Chapitre 3: Les algorithmes voraces

1 Introduction

Les algorithmes voraces sont appliqués à une grande variété de problèmes. La plupart, mais non tous, de ces problèmes exigent la recherche de solutions devant satisfaire certaines contraintes. Tout ensemble de solutions satisfaisant ces contraintes est appelé solution réalisable.

Dans certains cas, le problème consiste à déterminer, dans l'ensemble C de tous les candidats à la solution, une solution réalisable qui maximise ou minimise une fonction objectif donnée: ce sont ls problèmes d'optimisation. Pour les problèmes qui ne sont pas des problèmes d'optimisation, les algorithmes voraces construisent leur solution, dans ce cas, en considérant les données définissant le problème considéré dans un certain ordre.

L'intérêt et la force des algorithmes voraces résident dans leur simplicité de construire une solution. De plus, cette solution se construit généralement en des temps raisonnables. L'inconvénient de cette technique est de toujours recourir à une preuve pour montrer l'optimalité de la solution ainsi générée, si optimalité il y a.

Dans ce chapitre, nous allons nous concentrer sur les problèmes d'optimisation.

Definition 1 *Un algorithme vorace (glouton, en anglais greedy) est une procédure algorithmique qui construit d'une manière incrémentale une solution. À chaque étape, cette technique prend la direction la plus prometteuse, suivant des règles bien simples, en considérant une seule donnée à la fois.*

Il est utile de signaler que ce choix, localement optimal, n'a aucune raison de conduire à une solution globalement optimale. Cependant, certains problèmes peuvent être résolus d'une manière optimale ainsi. Autrement dit, l'optimalité locale conduit à l'optimalité globale. Dans d'autres cas, la méthode gloutonne est seulement une heuristique qui ne conduit qu'à une solution sous-optimale, mais qui peut être utilisée quand on ne connaît pas d'algorithme exact efficace. On reviendra, dans le dernier chapitre de ce cours, sur les algorithmes voraces en tant que solutions heuristiques.

L'algorithme générique d'une procédure vorace (gloutonne) peut être résumé comme suit :

```
Algorithme vorace {
  S = ∅; C: ensemble des candidats à la solution;
  Tant que S n'est pas une solution et C ≠ ∅
  {
    x = choisir un élément de C le plus prometteure;
    C = C − x;
    si réalisable(solution,x)
      solution = solution ∪{x}
  }
  Si S est une solution alors retourner S
  sinon retourner pas de solution;
}
```

Le choix de la donnée la plus prometteuse, lors de la construction de la solution, se fait suivant une règle. Cette règle définit en réalité la stratégie de l'algorithme vorace ainsi conçu.

Remarque 1: Il est utile de noter que, à la fin de leur exécution, les algorithmes voraces ne génèrent qu'une seule solution. De plus, un algorithme vorace ne revient jamais sur une décision prise précédemment.

Remarque 2: La fonction de sélection est généralement basée sur la fonction objectif à optimiser.

Considérons l'exemple suivant pour mieux clarifer la terminologie introduite ci-dessus.

Exemple 1 *Soit le problème de rendre la monnaie à un client en lui donnant le moins de pièces possibles. Les éléments du schéma ci-dessus sont résumés comme suit:*

1. les candidats potentiels à la solution: ensemble fini de pièces de monnaie, par exemple: 1, 5, 10 et 25 centimes.
2. une solution: le total de l'ensemble de pièces choisies correspondant exactement au montant à payer.
3. une solution réalisable: le total de l'ensemble de pièces chosies ne dépasse pas le montant à payer.
4. la donnée la plus prometteuse est la grande pièce qui reste dans l'ensemble des candidats.
5. fonction objectif: minimiser le nombre de pièces utilisées dans la solution.

L'algorithme vorace pour cet exemple est le suivant:

$S = \emptyset$; total = 0;
tant que total < somme à payer et $C \neq \emptyset$ {
x = plus grande pièce;
$C = C - x$;
si total (S) + x < somme à payer
solution = solution $\cup\{x\}$
}
Si S est une solution alors retourner S
sinon retourner pas de solution;
}

Soit par exemple la monnaie de 87sous à rendre en utilisant seulement $\{25, 10, 5, 1\}$. L'algorithme ci-dessus va générer la solution suivante: $3 \times 25 + 1 \times 10 + 2 \times 1$.

