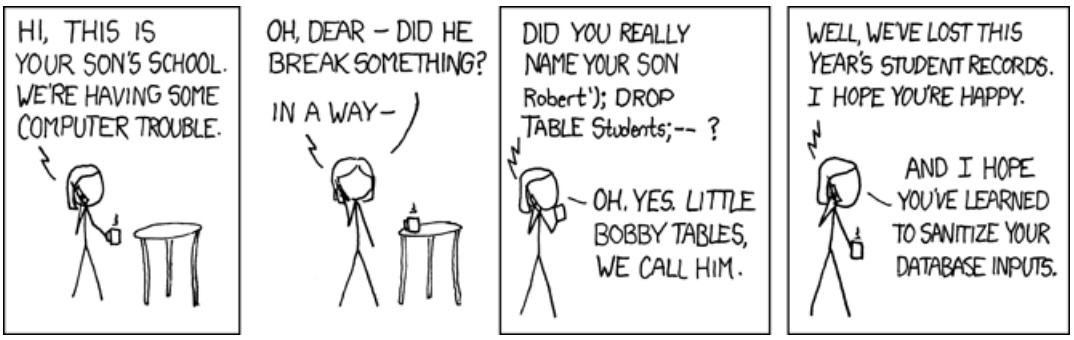
MPI* Info

Programme de khôlles

Édition n°3 (Partie 2)



Olivier Caffier

Table des matières

5 Donner un algo qui fournit une 2 approximation de MinVertexCover et prouver sa correction.

On rappelle la définition suivante :

Définition

Soient $G = (S, A)$ un graphe non-orienté.
Un *couplage* de G est un ensemble $C \subset A$ tel que $\forall (a, b) \in C^2, a \cap b = \emptyset$
Il est dit *maximal* si pour toute arête $a \notin C, C \cup \{a\}$ n'est pas un couplage.

On remarque deux choses :

1. **Un couplage maximal fournit effectivement une couverture par sommets :**
   Soit C un couplage maximal. On dénote l'ensemble de ses sommets :

   $$V = \bigsqcup_{\{x,y\} \in C} \{x, y\}$$

   Ainsi, si V n'était pas une couverture par sommets de S, alors il existerait une arête $a = \{x', y'\} \in A$ telle que $x', y' \notin V$. On peut donc la rajouter au couplage, qui en resterait un car ses sommets n'apparaissent pas dans la liste des sommets de C. Ce qui contredirait la maximalité de C.

   V est donc bien une couverture par sommets.
2. **Disposer d'un couplage maximal nous permet de fournir une 2-approx de notre problème :**
   En effet, si on prend C un couplage à k arêtes, alors toute couverture par sommets doit être de taille au moins k (les sommets de chaque arête dans le couplage étant différents à chaque fois, on doit au moins être en contact avec l'un des deux).
   Finalement, si on dispose d'un couplage maximal C de cardinal p, alors l'ensemble V construit précédemment est de taille $2p$, alors que la taille minimale d'une couverture par sommets sera $\geq p$.

   D'où la 2-approximation.

Avec notre méthode donc, il suffit donc juste de chercher un couplage maximal à travers G :

Algorithme - À la recherche d'un couplage maximal

$C \leftarrow \emptyset$
$V \leftarrow \emptyset$
while C n'est pas maximal, i.e il existe une arête $a = \{x, y\}$ telle que $C \cup \{a\}$ est un couplage **do**
    $C \leftarrow C \cup \{a\}$
    $V \leftarrow V \cup \{x, y\}$
return V

et on vient de prouver ci-dessus la correction de cette algorithme !

