Table des Matières

Le problème des activités

La donnée du problème est un ensemble d'activités $a_1, \ldots, a_n$; pour chaque activité a_i on a une "heure" de début s_i et une "heure" de fin f_i ($s_i < f_i$). Le problème est d'affecter des salles à toutes les activités, en respectant bien sûr la contrainte : "à un instant donné, il peut y avoir au plus une seule activité par salle" et en minimisant le nombre de salles utilisées.
On suppose qu'il n'y a pas de temps de battement entre deux activités, *i.e.* que dans la même salle dès qu'une activité est terminée, une autre peut commencer.
On considère un problème simplifié.

Problème (Les activités)

Entrée : Un ensemble $\{a_1, \ldots, a_n\}$ de n activités, et des valeurs $s_i \leq f_i$ de début et de fin de chaque activité $a_i \in A$

Sortie : Un ensemble d'activités de taille maximale d'activités mutuellement *compatibles*$^{(*)}$

Définition

Les activités a et b sont *compatible* si $s_a \geq f_b$ (a commence après b), ou $s_b \leq f_a$ (b commence après a).

Exemple

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Le sous-ensemble $\{3, 9, 11\}$ est solution, mais il n'est pas optimal : $\{1, 4, 8, 11\}$ marche aussi ou encore $\{2, 4, 9, 11\}$.

Remarque

On remarque que dans l'exemple les activités sont classées par ordre croissant d'heure de fin – on le justifie juste après.

Un critère intuitif pour sélectionner les activités

1. Imaginons qu'une activité se terminant à l'heure f vient d'être choisie.
2. Nous proposons alors de choisir/sélectionner, parmi toutes les activités qui commencent après f (*i.e.* les compatibles), celle qui se termine le plus tôt.
3. Intuitivement, on maximise ainsi le temps pendant lequel on peut encore placer des activités.

Un algorithme glouton

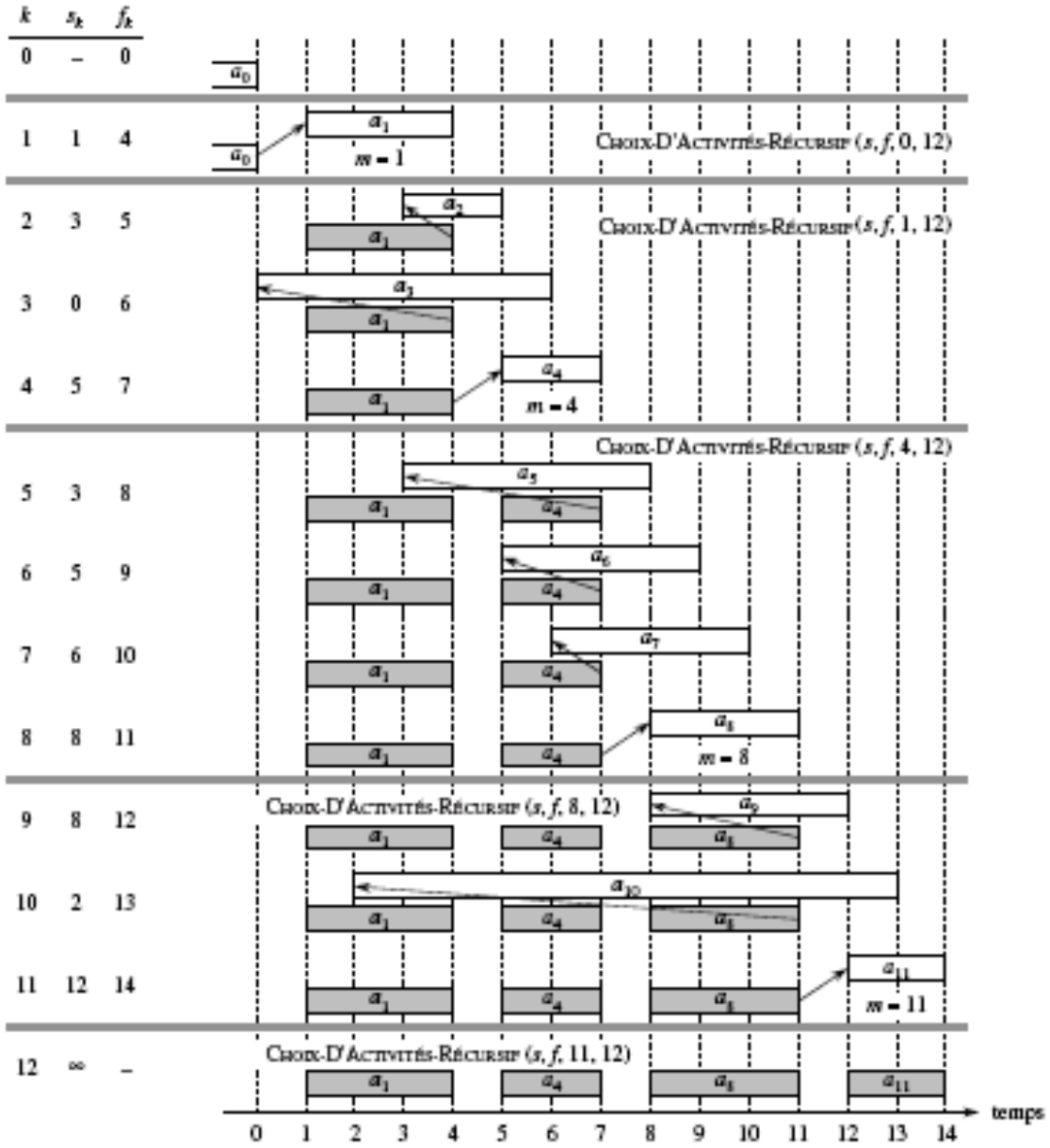
On supposera que :

- les activités sont numérotées $a_1, a_2, ..., a_n$
- on y ajoute 2 activités "fictives" a_0 et a_{n+1} telles que avec $s_0 = f_0 = 0$ et $s_{n+1} = f + n + 1 = \infty$.
- l'entrée de l'algorithme est deux tableaux $s[0..n+1]$ et $f[0..n+1]$ pour les heures de début et de fin de chaque activité
- le tableau d'heures de fin f est trié en ordre croissant, comme dans notre exemple.

