



M1-I2D

PROJET ALGORITHMIQUES & PROGRAMMATION
AVANCÉE


Problemes DUDC & UEP

Auteurs:

Benamara ABDELKADER
CHIHAB

Djelid AYMEN

January 2, 2021

Notre implémentation sur ([Github](#)) 

Exercice 1: (DUDC)

Input : Soit un ensemble \mathcal{P} de n points et un ensemble \mathcal{Q} de m cercles dans un espace \mathbb{R}^d avec $d \in \{1, 2\}$. Et un entier $k \geq 1$

Output : Un sous ensemble $\mathcal{Q}^* \subseteq \mathcal{Q}$ de taille au plus k qui couvre tout les points de \mathcal{P} .

I) Résoudre le probleme pour $d=1$

1) Algorithme glouton qui résout DUDC

Choix Glouton : Prendre le cercle le plus à droite qui contient le point courant (les points sont triés de gauche à droite ainsi que les cercles).

NB: Dans le cas ou on trouve un point qui n'est pas contenu dans aucun cercle (voir la condition $c == \text{None}$ dans l'algorithme) on renvoie \emptyset).

Algorithme 1 : DUDC- 1D – glouton

```
1 Input : Soit un ensemble  $\mathcal{P}$  de  $n$  points dans  $\mathbb{R}$  et un ensemble  $\mathcal{Q}$  de  $m$ 
   cercles dans un espace  $\mathbb{R}$  du type :  $\mathcal{Q} = \{[a_1, b_1], \dots, [a_m, b_m]\}$ .
2 Output : Un sous ensemble  $\mathcal{Q}^* \subseteq \mathcal{Q}$  de taille minimum qui couvre tout
   les points de  $\mathcal{P}$ .
3 begin
4   points_restants = les points de  $\mathcal{P}$  triés de gauche à droite.;
5   cercles = les cercles de  $\mathcal{Q}$  triés de gauche à droite. ;
6    $\mathcal{Q}^* = \emptyset$  .
7   tant que ( points_restants  $\neq \emptyset$  ) faire
8     Point  $p$  = le plus à gauche des points_restants;
9     Cercle  $c$  = le plus à droite des cercles qui contient  $p$ ;
10    si (  $c == \text{None}$  ) alors
11      return  $\emptyset$ 
12    fin
13     $\mathcal{Q}^* = \mathcal{Q}^* \cup \{c\}$ 
14    points_restants = points_restants \ points couverts par  $c$ 
15  fin
16  return  $\mathcal{Q}^*$ 
17 end
```

La complexité : Les deux tris on peut les faire en $O(n \log(n))$ et $O(m \log(m))$ et pour la boucle au pire de cas on peut parcourir en $O(n^2)$. Donc la complexité de l'algo est en $O(n^2)$.

2) Optimalité de l'algorithme DUDC

Soit $(\mathcal{P}, \mathcal{Q})$ une instance de DUDC en dimension 1. On distingue deux cas:

1. Si il n'existe pas de solution (pas de sous ensemble de cercles qui couvrent \mathcal{P} et ce cas est bien traité dans l'algorithme ci-dessus.

2. Supposons qu'une solution existe pour cette instance de DUDC i.e tout les points de \mathcal{P} sont couverts par un sous ensemble $O \subseteq \mathcal{Q}$ tel que O soit une solution optimale dans ce cas là. Et soit Q^* la solution retournée par notre algorithme.

Supposons que O est l'ensembles des cercles de la solution optimale triés de gauche à droite on peut obtenir notre Q^* par la procédure suivante :

- (a) Pour chaque cercle c dans O on va prendre le point le plus à gauche qui est contenu dans c notant le p
- (b) Appliquons notre choix glouton . i.e : on cherche le cercle $c' \in Q$ le plus à droite qui contient p . Si $c' = c$ on change rien dans O . Sinon on remplace c par c' dans O .

A la fin de cette procédure O reste une solution valide (car elle couvre tout les points de \mathcal{P}) et aussi $|Q^*| = |O|$ car à chaque étape de cette procédure soit on garde le cercle de O soit on le remplace par exactement un seul cercle. Comme O reste valide et à la fin de la procédure elle sera égale à Q^* .

Conclusion : L'algorithme proposé est optimale.