Une fois l'algorithme est devenu un plus clair, il nous reste à savoir si la solution ainsi engendrée est toujours optimale. Cela est un autre problème!

Passons maintenant aux exemples suivants pour mieux comprendre le fonctionnement des algorithmes voraces.

2 Les algorithmes voraces sur les problèmes d'ordonnancement

Un problème d'ordonnancement consiste à réaliser un ensemble de n tâches sur un ensemble de m machines de telle manière à minimiser une certaine fonction objectif. Le passage d'une tâche sur une machine est appelé opération, et prends un certain temps d'exécution. Pour être réalisées, chacune des tâches possède un ordre de passage sur l'ensemble des machines. Il existe plusieurs modèles d'ordonnancement suivant le nombre d'opérations de chaque tâche, le nombre de machines et la nature de l'ordre de passage sur les machines.

Ainsi, si le nombre d'opérations de chacune des tâches est limitée à un, on distingue le mdoèle à une machine ou celui des machines parallèles. Si le nombre d'opérations est plus grand que l'unité, alors on distingue trois modèles: Dans le modele d'open shop, l'ordre de passage des taches sur les machines n'est pas pré-établie. Dans le modèle de flow shop, l'ordre de passage est le même pour toutes les tâches. Dans le modèle de job shop, chacune des tâches possède son propre odre de passage sur les machines.

2.1 Minimisation de l'attente sur une machine

Étant donnés donc n tâches et une seule machine, le problème consiste donc à trouver une manière d'exécuter ces tâches de telle manière que le temps moyen d'attente de chacune de ces

tâches soit minimisé. Le temps d'exécution de la tâche i étant dénoté par t_i.

Étant donné C_i le temps de fin de la tâche i dans une solution donnée, le temps moyen d'attente de ces n tâches dans cetet solution est alors exprimée par la relation suivante

$$T = \frac{1}{n}\sum_{i=1}^{n} C_i$$

Exemple 2 *Si nous avons 3 tâches avec les temsp d'éxécution suivants*

$$t_1 = 7; t_2 = 10; t_3 = 2$$

Nous avons $n! = 3! = 6$ *manières différentes d'éxcuter ces 4 tâches. Consiédrons quelques une de ces pemutations et voyons ce qu'elles génèrent comme temps moyen d'attente* T*:*

$$1-2-3 : T = \frac{1}{3}(7 + (7+10) + (7+10+2)) = 43/3 = 14.33$$
$$1-3-2 : T = \frac{1}{3}(7 + (7+2) + (7+2+10)) = 35/3 = 11.66$$
$$2-1-3 : T = \frac{1}{3}(10 + (10+7) + (10+7+2)) = 46/3 = 15.33$$
$$2-3-1 : T = \frac{1}{3}(10 + (10+2) + (10+2+7)) = 41/3 = 13.66$$
$$3-1-2 : T = \frac{1}{3}(2 + (2+7) + (2+7+10)) = 30/3 = 10.0$$
$$3-2-1 : T = \frac{1}{3}(2 + (2+10) + (2+10+7)) = 33/3 = 11.0$$

Comme on peut le constater, à chacune des ces solutions, correspond une valeur du temps moyen d'attente. Comme dans notre cas, toutes les solutions possibles ont été générées, il est facile de voir que la valeur optimale est 10 correspondant à la solution 3-1-2. Autrement dit, pour résoudre d'une manière optimale le problème ci-dessus, il est nécessaire d'exécuter les tâches dans l'ordre: 3-1-2.

La stratégie de résolution pourrait être comme suit: À chaque étape, parmi les tâches restantes, prendre celle dont le temps d'exécution est plus petit. En langage algorithmique, cette stratégie pourrait se tradire comme ci-dessous.