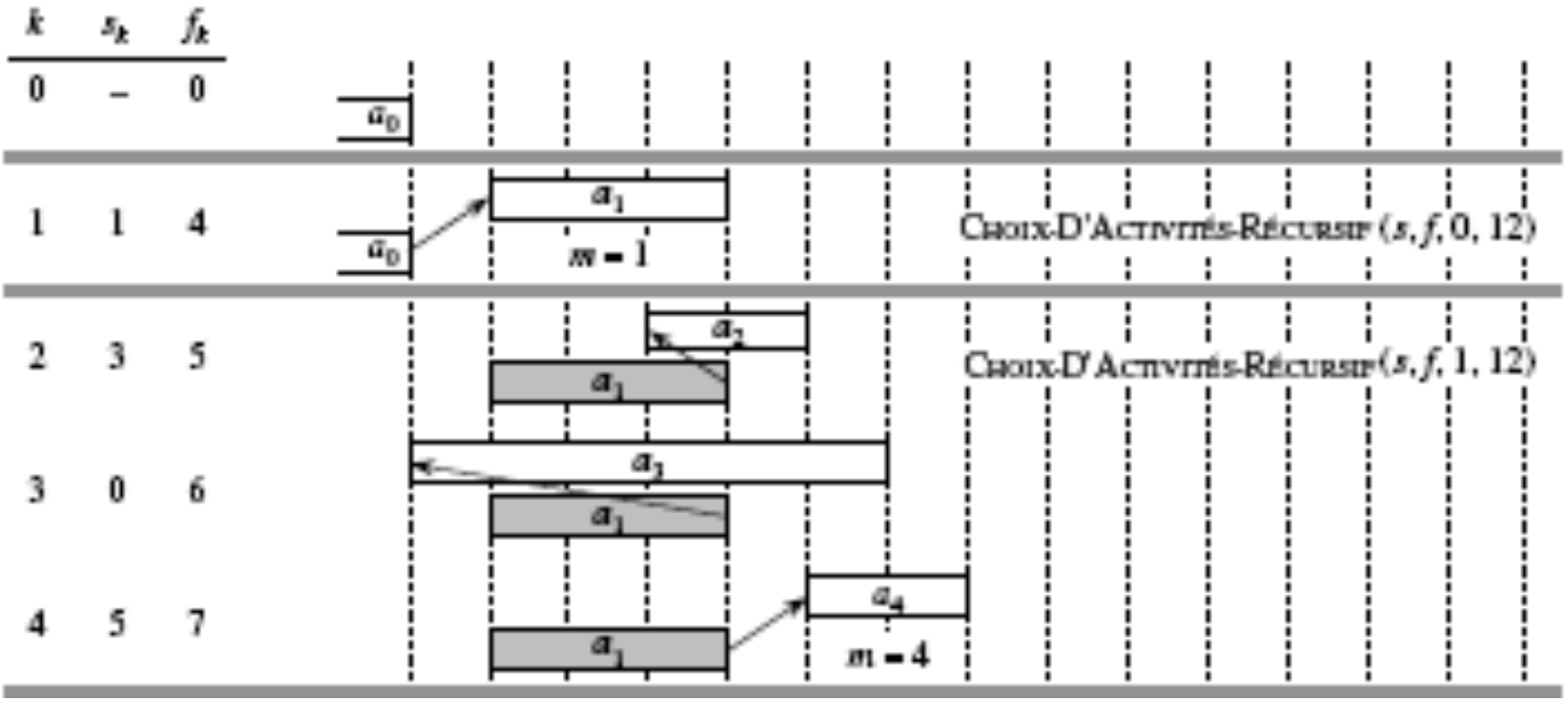
Algorithm 1 CHOIX-D'ACTIVITÉS-GLOUTON(s, f)

```
1:  n ← longueur[s]
2:  A ← {a_0}                                  ▷ on sélectionne a_0
3:  i ← 0                                      ▷ a_i est la dernière activité sélectionnée
4:  for m ← 1 to n + 1 do
5:      if s[m] ≥ f[i] then                    ▷ a_m se termine le plus tôt, parmi toutes les activités
6:          A ← A ∪ {a_m}                      ▷ compatibles avec a_i.
7:          i ← m                              ▷ a_i est la dernière activité sélectionnée
8:      end if
9:  end for
10: return A \ {a_0, a_{n+1}}
```