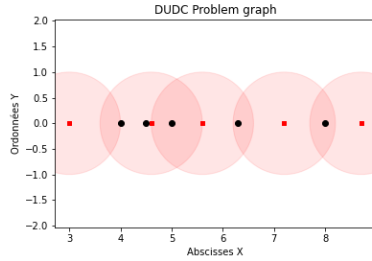


Figure 1: Probleme DUDC sur \mathbb{R}

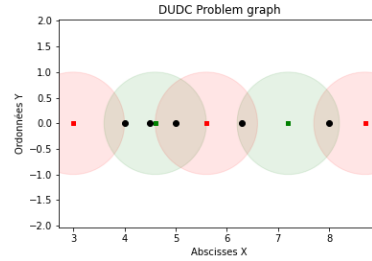


Figure 2: Solution du probleme

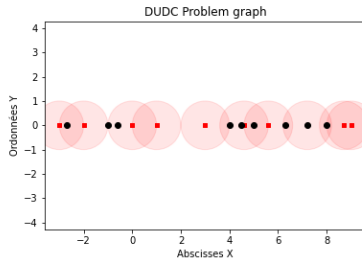


Figure 3: Probleme DUDC 2 sur \mathbb{R}

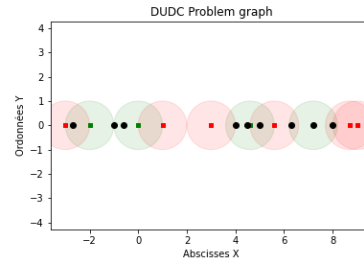


Figure 4: Solution du probleme 2

II) Le probleme DUDC pour d=2

1) Montrons que DUDC est NP-Difficile

On considere la réduction $SetCover \leq_T DUDC$ afin de montrer que DUDC est NP-difficile puisque on sait déjà que Set Cover est NP-complet.

Set Cover : Qui pour une instance $(\mathcal{U}, \mathcal{S})$ tels que $\mathcal{U} = \{u_1, \dots, u_n\}$ est une collection de n éléments et $\mathcal{S} = \{S_1, \dots, S_m\}$ est une famille de m sous-ensembles de \mathcal{U} . Et on a $\mathcal{S}^* \subseteq \mathcal{S}$ de taille au plus $k' \geq 1$ qui couvre \mathcal{U} .

Soit $\mathcal{I}=(\mathcal{U}, \mathcal{S})$ une instance de Set-Cover on construit $(\mathcal{P}, \mathcal{Q})$ une instance de DUDC de cette maniere :

1. $\mathcal{P} = \mathcal{U}$ est l'ensemble des points du plan.
2. \mathcal{Q} est l'ensemble des cercles C_i dans \mathbb{R}^2 tels que chaque $S_i \in \mathcal{S}$ représente un cercle C_i de \mathcal{Q} .
Et pour un point p_j de \mathcal{P} , On a $p_j \in C_i$ ssi $u_j \in S_i$ avec $u_j \in \mathcal{U}$. Et on pose $k = k'$.

Montrons que cette réduction est correcte :

\Rightarrow Supposons pour l'instance $\mathcal{I} = (\mathcal{U}, \mathcal{S})$ qui vérifie Set-Cover et soit $\mathcal{S}^* \subseteq \mathcal{S}$ qui couvre \mathcal{U} et $|\mathcal{S}^*| = k'$ d'après notre construction on obtient une instance $\mathcal{I}' = (\mathcal{P}, \mathcal{Q})$ qui a pour solution $\mathcal{Q}^* \subseteq \mathcal{Q}$, \mathcal{Q}^* qui peut être trouvé directement à partir de \mathcal{S}^* puisque chaque S_i dans \mathcal{S}^* qui couvre des éléments u_j de \mathcal{U} représente un cercle C_i dans \mathcal{Q}^* qui contient des points p_j de \mathcal{P} et que $|\mathcal{Q}^*| = |\mathcal{S}^*|$.

\Leftarrow Inversement soit \mathcal{Q}^* une solution de taille k pour l'instance $\mathcal{I} = (\mathcal{P}, \mathcal{Q})$ de DUDC. Tels que $\mathcal{Q}^* = \{C_i, \dots, C_j\}$ dans ce cas d'après notre construction. \mathcal{S}^* la solution de Set-Cover. Tel que $\mathcal{S}^* = \{S_i, \dots, S_j\}$ peut être trouvée directement à partir de \mathcal{Q}^* comme chaque S_i est représenté par un cercle C_i et les deux couvrent le même ensemble de points . Et qu'on a aussi $|\mathcal{S}^*| = |\mathcal{Q}^*| = k$ on a bien \mathcal{S}^* est bien une solution de Set-Cover.

2) Algorithme Branch & Bound qui résout DUDC

Pour l'algorithme Branch & Bound, nous considérons la liste des cercles triée par ordre décroissant de nombre des points couverts. Chaque nœud de l'arbre sera représenté par un couple (LB, UB) des bornes définies par :

1. **LB** : la borne inférieure qui représente une estimation de cercles qu'il faut prendre pour couvrir tout les points. (voir l'algorithme)
2. **UB** : la taille de la solution qui correspond au nœud actuel.

Chaque nœud de l'arbre de l'espace d'états se composera d'un ensemble de cercles pris et d'un ensemble de cercles explorés.