7 Savoir appliquer sur un exemple les algos suivants : LZW, Huffman, Rabin-Karp

7.1 Huffman

Définition - Code préfixe

Un code $c : \Sigma \to \{0;1\}^*$ est dit *préfixe* ssi :
$$\forall (a,b) \in \Sigma^2,\ c(a) \text{ n'est pas un préfixe de } c(b) \text{ et } c(b) \text{ n'est pas un préfixe de } c(a).$$

Problème

Soit $occ : \Sigma \to \mathbb{N}$, on veut construire $c : \Sigma \to \{0;1\}^*$ un code préfixe qui minimise
$$\sum_{a \in \Sigma} occ(a) \times \text{longueur}(c(a))$$

Algorithme de Huffman

Soit $m \in \Sigma^*$, on note $\text{occ}(a)$ le nombre d'occurrence de a dans m pour tout $a \in \Sigma$. On procède ensuite de la manière suivante :

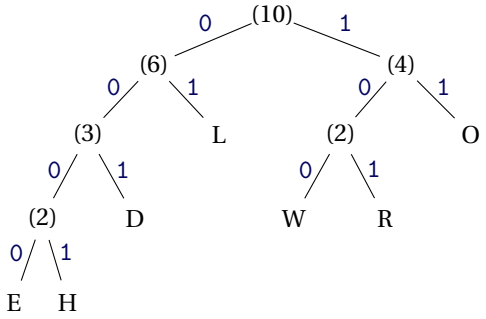
1. On construit la table d'occurrence, associant a et $\text{occ}(a)$ pour chaque lettre apparaissant dans m.
2. On initialise une forêt pondérée avec des arbres réduits à une feuille pour chaque elt a de Σ associé à $occ(a)$.
3. À chaque étape, on fusionne les deux arbres de poids minimal et on insère le résultat dans la forêt avec comme poids la somme des poids des 2 arbres fusionnés.
4. Quand la forêt est réduite à un seul arbre, alors on le renvoie. On en déduit alors un code binaire de manière assez classique : aller à gauche nous fait ajouter un 0, aller à droite nous fait ajouter un 1.

Un exemple vaut mieux qu'un long discours, compressons le mot "HELLOWORLD" :

1. On construit la table d'occurrence :

a	E	O	H	W	R	L	D
$\text{occ}(a)$	1	2	1	1	1	3	1

2. On construit l'arbre de Huffman. On dispose d'abord de arbres correspondant à des noeuds de chaque lettre, associés à leur poids. On fusionne ensuite les deux arbres avec les poids les plus faibles, le nouveau poids de cet arbre étant la somme des deux arbres.
3. On continue jusqu'à ce qu'il ne reste qu'un seul arbre :



4. On construit la table de compression à l'aide d'un parcours en profondeur :

E	H	D	L	W	R	O
0000	0001	001	01	100	101	11

5. On écrit dans le fichier le code de l'arbre, et en lisant le fichier texte lettre par lettre, on obtient le code suivant :

   `0001 0000 01 01 11 100 11 101 01 001`

   On comble le reste avec des 0 (en codant sur 8 bits par exemple), et on obtient : `00010000 01011110 01110101 00100000`

Code texte	00010000	01011110	01110101	00100000
Nombre associé	16	94	117	32
Code de l'arbre	00001E1H1D1L001W1R1O			

➢ Le code de l'arbre correspond au parcours préfixe de l'arbre, en notant `0` chaque noeud et `1X` chaque feuille (avec `X` la lettre en question). Ce code permet donc de représenter l'arbre de manière unique. D'où l'encodage recherché.

7.2 Lempel-Ziv-Welch

Intérêt de cette nouvelle approche :

- On compresse au fur et à mesure de la lecture sans faire de pré-traitement du texte
- On ne compresse pas lettre à lettre mais on va associer un code à des facteurs du texte

Principe de la compression Initialement, on dispose d'une table qui associe à chaque lettre de l'alphabet un code et au cours de la lecture on va ajouter des mots dans la table.

0	…	65	66	…	255	256
		A	B	…	ÿ	case libre

Algorithme - LZW

$m \leftarrow$ 1ère lettre du texte
while lecture du texte **do**
  $x \leftarrow$ lettre lue
  if mx est dans la table **then**
    $m \leftarrow mx$
  else
    On incrit dans le fichier compressé le code de m
    On insère mx dans la table
    $m \leftarrow x$
On inscrit dans le fichier compressé le code de m le mot restant.

EXEMPLE.