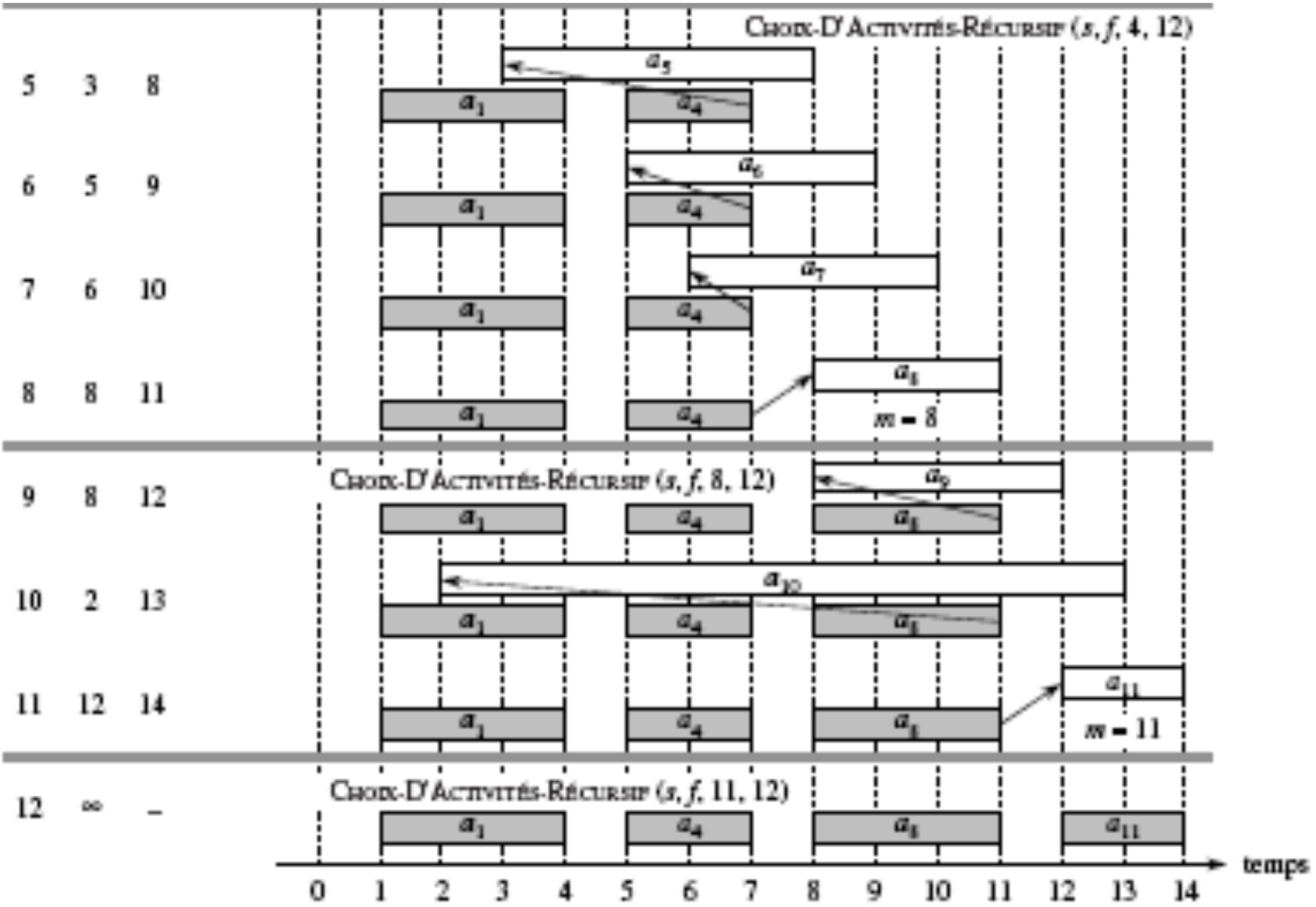
Exemple d'exécution



Exemple d'exécution



Exemple d'exécution



Correction de Choix-D'activités-Glouton(s, f)

Algorithm 2 Choix-D'activités-Glouton(s, f)

```
1:  n ← longueur[s]
2:  A ← {a_0}                                   ▷ on sélectionne a_0
3:  i ← 0                                       ▷ a_i est la dernière activité sélectionnée
4:  for m ← 1 to n + 1 do
5:      if s[m] ≥ f[i] then                     ▷ a_m se termine le plus tôt, parmi toutes les activités
6:          A ← A ∪ {a_m}                       ▷ compatibles avec a_i.
7:          i ← m                               ▷ a_i est la dernière activité sélectionnée
8:      end if
9:  end for
10: return A \ {a_0, a_{n+1}}
```

Théorème

L'algorithme Choix-D'activités-Glouton(s, f) *est optimal.*

Voir la preuve au tableau.

Principe des algorithmes gloutons

1. On transforme le problème (d'optimisation) en un problème dans lequel on fait un choix *glouton* de manière à se retrouver avec un seul sous-problème (d'optimisation) à résoudre.
2. On démontre qu'il existe toujours une solution optimale du problème initial qui justifie ce choix glouton.
3. On démontre qu'après ce choix glouton la résolution du sous-problème restant et ce choix glouton se combinent en une solution optimale au problème original.

Une version récursive de l'algorithme

Algorithm 3 CHOIX-D'ACTIVITÉS-GLOUTON-RECURSIF(s, f, i, n)

1: $m \leftarrow i$
2: **while** $m \leq n+1$ et $s[m] < f[i]$ **do** ▷ a_m n'est pas compatible avec a_i
3: $\quad m \leftarrow m+1$
4: **end while** ▷ $m \leq n+1$, car $s[n+1] = +\infty \geq f[i]$
5: **return** $\{a_m\} \cup$ CHOIX-D'ACTIVITÉS-RÉCURSIF(s, f, m, n)

Initialement, on appelle CHOIX-D'ACTIVITÉS-RÉCURSIF$(s, f, 0, n)$

Remarque

L'algorithme CHOIX-D'ACTIVITÉS-RÉCURSIF(s, f) est récursif *terminal*.