L'algorithme

$S = \emptyset$; attente = 0;
renommer les tâches de telles manières que $t_1 \leq t_2 \leq \cdots \leq t_n$;
for (i= 1; i<= n; i++) {
    total = total + t_i;

```
    solution = solution ∪{i};
  }
attente = attente / n;
```

Theorem 1 *L'algorithme ci-dessus génère une solution minimisant le temps moyen d'attente.*

Preuve: Soit $P = \{p_1, p_2, \ldots, p_n\}$ une permutation quelconque des entiers $\{1, 2, \ldots, n\}$, indiquant les tâches. Si les tâches sont exécutées dans l'ordre P, alors le temps moyen généré par cette permutation est comem suit:

$$\begin{aligned} attente &= \frac{1}{n}\left(t_{p_1} + (t_{p_1} + t_{p_2}) + (t_{p_1} + t_{p_2} + t_{p_3}) + \ldots (t_{p_1} + \cdots + t_{p_{n-1}} + t_{p_n})\right) \\ &= \sum_{i=1}^{n}(n - i + 1)t_i \end{aligned}$$

La preuve va se faire par l'absurde. Soit donc P_{opt} une solution optimale qui ne vérifie pas la propriété de notre algorithme vorace. Cela signifie qu'il existe au moins deux tâches en position a et b telles que $t_a > t_b$ et $a < b$. Le temps moyen d'attente de cette solution est comme suit:

$$T_{opt} = \frac{1}{n}\left((n - a + 1)t_a + (n - b + 1)t_b + \sum_{i=1_{i\neq(a,b)}}^{n}(n - i + 1)t_i\right)$$

Interchangeons maintenant ces deux tâches. On obtient alors l'expression suivante pour l'attente

$$T = \frac{1}{n}\left((n - a + 1)t_b + (n - b + 1)t_a + \sum_{i=1_{i\neq(a,b)}}^{n}(n - i + 1)t_i\right) \quad (1)$$

En faisons la soustraction de ces deux expressions, on obtient

$$\begin{aligned} T_{opt} - T &= \frac{1}{n}(n - a + 1)(t_a - t_b) + (n - b + 1)(t_b - t_a)) \\ &= \frac{1}{n}(b - a)(t_a - t_b) > 0 \end{aligned}$$

Autrment dit, la deuxième solution est mieux que la première. Ceci signifie que la solution vorace ne peut être qu'optimale.

2.2 Ordonnancer avec des dates d'échéances

Soit le problème de n tâches à exécuter sur une seule machine. Chacune des tâches i possède un temps d'exécution unitaire, un gain $g_i > 0$ à chaque fois elle est réalisée avant sa date d'échéance d_i. Le problème est de déterminer une solution qui exécute toutes les tâches de telle manière que le gain total soit maximisé.

Soit l'exemple suivant:

Exemple 3

tâche i	*1*	*2*	*3*	*4*
g_i	*50*	*10*	*15*	*30*
d_i	*2*	*1*	*2*	*1*

Les permutations à consiérer sont comme suit

séquence	profit
1	50
2	10
3	15
3	15
4	30
1,3	65
2,1	60
2,3	25
3,1	65
4,1	80
4,3	45

Remarquez, par exemple, que la sequence 3,2 n'est pas considérée pour la simple raison que la tâche 2 ne peut pas s'exéuter au temps 2, c'est-à-dire après sa date d'échénace. Pour maximiser le gain, il y a donc lieur d'exécuter la séquence 4,1.

Un ensemble de tâches est réalisable s'il existe au moins une séquence qui permet à toutes les tâches de l'ensemble d'être exécutées avant leur date d'échéance.

> Un algorithme vorace pourrait être celui qui ajoute à chaque étape la tâche non encore considérée ayant la plus grande valeur de gain g_i, à condition que l'ensemble soit réalisable.

Dans l'exemple précédent, on choisit en premier la tâche 1. Ensuite, la tâche 4. L'ensemble $\{1, 4\}$ est réalisable car il peut être exécuté dans l'ordre 4,1. Ensuite, on essaye l'ensemble $\{1, 3, 4\}$ mais il n'est pas réalisable. Donc, la tâche 3 est rejetée. Ensuite, on essaye la tâche 2, pour former l'ensemble $\{1, 2, 4\}$. Il est facile de voir que cet ensemble n'est pas réalisable; la tâche 2 est donc rejetée. Notre solution - optimal dans ce cas, est donc l'ensemble $\{1, 4\}$ qui ne peut être exécuté que dans l'ordre 4,1.

L'étape qui suit est de montrer que la stratégie qu'on vient de présenter génère toujours une solution optimal.

