ; on dira alors qu'on effectue un parcours à partir de r.

L'algorithme définit une application de $X - \{r\}$ dans X, application qui associe à x le sommet que nous appellerons *père*(x).

Tout algorithme de parcours à partir de r peut être décrit de la façon suivante :

- au départ, aucun sommet n'est marqué
- marquer r
- tant qu'il existe un sommet marqué x et un arc (x, y) non traversé, alors choisir un tel arc

(x, y), le traverser et, si y n'est pas marqué :

- marquer le sommet y
- poser *père*(y) = x

Les différents algorithmes se distinguent par la façon de choisir l'arc à traverser.

En général, lorsqu'on utilise un algorithme de parcours, on accompagne le marquage des sommets d'autres instructions utiles à l'objectif poursuivi ; de même, lorsqu'on traverse un arc, certaines opérations peuvent être effectuées sur l'arc traversé.

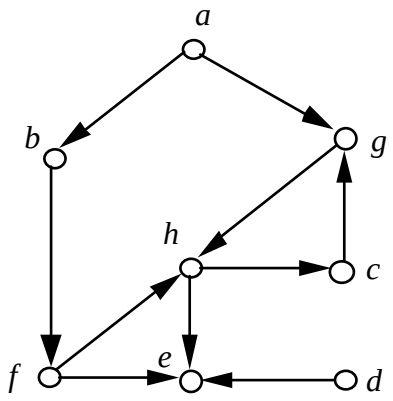
Proposition et définitions. *Si on note M l'ensemble des sommets marqués par un algorithme de parcours, le graphe A ayant M pour ensemble de sommets et pour arcs les arcs (père(x), x) pour x dans M – {r} est une arborescence de racine r. Cette arborescence s'appelle l'arborescence du parcours ; les arcs (père(x), x) s'appellent arcs arborescents.*

Preuve.— Il suffit de prouver que le nombre d'arcs de A est égal au nombre de sommets marqués moins un et que, pour tout sommet marqué x, il existe un chemin dans A de r à x. Pour prouver ces deux points, notons A_p le graphe ayant pour sommets l'ensemble M_p des sommets marqués après p passages dans la boucle « tant que » ($0 \leq p \leq |X|$) et pour arcs les arcs (*père*(y), y)) pour y dans $M_p - \{r\}$. On montre alors immédiatement par récurrence sur p que :

- le nombre d'arcs de A_p est égal au cardinal de M_p moins un ;
- pour tout sommet x de M_p, il existe un chemin dans A_p de r à x.

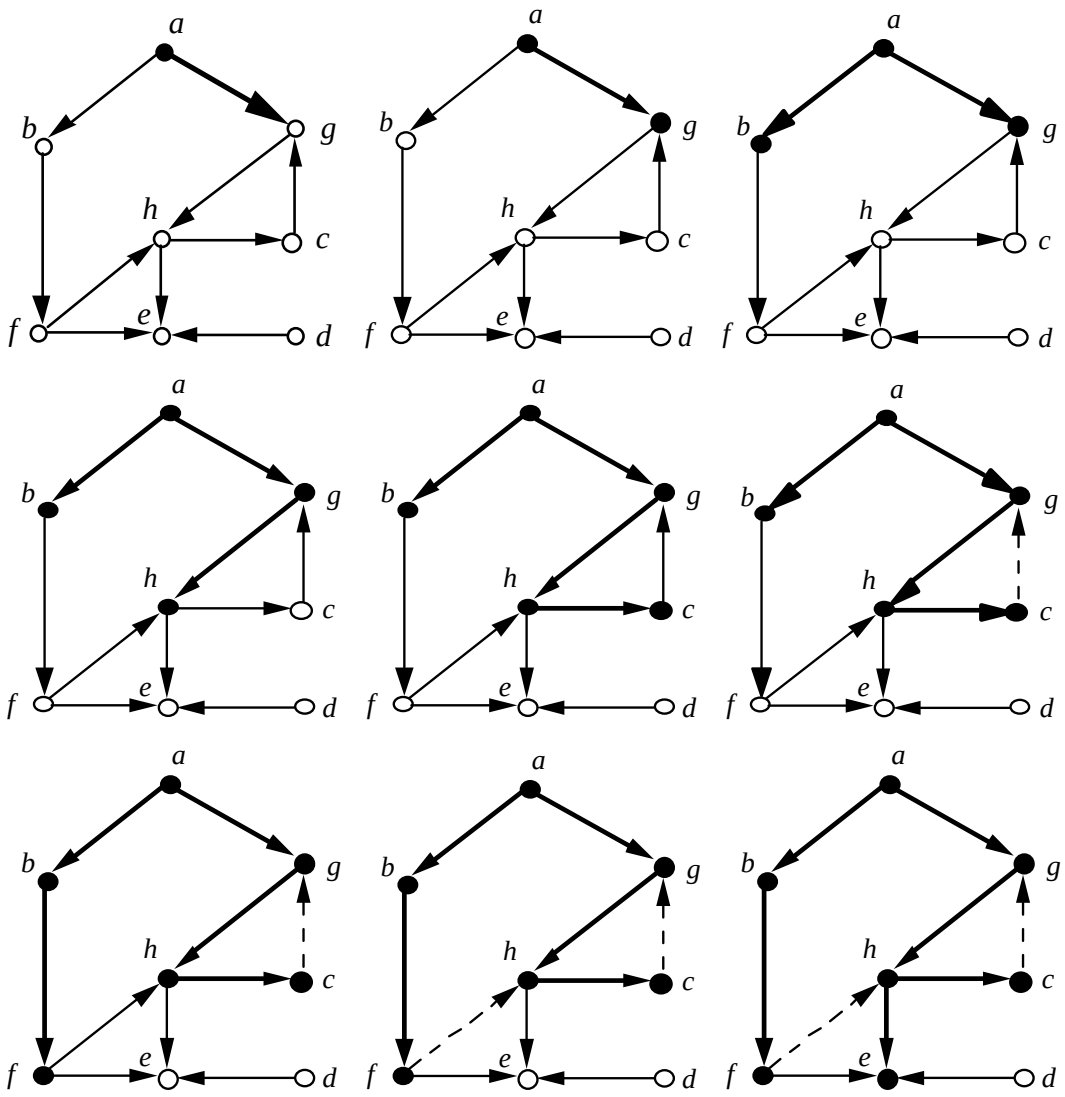
Nous allons prendre un exemple en insérant les instructions suivantes

- juste après avoir marqué le sommet, le colorier en noir
- juste après avoir traversé un arc
  - si son extrémité n'est pas encore marquée, alors, le mettre en gras (autrement dit, tracer en gras les arcs arborescents)
  - sinon, le tracer en pointillés



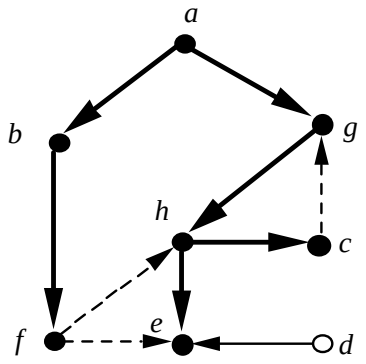
Le graphe considéré est ci-contre.

En utilisant un parcours à partir de a, on pourra avoir successivement, après le marquage de a puis après chaque passage dans la boucle « tant que » :



Le résultat final se trouve ci-contre.

On constate que l'ensemble des arcs en gras constituent bien une arborescence de racine *a*.



Enfin, on a clairement la propriété ci-dessous ; la preuve en est laissée au lecteur.

Propriété. *L'ensemble des sommets marqués durant le parcours à partir d'un sommet r est l'ensemble des sommets x pour lesquels il existe dans G un chemin de r à x.*

VIII.1.2. Cas non orienté

Soit $G = (X, E)$ un graphe non orienté. Dans un algorithme de parcours dans G, tout se passe comme si chaque arête $\{x, y\}$ du graphe avait été transformée en deux arcs (x, y) et (y, x) et qu'on faisait alors un parcours dans le graphe orienté obtenu. Néanmoins, de façon à éviter la transformation du graphe en un graphe orienté, on peut reprendre l'algorithme du cas orienté pour l'adapter au cas non orienté.

La notion de marquage des sommets est inchangée. Une arête peut être traversée à partir de l'une quelconque de ses extrémités ; son état passe alors de « non traversée » à « traversée ». L'algorithme de parcours à partir d'un sommet r devient :

- au départ, aucun sommet n'est marqué
- marquer r
- tant qu'il existe un sommet marqué x et une arête $\{x, y\}$ non traversée, alors choisir une telle arête $\{x, y\}$, la traverser et, si y n'est pas marqué :
  - marquer le sommet y
  - poser $père(y) = x$

On a :

Propriété et définitions. *Le graphe orienté A dont l'ensemble des sommets est l'ensemble M des sommets marqués par un algorithme de parcours à partir d'un sommet r et l'ensemble des arcs (père(x), x) pour x dans M – {r} est une arborescence de racine r. Cette arborescence s'appelle l'arborescence du parcours ; les arcs (père(x), x) s'appellent arcs arborescents.*

Propriété. *L'ensemble des sommets marqués durant le parcours à partir d'un sommet r est l'ensemble des sommets x pour lesquels il existe une chaîne entre r et x.*

Cette dernière propriété montre de façon évidente qu'un parcours permet de déterminer les composantes connexes d'un graphe et, bien sûr, de savoir si un graphe est connexe ou non. Pour ce faire, on lance un parcours à partir d'un sommet quelconque ; lorsque ce parcours est terminé, l'ensemble des sommets marqués constitue alors les sommets d'une composante connexe du graphe. Si on a pris le soin de compter le nombre de sommets marqués, en comparant ce nombre à l'ordre du graphe, on sait immédiatement si le graphe est connexe ou non. S'il existe des sommets non marqués, on reprend un parcours à partir d'un sommet non marqué et on marque dans ce deuxième parcours les sommets d'une deuxième composante connexe, et ainsi de suite.

VIII.1.3. Complexité

Si le graphe est codé par la liste des voisins (des successeurs dans le cas orienté) de chaque sommet, un algorithme de parcours de graphe a alors une complexité au plus de l'ordre du nombre m d'arcs ou d'arêtes du graphe ; l'algorithme est en $O(m)$. Si le graphe est codé par sa matrice d'adjacence, l'algorithme est en $O(n^2)$.

VIII.2. Les parcours « marquer-examiner »

VIII.2.1. Généralités

Soit $G = (X, E)$ un graphe orienté et $r \in X$. Nous aurons besoin ici d'une liste dite *d'attente* et notée L, qui est au départ vide et qui contiendra des sommets.

En reprenant la terminologie du paragraphe précédent, on définit la procédure « examiner » de la façon suivante. Examiner un sommet x, c'est :

- pour tout arc (x, y) :
  - traverser l'arc (x, y)
  - si y n'est pas marqué, alors
    - marquer y
    - poser $père(y) = x$
    - mettre y dans la liste d'attente L

Un parcours du type « marquer-examiner » s'énonce :

- marquer r et le mettre dans la liste d'attente L
- tant que la liste d'attente n'est pas vide :
  - en retirer un sommet x
  - examiner x

Selon l'ordre dans lequel les sommets sont choisis et retirés de la liste d'attente, on obtient un parcours différent. Si on gère la liste d'attente en file, on obtient ce qu'on appelle un parcours en largeur. Ce parcours présente l'avantage de marquer les sommets par « couches », d'abord la « couche » 0, constituée du sommet de départ r, puis la couche 1 constituée des sommets à distance 1 de r (la distance étant calculée par rapport au nombre d'arcs), puis la couche 2 constituée des sommets à distance 2 de r et ainsi de suite… On remarque que les arcs $(père(x), x)$ $(x \neq r)$ forment une arborescence des plus courts chemins (par rapport au nombre d'arcs) de r à tous les sommets du graphe accessibles depuis r (ces arcs sont les arcs dessinés en gras dans l'exemple ci-dessous) ; ce résultat se prouve très aisément par récurrence.

Si on gère la liste d'attente en pile, on obtient un parcours proche du parcours en profondeur décrit plus loin.

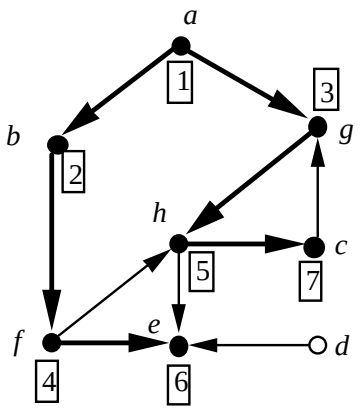
Nous allons illustrer ces deux parcours sur l'exemple considéré dans le paragraphe ci-dessus. Nous donnons les arborescences des parcours et la suite des états de la file d'attente.

VIII.2.2. Résultat d'un parcours « marquer-examiner » selon une file : le parcours en largeur

La file d'attente est successivement, au départ puis après chaque examen de sommets, et en supposant que les arcs issus d'un sommet sont traversés selon l'ordre alphabétique de leurs extrémités :

a, bg, gf, fh, he, ec, c, Ø

Les numéros indiquent l'ordre dans lequel les sommets ont été marqués.

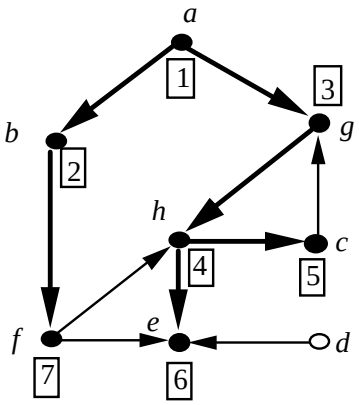


VIII.2.3. Résultat d'un parcours « marquer-examiner » suivant une pile

La pile d'attente est, au départ puis successivement après chaque examen de sommets, et en supposant que les arcs issus d'un sommet sont considérés selon l'ordre alphabétique de leurs extrémités :

$$[a],\ \begin{bmatrix} g \\ b \end{bmatrix},\ \begin{bmatrix} h \\ b \end{bmatrix},\ \begin{bmatrix} e \\ c \\ b \end{bmatrix},\ \begin{bmatrix} c \\ b \end{bmatrix},\ [b],\ [f],\ \varnothing$$

Les numéros indiquent l'ordre dans lequel les sommets ont été marqués



VIII.3. Les parcours en profondeur

VIII.3.1. Le cas orienté

Généralités

Un parcours en profondeur à partir d'un sommet *r*, appelé en anglais *Depth First Search*, est noté *DFS*(*r*). Ce parcours peut être décrit de manière récursive ou itérative ; nous en donnerons les deux versions. Pour définir le parcours *DFS*(*r*) de façon itérative, il faut, à tout moment de l'algorithme, qu'un sommet courant soit défini. Au départ, le sommet courant est le sommet *r* et seul le sommet *r* est marqué. L'algorithme peut alors s'écrire, de façon itérative :

- Tant que l'algorithme n'est pas terminé :
  - noter x le sommet courant
  - s'il existe un arc (x, y) non encore traversé, alors
    - traverser l'arc (x, y)
    - si y n'est pas encore marqué, alors
      marquer y
      poser $père(y) = x$
      « avancer » en y, c'est-à-dire poser que le sommet courant est y
  - sinon
    - si $x \neq r$, reculer au père de x, c'est-à-dire poser que le sommet courant devient $père(x)$
    - sinon, l'algorithme est terminé

De façon récursive, faire un parcours $DFS(x)$, c'est :

- marquer le sommet x
- pour tout arc (x, y), traverser l'arc (x, y) et, si y n'a pas encore été marqué :
  - poser $père(y) = x$
  - appliquer $DFS(y)$

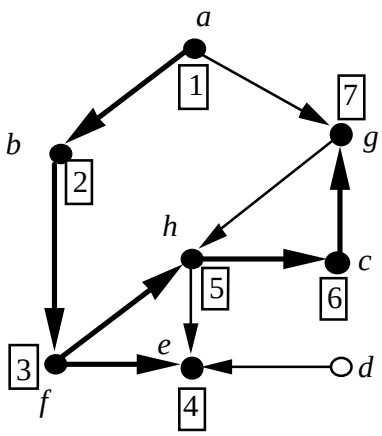
Pour faire un parcours en profondeur à partir de r avec l'algorithme écrit récursivement, il suffit alors de poser qu'aucun sommet n'est marqué et d'appeler $DFS(r)$.

EXEMPLE

Reprenons le graphe donné en exemple dans ce chapitre pour effectuer le parcours $DFS(a)$ en considérant les voisins extérieurs d'un sommet donné selon l'ordre alphabétique.

L'arborescence du parcours est tracée en gras.

Les sommets ont été numérotés dans l'ordre où ils sont marqués.



Définitions : numérotation préfixe, numérotation postfixe

Lorsqu'on effectue un parcours en profondeur, on peut associer d'une part ce qu'on appelle la *numérotation préfixe* des sommets, d'autre part la *numérotation postfixe* des sommets, appelée aussi *numérotation suffixe*.

Ces numérotations sont définies ci-dessous.

On initialise deux variables entières *pré* et *post* à 1. Attribuer son numéro préfixe (respectivement postfixe) à un sommet, c'est lui affecter le numéro *pré* (respectivement *post*) puis incrémenter la variable *pré* (respectivement *post*) de 1.

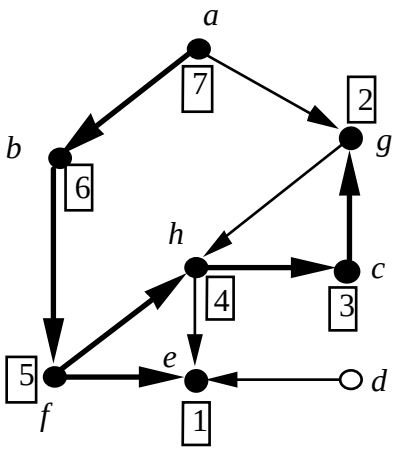
Dans la version itérative, on attribue le numéro préfixe d'un sommet au moment où on marque ce sommet. On attribue le numéro postfixe d'un sommet juste avant de reculer de ce sommet.

Dans la version récursive, si on veut déterminer les numérotations préfixe et postfixe, on récrit *DFS*(*x*) de la façon suivante :

- marquer le sommet *x* et lui attribuer son numéro préfixe
- pour tout arc (*x*, *y*), traverser l'arc (*x*, *y*) et, si *y* n'a pas encore été marqué :
  - poser *père*(*y*) = *x*
  - appeler *DFS*(*y*)
- attribuer à *x* son numéro postfixe

La numérotation préfixe obtenue par un parcours *DFS*(*a*) dans l'exemple du paragraphe précédent en respectant l'ordre alphabétique est la numérotation qui a été indiquée ci-dessus.

La numérotation postfixe obtenue par le même parcours est indiquée ci-contre.



VIII.3.2. Le cas non orienté

Généralités

Nous reprenons plus rapidement que ci-dessus ce que représente un parcours en profondeur dans un graphe non orienté à partir d'un sommet *r*, parcours noté *DFS*(*r*). Nous n'en donnons qu'une écriture récursive.

Au départ, aucun sommet n'est marqué ; on appelle *DFS*(*r*).

Faire un parcours *DFS*(*x*) à partir d'un sommet *x*, c'est :

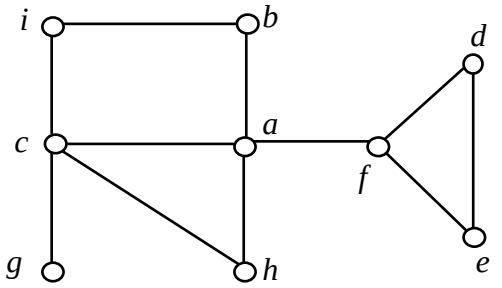
- Marquer le sommet *x*
- Pour toute arête {*x*, *y*}, si l'arête {*x*, *y*} n'est pas traversée, alors
  - traverser l'arête {*x*, *y*}
  - si *y* n'est pas marqué, alors
    - poser *père*(*y*) = *x*
    - faire un parcours *DFS*(*y*)

REMARQUE

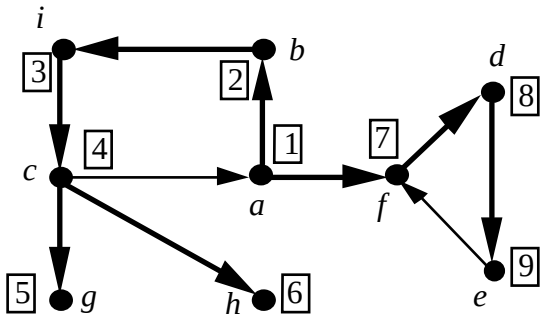
On pourrait aussi, si nécessaire, considérer que l'on traverse chaque arête deux fois, une fois à partir de chacune de ses extrémités.

EXEMPLE

Considérons l'exemple ci-contre, dans lequel on se propose de faire un parcours à partir du sommet *a* en respectant, lorsqu'un choix se présente, l'ordre alphabétique.



Nous obtenons le résultat suivant ; les sommets y sont numérotés selon l'ordre où ils ont été marqués. L'arborescence du parcours a été tracée en gras. Les arcs non arborescents ont aussi été orientés ; nous verrons plus loin comment est déterminé le sens de cette orientation.

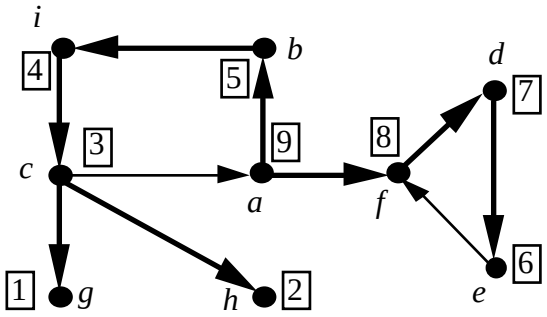


Définitions : arborescence du parcours DFS, numérotations préfixe et postfixe, arcs arrière

Lorsqu'on numérote les sommets dans l'ordre où on les marque, on obtient (comme sur l'exemple ci-dessus et de façon analogue au cas orienté) ce qu'on appelle la *numérotation préfixe*.

On appelle *numérotation postfixe* ou *suffixe* la numérotation obtenue (de façon analogue au cas orienté) en numérotant les sommets au fur et à mesure du parcours, le numéro étant attribué à un sommet *x* au moment où l'on termine le parcours *DFS*(*x*).

Sur l'exemple ci-dessus, la numérotation postfixe est donc :



Dans le parcours en profondeur, il est souvent utile d'orienter toutes les arêtes du graphe, ce qu'on fait au moment où on les traverse, l'arc ayant alors pour origine le sommet à partir duquel est faite la traversée. Cette orientation correspond à l'orientation choisie pour les arcs arborescents ; c'est aussi l'orientation retenue sur l'exemple ci-dessus.

Une arête qui a été orientée en un arc arborescent s'appelle *arête arborescente*. Un arc obtenu en orientant une arête non arborescente s'appelle un *arc arrière.* Toute arête conduit à un arc arborescent ou à un arc arrière.

Lemme. *Les arcs arrière vont d'un sommet x à un des ancêtres de x autre que son père dans l'arborescence. En conséquence, les arêtes non arborescentes joignent toutes un sommet à un de ses ascendants dans l'arborescence.*

Preuve.— Considérons un arc arrière (x, y). Au moment où, de x, on a considéré l'arête $\{x, y\}$, $DFS(y)$ avait été commencé (puisque y était déjà marqué) et n'était pas terminé (puisque l'arête $\{x, y\}$ n'avait pas encore été traversée) ; $DFS(x)$ a donc été appelé pendant $DFS(y)$; cela implique alors que x est descendant de y dans l'arborescence du parcours.

VIII.4. Applications des parcours en profondeur

VIII.4.1. Application à la détermination des composantes fortement connexes

Définition. *Étant donné un graphe orienté $G = (X, E)$, on définit une relation d'équivalence R sur X par : x R y si et seulement s'il existe un chemin de x à y et il existe un chemin de y à x. Les classes d'équivalence de x pour cette relation s'appellent les composantes fortement connexes de G.*

L'algorithme décrit et prouvé ci-dessous, dérivé de *DFS*, permet de déterminer les *composantes fortement connexes* d'un graphe orienté.

Si le graphe n'admet qu'une seule composante fortement connexe, on dit alors qu'il est *fortement connexe* ; autrement dit, un graphe est fortement connexe si, pour toute paire $\{x, y\}$ de sommets, il existe un chemin de x à y et un chemin de y à x.

Principe de l'algorithme

Phase 1. Choisir un sommet initial, lancer *DFS* à partir de ce sommet et numéroter les sommets en ordre postfixe. Si, lors de l'arrêt de la recherche, il reste des sommets non numérotés, relancer *DFS* à partir de tels sommets jusqu'à ce que tous les sommets soient numérotés : on obtient ainsi une forêt d'arborescences (c'est-à-dire une union disjointe de graphes qui sont des arborescences). Les sommets de la première arborescence sont numérotés de 1 à p, ceux de la suivante de $p + 1$ à q et ainsi de suite.

Phase 2. Inverser l'orientation de tous les arcs. Le nouveau graphe obtenu est noté G^{-}. Relancer *DFS*, sur G^{-}, en commençant par le sommet de plus grand numéro par rapport à la numérotation postfixe précédente. Lorsque la recherche s'arrête, les sommets marqués constituent une composante fortement connexe de G. S'il reste des sommets non numérotés, relancer *DFS* à partir du sommet non marqué de plus grand numéro : les sommets marqués lors de ce deuxième parcours constituent à nouveau une composante fortement connexe et ainsi de suite.

Preuve de l'algorithme

Nous utiliserons le lemme suivant, dont la preuve est élémentaire : soient $A_1, A_2, \ldots, A_k$ les différentes arborescences de parcours obtenues lorsqu'on applique l'algorithme précédent. Si (x, y) est un arc du graphe et si x est dans A_i, alors y est dans A_j avec $j \leq i$. Autrement dit, si on

considère, le long d'un chemin du graphe, les indices des arborescences auxquelles appartiennent ces sommets, ces indices forment une suite décroissante (au sens large).

Dire que x et y appartiennent à la même composante fortement connexe de G est équivalent à dire que, dans G et dans G^{-}, il existe un chemin de x à y et un autre de y à x.

Supposons que x et y appartiennent à une même composante fortement connexe de G. Nous pouvons supposer, sans perte de généralité, que, dans la phase 2, x a été marqué avant y. Supposons que x appartienne à la i^{e} arborescence obtenue dans la phase 2 et y à la j^{e} arborescence. x ayant été marqué avant y, $i \leq j$; par ailleurs, puisqu'il existe un chemin de x à y, d'après le lemme ci-dessus, $i \geq j$; d'où $i = j$: x et y appartiennent à une même arborescence *DFS* de la phase 2.

Pour prouver la réciproque, x étant un sommet quelconque, notons r la racine de l'arborescence *DFS* de G^{-} à laquelle appartient x dans la phase 2 ; montrons que x et r appartiennent à une même composante fortement connexe de G ; ce résultat entraînera que deux sommets appartenant dans la phase 2 à une même arborescence de la forêt couvrante de G appartiennent aussi à une même composante connexe de G. On peut supposer que $x \neq r$. Dans G^{-}, il existe un chemin de r à x, donc dans G il existe aussi un chemin, noté $x_0 = x$, x_1, …, $x_p = r$, de x à r. Il reste à montrer qu'il existe aussi dans G un chemin de r à x. Remarquons, à toutes fins utiles, que, d'après le choix de r en tant que racine, le numéro postfixe de r est supérieur à ceux des sommets x_i ($0 \leq i < p$).

Supposons tout d'abord que r a été marqué dans la phase 1 avant x. D'après la remarque ci-dessus, le numéro postfixe de r dans la phase 1 est supérieur à celui de x ; le parcours *DFS*(r) dans G a ainsi commencé avant le parcours *DFS*(x) mais en revanche il s'est terminé après : cela indique que x est descendant de r dans cette arborescence et il y a donc dans G un chemin de r à x.

Supposons maintenant que x a été marqué dans la phase 1 avant r. D'après le lemme ci-dessus et puisque x a été marqué avant r, c'est que tous les sommets du chemin appartiennent à la même arborescence couvrante de G. Soit i ($0 \leq i < p$) un indice tel que x_i ait été marqué avant r avec $x_{i+1} = r$ ou bien x_{i+1} marqué après r ; *DFS*(x_i) est lancé dans G à partir de x_i, alors que ni r ni x_{i+1} ne sont marqués ; lorsque *DFS*(x_i) se termine, x_{i+1}, étant successeur de x_i, a nécessairement été marqué, ce qui entraîne que r a aussi été marqué durant ce parcours : r est donc descendant de x_i dans l'arborescence construite durant *DFS*(x_i), ce qui entraîne que le numéro postfixe de x_i est strictement supérieur à celui de r. D'où une contradiction.

Complexité de l'algorithme

Si nous définissons le graphe orienté par sa matrice d'adjacence, chaque phase de l'algorithme est en $O(n^2)$, c'est donc aussi le cas de l'algorithme tout entier.

Si nous définissons le graphe par le tableau des listes des successeurs des sommets, chaque phase de l'algorithme (et donc l'algorithme tout entier) est en $O(n + m)$.

VIII.4.2. Application à la détermination des sommets d'articulation d'un graphe non orienté

Définitions. *On appelle sommet d'articulation d'un graphe non orienté un sommet dont la suppression augmente le nombre de composantes connexes du graphe. On peut également caractériser un tel point a en disant qu'il existe deux sommets x et y tels que toute chaîne entre x et y passe par a. Un graphe connexe est dit 2-connexe si et seulement si soit il est réduit à une arête, soit il n'a pas de sommet d'articulation.*

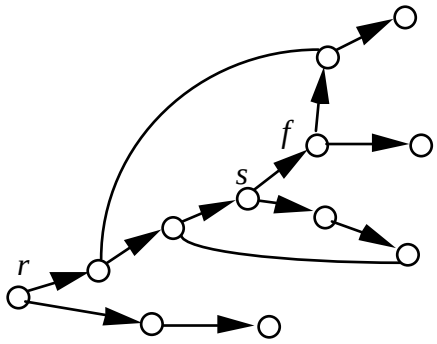
Nous allons prouver le théorème suivant :

Théorème. *Soit G = (X, E) un graphe connexe, non orienté, et soit A une arborescence DFS de G. Un sommet s est un sommet d'articulation de G si et seulement si :*

- *ou bien s est la racine de A et s possède au moins deux fils dans A ;*
- *ou bien s n'est pas la racine de A et, pour un fils f de s dans A, il n'y a pas d'arc arrière entre n'importe quel descendant de f (y compris f) et un ancêtre propre de s.*

Preuve.— D'après le lemme du paragraphe VIII.3.2, aucune arête ne joint deux « branches » différentes de *A* issues d'un sommet donné. En conséquence, il est clair que la racine de l'arborescence est un sommet d'articulation si et seulement s'il existe plusieurs branches issues de la racine, c'est-à-dire si et seulement si la racine admet plusieurs fils.

Soit *s* un sommet qui ne soit pas la racine ; on suppose que *s* admet un fils *f* tel qu'il n'y ait pas d'arc arrière entre n'importe quel descendant de *f* (y compris *f*) et un ancêtre propre de *s*. Toute chaîne issue de *f* ne pourra quitter la branche de *A* de racine *f* qu'en empruntant une arête allant d'un sommet de cette branche à un ancêtre propre de *f* ; s'il n'existe pas d'arc arrière entre n'importe quel descendant de *f* et un ancêtre propre de *s*, cette chaîne doit contenir l'arête {*s*, *f*} : *s* est sommet d'articulation.