Prenons le mot `ABBBABBAABBA`, on effectue alors les instructions suivantes :

- **Boucle 1 :**

A	B	B	B	A	B	B	A	A	B	B	A
	↑										

$\begin{cases} m = \texttt{A} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{AB}$ n'apparait pas dans notre table!

$$\Rightarrow \begin{cases} \texttt{add(AB,256)} \\ \texttt{code}(m) = 65 \\ m \leftarrow \texttt{B} \end{cases}$$

- **Boucle 2 :**

A	B	B	B	A	B	B	A	A	B	B	A
		↑									

$\begin{cases} m = \texttt{B} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{BB}$ n'apparait pas dans notre table!

$$\Rightarrow \begin{cases} \texttt{add(BB,257)} \\ \texttt{code}(m) = 66 \\ m \leftarrow \texttt{B} \end{cases}$$

- **Boucle 3 :**

A B | B | B A B B A A B B A

↑

$\begin{cases} m = \texttt{B} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{BB}$ apparait dans notre table !

$$\Rightarrow \{ m \leftarrow \texttt{BB}$$

- **Boucle 4 :**

A B | B B | A B B A A B B A

↑

$\begin{cases} m = \texttt{BB} \\ x = \texttt{A} \end{cases}$ et $mx = \texttt{BBA}$ n'apparait pas dans notre table !

$$\Rightarrow \begin{cases} \texttt{add(BBA,258)} \\ \texttt{code(}m\texttt{) = 257} \\ m \leftarrow \texttt{A} \end{cases}$$

- **Boucle 5 :**

A B B B | A | B B A A B B A

↑

$\begin{cases} m = \texttt{A} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{AB}$ apparait dans notre table !

$$\Rightarrow \{ m \leftarrow \texttt{AB}$$

- **Boucle 6 :**

A B B B | A B | B A A B B A

↑

$\begin{cases} m = \texttt{AB} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{ABB}$ n'apparait pas dans notre table !

$$\Rightarrow \begin{cases} \texttt{add(ABB,259)} \\ \texttt{code(}m\texttt{) = 256} \\ m \leftarrow \texttt{B} \end{cases}$$

- **Boucle 7 :**

A B B B A B | B | A A B B A

↑

$\begin{cases} m = \texttt{B} \\ x = \texttt{A} \end{cases}$ et $mx = \texttt{BA}$ n'apparait pas dans notre table !

$$\Rightarrow \begin{cases} \texttt{add(BA,260)} \\ \texttt{code(}m\texttt{) = 66} \\ m \leftarrow \texttt{A} \end{cases}$$

- **Boucle 8 :**

A B B B A B B | A | A B B A
↑

$\begin{cases} m = \texttt{A} \\ x = \texttt{A} \end{cases}$ et $mx = \texttt{AA}$ n'apparait pas dans notre table!

$$\Rightarrow \begin{cases} \texttt{add(AA,261)} \\ \texttt{code}(m) = 65 \\ m \leftarrow \texttt{A} \end{cases}$$

- **Boucle 9 :**

A B B B A B B A | A | B B A
↑

$\begin{cases} m = \texttt{A} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{AB}$ apparait dans notre table!

$$\Rightarrow \{m \leftarrow \texttt{AB}$$

- **Boucle 10 :**

A B B B A B B A | A B | B A
↑

$\begin{cases} m = \texttt{AB} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{ABB}$ apparait dans notre table!

$$\Rightarrow \{m \leftarrow \texttt{ABB}$$

- **Boucle 11 :**

A B B B A B B A | A B B | A
↑

$\begin{cases} m = \texttt{ABB} \\ x = \texttt{A} \end{cases}$ et $mx = \texttt{ABBA}$ n'apparait pas dans notre table!

$$\Rightarrow \begin{cases} \texttt{add(ABBA,262)} \\ \texttt{code}(m) = 259 \\ m \leftarrow \texttt{A} \end{cases}$$

- **Fin de la boucle `while`**

$$\{\texttt{code}(m) = 65$$

Donc, en reprenant les étapes balisées en rouge, on obtient le code suivant :

65 66 257 256 66 65 259 65

La conversion en binaire est laissée au lecteur...

