MATROÏDES ET ALGORITHMES GLOUTONS : UNE INTRODUCTION

par

Pierre Béjian

GLOUTON, -ONNE adj. et n. XIe siècle, *gloton*; XVIIe siècle, comme nom d'animal. Issu du latin impérial *glut(t)onem*, « glouton », dérivé de *glut(t)us*, « gosier ».
1. Adj. Qui mange avec avidité et excès, voracement. *Un enfant glouton.* Par méton. *Un appétit glouton. Une faim gloutonne.* Subst. *Cet homme est un glouton.* Fig. Avide. *Une jeunesse gloutonne de plaisirs. Une curiosité gloutonne.* **2.** N. m. Mammifère carnivore de la famille des Mustélidés. *Les gloutons vivent dans les régions arctiques.*

Introduction

La résolution d'un problème algorithmique peut parfois se faire à l'aide de techniques générales (« paradigmes ») qui ont pour avantage d'être applicables à un grand nombre de situations. Parmi ces méthodes on peut citer par exemple le fameux principe *diviser pour régner* ou encore la *programmation dynamique*. Les *algorithmes gloutons*, que l'on rencontre principalement pour résoudre des problèmes d'optimisation, constituent l'une de ces techniques générales de résolution.
Nous allons tout d'abord donner une définition intuitive de cette méthode, accompagnée de divers exemples. Nous allons ensuite formaliser cette approche à l'aide de la notion de matroïde, la dernière partie étant consacrée à une généralisation possible de cette notion.

1. Le principe glouton

1.1. Définition. — Lors de la résolution d'un problème d'optimisation, la construction d'une solution se fait souvent de manière séquentielle, l'algorithme faisant à chaque étape un certain nombre de choix. Le *principe glouton* consiste à faire le choix qui semble le meilleur sur le moment (choix local), sans se préoccuper des conséquences dans l'avenir, et sans revenir en arrière. Un algorithme glouton est donc un algorithme qui ne se remet jamais en question et qui se dirige le plus rapidement possible vers une solution. Aveuglé par son appétit démesuré, le glouton n'est pas sûr d'arriver à une solution optimale, mais il fournit un résultat rapidement. Même si la solution n'est pas optimale, il n'est pas rare que l'on s'en contente (par exemple pour un problème NP-difficile). On rentre alors dans le monde des algorithmes d'approximation.
Pour que la méthode gloutonne ait une chance de fonctionner, il faut que le choix local aboutisse à un problème similaire plus petit. La méthode gloutonne ressemble ainsi à la programmation dynamique. Mais la différence essentielle est que l'on fait d'abord un choix local et on résout *ensuite* un problème plus petit (progression descendante). En programmation dynamique, au contraire, on commence par résoudre des sous-problèmes, dont on combine ensuite les résultats (progression ascendante).

1.2. Premier exemple : location d'un camion. — Une agence de location de véhicules dispose d'un seul camion pour lequel des clients ont formulé une demande caractérisée par la date de début et la date de fin. Notons E l'ensemble de ces demandes, et pour tout $e \in E$ $d(e)$ (resp. $f(e)$) la date de début (resp. de fin). La politique commerciale de l'entreprise consiste à satisfaire le plus grand nombre de clients. Le problème est donc de trouver un sous-ensemble de E, constitué de demandes compatibles et de cardinal maximum. Formellement, la compatibilité des demandes e_1 et e_2 signifie que

$$]d(e_1), f(e_1)[\cap]d(e_2), f(e_2)[= \varnothing.$$

Pour résoudre ce problème on propose l'algorithme suivant :

LocationCamion(E)
  Trier les élements de E par date de fin croissante $f(e_1) \leq f(e_2) \leq \cdots \leq f(e_n)$
  $s_1 \leftarrow e_1$ // *on choisit la demande qui termine le plus tôt*
  $k \leftarrow 1$ // *k désigne le nombre de clients actuellement satisfaits*
  pour $i = 2$ à n **faire**
    si $d(e_i) \geq f(s_k)$ **alors**
      $s_{k+1} \leftarrow c_i$
      $k \leftarrow k + 1$
    fin si
  fin pour
  retourner $\{s_1, ..., s_k\}$

Cet algorithme est glouton car il construit l'ensemble $S = \{s_1, ..., s_k\}$ de manière séquentielle et qu'il fait à chaque étape le choix le moins coûteux (c'est à dire la demande compatible qui termine le plus tôt).
Notons que l'ensemble S est bien constitué de demandes compatibles. Il reste donc à prouver qu'il est de cardinal maximum.

Théorème 1.1. — *L'ensemble S produit par l'algorithme* LocationCamion *est bien une solution optimale, c'est à dire de cardinal maximum.*

Démonstration. — Dans toute la preuve, les ensembles seront ordonnés par date de fin croissante. Si $X = \{x_1, ..., x_r\}$ est un ensemble ainsi ordonné, les demandes $x_1, ..., x_r$ sont compatibles si et seulement si

$$d(x_1) < f(x_1) \leq d(x_2) < f(x_2) \leq \cdots \leq d(x_r) < f(x_r).$$

a) Nous allons tout d'abord montrer qu'il existe une solution optimale $T = \{t_1, ..., t_l\}$ telle que $S \subset T$. Pour se faire montrons par récurrence que pour tout $j \in \{1..k\}$ il existe une solution optimale dont les j premiers éléments coïncident avec $s_1, ..., s_j$.
– *Pour $j = 1$:* soit $t_1, ..., t_l$ est une solution optimale. Puisque les t_i sont des demandes compatibles ordonnées par date de fin croissante, on a

$$d(t_1) < f(t_1) \leq d(t_2) < f(t_2) \leq \cdots \leq d(t_r) < f(t_r).$$

Par ailleurs s_1 étant la demande qui termine le plus tôt, on a $f(s_1) \leq f(t_1)$ et on en déduit donc