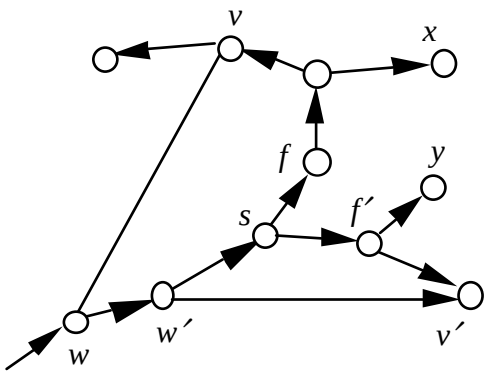


Soit maintenant un sommet *s* autre que la racine de *A* qui soit un point d'articulation ; soient *x* et *y* deux sommets qui sont dans des composantes connexes différentes de *G* – *s*, ce qui implique que toute chaîne entre *x* et *y* passe par *s*. Si ni *x* ni *y* n'étaient descendants de *s* dans *A*, on trouverait une chaîne de *A* entre *x* et *y* évitant *s*, contrairement à la définition de *x* et *y*. Supposons donc *x* descendant de *s*. Appelons *f* le fils de *s* qui est ancêtre de *x* dans *A* (on peut avoir *f* = *x* si *x* est fils de *s*). S'il n'y a pas d'arête entre *f*, ou ses descendants, et un ancêtre propre de *s*, on a exactement ce qu'on veut. Supposons qu'il existe une arête entre, disons *v*, descendant de *f*, et *w*, ancêtre propre de *s*.

De deux choses l'une : ou bien *y* n'est pas descendant de *s* ou bien il l'est.

Traitons d'abord le cas où *y* est descendant de *s*. *y* ne peut être descendant de *f*, sinon on aurait dans *A* une chaîne entre *x* et *y*, passant éventuellement par *f*, et évitant *s*.

Appelons alors *f´* le fils de *s* qui est ancêtre de *y* dans *A*. S'il n'existe pas d'arête entre *f´* et un ancêtre propre de *s* dans *A*, on a exactement ce qu'on veut. Supposons donc qu'il existe une arête entre *v´*, descendant de *f´*, et *w´* ancêtre propre de *s*.



On peut extraire de la chaîne *x … f … v w … w´v´… f´… y* (où *a … b* indique que l'on suit la chaîne de *A* limitée par les 2 points *a* et *b*) une chaîne entre *x* et *y* qui évite *s*, contrairement à la définition de *s*.

Le cas où *y* n'est pas descendant de *s* est plus simple : considérons en effet *f* défini comme précédemment. S'il existait une arête entre *f* ou un de ses descendants *v* et un ancêtre propre *w* de *s*, on pourrait trouver une chaîne extraite de *x … f … v w … y* entre *x* et *y* , évitant *s*, ce qui est impossible. Il n'y a donc pas de telles arêtes , ce qui est exactement ce qu'on voulait. →

Nous allons voir maintenant comment on peut mettre en œuvre ce théorème et l'utiliser pour trouver les sommets d'articulation d'un graphe connexe.

Nous avons besoin, pour déterminer les sommets d'articulation, de deux numérotations. L'une d'elles est la numérotation préfixe définie précédemment. L'autre, appelée *numérotation basse*, est définie comme suit : *basse*(*v*) est la plus petite valeur préfixe d'un sommet dans un chemin qui commence à *v* et qui soit la concaténation d'un chemin d'origine *v* dans l'arborescence *DFS* et d'un arc arrière.

On peut remarquer qu'on a alors la relation :

basse(*v*) = Min{*préfixe*(*v*), *préfixe*(*z*) pour tout *z* tel que (*v*, *z*) soit un arc arrière, *basse*(*u*) pour tout fils *u* de *v* dans l'arborescence *DFS*}.

Pour déterminer *basse*(*v*) pour tout sommet *v*, nous considérons les sommets en ordre postfixe. Pour notre exemple, nous trouvons successivement :

basse(*g*) = 5 puisque *préfixe*(*g*) = 5

basse(*h*) = 1 puisque *préfixe*(*a*) = 1

basse(*c*) = 1 puisque *préfixe*(*a*) = 1

basse(*i*) = 1 puisque *basse*(*c*) = 1

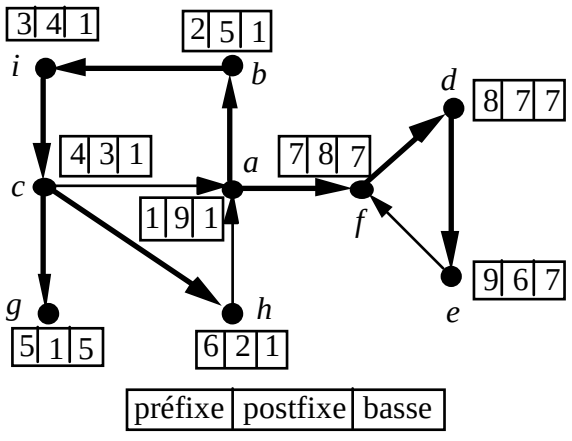
basse(*b*) = 1 puisque *basse*(*i*) = 1

basse(*e*) = 7 puisque *préfixe*(*f*) = 7

basse(*d*) = 7 puisque *basse*(*e*) = 7

basse(*f*) = 7 puisque *préfixe*(*f*) = 7

basse(*a*) = 1 puisque *préfixe*(*a*) = 1



Sur le schéma ci-dessus, nous avons fait figurer à côté de chaque sommet les trois numéros : *préfixe*, puis *postfixe*, puis *basse*.

Nous pouvons alors caractériser les sommets d'articulation en faisant la même distinction que dans le théorème précédent.

Théorème.

1. Le point de départ de DFS est sommet d'articulation si et seulement s'il a au moins deux fils dans l'arborescence.

2. Un sommet s autre que la racine est un sommet d'articulation si et seulement s'il a au moins un fils f dans l'arborescence tel que :

basse(f) $\geq$ *préfixe(s).*

Preuve

1. Cette condition est identique à celle du théorème précédent.

2. Dire que *basse*(*f*) $\geq$ *préfixe*(*s*) signifie qu'il n'existe aucune arête entre un descendant de *f* (y compris *f*) et un ancêtre propre de *s* dans l'arborescence. Le résultat est alors obtenu immédiatement à partir du théorème précédent.

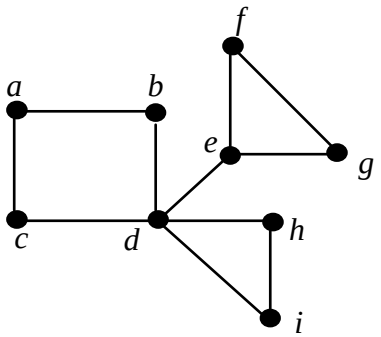
En appliquant ce dernier théorème à l'exemple choisi dans ce paragraphe, on voit que les points d'articulation du graphe sont :
a parce qu'il est racine de l'arborescence du parcours et qu'il a plusieurs fils ;
f parce que son fils *d* vérifie : *basse*(*d*) $\geq$ *préfixe*(*f*) ;
c parce que son fils *g* vérifie : *basse*(*g*) $\geq$ *préfixe*(*c*).

VIII.4.3. Application à la détermination des composantes 2-connexes d'un graphe non orienté

Définitions. *Soit G un graphe non orienté. On dit que G est 2-connexe s'il est connexe et qu'il n'admet aucun sommet d'articulation. On appelle composante 2-connexe tout sous-graphe 2-connexe maximal pour l'inclusion.*

Par exemple, dans le graphe ci-contre, il y a quatre composantes 2-connexes qui sont les sous-graphes induits par les sommets :

composante 1 : {*a*, *b*, *c*, *d*}
composante 2 : {*d*, *e*}
composante 3 : {*e*, *f*, *g*}
composante 4 : {*d*, *h*, *i*}



Nous allons expliciter un algorithme basé sur un parcours *DFS* qui permet de déterminer les composantes 2-connexes d'un graphe non orienté. Plus exactement, cet algorithme permet de donner la liste des arêtes de chaque composante 2-connexe du graphe.

On définit une variable entière *pré* qu'on initialise à 1, qui servira à construire la numérotation préfixe des sommets, numérotation que construit l'algorithme. L'algorithme utilise par ailleurs la numérotation *basse*. Il est aussi nécessaire d'utiliser une pile, au départ

vide, qui servira à empiler des arêtes.

Au départ, aucun sommet n'est marqué. On appelle *Parcours*(*r*), où *r* est un sommet quelconque, l'algorithme récursif décrit ci-dessous.

Effectuer *Parcours*(*x*), c'est :

- marquer (*x*)
- poser *préfixe*(*x*) = *pré*, puis incrémenter la variable *pré* de 1
- poser *basse*(*x*) = *préfixe*(*x*)
- pour tout voisin *y* de *x*, si l'arête {*x*, *y*} n'a pas été traversée (ce qui peut être caractérisé par le fait que soit *y* n'est pas marqué, soit à la fois *y* n'est pas le père de *x* et le préfixe de *y* est plus petit que celui de *x*), alors :
    - empiler cette arête
    - si *y* est déjà marqué, alors
        - poser *basse*(*x*) = minimum(*basse*(*x*), *préfixe*(*y*))
    - sinon
        - poser *père*(*y*) = *x*
        - appeler *Parcours*(*y*)
        - si *basse*(*y*) ≥ *préfixe*(*x*), alors
            dépiler toutes les arêtes de la pile empilées depuis l'arête {*x*, *y*}, y compris cette dernière : ces arêtes sont les arêtes d'une composante 2-connexe
        - sinon poser *basse*(*x*) = minimum(*basse*(*x*), *basse*(*y*))

Nous laissons au lecteur le soin de prouver cet algorithme. Il permet de déterminer les composantes 2-connexes d'un graphe avec une complexité en O(*m*), *m* étant le nombre d'arêtes du graphe.

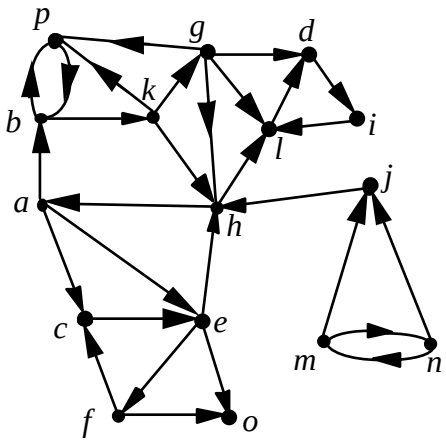
VIII.5. Exercices

Exercice 1

On cherche à déterminer si un graphe *G* = (*X*, *E*) non orienté connexe est biparti, c'est-à-dire tel qu'il existe une bipartition des sommets en deux classes sans arête entre deux sommets de la même classe. Écrire et prouver un algorithme utilisant un parcours de graphe qui résolve ce problème.

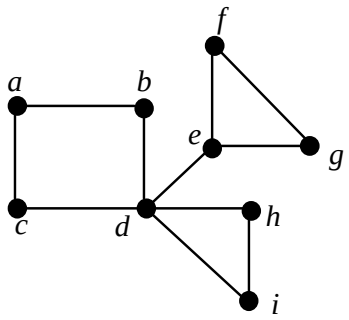
Exercice 2

Utiliser un algorithme explicité dans ce chapitre pour déterminer les composantes fortement connexes du graphe ci-contre :



Exercice 3

Utiliser l'algorithme explicité dans ce chapitre pour déterminer les composantes 2-connexes du graphe cité en exemple, que nous reproduisons ci-contre :



Exercice 4

a. Soit *G* un graphe orienté de racine *a*. Montrer que *G* possède un circuit si et seulement si, dans un parcours en profondeur à partir de *a*, on rencontre au moins un arc arrière, c'est-à-dire un arc joignant un sommet de l'arborescence du parcours à un de ses ancêtres.

b. Soit *G* un graphe non orienté connexe. Montrer que *G* possède un cycle si et seulement si, dans un parcours en profondeur à partir d'un sommet quelconque, on rencontre au moins un arc arrière.

IX. Flot maximum et coupe de capacité minimum

IX.1. Introduction, théorème du flot et de la coupe

IX.1.1. Introduction

La recherche d'un flot de valeur maximum dans un réseau a des applications directes, par exemple en termes de trafic, qu'il s'agisse d'un réseau de circulation ou d'un réseau téléinformatique, mais il en a bien d'autres : c'est ainsi que nous étudierons dans le chapitre « Applications de la théorie des flots » la détermination d'un couplage de cardinal maximum dans un graphe biparti, modèle par exemple de problèmes d'affectation, ou encore la détermination de la connectivité d'un graphe, paramètre très lié à la résistance aux pannes dans un réseau. La ressemblance entre l'énoncé du théorème liant valeur maximum d'un flot et capacité minimum d'une coupe et le théorème de la dualité en programmation linéaire n'a rien de fortuit : en effet, le problème du flot de valeur maximum est un problème de programmation linéaire dont le problème de la coupe est le dual. Cependant, dans cet ouvrage, nous donnons une solution graphique à ce problème et non une solution de type simplexe.

Ce chapitre comprend deux parties. La première est consacrée aux résultats théoriques liant le problème du flot maximum à celui de la coupe de capacité minimum. La seconde présente l'algorithme de Ford et Fulkerson, qui permet de résoudre ces deux problèmes simultanément.

IX.1.2. Définitions, notations et problèmes

Dans ce chapitre, on considère un *réseau*, c'est-à-dire, pour les graphistes, un graphe orienté $G = (X, U)$ pour lequel on a défini une application *capacité*, notée c, de U dans $]0, +\infty[$, et choisi deux sommets privilégiés, s, comme *source*, et p, comme *puits*.

Soit S un ensemble de sommets contenant s et ne contenant pas p : on dit que S *sépare* s de p. On note $\overline{S}$ le complémentaire de S dans X, et $(S, \overline{S})$ l'ensemble des arcs dont l'origine est dans S et l'extrémité dans $\overline{S}$. L'ensemble $(S, \overline{S})$ s'appelle une *coupe* séparant la source du puits.

On appelle *flot* dans le réseau une application f de U dans R qui vérifie les deux propriétés suivantes :

- pour tout arc u, $0 \leq f(u) \leq c(u)$,
- pour tout sommet x autre que p et s, il y a conservation du flot en x, ce qui signifie que le flot total *entrant* en x (somme des valeurs de $f(u)$ pour tous les arcs u d'extrémité finale x) est égal au flot total *sortant* de x (somme des valeurs de $f(u)$ pour tous les arcs u d'origine x).

On appelle *flux* d'un arc u la quantité $f(u)$ qu'il porte.

On définit la *valeur du flot f*, que l'on note *val*(*f*), par l'une des trois expressions suivantes (leur égalité, conséquence élémentaire de la conservation, est prouvée dans le paragraphe suivant) :

- flot total quittant s moins flot total entrant en s (valeur à la source) ;
- flot total entrant en p moins flot total quittant p (valeur au puits) ;
- somme des flux sur les arcs de $(S, \bar{S})$ (c'est-à-dire ayant leur origine dans S et leur extrémité dans $\bar{S}$), que l'on notera $f(S, \bar{S})$, diminuée de la somme des flux sur les arcs de $(\bar{S}, S)$, que l'on notera $f(\bar{S}, S)$.

Par ailleurs, on définit la *capacité* d'une coupe $(S, \bar{S})$ séparant la source du puits par :

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

Ce chapitre est consacré notamment à la résolution de deux problèmes relatifs à un réseau. Le premier est la recherche d'un *flot de valeur maximum*. Le second est la recherche d'une *coupe de capacité minimum*. Nous allons voir maintenant les liens théoriques existant entre ces deux problèmes.

IX.1.3. Résultats théoriques

Montrons d'abord l'équivalence des définitions données pour la valeur d'un flot.

Proposition. *Si f est un flot et $(S, \bar{S})$ une coupe séparant la source du puits, la quantité :*

$$f(S, \bar{S}) - f(\bar{S}, S)$$

est indépendante du choix de S, ensemble séparant la source du puits, et définit donc sans ambiguïté la valeur du flot.

Preuve.— Pour tout sommet x de S autre que s on a, d'après le principe de conservation du flot,

$$\sum_{u \text{ d'origine } x} f(u) \quad - \quad \sum_{u \text{ d'extrémité } x} f(u) = 0 ;$$

par ailleurs, en retenant la définition de la valeur du flot calculée à la source, on a :

$$\sum_{u \text{ d'origine } s} f(u) \quad - \quad \sum_{u \text{ d'extrémité } s} f(u) = val(f).$$

Faisant la somme de toutes ces égalités sur l'ensemble des sommets de S, on voit certains des flux $f(u)$ apparaître deux fois : ce sont ceux dont les deux extrémités appartiennent à S, et ils apparaissent alors une fois avec le signe « + », dans le terme correspondant à leur origine, une autre fois avec le signe « – », dans le terme correspondant à leur extrémité terminale ; ces deux valeurs s'annulent donc. Par ailleurs, pour les arcs u de $(S, \bar{S})$, la valeur de $f(u)$ apparaît une fois exactement avec le signe « + », et pour ceux de $(\bar{S}, S)$, elle apparaît exactement une fois avec le signe « – ». Le résultat de la sommation indiquée donne donc

$$f(S, \bar{S}) - f(\bar{S}, S) = val(f).$$

Pour tout arc u de cette coupe, par définition d'un flot, nous avons $f(u) \leq c(u)$, de sorte que $c(S, \bar{S})$ est supérieure ou égale à $f(S, \bar{S})$ pour tout flot f. Par ailleurs, pour tout arc u de $(\bar{S}, S)$, nous avons $f(u) \geq 0$. On déduit, de ces remarques et de la proposition ci-dessus, le théorème suivant :

Théorème. *Soit f un flot dans un réseau et $(S, \bar{S})$ une coupe séparant s de p. On a alors :*

1. $val(f) \leq c(S, \bar{S})$;

2. si $val(f) = c(S, \bar{S})$, alors f est de valeur maximum et $(S, \bar{S})$ de capacité minimum ;

3. $val(f) = c(S, \bar{S})$ si et seulement si :

- *pour tout arc de $(S, \bar{S})$, $f(u) = c(u)$;*
- *pour tout arc de $(\bar{S}, S)$, $f(u) = 0$.*

On peut s'attendre alors à ce que le théorème ci-dessus soit complété par la réciproque suivante : si f est de valeur maximum et $(S, \bar{S})$ de capacité minimum, alors $val(f) = c(S, \bar{S})$. Ce résultat est exact et sera prouvé plus loin.

IX.1.4. Lien avec la programmation linéaire

Le problème du flot de valeur maximum est un problème d'optimisation linéaire où les variables sont les flux circulant sur les différents arcs du réseau ; les contraintes s'expriment clairement de façon linéaire de même que la valeur du flot qui est alors la fonction objectif. Le problème de la coupe de capacité minimum est aussi un problème d'optimisation linéaire, mais cela est plus difficile à mettre en évidence ; nous ne le faisons pas ici. Enfin, on pourrait montrer que les deux problèmes sont en fait duaux l'un de l'autre.

IX.2. Algorithme de Ford et Fulkerson

L'algorithme de Ford et Fulkerson détermine un flot de valeur maximum, ainsi qu'une coupe de capacité minimum. Il repose sur le principe suivant : partant d'un flot dans le réseau (par exemple le flot nul), on cherche s'il existe une « chaîne augmentante » (voir définition ci-dessous) pour ce flot de s à p. S'il n'en existe pas, le flot est maximum ; s'il en existe une, on augmente le flot le long des arcs de cette chaîne et on recommence avec le nouveau flot obtenu. Ce principe est détaillé par les explications qui suivent.

IX.2.1. Chaîne augmentante

Supposons établi un flot f dans le réseau. On considère alors une *chaîne* dans le réseau, d'extrémités s et p. Par chaîne, on entend une chaîne du sous-graphe non orienté sous-jacent. Si on veut augmenter la valeur du flot dans le réseau en modifiant $f(u)$ pour les arcs u de cette chaîne, sans modifier le flux des arcs ne faisant pas partie de la chaîne, il est facile de voir que l'on doit, pour assurer la conservation du flot, augmenter d'une quantité constante α les flux $f(u)$ pour les arcs u de cette chaîne à l'endroit (c'est-à-dire les arcs dont on rencontre l'origine

avant l'extrémité lorsqu'on parcourt la chaîne de s vers p) et diminuer, de cette même quantité α, les flux $f(u)$ pour les arcs u à l'envers (c'est-à-dire les arcs dont on rencontre l'extrémité avant l'origine lorsqu'on parcourt la chaîne de s vers p). Considérons en effet deux arcs u et v consécutifs sur la chaîne, incidents en x, et par exemple parcourus à l'endroit. Si on augmente le flot d'une valeur α sur l'arc u, pour continuer à assurer la conservation du flot en x, sous l'hypothèse qu'elle l'était avant cette modification, on voit que l'on est amené à l'augmenter de la même quantité sur l'arc v : la quantité excédentaire α qui entre en x doit en effet quitter x. Tous les autres cas se traitent de manière analogue.

On ne peut donc augmenter le flot en utilisant la chaîne comme on l'a précisé ci-dessus que si l'on a $c(u) - f(u) \geq \alpha$ pour les arcs à l'endroit et $f(u) \geq \alpha$ pour les arcs à l'envers. S'il existe un $\alpha > 0$ vérifiant toutes ces contraintes, la chaîne est dite *augmentante* pour le flot f de s à p. En désignant par C^+ l'ensemble des arcs à l'endroit d'une chaîne augmentante et par C^- l'ensemble des arcs à l'envers de cette chaîne, on prendra alors

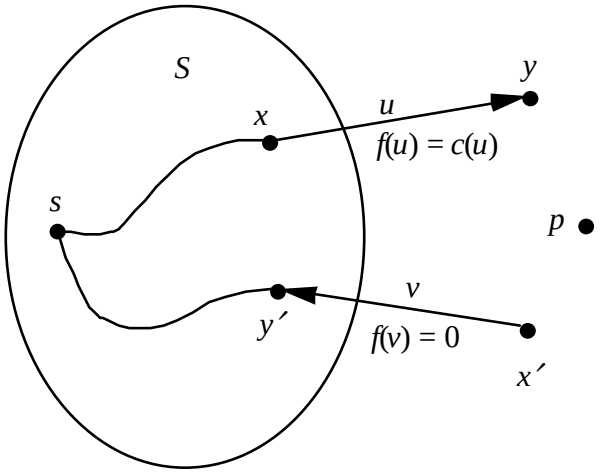
$$\alpha = \min\left\{ \min_{u \in C^+} \left[c(u) - f(u)\right] ; \ \min_{u \in C^-} f(u)\right\}.$$

Ayant ainsi déterminé une chaîne augmentante pour le flot et la valeur de α correspondante, on augmente le flux $f(u)$ de α pour les arcs u de C^+ et on le diminue de α pour les arcs u de C^- : la valeur du flot a augmenté de α.

Proposition. *Soit f un flot dans un réseau. f est un flot de valeur maximum si et seulement s'il n'existe pas de chaîne augmentante de s à p.*

Preuve.— L'existence d'une chaîne augmentante de s à p implique clairement que le flot n'est pas de valeur maximum.

Démontrons la réciproque. Supposons qu'il n'existe pas de chaîne augmentante de s à p. On définit une chaîne augmentante de s à un sommet x quelconque comme ci-dessus en remplaçant p par x. Appelons S l'ensemble des sommets x pour lesquels il existe une chaîne augmentante de s à x (on considérera que s fait partie d'un tel ensemble, en posant conventionnellement qu'une chaîne réduite à s est augmentante de s à s). Le sommet s est dans S et, par hypothèse, on suppose que p n'est pas dans S. $(S, \bar{S})$ est donc une coupe séparant s de p. Soit (x, y) un arc dont l'origine x est dans S et l'extrémité y dans $\bar{S}$.



Il existe une chaîne augmentante de s à x mais cette chaîne ne peut pas être prolongée en une chaîne augmentante de s à y : c'est que l'arc (x, y) est saturé (le flux qui le traverse est égal à sa capacité). Soit (x', y') un arc dont l'origine x' est dans $\bar{S}$ et l'extrémité y' dans S. Il existe une chaîne augmentante de s à y' mais cette chaîne ne peut pas être prolongée en une chaîne augmentante de s à x' : c'est que sur l'arc (x', y') circule un flux nul.

D'après le théorème du paragraphe IX.1.3, on déduit que f est de valeur maximum.

On peut maintenant énoncer le théorème suivant :

Théorème de la coupe et du flot. *Soit G un réseau ; la valeur maximum d'un flot dans G est égale à la capacité minimum d'une coupe de G.*

Preuve.— La valeur du flot est définie sur une partie fermée de R^m, où m est le nombre d'arcs du réseau. Elle est de plus majorée par la capacité de toute coupe. Elle atteint par conséquent sa borne supérieure.

Soit donc f un flot maximum. Il n'existe donc pas pour f de chaîne augmentante de s à p. En reprenant la démonstration de la proposition précédente et d'après le théorème du paragraphe IX.1.3, la coupe $(S, \bar{S})$ obtenue en mettant dans S la source s ainsi que tous les sommets x pour lesquels il existe une chaîne augmentante de s à x (y compris s) a une capacité égale à la valeur du flot.

IX.2.2. Description de l'algorithme de Ford et Fulkerson

Comme il a été dit plus haut, chaque itération de l'algorithme de Ford et Fulkerson est constituée de deux phases. La première est constituée d'un « algorithme de marquage » destiné à exhiber une chaîne augmentante s'il en existe une. La seconde exploite la chaîne augmentante pour améliorer le flot courant.

Avant de détailler l'algorithme de Ford et Fulkerson, nous devons, ici encore, définir quelques termes propres. On considère un flot f dans le réseau. On appelle « sommet marqué » un sommet tel qu'on a exhibé une chaîne augmentante de s à ce sommet pour le flot f. Examiner un sommet marqué x, c'est regarder si l'on peut prolonger la chaîne augmentante de s à x jusqu'à un nouveau sommet voisin de x non encore marqué. Pour cela, on attribue à chaque sommet marqué une « marque » à deux places. Dans la première place de la marque de x, nous gardons le nom du sommet qui nous a permis de marquer x, c'est-à-dire du sommet t dont l'examen a permis de prolonger la chaîne augmentante de s à t jusqu'à x. Dans la seconde place, nous conservons deux renseignements. Le signe « + » indique que, lorsque x a été marqué à l'aide de t, l'arc utilisé était (t, x), c'est-à-dire un arc à l'endroit par rapport à la chaîne augmentante ; le signe « – » indique que, lorsque x a été marqué, l'arc utilisé était (x, t), c'est-à-dire un arc à l'envers par rapport à la chaîne augmentante. Le second renseignement indique de combien on peut espérer augmenter le flot à l'aide de cette chaîne, jusqu'à x : c'est le minimum de la valeur absolue de la marque de t et de $c(t, x) - f(t, x)$ si le signe de x est « + », ou bien, si le signe de x est « – », le minimum de $f(x, t)$ et de la valeur absolue de la marque de t.

Pour initialiser l'algorithme de marquage, qui constitue la première phase d'une itération de l'algorithme de Ford et Fulkerson, nous marquons le seul sommet s avec, en première place, un sommet fictif Δ dont le rôle est symbolique, et, en seconde place, une valeur infinie ; aucun autre sommet n'est marqué ; aucun sommet n'est examiné. La procédure de recherche d'une chaîne augmentante consiste, tant que le puits n'est pas marqué et qu'il existe un sommet marqué et non examiné, à examiner un tel sommet. Plus précisément, l'algorithme de marquage peut être décrit de la manière suivante, en appelant f le flot courant :

- Marquer s par $(\Delta, +\infty)$
- pour tout sommet $x \neq s$, considérer x comme non marqué
- considérer qu'aucun sommet n'est examiné
- tant que p est non marqué et qu'il reste un sommet marqué non examiné, faire
    - soit x un sommet marqué non examiné et soit α la valeur absolue du second paramètre de la marque de x
    - pour tout successeur y de x qui n'est pas marqué, faire
        - si $c(x, y) > f(x, y)$, alors
            $\beta \leftarrow \min\{\alpha, c(x, y) - f(x, y)\}$
            marquer y par $(x, +\beta)$
    - pour tout prédécesseur z de x qui n'est pas marqué, faire
        - si $f(z, x) > 0$, alors
            $\beta \leftarrow \min\{\alpha, f(z, x)\}$
            marquer z par $(x, -\beta)$
    - considérer x comme examiné

À la sortie de l'algorithme de marquage, si le puits est marqué, une chaîne augmentante C est mise en évidence ; on peut la reconstituer de proche en proche, en partant de p, à l'aide des indications conservées dans la marque. On utilise alors C pour améliorer le flot courant d'une quantité égale à la valeur absolue α du second paramètre de la marque de p : un arc (x, y) parcouru à l'endroit aura son flux augmenté de α ; sinon (arc à l'envers), le flux de l'arc (x, y) est diminué de α. Si le puits n'est pas marqué, il n'existe pas de chaîne augmentante de s à p et le flot courant est maximum.

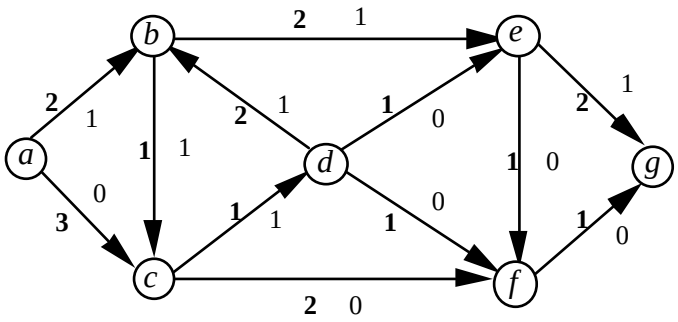
Ce qui donne la description suivante de l'algorithme de Ford et Fulkerson :

- Définir un flot f, éventuellement le flot nul
- répéter
    - appliquer l'algorithme de marquage
    - si p est marqué, alors
        - soit C la chaîne augmentante de s à p
        - soit α la valeur absolue du second paramètre de la marque de p
        - pour tout arc (x, y) de C parcouru à l'endroit, faire
            $f(x, y) \leftarrow f(x, y) + \alpha$
        - pour tout arc (x, y) de C parcouru à l'envers, faire
            $f(x, y) \leftarrow f(x, y) - \alpha$
- jusqu'à ce que p ne soit plus marqué

À la sortie de cette boucle, le flot est de valeur maximum.

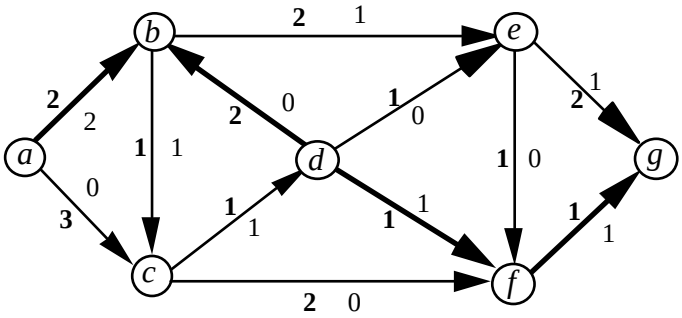
IX.2.3. Un exemple

Considérons le réseau représenté ci-après. Les nombres en gras, à côté des arcs, indiquent les capacités des arcs. Les autres (en caractères normaux) indiquent la valeur d'un flot possible. Le sommet s est en fait le sommet a, le sommet p n'est autre que g.