Exercice

Qu'est-ce qu'un algorithme récursif terminal ?

La satisfaisabilité des formules de Horn

Rappels

- Propositions : $p, q, r, \ldots$
- Clauses : $C := p \vee \neg q$, $C' := \neg r \vee \neg q$, $C'' := \neg r \vee p \vee q$

Définition

Une clause de Horn est une clause contenant au plus un litéral positif.
Les clauses de Horn sans littéraux positifs sont appelées négative pure et les autres des implication.

Exemple

$C := p \vee \neg q$ et $C' := \neg r \vee \neg q$ sont des clauses de Horn, mais pas $C'' := \neg r \vee p \vee q$.

La valuation ν telle que $\nu(p) = \nu(q) = \nu(r) = 0$ satisfait à lafois C et C'

Problème (Horn Clauses)

Entrée : Un ensemble fini de clauses de Horn

Sortie : Si elle existe, une valuation des variables propositionnelles de ces clauses qui rend toutes les clauses de l'ensemble vraies

Exemple d'une entrée de Horn Clauses

Exemple

On se donne des faits

- x pour "le meurtre a eu lieu dans la cuisine"
- y pour "le majordome est innocent"
- z pour " le colonnel dormait à 20h00"
- w pour "le meurtre a eu lieu à 20h00"
- u pour " le colonnel est innocent", v pour "

On pourra écrire des clauses telles que :

- $\rightarrow x$ (pour $vrai \rightarrow x$) exprime que "le meutre a bien eu lieu dans la cuisine"
- $(z \wedge w) \rightarrow u$
- $(\neg u \vee \neg v \vee \neg y)$ pour "Ils ne sont pas tous innocents"

On considère l'ensemble de clauses

$$(w \wedge y \wedge z) \rightarrow x, (x \wedge z) \rightarrow w, x \rightarrow y, \rightarrow x, (x \wedge y) \rightarrow w, (\neg w \vee \neg x \vee \neg y), (\neg z)$$

Un algorithme glouton pour les clauses de Horn

On s'inspire de [DPV06] "Algorithms" par Dasgupta/Papadimitriou/Vazirani

Algorithm 4 SAT-HORN-GLOUTON($\mathcal{C}$)

Soit *Prop* l'ensemble des propositions des clauses de $\mathcal{C}$
Soit $\mathcal{C}_I \subseteq \mathcal{C}$ l'ensemble des implications
Soit $\mathcal{C}_N \subseteq \mathcal{C}$ l'ensemble des négatives pures
for all $p \in Prop$ **do**
    $\nu(p) \leftarrow$ `false` ▷ Initialiser toutes les variables à faux
end for
while Il existe une implication $C \in \mathcal{C}_I$ avec $\nu(C) =$ `false` **do**
    Soit $C \in \mathcal{C}_I$ de la forme $C = \ldots \rightarrow p$
    $\nu(p) \leftarrow$ `true` ▷ mettre la variable droite de implication C à vrai
end while
if pour chaque $C \in \mathcal{C}_N$, $\nu(C) =$ `true` **then**
    return la valuation ν
else
    return "la formule n'est pas satisfaisable"
end if

Correction de l'algorithme Sat-Horn-Glouton

Algorithm 5 Sat-Horn-Glouton($\mathcal{C}$)

1: Soit *Prop* l'ensemble des propositions des clauses de $\mathcal{C}$
2: Soit $\mathcal{C}_I \subseteq \mathcal{C}$ l'ensemble des implications
3: Soit $\mathcal{C}_N \subseteq \mathcal{C}$ l'ensemble des négatives pures
4: **for all** $p \in Prop$ **do**
5: $\quad \nu(p) \leftarrow$ `false` ▷ Initialiser toutes les variables à faux
6: **end for**
7: **while** Il existe une implication $C \in \mathcal{C}_I$ avec $\nu(C) =$ `false` **do**
8: $\quad$ Soit $C \in \mathcal{C}_I$ de la forme $C = \ldots \rightarrow p$
9: $\quad \nu(p) \leftarrow$ `true` ▷ mettre la variable droite de implication C à vrai
10: **end while**
11: **if** pour chaque $C \in \mathcal{C}_N$, $\nu(C) =$ `true` **then**
12: $\quad$ **return** la valuation ν
13: **else**
14: $\quad$ **return** "la formule n'est pas satisfaisable"
15: **end if**

- Si l'algorithme retourne une valuation, cette valuation satisfait à la fois les implications et les clauses négatives pures, c'est donc bien une solution.

Correction de l'algorithme SAT-HORN-GLOUTON

- Pour le cas où l'algorithme répond "la formule n'est pas satisfaisable", on établit un invariant.