Dans le nœud racine (c'est-à-dire le nœud prometteur initial), ces deux ensembles sont vides au départ.

L'ensemble des cercles explorés est l'ensemble des cercles qu'on a déjà du passer par mais sans les considérer dans notre solution.

Algorithme 2 : DUDC Branch & Bound

```
1 Input : Une liste de taille m des cercles C restants qui n'ont pas été
   pris ni exploré triée par ordre décroissant de nombre de points couverts
   pour chaque c élément de C , et un ensemble des points restants P.
2 Output : La borne inférieure (LB) pour estimer le nombre minimum
   de cercles à prendre pour couvrir tous les points à partir d'un noeud
   donné de l'arbre.
3 begin
4   nb_points = |P| ;
5   LB = nombre des cercles déjà pris;
6   i = 0;
7   tant que (nb_points > 0)  $\wedge$  (i  $\leq$  m) faire
8     Cercle c = C[i];
9     nb_points = np_points - (# points couverts par c);
10    LB ++ ; i++;
11  fin
12  si ( nb_points > 0 ) alors
13    return + $\infty$ 
14  fin
15  return LB
16 end
```

Méthode de Branchement & parcours :

Pour un noeud : un cercle $c \in Q$ a deux possibilités soit $c \in Q^*$ soit $c \notin Q^*$ (soit on le prend ou pas)

Un noeud est considéré comme prometteur si sa (LB) n'est pas $+\infty$ et si elle n'est pas supérieure à la limite d'une solution déjà trouvée. Jusqu'à ce qu'il ne reste plus de noeud prometteur, nous considérons le noeud le plus prometteur, c'est-à-dire le noeud avec la borne **LB** la plus basse.

Nous effectuons un branchement. On considère le premier cercle c dans la liste des cercles qui n'a pas été pris ni déjà exploré (s'il n'y a pas de cercle, cela signifie que le problème n'admet pas de solution). Nous créons ensuite deux noeuds enfants à partir du noeud actuel, le noeud enfant gauche ayant c dans son ensemble de cercles pris, et le noeud enfant droit ayant c dans son ensemble de cercles déjà explorés (on prend pas le cercle c dans sa liste des cercles pris).

La solution optimale est trouvée lorsqu'au moins une solution a été trouvée et qu'il n'y a pas noeud prometteur à gauche. S'il existe plusieurs solutions, la solution optimale est celle avec le le plus petit nombre de cercles pris.

Voici l'exemple donné sur le sujet pour illustrer l'algorithme de branch & bound

.