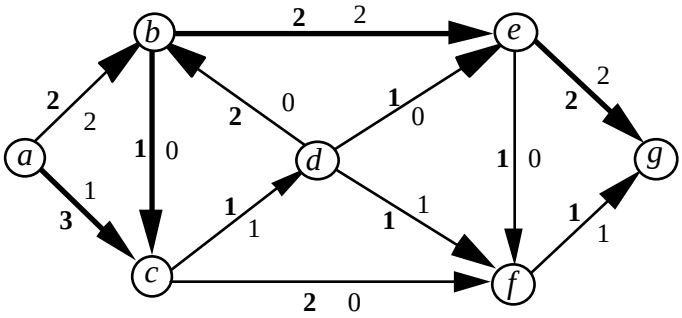


Le déroulement de l'algorithme peut être le suivant : on marque *a*, que l'on examine ensuite. Dans l'examen de *a*, on marque *b* par *a* et +1, et *c* par *a* et +3. Ensuite, on examine par exemple *b*, dont l'examen nous permet de marquer *d* par *b* et –1, et *e* par *b* et +1. Si on examine alors *d* (l'algorithme de Ford et Fulkerson n'impose rien sur l'ordre d'examen des sommets marqués), on marque *f* par *d* et +1. L'examen de *f* permet enfin de marquer *g* par *f* et +1.

L'application de l'algorithme à l'exemple permet ainsi de trouver la chaîne augmentante indiquée en traits gras ci-dessous, la valeur de α étant égale à 1 ; les arcs *(a, b)*, (*d, f*), (*f, g*) sont des arcs à l'endroit de cette chaîne augmentante, l'arc (*d, b*) est un arc à l'envers. La valeur du flot avant augmentation est de 1, après augmentation du flot de la valeur α (on ajoute 1 sur les arcs à l'endroit et on retranche la même quantité sur l'arc à l'envers), elle devient égale à 2. Dans le réseau ci-dessous, les valeurs des flux ont été actualisées.



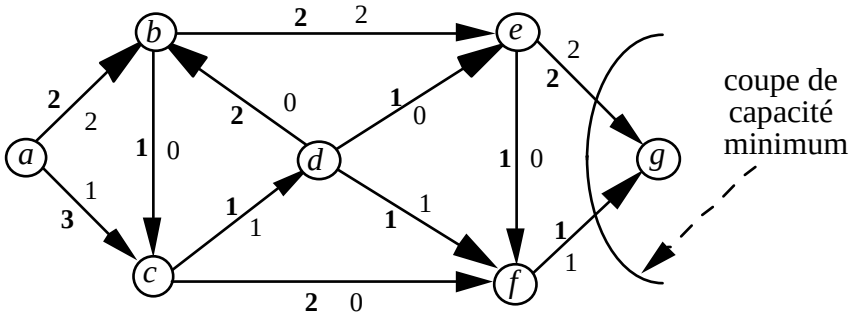
Il convient encore de continuer pour obtenir le flot maximum : l'itération suivante exhibe par exemple la chaîne augmentante *a c b e g* qui permet d'augmenter le flot d'une unité. Ce qui donne le flot représenté dans la figure suivante.



Puis l'algorithme de marquage atteint tous les sommets sauf *g*, ce qui provoque la fin de l'application de l'algorithme de Ford et Fulkerson. Le flot représenté ci-dessus est donc de valeur maximum.

On a aussi, du même coup, résolu le problème de la coupe de capacité minimum. En effet, une telle coupe est obtenue en isolant les sommets marqués lors de la dernière itération de l'algorithme de Ford et Fulkerson (ici, les sommets *a*, *b*, *c*, *d*, *e* et *f*) des autres (ici, le sommet

g seulement). Cette coupe, de capacité égale à la valeur maximum du flot (ici 3), est matérialisée ci-dessous par un trait séparant *g* des autres sommets.



IX.2.4. Preuve, convergence et complexité de l'algorithme

Supposons que l'algorithme de marquage se termine sans avoir marqué *p*. Soit *S* l'ensemble des sommets qui ont été marqués au cours de cette dernière phase. *S* contient *s* (puisque toute itération de l'algorithme de marquage commence par marquer *s*), alors que *p* ne lui n'appartient pas. *S* est donc un ensemble séparant la source du puits. Par construction du marquage, tous les arcs de $(S, \bar{S})$ sont saturés et tous les arcs de $(\bar{S}, S)$ portent un flux nul. La capacité de la coupe $(S, \bar{S})$ est égale à la valeur du flot *f* : par application du théorème du paragraphe IX.1.3, le flot est de valeur maximum et la coupe est de capacité minimum.

Supposons que toutes les capacités sont des entiers. Cette hypothèse n'est pas contraignante lorsque l'on s'intéresse à l'implémentation concrète de l'algorithme : les réels sont en fait codés comme des nombres décimaux et en multipliant toutes les capacités par une puissance de 10 appropriée on obtiendrait des capacités entières (on multiplierait aussi par cette quantité la fonction *f* sur tous les arcs). Partant du flot nul, on augmente, à toute étape autre que la dernière, la valeur du flot d'une quantité entière strictement positive ; or, on sait que la valeur maximum du flot est bornée par la capacité de toute coupe séparant la source du puits. On est donc assuré d'avoir un nombre fini de recherches de chaînes augmentantes. Dès que les capacités sont entières, il y a donc convergence en un nombre fini d'itérations. En revanche, dans le cas de capacités réelles, on peut exhiber des exemples où l'algorithme de Ford et Fulkerson ne converge pas en un temps fini.

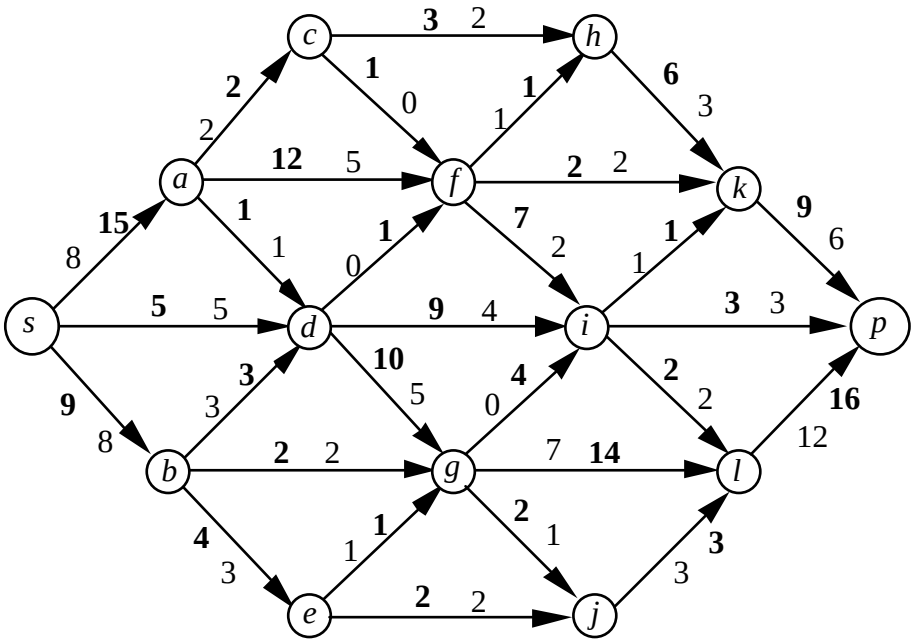
Lorsque l'on est dans le cas où toutes les capacités sont entières, on vient de voir que le nombre de phases de l'algorithme est borné par la valeur du flot, et donc par la capacité de toute coupe séparant la source du puits, par exemple par celle qui est définie par $S = \{s\}$. Par ailleurs, au cours d'une itération, on examine au plus tous les sommets autres que *p*, c'est-à-dire $n - 1$ sommets si l'ordre du graphe est *n*. Or, examiner un sommet consiste à effectuer au plus un nombre d'opérations élémentaires proportionnel au degré de ce sommet, pour voir si l'on peut prolonger la chaîne augmentante arrivant à ce sommet. Le nombre d'opérations élémentaires à effectuer est donc, au cours d'une étape, borné par la somme des degrés des sommets, c'est-à-dire deux fois le nombre d'arcs *m*. La complexité de l'algorithme de Ford et Fulkerson est donc $O(m.val_{max})$ où val_{max} désigne la valeur maximum du flot, ou aussi $O\left(m.c\left(S,\bar{S}\right)\right)$ où $\left(S,\bar{S}\right)$ est une coupe quelconque, ou encore $O(m.n.c_{max})$ où c_{max} désigne la capacité maximum des arcs, majorant que l'on obtient par exemple en considérant la coupe définie par $S = \{s\}$, de capacité majorée par $n.c_{max}$.

Enfin, la démonstration précédente implique le théorème suivant :

Théorème. *Dans un réseau à capacités entières, il existe un flot maximum tel que tous les flux soient entiers.*

IX.3. Exercice

Déterminer un flot de valeur maximum de *s* à *p* dans le réseau ci-dessous, et prouver que le flot calculé est bien maximum. Le chiffre en gras à côté d'un arc indique la capacité de l'arc ; l'autre chiffre indique le flux sur l'arc, trouvé par un élève astucieux, mais paresseux : on peut faire mieux.



X. Applications de la théorie des flots

X.1. Application à la détermination des connectivités d'un graphe

Nous allons étudier une application très importante de la théorie des flots : en effet, elle va nous permettre de calculer de façon efficace la connectivité d'un graphe et d'établir les théorèmes de Menger ; nous devons tout d'abord donner quelques définitions relatives aux graphes. Soit $G = (X, E)$ un graphe non orienté.

Définition. *G est dit k-connexe s'il a au moins k + 1 sommets et si la suppression d'au plus k – 1 sommets quelconques résulte en un graphe connexe. On appelle sommet-connectivité de G le plus grand entier k tel que G soit k-connexe.*

Définition. *G est dit k-arête-connexe si la suppression d'au plus k – 1 arêtes quelconques résulte en un graphe connexe. On appelle arête-connectivité de G le plus grand entier k tel que G soit k-arête-connexe.*

Pour un graphe orienté, on peut définir la forte sommet-connectivité et la forte arc-connectivité d'une façon analogue.

X.1.1. Forte arc-connectivité d'un graphe orienté

Nous allons commencer par l'étude de la forte arc-connectivité d'un graphe orienté, la déterminer par application de l'algorithme de Ford et Fulkerson, et établir un des théorèmes de Menger.

Soient a et b deux sommets d'un graphe orienté $G = (X, U)$. On considère le réseau R_{ab} déterminé par

- ce graphe G ;
- le sommet source a et le sommet puits b ;
- une capacité égale à 1 sur tous les arcs de G.

Par ailleurs, définissons deux fonctions à valeurs entières, P et N, définies sur les couples de sommets distincts. $P(a, b)$ représente le nombre maximum de chemins d'origine a et d'extrémité b, deux à deux arc-disjoints et $N(a, b)$ représente le nombre minimum d'arcs qu'il faut supprimer pour qu'il n'existe plus de chemin de a vers b. Il est trivial de constater que l'on a $P(a, b) \leq N(a, b)$: en effet, il faut détruire au moins un arc sur chacun des $P(a, b)$ chemins pour espérer détruire tout chemin de a vers b.

Théorème. *Si f_{ab} est un flot de valeur maximum $val(f_{ab})$ dans le réseau R_{ab}, alors $val(f_{ab}) = P(a, b)$. De plus, pour toute paire de sommets distincts a et b de G, $P(a, b) = N(a, b)$, c'est-à-dire que le nombre minimum d'arcs qu'il faut détruire pour supprimer tout chemin de a à b est égal au nombre maximum de chemins deux à deux arc-disjoints de a à b (ce dernier résultat constitue un des théorèmes de Menger).*

Preuve.— Considérons un flot de valeur maximum que l'on peut supposer, comme on l'a vu dans le chapitre précédent, à valeurs entières puisque les capacités sont entières. Les arcs de R_{ab} qui portent un flux égal à 1 forment des chemins de a vers b, 2 à 2 arc-disjoints (la preuve de cette propriété est aisée à établir) et ceci permet d'obtenir : $val(f_{ab}) \leq P(a, b)$.

Considérons maintenant une coupe de capacité minimum $(S, \overline{S})$. D'après le théorème de la coupe et du flot, cette coupe est de capacité $val(f_{ab})$. Or, les capacités des arcs étant toutes égales à un, la capacité d'une coupe est égale au nombre d'arcs de cette coupe. La destruction des arcs de cette coupe détruit tout chemin de a à b puisque, $(S, \overline{S})$ séparant a de b, pour aller de a à b il faut, à un moment donné, utiliser un arc de $(S, \overline{S})$. On voit donc que l'on a : $N(a, b) \leq val(f_{ab}) \leq P(a, b)$; mais comme l'inégalité $P(a, b) \leq N(a, b)$ a été constatée précédemment, il vient $P(a, b) = N(a, b)$.

Comment peut-on alors déterminer pratiquement la forte arc-connectivité de G ? C'est le minimum de $N(a, b)$ pris sur tous les couples de sommets (a, b). Cependant, grâce au lemme suivant, nous verrons qu'il n'est pas nécessaire de faire ce calcul pour $n.(n - 1)$ couples de sommets, et qu'il suffit d'envisager n couples.

Lemme de Zorn. *Supposons définie une numérotation des sommets de G de 0 à n – 1. La forte arc-connectivité de G est le minimum de $N(x_i, x_{i+1})$ quand i varie de 0 à n – 1, en posant $x_n = x_0$.*

Preuve : La preuve proposée est par l'absurde. Supposons que le minimum de $N(x_i, x_{i+1})$ quand i varie de 0 à $n - 1$ soit strictement plus grand que la forte arc-connectivité k du graphe, c'est-à-dire aussi plus grand que le cardinal minimum d'une coupe déconnectante. En détruisant les k arcs d'une coupe déconnectante dans G, ce qui sépare un certain sommet a d'un certain sommet b (au sens qu'il ne subsiste plus de chemin de a vers b après la suppression de ces arcs), on ne sépare donc aucun sommet x_l du sommet x_{l+1}, pour tout indice l compris entre 0 et $n - 1$. Mais $a = x_i$ et $b = x_j$ pour certains indices i et j. Par hypothèse, les indices suivants étant à considérer modulo n, il existe un chemin de x_i à x_{i+1}, de x_{i+1} à x_{i+2}, ..., de x_{j-1} à x_j, et cela après la suppression des k arcs précédents. En concaténant ces différents chemins, on obtient un chemin de a à b, ce qui est contraire au fait qu'il n'en subsiste plus.

Complexité de la détermination de la forte arc-connectivité

Rappelons que l'algorithme de Dinic est en $O(mn^2)$, n étant comme d'habitude l'ordre du graphe et m son nombre d'arcs. Grâce au lemme de Zorn, on voit que l'on peut déterminer la forte arc-connectivité d'un graphe orienté en $O(mn^3)$ (et plus généralement avec une complexité majorée par n fois la complexité d'un algorithme déterminant un flot maximum).

X.1.2. Détermination de l'arête-connectivité d'un graphe non orienté

Soit G un graphe non orienté et G^* le *symétrisé* de G, c'est-à-dire que G^* a même ensemble de sommets que G et que toute arête $\{u, v\}$ de G donne naissance, dans G^*, aux deux arcs opposés (u, v) et (v, u). Par ailleurs, on attribue à chaque arc de G^* une capacité de 1.

Étant donnés deux sommets a et b de G (et donc de G^*), on considère les quatre paramètres suivants :

- $N(a, b)$ est le nombre minimum d'arêtes à supprimer dans G pour qu'il n'existe plus de chaîne entre a et b ;
- $N^*(a, b)$ est le nombre minimum d'arcs à supprimer de G^* pour qu'il n'existe plus de chemin de a vers b dans G^* ;
- $P(a, b)$ est le nombre maximum de chaînes deux à deux arête-disjointes de G entre a et b ;
- $P^*(a, b)$ est le nombre maximum de chemins deux à deux arc-disjoints de G^* de a vers b.

Théorème. *Soit $val(f_{ab})$ la valeur maximum d'un flot de a à b dans G^*. On a alors :*

$$N(a, b) = N^*(a, b) = P(a, b) = P^*(a, b) = val(f_{ab}).$$

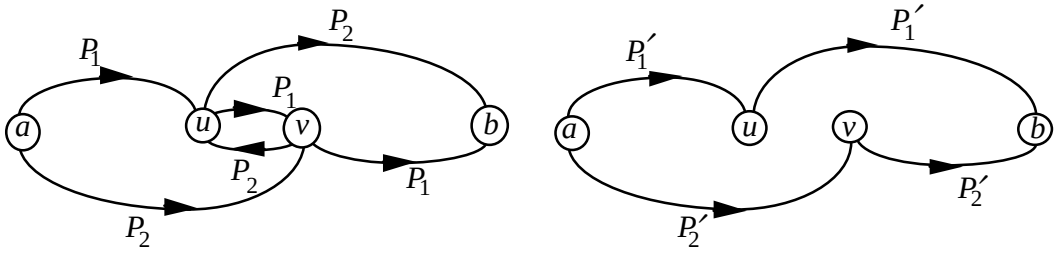
Preuve

1. On a de façon évidente $N(a, b) \geq P(a, b)$.

2. Montrons $P(a, b) \geq P^*(a, b)$.

Si les $P^*(a, b)$ chemins de a vers b n'utilisent pas, l'un, un arc (u, v), un autre, l'arc (v, u), on peut oublier les orientations et fabriquer $P^*(a, b)$ chaînes entre a et b. Supposons donc qu'un chemin P_1 et un chemin P_2 partagent une arête $\{u, v\}$, en utilisant l'un (u, v), l'autre (v, u).

On transforme P_1 et P_2 en P_1' et P_2' comme indiqué ci-dessous.



On voit que l'on partage ainsi exactement une arête de moins. Ceci permet de transformer les $P^*(a, b)$ chemins de a vers b, deux à deux arc-disjoints en $P^*(a, b)$ chemins deux à deux sans arête partagée et, en oubliant les orientations, en P^*(a, b) chaînes deux à deux arête-disjointes. D'où le résultat annoncé.

3. Montrons $N^*(a, b) \geq N(a, b)$.

Supprimons du graphe G les arêtes qui donnent naissance à $N^*(a, b)$ arcs d'un ensemble déconnectant a de b dans G^* ; il y a au plus $N^*(a, b)$ telles arêtes. L'existence d'une chaîne entre a et b dans le graphe obtenu impliquerait l'existence d'un chemin de a à b dans G^* ne contenant aucun des $N^*(a, b)$ arcs de l'ensemble déconnectant, ce qui ne se peut pas. C'est

donc qu'on peut déconnecter a de b dans G en supprimant au plus $N^*(a, b)$ arêtes. D'où la relation.

Nous avons finalement les inégalités suivantes :

$$N^*(a, b) \geq N(a, b) \geq P(a, b) \geq P^*(a, b).$$

Or, il a été montré au paragraphe X.1.1 qu'on a aussi

$$P^*(a, b) = N^*(a, b) = val(f_{ab}).$$

Toutes ces quantités sont donc égales, ce qui achève la démonstration du théorème.

X.1.3. Forte connectivité d'un graphe orienté, connectivité d'un graphe non orienté

Considérons un graphe orienté G. On construit alors un nouveau réseau de la façon suivante : à tout sommet x de G on associe un arc $(x', x\delta)$ et à un arc (u, v) de G on associe l'arc $(u\delta, v')$; tous les arcs ont une capacité égale à 1. On prend comme source $a\delta$ et comme puits b'. On peut alors montrer que le nombre minimum de sommets à supprimer pour détruire tout chemin de a vers b dans G est égal à la valeur d'un flot maximum de $a\delta$ à b' dans le réseau obtenu.

Si maintenant il s'agit de calculer la connectivité d'un graphe non orienté G, on utilise le symétrisé de G pour se ramener au cas précédent.

L'égalité de la valeur maximum du flot et de la capacité minimum d'une coupe permet en fait d'établir les deux théorèmes de Menger suivants :

Théorème. *Soit G = (X, E) un graphe non orienté. Il est k-connexe si et seulement si, entre deux sommets quelconques, il existe k chaînes n'ayant en commun que leurs extrémités.*

Théorème. *Soit G = (X, U) un graphe orienté. Il est k-fortement connexe si et seulement si, étant donnés deux sommets quelconques, il existe k chemins de l'un vers l'autre n'ayant en commun que leurs extrémités.*

X.2. Couplage maximum dans un graphe biparti

X.2.1. Un exemple et quelques définitions

Une agence matrimoniale a pour clients un ensemble H d'hommes et un ensemble F de femmes. Après présentation des dossiers des hommes aux femmes et réciproquement, il est apparu que certains mariages étaient tout à fait « envisageables » et les autres non. L'agence souhaite réaliser le maximum de mariages.

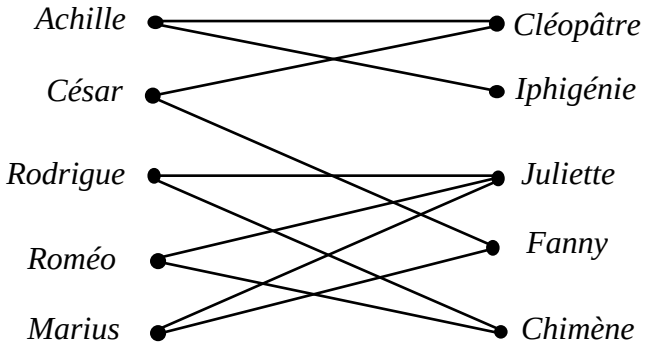
Formuler en termes de graphes le problème de la maximisation du nombre de mariages monogames (hétérosexuels) envisageables, puis donner un algorithme de résolution.

Résoudre le problème avec les données suivantes, les mariages envisageables étant notés par ***.

	Cléopâtre	*Iphigénie*	*Juliette*	*Fanny*	*Chimène*
Achille	***	***			
César	***			***	
Rodrigue			***		***
Roméo			***		***
Marius			***	***	

REPRESENTATION GRAPHIQUE

Construisons un graphe dont les sommets représentent les hommes et les femmes et où une arête lie deux sommets représentant les individus *a* et *b* si et seulement si un mariage est envisageable entre *a* et *b* :



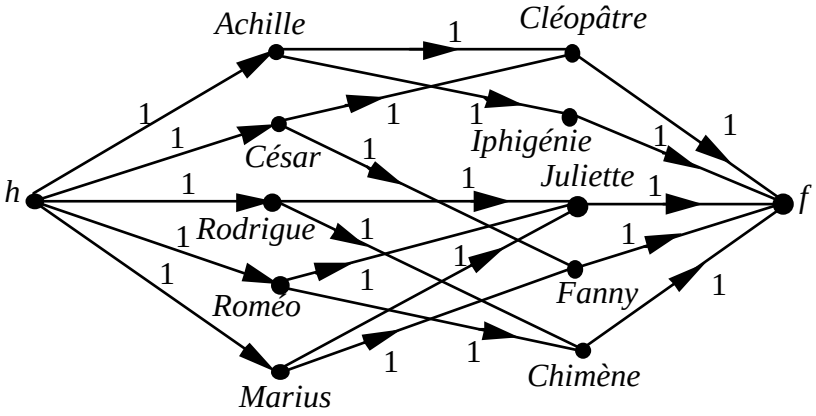
Les mariages étant hétérosexuels, aucune arête ne lie deux éléments de *H* ou de *F*. Le graphe est dit dans ce cas *biparti*, parce qu'il existe une bipartition des sommets en deux classes, sans arête entre deux sommets de la même classe. Chercher un ensemble de mariages envisageables le plus grand possible, revient à chercher parmi les arêtes de ce graphe un couplage de cardinal maximum. En effet, on appelle *couplage* un ensemble d'arêtes tel que deux de ces arêtes n'aient pas d'extrémité commune. Cette condition traduit le fait que les mariages correspondant à ces arêtes doivent être monogames.

X.2.2. Modélisation d'un problème d'affectation et solution du problème des mariages

En fait, le problème ci-dessus est un cas particulier d'un problème plus général appelé *problème d'affectation*, où il s'agit d'affecter des tâches à des ouvriers, ou des hommes à des femmes, etc. Sous certaines contraintes (ici la monogamie et l'hétérosexualité), il se traduit par la recherche d'un couplage maximum dans un graphe biparti.

Pour résoudre ce problème, nous allons nous ramener à la recherche d'un flot maximum dans un réseau que nous définissons ci-après dans le cas de l'agence matrimoniale, ceci ayant valeur d'exemple aisément généralisable. On ajoute deux sommets, *h* et *f*, au graphe précédent, *h* étant relié à l'ensemble des sommets de *H* et *f* à l'ensemble des sommets de *F*. On oriente en arcs les arêtes de *h* vers *H*, de *H* vers *F* puis de *F* vers *f*. Enfin, on donne 1

comme capacité à tous les arcs entre *h* et *H* comme entre *F* et *f*, et une valeur entière au moins égale à 1 aux arcs de *H* vers *F*, par exemple 1. On obtient le réseau représenté ci-dessous :



La recherche par l'algorithme de Ford et Fulkerson d'un flot de valeur *k* dans ce réseau, à partir du flot nul, résultera en un flot dont les valeurs sur les arcs seront 0 ou 1. Il est facile de se convaincre que les arcs entre *H* et *F* portant le flot 1 constitueront un couplage de cardinal *k*, grâce au choix des capacités et à la conservation des flots. Réciproquement, à un couplage de cardinalité *k*, on peut associer un flot dans le réseau de valeur *k*, en faisant circuler sur les arcs de *h* vers *H* incidents aux arcs du couplage un flux égal à 1, ainsi que sur ceux du couplage, et aussi sur ceux qui vont de *F* à *f* et sont incidents aux arcs du couplage ; on donne bien entendu la valeur 0 au flux des autres arcs du réseau. Ainsi un couplage de cardinalité maximum est associé à un flot de valeur maximum.

On peut alors résoudre le problème de la maximisation du nombre de mariages (c'est-à-dire la détermination d'un flot maximum dans le graphe que l'on vient de définir) à l'aide de l'algorithme de Ford et Fulkerson. Les chaînes augmentantes de *h* à *f* sont résumées par le tableau suivant :

itération	*A*	*Cés*	*Rod*	*Rom*	*M*	*Cléo*	*I*	*J*	*Fa*	*Chi*	*f*
1	+ *h*					+ *A*					+ *Cléo*
2		+ *h*							+ *Cés*		+ *Fa*
3			+ *h*					+ *Rod*			+ *J*
4				+ *h*						+ *Rom*	+ *Chi*
5	– *Cléo*	– *Fa*	– *J*		+ *h*	+ *Cés*	+ *A*		+ *M*		+ *I*

Chaînes augmentantes et flot trouvé pour chaque itération :

- itération 1 : *h* — *A* — *Cléo* — *f*, flot = 1 ;
- itération 2 : *h* — *Cés* — *Fa* — *f*, flot = 1 + 1 = 2 ;
- itération 3 : *h* — *Rod* — *J* — *f*, flot = 2 + 1 = 3 ;
- itération 4 : *h* — *Rom* — *Chi* — *f*, flot = 3 + 1 = 4 ;
- itération 5 : *h* — *M* — *Fa* — *Cés* — *Cléo* — *A* — *I* — *f*, flot = 4 + 1 = 5.

Et ainsi tout le monde peut se marier : *Marius* et *Fanny*, *César* et *Cléopâtre*, *Achille* et *Iphigénie*, *Rodrigue* et *Juliette*, *Roméo* et *Chimène* !…

XI. Graphes planaires et coloration de graphes

XI.1. Coloration : définitions et résultats combinatoires

Soit $G = (X, A)$ un graphe non orienté quelconque et k un entier strictement positif. Une *k-coloration* de G est une application φ de X dans $\{1, 2, ..., k\}$ telle que deux sommets de G reliés par une arête aient une image différente : pour toute arête $\{x, y\}$, $\varphi(x) \neq \varphi(y)$; l'entier $\varphi(x)$ s'appelle *la couleur de x*. Une k-coloration de G peut ne pas exister si k est trop petit par rapport à la structure de G ; le nombre minimum de couleurs nécessaires pour colorier G s'appelle le *nombre chromatique* $\gamma(G)$ de G. La détermination de $\gamma(G)$ constitue, pour un graphe G quelconque, un problème difficile (plus précisément, *NP*-difficile) à résoudre de manière exacte. Le théorème suivant établit un majorant de $\gamma(G)$ pour tout graphe G.

Théorème 6. *Soit G un graphe quelconque et soit $\Delta(G)$ le degré maximum de G : $\Delta(G) = \max_{x \in X} d(x)$. Alors $\gamma(G) \leq \Delta(G) + 1$.*

Preuve.— Considérons l'algorithme de $(\Delta(G) + 1)$-coloration suivant, pour lequel les sommets de G sont supposés s'appeler $x_1, x_2, ..., x_n$:

* Pour i variant de 1 à n, attribuer à x_i une couleur de $\{1, 2, ..., \Delta(G) + 1\}$ qui ne soit attribuée à aucun sommet x_j voisin de x_i avec $1 \leq j < i$.

Pour vérifier que cet algorithme glouton convient bien, il suffit de constater que, quand on examine x_i, ses voisins déjà coloriés (ceux d'indice inférieur à i) « consomment » au plus $d(x_i)$ couleurs, donc au plus $\Delta(G)$ couleurs. Il reste par conséquent au moins une couleur libre que l'on peut en effet attribuer à x. L'algorithme proposé construit donc bien une coloration de G n'utilisant pas plus de $\Delta(G) + 1$ couleurs. ♦

Remarquons que $\Delta(G) + 1$ couleurs peuvent être nécessaires pour colorier certains graphes G. C'est par exemple (mais pas seulement ; voir l'exercice 5) le cas du graphe complet K_n, pour lequel on a trivialement $\Delta(K_n) = n - 1$ et $\gamma(K_n) = n$. Mais la différence entre $\Delta(G) + 1$ et $\gamma(G)$ peut être arbitrairement grande, comme le montre le cas de l'« étoile » $K_{1,n-1}$: on a trivialement $\Delta(K_{1,n-1}) = n - 1$ et $\gamma(K_{1,n-1}) = 2$.

On peut d'autre part exhiber un minorant de $\gamma(G)$ valable pour tout graphe G. Pour cela, appelons $\omega(G)$ le cardinal du plus gros (au sens du nombre de sommets) sous-graphe complet de G. Ce nombre donne un minorant de $\gamma(G)$.

Proposition 7. *Pour tout graphe G, on a $\omega(G) \leq \gamma(G)$.*

Preuve.— Soit $K_{\omega(G)}$ la plus grosse clique de G. Pour colorier les sommets de $K_{\omega(G)}$, $\omega(G)$

couleurs sont nécessaires. *A fortiori* pour colorier tous les sommets de G. ♦

On trouve facilement des graphes G (dits *γ-parfaits*) pour lesquels on a $\omega(G) = \gamma(G)$. Malheureusement, $\omega(G)$ est difficile à calculer dans le cas général. En pratique (par exemple pour concevoir une « méthode de recherche par séparation et évaluation »), on se contente d'une clique maximale pour l'inclusion au lieu d'une clique de cardinal maximum (voir plus bas).

Le théorème suivant fournit un autre minorant de $\gamma(G)$ (il en existe d'autres ; voir par exemple l'exercice 6).