Lemma 1 *Soit J un ensemble de k tâches. sans perte de généralté, on suppose que ces tâches sont triées dans l'ordre $d_1 \leq d_2 \leq \cdots \leq d_n$. L'ensemble J est réalisable si et seulement si la séquence $1, 2, \ldots, k$ est aussi réalisable.*

Preuve: L'implication du "si" est évidente. L'implication du "seulement si" se fait comme suit par la contreverse. Supposons que la séquence $1, 2, \ldots, k$ n'est pas réalisable. Alors il existe au moins une tâche dans cette séquence qui est exécutée après sa date d'échéance. Soit r cette tâche, alors nous avons bien $d_r \leq r - 1$. Comme les tâches sont exécutées dans l'ordre croissant des d_i, ce qui signifie qu'au moins r tâches ont une date d'échéance $\leq r-1$. Toutefois, ces tâches sont exécutées, la dernière d'entre-elle est toujours exécutée en retard. Par conséquent, J n'est pas réalisable. □

Ce résultat montre qu'il est suffisant de vérifier qu'une seule séquence, dans l'ordre croissant des dates d'échéance, pour savoir si un ensemble J est réalisable ou non.

Theorem 2 *L'algorithme vorace ci-dessus génère toujours une solution optimale.*

Preuve: Supposons que l'algorithme vorace choisisse un ensemble de tâches I et supposons que $J \neq I$. soit optimal.Coisiérons deux séquences réalisable S_I et S_J pour les dux ensembles en question. Il est clair qu'en faisant des transpositions appropriées dans S_I et S_J, nous pouvons construire deux séquences S'_I et S'_J telles que ttoute tâche commune à I et J soit exécutées au même moment dans les deux séquences quitte à à laisser des trous dans ces séquences. Donnez un exemple en cours !!!

Les séquences S'_I et S'_J sont différentes puisque $I \neq J$. Soit t un instant quelconque pour lequel les tâches prévues dans S'_I et S'_J sont dsitinctes.

1. Si S'_I prévoit une tâche a alors que S'_J ne prévoit rien (il y aun trou en cet endroit dans la squence S'_J et que la tâche a n'apparaît pas dans l'densemble J. On pourra alors ajouter a à J et le gain sera augmenté. Ce qui contredit l'optimalité de J.

2. Si S'_J prévoit une tâche b alors que S'_I ne prévoit rien. L'algorithme vorace aurait pu ajouter b à I. Donc cette situation est impossible.

3. La seule possibilité restant est donc S'_J prévoit une tâche a et S'_I prévoit une autre tâche b. Encore une fois, ceci implique que a n'apparaît pas dans J et que b n'apparaît pas dans I.

   - si $g_a > g_b$, on pourrait remplacer b par a dans J et l'améliorer: contradiction avec l'optimalité de J.
   - si $g_a < g_b$, l'algorithme vorace aurait choisi b avanr de considérer a puisque $(I - a) \cup b$ serait réalisable. c,est donc impossible puisqu'il ne l,a pas fait.
   - il ne reste qu'une seule possibilité $g_a = g_b$.

À chaque position dans les deux séquences, nous avons donc soit aucune tâche, soit la même tâche, soit deux tâches de gain identique. Nous en concluons que la valeur totale du gain dans l'ensemble I est égale à celle de l'ensemble optimal J. cela implique que I est aussi optimal.

Complexité de l'algorithme: Exercice.

2.3 Le problème du sac à dos

Soient un ensemble de n objets $N = \{1, \ldots, n\}$, et un sac à dos pouvant contenir un poids maximal de W. Chaque objet a un poids ω_i et un gain v_i. Le problème consiste à choisir un ensemble d'objets par N, au plus un de chaque, de telle manière que le gain soit maximisé sans dépasser la contenance W du sac. Il est clair que $\sum_{i=1}^{n} w_i x_i > W$, autrement, le problème sera évident à résoudre.

Dans la version que nous allons voir, on permet le choix d'une fraction d'un objet. Autremen dit, si x_i représente la fraction de L'objet i à choisir, alors nous avons le problème suivant:

$$\begin{aligned} &\max \sum_{i=1}^{n} x_i v_i \\ &\text{telle que} \\ &\quad \sum_{i=1}^{n} w_i x_i \leq W \\ &\quad 0 \leq x_i \leq 1 \end{aligned}$$

Il est clair que dans une solution optimale, la valeur W doit être atteinte autrement on peut toujours y ajouter une fraction d'un objet restant et augmneter ainsi la valeur du gain total. Par conséquent, dans une solution optimale, nous avons toujours $\sum_{i=1}^{n} w_i x_i = W$.