Algorithm 6 SAT-HORN-GLOUTON($\mathcal{C}$)

1: Soit *Prop* l'ensemble des propositions des clauses de $\mathcal{C}$
2: Soit $\mathcal{C}_I \subseteq \mathcal{C}$ l'ensemble des implications
3: Soit $\mathcal{C}_N \subseteq \mathcal{C}$ l'ensemble des négatives pures
4: **for all** $p \in Prop$ **do**
5: $\quad \nu(p) \leftarrow$ `false` ▷ Initialiser toutes les variables à faux
6: **end for**
7: **while** Il existe une implication $C \in \mathcal{C}_I$ avec $\nu(C) =$ `false` **do**
8: $\quad$ Soit $C \in \mathcal{C}_I$ de la forme $C = \ldots \rightarrow p$
9: $\quad \nu(p) \leftarrow$ `true` ▷ mettre la variable droite de implication C à vrai
10: **end while**
11: **if** pour chaque $C \in \mathcal{C}_N$, $\nu(C) =$ `true` **then**
12: $\quad$ **return** la valuation ν
13: **else**
14: $\quad$ **return** "la formule n'est pas satisfaisable"
15: **end if**

Lemme

La propriété "Si une variable est mise à vrai alors elle vaut vrai dans toute valuation solution" est un invariant de la boucle **While** *en ligne 7 de l'agorithme* SAT-HORN-GLOUTON.

Exercice

Établir cet invariant.

Donc si la valuation trouvée à la ligne 10 ne satisfait pas une clause négative pure, il ne peut y avoir de valuation qui satisfait l'ensemble des clauses.

Intérêt pratique des clauses de Horn

Remarque

Les clauses de Horn sont au coeur de Prolog ("programming by logic"), un langage dans lequel on programme les propriétés attendues en utilisant des expression logiques simples.

Le cheval de bataille des interpreteurs Prolog est notre algorithme glouton, qui a une complexité linéaire en la taille de l'entrée

Exercice

Informez-vous un minimum sur Prolog.

Codage de Huffman

Problème (Codage de Huffman)

Entrée : Un texte dont on connait l'alphabet fini des symboles, et pour chaque symbole son nombre d'occurrence dans le texte

Sortie : Un encodage qui minimise la longueur du code du texte

Impact du choix d'un codage

Exemple

On veut stocker un texte (une chaine de caratères) qui contient les symboles "A,B,C,D".
On peut coder selon

Lettre	Encodage
A	00
B	01
C	10
D	11

Pour un texte tel que

Lettre	Nbre d'occurrences
A	70 millions
B	3millions
C	20millions
D	37millions

On aura besoin de $(70 + 3 + 20 + 31) \times 2$ millions de bits pour stocker.

C'est volumineux !

Tous les copdages ne marchent pas !

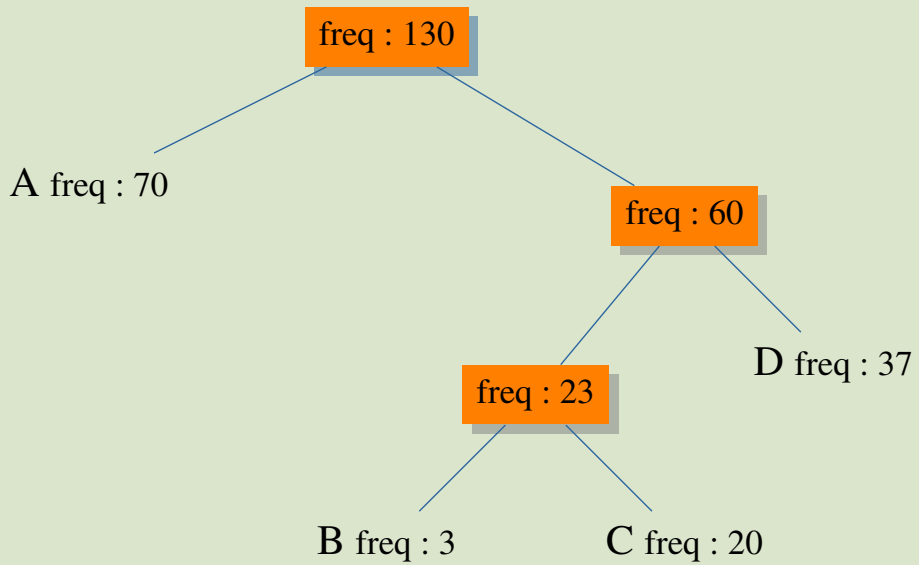
Exemple

Lettre	Encodage
A	0
B	001
C	01
D	11

Comment décoder 001 ?