Théorème 8. *Soit G un graphe à n sommets et m arêtes ; on a :* $\dfrac{n^2}{n^2 - 2m} \leq \gamma(G)$.

Preuve.— Considérons une $\gamma(G)$-coloration de G et, pour $1 \leq i \leq \gamma(G)$, appelons X_i l'ensemble des sommets qui reçoivent la couleur i. Posons $|X_i| = n_i$ $(1 \leq i \leq \gamma(G))$; il vient : $\sum_{i=1}^{\gamma(G)} n_i = n$. Considérons la matrice d'adjacence M de G. Elle contient n^2 bits dont la somme vaut $2m$ (car par exemple la somme des bits de la i^{e} ligne donne le degré du i^{e} sommet) ; ceci donne donc le nombre N_1 de bits égaux à 1. Évaluons d'autre part un minorant du nombre N_0 de bits égaux à 0 dans M. Pour chaque i compris entre 1 et $\gamma(G)$, les éléments de X_i n'étant pas joints entre eux par des arêtes, ils définissent n_i^2 bits nuls ; d'où $N_0 \geq \sum_{i=1}^{\gamma(G)} n_i^2$. Or, d'après l'inégalité de Cauchy-Schwartz, $\sum_{i=1}^{\gamma(G)} n_i^2 \geq \dfrac{1}{\gamma(G)} \left(\sum_{i=1}^{\gamma(G)} n_i \right)^2$, et donc $N_0 \geq \dfrac{n^2}{\gamma(G)}$. On obtient par conséquent $n^2 = N_0 + N_1 \geq \dfrac{n^2}{\gamma(G)} + 2m$, d'où le résultat. ♦

On remarquera que l'inégalité du théorème 8 peut être une égalité. C'est le cas si tous les n_i sont égaux (pour que l'inégalité de Cauchy-Schwartz devienne une égalité) et que toutes les arêtes possibles entre X_i et X_j pour $i \neq j$ existent effectivement dans G (pour avoir $N_0 = \sum_{i=1}^{\gamma(G)} n_i^2$).

Pour finir cette partie, donnons quelques résultats faciles à établir concernant certaines familles de graphes et dont les preuves sont laissées en exercice au lecteur (voir exercice 7).

Proposition 9. *Soit G un graphe connexe à $n \geq 2$ sommets.*

1. $\gamma(G) \geq 2$; $\gamma(G) = 2$ si et seulement G est biparti.

2. $\gamma(G) = n$ si et seulement si $G = K_n$.

3. Si G s'obtient à partir de K_n en ôtant c arêtes formant un couplage, alors $\gamma(G) = n - c$.

4. $\gamma(G) = n - 1$ si et seulement si G s'obtient en ôtant de K_n un nombre d'arêtes partageant toutes une même extrémité compris entre 1 et $n - 2$.

XI.2. Algorithmes généraux de coloration

Comme on l'a dit plus haut, la détermination de $\gamma(G)$ et d'une $\gamma(G)$-coloration constitue un problème difficile dans le cas général d'un graphe G quelconque. Plus précisément, on ne connaît pas (ce qui ne signifie pas qu'il n'en existe pas...) d'algorithme dont la complexité soit majorée par un polynôme en n, même pour un graphe planaire.

Il existe cependant une exception notoire : les graphes 2-coloriables. En effet, d'après le premier énoncé de la proposition 9, les graphes 2-coloriables coïncident exactement avec les graphes bipartis. Or, on sait reconnaître les graphes bipartis (qui sont aussi les graphes sans cycle de longueur impair) à l'aide des algorithmes de parcours de graphe. La complexité de ceux-ci étant en $O(n + m)$ (ou $O(m)$ si on ne s'intéresse qu'à des graphes connexes), on en déduit qu'on peut reconnaître si un graphe est 2-coloriable en $O(n + m)$.

Nous présentons ci-dessous deux algorithmes calculant $\gamma(G)$ applicables à des graphes G quelconques. Compte tenu de ce qui vient d'être dit, la complexité de ces algorithmes est exponentielle en n dans le pire des cas. Nous présentons ensuite deux algorithmes polynomiaux en n, toujours applicables à des graphes quelconques, mais conduisant à un majorant de $\gamma(G)$ pour le premier, et à un minorant de $\gamma(G)$ pour le second.

XI.2.1. Premier algorithme exact

Soit $G = (X, A)$ le graphe à traiter. Le principe du premier algorithme exact (perfectible en s'inspirant des « méthodes de recherche par séparation et évaluation ») est le suivant. On part d'un nombre k de couleurs pour lequel on est sûr qu'il existe une k-coloration, par exemple $k = \Delta(G) + 1$. On décrémente la valeur de k (k devient $k - 1$) et on essaie successivement toutes les applications de X dans A. Si on en trouve une qui convient, on décrémente de nouveau la valeur de k et on recommence avec la nouvelle valeur. Sinon, c'est qu'il n'existe pas de k-coloration pour la valeur courante de k et on peut conclure : $\gamma(G) = k + 1$ (c'est-à-dire la dernière valeur de k pour laquelle on a trouvé une k-coloration). Cet algorithme est décrit succinctement ci-dessous.

* Initialiser k à l'aide d'une valeur pour laquelle il existe une k-coloration

* répéter

- $k \leftarrow k - 1$
- soit $Z(X, k)$ l'ensemble des applications de X dans $\{1, 2, ..., k\}$ et soient $z_1, z_2, ..., z_{Z(X, k)}$ les éléments de $Z(X, k)$
- $i \leftarrow 1$
- tant que $i \leq |Z(X, k)|$ et que z_i ne définit pas une k-coloration de G, faire $i \leftarrow i + 1$

* tant qu'il existe une k-coloration

* $\gamma(G) \leftarrow k + 1$.

Remarquons que lorsqu'on sort de la boucle « tant que » portant sur le compteur i, il existe une k-coloration de G (celle définie par z_i) si et seulement si i est inférieur ou égal à $|Z(X, k)|$.

Une variante de cet algorithme consisterait à partir d'une valeur de k pour laquelle il n'existe pas de k-coloration et à faire croître k jusqu'à trouver une k-coloration...

XI.2.2. Second algorithme exact

Le second algorithme exact repose sur deux opérations applicables à tout graphe non complet : l'ajout d'une arête entre deux sommets non adjacents ou la superposition de ces deux sommets. Plus précisément, soit $G = (X, A)$ le graphe à traiter qu'on supposera non complet (sinon, la proposition 9 permet de conclure). Soient x et y deux sommets non adjacents de G. On considère alors les deux graphes $G_1(G, \{x, y\})$ et $G_2(G, \{x, y\})$ définis comme suit :

• $G_1(G, \{x, y\})$ aura X comme ensemble de sommets et $A \cup \{\{x, y\}\}$ comme ensemble d'arêtes ;

• $G_2(G, \{x, y\})$ aura $(X - \{x, y\}) \cup \{xy\}$ comme ensemble de sommets, où xy est un nouveau sommet ; l'ensemble des arêtes de $G_2(G, \{x, y\})$ est obtenu en remplaçant les arêtes de G incidentes à x ou à y par des arêtes reliant le nouveau sommet xy aux anciens voisins de x ou de y ; formellement, cet ensemble est donc $\{a \in A$ pour a non incidente à x ou $y\} \cup \{\{xy, z\}$ pour tout voisin z de x ou de y dans $G\}$.

L'intérêt de ces opérations pour la coloration de G vient de la remarque suivante. Considérons une coloration quelconque de G. Soit elle attribue des couleurs différentes à x et à y, soit elle leur attribue au contraire la même couleur. Dans le premier cas, ajouter l'arête $\{x, y\}$ est compatible avec la coloration considérée ; autrement dit, cette coloration de G définit aussi une coloration de $G_1(G, \{x, y\})$. Réciproquement, toute coloration de $G_1(G, \{x, y\})$ définit une coloration de G pour laquelle x et y reçoivent des couleurs différentes. Dans le second cas, x et y ayant la même couleur, on peut aussi définir une coloration de $G_2(G, \{x, y\})$ à partir de la coloration de G considérée : on attribue à xy la couleur commune à x et à y et, pour tout autre sommet z de G, on ne change pas la couleur de z. Réciproquement, à partir de toute coloration de $G_2(G, \{x, y\})$, on peut trivialement définir une coloration de G qui attribue la même couleur à x et à y. On déduit de tout ceci la relation suivante :

$$\forall \{x, y\} \in A, \gamma(G) = \min[\gamma(G_1(G, \{x, y\})), \gamma(G_2(G, \{x, y\}))].$$

On peut en déduire un algorithme pour calculer $\gamma(G)$ et une $\gamma(G)$-coloration de G. Il suffit pour cela, si G n'est pas complet, de calculer récursivement $\gamma(G_1(G, \{x, y\}))$ et $\gamma(G_2(G, \{x, y\}))$. C'est ce que fait l'algorithme ci-dessous. Pour cela, on utilise une fonction récursive appelée *coloration* admettant le graphe G à colorier comme paramètre et renvoyant comme résultat un couple (γ, Col) où γ donne la valeur trouvée pour $\gamma(G)$ et où Col est un tableau ayant autant de cases qu'il y a de sommets dans G et définissant une γ-coloration de G : $Col(i)$ donne la couleur du i^e sommet de G. Le programme principal se contentera d'appeler *coloration* avec le graphe à traiter comme paramètre. Ce qui suit ne décrit que la fonction *coloration*.

Fonction coloration(G)

* si G est complet à n sommets, retourner le couple (n, Col) où Col est le tableau défini par $Col(i) = i$ pour $1 \leq i \leq n$

* sinon

  • soit $\{x, y\}$ une arête ne figurant pas dans G

  • construire le graphe $G_1(G, \{x, y\})$ par l'ajout de $\{x, y\}$ dans G

  • $(\gamma_1, Col_1) \leftarrow coloration(G_1(G, \{x, y\}))$

  • construire le graphe $G_2(G, \{x, y\})$ par superposition de x et de y dans G

  • $(\gamma_2, Col_2) \leftarrow coloration(G_2(G, \{x, y\}))$

  • si $\gamma_1 \leq \gamma_2$, alors renvoyer le couple (γ_1, Col_1)

  • sinon

    - pour $z \in \{x, y\}$, $Col(z) \leftarrow Col_2(z)$

    - $Col(x) \leftarrow Col_2(xy)$

    - $Col(y) \leftarrow Col_2(xy)$

  - renvoyer le couple (γ_2, Col)

La figure suivante illustre le fonctionnement de cet algorithme. On part ici d'un graphe G à 5 sommets appelés a, b, c, d et e. À partir de G, on construit à gauche le graphe $G_1(G, \{a, b\})$ obtenu en ajoutant l'arête $\{a, b\}$ à G et à droite le graphe $G_2(G, \{a, b\})$ obtenu en superposant a et b. Ce dernier graphe étant complet (il s'agit de K_4), la fonction *coloration* appliquée à $G_2(G, \{a, b\})$ ne fera pas d'appel récursif et renverra 4 comme nombre chromatique. Au contraire, la branche de gauche se subdivise en deux nouvelles branches, correspondant à gauche à l'ajout de l'arête $\{a, d\}$ à $G_1(G, \{a, b\})$, ce qui donne $G_1(G_1(G, \{a, b\}), \{a, d\})$, et correspondant à droite à la superposition de a et d dans $G_1(G, \{a, b\})$, ce qui donne $G_2(G_1(G, \{a, b\}), \{a, d\})$. Chacun de ses deux graphes donne de nouveau deux branches, selon que l'on ajoute l'arête $\{b, e\}$ ou que l'on superpose b et e. Dans l'exemple, les graphes obtenus sont alors complets et l'application de l'algorithme est achevée.

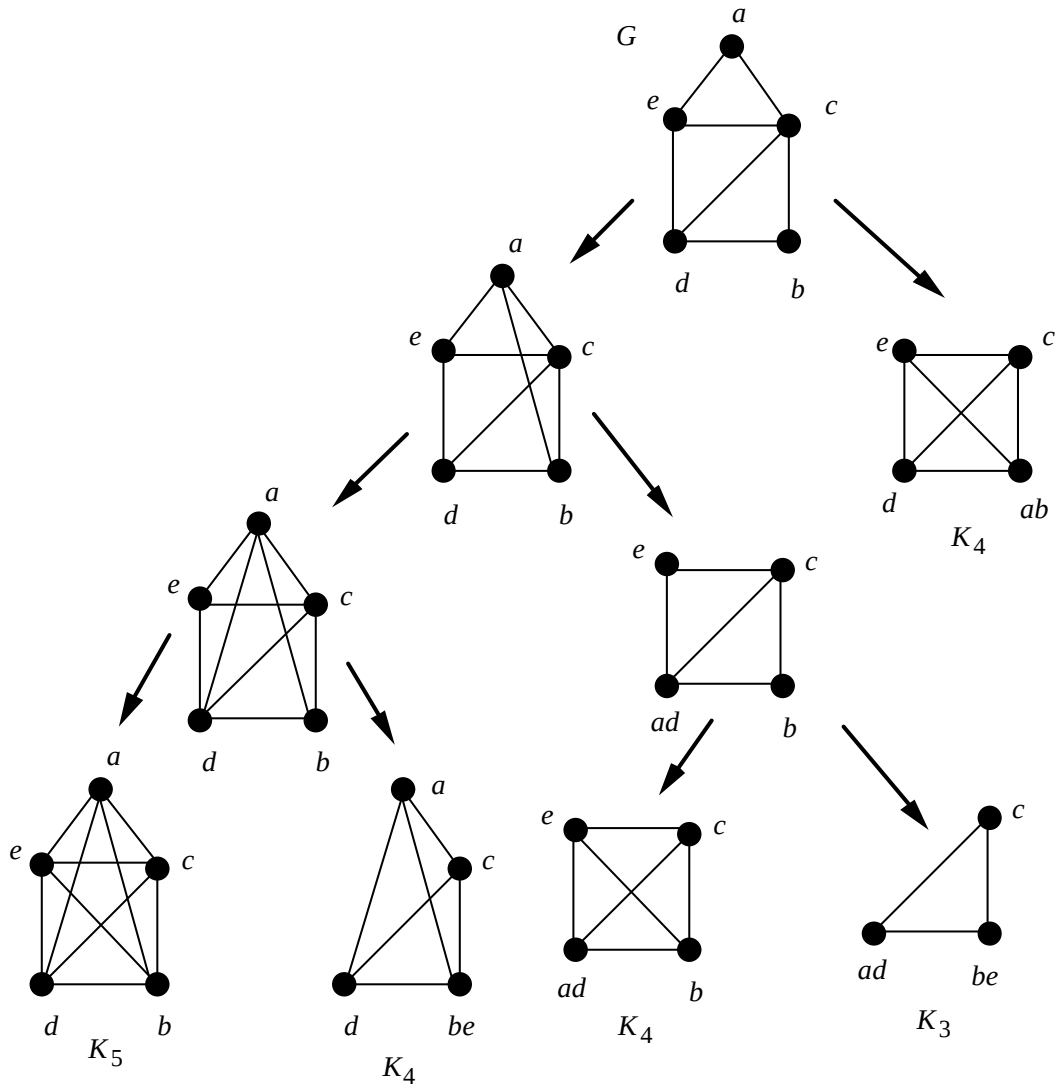


Figure 2. Exemple de calcul de $\gamma(G)$.

Les graphes complets associés aux feuilles de l'arbre ainsi construit étant K_3, K_4 ou K_5, on en déduit la valeur de $\gamma(G)$: $\gamma(G) = \gamma(K_3) = 3$. On peut de plus exhiber une 3-coloration de G à l'aide d'une coloration de la feuille associée à K_3. En effet, attribuons la couleur 1 au sommet *ad*, la couleur 2 à *be* et la couleur 3 à *c*. Comme K_3 a été obtenu à partir de son père par superposition de *b* et *e*, l'algorithme précédent nous indique que l'on peut colorier le graphe père de K_3 à l'aide de la 3-coloration suivante : on attribue la couleur 1 à *ad*, la couleur 2 à *b* et à *e*, la couleur 3 à *c*. De la même façon, l'algorithme permet d'obtenir une coloration de $G_1(G, \{a, b\})$ à partir de celle de son fils droit : on attribue la couleur 1 à *a* et à *d*, la couleur 2 à *b* et à *e*, la couleur 3 à *c*. On constate que, conformément à ce qu'indique l'algorithme, ceci définit aussi une 3-coloration de *G*.

XI.2.3. Algorithme approché donnant un majorant de $\gamma(G)$

Pour majorer le nombre chromatique $\gamma(G)$ de *G*, toute coloration de *G* convient. On peut par exemple appliquer l'algorithme glouton suivant. On commence par numéroter les sommets de 1 à *n*. On attribue la couleur 1 au sommet 1. Puis on examine le sommet 2 : s'il

est voisin de 1, on lui attribue la couleur 2 ; sinon, on lui attribue la couleur 1. On recommence ainsi avec les sommets suivants : quand on examine le sommet i, appelons $\gamma(i)$ le nombre de couleurs déjà utilisées par l'algorithme ; alors, si les voisins déjà coloriés de i « consomment » les $\gamma(i)$ couleurs disponibles, on attribue à i la (nouvelle) couleur $\gamma(i) + 1$ (et on a $\gamma(i + 1) = \gamma(i) + 1$), sinon, au moins une des couleurs déjà utilisées au moins une fois est disponible, et on l'attribue à i.

Dans l'exemple de la figure 2, on peut ainsi obtenir les attributions suivantes, en supposant que l'on examine les sommets selon l'ordre alphabétique et que l'on donne à chaque sommet la première couleur disponible :

* on attribue la couleur 1 à *a* ;

* on attribue la couleur 1 à *b*, puisque aucun des voisins déjà coloriés (il n'y en a pas !) n'utilise la couleur 1 ;

* on attribue la couleur 2 à *c*, puisque *a* (voisin de *c*) utilise déjà la couleur 1 ;

* on attribue la couleur 3 à *d*, puisque au moins un des voisins de *d* (ici *b*) utilise déjà la couleur 1 et qu'au moins un autre (ici *c*) utilise la couleur 2 ;

* on attribue la couleur 4 à *e*, puisque au moins un des voisins de *e* (ici *a*) utilise déjà la couleur 1, qu'au moins un autre (ici *c*) utilise la couleur 2 et qu'au moins un troisième (ici *d*) utilise la couleur 3 ;

* on en déduit la majoration suivante : $\gamma(G) \leq 4$.

Cet exemple montre qu'il ne s'agit bien dans le cas général que d'une méthode approchée. Des variantes sont envisageables selon l'ordre dans lequel on examine les sommets et selon la couleur que l'on choisit d'attribuer au sommet examiné quand plusieurs couleurs sont disponibles.

XI.2.4. Algorithme approché donnant un minorant de $\gamma(G)$

Pour obtenir un minorant de $\gamma(G)$, on peut chercher un sous-graphe de G qui soit une clique. C'est ce que fait l'algorithme glouton suivant. On sélectionne un sommet de plus grand degré ; soit x_1 ce sommet, et soit G_1 le sous-graphe de G engendré par l'ensemble des voisins de x_1. En plus de x_1, on sélectionne ensuite un sommet x_2 de plus grand degré dans G_1, et on construit le graphe G_2 engendré par les sommets de G_1 qui sont voisins de x_2. On recommence alors avec G_2, et ainsi de suite, tant que le graphe considéré n'est pas vide.

Plus précisément, soient $x_1, x_2, ..., x_i$ les sommets sélectionnés jusqu'à l'itération courante. On considère alors le sous-graphe G_i engendré par les sommets qui sont des voisins de x_1, de x_2, ..., de x_{i-1} et de x_i simultanément. Si G_i est vide, on s'arrête : $x_1, x_2, ..., x_i$ engendrent une clique maximale de G et on a $i \leq \omega(G) \leq \gamma(G)$. Sinon, on sélectionne un sommet x_{i+1} de G_i de degré maximum dans G_i, on l'ajoute à la suite des sommets $x_1, x_2, ..., x_i$ déjà sélectionnés et on recommence.

On obtient ainsi l'algorithme suivant :

* $C \leftarrow \varnothing$
* $i \leftarrow 0$
* $G_i \leftarrow G$
* tant que G_i est non vide, faire :
    - soit x_i un sommet de G_i de degré maximum
    - $C \leftarrow C \cup \{x_i\}$
    - $G_{i+1} \leftarrow$ sous-graphe de G_i engendré par les voisins de x_i dans G_i
* renvoyer C comme clique maximale de G et $|C|$ comme minorant de $\gamma(G)$.

Ainsi, quand on applique l'algorithme à l'exemple de la figure 2 (rappelé ci-dessous à gauche de la figure 3), on sélectionne d'abord c (seul sommet de degré maximum) puis on se place dans le graphe du milieu de la figure 3, engendré par les voisins de c. Dans ce graphe, d (par exemple) est de degré maximum : on le sélectionne et on construit le graphe de droite de la figure 3, obtenu en ne conservant que les voisins de d dans le graphe précédent. Dans ce nouveau graphe, on sélectionne b (on aurait pu sélectionner e) et on s'arrête, puisque le sous-graphe du graphe de droite engendré par les voisins de b est vide. Au total, on a sélectionné trois sommets (b, c, d) : on conclut que $\gamma(G)$ vaut au moins 3. Dans cet exemple, l'algorithme conduit à la valeur exacte de $\omega(G)$, mais ce n'est pas le cas général (voir exercice 8).

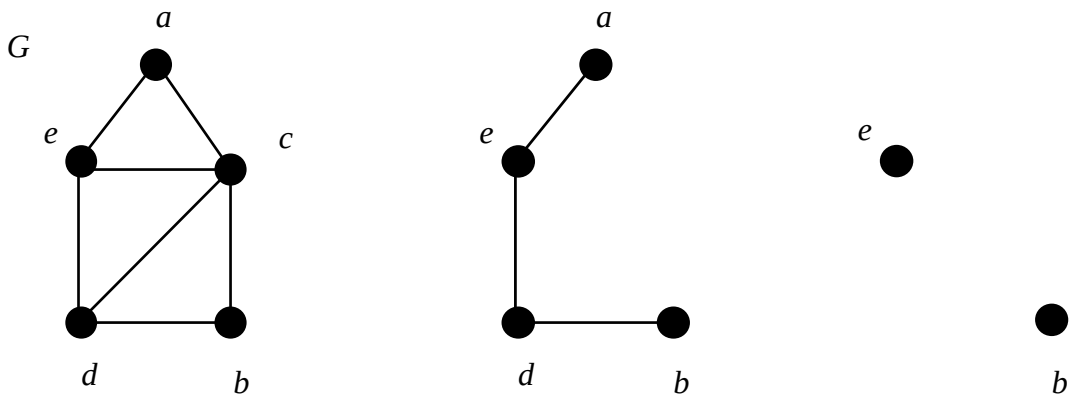


Figure 3. Illustration de l'algorithme déterminant un minorant de $\gamma(G)$.

XI.4. Exercices

Exercice 1

Exhiber une famille infinie de graphes G non complets vérifiant $\Delta(G) = 2$ et $\gamma(G) = 3$.

Exercice 2

1. À l'aide d'un exemple, montrer que l'algorithme de la partie 2.4. ne donne pas nécessairement la valeur de $\omega(G)$.

2. Donner un exemple de graphe G pour lequel on a $\omega(G) < \gamma(G)$.

XII. Complexité d'un problème

XII.1. Présentation et premières définitions

XII.1.1. Problème du voyageur de commerce

Pour illustrer ce chapitre, nous allons présenter un problème fort célèbre de la théorie des graphes : le *problème du voyageur de commerce* (en anglais *Travelling Salesman Problem*).

Étant donné un graphe complet valué à n sommets (à chaque arête est associée une « longueur »), on cherche un cycle hamiltonien de longueur minimum de ce graphe, un *cycle hamiltonien* étant un cycle qui passe une fois et une seule par chaque sommet et la longueur d'un cycle étant la somme des longueurs des arêtes le constituant.

Un algorithme trivial est le suivant : énumérer tous les cycles hamiltoniens du graphe et calculer leur longueur afin d'en retenir un minimisant la fonction longueur. Malheureusement, cet algorithme est en pratique impossible à mettre en œuvre sur un graphe même d'ordre assez faible du fait du grand nombre de solutions : un graphe complet à n sommets possède $\frac{(n-1)!}{2}$ cycles hamiltoniens (en effet, un cycle hamiltonien peut être décrit par la suite des n sommets selon l'ordre dans lequel on les rencontre : il y a $n!$ telles suites ; mais toute permutation circulaire laisse inchangé le cycle, de même que l'inversion de l'ordre d'écriture ; tout cycle est donc représenté $2n$ fois, d'où le résultat). Ce nombre entraîne un temps de calcul rapidement trop élevé : par exemple, pour $n = 20$, il faudrait environ 19 siècles de calcul à un ordinateur traitant un million de cycles hamiltoniens à la seconde.

Hélas, si inextricable que soit la mise en œuvre de cet algorithme, on n'en connaît pas de fondamentalement meilleur. On peut accélérer la vitesse de résolution par des méthodes de type *branch and bound*, dites aussi *méthodes par séparation et évaluation*, qui seront expliquées dans le chapitre suivant, mais tous les algorithmes exacts actuellement connus requièrent un temps de calcul rapidement trop important pour qu'il soit raisonnable de vouloir les employer d'un point de vue pratique quand l'ordre du graphe devient lui-même un peu élevé. Nous allons voir que ce problème est loin d'être le seul pour lequel on ne connaisse pas de « bon » algorithme de résolution exacte et qu'il est peu vraisemblable que l'on en trouve un jour (ce qui ne signifie pas *a priori* qu'il n'en existe pas).

Avant d'aller plus loin, définissons ce qu'on appelle *taille d'une instance*, notion beaucoup moins intuitive qu'il n'y paraît, et formalisons les notions de *complexité d'un algorithme* et de *bon algorithme* à l'aide des *machines de Turing*.

XII.1.2. Taille d'une instance

Lorsqu'on veut définir précisément la taille d'une instance d'un problème, il est important de prendre en compte tous les facteurs. Par exemple, dans le cas d'une instance du problème de plus court chemin, il faut tenir compte non seulement du nombre de sommets du graphe mais aussi des distances entre les paires de sommets. La description d'une instance d'un problème peut être vue comme une chaîne de caractères appartenant à un alphabet fixé. On suppose que le codage de cette description est fixé. C'est la longueur de cette chaîne de caractères qui définira la taille de l'instance. On ne s'intéresse ici qu'à ce qu'on appellera un codage
« raisonnable », c'est-à-dire, par exemple, que l'on n'admet pas de coder un entier n sur n caractères, mais sur $\log(n)$ caractères, la base du logarithme étant sans importance.

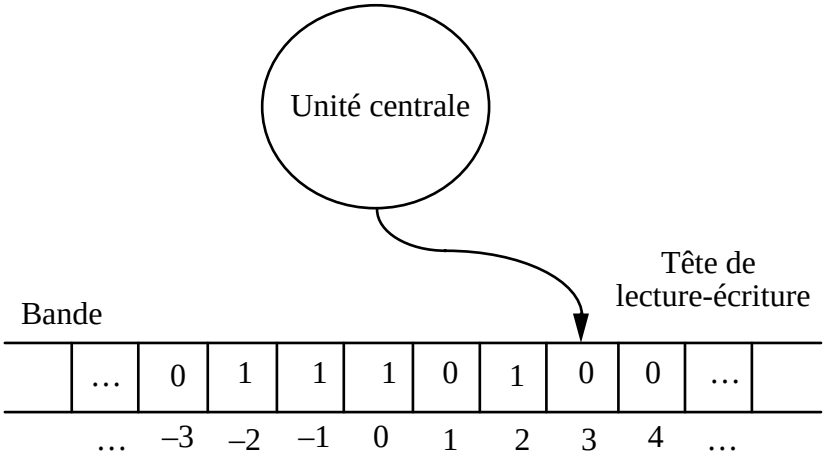
L'alphabet binaire {0, 1} permet un tel codage. Ainsi, représenter un entier k strictement positif à l'aide de ce codage nécessitera $\lceil \log_2(k+1) \rceil$ bits (où $\lceil x \rceil$ représente la partie entière par excès de x) ; représenter un graphe à n sommets par sa matrice d'adjacence nécessitera autant de bits qu'il y a de couples de sommets, soit n^2. Dans ce système binaire, la taille d'une instance est donc le nombre de bits nécessaires pour représenter toutes les données définissant l'instance, qu'il s'agisse de nombres, d'ensembles, de graphes ou encore d'autres structures. On adopte généralement ce codage pour les études de complexité ; ce sera le cas pour les considérations qui suivent.

XII.1.3. Machine de Turing et complexité d'un algorithme

Pour préciser la notion de complexité d'un algorithme, nous allons définir ce qu'est une *machine de Turing déterministe à une bande*. Une telle machine est constituée de trois parties :

- un ensemble d'états de la machine, constituant une « unité centrale » qui contrôle l'ensemble du déroulement du programme, à l'aide d'une « table » contenant la description des opérations à effectuer ;
- d'une bande infinie constituée de cases numérotées …, –3, –2, –1, 0, 1, 2, …, sur laquelle sont initialement inscrites les données et où seront inscrits les calculs intermédiaires et les résultats finals ;
- d'une tête de lecture-écriture susceptible de lire et de modifier les informations figurant sur la bande conformément aux instructions fournies par l' « unité centrale ».

La figure suivante schématise une machine de Turing.



Le comportement de la machine de Turing dépend de la table contenue dans l'unité centrale et des symboles lus dans les cases de la bande. À chaque pas, la tête de lecture-écriture lit le symbole contenu dans la case de la bande sur laquelle elle pointe. En fonction de ce symbole et de l'état actuel de la machine, la table indique à la tête ce qu'elle doit écrire dans la case sur laquelle elle pointe, détermine le déplacement de la tête (une case dans un sens ou dans l'autre, à moins qu'elle ne reste immobile) et définit l'état suivant de la machine.

L'exemple suivant illustre le fonctionnement d'une machine de Turing et aussi la modélisation d'un programme par une telle machine, et ceci nous permettra de donner une définition précise de ce que sont les algorithmes polynomiaux.

La machine que nous décrivons est constituée :

1. D'un ensemble fini Q d'états contenant un état initial q_0 et deux états de fin q_s et q_e (s comme succès et e comme échec ; si on applique cette machine de Turing à un problème de reconnaissance, c'est-à-dire un problème dans lequel on pose une question dont la réponse est « oui » ou « non » (voir plus loin, paragraphe XII.1.4), le succès pourra s'interpréter comme une réponse « oui » et l'échec comme une réponse « non »), comme toute machine, mais aussi des états q_1 et q_2.

2. D'un ensemble fini de valeurs que l'on peut lire ou écrire dans les cases de la bande, qui contient un mot de $\{0, 1\}^*$ (si A désigne un alphabet, c'est-à-dire un ensemble de symboles, A^* désigne l'ensemble des mots qu'on peut écrire à partir de A, c'est-à-dire toute chaîne de symboles appartenant tous à A) et un symbole particulier, que l'on appelle « blanc » b.

3. D'une fonction de transition qui, à un état de la machine autre que q_s et q_e, associe un triplet dont le premier élément indique le nouvel état de la machine, le deuxième le caractère qui sera écrit à la place du caractère lu, le troisième le sens de déplacement de la tête de lecture (déplacement d'au plus une case : on notera +1 un déplacement vers la droite, –1 un déplacement vers la gauche, et 0 l'absence de déplacement) ; la machine s'arrête sur l'état q_s ou q_e.