$$d(s_1) < f(s_1) \leq d(t_2) < f(t_2) \leq \cdots \leq d(t_r) < f(t_r).$$

L'ensemble $\{s_1, t_2, ..., t_l\}$ est ainsi une solution optimale dont le premier élément est commun avec S.
– $\mathscr{P}(j) \Rightarrow \mathscr{P}(j+1)$: soit $t_1, ..., t_l$ est une solution telle que $s_1 = t_1, ..., s_j = t_j$. Par construction s_{j+1} est une demande appartenant à $E \setminus \{s_1, ..., s_j\}$ compatible avec $s_1, ..., s_j$ dont la date de fin est minimale. Puisque $s_1 = t_1, ..., sj = t_j$, t_{j+1} est aussi une demande appartenant à $E \setminus \{s_1, ..., s_j\}$ compatible avec $s_1, ..., s_j$, on a $f(s_{j+1}) \leq f(t_{j+1})$. On en déduit alors que s_{j+1} est compatible avec $t_{j+2}, ..., t_l$. Ainsi $\{s_1, ..., s_{j+1}, t_{j+2}, ..., t_l\}$ est une solution dont les $j + 1$ premiers éléments coïncident avec ceux de S.
Il existe donc une solution optimale $T = \{t_1, ..., t_l\}$ telle que $s_1 = t_1, ..., s_k = t_k$, ce qui termine le premier point de la preuve.

b) Montrons pour finir que $k = l$. Remarquons que l'algorithme ne s'arrête que quand il n'y a plus de demandes compatibles. Si par l'absurde $k < l$, alors t_{k+1} est une demande compatible avec les demandes précédentes, mais l'algorithme s'est déjà arrêté sur s_k ce qui est une contradiction. ■

On peut ensuite s'intéresser à la complexité de l'algorithme. Il y a tout d'abord le tri de n objets, ce que l'on peut réaliser en $\mathcal{O}(n \log n)$. Ensuite à chaque itération il y a un test de compatibilité puis une affectation dans le cas où le test est positif. Le test de compatibilité consistant à vérifier une inégalité, il est clair que chaque itération se fait en temps constant. La complexité globale de l'algorithme esst donc en $\mathcal{O}(nlogn + n) = \mathcal{O}(n \log n)$.

Exemple. — Considérons l'ensemble de demandes représenté dans la tableau ci dessous :

	e_1	e_2	e_3	e_4
d	0	0	2	3
f	1	3	6	6

Le tri des demandes donne $f(e_1) < f(e_2) < f(e_3) = f(e_4)$.
Voici ensuite les différentes itérations :
- $S = \{e_1\}$;
- e_2 étant incompatible avec e_1, rien ne se passe ;
- e_3 étant compatible avec e_1 on a $S = \{e_1, e_3\}$;
- e_4 étant incompatible avec e_1, e_3 et e_5 rien ne se passe.

La solution retournée est donc $S = \{e_1, e_3\}$.

Remarque 1. — Le temps de location total correspondant à la solution produite est de 5 unités. Par ailleurs il est clair que $\{e_2, e_4\}$ est aussi une solution dont le temps total de location est de 6 unités. Ceci prouve que notre algorithme ne permet pas de maximiser le temps de location.

Remarque 2. — Pour maximiser le temps de location on pourrait penser adapter l'algorithme en classant les demandes par durée décroissante. Notons $l(e) = f(e) - d(e)$ la durée de la demande e. Reprenons notre exemple précédent.
Le tri des demandes donne $l(e_3) > l(e_2) = l(e_4) > f(e_1)$.
Voici ensuite les différentes itérations :
- $S = \{e_3\}$;
- e_2 étant incompatible avec e_3, rien ne se passe ;
- e_4 étant incompatible avec e_3, rien ne se passe.
- e_1 étant compatible avec e_3 on a $S = \{e_1, e_3\}$;

La solution retournée est donc $S = \{e_1, e_3\}$, dont la durée totale n'est pas maximale. Cette méthode ne permet donc pas de maximiser la durée totale de location.

Remarque 3. — Notons que le classement des demandes par date de *début* croissante ne fournit pas non plus de solution optimale. On peut le voir très facilement sur l'exemple ci-dessous :

	e_1	e_2	e_3
d	0	1	2
f	3	2	3

Remarque 4. — Si l'agence dispose de plusieurs camions, on pourrait penser adapter la même méthode, en affectant le premier camion disponible. Malheureusement cette méthode ne donne pas de solution optimale. Pour la location de deux camions prenons l'exemple ci-dessous :

	e_1	e_2	e_3	e_4
d	0	0	2	1
f	1	2	3	4

Le tri des demandes donne $f(e_1) < f(e_2) < f(e_3) < f(e_4)$.
Voici ensuite les différentes itérations :
- $S_1 = \{e_1\}$;
- e_2 étant incompatible avec e_1, on utilise le deuxième camion $S_2 = \{e_2\}$;
- e_3 étant compatible avec e_1, on a $S_1 = \{e_1, e_3\}$

- e_4 étant incompatible avec e_1 et e_3 d'une part et avec e_2 d'autre part, rien ne se passe.
La solution retournée est donc $S_1 = \{e_1, e_3\}$ et $S_2 = \{e_2\}$. Celle ci n'est pas optimale car la solution $T_1 = \{e_1, e_4\}$ et $T_2 = \{e_2, e_3\}$ permet de satisfaire tous les clients.
Le problème se situe au niveau du traitement de la demande e_3 : c'est le deuxième camion qu'il faut lui affecter et non pas le premier. Mais s'il est facile pour nous a posteriori de faire cette constatation, l'algorithme glouton n'a lui aucune raison d'affecter le deuxième camion à la demande e_3.