L'idée de l'algorothme vorace est de sélectionner les objets un à un, dans un certain ordre, et de prendre la plus grande fraction de l'obje choisi dans le sac. On arrête quand le sac est plein. L'agorithme est comme suit:

```
for(i = 1; i<= n,i++)
  x[i] = 0; //initialisation
poidscourant = 0;
while poidscourant < W {
  i = l'objet restant le plus prometteur
  if poidscourant + w_i ≤ W {
       x[i] = 1;
       poidscourant = poidscourant +w_i
  }
  else {
```

```
            x[i] = (W - poidscourant)/w_i; // prendre une fraction de l'objet i
            poidscourant = W;
          }
} // fin de l'algoorithme
```

Tel que décrit, il existe au moins trois manière plausibles de choisir le prochain obejt dans le sac. En effet, à chaque étape de l'algorithme, parmi les objets restant, nous pouvons choisir l'objet ayant le plus grand gain (car augmenatnt le plus le gain), l'obget ayant le plus petit poids (remplisant le sac le moins possible) ou alors, on peut éviter ces cas extrêmes en choisissant l'objet ayant le gain par unité de poids la plus grande possible. Pour se convaincre de la stratégie la plus profitable, considérons l'exemple suivant:

Exemple 4 $n = 5; W = 100$

v	20	30	66	40	60
w	10	20	36	40	50
v/w	2.0	1.5	2.2	1.0	1.2

Si on considère les objets dans l'ordre décroissant des gains, on choisit en premier l'objet 3, ensuite l'objet 5 et on complète le sac par la moitié de l'objet 4. Ce qui nous donne comme gain total de 66+ 60 +40/2 = 146. Si nous choisissons les objets dans l'ordre croissant des poids, alors on prend dans cet ordre les objets 1, 2, 3 et 4, et le sac est plein. La valeur du gain obtenue est 20+30+ 66 + 40 = 156. Maitenant, si nous choisissons les objets dans l'ordre croissant des v_i/w_i, alors on prend en premier l'objet 3, l'objet 1, ensuite l'objet 2 et finalement on complète le sac par le 4/5 de l'objet 5. cette stratégie génère le gain suivant: $20+30+66+0.8\times 60 = 164$.

Le résulat nous montre que'effectivement la dernières tarégie est optimale.

Theorem 3 *Si les objets sont choisis dans l'ordre décroissant des $v_i w_i$, alors l'algorthme vorace ci-dessus génère une solution optimale.*

Preuve: Sans perte de généralité, supposons que les objets sont renommés de telle manière

$$v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$$

Soit $X = \{x_1, x_2, \ldots, x_n\}$ une solution générée par l'algoithme ci-dessus. Si tous les x_i sont égaux à 1, alors clairement la solution est optimale. Sinon, soit j le plus petit indice tel que $x_j < 1$. En regardant de près le fonctionnement de l'algorithme, il clair que $x_i = 1$ pour $i < j$ et $x_i = 0$ pour tout $i > j$ (cela est clairement indiqué dans le else sinon de l'instruction de test où une fraction d'un objet est prise pour compléter la capacité du sac et l'aolgorithme s'arrête à la prochaine itération), et bien entendu nous avons $\sum_{i=1}^{n} x_i w_i = W$. Soit $V(X) = \sum_{i=1}^{n} x_i v_i$.

Considérons maintenant une autre réalisable soution $Y = \{y_1, y_2, \ldots, y_n\}$. Comme Y est réaslisable, nous avons bien $\sum_{i=1}^{n} y_i w_i \leq W$. Par conséquent, nous avons bien $\sum_{i=1}^{n}(x_i - y_i)w_i \geq 0$. Soit $V(Y) = \sum_{i=1}^{n} y_i v_i$.

$$V(X) - V(Y) = \sum_{i=1}^{n}(x_i - y_i)v_i = \sum_{i=1}^{n}(x_i - y_i)w_i\frac{v_i}{w_i}$$

Quand $i < j$, $x_i = 1$ et $x_i - y_i \geq 0$, tandis que $v_i/w_i \geq v_j/w_j$.

Si $i > j$, $x_i = 0$ et $x_i - y_i \leq 0$, tandis que $v_i/w_i \leq v_j/w_j$.

Et $i = j$, $v_i/w_i = v_j/w_j$.