La table ci-dessous indique les transitions de la machine considérée. L'indice de ligne correspond à l'état de la machine, l'indice de colonne au caractère lu sur la bande.

	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_e, b, 0)$	
q_2	$(q_s, b, -1)$	$(q_e, b, 0)$	$(q_e, b, 0)$

L'effet d'un tel programme est le suivant : la donnée est une chaîne X de symboles appartenant à {0, 1}. La chaîne X est placée sur la bande, un symbole par case, le premier en position 1, le dernier en position $|X|$. Les autres cases de la bande contiennent initialement b. La machine est initialement en état q_0, et la tête de lecture-écriture est positionnée sur la case de numéro 1. Le calcul se déroule alors étape par étape.

Il est facile de voir que l'état q_0 consiste à se déplacer vers la droite à la recherche d'un blanc, et donc de la fin du mot, sans rien changer. Quand on a trouvé le premier blanc, on change d'état et on passe à q_1 en revenant en arrière d'une case : on pointe désormais sur le dernier bit du mot. Si celui-ci est un « 1 », on s'arrête dans l'état q_e : on a échoué ; si on lit au contraire un « 0 », on passe à l'état q_2 et on recule encore d'une case (on notera qu'il n'est pas possible de lire un blanc). Si le nouveau bit lu est un « 0 », on s'arrête dans l'état q_s : la réponse sera « oui » ; dans les deux autres cas, l'état final q_e est atteint : la réponse est non. Finalement, on constate que cette machine modélise un programme qui teste si un mot de {0, 1}* se termine par 00 : c'est en effet une condition nécessaire et suffisante pour qu'il soit reconnu par la machine, ou encore pour que la machine qui l'admet en lecture termine le calcul dans l'état q_s.

Un problème de reconnaissance peut se traduire en terme de machine de Turing déterministe (à une bande). Par exemple la machine de Turing ci-dessus est associée au problème suivant : étant donné un entier $N > 0$, existe-t-il un entier m tel que $N = 4m$? Le codage associé à une instance du problème est simple, puisqu'il suffit de donner une représentation binaire de N. La machine de Turing définie ci-dessus se termine en fait pour toute donnée (suite des éléments rangés initialement dans la bande) : le calcul se termine dans l'état « succès » si les deux bits à droite sont 0, elle se termine dans l'état « échec » sinon.

Nous pouvons maintenant donner une définition précise de la *complexité* de la machine M, ou encore de l'algorithme modélisé par M. Si, pour toute donnée, on atteint un état final (q_e ou q_s) en un nombre fini d'étapes, on peut définir un « temps de calcul », ou complexité, comme étant le nombre maximum de pas effectués par M pour pouvoir traiter n'importe quelle donnée, de taille fixée, du problème. Ce nombre de pas, ou ce temps de calcul $T_M(n)$ est donc une fonction de la taille $n = |X|$ de la donnée X. S'il existe un polynôme p tel que $T_M(n) \leq p(n)$, on parle d'une machine de Turing polynomiale, et on dit que l'algorithme modélisé par M est *polynomial* (ce qui ne signifie pas que sa complexité peut elle-même s'exprimer comme un polynôme en la taille des données, mais seulement qu'on peut la majorer par un tel polynôme ; ainsi un algorithme dont la complexité vaut $\log_2 n$, où n est la taille des données, est polynomial). Un algorithme dont la complexité n'est pas majorable par un polynôme en la taille des données est dit *exponentiel*, même si sa complexité ne s'exprime pas à l'aide d'une exponentielle au sens mathématique du terme (par exemple, un algorithme de complexité $n!$ où n est la taille de la donnée, est exponentiel : la fonction factorielle n'est pas majorable par un polynôme en n, sans pourtant être elle-même une fonction exponentielle). On donne parfois l'appellation de « bons algorithmes » ou encore d' « algorithmes efficaces » aux algorithmes polynomiaux ; il arrive cependant, dans la pratique, que des algorithmes exponentiels aient des performances remarquables (c'est le cas par exemple de l'algorithme du simplexe pour les problèmes de programmation linéaire).

En pratique, cette définition de la complexité d'un algorithme est souvent remplacée par une notion plus facile à manipuler. Au nombre de pas effectués par une machine de Turing, on substitue en général le nombre d'opérations considérées comme élémentaires effectuées par un ordinateur séquentiel ; par élémentaire, on entend une opération modélisable par un

nombre polynomial (par rapport à la taille des données) de pas d'une machine de Turing : c'est le cas notamment des quatre opérations arithmétiques, des différentes opérations de comparaison, ou encore de l'affectation. C'est cette notion de complexité qui a été introduite au chapitre IV et qui a été utilisée depuis. Elle a l'avantage de ne pas nécessiter la formalisation d'un algorithme à l'aide d'une machine de Turing (ce qui serait bien pénible...) et de permettre l'utilisation de concepts d'opérations plus familiers que celui de pas d'une machine de Turing, sans apporter de grosses différences quant au résultat qualitatif. En particulier, un algorithme admet une complexité polynomiale au sens des opérations élémentaires si et seulement si sa complexité au sens des machines de Turing l'est aussi.

XII.1.4. Problème de reconnaissance

Afin de donner des définitions correctes des notions que nous traiterons, nous nous intéresserons dans ce chapitre, sauf mention contraire, à des *problèmes de reconnaissance*, appelés aussi *problèmes de décision*, c'est-à-dire à des énoncés pour lesquels la réponse est « oui » ou « non », et non pas à des problèmes d'optimisation. Il existe un lien fort entre problème de reconnaissance et problème d'optimisation, lien que nous allons illustrer sur un exemple, celui bien sûr du voyageur de commerce. Le problème d'optimisation du voyageur de commerce, que nous noterons *OVC*, peut s'écrire de la façon suivante (pour simplifier les développements ultérieurs, les valuations du graphe seront supposées entières et strictement positives) :

Nom : *OVC*
Données : étant donné un graphe *G* d'ordre *n*, complet et valué par une fonction à valeurs entières strictement positives,
Objectif : déterminer la longueur minimum d'un cycle hamiltonien dans le graphe *G*.

Quant au problème de reconnaissance, que nous noterons *RVC*, associé au problème du voyageur de commerce, on peut l'énoncer ainsi :

Nom : *RVC*
Données : étant donné un entier *k*, étant donné un graphe *G* d'ordre *n*, complet et valué par une fonction à valeurs entières strictement positives,
Question : existe-t-il dans le graphe *G* un cycle hamiltonien de longueur inférieure ou égale à *k* ?

Il est clair que si nous résolvons *OVC* pour le graphe *G*, nous résolvons du même coup *RVC*. Par conséquent, les tailles des instances de *RVC* et de *OVC* ne différant l'une de l'autre que du terme additif $\lceil \log_2(k+1) \rceil$, s'il existe un algorithme polynomial (par rapport à la taille des instances de *OVC*) pour résoudre *OVC*, alors il existe un algorithme polynomial (par rapport à la taille des instances de *RVC*) pour résoudre *RVC*.

Réciproquement, supposons que nous résolvions successivement un nombre fini d'instances de *RVC*, en donnant à *k* des valeurs entières, décroissant de 1 à chaque étape. En partant d'une valeur initiale de *k* pour laquelle nous sommes assurés que la réponse est « oui », la dernière valeur de *k* pour laquelle la réponse est « oui » nous donne la valuation minimum d'un cycle hamiltonien.

On peut même faire mieux pour déterminer la valuation minimum, en utilisant un procédé dichotomique pour encadrer ce minimum : on part comme précédemment d'une valeur *k* pour laquelle la réponse est « oui », par exemple $n.val_{max}$, où val_{max} désigne la plus grande des

valuations du graphe ; puis on résout *RVC* en posant $k = \left\lfloor \frac{val_{max}}{2} \right\rfloor$ (où $\lfloor x \rfloor$ désigne la partie entière par défaut de *x*) : si la réponse reste « oui », on recommence avec $k = \left\lfloor \frac{val_{max}}{4} \right\rfloor$, sinon avec $k = \left\lfloor \frac{3.val_{max}}{4} \right\rfloor$, et ainsi de suite. Ainsi, aux parties entières près, l'intervalle de recherche diminue de moitié et il suffit d'environ $\log_2(n.val_{\max})$ résolutions d'instances de *RVC* pour résoudre l'instance associée de *OVC*.

Par conséquent, tout algorithme A_R de complexité $c(A_R)$ permettant de résoudre *RVC* donne un algorithme A_O de complexité environ $\log_2(n.val_{\max}).c(A_R)$ permettant de résoudre *OVC*. Or, si on appelle *val* la fonction de valuation de *G*, la taille d'une instance de *OVC* est de l'ordre de $T_O = \sum_{\{i,j\} \in G} \log_2(val(i, j) + 1)$, puisqu'on doit coder les entiers $val(i, j)$ pour toutes les arêtes $\{i, j\}$ du graphe (on notera que cette quantité est supérieure ou égale à $\frac{n(n-1)}{2}$, et donc *a fortiori* supérieure à $\log_2 n$), et la taille d'une instance de *RVC* est de l'ordre de $T_R = \log_2(k + 1) + \sum_{\{i,j\} \in G} \log_2(val(i, j) + 1)$. En remarquant qu'on peut ne s'intéresser qu'aux instances de *RVC* pour lesquelles on a $k < n.val_{max}$ (les autres instances admettent trivialement la réponse « oui »), il est facile de montrer le résultat suivant : s'il existe un polynôme *p* en T_R qui majore la complexité de A_R, alors il existe un polynôme *q* en T_O qui majore la complexité de A_O. Autrement dit, s'il existe un algorithme polynomial (par rapport à la taille des instances de *RVC*) pour résoudre *RVC*, alors il existe un algorithme polynomial (par rapport à la taille des instances de *OVC*) pour résoudre *OVC*.

Ces résultats montrent le lien fort entre *RVC* et *OVC* : l'un admet des algorithmes de résolution polynomiaux si et seulement l'autre en admet aussi. Cette propriété, en fait générale, nous permet de nous intéresser aux problèmes de reconnaissance pour obtenir des renseignements sur la difficulté des problèmes d'optimisation auxquels ils sont associés.

XII.2. Classes P et NP ; problèmes NP-complets

Les notions que nous allons développer dans cette partie ne s'appliquent qu'à des problèmes de reconnaissance mais, comme il a été dit plus haut, les résultats que nous obtiendrons nous donneront aussi des indications relatives aux problèmes d'optimisation.

XII.2.1. La classe P

Définitions. *Un problème est dit polynomial s'il existe un algorithme de complexité polynomiale permettant de répondre à la question posée dans ce problème, quelle que soit la donnée de celui-ci. La classe P est l'ensemble de tous les problèmes de reconnaissance polynomiaux.*

Bon nombre de problèmes sont connus pour être polynomiaux. C'est ainsi le cas pour les problèmes de reconnaissance associés aux problèmes traités dans les chapitres précédents de

cet ouvrage (y compris les problèmes de programmation linéaire, même si ce n'est pas l'algorithme du simplexe qui permet d'établir ce résultat). Mais ne pas connaître d'algorithme polynomial résolvant un problème donné ne signifie pas qu'il n'en existe pas. Pour prendre en compte cette difficulté, on définit une classe de problèmes plus large, la classe NP.

XII.2.2. La classe NP

Définition. *Un problème de reconnaissance est dans la classe NP si, pour toute instance de ce problème, on peut vérifier, en un temps polynomial par rapport à la taille de l'instance, qu'une solution proposée ou devinée permet d'affirmer que la réponse est « oui » pour cette instance.*

Notons qu'on ne s'intéresse pas ici à la justification d'une réponse négative : on cherche seulement à vérifier une réponse « oui ».

On peut préciser cette notion de « solution devinée » à l'aide des *machines de Turing non déterministes*, mais nous ne le ferons pas ici. Nous nous contenterons d'illustrer cette définition à l'aide, bien sûr, du problème du voyageur de commerce *RVC* (nous reprenons les notations de la définition donnée plus haut).

Proposition. *RVC est dans la classe NP.*

Preuve.— Il suffit de montrer comment on peut vérifier, en un temps polynomial, qu'une soi-disant solution permet bien d'établir la réponse « oui » pour l'instance considérée. Imaginons pour cela qu'un tiers nous dise que la réponse à cette instance est « oui » et, pour nous convaincre de ce résultat, exhibe une façon de visiter les villes en prétendant qu'il s'agit d'un cycle hamiltonien de longueur inférieure ou égale à la borne *k*.

Nous aurons alors à vérifier deux choses :

1. La structure proposée est un cycle hamiltonien.
2. La longueur de ce cycle est inférieure ou égale à *k*.

Pour cela, on vérifie d'abord que la suite *S* des sommets proposée comme solution contient le bon nombre *n* de sommets ; ensuite on crée un tableau *T* à *n* cases, indicé par les sommets et initialisé à 0 ; puis on parcourt à nouveau les sommets de *S* et on incrémente au fur et à mesure les valeurs contenues dans *T* : à la fin, les valeurs de *T* donnent le nombre d'occurrences des sommets dans *S*. Il est facile d'en déduire si *S* est bien un cycle hamiltonien. On peut aussi, lors de ce parcours, calculer la longueur de *S* ; en la comparant à *k*, on vérifie ainsi la partie 2.

Toutes ces vérifications nécessitent un nombre d'opérations en O(*n*), ce qui est bien majorable par un polynôme en $\log_2(k+1) + \sum_{\{i,j\}\in G} \log_2(val(i,j)+1)$, ordre de grandeur de la taille de l'instance de *RVC*.

On a pu ainsi vérifier en un temps polynomial qu'une solution pressentie pour donner la réponse « oui » la donnait bien. Par conséquent, *RVC* est dans NP.

Par ailleurs, considérons un problème polynomial. Il est toujours possible de vérifier que la réponse est « oui » en un temps polynomial, pour toute instance : il suffit de résoudre, en un temps polynomial, le problème pour l'instance considérée et de voir si la réponse est « oui » ou « non ». D'où la proposition suivante :

Proposition. $P \subseteq NP$.

Nous avons défini assez rigoureusement les problèmes de la classe P, un peu plus intuitivement ceux de la classe NP. Il se trouve qu'à ce jour on ne sait pas si ces deux classes coïncident ou si l'inclusion de la classe P dans la classe NP est stricte. On peut néanmoins s'intéresser aux problèmes de NP qu'on peut considérer comme « les plus difficiles » de cette classe, au sens où l'existence d'un algorithme polynomial pour résoudre un tel problème entraînerait l'existence d'algorithmes polynomiaux pour résoudre n'importe quel problème de NP : ce sont les problèmes NP-complets.

XII.2.3. Problèmes NP-complets

Transformation polynomiale

Une notion essentielle dans la théorie de la NP-complétude est celle de *transformation polynomiale* : étant donnés deux problèmes de décision D_1 et D_2, nous écrirons $D_1 \prec D_2$ si les conditions suivantes sont réalisées :

1. Il existe une application *f* qui transforme toute instance *I* de D_1 en une instance *f*(*I*) de D_2, et un algorithme polynomial, par rapport à la taille de *I*, pour calculer *f*(*I*).

2. Il y a équivalence entre les deux énoncés « D_1 admet la réponse "oui" pour l'instance *I* » et « D_2 admet la réponse "oui" pour l'instance *f*(*I*) ».

Si $D_1 \prec D_2$ et s'il existe un algorithme polynomial pour résoudre D_2, alors il existe un algorithme polynomial pour résoudre D_1. En effet, pour obtenir la réponse relative à l'instance *I* de D_1, il suffit de transformer *I* polynomialement en l'instance *f*(*I*) de D_2 et de résoudre celle-ci, toujours polynomialement ; comme *f* conserve les réponses, on résout *I* en résolvant *f*(*I*). On peut interpréter la relation $\prec$ en disant que D_1 n'est pas plus difficile que D_2, au sens introduit à la fin de la partie précédente. Il est facile de plus de montrer que $\prec$ est une relation transitive : si $D_1 \prec D_2$ et $D_2 \prec D_3$, alors $D_1 \prec D_3$.

Définition. *Un problème Q est dit NP-complet s'il est dans la classe NP et si, pour tout problème Q´de la classe NP, on a Q´* $\prec$ *Q.*

Remarquons tout de suite des aspects importants de cette définition : si un problème NP-complet s'avère polynomial, alors P = NP ; si *Q* et *Q´*sont deux problèmes de la classe NP, si *Q* est NP-complet et si $Q \prec Q'$ alors *Q´* est NP-complet. Enfin, notons que, dans le cas $P \neq NP$, les problèmes NP-complets et les problèmes polynomiaux sont loin d'épuiser la classe NP ; si $P \neq NP$, on peut montrer au contraire qu'il existe un nombre infini de classes de problèmes de NP de difficulté « intermédiaire » : par rapport à ces classes, P est la classe « la moins difficile » et les problèmes NP-complets constituent la classe « la plus difficile ».

NP-complétude du problème de satisfiabilité

Un problème essentiel posé par la définition précédente est le suivant : existe-t-il des problèmes NP-complets ? Une réponse positive à cette question est fournie par le théorème de Cook, que nous ne démontrerons pas ici. Il porte sur un problème de logique appelé « Satisfiabilité » (ou *SAT*). Pour le définir, on considère un ensemble *U* de variables booléennes ; à chaque variable *u* est associée la variable $\overline{u}$ appelée variable conjuguée de *u* et

qui prend la valeur « vrai » quand *u* prend la valeur « faux » et réciproquement ; les variables et leurs conjuguées sont appelées des *littérales*. Si v_1, v_2, …, v_k sont des littérales, l'ensemble $\{v_1, v_2, \ldots, v_k\}$ est appelé une *clause* ; on dit qu'une clause *c* prend la valeur « vrai » si et seulement si au moins une littérale de *c* possède la valeur « vrai ».

Théorème de Cook. *Le problème suivant est NP-complet :*

Nom : SAT
Données : étant donnés un ensemble U de variables booléennes et un ensemble C de m clauses $C_1, C_2, \ldots, C_m$ *définies à partir de U,*
Question : existe-t-il une fonction f définie de U dans {« vrai », « faux »} telle que chaque clause C_i $(1 \leq i \leq m)$ *possède au moins une littérale* v_i *vérifiant* $f(v_i)$ *= « vrai » ?*

Si la réponse est « oui », on dit que *C* est *satisfiable*.

Un exemple de problème NP-complet : le problème 3-SAT

Le théorème de Cook étant supposé acquis, nous allons voir, sur un exemple, comment on peut démontrer qu'un problème *Q* est NP-complet. Les étapes d'une telle démonstration sont :

1. Prouver que *Q* est dans NP.
2. Choisir un problème Q' NP-complet *ad hoc*.
3. Exhiber une transformation polynomiale permettant d'établir : $Q' \prec Q$.

Nous allons établir le théorème suivant, qui montre que le problème de satisfiabilité reste NP-complet si on impose aux clauses de contenir exactement trois littérales distinctes ; ce problème s'appelle 3-*SAT*.

Théorème. *Le problème suivant est NP-complet :*

Nom : 3-SAT
Données : étant donnés un ensemble U de variables booléennes et un ensemble de m clauses $C_1, C_2, \ldots, C_m$ *définies à partir de U et telles que, pour* $1 \leq i \leq m$, $|C_i| = 3$,
Question : existe-t-il une fonction f définie de U dans {« vrai », « faux »} telle que chaque clause C_i $(1 \leq i \leq m)$ *possède au moins une littérale* v_i *vérifiant* $f(v_i)$ *= « vrai » ?*

Preuve

1. 3-*SAT* est dans NP. En effet, on peut tester, en un temps polynomial, qu'une suite de valeurs proposées pour les variables donne bien la valeur « vrai » à toutes les clauses. Il suffit pour cela de tester les valeurs des trois littérales de chaque clause, ce qui nécessite 3*m* tests. Le lecteur n'aura pas de difficulté à montrer que cette complexité est polynomiale par rapport à la taille des données. (On aurait pu aussi constater qu'il s'agit d'un sous-problème de *SAT*, qui est NP, et que tout sous-problème d'un problème de NP est lui-même dans NP lorsqu'on ne change pas qualitativement la taille du codage, ce qui est bien le cas ici...)

2. Nous n'avons pas, pour le moment, vraiment le choix du problème NP-complet : nous prenons $Q' = SAT$.

3. Soit U un ensemble de variables booléennes et $C = \{C_1, C_2, ..., C_m\}$ un ensemble de m clauses, constituant une instance de *SAT*. Nous allons construire une instance de 3-*SAT*, c'est-à-dire un ensemble C' de clauses dont chacune aura 3 littérales, sur un ensemble U' de variables, telle que C' soit satisfiable si et seulement si C l'est. Pour cela, nous allons transformer localement les clauses de l'instance de *SAT* en clauses de 3 littérales, en ajoutant éventuellement de nouvelles variables booléennes.

Considérons une clause $C_j = \{z_1, z_2, ..., z_k\}$ (en fait, on devrait écrire $\{ z_1^j , z_2^j , ..., z_{k_j}^j \}$; afin d'alléger l'écriture, on ne fait pas apparaître l'indice j) où les z_i sont des littérales dérivées des variables de U et formons de nouvelles clauses de trois littérales comme suit (cela dépend du nombre k de littérales utilisées par la clause C_j). Les variables qui seront ajoutées lors de la transformation de C_j ne seront utilisées qu'à cette fin et n'interviendront pas pour le traitement des autres clauses.

- si $k = 1$, on ajoute deux nouvelles variables v_j et w_j et on construit les quatre clauses à trois littérales $\{z_1, v_j, w_j\}$, $\{z_1, v_j, \overline{w_j}\}$, $\{z_1, \overline{v_j}, w_j\}$, $\{z_1, \overline{v_j}, \overline{w_j}\}$;
- si $k = 2$, on ajoute une nouvelle variable v_j et on construit les deux clauses à trois littérales $\{z_1, z_2, v_j\}$, $\{z_1, z_2, \overline{v_j}\}$;
- si $k = 3$, on n'ajoute pas de variable et on conserve C_j sans rien changer ;
- si $k > 3$, on ajoute $k - 3$ nouvelles variables $v_{j,i}$, pour $1 \leq i \leq k - 3$, et on construit les $k - 2$ clauses suivantes, à trois littérales : la clause $\{z_1, z_2, v_{j,1}\}$, les $k - 4$ clauses $\{\overline{v_{j,i}}, z_{i+2}, v_{j,i+1}\}$ pour $1 \leq i \leq k - 4$, la clause $\{\overline{v_{j,k-3}}, z_{k-1}, z_k\}$.

Une étude attentive des nouvelles clauses permet de constater que celles-ci ne peuvent être satisfaites simultanément que si C_j est satisfiable, et réciproquement. Par exemple, dans le dernier cas, si on suppose toutes les littérales z_i à « faux », alors il faut, pour satisfaire la clause $\{z_1, z_2, v_{j,1}\}$, mettre $v_{j,1}$ à « vrai » ; ce qui impose, pour satisfaire la clause suivante, de mettre aussi $v_{j,2}$ à « vrai », et ainsi de suite pour toutes les variables ajoutées $v_{j,i}$; mais alors la dernière clause ne peut pas être satisfaite. Réciproquement, supposons que C_j soit satisfaite parce qu'on a attribué la valeur « vrai » à la littérale z_{i+2} avec $1 \leq i \leq k - 4$ (les autres cas se traitent d'une façon similaire et sont laissés au lecteur) ; alors il est facile de satisfaire toutes les nouvelles clauses en mettant les variables $v_{j,l}$ à « vrai » pour $1 \leq l \leq i$ et à « faux » pour les autres.

Les variables de l'instance de 3-*SAT* que l'on crée sont les variables de l'instance de *SAT* et celles qu'on ajoute au cours de la transformation décrite ci-dessus ; les clauses de cette instance de 3-*SAT* sont celles que l'on a construites selon cette transformation. D'après ce qui vient d'être dit, et grâce à la « localité » des nouvelles variables, l'instance de 3-*SAT* ainsi créée est satisfiable si et seulement si l'instance de départ de *SAT* l'est.

De plus, cette transformation est bien polynomiale puisque le traitement d'une clause C_j à k littérales nécessite la construction d'au plus $\text{Max}(4, k - 2)$ clauses et l'introduction d'au plus $\text{Max}(2, k - 3)$ nouvelles variables booléennes.

Cette transformation respecte donc tous les critères permettant d'établir que l'on a *SAT* $\prec$ 3-*SAT*. Comme de plus *SAT* est NP-complet et que 3-*SAT* est dans NP, ceci montre la NP-complétude de 3-*SAT*.

On remarquera que, pour être valide, une transformation ne doit pas dépendre de la réponse de l'instance de départ. Ainsi, dans ce qu'on vient de proposer, la transformation est décrite sans qu'on sache si la réponse à l'instance de départ est « oui » ou « non ». On remarquera aussi qu'il n'est pas nécessaire d'atteindre toutes les instances de 3-*SAT* ; en revanche, il est indispensable de transformer toutes les instances de *SAT* en instances de 3-*SAT*.

Conséquences de la NP-complétude d'un problème

À quoi peut bien servir la démonstration, souvent longue et compliquée, du fait qu'un problème est NP-complet ?

Tout d'abord, cette preuve peut permettre d'éviter de perdre du temps à rechercher un algorithme polynomial pour résoudre le problème : n'en pas trouver ne constitue pas une preuve d'incompétence, ou alors la même preuve vaut aussi pour les centaines de chercheurs qui se sont attaqués, en vain, à la même investigation pour un très grand nombre d'autres problèmes NP-complets, en logique, en théorie des nombres, en théorie des graphes, etc.

Par ailleurs, si l'on persévère, nonobstant cette découverte, dans la recherche d'une méthode exacte, on doit se tourner vers les méthodes comme celles qui seront présentées dans les deux chapitres suivants (méthodes par séparation et évaluation, dites encore *branch and bound*, et programmation dynamique).

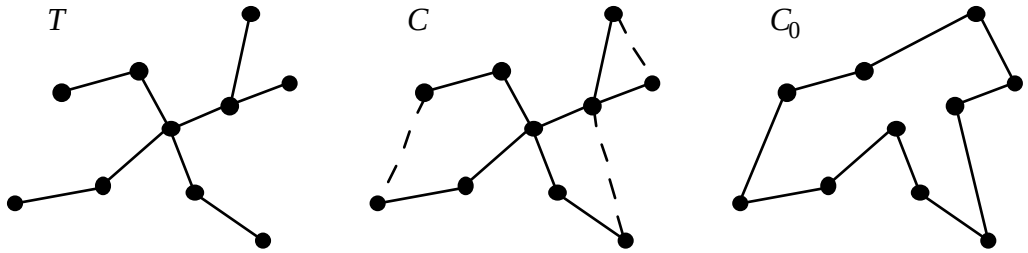
Enfin, cela justifie que l'on recherche des heuristiques, c'est-à-dire des algorithmes qui, du moins pour des problèmes énoncés comme problèmes d'optimisation, donnent, à défaut d'un optimum, une valeur approchée de ce dernier.

Pour notre problème favori dans ce chapitre, c'est-à-dire le problème du voyageur de commerce, nous ne démontrerons pas ici qu'il est NP-complet. Nous nous contenterons de donner un algorithme approché qui, dans le cas où les longueurs vérifient l'inégalité triangulaire, fournit un cycle hamiltonien dont la longueur n'est pas plus qu'une fois et demie la longueur d'une solution optimale : il s'agit de l'heuristique de N. Christofidès.

Heuristique de N. Christofidès

On considère une instance du problème du voyageur de commerce constituée d'un graphe G complet valué, avec une valuation qui satisfait l'hypothèse de l'inégalité triangulaire, dans lequel on cherche un cycle hamiltonien de longueur minimum.

L'heuristique de Christofidès consiste à construire un arbre couvrant T de longueur minimum du graphe, puis à construire sur l'ensemble des sommets de degré impair de cet arbre un couplage maximum M de longueur minimum (comme il y a un nombre pair de sommets de degré impair et que le graphe est complet, tous ces sommets pourront être couplés). On peut montrer que le graphe H ayant pour ensemble d'arêtes la réunion des arêtes de T et de celles de M possède un cycle (en général non élémentaire) C passant au moins une fois par toutes les arêtes de H. On construit un cycle hamiltonien C_0 de G en reliant les sommets selon l'ordre dans lequel on les rencontre sur C et en « sautant » les sommets déjà rencontrés. La figure suivante illustre ces différentes étapes (le cycle hamiltonien C_0 que l'on obtient n'est pas unique, il dépend de la façon dont on décide de parcourir C).



On admettra que toutes ces opérations peuvent se réaliser en un temps polynomial (plus précisément, en $O(n^3)$, si n est le nombre de sommets de G). La preuve du facteur 1,5 est laissée en exercice au lecteur.

XII.2.4. Problèmes NP-difficiles

Nous ne donnerons ici qu'une idée très informelle de ce qu'est un problème NP-difficile. Contrairement aux problèmes de la classe NP (et donc en particulier aux problèmes NP-complets et aux éléments de P), on ne s'intéresse plus exclusivement aux problèmes de reconnaissance. Informellement, un problème *NP-difficile* est un problème au moins aussi difficile qu'un problème NP-complet. On en déduit immédiatement que les problèmes NP-complets sont aussi NP-difficiles et que ce sont les seuls problèmes NP-difficiles de NP. Mais il existe des problèmes de reconnaissance qui sont NP-difficiles sans être dans NP. De plus, on peut montrer le résultat suivant :

Proposition. *Soit O un problème d'optimisation. Si le problème de décision associé à O est NP-complet, alors O est lui-même NP-difficile.*

Par exemple, il résulte de tout ce qui a été dit dans ce chapitre que le problème appelé plus haut *OVC* est NP-difficile.

XII.3. Exercices

Exercice 1

On appelle *clique* dans un graphe non orienté un ensemble de sommets deux à deux adjacents. Que peut-on bien appeler « problème de la clique maximum » ? Comment s'exprime le problème de décision associé, qu'on appellera *CLIQUE* ? Prouver que ce problème est NP-complet.

INDICATION

On pourra compléter la transformation suivante pour réduire *SAT* à *CLIQUE*. Soit une instance quelconque de *SAT*, définie par un ensemble U de variables booléennes et un ensemble C de m clauses $C_1, C_2, ..., C_m$ définies sur les littérales dérivées de U : pour $1 \leq i \leq m$, $C_i = \{z_{i,1}, z_{i,2}, ..., z_{i,r_i}\}$ avec $z_{i,j} \in U$ ou $\overline{z_{i,j}} \in U$ pour $1 \leq j \leq r_i$; $z_{i,j}$ est donc la j^e littérale de la i^e clause. On construit le graphe G dans lequel on cherche une clique d'une certaine taille (qu'on précisera) de la manière suivante. Les sommets de G sont des couples (i, j), i faisant référence au numéro d'une clause de *SAT*, et j au numéro d'une littérale de cette clause : la clause $C_i = \{z_{i,1}, z_{i,2}, ..., z_{i,r_i}\}$ donnera les r_i sommets $(i, 1)$, $(i, 2)$, …, (i, r_i) ; les sommets (i, j) et (r, s) sont adjacents dans G si et seulement si $i \neq r$ (les arêtes ne relient que des sommets provenant de clauses différentes) et $z_{i,j} \neq \overline{z_{r,s}}$.