2. Arbres couvrants de poids maximum

Commençons par un peu de vocabulaire. Un *graphe* (non-orienté) $G = (S, A)$ est la donnée d'un ensemble S de *sommets* et d'un ensemble A d'*arêtes*, chaque arêtes étant par définition une paire de sommets. Une *chaîne* (ou *sommet*) de longeur k reliant les sommets u et v est une suite finie de sommets $(v_1, ..., v_k)$ telle que $v_1 = u$, $v_k = v$ et $\{v_i, v_{i+1}\} \in A$ pour tout $i \in \{1, ..., k-1\}$. Une chaîne est *élémentaire* si ses sommets sont deux à deux distincts.
Un *cycle* est une chaîne $(v_1, ..., v_k)$ telle que $v_1 = v_k$. Ce cycle est élémentaire si $v_1, ...v_{k-1}$ sont deux à deux distincts.
On dit qu'un sommet v est *accessible* depuis le sommet u s'il existe une chaîne reliant u et v. L'accessibilité est une relation d'équivalence sur S dont les classes sont les *composantes connexes*. S'il n'y a qu'une classe le graphe est dit *connexe*.
Un graphe est *acyclique* s'il ne contient aucun cycle élémentaire non trivial (on parle aussi de *forêt*).
Un *arbre* est un graphe connexe acyclique.
Un arbre *couvrant* d'un graphe G est par définition un arbre qui contient tous les sommets de G. Pour avoir un arbre couvrant, un graphe doit être connexe. Dans le cas contraire on peut s'intéresser aux arbres couvrants de chaque composantes connexes de G. Notons qu'un sous-arbre de G est couvrant si et seulement s'il est maximal pour l'inclusion.
Par ailleurs on peut montrer qu'un graphe connexe possédant s sommets et a arêtes est un arbre si et seulement si $a = s - 1$.
On considère un graphe (non orienté) connexe $G = (S, A)$ (on note $s = |S|$ et $a = |A|$) et une fonction de pondération $w : A \to \mathbb{R}_+$. Pour toute partie $X \subset A$ le poids de X est défini par

$$f(X) = \sum_{x \in X} w(x).$$

Le but est de trouver un arbre couvrant de G de poids maximum.

2.1. Algorithme de Kruskal. — Pour résoudre ce problème Kruskal a proposé un algorithme qui est historiquement le premier algorithme glouton (celui-ci date de 1956).

KRUSKAL(G,w)
  Trier les élements de A par ordre de poids décroissants $w(e_1) \geq w(e_2) \geq \cdots \geq w(e_a)$
  $T \leftarrow \varnothing$
  pour $i = 1$ à a **faire**
    si $T \cup \{e_i\}$ est acyclique **alors**
      $T \leftarrow T \cup \{e_i\}$
    fin si
  fin pour
  retourner T

Comme pour l'exemple précédent, cet algorithme est glouton car il construit l'ensemble T de manière séquentielle et qu'il fait à chaque étape le meilleur choix local (c'est à dire une arête de poids maximum).
La figure 1 donne un exemple d'exécution.
L'optimalité de cet algorithme est une conséquence du théorème 3.8 dans le cas d'un matroïde graphique (voir section 3).