Bon, on va pas se mentir, c'est fastidieux... À l'oral on présenterait plus comme ceci :

m	**code obtenu**	code ajouté dans la table
A	**65**	256 : AB
B	**66**	257 : BB
B		258 : BBA
BB	**257**	259 : ABB
	256	260 : BA
A		261 : AA
B	**66**	262 : ABBA
...	**65**	
...	**259**	
...	**65**	

et à l'écrit? Il y a des questions qu'il ne vaut mieux pas poser...

Algorithme LZW - Décompression

On a le pseudo-code suivant, en notant C le code obtenu et `t` notre table de base, notre "dictionnaire" (donc qui va de 0 à 255) :

```
n ← |C|
v ← lire C
m ← t[v]
Afficher m
for i ← 1 to n − 1 do
    v ← lire C
    if v est déjà une clé t then
        mot_tmp ← t[v]
    else
        mot_tmp ← Concaténer(m,m[0])
    Afficher mot_tmp
    Ajouter(Concaténer(m,mot_tmp[0]))
    m ← mot_tmp
```

Le mot d'ordre est donc le suivant :

On simule la compression

Reprenons notre code C = 65 66 257 256 66 65 259 65

- $n = 8$
- v ← 65
- m ← A
- mot_vide ← ε

$$\Rightarrow \{\texttt{Afficher(m) = A}$$

- **Entrée dans la boucle :**
  1. ∘ v ← 66
     ∘ v est déjà une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \texttt{t[66] = B} \\ \texttt{Afficher(mot\_tmp) = B} \\ \texttt{Add(Concaténer(A,B),256) = Add(AB,256)} \\ \texttt{m} \leftarrow \texttt{B} \end{cases}$$

  2. ∘ v ← 257
     ∘ v n'est pas une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \text{Concaténer(B,B)} = \text{BB} \\ \texttt{Afficher(mot\_tmp) = BB} \\ \texttt{Add(Concaténer(B,B),257) = Add(BB,257)} \\ \texttt{m} \leftarrow \text{BB} \end{cases}$$

  3. ∘ v ← 256
     ∘ v est déjà une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \texttt{t[256] = AB} \\ \texttt{Afficher(mot\_tmp) = AB} \\ \texttt{Add(Concaténer(BB,A),258) = Add(BBA,258)} \\ \texttt{m} \leftarrow \texttt{AB} \end{cases}$$

  4. ∘ v ← 66
     ∘ v est déjà une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \texttt{t[66] = B} \\ \texttt{Afficher(mot\_tmp) = B} \\ \texttt{Add(Concaténer(AB,B),259) = Add(ABB,259)} \\ \texttt{m} \leftarrow \texttt{B} \end{cases}$$

5. ◦ v ← 65
   ◦ v est déjà une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \texttt{t[65] = A} \\ \texttt{Afficher(mot\_tmp) = A} \\ \texttt{Add(Concaténer(B,A),260) = Add(BA,260)} \\ \texttt{m} \leftarrow \texttt{A} \end{cases}$$

6. ◦ v ← 259
   ◦ v est déjà une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \texttt{t[259] = ABB} \\ \texttt{Afficher(mot\_tmp) = ABB} \\ \texttt{Add(Concaténer(A,A),261) = Add(AA,261)} \\ \texttt{m} \leftarrow \texttt{ABB} \end{cases}$$

7. ◦ v ← 65
   ◦ v est déjà une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \texttt{t[65] = A} \\ \texttt{Afficher(mot\_tmp) = A} \\ \texttt{Add(Concaténer(ABB,A),262) = Add(ABBA,262)} \\ \texttt{m} \leftarrow \texttt{A} \end{cases}$$

- **Fin de la boucle** - on a donc trouvé le mot (en mettant un espace à chaque différence d'affichage)

`A B BB AB B A ABB A`

qui est bien le mot qu'on avait encodé au début de ce paragraphe :)

7.3 Rabin-Karp

Le plus coûteux dans une recherche textuelle, c'est la comparaison du motif avec un facteur du texte. Pour éviter ces comparaisons, l'algorithme de Rabin-Karp consiste à calculer une **empreinte** (un *hash*) du motif et à la comparer avec l'empreinte du facteur (de même taille que le motif) à la position actuelle.