Exercice 2

On appelle *transversal* dans un graphe non orienté un ensemble de sommets rencontrant toutes les arêtes. Que peut-on bien appeler « problème du transversal minimum » ? Comment s'exprime le problème de décision associé, qu'on appellera *TRANSVERSAL* ? Prouver que ce problème est NP-complet.

INDICATION

On pourra transformer *CLIQUE* en *TRANSVERSAL* en montrant que le complémentaire d'un transversal d'un graphe G définit une clique du graphe $\overline{G}$ ayant le même ensemble de sommets que G et tel que deux sommets distincts sont adjacents dans $\overline{G}$ si et seulement si ce n'est pas le cas dans G, et réciproquement.

XIII. Méthodes par séparation et évaluation

XIII.1. Introduction

Un problème de programmation linéaire en nombres entiers est un problème de programmation linéaire standard avec des *contraintes d'intégrité*, c'est-à-dire qu'on impose aux variables d'être entières. Autrement dit, si n désigne le nombre de variables, c'est un problème de la forme :

$$\text{Maximiser } c.x$$
$$\text{avec les contraintes } \begin{cases} Ax \leq b \\ x \in \mathrm{IN}^n \end{cases}$$

Alors qu'il existe des algorithmes polynomiaux pour résoudre les problèmes de programmation linéaire en variables réelles (l'algorithme du simplexe n'en est pas un), ce qui établit que ces problèmes sont polynomiaux, le problème de décision associé au problème de programmation linéaire en variables entières précédent est NP-complet.

Comme il a été dit au chapitre XII, parmi les conséquences du fait qu'un problème est NP-complet figurent les propositions suivantes :

1. On ne connaît pas d'algorithme polynomial pour résoudre ce problème.

2. Si l'on connaissait un tel algorithme, alors on aurait automatiquement des algorithmes polynomiaux pour résoudre tous les problèmes de la classe NP, c'est-à-dire à l'heure actuelle un très grand nombre de problèmes répertoriés, venant d'horizons divers, et pour certains très étudiés ; de là à penser qu'il n'existe pas de tel algorithme...

3. Avoir prouvé qu'un problème est NP-complet a des conséquences sur la façon dont on cherche à le résoudre.

- Lorsque l'on doit traiter des problèmes de grande dimension, on peut s'attendre à ce que les algorithmes exacts ne réussissent pas à donner la solution optimale... faute de temps. On applique alors des méthodes appelées *heuristiques*, qui sont censées donner, en un temps raisonnable, une approximation de la solution (sans que l'on puisse parfois dire beaucoup de la façon dont elles approchent l'optimum). Les trois derniers chapitres de ce livre présentent certaines d'entre elles.
- Pour les algorithmes exacts, on fera appel à des algorithmes comme la *programmation dynamique* (qui fera l'objet du chapitre suivant) ou comme les *méthodes par séparation et évaluation*, appelées aussi *méthodes arborescentes* ou encore *branch and bound*.

Dans ce chapitre, nous allons expliquer le fonctionnement des méthodes par séparation et évaluation à l'aide du problème du sac à dos ; puis nous indiquerons une façon, parmi d'autres, d'appliquer ces méthodes au problème du voyageur de commerce.

XIII.2. Problème du sac à dos : méthodes heuristiques

Supposons que nous désirions constituer le contenu d'un sac à dos et que nous ayons envie d'y mettre n objets de volumes respectifs v_1, ..., v_n. Ayant constaté que la somme des volumes des n objets était supérieure au volume V du sac à dos, nous affectons à chaque objet une utilité u_1, ..., u_n. Nous modélisons alors le problème en introduisant n variables dites *variables de décision* x_1, ..., x_n, à valeurs dans {0, 1}, la valeur 0 attribuée à la variable x_i signifiant que nous n'emporterons pas l'objet i, la valeur 1 que nous le prenons. Si nous voulons avoir le sac le plus utile, nous voyons que nous sommes amenés à résoudre le problème suivant :

$$\text{Max} \sum_{j=1}^{n} u_j.x_j$$

avec les contraintes $\begin{cases} \sum_{j=1}^{n} v_j.x_j \leq V \\ x_j \in \{0,1\} \text{ pour } 1 \leq j \leq n \end{cases}$

Il s'agit là d'un problème de programmation linéaire en variables bivalentes (on dit aussi « en 0-1 ») à une seule contrainte, connu sous le nom de *problème du sac à dos* (*knapsack problem* en anglais). Cependant, par abus de langage, on parlera encore de problème de sac à dos (ou parfois de *problème de sac à dos généralisé*) lorsque la contrainte reste unique mais que les variables peuvent être entières positives ou nulles et non plus nécessairement bivalentes. C'est ce problème (variables entières) que nous considérons dans ce qui suit.

XIII.2.1. Première heuristique

Pour résoudre le problème du sac à dos (généralisé), on peut avoir l'idée de « relâcher les contraintes d'intégrité », c'est-à-dire résoudre le problème en variables réelles, puis d'arrondir les solutions obtenues en prenant leurs parties entières par défaut. Nous allons voir que cette méthode ne peut être considérée que comme une heuristique sur l'exemple suivant :

$$\text{Maximiser } 10x_1 + 8x_2 + 5x_3$$

avec les contraintes $\begin{cases} 6x_1 + 5x_2 + 4x_3 \leq 9 \\ x_j \in \text{IN pour } 1 \leq j \leq 3 \end{cases}$

La relaxation des contraintes d'intégrité conduit à un problème de programmation linéaire dont la résolution, immédiate par la méthode du simplexe (*cf.* exercice 1), donne $x_1^* = 1{,}5$, $x_2^* = 0$, $x_3^* = 0$ et $z^* = 15$. La méthode des arrondis donne $x_1^* = 1$, $x_2^* = x_3^* = 0$ et $z^* = 10$, alors que le problème admet la solution $x_1^* = 0$, $x_2^* = 1$, $x_3^* = 1$ et $z^* = 13$, dont nous verrons qu'elle est optimale.

XIII.2.2. Seconde heuristique

On peut tout de suite imaginer une autre heuristique pour le problème du sac à dos (généralisé). Une approche de style qualité/prix, chère aux unions de consommateurs, incite à un classement des variables dans l'ordre décroissant des rapports utilité/volume. On peut alors examiner les variables dans cet ordre et choisir de donner, à la variable examinée, la plus grande valeur entière permettant de satisfaire à la contrainte compte tenu des choix déjà faits. Un tel algorithme est dit « glouton » parce qu'on traite les variables les unes après les autres en prenant des décisions que l'on ne remet pas en cause.

Pour notre exemple, les rapports en question sont 5/3 pour x_1, 8/5 pour x_2 et 5/4 pour x_3. L'ordre induit par l'examen des rapports est donc x_1, x_2, x_3. Compte tenu de la contrainte, la plus grande valeur possible pour x_1 est 1. Ayant donné la valeur 1 à x_1, on voit qu'on doit donner la valeur 0 à x_2 et à x_3. On retrouve ici la même solution que celle fournie par la méthode précédente (mais c'est un hasard ; cette seconde heuristique est en fait toujours au moins aussi bonne que la première). Nous voyons aussi que pour l'une comme pour l'autre on ne peut parler que d'heuristique.

XIII.3. Méthode par séparation et évaluation pour le problème du sac à dos

Ce type de méthode s'applique à la résolution de problèmes d'optimisation combinatoire avec un grand nombre de solutions envisageables et, en particulier, à la résolution de problèmes de programmation linéaire en nombres entiers où l'on peut définir la notion de solution réalisable, c'est-à-dire de solution satisfaisant les contraintes. Nous allons associer à cette méthode une arborescence dont les sommets correspondent à des ensembles de solutions réalisables, la racine de l'arborescence contenant l'ensemble de toutes ces solutions réalisables. L'expansion de cette arborescence est régie selon quatre ingrédients : un *principe de séparation*, un *principe d'évaluation*, l'utilisation d'une *borne* et une *stratégie de développement*.

XIII.3.1. Principe de séparation

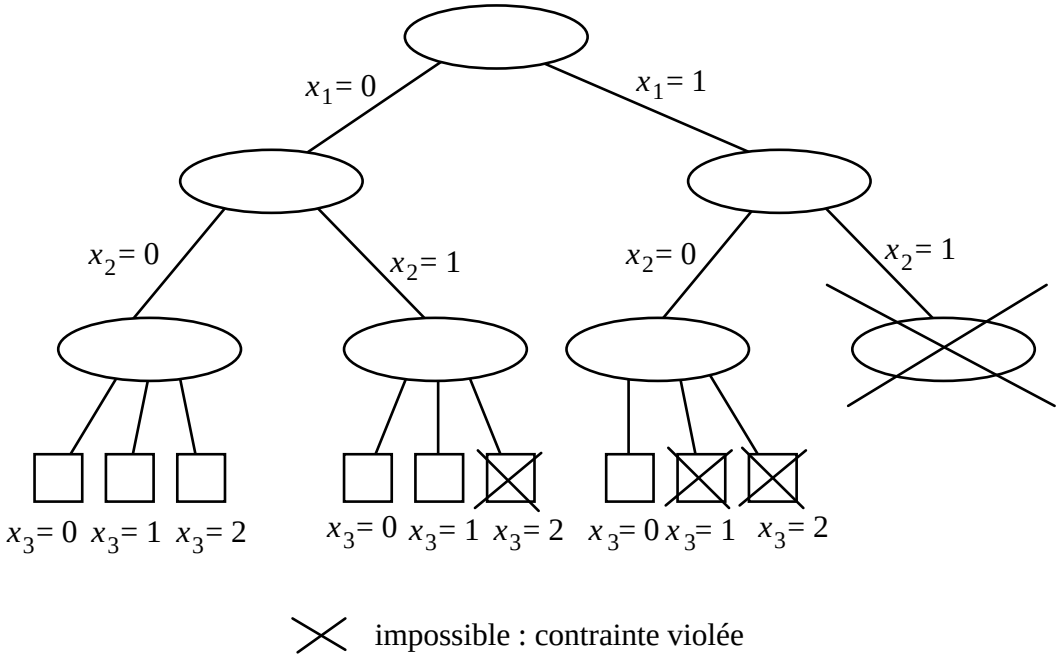
Le mot *séparation* représente le fait que nous partageons, en fonction d'un certain critère, l'ensemble des solutions réalisables contenues dans un sommet de cette arborescence (ensemble que nous ne connaissons pas explicitement) en sous-ensembles, ceux-ci devenant dans l'arborescence les fils du sommet considéré. La seule exigence dans ce partage est que nous ne perdions ni n'ajoutions de solution (autrement dit, l'union des sous-ensembles associés aux fils d'un sommet doit être égale à l'ensemble associé à ce sommet). Si on n'appliquait que le principe de séparation, celle-ci serait effectuée pour tout sommet contenant plus d'une solution.

Nous allons illustrer le principe de séparation seul à partir du problème de sac à dos (généralisé) considéré plus haut. Pour cet exemple, nous pouvons appliquer les critères de branchement suivants :

- une première séparation sera effectuée selon les valeurs de x_1 ; comme, d'après la contrainte, x_1 ne peut prendre que les valeurs 0 ou 1, nous séparons l'ensemble de toutes les solutions associé à la racine en deux sous-ensembles : l'un contient toutes les solutions pour lesquelles x_1 vaut 0 et l'autre contient celles pour lesquelles x_1 vaut 1 ;
- une deuxième séparation sera faite de la même façon suivant les valeurs possibles de x_2, à savoir 0 ou 1 ;
- enfin, une troisième séparation aura lieu selon les valeurs possibles de x_3 : 0, 1 ou 2, sauf pour le sommet vide correspondant aux choix $x_1 = x_2 = 1$.

Le dessin suivant montre ce qu'est l'arborescence obtenue pour notre exemple. Les carrés y représentent les cas pour lesquels les trois variables sont entièrement déterminées. Les sommets de l'arborescence barrés d'une croix en sont les sommets vides.

Il suffit, pour connaître la solution optimale, de calculer la valeur de la fonction objectif pour toutes les feuilles non vides de l'arborescence obtenue. Cette méthode peut être améliorée pour éviter l'examen de certaines branches. Il existe en effet deux raisons qui permettent de ne pas développer un sommet de l'arborescence : lorsque l'on peut montrer que ce sommet ne contient pas la solution optimale et lorsque que l'on sait résoudre directement le problème correspondant à ce sommet. Le principe d'évaluation et la borne peuvent fournir de telles indications.



XIII.3.2. Principe d'évaluation et utilisation de la borne

Dans un problème d'optimisation écrit sous forme de maximisation d'un certain objectif, la *borne* correspond à une valeur qu'on sait atteindre pour l'objectif, à l'aide d'une certaine solution réalisable, et qui est donc par définition un minorant du maximum. Ainsi, pour le problème de sac à dos (généralisé) abordé plus haut, les heuristiques précédentes ont montré que 10 est une borne inférieure du maximum cherché.

Toujours dans le cas où on cherche un maximum, *évaluer un sommet*, c'est déterminer un

majorant de l'ensemble des valeurs de l'objectif correspondant aux solutions contenues en ce sommet. Ainsi, ayant évalué un sommet *S*, on saura que pour l'ensemble des solutions correspondant à *S*, on ne peut pas faire mieux que la valeur de l'évaluation. Une évaluation d'un sommet est dite *exacte* si la valeur donnée par l'évaluation est atteinte par un élément de l'ensemble associé au sommet. Pour l'exemple précédent, la relaxation des contraintes d'intégrité avait permis d'obtenir la valeur 15. Ceci est donc un majorant du maximum cherché en valeurs entières (en effet, quand on relâche les contraintes d'intégrité, le domaine des solutions réalisables grandit et le maximum dans ce domaine ne peut qu'être supérieur ou égal à celui du domaine en valeurs entières), et on peut attribuer cette valeur comme évaluation de la racine. Cette évaluation n'est pas exacte, car la solution qui donne 15 n'est pas à composantes entières.

La borne et l'évaluation permettent de ne pas développer un sommet *S* dans les deux cas suivants (toujours pour une maximisation) qui viennent donc s'ajouter au cas précédent (les contraintes ne sont pas satisfaites) :

- l'évaluation de *S* est inférieure ou égale à la borne ;
- l'évaluation de *S* est exacte ; dans ce cas, si cette évaluation est strictement supérieure à la borne (sinon on est dans le cas précédent), on a exhibé une solution meilleure que celle associée à la borne : on modifie donc la borne et on se retrouve dans le cas précédent.

On voit qu'on a intérêt à calculer une bonne borne et à concevoir une fonction d'évaluation assez fine afin d'éliminer le plus tôt possible le plus de branches possible. Mais en même temps, ces calculs doivent pouvoir être effectués en un temps raisonnable...

Si on récapitule les cas qui permettent de ne pas séparer un sommet *S*, on obtient finalement :

1. On sait que toutes les solutions contenues dans *S* sont au plus aussi bonnes qu'une solution qu'on connaît déjà (par comparaison entre l'évaluation de *S* et la borne courante).

2. *S* est vide ou on connaît la meilleure des solutions contenues dans *S*, soit parce que l'évaluation est exacte, soit parce qu'une méthode directe a permis de conclure pour ce sommet.

XIII.3.3. Stratégie de développement

Le quatrième ingrédient consiste à déterminer la façon dont on va construire l'arborescence, c'est-à-dire dans quel ordre on va appliquer le critère de séparation aux sommets de l'arborescence.

Dans une première stratégie, appelée « en profondeur », on peut faire une construction en profondeur d'abord en descendant dans les branches jusqu'à ce qu'on trouve un sommet que l'on peut éliminer, auquel cas on remonte pour redescendre dans une autre direction (s'il en reste). Si l'arborescence était connue explicitement, cette construction correspondrait à l'application d'un parcours en profondeur à cette arborescence (*cf.* chapitre VIII). Cette façon de procéder présente plusieurs avantages, principalement au niveau de la place mémoire : l'encombrement en place mémoire est souvent relativement peu important, puisqu'il ne faut conserver que la description de la branche qu'on explore (il n'est pas nécessaire de conserver la description complète de l'arborescence) ; elle possède également des avantages au niveau des accès disques (en effet bien souvent on ne peut traiter tout le problème, vu sa taille, en

mémoire centrale) ; enfin, elle permet plus facilement d'exploiter, quand il y en a, des informations disponibles au niveau du sommet qu'on examine et au moins partiellement utilisables pour les fils de celui-ci.

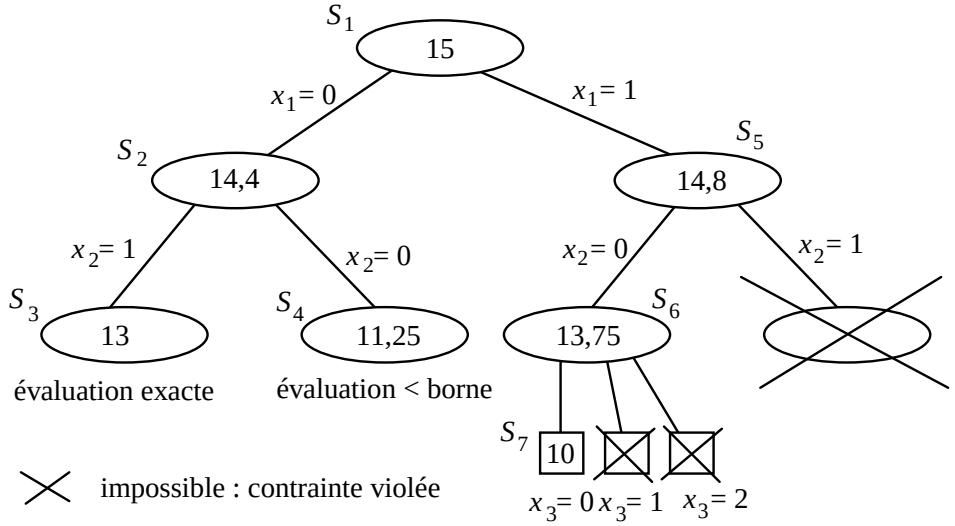
On peut aussi commencer par séparer les sommets qui ont l'évaluation la plus grande (pour une maximisation) en suspectant que ce sont eux qui contiennent la solution optimale : c'est dans le sommet de plus grande évaluation qu'on a l'espoir, justifié ou non, de trouver la meilleure solution. Cette stratégie, de type « meilleur d'abord », peut, pour certains problèmes, être plus rapide du fait qu'on essaie à chaque itération d'explorer la direction qui semble la plus prometteuse ; en revanche, la gestion de l'arborescence n'est pas triviale et consomme généralement beaucoup de place mémoire.

Enfin, on peut envisager une exploration en largeur d'abord, mais cette méthode est très rarement implémentée pour des questions d'efficacité et d'encombrement de la mémoire.

XIII.3.4. Application au problème de sac à dos

Nous pouvons reprendre l'étude de l'arborescence précédente en utilisant comme fonction d'évaluation la résolution du simplexe en variables réelles, qui est immédiate à effectuer dans le cas d'une seule contrainte : en effet, il est facile de se convaincre que, lorsque l'on applique l'algorithme du simplexe à un problème à une seule contrainte, si on choisit comme variable entrante la variable de choix de plus petit indice, les variables étant indicées dans l'ordre décroissant des rapports utilité/volume (*cf.* exercice 1), on obtient l'optimum en une itération. La stratégie retenue sera celle du parcours en profondeur d'abord. Enfin, on rappelle que la borne obtenue à l'aide des heuristiques vaut 10.

On obtient alors l'arborescence ci-dessous, en supposant que l'on a procédé à l'évaluation du sommet $x_1 = 0$, $x_2 = 1$ avant d'entamer la séparation du sommet $x_1 = x_2 = 0$, ce qui nous a permis d'améliorer la borne grâce à une évaluation de 13, qui était exacte. Les sommets de l'arborescence sont numérotés dans l'ordre selon lequel on les rencontre lors du développement en profondeur d'abord.



L'évaluation du sommet S_3 est obtenue par la solution $x_3 = 1$, en plus des choix $x_1 = 0$ et $x_2 = 1$. C'est donc une évaluation exacte. Par conséquent, il est inutile de développer ce sommet et on interrompt l'exploration de cette branche. De plus, la valeur obtenue, réalisable, est meilleure que la borne ; on change donc celle-ci, qui passe de 10 à 13.

Renonçant au développement de S_3, on remonte à S_2 pour examiner, s'il en existe, les autres choix relatifs à x_2, ce qui nous conduit à S_4 avec $x_2 = 0$. On obtient alors 11,25 comme évaluation de S_4. Celle-ci étant inférieure ou égale à la nouvelle borne, on ne développe pas S_4 et on remonte en S_2 puis en S_1, puisque aucun autre choix n'est possible pour x_2 au niveau de S_2.

Quand on arrive en S_5, la contrainte fait qu'il n'y a qu'une seule possibilité viable pour x_2 : 0, ce qui donne S_6. Les choix pour x_3, au niveau de S_6, conduisent soit à des solutions non réalisables, soit à un sommet S_7 d'évaluation exacte mais moins bonne que la borne actuelle. On interrompt donc l'exploration de S_6, on remonte en S_5 où il n'y a plus de prolongement réalisable, puis en S_1 où, de même, il n'y a plus de prolongement à envisager. On a donc terminé : le maximum cherché vaut 13 et est atteint par $x_1 = 0$, $x_2 = x_3 = 1$.

XIII.4. Application au problème du voyageur de commerce

Rappelons que le problème du voyageur de commerce est la recherche, dans un graphe complet valué, d'un cycle hamiltonien (c'est-à-dire passant une fois et une seule par tous les sommets) de valuation totale minimum. Ce problème est NP-difficile (plus précisément, le problème de reconnaissance associé est NP-complet). Nous allons montrer qu'il fait partie de la classe des problèmes de programmation linéaire en variables 0-1, dont il est d'ailleurs un représentant extrêmement célèbre. Nous décrirons ensuite une méthode par séparation et évaluation permettant de le résoudre.

XIII.4.1. Forme linéaire en 0-1 du problème du voyageur de commerce

Considérons le graphe complet $K_n = (X, E)$ à n sommets, c'est-à-dire le graphe d'ordre n où tous les sommets sont adjacents deux à deux. On suppose de plus qu'on a défini une valuation sur l'ensemble E des arêtes de ce graphe, autrement dit une application p de E dans R que l'on appelle « poids », le poids d'une arête $\{u, v\}$ étant noté p_{uv}. Associons à chaque arête $\{u, v\}$ de K_n une variable bivalente x_{uv} qui vaut 1 si on garde l'arête $\{u, v\}$ pour constituer le cycle hamiltonien, 0 sinon.

On peut représenter la contrainte de constituer un cycle hamiltonien à l'aide des contraintes suivantes :

- pour tout sommet v, la somme des valeurs des variables associées aux arêtes ayant v comme extrémité est égale à deux, ceci traduisant le fait que dans un cycle hamiltonien tout sommet est de degré deux ;
- pour tout sous-ensemble Y de X autre que X, le nombre d'arêtes ayant ses deux extrémités dans Y est strictement plus petit que $|Y|$; cette condition implique que l'ensemble des arêtes gardées ne se décompose pas en plusieurs cycles.

Toutes ces contraintes s'expriment bien linéairement par rapport à l'ensemble des variables du problème, comme le montre la forme suivante :

$$\text{Min} \sum_{\{u,v\}\in E} p_{uv}.x_{uv}$$

$$\text{avec les contraintes} \quad \begin{cases} \forall u \in X, \ \sum_{v\in X} x_{uv} = 2 \\ \forall Y \subset X \text{ avec } Y \neq X, \ \sum_{\substack{\{u,v\}\in E \\ u\in Y,\ v\in Y}} x_{uv} < |Y| \\ \forall \{u,\ v\} \in E,\ x_{uv} \in \{0,\ 1\} \end{cases}$$

XIII.4.2. Définition d'une fonction d'évaluation

Soit G un graphe dans lequel on cherche un cycle hamiltonien de poids minimum. Une évaluation possible d'un sommet de l'arborescence d'une procédure par séparation et évaluation est justifiée par les remarques triviales ci-après :

- une chaîne constitue un arbre particulier ; la chaîne obtenue en supprimant d'un cycle hamiltonien de G un sommet quelconque x_0 et les deux arêtes qui lui sont adjacentes est un arbre couvrant de $G - x_0$: elle est donc de poids supérieur ou égal au poids d'un arbre de poids minimum couvrant $G - x_0$;
- dans un cycle hamiltonien, tout sommet a un degré égal à 2 ; la somme des poids des deux arêtes d'un cycle hamiltonien adjacentes à un sommet x_0 fixé est supérieure ou égale à la somme des poids des deux arêtes les plus légères incidentes à x_0 dans G.

Choisissant de façon arbitraire un sommet x_0, le poids d'un cycle hamiltonien quelconque est donc supérieur ou égal à la somme des poids des deux arêtes les plus légères adjacentes à x_0 plus le poids d'un arbre de poids minimum couvrant $G - x_0$. Cette quantité constitue un minorant du poids d'un cycle hamiltonien optimal et pourra par conséquent constituer une fonction d'évaluation pour l'arborescence (l'évaluation doit en effet fournir un tel minorant puisque le problème de départ est un problème de minimisation).

XIII.4.3. Description d'une méthode par séparation et évaluation

Nous pouvons maintenant décrire, parmi d'autres, une méthode par séparation et évaluation utilisant l'évaluation définie ci-dessus (l'exercice 2 en donne une illustration).

On suppose qu'une borne a été calculée au préalable, par exemple à l'aide d'une des méthodes exposées dans les deux derniers chapitres de ce livre (faute de mieux, on peut toujours partir d'un cycle hamiltonien choisi au hasard, mais la borne qui en découlera risque d'être assez mauvaise).

Choisissons une fois pour toutes un sommet x_0, afin de procéder à l'évaluation décrite ci-dessus. Comme toujours, la racine de l'arborescence contient toutes les solutions réalisables possibles, c'est-à-dire ici l'ensemble de tous les cycles hamiltoniens du graphe. Arrivés à un sommet S de l'arborescence (y compris la racine), évaluons-le. Si l'évaluation de S est supérieure ou égale à la borne (attention, nous sommes en train de résoudre un problème de minimisation...), nous ne développons pas S et S est abandonné : nous savons déjà faire au moins aussi bien. Sinon, construisons le graphe partiel H correspondant à l'évaluation en ajoutant les deux arêtes adjacentes à x_0 de moindre poids dans le graphe courant G à celles

d'un arbre de poids minimum couvrant $G - x_0$: H est un graphe connexe d'ordre n possédant n arêtes. Si H est un cycle hamiltonien, par définition l'évaluation de S est exacte ; on peut alors abandonner S : on ne fera pas mieux, en ce qui concerne les solutions contenues dans S, que ce qu'on vient d'obtenir ; en outre si cette évaluation exacte est inférieure à la valeur actuelle de la borne, on remet celle-ci à jour. Si H n'est pas hamiltonien, puisqu'il est connexe et comporte n arêtes, c'est qu'il contient au moins un sommet de degré strictement supérieur à deux. Considérons un tel sommet et les i ($i \geq 3$) arêtes e_1, e_2, ..., e_i auxquelles il est adjacent dans H. Pour définir le critère de séparation, remarquons, puisqu'un cycle hamiltonien est un graphe dans lequel tous les sommets ont un degré égal à 2, qu'on ne perd aucune solution en considérant successivement celles qui ne contiennent pas e_1, puis celles qui ne contiennent pas e_2, … et ainsi de suite jusqu'à l'arête e_i ; le sommet S de l'arborescence sera donc séparé selon i branches, chacune d'entre elles correspondant à l'interdiction d'une des i arêtes e_k ($1 \leq k \leq i$). On traduit la contrainte « ne pas contenir l'arête e_k » par le fait de mettre le poids de e_k à l'infini dans le graphe courant G.

XIII.5. Exercices

Exercice 1

On considère un problème de programmation linéaire standard, à une seule contrainte, défini par

$$\text{Max} \sum_{j=1}^{n} u_j . x_j \text{ avec } \sum_{j=1}^{n} v_j . x_j \leq V \text{ et } x_j \geq 0 \text{ pour } 1 \leq j \leq n.$$

Tous les coefficients u_j et v_j sont supposés strictement positifs et l'on suppose les variables classées par rapports utilité/volume décroissants ou, pour rester dans un formalisme plus mathématique, suivant les valeurs décroissantes des rapports u_j / v_j . Montrer que la variable x_1 est entrante et que, en la faisant entrer en base, on atteint l'optimum de l'objectif en une seule étape. Exprimer la valeur de l'objectif en fonction des différents coefficients.

Exercice 2

Appliquer la méthode par séparation et évaluation décrite ci-dessus pour résoudre la problème du voyageur de commerce dans le graphe K_6 (c'est-à-dire le graphe complet à six sommets) dont la matrice des poids est la suivante :

	x	*y*	*z*	*t*	*u*	*v*
x	+∞	4	7	2	5	4
y	4	+∞	3	2	1	2
z	7	3	+∞	2	6	3
t	2	2	2	+∞	5	3
u	5	1	6	5	+∞	2
v	4	2	3	3	2	+∞

Que faudrait-il faire si on voulait connaître tous les cycles hamiltoniens de poids minimum ?

Corrigé des exercices

Chapitre VI

Corrigé de l'exercice 1

Considérons une chaîne élémentaire de longueur maximum, et appelons a et b ses extrémités. La chaîne ne pouvant être prolongée, les voisins de a par exemple sont sur cette chaîne qui contient donc au moins le sommet a et δ autres sommets, puisque a possède au moins δ voisins. La chaîne est donc de longueur au moins δ.

Considérons la même chaîne et l'arête qui lie a à un sommet c qui lui soit adjacent (un de ses voisins donc) et qui soit le plus loin possible de a sur la chaîne. L'union de cette arête $\{a, c\}$ et de la portion de chaîne entre a et c constitue un cycle ; or, cette portion de chaîne contient au moins $\delta + 1$ sommets, d'où le résultat annoncé.

Corrigé de l'exercice 2

Si on considère la somme, étendue à tous les sommets x de G, des degrés $d(x)$, on compte exactement deux fois chaque arête. Cette somme est donc égale à $2m$ et est par conséquent paire. Le nombre des termes impairs qui interviennent dans cette somme est donc nécessairement pair, comme l'indique l'énoncé. Par conséquent, un graphe ayant 5 sommets ne peut pas avoir tous ses sommets de degré impair et, en particulier, ne peut pas être cubique.

Corrigé de l'exercice 3

Nous établissons d'abord deux lemmes.