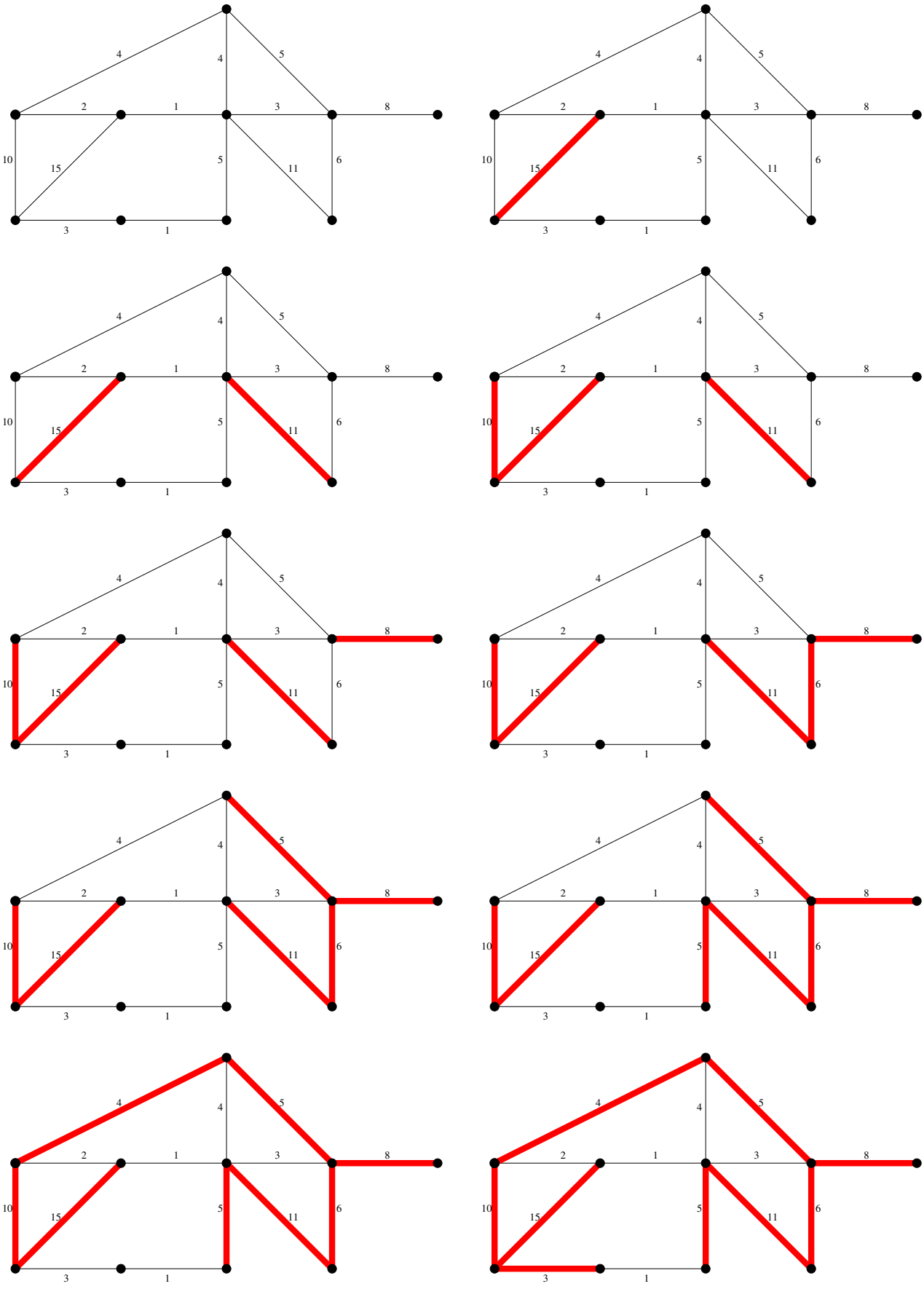

FIGURE 1. Algorithme de Kruskal

Remarque 1. — Si l'on désire trouver un arbre couvrant de poids *minimum*, on peut modifier la fonction de pondération de la façon suivante :

$$M = \max\{w(e) \mid e \in A\}$$
$$\forall e \in A \quad w'(e) = M - w(e).$$

L'exécution de KRUSKAL(G,w') fournit alors un arbre couvrant de poids minimum. Ce problème se rencontre par exemple pour minimiser la longueur de cable nécessaire pour mettre en réseau un ensemble d'ordinateurs.

Remarque 2. — Cet algorithme fonctionne également si le graphe G n'est plus connexe. Le résultat est alors un arbre couvrant de poids maximum dans chaque composante connexe de G. Ce que l'on obtient est en fait une forêt maximale (au sens de l'inclusion), de poids maximum. Contrairement au cas des arbres, une *forêt couvrante*, n'est pas nécessairement maximale (pour l'inclusion). Par contre une forêt maximale est bien sûr couvrante.

Remarque 3. — Revenons maintenant à un graphe connexe. Si la pondération est strictement positive, une sous-forêt de poids maximum est nécessairement maximale. Si l'on autorise des poids négatifs, il va par contre exister des sous-forêts de poids strictement supérieur au poids d'un arbre couvrant de poids maximum. Tout dépend alors de l'objectif : veux-t-on absolument un ensemble *couvrant* ou bien le but est-il plutôt de maximiser f ? Dans le second cas on utilise alors le même algorithme mais en laissant de côté les arêtes de poids négatif. Bien sûr ce problème ne se pose pas avec des poids positifs.

Remarque 4. — La complexité de l'algorithme va dépendre des structures de données utilisées (pour le graphe G et la forêt T). Le tri demande tout d'abord un temps $\mathscr{O}(a \log a)$. A chaque itération il faut tester si l'arête e_i est compatible avec T, puis éventuellement la rajouter à T. Il est possible d'implémenter l'algorithme de sorte que chaque itération soit en $\mathscr{O}(logs)$ (pour plus de détail voir les sections 21.3, 21.4 et 23.2 de [**3**]). Ainsi la boucle est en $\mathscr{O}(a \log s)$. Comme $a \leq s^2$ on a $\log a = \mathscr{O}(\log s)$ et la complexité totale de l'algorithme est en $\mathscr{O}(a \log s)$.

2.2. Algorithme de Prim. — Nous allons maintenant évoquer un autre algorithme permettant de trouver un arbre couvrant de poids maximum, il s'agit de l'algorithme de Prim (datant de 1957). Son principe est le même que pour l'algorithme de Kruskal, sauf que l'ensemble T est à chaque étape un arbre contenant un sommet fixé v_0. Cet algorithme ne s'applique que pour un graphe G connexe.

PRIM(G,w,v_0)
  $T \leftarrow \varnothing$
  tant que T n'est pas couvrant **faire**
    Choisir une arête e de poids maximum telle que $T \cup \{e\}$ soit un arbre contenant v_0
    $T \leftarrow T \cup \{e\}$
  fin tant que
  retourner T

La figure 2 donne un exemple d'exécution de l'algorithme.

Remarque. — Comme pour l'algorithme de Kruskal, on peut montrer que l'algorithme de Prim est exécuté en temps $\mathscr{O}(a \log s)$. Ce temps de calcul peut même être légèrement amélioré en utilisant des structures de données judicieuses (voir la section 23.2 de [**3**]).

Pour finir, la preuve de l'optimalité de l'algorithme de Prim est la conséquence de résultats généraux que nous verrons dans la section 4.