Dans tous les cas, nous avons bien

$$(x_i - y_i)(v_i/w_i) \geq (x_i - y_i)(v_j/w_j)$$

Par conséquent,

$$V(X) - V(Y) \geq (v_j/w_j)\sum_{i=1}^{n}(x_i - y_i)w_i \geq 0$$

Ce qui revient à dire que quelque soit la solution réalisable, celle générée par l'agorithme vorace ci-dessus est meilleure qu'elle. D'où son optimalité.
Complexité de cet algorihme: Il est clair que la complexité temporelle de cet algorithme est dominée par la procédure de tri des tâches suivant l'ordre croissant des v_i/w_i qui peut être réalisée en $O(n \log n)$.

3 Problèmes de graphes

Considérons le problème, connu dans la théorie des graphes, la recherche de l'arbre sous-tendant de poids minimum (en anglais minimum spanning tree).

Soit un réseau comportant des machines A, B, C, D, et E qui doivent pouvoir communiquer entre elles. Les liaisons envisagées sont représentées par le graphe suivant (les arêtes sont étiquetées par la distance entre les machines):

	A	B	C	D	E
A	0	5	0	0	4
B	5	0	2	4	6
C	0	2	0	3	0
D	0	4	3	0	2
E	4	6	0	2	0

Question: Comment câbler le réseau à moindre coût ?

Réponse: Il s'agit d'enlever des arêtes au graphe de façon qu'il reste connexe, et que la somme des pondérations des arêtes soit la plus petite possible. Remarquons que le graphe partiel recherché est sans cycle (pourquoi ?)

Le problème est ramené au problème suivant :

Definition 2 *Le problème de l'arbre recouvrant de poids minimal est celui qui consiste à déterminer un arbre qui soit un graphe partiel d'un graphe G simple connexe et dont le poids total est minimal.*

Algorithme de Kruskal

La stratégie de cet algorithme est vorace. Elle consiste à construire l'arbre en question comme suit. On part d'une solution vide. On choisit, à chaque fois, une arête de G de poids minimum et qui ne crée pas de cycle. Soit E l'ensemble des sommets de G. On utilisera un ensemble d'arêtes T qui sera en sortie l'arbre couvrant minimal et un ensemble d'arêtes F qui représentera les arêtes qui peuvent être choisies.

T = ∅; F = E ;
tant que $|T| < n - 1$ faire
{
  trouver une arête e de F de poids minimal
  F = F - e
  si T + e est acyclique
   alors T = T + {e};
}

En appliquant l'algorithme ci-dessus, on obtient l'arbre sous-tendant ci-dessous:

poids	arête	
2	$B-C$	
2	$D-E$	
3	$C-D$	
4	$B-D$	éliminée
4	$A-E$	
5	$A-B$	éliminé
6	$B-E$	éliminé

Preuve d'optimalité: Soit n l'ordre du graphe. L'algorithme produit bien un arbre recouvrant du graphe puisqu'il termine lorsque les $(n-1)$ arêtes sont choisies et T est acyclique. Supposons maintenant que l'arbre recouvrant T ne soit pas minimal. Si $e_1, e_2, \ldots, e_{n-1}$ sont ces arêtes alors nous avons par construction $p(e_1) \leq p(e_2) \leq \cdots p(e_{n-1})$ par construction ($p(e_i)$ étant le poids de

l'arête e_i). Soit alors A un arbre couvrant minimal tel que l'indice de la première arête de T, qui ne soit pas une arête de A, soit maximum. Soit donc k cet indice. On a alors $e_1, e_2, \ldots, e_k \in T$, $e_1, e_2, \ldots, e_{k-1} \in A$ et e_k n'est pas dans A. Alors $A + e_k$ contient un unique cycle C. C ne peut pas être constitué uniquement d'arêtes de T (hormis ek) car sinon T contiendrait un cycle. Soit alors e une arête de C appartenant à A mais non à T. Alors $A+e_k-e$ donc un arbre couvrant du graphe, et $p(A+e_k-e) = p(A)+p(e_k)-p(e)$. Dans l'exécution de l'algorithme de Kruskal, l'arête e_k a été choisie de poids minimal telle que le graphe contenant le sous-graphe contenant les arêtes $e_1, e_2, \ldots, e_k$ soit acyclique. Mais le sous-graphe contenant les arêtes $e_1, e_2, \ldots, e_{k-1}, e$ est aussi acyclique, puisqu'il est sous-graphe de A. Par conséquent, $p(e) \geq p(e_k)$ et $p(A+e_k-e) \leq P(A)$. On en déduit que $(A + e_k - e)$ est optimal et diffère de T par une arête strictement supérieure à k: contradiction.