Une empreinte est le résultat d'une **fonction de hachage**. C'est donc un entier : on peut comparer deux empreintes en temps $\mathcal{O}(1)$.

- Si les deux empreintes sont différentes : on est certain que le facteur étudié ne correspond pas au motif, on passe donc à l'indice suivant.
- Sinon : il faut comparer manuellement les caractères du motif et du facteur de texte et si ils sont tous égaux, on a alors trouvé une occurence du motif et sinon, c'est qu'il s'agissait d'une **collision** avec le hash du motif, on passe à l'indice suivant.

EXEMPLE. Prenons l'empreinte f suivante :

$$f : \texttt{t[k,...,k+|m|-1]} \mapsto \text{nombre de } \texttt{i} \text{ dans } \texttt{t[k,...,k+|m|-1]}$$

et considérons cet extrait de « À une passante » de Charles Baudelaire :

« Un éclair … puis la nuit! Fugitive beauté »

et recherchons le motif `m = nuit` qui possède un seul `i` :

U	n		é	c	l	a	i	r	…	p	u	i	s		l	a		n	u	i	t

Ici, rien ne sert de lancer le scan vu que la partie rouge ne dispose pas du bon nombre de `i`, on passe donc à l'indice suivant :

U	n		é	c	l	a	i	r	…	p	u	i	s		l	a		n	u	i	t

De même ici, rien ne sert de lancer le scan vu que la partie rouge ne dispose pas du bon nombre de `i`, on passe donc à l'indice suivant etc, jusqu'à arriver à la situation suivante :

U	n		é	c	l	a	i	r	…	p	u	i	s		l	a		n	u	i	t

Ici, on a le bon nombre de `i` mais le `t` de `m` ne colle pas au `i` :

U	n		é	c	l	a	i	r	…	p	u	i	s		l	a		n	u	i	t
				n	u	i	t														

On peut donc décaler etc, jusqu'à arriver à :

U	n		é	c	l	a	i	r	…	p	u	i	s		l	a		n	u	i	t
																	n	u	i	t	

On décale donc d'un, on garde le bon nombre de `i` donc on peut lancer le scan et on trouve :

U	n		é	c	l	a	i	r	…	p	u	i	s		l	a		n	u	i	t
																		n	u	i	t

Remarques importantes :

➢ On remarque déjà **l'importance d'avoir un bon critère de différenciation** pour l'empreinte, choisir le nombre `i` (alors que c'est une des voyelles les plus utilisées, 6.59% du corpus de Wikipedia quand même) n'est peut-être pas le choix le plus optimal!

➢ En plus de cela, **il faudrait que le pré-traitement se fasse en** $\mathcal{O}(1)$ pour éviter d'avoir une complexité catastrophique en calculant à chaque fois l'empreinte et en scannant par la suite (on serait en $\mathcal{O}(|t| \times (|m| + f_{|m|}))$ en fait). Il faut donc essayer de trouver un moyen de **calculer l'empreinte avec l'empreinte précédente** (surtout qu'on ne fait que des sauts de 1 donc on voit très bien une formule de récurrence arriver).

Pour récapituler :

Ce que doit respecter notre empreinte

Une empreinte utilisée par notre algorithme doit à tout prix :

- Avoir peu de collisions (donc avoir un bon critère de différenciation).
- Le calcul de l'empreinte du motif doit se faire en $\mathcal{O}(|m|)$.
- On doit pouvoir calculer l'empreinte du facteur en position `i+1` **en temps constant** à partir de l'empreinte du facteur en position `i`, on parle alors d'**empreinte tournant ou glissante** (*rolling hash* en anglais).