3. Matroïdes et algorithmes gloutons

La formalisation des algorithmes gloutons exposée ici date de 1971 (voir [**4**]). Celle-ci ne couvre pas toutes les applications possibles des méthodes gloutonnes. Il existe des problèmes qui ne rentrent pas

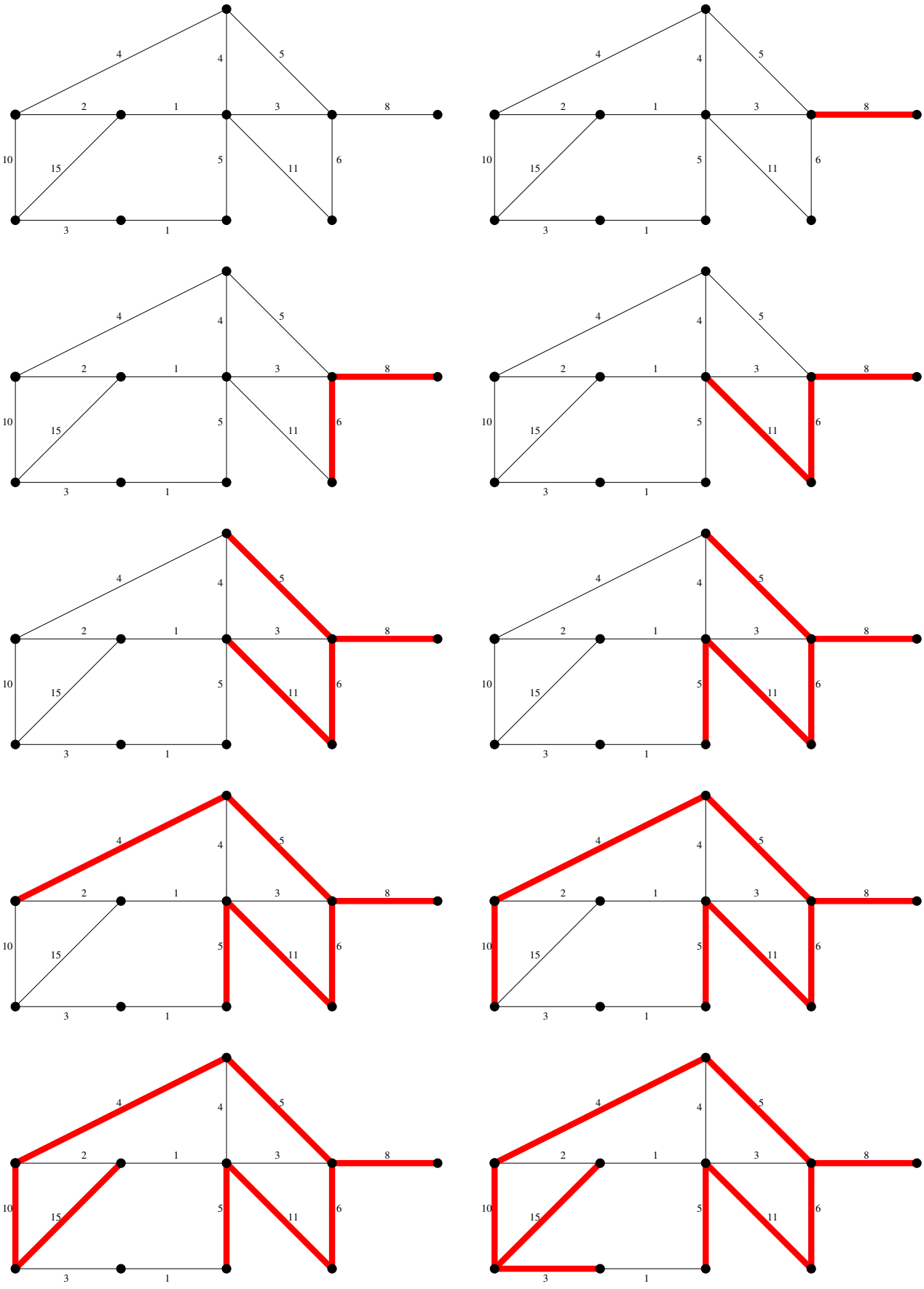

FIGURE 2. Algorithme de Prim (v_0 est le sommet le plus à droite)

dans cette théorie, bien qu'un algorithme glouton permette de les résoudre. Citons par exemple les problèmes suivants (liste non exhaustive) :
– location d'un véhicule ;
– codage de Huffman ;
– algorithme de Dijkstra pour le problème du plus court chemin à origine unique (avec poids positifs) ;
– problème de la couverture d'ensemble (algorithme d'approximation).
Néanmoins cette théorie est assez générale pour permettre de comprendre de nombreux exemples. De plus un certain nombre de généralisations ont ensuite vu le jour (voir la section suivante).
Pour plus de détails on pourrra consulter [**3**], [**5**], [**1**] ou bien [**2**].

3.1. Définitions et exemples. — Les matroïdes ont été introduits par Whithey en 1935 (voir [**12**]) pour formaliser la notion d'indépendance linéaire dans un espace vectoriel. La théorie des matroïdes a ensuite connu un grand nombre d'applications, par exemple en combinatoire ou en géométrie.

Définition 3.1. — Un *système ensembliste* $\mathscr{S} = (E, \mathscr{F})$ est la donnée d'un ensemble fini E et d'une collection $\mathscr{F}$ de sous-ensembles de E.

Définition 3.2. — On appelle *extension* de $F \in \mathscr{F}$ tout élément $x \notin F$ tel que $F \cup \{x\} \in \mathscr{F}$. On notera $ext(F)$ l'ensemble des extensions de F.

Définition 3.3. — Un ensemble de la collection $\mathscr{F}$ est dit *indépendant*. Un ensemble indépendant maximal pour l'inclusion est une *base*. On notera $\mathscr{B}$ l'ensemble des bases.

Les systèmes que nous allons étudier vont vérifier certains des axiomes suivants :

(trivial)	$\varnothing \in \mathscr{F}$
(hérédité)	$\forall X \in \mathscr{F}, \forall Y \subset E,\ Y \subset X \Rightarrow Y \in \mathscr{F}$
(augmentation)	$\forall X, Y \in \mathscr{F},\ \lvert X\rvert = \lvert Y\rvert + 1 \Rightarrow \exists x \in X \setminus Y,\ Y \cup \{x\} \in \mathscr{F}$
(échange)	$\forall X, Y \in \mathscr{F},\ \lvert X\rvert < \lvert X\rvert \Rightarrow \exists x \in X \setminus Y,\ Y \cup \{x\} \in \mathscr{F}$

Définition 3.4. — Un système est *héréditaire* s'il vérifie l'axiome trivial et l'axiome d'hérédité. Un système héréditaire est un *matroïde* s'il vérifie en plus l'axiome d'augmentation.