Notion d'arbre préfixe et codage induit

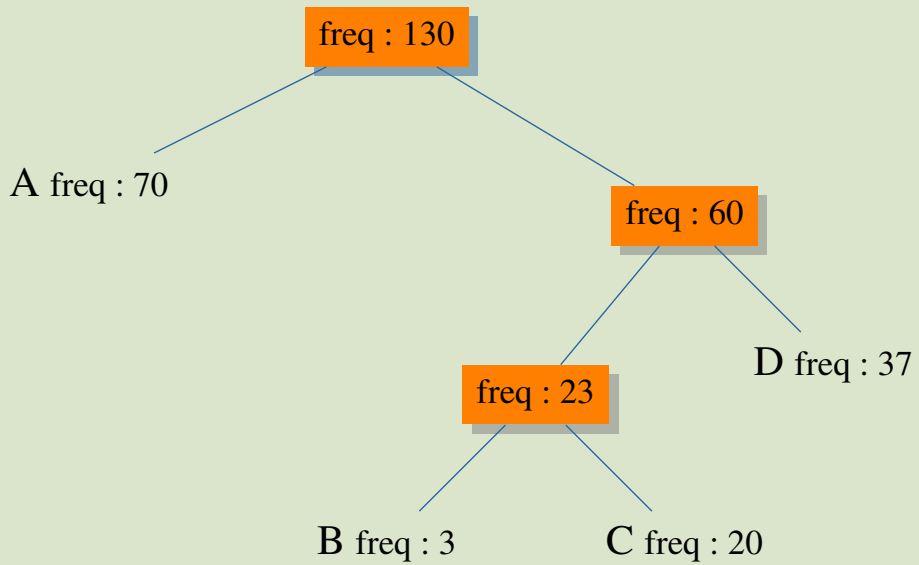
Exemple



lettres	Codage
A	0
B	100
C	101
D	11

Notion d'arbre préfixe et codage induit

Exemple



lettres	Codage
A	0
B	100
C	101
D	11
meta-lettres	**Codage**
{B,C}	10
{B,C,D}	1

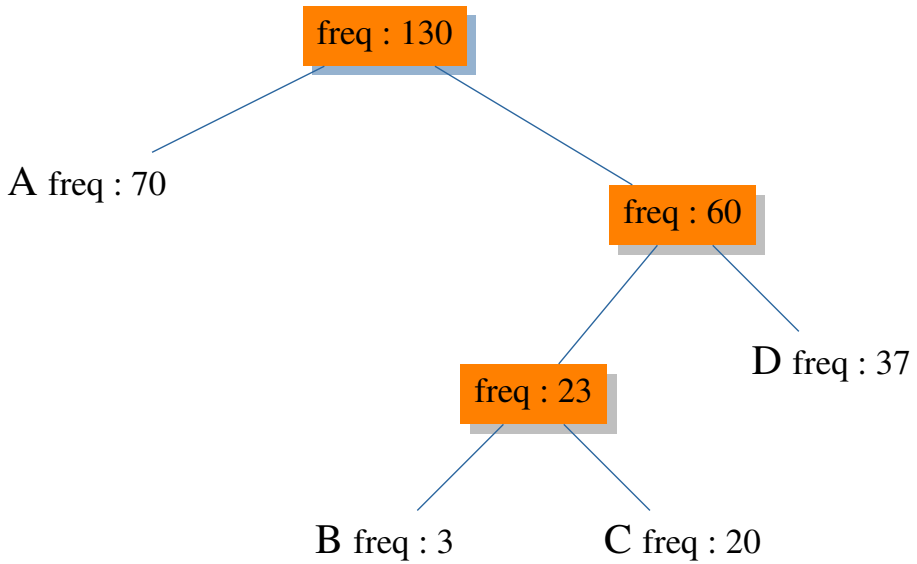
Exercice

Définir formellement la notion d'arbre préfixe.

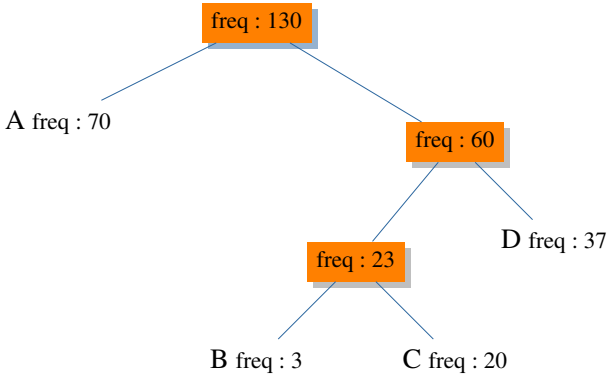
Principe pour la construction de l'arbre préfixe

La construction d'un arbre préfixe fonctionne de la manière suivante :

- On commence avec une forêt d'arbres triviaux à un noeud pour chaque lettre ℓ de l'alphabet $\mathcal{A}$ du texte.
- Au cours de l'algorithme, on récupère les deux arbres A_1 (de fréquence f_1) et A_2 (de fréquence f_2) de fréquences minimales et on les colle ensemble pour obtenir un arbre A (de fréquence $f_1 + f_2$).
- L'algorithme s'arrête lorsque la forêt ne contient plus qu'un seul arbre : c'est l'arbre préfixe que l'on utilisera pour coder les lettres du texte.



Coût $c(T)$ d'un arbre préfixe T



Lettre	Encodage	Profondeur dans l'arbre
A	0	1
B	100	3
C	101	3
D	11	2

La profondeur d'une lettre est la longueur de son code. Le codage est directement issu de la construction de l'arbre préfixe, et son efficacité est donnée par :

Définition

Soit $\mathcal{A}$ l'alphabet du texte.

Le coût d'un arbre préfixe T est $c(T) = \begin{cases} freq[\ell] & \text{si } \mathcal{A} = \{\ell\} \\ \sum_{\ell \in \mathcal{A}} freq[\ell] \times p_\ell & \text{sinon} \end{cases}$

où p_ℓ est la profondeur de la lettre ℓ dans T (*i.e.* la longueur de son code).

Exemple

Pour l'arbre préfixe T en haut, on a $c(T) = 70 + 3 \times 3 + 20 \times 3 + 37 \times 2$.

Algorithme de codage

On utilise une *file de priorité* $\mathcal{F}$ (pour "forêt") qui contient des arbres ordonnés par la fréquence de leur racine.
Initialement, on part de la forêt $\mathcal{F}_0$ formée des arbres préfixes *simples* à un noeud correspondant à une lettre ℓ de l'alphabet et de fréquence la fréquence de cette lettre dans le texte. Alors $Card(\mathcal{F}_0)$ est le cardinal de l'alphabet.