Reprenons

$$f : t[k,\dots,k+|m|-1] \mapsto \text{nombre de } i \text{ dans } t[k,\dots,k+|m|-1]$$

Pour calculer $f(t[k+1,\dots,k+|m|])$, on a juste besoin de savoir si `t[k]` était un `i`, de même pour `t[k + |m|]`, et de l'empreinte $f(t[k,\dots,k+|m|-1])$. On revient à la formule de récurrence suivante :

$$\begin{cases} f(t[0,\dots,|m|-1]) = \text{nombre de } i \text{ dans } t[0,\dots,|m|-1] \\ \forall k \text{ tq. } 0 < k < |t|-|m|,\ f(t[k+1,\dots,k+|m|]) = f(t[k,\dots,k+|m|-1]) - \delta_{(t[k]==i)} + \delta_{(t[k+|m|]==i)} \end{cases}$$

Encore une fois, compter le nombre de `i` n'est pas la meilleure des techniques, on va donc utiliser une empreinte spécifique très connue : **l'empreinte de Rabin**.

Définition - Empreinte de Rabin

Soit p un nombre premier (on le prend premier pour justement limiter les collisions).
On considère que le texte et le motif sont passés en paramètres dans le code ASCII, qui va nous permettre d'utiliser directement des opérations algébriques.
On définit alors l'empreinte de Rabin f de la manière suivante :

$$\begin{cases} f(x_0 x_1 \dots x_{|m|}) = x_0 \times p^{|m|-1} + x_1 \times p^{|m|-2} + \dots x_{|m|-1} \\ \forall k \in [\![0; |t|-|m|]\!],\ f(x_{k+1} \dots x_{k+|m|}) = \left(f(x_k \dots x_{k+|m|}) - x_k \times p^{|m|-1}\right) \times p + x_{k+|m|} \end{cases}$$

➢ L'algorithme de Horner pourra effectivement s'avérer utile pour le calcul du cas initial. On remarque qu'on a besoin de garder en mémoire uniquement :

- $p^{|m|-1}$ et p.
- La valeur de l'empreinte précédente.
- La valeur de l'empreinte du motif.

8 Expliquer la différence entre `WHERE` et `HAVING`

`WHERE` et `HAVING` - Meilleurs ennemis ?

- Le `WHERE` agit avant le `GROUP BY`
- Une fonction d'agrégation (typiquement `COUNT()`) ne peut pas agir tant que le partitionnement par `GROUP BY` n'est pas fait. Donc l'attribut temporaire associé n'est pas disponible pour le `WHERE` (cela provoque l'erreur "`Result : misuse of aggregate: COUNT()`".
- `HAVING` sert à filtrer les groupes générés par `GROUP BY`
- Retenons la chronologie
  - `WHERE` agit avant le `GROUP BY`
  - `HAVING` agit après

➢ Si l'on veut avoir une phrase facile à retenir :

« `GROUP BY` groupe les données en paquets et `HAVING` filtre ces groupes. »

EXEMPLE.

Si l'on prend cette base de donnée très simple :

`personnes`

id	nom	prénom	age
1	Beauval	Élise	18
2	Martin	Pierre	35
3	Galois	Évariste	25
4	Martin	Filomena	20

et que l'on veut avoir la réponse à la question « Quelle sont les familles comportant plus de 2 membres ? », on groupe d'abord par nom de famille :

```
SELECT nom
FROM personnes
GROUP BY nom ;
```

Cela va donc nous modéliser un résultat de ce type :

nom	...
Beauval	...
Martin	...
Galois	...

et donc pour répondre à notre question, on *filtre* les groupes avec `HAVING` !

```
SELECT nom
FROM personnes
GROUP BY nom
HAVING COUNT(*) >=2 ;
```

9 Maîtriser l'utilisation du `GROUP BY` et son lien avec les fonctions d'agrégation.

cf. ci-dessus, en gardant à l'esprit que les fonctions d'agrégation en SQL sont les suivantes :

- `AVG()` : calcule la moyenne d'un enregistrement.
- `COUNT()` : compte le nombre d'enregistrement sur une table ou une colonne distincte.
- `MAX()` : pour récupérer la valeur du max... Cela s'applique à la fois pour des données numériques ou alphanumériques !
- `MIN()` : ...
- `SUM()` : pour calculer la somme sur un ensemble d'enregistrement.

Pour plus d'exemples, je vous renvoie vers le poly disponible sur CahierDePrépa, très détaillé, au sujet du langage SQL.

Olivier Caffier - MPI* 2023-2025