Lemme 3.5. — *Un matroïde vérifie l'axiome d'échange.*

Démonstration. — Soit $X, Y \in \mathscr{F}$ avec $|Y| = |X| + 1$. Soit $Z \subset X$ de cardinal $|Y| + 1$. En utilisant l'hérédité on a $Z \in \mathscr{F}$. En utilisant l'augmentation appliquée à Y et Z, on trouve $x \in Z \setminus Y \subset X \setminus Y$ tel que $Y \cup \{x\} \in \mathscr{F}$. ■

Lemme 3.6. — *Soit $\mathscr{S}$ un système vérifiant l'axiome d'échange (par exemple un matroïde). Alors les bases de $\mathscr{S}$ ont toutes le même cardinal (appelé rang de $\mathscr{S}$).*

Démonstration. — Si $B, C \in \mathscr{S}$, et que par l'absurde $|B| < |C|$ alors l'axiome d'échange assure l'existence d'un élément $x \in C \setminus B$ tel que $B \cup \{x\} \in \mathscr{F}$. Mais alors B n'est pas une base : contradiction. ■

Exemple 1. — Soit E un ensemble fini de vecteurs de $\mathbb{R}^n$ (par exemple les colonnes d'une matrice). Soit $\mathscr{F}$ la collection des ensembles formés de vecteurs linéairement indépendants. Alors $(E, \mathscr{F})$ est un matroïde. C'est historiquement la première famille de matroïdes (voir [**12**]) et le mot matroïde vient d'ailleurs du mot matrice. Notons au passage que le vocabulaire utilisé (ensemble *indépendant*, *base*,...) provient de l'algèbre linéaire.

Exemple 2. — Dans le problème de la location d'un camion, notons E l'ensemble des demandes et $\mathscr{F}$ la collection formée des ensembles de demandes compatibles. Il est clair que $(E, \mathscr{F})$ est un système héréditaire.
Considérons maintenant l'exemple suivant :

	e_1	e_2	e_3
d	0	0	1
f	2	1	2

On obtient alors $\mathscr{F} = \{\{e_1\}, \{e_2\}, \{e_3\}, \{e_2, e_3\}\}$ et $\mathscr{B} = \{\{e_1\}, \{e_2, e_3\}\}$. Ainsi le système $(E, \mathscr{F})$ possède des bases de cardinalité différente : ce n'est pas un matroïde.

Exemple 3. — Soit $G = (S, A)$ un graphe, posons $E_G = A$ et soit $\mathscr{F}_G$ la collection des ensembles acycliques de G. Le système $\mathscr{S}_G = (E_G, \mathscr{F}_G)$ est un matroïde appelé *matroïde graphique* associé à G.

Démonstration. — L'axiome trivial et l'axiome d'hérédité sont clairement satisfaits. On s'intéresse donc à l'augmentation : soit X, Y deux ensembles acycliques avec $|X| = |Y| + 1$.
La forêt (S, Y) se décompose en composantes connexes (S_i, Y_i) avec $i \in \{1, ..., l\}$. On suppose la numérotation telle que $S_{k+1}, ..., S_l$ soient les composantes connexes réduites à un singleton. Ainsi $Y_{k+1} = ... = Y_l = \varnothing$ et on a $Y = \bigsqcup_{i=1}^{k} Y_i$.
On va maintenant partitionner X à l'aide des S_i. Pour $i \in \{1, ..., k\}$ on appelle X_i l'ensemble des arêtes de X dont les deux extrémités sont dans S_i. On notera X' l'ensemble des arêtes de X dont les extrémités sont dans des S_i différents. On a donc $X = \left(\bigsqcup_{i=1}^{k} X_i\right) \sqcup X'$.
Remarquons maintenant qu'une arête de X' peut être ajoutée à Y sans créer de cycle puisqu'elle relie des sommets de Y situés dans des composantes connexes différentes. Pour terminer notre preuve il suffit donc de prouver que X' est non vide.
Puisque X_i est acyclique avec sommets dans S_i, on a $|X_i| \leq |S_i| - 1$. Par ailleurs Y_i étant un arbre de sommets S_i on a l'égalité $|Y_i| = |S_i| - 1$ et donc $|X_i| \leq |Y_i|$. En sommant pour $i \in \{1, ..., k\}$ on obtient $|\bigsqcup_{i=1}^{k} X_i| \leq |Y|$. Comme $|X| = |Y| + 1$, cela force X' à être non vide. ■

3.2. Lien avec les algorithmes gloutons. — On considère un système héréditaire $\mathscr{S} = (E, \mathscr{F})$ et une fonction (pondération) $w : E \to \mathbb{R}$. Pour tout $F \in \mathscr{F}$ le poids de F est

$$f(X) = \sum_{x \in X} w(x).$$

Le but est de trouver une *base* de $\mathscr{S}$ de poids maximum. On propose l'algorithme glouton suivant :

GLOUTON1($\mathscr{S}$,w)
  Trier les élements de E par ordre de poids décroissants $w(e_1) \geq w(e_2) \geq \cdots \geq w(e_n)$
  $F \leftarrow \varnothing$
  pour $i = 1$ à n **faire**
    si $F \cup \{e_i\} \in \mathscr{F}$ **alors**
      $F \leftarrow F \cup \{e_i\}$
    fin si
  fin pour
  retourner F

La complexité de cet algorithme dépend essentiellement du coût du test d'indépendance. Si celui ci se fait en $\mathscr{O}(T(n))$, le temps de calcul total est en $\mathscr{O}(n(\log n + T(n)))$.