On appelle alors $Huffman(\mathcal{F}_0)$, où :

Algorithm 7 Fonction $Huffman(\mathcal{F})$

1: **if** $Card(\mathcal{F}) = 1$ **then return** l'unique arbre de $\mathcal{F}$
2: **else**
3: $T_1 = \mathcal{F}.defilerMin$
4: $T_2 := \mathcal{F}.defilerMin$
5: $T := T_1 \oplus T_2$ ▷ l'arbre dont le fils gauche est T_1 et le fils droit est T_2
6: $freq[T] := freq[T_1] + freq[T_2]$
7: $\mathcal{F}.enfiler(T)$
8: **return** $Huffman(\mathcal{F})$
9: **end if**

Terminaison de l'algorithme $Huffman(\mathcal{F})$

Algorithm 8 Fonction $Huffman(\mathcal{F})$

1: **if** $Card(\mathcal{F}) = 1$ **then return** l'unique arbre de $\mathcal{F}$
2: **else**
3: $T_1 = \mathcal{F}.defilerMin$
4: $T_2 := \mathcal{F}.defilerMin$
5: $T := T_1 \oplus T_2$ ▷ l'arbre dont le fils gauche est T_1 et le fils droit est T_2
6: $freq[T] := freq[T_1] + freq[T_2]$
7: $\mathcal{F}.enfiler(T)$
8: **return** $Huffman(\mathcal{F})$
9: **end if**

Exercice

Rédiger l'argumentaire proprement.

Correction de l'algorithme *Huffman*($\mathcal{F}$)

Théorème

L'algorithme Huffman($\mathcal{F}$) retourne un arbre A dont le coût $c(A)$ est minimal parmi tous les arbres préfixes.

Nous verrons cette preuve en TD.

Le problème du Sac à dos (fractionnaire)/Knapsack (Fractional) Problem

Problème (Knapsack Problem (KP))

Entrée : n objets, pour chaque objet i deux valeurs w_i (≥ 0 son poids) et p_i (son profit/sa valeur), une constante $K \geq 0$

Sortie : Des valeurs $x_1, x_2, \ldots, x_n$ dans $\{0, 1\}$ telles que $\Sigma_{i=1}^{n} w_i.x_i \leq K$ et qui maximise le profit $\Sigma_{i=1}^{n} p_i.x_i$.

Exercice

On cherche à maximiser $20.x_1 + 16.x_2 + 11.x_3 + 9.x_4 + 7.x_5 + x_6$
avec la contrainte $9.x_1 + 8.x_2 + 6.x_3 + 5.x_4 + 4.x_5 + x_6 \leq 12$
et que les x_is valent soit 0 soit 1.

Combien y a-t-il d'objets ? Quels sont les poids et valeurs de ces objets ?

Une variante de (KP)

Problème (Knapsack Fractional Problem (kfp))

Entrée : n objets, pour chaque objet i deux valeurs w_i (≥ 0 son poids) et p_i (son profit/sa valeur), une constante $K \geq 0$

Sortie : Une valuation dans $[0, 1]$ des variables $x_1, x_2, \ldots, x_n$ telle que $\Sigma_{i=1}^{n} w_i.x_i \leq K$ et qui maximise le profit $\Sigma_{i=1}^{n} p_i.x_i$.

Définition

On dit que (KFP) est une relaxation de (KP), car les variables peuvent prendre des valeurs non entières.

Une approche naturelle pour résoudre ce (KFP) :

> *Le voleur prend d'abord tout l'or, puis tout l'argent et finit de remplir son sac avec le bronze qui peut encore y entrer. Il ne lui viendrait pas à l'idée de prendre moins d'or pour le remplacer par quelque chose de valeur moindre.*

Un algorithme glouton pour Knapsack Fractional Problem

Quitte à les trier, on suppose que les objets $1, \ldots, n$ sont numérotés de telle sorte que

$$p_1/w_1 \geq p_2/w_2 \geq \ldots p_n/w_n$$

c'est à dire, selon le meilleur rapport profit/poids.

On appelle $GreedyKFP(1, K)$ qui est basé sur notre approche "naturelle", où :

Algorithm 9 $GreedyKFP(i, k)$

1: **if** $w_i \leq k$ **then**
2: $\quad x_i \leftarrow 1$
3: $\quad GreedyKFP(i+1, k-w_i)$ ▷ Appel du sous-problème
4: **else**
5: $\quad x_i \leftarrow k/w_i$ ▷ On prend la proportion qui finit de remplir le sac
6: $\quad x[i+1 \ldots n] \leftarrow 0$
7: **end if**

Correction de l'algorithme "GreedyKFP"

Algorithm 10 *GreedyKFP*(i, k)

1: **if** $w_i \leq k$ **then**
2: $\quad x_i \leftarrow 1$
3: $\quad$ *GreedyKFP*$(i+1, k-w_i)$ ▷ Appel du sous-problème
4: **else**
5: $\quad x_i \leftarrow k/w_i$ ▷ On prend la proportion qui finit de remplir le sac
6: $\quad x[i+1 \ldots n] \leftarrow 0$
7: **end if**

On établit que :

1. notre premier choix glouton x_1 est inclus dans une solution optimale,
2. une solution optimale commençant par x_1 contient une solution optimale du sous-problème KFP pour la constante $K - x_1 w_1$ et les objets $\{2, \ldots, n\}$.

Le choix glouton x_1 est inclus dans une solution optimale

Soit $O = (y_1, y_2, \ldots, y_n)$ une solution optimale du problème KFP, et soit $G = (x_1, x_2, \ldots, x_n)$ une solution de $GreedyKFP(1, K)$.