Lemme 1. *Si G est connexe d'ordre n, alors la taille m de G est au moins égale à $n - 1$.*

Preuve du lemme 1.— Elle se fait par récurrence sur n. Le résultat est vrai pour $n = 1$. Supposons-le vrai pour tout $p < n$. Considérons alors un graphe connexe G d'ordre n. Soient x un sommet quelconque et $G - x$ le sous-graphe de G engendré par $X - \{x\}$. Soit k le nombre de composantes connexes de ce sous-graphe et G_1, G_2, ..., G_k ses composantes connexes. D'après l'hypothèse de récurrence, la taille m_i de G_i est au moins $n_i - 1$, où n_i désigne l'ordre de G_i. Puisque le graphe était initialement connexe, x a au moins un voisin dans chacune des composantes ; en effet, pour aller d'un sommet d'une composante à un

sommet d'une autre composante, il est nécessaire de passer par *x*. Le nombre total *m* d'arêtes du graphe est donc : $m = \sum_{i=1}^{k} m_i + d(x) \geq \sum_{i=1}^{k} (n_i - 1) + k = n - 1$

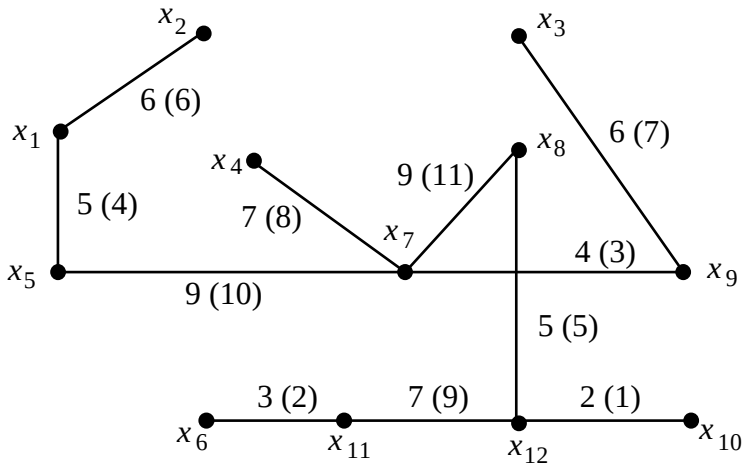
Lemme 2. *Si G est sans cycle et d'ordre n, alors la taille de G est au plus égale à n – 1.*

Preuve du lemme 2.— Elle se fait à nouveau par récurrence sur l'ordre de *G*, la proposition étant clairement vraie pour les petites valeurs de *n*. Si dans un graphe fini tout sommet est de degré au moins 2, alors le graphe contient au moins un cycle, puisque dans une promenade commençant en un sommet quelconque, on peut toujours repartir d'un sommet par une arête autre que celle par laquelle on est arrivé. Si un graphe est sans cycle, il contient donc nécessairement un sommet de degré 0 ou 1. Soit *x* un tel sommet. *G* étant sans cycle, il en est de même du sous-graphe $G - x$. Par hypothèse de récurrence, la taille de $G - x$ est au plus $n - 2$, et, par le choix de *x*, la taille de *G* est au plus $n - 1$.

Nous sommes maintenant en mesure de prouver les implications suivantes : a ⇒ b, a ⇒ c, b ⇒ d, c ⇒ e, d ⇒ f, e ⇒ f, f ⇒ a, et cette tâche, facile, est laissée au lecteur.

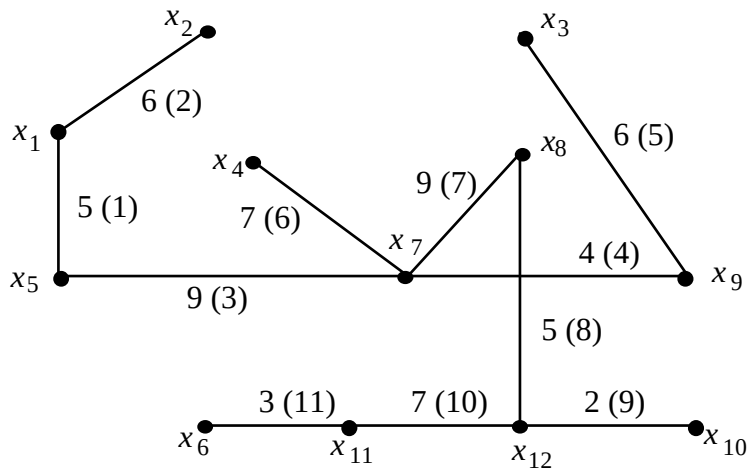
Corrigé de l'exercice 4

a. L'application de l'algorithme de Kruskal au graphe proposé donne l'arbre couvrant de poids minimum suivant.



L'arbre *T* ainsi déterminé est de poids 63. Sur chaque arête de *T*, il est indiqué, outre le poids de celle-ci, un nombre entre parenthèses qui donne le numéro d'entrée de cette arête dans l'arbre selon l'algorithme (l'ordre d'entrée des arêtes de même poids est arbitraire).

b. L'algorithme de Prim redonne ici l'arbre précédent (un arbre couvrant de poids minimum n'étant pas *a priori* unique, l'algorithme de Kruskal et celui de Prim peuvent donner des arbres différents, mais bien sûr de même poids) ; on a pris x_1 comme point de départ ; chaque arête porte deux nombres : son poids et, entre parenthèses, le numéro d'entrée de cette arête dans l'arbre selon l'algorithme.



Corrigé de l’exercice 5

Appelons $T = (e_1 \ldots e_k\ e_{k+1} \ldots e_{n-1})$ l’arbre couvrant construit par l’algorithme de Prim. Supposons qu’il n’est pas de poids minimum et appelons T_0 un arbre couvrant de poids minimum coïncidant aussi loin que possible avec T, c’est-à-dire que $T_0 = (e_1 \ldots e_k\ f_{k+1} \ldots f_{n-1})$, avec k le plus grand possible. L’arête e_{k+1} n’appartient pas à T_0, à cause du choix de T_0 donc $T_0 \cup e_{k+1}$ contient un cycle C. Au moment où on a choisi l’arête e_{k+1} dans la construction de T, on a relié une partie R (voir sa définition dans la description de l’algorithme) de G à $G - R$. Si on considère la chaîne $C - e_{k+1}$, elle joint, elle aussi, R à $G - R$ et une de ses arêtes au moins a une extrémité dans R et l’autre dans $G - R$. Appelons f une telle arête ; les arêtes e_i pour $1 \leq i \leq k$ ayant toutes leurs deux extrémités dans $G - R$, f n’est pas l’une d’entre elles ; d’autre part, f étant une arête de $C - e_{k+1}$, elle est dans T_0. D’après le choix de e_{k+1}, on a $poids(f) \geq poids(e_{k+1})$. Si l’inégalité est stricte, l’arbre $(T_0 \cup e_{k+1}) - f$ est de poids strictement inférieur à celui de T_0, ce qui est contradictoire avec le fait que T_0 est de poids minimum. S’il y a égalité, l’arbre $(T_0 \cup e_{k+1}) - f$ est de poids minimum et coïncide davantage avec T que T_0, et il y a encore contradiction avec le choix de T_0. Dans tous les cas, il y a une contradiction : c’est donc que T est de poids minimum.

Chapitre VII

Corrigé de l’exercice 1

Preuve du théorème 1.— La condition sur la non-vacuité de Y n’est qu’une façon de dire qu’il existe bien des chemins de x vers y. S’il n’y a pas de circuit absorbant rencontrant un chemin de x vers y, il n’y a aucun intérêt pour aller de x à y à emprunter un circuit (au pire cela rallonge, au mieux la longueur de ce circuit est nulle et cela ne raccourcit pas). On peut donc chercher les plus courts chemins parmi des chemins élémentaires et, comme ceux-ci sont en nombre fini, le problème d’un chemin de longueur minimum a bien une ou des solutions ; les autres plus courts chemins (donc non élémentaires) se déduisent des précédents en ajoutant d’éventuels circuits de longueur nulle.

Les preuves des théorèmes 2 et 3 sont tout à fait analogues.

Corrigé de l'exercice 2

Nous donnons ci-dessous la distance (*dist*) et le prédécesseur (*préd*, désigné par l'initiale du sommet) obtenus à l'initialisation et après chaque itération. À la fin de chaque étape, le sommet à examiner à l'itération suivante est encadré ; les valeurs en caractères gras dans le tableau des distances correspondent aux sommets qui restent à examiner.

dist, préd	Paris	Hamb.	Lond.	Amst.	Berlin	Stock.	Oslo	Édimb.	Rana
init.	**0**	∞	∞	∞	∞	∞	∞	∞	∞
étape 1	0	**7, P**	**4, P**	**3 , P**	∞	∞	∞	∞	∞
étape 2	0	**5, A**	**4 , P**	3, P	∞	∞	**11, A**	∞	∞
étape 3	0	**5** , A	4, P	3, P	∞	∞	**11, A**	**6, L**	∞
étape 4	0	5, A	4, P	3, P	**6** , H	**6**	**11, A**	**6, L**	∞
étape 5	0	5, A	4, P	3, P	6, H	**6**	**9, B**	**6, L**	∞
étape 6	0	5, A	4, P	3, P	6, H	6	**8, S**	**6 , L**	**11, S**
étape 7	0	5, A	4, P	3, P	6, H	6	**8** , S	6, L	**11, S**
étape 8	0	5, A	4, P	3, P	6, H	6	8, S	6, L	**10 , O**

Les valeurs *dist* indiquent que la longueur d'un plus court chemin de Paris à Rana vaut 10. Pour le reconstituer, utilisons les valeurs *préd*. Celles-ci nous indiquent que le prédécesseur de Rana sur ce plus court chemin est Oslo, que le prédécesseur d'Oslo est Stockholm, que le prédécesseur de Stockholm est Hambourg, que le prédécesseur de Hambourg est Amsterdam, que le prédécesseur d'Amsterdam est Paris : le plus court chemin de Paris à Rana est donc

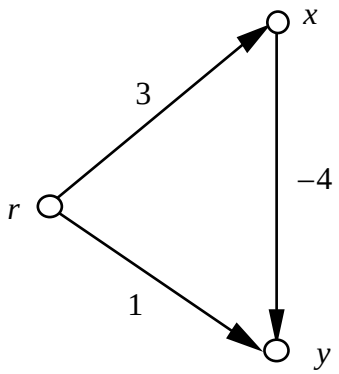
Paris → Amsterdam → Hambourg → Stockholm → Oslo → Rana.

Corrigé de l'exercice 3

Les sommets non encore dans A pour lesquels, à une étape donnée, la valeur de la fonction π est infinie sont des sommets qui n'ont pas de père dans A. Si, à une étape donnée, pour tout sommet x non dans A, $\pi(x) = \infty$, c'est qu'il n'existe pas d'arc dont l'origine soit dans A et l'extrémité non dans A. C'est dire qu'il n'existe aucun chemin de A vers $G - A$ et en particulier de r à x. La réciproque est évidente. Quant à la modification de l'algorithme lui-même, il faut simplement s'arrêter si, à une étape donnée, le minimum sur l'ensemble des sommets non dans A des valeurs de la fonction π est infini.

Corrigé de l'exercice 4

Appliquons l'algorithme de Dijkstra au graphe ci-dessous :



Initialisation : $\pi(r) \leftarrow 0$
Étape 1 : $\pi(x) \leftarrow 3$, $père(x) \leftarrow r$
$\pi(y) \leftarrow 1$, $père(y) \leftarrow r$
$pivot \leftarrow y$, $A \leftarrow \{r, y\}$
Étape 2 : $pivot \leftarrow x$, $A \leftarrow \{r, x, y\}$

L'arborescence obtenue, formée des arcs (r, x) et (r, y), n'est pas constituée des plus courts chemins, puisque le plus court chemin de r à y est $r \rightarrow x \rightarrow y$ et non $r \rightarrow y$.

L'algorithme de Dijkstra permet toujours de trouver une arborescence de racine r couvrant tous les sommets que l'on peut atteindre à partir de r, mais, lorsque certaines valuations sont négatives, cette arborescence n'est pas nécessairement une arborescence de plus courts chemins. Pour tout arc (x, y) de l'arborescence fabriquée par l'algorithme de Dijkstra, on a $\pi(y) - \pi(x) = p(x, y)$.

Corrigé de l'exercice 5

a. Considérons un chemin de longueur maximum de G, graphe sans circuit ; l'origine x de ce chemin ne peut avoir :

- ni prédécesseur sur le chemin, car alors G posséderait un circuit ;
- ni prédécesseur à l'extérieur du chemin, car alors le chemin ne serait pas de longueur maximum.

Le sommet x a donc un demi-degré intérieur nul.

b. On suppose que $G = (X, U)$ est un graphe sans circuit. Considérons l'algorithme suivant :

- Pour i variant de 1 à n, faire
  - choisir un sommet x de G de demi-degré intérieur nul
    (un tel sommet existe d'après la question précédente)
  - faire $num(x) \leftarrow i$
  - faire $G \leftarrow G - \{x\}$
    ($G - \{x\}$ est obtenu à partir de G en retirant x et les arcs adjacents à x)

Au cours de l'algorithme, tous les arcs de G ont été retirés ; lorsqu'un arc est retiré, son origine vient d'être numérotée et son extrémité le sera plus tard ; la numérotation num ainsi construite vérifie donc la propriété cherchée.

Réciproquement, supposons que G admette une numérotation topologique ; considérons un chemin de G : $x_1, x_2, ..., x_p$; on a :

$$num(x_1) < num(x_2) < ... < num(x_p) ;$$

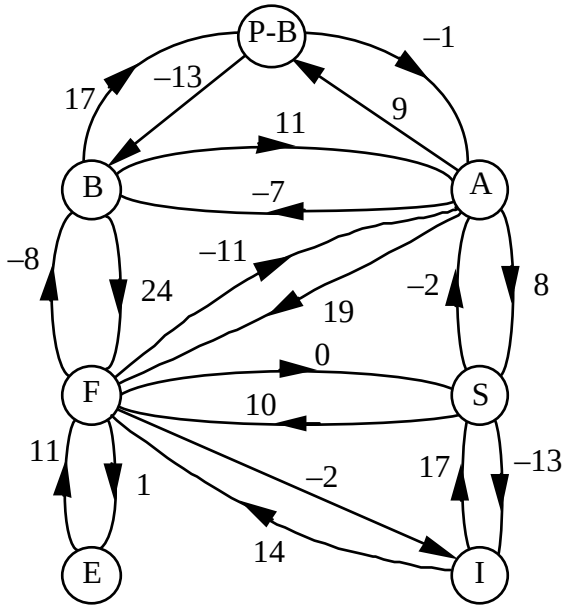
il ne peut y avoir d'arc de x_p vers x_1, car sinon on aurait $num(x_p) < num(x_1)$: G ne possède donc pas de circuit.

La condition nécessaire et suffisante pour qu'un graphe admette une numérotation topologique est par conséquent qu'il soit sans circuit.

Corrigé de l’exercice 6

a. On trace le graphe ayant pour sommets les sept pays, pour arcs les arcs joignant les pays voisins, pour valuation les coûts de passage d’un pays à un autre, compte tenu des aides (comptabilisées comme des coûts négatifs), des taxes et des coûts de transport.

Il faut alors déterminer un chemin entre les Pays-Bas et l’Espagne qui minimise les coûts.



b. On applique l’algorithme de Ford (l’algorithme de Dijkstra ne s’applique pas car il y a des poids négatifs, l’algorithme de Bellman non plus car il y a des circuits). On obtient le tableau suivant, dans lequel nous avons regroupé distances (*dist*) et prédécesseurs (*préd*) :

	P-B		B		A		F		S		I		E	
Étape	*dist*	*préd*	*dist*	*préd*	*dist*	*préd*	*dist*	*préd*	*dist*	*préd*	*dist*	*préd*	*dist*	*préd*
0	0	*	∞	*	∞	*	∞	*	∞	*	∞	*	∞	*
1	0	*	–13	P-B	–1	P-B	∞	*	∞	*	∞	*	∞	*
2	0	*	–13	P-B	–2	B	11	B	7	A	∞	*	∞	*
3	0	*	–13	P-B	–2	B	11	B	6	A	–6	S	12	F
4	0	*	–13	P-B	–2	B	8	I	6	A	–7	S	12	F
5	0	*	–13	P-B	–3	F	7	I	6	A	–7	S	9	F
6	0	*	–13	P-B	–4	F	7	I	5	A	–7	S	8	F
7	0	*	–13	P-B	–4	F	7	I	4	A	–8	S	8	F

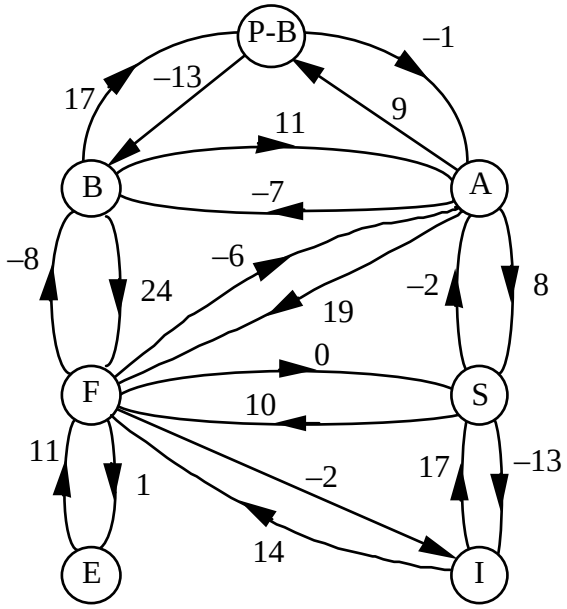
Le numéro d’étape est maintenant égal à l’ordre du graphe et la suite des distances n’est pas stabilisée ; c’est donc que le graphe contient un circuit absorbant. Le coût calculé du transport entre les Pays-Bas et la Suisse a diminué à l’étape 7 : le plus court chemin de longueur au plus 7 (en nombre d’arcs) entre les Pays-Bas et la Suisse est en fait de longueur exactement 7 et n’est donc pas élémentaire ; il contient donc un circuit absorbant. Pour déterminer ce circuit, on « remonte » à partir de la Suisse :

$$\textit{préd}(\text{S}) = \text{A} \ ; \ \textit{préd}(\text{A}) = \text{F} \ ; \ \textit{préd}(\text{F}) = \text{I} \ ; \ \textit{préd}(\text{I}) = \text{S}.$$

Le circuit déterminé est : S → I → F → A → S qui est de coût –2 ; c’est bien un circuit absorbant.

c. L’aide pour l’exportation de la France vers l’Allemagne diminuant, la valuation de l’arc (F, A) change. Le graphe devient le graphe ci-dessous auquel on applique à nouveau

l’algorithme de Ford.



	P-B		B		A		F		S		I		E	
Étape	*dist*	*préd*	*dist*	*préd*	*dist*	*préd*	*dist*	*préd*	*dist*	*préd*	*dist*	*préd*	*dist*	*préd*
0	0	*	∞	*	∞	*	∞	*	∞	*	∞	*	∞	*
1	0	*	–13	P-B	–1	P-B	∞	*	∞	*	∞	*	∞	*
2	0	*	–13	P-B	–2	B	11	B	7	A	∞	*	∞	*
3	0	*	–13	P-B	–2	B	11	B	6	A	–6	S	12	F
4	0	*	–13	P-B	–2	B	8	I	6	A	–7	S	12	F
5	0	*	–13	P-B	–2	B	7	I	6	A	–7	S	9	F
6	0	*	–13	P-B	–2	B	7	I	6	A	–7	S	8	F
7	0	*	–13	P-B	–2	B	7	I	6	A	–7	S	8	F

La suite des distances est stabilisée ; il n’y a pas de circuit absorbant accessible à partir des Pays-Bas ; le chemin le plus économique des Pays-Bas à l’Espagne est déterminé grâce à *préd*(E) = F, *préd*(F) = I, *préd*(I) = S, *préd*(S) = A, *préd*(A) = B, *préd*(B) = P-B. C’est le chemin P-B → B → A → S → I → F → E.

Chapitre VIII

Corrigé de l’exercice 1

L’algorithme proposé s’appuie sur un parcours « marquer-examiner ».

Nous utilisons un tableau, noté *table*, indicé par les sommets, que nous initialisons en attribuant la valeur 0 à chacun des sommets. Lorsque *table*(*x*) est non nul, on dit que *x* est marqué. Nous utilisons aussi un booléen *biparti*, initialisé à vrai. Nous utiliserons une procédure « examiner » définie ci-dessous. Examiner un sommet *x*, c’est :

- Pour tout voisin *y* de *x*, faire
    - si *y* n'est pas marqué, alors
        - poser *table*(*y*) = 3 – *table*(*x*)
        - considérer *y* comme marqué
    - sinon
        - si *table*(*y*) = *table*(*x*), alors
        - poser *biparti* = faux

L'algorithme proposé s'écrit alors :

- Choisir un sommet *a* quelconque
- poser *table*(*a*) = 1
- poser *biparti* = vrai
- tant que *biparti* est à vrai et qu'il existe un sommet *x* marqué non examiné, faire
    - examiner *x*
- si *biparti* est à vrai, conclure que le graphe est biparti
- sinon conclure qu'il ne l'est pas

Prouvons cet algorithme.

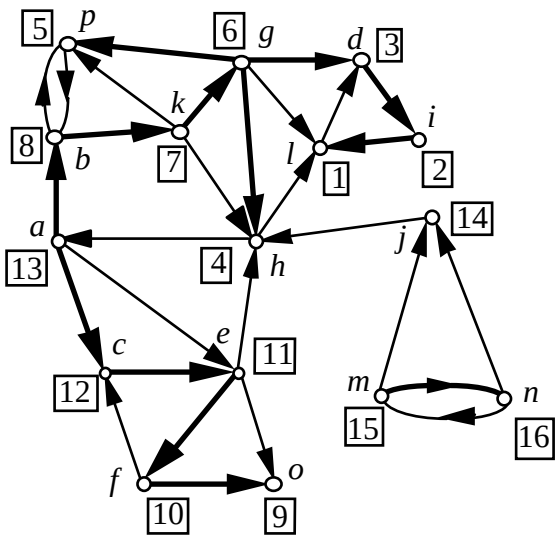
Supposons d'abord que l'algorithme permet de conclure que le graphe est biparti. La variable *biparti* n'a alors jamais été mise à faux. On a effectué un parcours du graphe où tout sommet *x* s'est vu attribuer, dans le tableau *table*, la valeur 1 ou la valeur 2. Supposons que {*x*, *y*} soit une arête et que *x* ait été examiné avant *y* ; lorsqu'on a examiné *x*, on a considéré *y* ; si *table*(*y*) valait 0, on a posé *table*(*y*) = 3 – *table*(*x*) (c'est-à-dire 2 si *table*(*x*) valait 1 et 1 si *table*(*x*) valait 2) ; si *table*(*y*) ne valait pas 0, sa valeur était nécessairement différente de *table*(*x*), sinon *biparti* serait passé à faux. Les valeurs de *table*(*x*) et de *table*(*y*) sont donc nécessairement différentes. La partition des sommets en deux classes selon la valeur de *table* qui leur a été attribuée montre que le graphe est biparti puisque deux sommets adjacents ne peuvent être dans la même classe.

Supposons maintenant que le graphe soit biparti. Notons *Y* et *Z* deux ensembles de sommets tels que $X = Y \cup Z$ et tels que toute arête de *G* ait une extrémité dans *Y* et une dans *Z*. Supposons, sans perte de généralité, que *a* appartienne à *Y*. Montrons alors que tout sommet marqué 1 par l'algorithme appartient à *Y* et tout sommet marqué 2 appartient à *Z*. C'est vrai pour le sommet *a*. Raisonnons par l'absurde et notons *y* le premier sommet marqué qui ne vérifie pas cette règle ; supposons par exemple que *y* appartienne à *Z* ; dire qu'il ne vérifie pas la règle, c'est dire que *table*(*y*) = 1. *y* a été marqué à partir d'un sommet *x* adjacent à *y* ; *x* appartient à *Y* ; *x* vérifiant la règle, *table*(*x*) = 1 ; mais alors, l'algorithme pose *table*(*y*) = 2 ; il y a une contradiction. Le raisonnement est le même si *y* appartient à *Y*. La règle annoncée est vérifiée. Le test « *table*(*x*) = *table*(*y*) » effectué sur deux sommets adjacents ne sera jamais vrai : l'algorithme conclut à la bipartition.

Corrigé de l'exercice 2

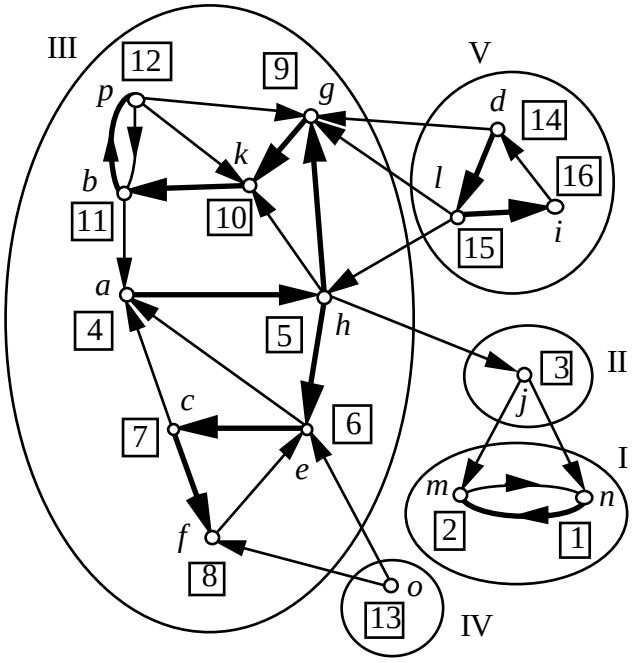
On applique l'algorithme en deux phases qui a été donné dans le chapitre VI.

Lors de la phase 1 de l'algorithme, on démarre le parcours à partir du sommet *a* ; lorsque ce parcours est terminé, certains sommets ne sont pas encore marqués ; on repart du sommet *j* puis à partir du sommet *m*. Lorsqu'un choix est possible, on retient toujours l'ordre alphabétique. Les arborescences de parcours ainsi obtenues sont en gras dans l'illustration ci-dessous ; les numéros indiqués sont les numéros postfixes.



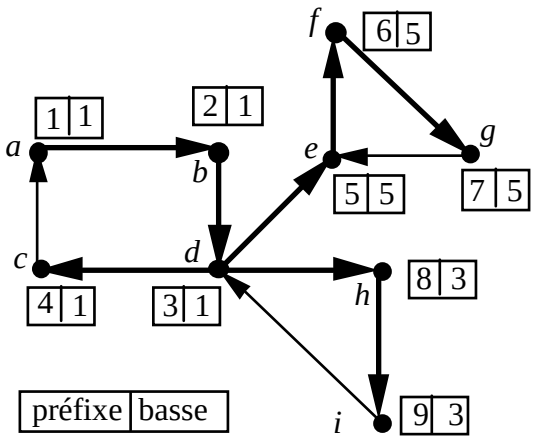
Lors de la phase 2 de l'algorithme, on effectue une suite de parcours dans G^{-} ; on a indiqué ci-dessous la numérotation préfixe obtenue ; on a tracé en gras les arborescences de parcours.

Pour représenter le résultat, on a entouré les ensembles de sommets correspondant à une même composante fortement connexe de G. Les composantes fortement connexes sont numérotées (en chiffres romains) dans l'ordre selon lequel elles ont été constituées.



Corrigé de l'exercice 3

Si on applique l'algorithme à partir de *a*, on obtient le résultat ci-contre. Les arêtes ont toutes été orientées en arcs arborescents ou arcs arrières. Les numéros *préfixe* (à gauche) et *basse* (à droite) ont été indiqués.



La pile des arêtes est successivement :

$$[\varnothing],\ [\{a,b\}],\ \begin{bmatrix}\{b,d\}\\ \{a,b\}\end{bmatrix},\ \begin{bmatrix}\{d,c\}\\ \{b,d\}\\ \{a,b\}\end{bmatrix},\ \begin{bmatrix}\{c,a\}\\ \{d,c\}\\ \{b,d\}\\ \{a,b\}\end{bmatrix},\ \begin{bmatrix}\{d,e\}\\ \{c,a\}\\ \{d,c\}\\ \{b,d\}\\ \{a,b\}\end{bmatrix},\ \begin{bmatrix}\{e,f\}\\ \{d,e\}\\ \{c,a\}\\ \{d,c\}\\ \{b,d\}\\ \{a,b\}\end{bmatrix},\ \begin{bmatrix}\{f,g\}\\ \{e,f\}\\ \{d,e\}\\ \{c,a\}\\ \{d,c\}\\ \{b,d\}\\ \{a,b\}\end{bmatrix},\ \begin{bmatrix}\{g,e\}\\ \{f,g\}\\ \{e,f\}\\ \{d,e\}\\ \{c,a\}\\ \{d,c\}\\ \{b,d\}\\ \{a,b\}\end{bmatrix}.$$

Au moment où on recule de *f* en *e*, on a : *basse*(*f*) ≥ *préfixe*(*e*) ; on dépile donc les arêtes {*g*, *e*}, {*f*, *g*}, {*e*, *f*} qui constituent les arêtes de la composante 2-connexe induite par les sommets *e*, *f* et *g* ; la pile est à ce moment-là celle représentée à droite. On doit alors reculer de *e* en *d* ; comme *basse*(*e*) ≥ *préfixe*(*d*), on dépile l'arête {*d*, *e*}, qui correspond à la composante 2-connexe (réduite à une arête) induite par les sommets *d* et *e*. La pile est alors successivement :

$$\begin{bmatrix}\{d,e\}\\ \{c,a\}\\ \{d,c\}\\ \{b,d\}\\ \{a,b\}\end{bmatrix}$$

$$\begin{bmatrix}\{c,a\}\\ \{d,c\}\\ \{b,d\}\\ \{a,b\}\end{bmatrix},\ \begin{bmatrix}\{d,h\}\\ \{c,a\}\\ \{d,c\}\\ \{b,d\}\\ \{a,b\}\end{bmatrix},\ \begin{bmatrix}\{h,i\}\\ \{d,h\}\\ \{c,a\}\\ \{d,c\}\\ \{b,d\}\\ \{a,b\}\end{bmatrix},\ \begin{bmatrix}\{i,d\}\\ \{h,i\}\\ \{d,h\}\\ \{c,a\}\\ \{d,c\}\\ \{b,d\}\\ \{a,b\}\end{bmatrix}.$$

Au moment où on recule de *h* en *d*, on a : *basse*(*h*) ≥ *préfixe*(*d*) ; on dépile donc les arêtes {*i*, *d*}, {*h*, *i*}, {*d*, *h*} qui constituent les arêtes de la composante 2-connexe induite par les sommets *d*, *i* et *h* ; la pile devient :

$$\begin{matrix}\lceil \{c,\ a\} \rceil \\ \mid \{d,\ c\} \mid \\ \mid \{b,\ d\} \mid \\ \lfloor \{a,\ b\} \rfloor\end{matrix}.$$

Enfin, au moment où l'on recule de b en a, on a $basse(b) \geq préfixe(a)$, on dépile donc les arêtes $\{c, a\}$, $\{d, c\}$, $\{b, d\}$, $\{a, b\}$; ces arêtes sont les arêtes de la composante 2-connexe induite par les sommets a, b, c et d.

Corrigé de l'exercice 4

a. Il est clair que s'il existe un arc arrière (y, x), alors il existe un circuit constitué du chemin de l'arborescence allant de x à y et de l'arc (y, x).

Réciproquement, supposons qu'il existe dans G un circuit. Notons $x_1, x_2, \ldots, x_p$ ce circuit. On peut supposer sans perte de généralité que le premier sommet du circuit marqué pendant le parcours est le sommet x_1. Montrons alors par récurrence sur i que tous les sommets du circuit sont marqués pendant $DFS(x_1)$ et donc sont descendants de x_1. Pour $i = 1$, ceci est trivialement vérifié. Supposons que ce le soit pour $1 \leq i < p$ et montrons ce même résultat pour $i + 1$; si x_{i+1} est marqué avant x_i, sachant qu'il est marqué après x_1, c'est qu'il est lui aussi marqué pendant $DFS(x_1)$; si x_{i+1} est marqué après x_i, comme il est successeur de x_i, c'est qu'il est marqué durant $DFS(x_i)$ qui se déroule à l'intérieur de $DFS(x_1)$. Dans les deux cas, x_{i+1} est marqué pendant $DFS(x_1)$. Or, x_p est descendant de x_1 dans l'arborescence : l'arc (x_p, x_1) est un arc arrière.

b. Le raisonnement est absolument similaire à celui développé dans la partie a.