Remarque 1. — Dans le cas d'un matroïde graphique on retrouve exactement l'algorithme de Kruskal.

Remarque 2. — S'il y a des poids négatifs, un ensemble indépendant non maximal peut avoir un poids supérieur au poids d'une base. On retrouve exactement la même discussion que pour l'algorithme de Kruskal.

Lemme 3.7. — *Soit $\mathscr{S} = (E, \mathscr{F})$ un système héréditaire. Alors l'ensemble retourné par l'algorithme* GLOUTON1 *est une base de $\mathscr{S}$.*

Démonstration. — Soit T l'ensemble produit par l'algorithme. Si par l'absurde T n'est pas une base alors il existe $B \in \mathscr{B}$ tel que $T \subsetneq B$. Soit $e_0 \in B \setminus T$. Puisque $T \cup \{e_0\} \subset B$, par hérédité on a $T \cup \{e_0\} \in \mathscr{F}$. Lors de l'examen de e_0 par GLOUTON1, le test est donc positif. Ceci est une contradiction puisque $e_0 \notin T$. ■

***Théorème 3.8* (Edmonds 1971).** — *Soit $\mathscr{S} = (E, \mathscr{F})$ un système héréditaire. Alors l'algorithme* Glouton1 *fournit une solution optimale pour toute pondération w si et seulement si $\mathscr{S}$ est un matroïde.*

Démonstration. — Pour commencer, d'après le lemme précédent le résultat est toujours une base.
Sens direct : il faut prouver l'augmentation, soit donc $X, Y \in \mathscr{F}$ avec $|X| = |Y|+1$. Posons $p = |X \cap Y|$ et $q = |Y \setminus X|$: ainsi $|Y| = p + q$ et $|X| = p + q + 1$. On définit $w : E \to \mathbb{R}$ par

$$w(e) = \begin{cases} 2q+2 & \text{si } \ e \in Y \\ 2q+1 & \text{si } \ e \in X \setminus Y \\ 0 & \text{sinon} \end{cases}$$

Notons T le résultat de l'algorithme. Lors de l'exécution, les éléments de Y sont examinés et ajoutés en premier. On a donc $Y \subset T$. Par ailleurs on voit facilement que $f(Y) = (p+q)(2q+2)$ et $f(X) = p(2q+2) + (q+1)(2q+1) = f(Y) + q + 1$. Ainsi X est un ensemble indépendant de poids strictement supérieur au poids de Y. Puisque T est optimal, cela force T à contenir au moins un élément de $X \setminus Y$. Si e est un tel élément alors $Y \cup \{e\} \subset T$ est indépendant par hérédité, ce qui prouve l'augmentation.
Réciproque : notons X le résultat de l'algorithme et soit Y une base optimale avec (par l'absurde) $f(Y) > f(X)$ et telle que $|Y \cap X|$ soit maximum (parmis les Y ayant la même propriété).
Notons que $X \setminus Y$ est non vide (sinon $X \subset Y$ donc $X = Y$ ce qui contredit $f(Y) > f(X)$). Soit x_0 le premier élément choisi par l'algorithme qui n'est pas dans Y.
Montrons que les éléments de $Y \setminus X$ sont de poids inférieur ou égal à $w(x_0)$. En effet si y est un tel élément avec $w(y) > w(x_0)$, cela signifie qu'il est examiné par l'algorithme avant x_0. Mais avant x_0 tous les éléments de T sont dans Y et donc y est compatible avec ceux-ci. On a une contradiction puisque y n'a pas été retenu par l'algorithme.
Nous gardons ce résultat de côté, et nous nous intéressons maintenant à $(X \cap Y) \cup \{x_0\}$. C'est un élément de $\mathscr{F}$ avec $|(X \cap Y) \cup \{x_0\}| \leq |Y|$. Par augmentation successive il existe $Y' \subset Y \setminus (X \cap Y) \cup \{x_0\}$ tel que $Z = (X \cap Y) \cup \{x_0\} \cup Y'$ soit une base.
Puisque $|(X \cap Y) \cup Y'| = |Z| - 1 = |Y| - 1$, il existe $y_0 \in Y \setminus (X \cap Y) \cup Y'$ tel que $Y = (X \cap Y) \cup Y' \cup \{y_0\}$. Puisque $y_0 \in Y \setminus X$, d'après le petit résultat prouvé précédemment on a $w(y_0) \leq w(x_0)$. On en déduit que Z est une base telle que $f(Z) \geq f(Y)$. Puisque Y est optimale, Z l'est donc aussi (et il y égalité des poids). Mais $Z \cap X = (X \cap Y) \cup \{x_0\}$ a un élément de plus que $X \cap Y$, ce qui contredit la maximalité de $|X \cap Y|$. ■

Ce théorème démontre donc la validité de l'algorithme glouton dans le cas des matroïdes. En particulier on dispose d'une preuve d'optimalité de l'algorithme de Kruskal.

4. Généralisation : les gloutonoïde

La notion de matroïde est trop forte pour couvrir un nombre suffisant de cas. Il a donc été nécessaire d'alléger les axiomes (en particulier l'axiome d'hérédité).