- Clairement $x_1 \geq y_1$ car le choix glouton sélectionne une quantité maximum de l'objet de valeur maximum.
- Si $y_1 = x_1$, alors on a fait dans G le même premier choix que pour la solution optimale O.
- sinon $(x_1 > y_1)$, mais alors le poids de la solution O est :

$$w(O) = \sum_{i=1}^{n} y_i w_i \leq K$$

mais aussi

$$w(O) = x_1 w_1 + \sum_{i=1}^{n} y_i w_i - (x_1 - y_1) w_1$$

On modifie alors O en la solution O' obtenue en prélevant un poids de $(x_1 - y_1)w_1$ en partant des objets les moins précieux, que l'on rajoute en quantité de l'objet 1.

$O' = (x_1, z_2, \ldots, z_n)$ est une solution puisque $w(O') = w(O) \leq K$, et elle est aussi bonne que O, i.e., $p(O') \geq p(O)$, puisque l'objet 1 est plus précieux que les autres.

Si $O' = (x_1, z_2, \ldots, z_n)$ est une solution optimale de KFP alors $(z_2, \ldots, z_n)$ est une solution optimale pour le sous-problème sur les objets $\{2, \ldots, n\}$ avec la borne $K - x_1 w_1$.

On rappelle que O' est obtenue en prélevant un poids de $(x_1 - y_1)w_1$ en partant des objets les moins précieux, que l'on rajoute en quantité de l'objet 1.

On note $KFP(1)$ le sous-problème sur les les objets $\{2, \ldots, n\}$ avec la borne $K - x_1 w_1$, et on notera naturellement $KFP(0)$ notre problème initial.

- $(z_2, \ldots, z_n)$ est une solution de $KFP(1)$ car

$$w((z_2, \ldots, z_n)) = w(O') - x_1 w_1 \leq K - x_1 w_1$$

- Supposons $(z_2, \ldots, z_n)$ non-optimale. Alors il existe $O'' = (t_2, \ldots, t_n)$ une solution de $KFP(1)$ telle que $p(O'') > p((z_2, \ldots, z_n))$, mais alors la solution $(x_1, t_2, \ldots, t_n)$ est une solution de $KFP(0)$ telle que $p((x_1, t_2, \ldots, t_n)) > p(O')$, ce qui contredit l'optimalité de O'.

En récapitulant pour KFP

- Après le choix glouton x_1 pour $KFP(0)$, on est ramené à résoudre optimalement le sous-problème problème $KFP(1)$ analogue au problème d'origine ;
- pour $KFP(1)$ il existe une solution optimale qui fait le choix x_2 et on doit ensuite résoudre le sous-problème $KFP(2)$, celui sur les objets $\{3, \ldots, n\}$ avec la borne $K - x_1 w_1 - x_2 w_2$;
- et ainsi de suite, prouvant ainsi que la solution gloutonne est elle-même optimale.

De manière générale, l'optimalité d'un glouton s'établit en montrant :

1. qu'à chaque étape, il existe une solution optimale qui contient notre choix glouton, c'est la propriété du "choix glouton".
2. que toute solution optimale repose sur des solutions optimales de sous-problèmes, c'est la propriété de "sous-structure optimale".

Version itérative de l'algorithme glouton pour *KFP*

Algorithm 11 *Greedy-KFP2*(w, p, K) // w et p sont des tableaux réels $[1..n]$

```
1:  variable local c
2:  c ← K
3:  for i = 1 to n do
4:      if w_i ≤ c then
5:          x_i ← 1
6:          c ← c − w_i
7:      else
8:          x_i ← c/w_i
9:          c ← 0
10:     end if
11: end for
```

Knapsack Problem non fractionnaire

On se pose naturellement la question d'exploiter le principe de *Greedy-KFP2*(w, p, K) pour résoudre le problem *KP*, et en particulier en ne gardant que les items "pleins".
On obtient alors l'algorithme :

Algorithm 12 *Greedy-KP*(w, p, K) // w et p sont des tableaux réels $[1..n]$

Require: variable locale c

1: $c \leftarrow K$
2: **for** $i = 1$ to n **do**
3: **if** $w_i \leq c$ **then**
4: $c \leftarrow c - w_i$
5: $x_i \leftarrow 1$
6: **else**
7: $x_i \leftarrow 0$ ▷ au lieu de $x_i \leftarrow c/w_i$
8: $c \leftarrow 0$
9: **end if**
10: **end for**

Exercice

L'algorithme *Greedy-KP*(w, p, K) est-il correct, c-à-d. retourne-t-il une solution optimale ?

Greedy-KP(w, p, K) n'est pas correct

Exemple

On cherche à maximmiser $20.x_1 + 16.x_2 + 11.x_3 + 9.x_4 + 7.x_5 + x_6$

avec les contraintes $\begin{cases} 9.x_1 + 8.x_2 + 6.x_3 + 5.x_4 + 4.x_5 + x_6 \leq 12 \\ x_i \in \{0, 1\} \text{ pour } i = 1, \ldots, 6\} \end{cases}$

En appliquant *Greedy-KP*$([9, 8, 6, 5, 4, 1], [20, 16, 11, 9, 7, 1], 12)$, on obtient successivement :

0. $c = 12$
1. $x_1 = 1$ et $c = 3$
2. $x_2 = 0$ et $c = 3$, puis ... puis $x_5 = 0$ et $c = 3$
3. $x_6 = 1$ et $c = 2$.

À la fin de l'algorithme $c = 2$, le sac n'est pas plein, et la solution $(1, 0, 0, 0, 0, 1)$ calculée à la valeur 21.

Or il existe une meilleure solution : la valeur de $(0, 1, 0, 0, 1, 0)$ et $z = 23$ (et le sac est plein).