Chapitre IX

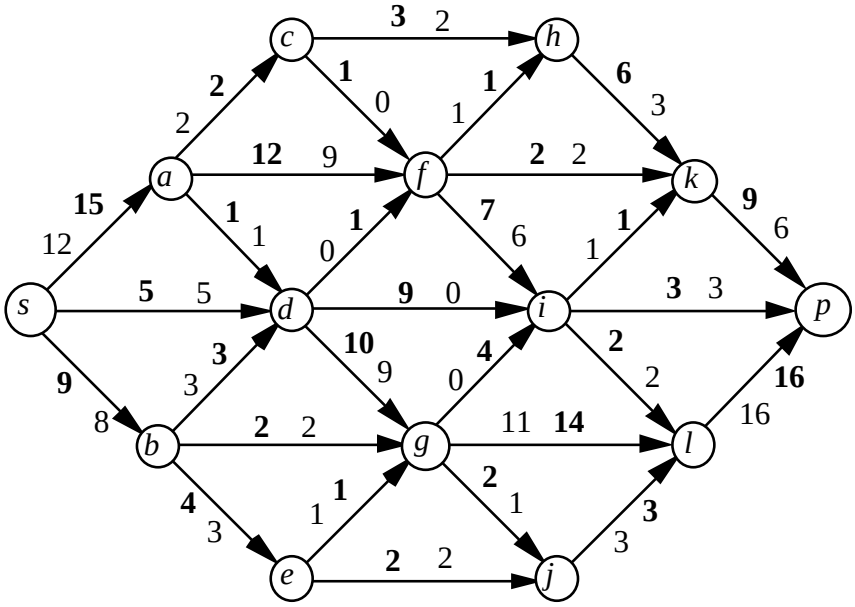
Corrigé de l'exercice

On fait tourner l'algorithme de Ford et Fulkerson. Les étapes successives sont représentées dans le tableau ci-contre.

sommets examinés	sommets marqués
	(Δ, ∞)
s	$a(s, +7)$ et $b(s, +1)$
a	$f(a, +7)$
b	$e(b, +1)$
f	$i(f, +5)$
e	
i	$d(i, -4)$
d	$g(d, +4)$
g	$l(g, +4)$ et $j(g, +1)$
l	$p(l, +4)$

On remarquera que l'examen de *i* donne à *d* la marque (*i*, –4) et non (*i*, –5), car on considère ici le flux de l'arc (*i*, *d*) et non la différence entre capacité et flux. D'autre part, l'examen de *d* donne à *g* la marque (*d*, +4) et non *g*(*d*, +5), car la marque de *g* est aussi limitée par la valeur absolue de celle de *d*.

On a trouvé une chaîne augmentante de 4 : la chaîne *s*, *a*, *f*, *i*, *d*, *g*, *l*, *p*. Les nouveaux flux sont notés ci-dessous.



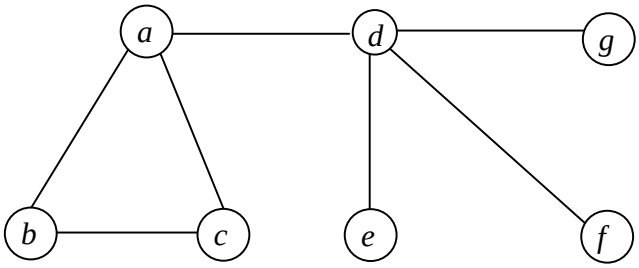
Chapitre XI

Corrigé de l'exercice 1

Tout cycle de longueur impaire vérifie $\Delta(G) = 2$ et $\gamma(G) = 3$.

Corrigé de l'exercice 2

1. Considérons le graphe ci-dessous :



L'algorithme recherchant une clique maximale sélectionne d'abord le sommet *d*. Le voisinage de ce sommet étant le stable composé des sommets *a*, *e*, *f* et *g*, l'algorithme sélectionne alors un de ces quatre sommets, par exemple le sommet *a*, puis se termine : on obtient ainsi la clique constituée des sommets *a* et *d*, de cardinal 2.

Pourtant le graphe possède une clique de cardinal 3 : le triangle composé des sommets *a*, *b* et *c* : $\omega(G) = 3$. L'algorithme ne donne pas cette valeur.

2. Tout cycle de longueur impaire vérifie $\omega(G) = 2$ et $\gamma(G) = 3$.

Chapitre XII

Corrigé de l'exercice 1

Le problème d'optimisation de la clique maximum consiste à déterminer, dans un graphe *G* donné, une clique dont toutes les arêtes soient des arêtes de *G* et qui contienne le plus de sommets possible. Le problème de décision associé peut être décrit de la façon suivante :

Nom : *CLIQUE*
Données : étant donné un entier *k*, étant donné un graphe non orienté $G = (X, E)$,
Question : existe-t-il dans *G* une clique de taille $\geq k$, c'est-à-dire $Y \subseteq X$ tel que le graphe $H = (Y, (Y \times Y) \cap E)$ soit une clique et que $|Y| \geq k$?

Montrons que *CLIQUE* est NP-complet en lui réduisant polynomialement *SAT*.

Auparavant, constatons que *CLIQUE* est dans NP : si on veut vérifier qu'un graphe *H* convient, il faut compter les sommets de *H* et comparer ce nombre à *k*, puis il faut vérifier que toutes les arêtes de *H* sont aussi arêtes de *G*. Comme *H* a au plus autant de sommets que *G*, il est facile d'effectuer ces vérifications en un nombre d'opérations polynomial par rapport à la taille de l'instance.

Considérons une instance quelconque de *SAT*, définie par un ensemble *U* de variables booléennes et un ensemble *C* de *m* clauses $C_1, C_2, ..., C_m$ définies sur les littérales dérivées de *U* : pour $1 \leq i \leq m$, $C_i = \{z_{i,1}, z_{i,2}, ..., z_{i,r_i}\}$ avec $z_{i,j} \in U$ ou $\overline{z_{i,j}} \in U$ pour $1 \leq j \leq r_i$; $z_{i,j}$ est donc la j^{e} littérale de la i^{e} clause.

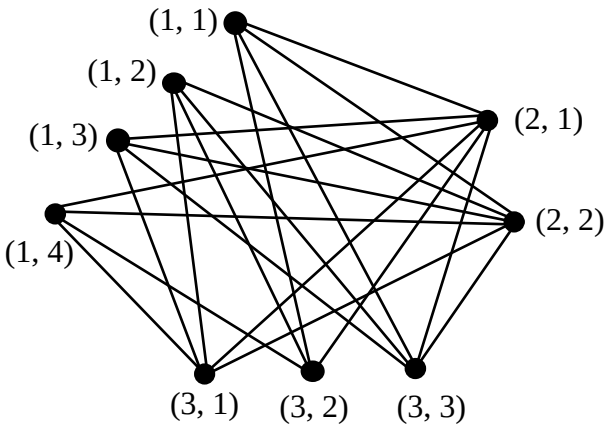
On construit un graphe *G* dont les sommets sont des couples (i, j), *i* faisant référence au numéro de la clause, et *j* au numéro de la littérale : la clause $C_i = \{z_{i,1}, z_{i,2}, ..., z_{i,r_i}\}$ donnera les r_i sommets $(i, 1), (i, 2), \ldots, (i, r_i)$. Les sommets (i, j) et (r, s) sont adjacents dans *G* si et seulement si $i \neq r$ (les arêtes ne relient que des sommets provenant de clauses différentes) et $z_{i,j} \neq \overline{z_{r,s}}$ (de cette façon, si on sélectionne (i, j) et (r, s) pour constituer *Y*, on pourra attribuer la valeur « vrai » à $z_{i,j}$ et $z_{r,s}$ sans créer d'incompatibilité).

La figure ci-contre montre le graphe G que l'on obtient si l'instance de *SAT* est donnée par les 3 clauses :

$C_1 = \{u_1,\ \overline{u_2},\ u_3,\ u_4\}$,

$C_2 = \{u_2,\ u_3\}$,

$C_3 = \{\overline{u_1},\ \overline{u_3},\ \overline{u_4}\}$.



On complète l'instance de *CLIQUE* en posant $k = m$. Compte tenu de la forme de G (absence d'arêtes entre des sommets provenant d'une même clause), toute clique de G contient au plus m sommets. Le choix de k fait donc que la réponse à l'instance ainsi créée de *CLIQUE* est « oui » si et seulement si on peut exhiber une clique de taille exactement m, ce qui signifie que l'on cherche à sélectionner exactement un sommet dans chaque paquet de sommets traduisant une clause.

Cette transformation conserve bien la réponse. Imaginons que la réponse à l'instance de *SAT* soit « oui » : il existe une fonction f à valeur dans {« vrai », « faux »} telle que chaque clause C_i contient au moins une littérale dont l'image par f soit « vrai ». Soit z_{i,j_i} une telle littérale pour tout i compris entre 1 et m. Alors l'ensemble $Y = \{(i, j_i)$ pour $1 \leq i \leq m\}$ définit une clique de G de taille m : les sommets (i, j_i) et (r, s_r) avec $i \neq r$ sont bien reliés par une arête puisque, sinon, il faudrait avoir $z_{i,j_i} = \overline{z_{r,s_r}}$, ce qui n'est pas compatible avec $f\left(z_{i,j_i}\right) = f\left(z_{r,s_r}\right)$ = « vrai ».

Réciproquement, soit Y un ensemble de sommets de taille m définissant une clique de G. Soit f une fonction à valeur dans {« vrai », « faux »} qui attribue la valeur « vrai » aux littérales associées aux sommets de Y (ceci est bien possible, d'après une remarque faite plus haut). Comme Y possède m sommets et que chaque sommet de Y provient d'une clause qui lui est propre, chaque clause possède une littérale dont la valeur par f est « vrai » : toutes les clauses sont donc satisfaites par f et la réponse à l'instance de *SAT* est « oui ».

Enfin la transformation est polynomiale. En effet, si U possède n variables booléennes, on dispose de $2n$ littérales, que l'on peut supposer numérotées de 1 à $2n$; coder une clause C_i à r_i littérales peut se faire en énumérant les uns à la suite des autres les numéros des r_i littérales constituant C_i : il faut alors plus de r_i bits pour coder C_i ; la taille de l'instance de *SAT* est donc au moins $\sum_{i=1}^{m} r_i$. Or, G possède $\sum_{i=1}^{m} r_i$ sommets ; la description de G par exemple au moyen de sa matrice d'adjacence nécessite par conséquent $\left(\sum_{i=1}^{m} r_i\right)^2$ bits ; par ailleurs, il faut environ $\log_2 m \leq \sum_{i=1}^{m} r_i$ (car $m \leq \sum_{i=1}^{m} r_i$) bits pour coder k. La construction de l'instance de *CLIQUE* peut donc se faire en un nombre d'opérations borné par un polynôme en $\sum_{i=1}^{m} r_i$, et la transformation est polynomiale.

On a réussi à réduire *SAT* à *CLIQUE* à l'aide d'une transformation polynomiale (on aurait tout aussi bien pu partir de 3-*SAT* et utiliser la même transformation). La NP-complétude de *SAT* entraîne donc celle de *CLIQUE*.

Corrigé de l'exercice 2

Le problème d'optimisation du transversal minimum consiste à déterminer, dans un graphe G donné, un sous-ensemble de sommets de cardinal minimum tel que toute arête de G possède au moins une de ses extrémités dans ce sous-ensemble. Le problème de décision associé peut être décrit de la façon suivante :

Nom : *TRANSVERSAL*
Données : étant donné un entier k, étant donné un graphe non orienté $G = (X, E)$,
Question : existe-t-il dans G un transversal de taille $\leq k$, c'est-à-dire $Y \subseteq X$ avec $|Y| \leq k$ et tel que $\forall \{x, y\} \in E, x \in Y$ ou $y \in Y$?

Montrons que *TRANSVERSAL* est NP-complet en lui réduisant polynomialement *CLIQUE*.

Auparavant, constatons que *TRANSVERSAL* est dans NP : si on veut vérifier qu'un sous-ensemble Y convient, il faut compter les éléments de Y et comparer ce nombre à k, puis, pour chaque arête de G, vérifier qu'au moins une de ses extrémités se trouve dans Y. Il est facile d'effectuer ces vérifications en un nombre d'opérations polynomial par rapport à la taille de l'instance.

Transformons une instance quelconque de *CLIQUE* en une instance de *TRANSVERSAL* de la façon suivante. Soit (H, r) une instance de *CLIQUE*, avec $H = (X, F)$; l'instance (G, k) associée de *TRANSVERSAL* sera définie par $G = \overline{H}$ et $k = n - r$, où $\overline{H}$ désigne le *complémentaire* de G, c'est-à-dire le graphe ayant même ensemble de sommets que G et tel que deux sommets distincts sont adjacents dans $\overline{H}$ si et seulement si ce n'est pas le cas dans H (plus formellement, si on pose $G = (X, E)$, alors

$$E = X \times X - (F \cup \{(x, x) \text{ pour } x \in X\}).$$

Montrons que si Y définit une clique de H de taille $\geq r$, alors $X - Y$ est un transversal de $G = \overline{H}$ de taille $\leq k$ et réciproquement. En effet, soit Y un sous-ensemble de cardinal $\geq r$ définissant une clique de H. Alors Y est un stable (c'est-à-dire un ensemble de sommets deux à deux non adjacents) de $\overline{H}$ de cardinal $\geq r$. Par conséquent, toute arête de G possède au moins une extrémité dans $X - Y$, ce qui montre que $X - Y$ est un transversal de G de taille inférieure ou égale à $n - r = k$. Réciproquement, si $X - Y$ est un transversal de G de taille $\leq k$, alors il n'y a pas d'arête dans G dont les deux extrémités soient dans Y, ce qui entraîne que Y définit un stable de taille supérieure ou égale à $n - k$ dans $G = \overline{H}$, et donc une clique de taille supérieure ou égale à $n - k = r$ dans H.

Il s'ensuit que la transformation proposée plus haut conserve la réponse « oui » ou « non ». Comme de plus elle est clairement polynomiale, que *TRANSVERSAL* est dans NP et que *CLIQUE* est NP-complet, *TRANSVERSAL* est lui aussi NP-complet.

On peut enfin remarquer que les mêmes arguments permettraient de démontrer que la recherche d'un stable de cardinal maximum est un problème NP-difficile, comme les deux problèmes d'optimisation dont *CLIQUE* et *TRANSVERSAL* sont les problèmes de décision.

12. Chapitre XIII

13.1. Corrigé de l'exercice 1

Le coefficient de la variable x_1 étant, par hypothèse, positif, la variable x_1 est entrante et on l'échange donc avec l'unique variable en base, qui correspond à l'unique contrainte. On avait

$$x_{n+1} = V - \sum_{j=1}^{n} v_j . x_j$$

et, après l'échange, on obtient :

$$x_1 = \frac{1}{v_1}\left(V - \sum_{j=2}^{n} v_j . x_j - x_{n+1} \right)$$

En reportant cette valeur dans la fonction objectif, il vient :

ou encore $$z = \frac{u_1 . V}{v_1} + \sum_{j=2}^{n}\left(u_j - \frac{u_1 . v_j}{v_1} \right). x_j - \frac{u_1}{v_1} x_{n+1}$$

Compte tenu de la numérotation adoptée, les coefficients de toutes les variables qui interviennent dans l'écriture de z sont négatifs ou nuls. On a donc trouvé la valeur maximum de z en une itération, et celle-ci est égale à $u_1.V/v_1$.

13.2. Corrigé de l'exercice 2

DETERMINATION D'UNE BORNE B

On applique une heuristique de prolongement d'une chaîne en choisissant à chaque étape l'arête la moins lourde permettant ce prolongement (bien sûr en ne créant un cycle qu'à la fin). En partant de x, on peut par exemple choisir successivement $\{x, t\}$, $\{t, y\}$, $\{y, u\}$, $\{u, v\}$, choix que l'on complète nécessairement par l'adjonction des arêtes $\{x, z\}$ et $\{z, v\}$, ce qui donne le cycle hamiltonien $x\ t\ y\ u\ v\ z\ x$, de poids 17. Nous commençons la recherche avec cette borne : $B = 17$.

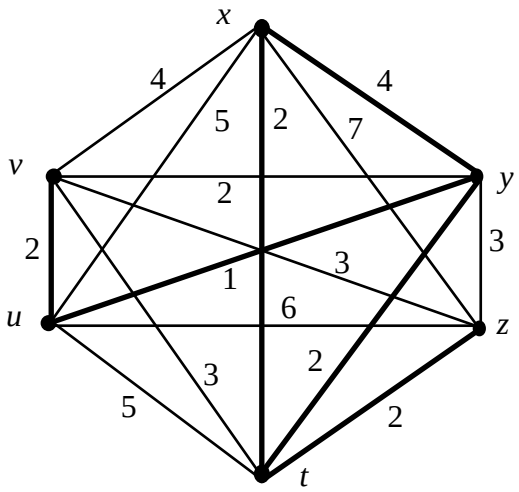
STRATEGIE DE DEVELOPPEMENT

Bien qu'elle soit plus consommatrice de place mémoire et qu'elle pose parfois des problèmes pour gérer l'arborescence et déterminer le sommet auquel appliquer le principe de séparation, nous adopterons ici une stratégie de type « meilleur d'abord », assez facile à mettre en œuvre à la main. Par conséquent, quand on effectuera une séparation, tous les sommets obtenus seront évalués, et on repartira du sommet non encore séparé de moindre évaluation.

ÉVALUATION DE LA RACINE DE L'ARBORESCENCE

Prenons x pour jouer le rôle du x_0 du texte ; les deux arêtes de poids minimum incidentes à x sont les arêtes $\{x, t\}$ et par exemple $\{x, y\}$. Un arbre de poids minimum de $K_6 - x$ nous sera fourni par l'algorithme de Kruskal ou par celui de Prim. Avec les deux arêtes incidentes à x,

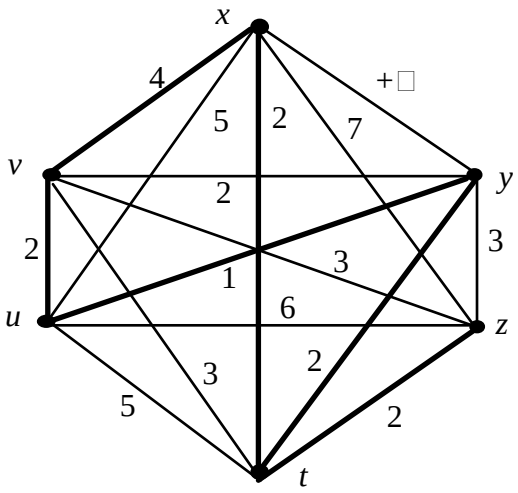
on obtient par exemple le graphe H en gras, de poids 13, poids qui donne l'évaluation de la racine.



Ce graphe n'est pas un cycle hamiltonien parce que certains sommets ont un degré strictement supérieur à 2. C'est le cas par exemple du sommet y. Nous allons utiliser le fait que dans H ce sommet est de degré 3 pour effectuer les trois branchements décrits ci-dessous.

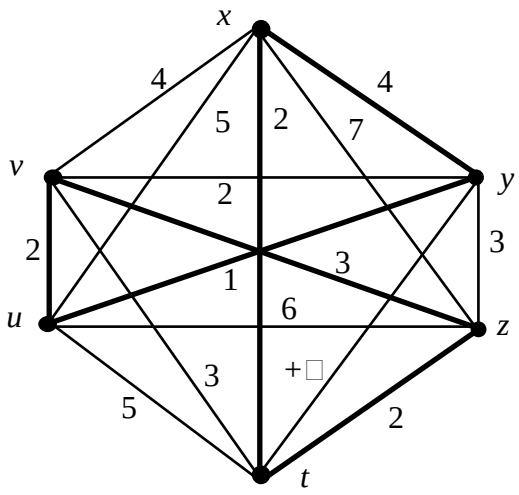
1ER BRANCHEMENT

L'arête $\{x, y\}$ est interdite, c'est-à-dire que son poids devient infini, ce qui ne modifie pas l'arbre couvrant de $K_6 - x$. Les deux arêtes de poids minimum incidentes à x sont maintenant $\{x, t\}$ et $\{x, v\}$. L'évaluation de ce sommet est toujours 13, mais elle n'est pas exacte, le graphe correspondant (ci-contre) n'étant pas un cycle hamiltonien.



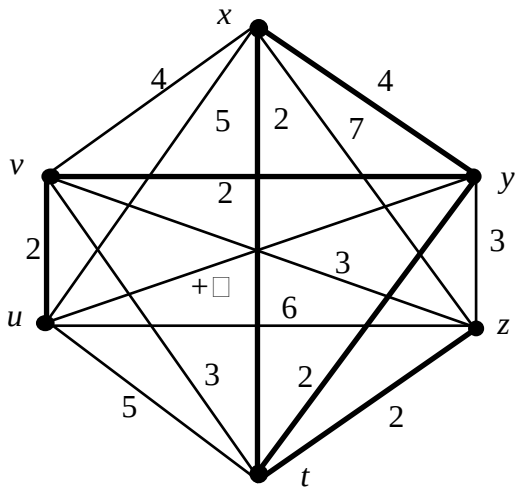
2E BRANCHEMENT

L'arête $\{y, t\}$ est interdite (son poids devient infini), ce qui modifie l'arbre couvrant de $K_6 - x$. Parmi les nouveaux arbres de poids minimum, considérons celui qui, ainsi que les deux arêtes de poids minimum incidentes à x sont en gras dans le graphe ci-dessous. On obtient un cycle hamiltonien de poids 14. On peut donc mettre à jour la borne B qui passe de 17 à 14, et d'autre part ne pas développer ultérieurement le sommet auquel on vient d'aboutir dans l'arborescence, son évaluation étant égale à la nouvelle valeur de B.

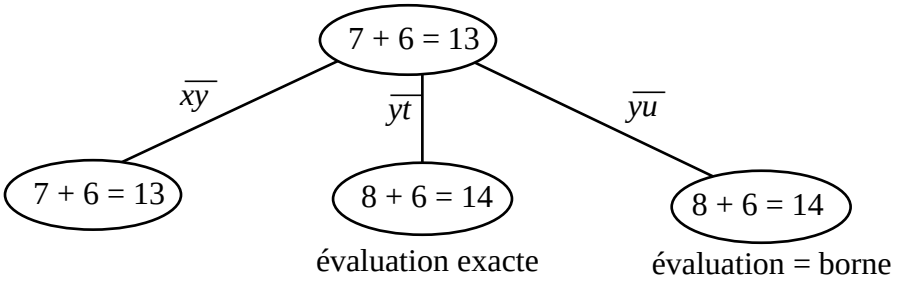


3E BRANCHEMENT

L'arête $\{y, u\}$ est interdite (son poids devient infini), ce qui modifie l'arbre couvrant de $K_6 - x$. Les nouveaux arbres de poids minimum ont un poids égal à 8 (voir graphe ci-contre). Avec les deux arêtes de poids minimum incidentes à x, on obtient ainsi une évaluation de 14, donc égale à B. Par conséquent, il est inutile de développer l'arborescence à partir de ce sommet : il ne conduira pas à un cycle hamiltonien plus intéressant que celui qu'on connaît déjà.



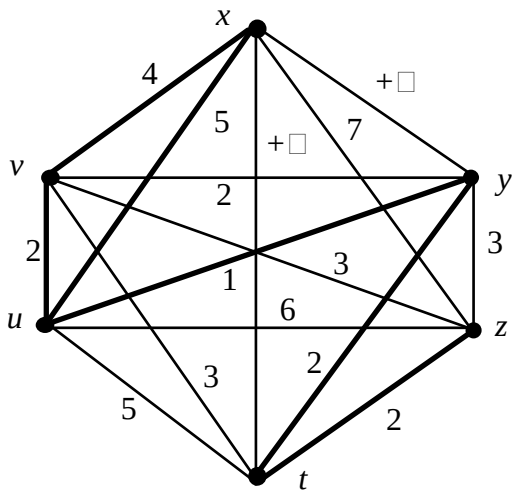
Les opérations que l'on vient d'effectuer peuvent être résumées par le dessin suivant, qui donne l'état actuel de l'arborescence. Les arêtes $\{\alpha, \beta\}$ interdites y sont représentées par $\overline{\alpha\beta}$.



Il ne reste plus à développer que le sommet situé à gauche dans le dessin précédent. Dans le graphe correspondant à cette évaluation (voir plus haut le graphe obtenu par le premier branchement), le sommet t est de degré 3. On effectue donc trois nouveaux branchements à partir de ce sommet de l'arborescence, en interdisant successivement, en plus de l'arête $\{x, y\}$, les arêtes $\{t, x\}$, $\{t, y\}$ puis $\{t, z\}$.

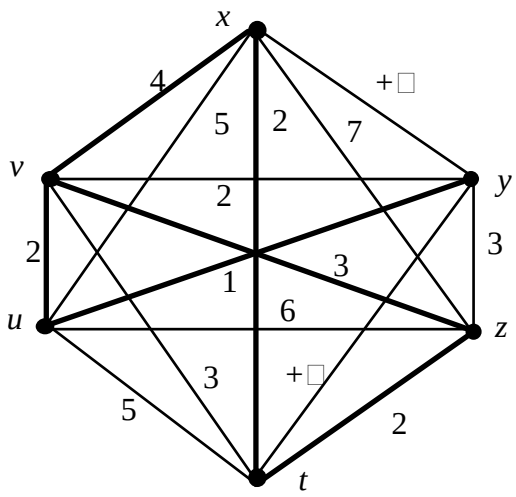
4E BRANCHEMENT

L'arête $\{t, x\}$ est interdite (son poids devient infini). Les deux arêtes de poids minimum incidentes à x sont maintenant $\{x, v\}$ et $\{x, u\}$. Le poids du graphe en gras dans le dessin ci-dessous, égal à 16 (= 7 + 9), donne l'évaluation du sommet de l'arborescence auquel on aboutit. Cette évaluation étant supérieure à B, on ne développera pas ce sommet.



5E BRANCHEMENT

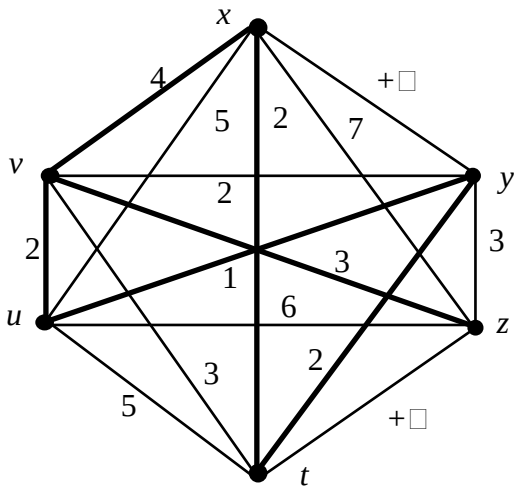
L'arête $\{t, y\}$ est interdite (son poids devient infini). Un arbre de poids minimum de $K_6 - x$ avec l'infini pour poids de $\{t, y\}$ a déjà été calculé pour le 2e branchement, de poids 8. Comme de plus les deux arêtes de poids minimum incidentes à x sont $\{x, t\}$ et $\{x, v\}$, on obtient 8 + 6 = 14 comme évaluation de ce nouveau sommet de l'arborescence. Cette évaluation étant égale à B, on ne développera pas ce sommet.



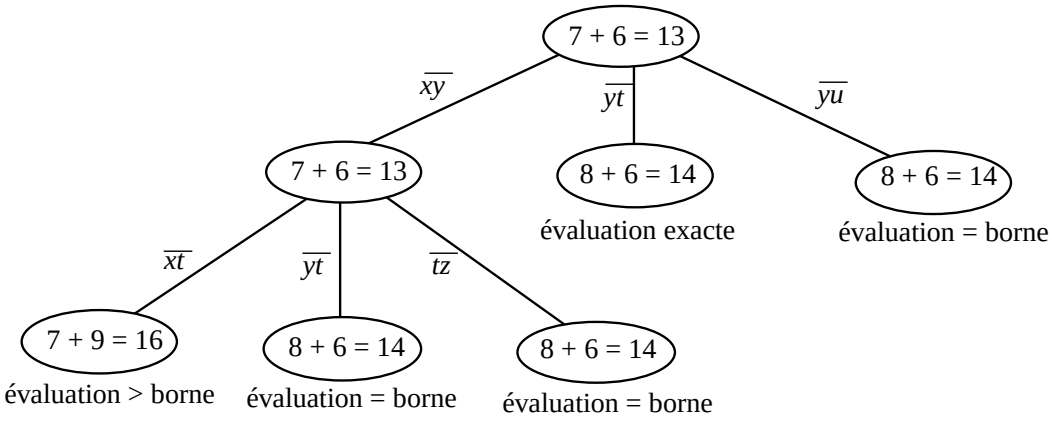
On pouvait d'ailleurs prévoir qu'il ne serait pas intéressant de développer cette branche, puisque, par rapport au 2e branchement, on a interdit une arête supplémentaire ; on ne pouvait donc pas obtenir mieux que par le 2e branchement.

6E BRANCHEMENT

L'arête $\{t, z\}$ est interdite (son poids devient infini). L'évaluation de ce nouveau sommet de l'arborescence est donnée par le poids du graphe en gras dans le dessin ci-contre, et vaut donc 14, ce qui n'est pas mieux que la borne : on ne développera pas ce sommet.



Il ne reste plus de sommet à développer : on a terminé. L'arborescence totale est donc réduite aux sommets du dessin suivant :



Le poids minimum d'un cycle hamiltonien vaut 14 et le cycle hamiltonien obtenu au 2e branchement est une solution optimale du problème.

Si on voulait déterminer tous les cycles hamiltoniens de poids minimum, il faudrait encore développer tous les sommets dont l'évaluation vaut 14, puisqu'ils sont susceptibles de contenir de telles solutions. Plus généralement, il conviendrait de modifier la description des méthodes par séparation et évaluation de façon à n'éliminer que les sommets de l'arborescence qui ne contiennent aucune solution réalisable (les contraintes du problème ne sont pas toutes respectées) ou dont l'évaluation est strictement moins bonne que la borne (c'est-à-dire strictement supérieure dans le cas d'une minimisation et strictement inférieure dans le cas d'une maximisation) ; les sommets d'évaluation égale à la borne seraient donc développés, y compris lorsque l'évaluation est exacte.

Bibliographie

A. Aho, J. Hopcroft, J. Ullman, *Structures de données et algorithmes*, InterEditions, Paris, 1987.

A. Aho, J. Ullman, *Concepts fondamentaux de l'informatique*, Dunod, Paris, 1993.

D. Beauquier, J. Berstel, P. Chrétienne, *Éléments d'algorithmique*, Masson, Paris, 1992.

T. Cormen, C. Leiserson, R. Rivest, *Introduction à l'algorithmique*, Dunod, Paris, 1994.

M.-C. Gaudel, M. Soria, C. Froidevaux, *Types de données et algorithmes*, McGraw Hill, Paris, 1990.

D.E. Knuth, *The art of computer programming*, Reading, Massachusetts, Addison-Wesley, 1968.

N. Wirth, *Algorithmes et structures de données*, Eyrolles, Paris, 1987.

Index