4.1. Gloutonoïdes. — La théorie des gloutonoïdes a été engagée par Korte et Lovász (voir [**7**], [**8**], [**9**], [**10**] et [**11**]). On peut aussi consulter [**1**], ou bien [**6**].
On considère les nouveaux axiomes suivant :

(accessibilité) $\forall X \in \mathscr{F},\ X \neq \varnothing \Rightarrow \exists x \in X,\ X \setminus \{x\} \in \mathscr{F}$
(échange fort) $\forall X \in \mathscr{F}, \forall B \in \mathscr{B}$, si $x \notin B$ et $X \cup \{x\} \in \mathscr{F}$
alors $\exists y \in B \setminus X$ tel que $X \cup \{y\} \in \mathscr{F}$ et $B \cup \{x\} \setminus \{y\} \in \mathscr{F}$

***Définition 4.1*.** — Un système $\mathscr{S} = (E, \mathscr{F})$ est dit *accessible* s'il vérifie l'axiome trivial et l'axiome d'accessibilité. Un gloutonoïde est un système accessible vérifiant l'axiome d'augmentation.

***Lemme 4.2*.** — *Si $\mathscr{S}$ vérifie l'axiome trivial, alors l'axiome d'échange est équivalent aux axiomes d'accessibilité et d'augmentation. En particulier un gloutonoïde vérifie l'axiome d'échange.*

***Corollaire 4.3*.** — *Les bases d'un gloutonoïde ont toutes le même cardinal.*

Exemple. — Soit $G = (S, A)$ un graphe.
Posons $E_G = A$ et soit $\mathscr{F}_{G,v_0}$ la collection des arbres contenant v_0.
Le système $\mathscr{M}_G = (E_G, \mathscr{F}_G)$ est un gloutonoïde appelé *gloutonoïde graphique* associé à G, enraciné au point v_0.

Remarque 1. — De la même façon que les matroïdes permettent de formaliser le concept d'indépendance linéaire, il existe une notion d'*antimatroïde* pour formaliser la notion de convexité. Les antimatroïdes sont aussi des gloutonoïdes.

Remarque 2. — Sans la racine v_0 l'axiome d'augmentation n'est pas vérifié.
Il suffit pour le voir de prendre deux arbres n'ayant aucun sommet en commun.

4.2. Algorithme glouton sur un gloutonoïde. — On propose maintenant une version un peu modifié de l'algorithme GLOUTON1 (qui ne marche que pour un sytème héréditaire) :

GLOUTON2($\mathscr{S}$,w)
  $F \leftarrow \varnothing$
  tant que $ext(F) \neq \varnothing$ **faire**
    Choisir $e \in ext(F)$ de poids maximum
    $F \leftarrow F \cup \{e\}$
  fin tant que
  retourner F

Sous cette forme générale l'algorithme a un coût très important puisqu'il calcule $ext(F)$ à chaque étape. Mais là encore on ne peut pas en dire plus car la complexité précise dépend de l'implémentation.
On dispose ensuite d'un théorème permettant de caractériser les gloutonoïdes pour lesquels l'algorithme est optimal.

***Théorème 4.4* (Korte-Lovász 1984).** — *Soit $\mathscr{S} = (E, \mathscr{F})$ un gloutonoïde. Alors l'algorithme* GLOUTON2 *fournit une solution optimale pour toute pondération w si et seulement si $\mathscr{S}$ vérifie l'axiome d'échange fort.*

Remarque 1. — L'accessibilité est une condition de base pour que l'algorithme glouton puisse contruire une solution séquentiellement.

Remarque 2. — Dans le cas d'un gloutonoïde graphique enraciné, on retrouve bien sûr l'algorithme de Prim. On peut d'ailleurs vérifier que l'axiome d'échange est alors vérifié.

Remarque 3. — Pour l'algorithme de Prim il est important d'imposer à chaque étape que l'arbre en construction contienne un sommet donné pour prouver que l'on a un gloutonnoïde. Mais si on s'affranchit de cette condition, on constate que l'algorithme fonctionne encore. En fait on obtient alors un *plongement matroïdique*, une autre généralisation possible des matroïdes.

Références

[1] B. Charlier, *The Greedy Algorithms Class : Formalization, Synthesis and Generalion*, `ftp://ftp.info.ucl.ac.be/pub/reports/95/rr95-11.ps.gz`.

[2] R. Cori, G. Hanrot et J-M Steyaert, *Conception et analyde d'algorithmes*, `http://www.lix.polytechnique.fr/~kenyon/Enseignement/poly1.ps`

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein, *Introduction à l'algorithmique*, 2e édition, Dunod, 2002.

[4] J. Edmonds, *Matroids and the greedy algorithm*, Math. Programming **1** (1971), 127-136.

[5] M. Gondran et M. Minoux, *Graphes et algorithmes*, Eyrolles, 1979.

[6] P. Helman, B. M.E. Moret et H. D. Shapiro, *An Exact Characterization of Greedy Structures*, `http://www.cs.unm.edu/~moret/hms_IPCO.ps`

[7] B. Korte et L. Lovász, *Mathematical structures underlying greedy algorithms*, Lectures Notes in Comput. Sci. *177*, 205-209, Springer Verlag, 1981.

[8] B. Korte et L. Lovász, *Structural properties of greedoïds*, Combinatorica **3** (1983), 359-374.

[9] B. Korte et L. Lovász, *Greedoïds – a structural framework for the greedy algorithm*, in W. Pulleybank (Ed.), Progress in Combinatorial Optimization, 221-243, Academic Press, 1984.

[10] B. Korte et L. Lovász, *Greedoïds and linear objective functions*, SIAM J. ALg. Disc. Meth. **5** (1984), 229-238.

[11] B. Korte, L. Lovász et R. Scrader, *Greedoïds*, Springer Verlag, 1991.

[12] H. Whitney, *On the abstract properties of linear dependence*, Am. J. Math. **57** (1935), 507-553.

---

7 mai 2003

Pierre Béjian • *E-mail :* `bejian@free.fr` • *Url :* `http://bejian.free.fr`