Complexité: Montrer que la complexité de cet algorihme est en $O(n \log n)$.

3.1 Problèmes de mariages stables

Soient un ensemble de n hommes et un ensemble de n femmes. Chaque homme et chaque femme liste ses candidats par ordre de préférence. L'objectif est de trouver un ensemble de mariages stables, en notant que chaque homme doit épouser une et une seule femme. De même que chaque femme doit épouser épouser un et seul seul homme.

Les mariages stables sont les mariages où il n'existe pas de couple se préférenant mutuellement à leur conjoint actuel.

À titre illustratif, considérons l'exemple suivant: Soit $M = \{a, b, c\}$ l'ensemble des hommes, et $F = \{X, Y, Z\}$, l'ensemble des femmes. Soient aussi les préférences des uns et autres résumées come suit:

Les préférences des hommes dans l'ordre décroissant

a	Y	X	Z
b	Z	X	Y
c	Z	Y	X

Les préférences des femmes dans l'ordre décroissant

X	a	b	c
Y	b	c	a
Z	a	c	b

L'ensemble de marriages suivant ne constitue pas un ensemble stable:

$$\{aZ, bX, cY\}$$

La raison est que le couple (a, X) se préfère mutuellement à leur conjoint. En revanche, l'ensemble suivant est stable:

$$\{aY, bX, cZ\}$$

L'algorithme glouton, pour trouver une solution à ce problème est comme suit:

- Chaque homme (femme) demande à chaque femme (homme) dans l'ordre de leur préférence de s'engager et arrête lorsqu'il obtient une réponse favorable.
- Le suivant fait ainsi de même et peut ainsi briser des engagements.
- Un homme dont l'engagement est brisé reprend là où il avait laissé.
- Lorqu'on a terminé, on sait que chaque homme a la femme qu'il préfère parmi celles qui n'avaient pas un homme qu'elles aiment mieux. Ainsi, dans chaque couple potentiel, il existe un des comparses qui préfère son conjoint à ceux qui auraient pu être disponibles.

L'algorithme vorace est donc comme suit:

While (il existe un homme m non marié) {
  m se propose à la femme f qu'il apprecié le mieux parmi celle qui restent
  if (f non mariée)
    marier m à f
  else if (f préfère m à son époux t {
    divorcer (f, t);
    marier f à m;
    t supprime f de sa liste;
    }
    else (f heureuse avec t)
      m supprime f de sa liste;
}

Illustration: Appliquons cet algorithme sur l'exemple ci-dessus, c'est-à-dire:

a	Y	X	Z
b	Z	X	Y
c	Z	Y	X

X	a	b	c
Y	b	c	a
Z	a	c	b

1. a se propose à $Y \longrightarrow \{aY\}$
2. b se propose à $Z \longrightarrow \{aY, bZ\}$
3. c se propose à Z, Z prefère $c \longrightarrow$ divorcer $bC \longrightarrow \{aY, cZ\}$
4. b se propose à $X \longrightarrow \{aY, bZ, bX\}$ = mariage stable.

Test:
a: satisfait; b prefere juste Z; c satifait.

X: prefere juste a; Y: prefère b et c; C prefère juste a.

Proposition 1 *L'algorithme ci-dessus s'arrête après un nombre fini d'étapes.*

Preuve: À chaque étape de l'agorithme, une femme est soit mariée soit supprimée de la liste d'un homme. De plus, une femme mariée le reste tout au long de l'algorithme.

Proposition 2 *L'algorithme ci-dessus génère un ensemble de mariages complet et stables.*

Preuve: Supposns qu'il reste une femme f non mariée. Alors il doit exister également un homme m non marié. Mais m s'est proposée à f.

Supposons qu'il existe un couple instable, c'est-à-dire un couple (m, f) qui se préfèrent plus que leur partenaire. Cela implique que m s'est proposé à f avant sa partenaire mais s'est vu refusé. Cela est impossible car une femme ne peut qu'améliorer sa préférence.