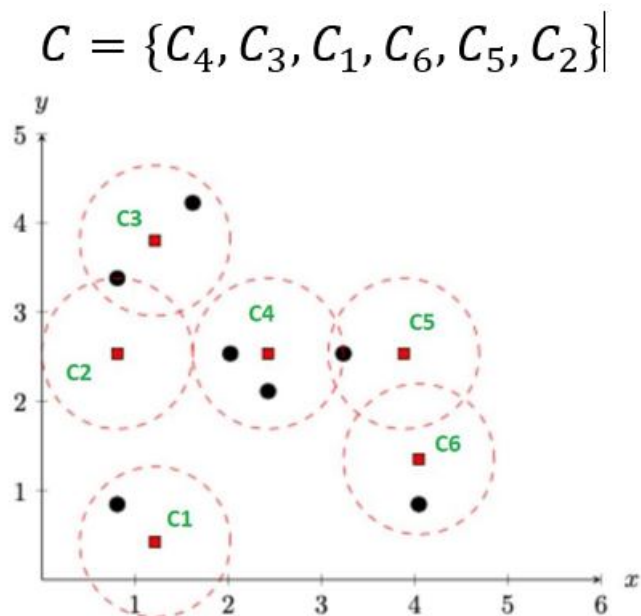


Figure 5: Exemple donné sur le sujet

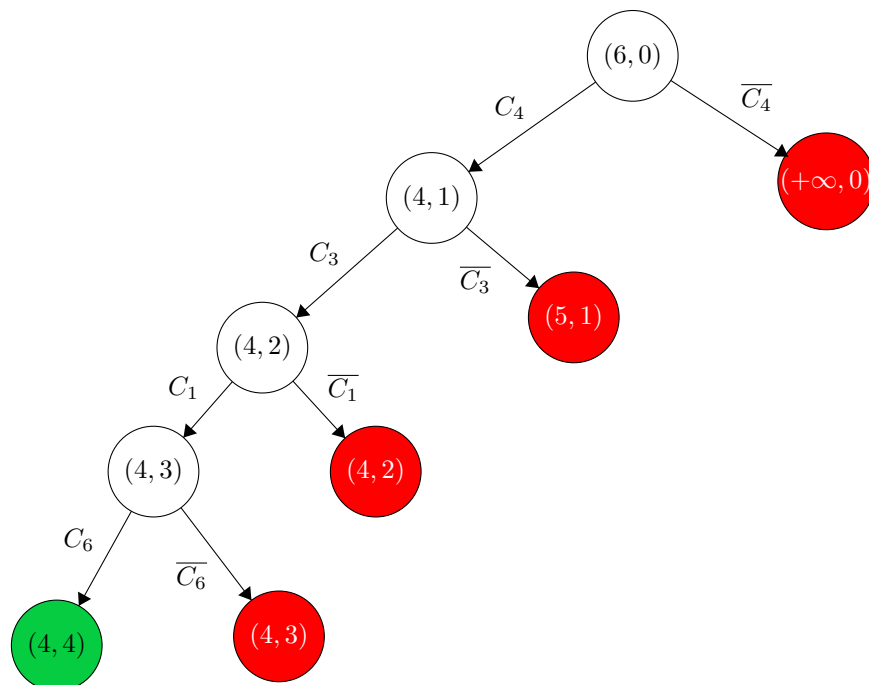


Figure 6: Parcours de l'algorithme B & B sur l'exemple

Exemples d'implémentation de la solution DUDC

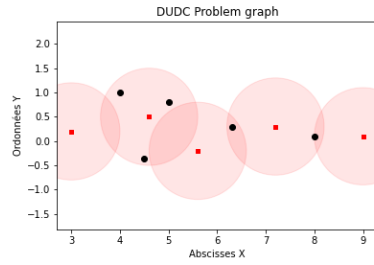


Figure 7: Probleme DUDC sur \mathbb{R}^2

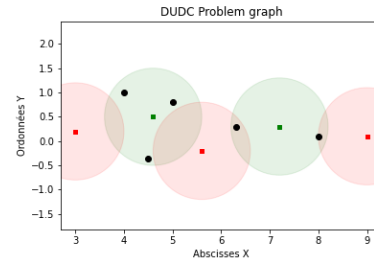


Figure 8: Solution du probleme

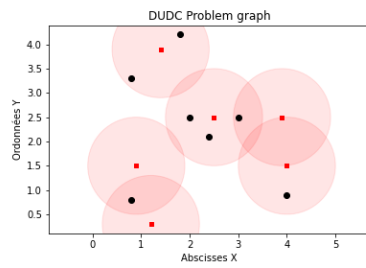


Figure 9: Probleme DUDC sur \mathbb{R}^2

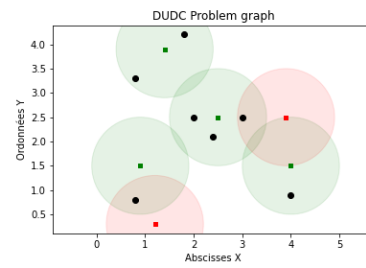


Figure 10: Solution du probleme

Exercice 2: (UEP)

I-1)-Algorithme diviser pour régner pour UEP

Algorithme 3 : UEP – diviser pour régner

```

1 Input : Soit un ensemble  $\mathcal{F}$  de  $n$  fonctions linéaires  $\mathcal{F} = \{y_1, \dots, y_n\}$ 
   et  $y_i = ax_i + b_i$ .
2 Output : Une fonction  $f$  donnée par  $f(x) = \max_{i \leq n} (y_i)$  ( elle sera
   représentée par un ensemble des fonctions contribuant au points de
   cassures) et  $\mathcal{C}$  un ensemble de points de cassure.
3 begin
4   si (  $n \leq 1$  ) alors
5     return  $\mathcal{F}, \emptyset$ 
6   fin
7    $\mathcal{F}_1, \mathcal{C}_1 = \text{UEP}(\mathcal{F}[0 : \frac{n}{2}])$  ;
8    $\mathcal{F}_2, \mathcal{C}_2 = \text{UEP}(\mathcal{F}[\frac{n}{2} : n])$  ;
9    $f, \mathcal{C} = [], []$  ;
10   $\text{intervals} = \text{les intervalles valides dans } \mathcal{C}_1 \times \mathcal{C}_2 \times \{-\infty, +\infty\}$  ;
11  pour chaque  $I \in \text{intervals}$  faire
12     $f_1 = \text{une fonction de } \mathcal{F}_1 \text{ qui contient au moins une des}$ 
       $\text{extrémités de } I$ ;
13     $f_2 = \text{une fonction de } \mathcal{F}_2 \text{ qui contient au moins une des}$ 
       $\text{extrémités de } I$ ;
14     $x = \text{l'abscisse de } f_1 \cap f_2$ ;
15    si (  $x \in I$  ) alors
16       $g = \text{la fonction avant l'intersection } x$  ;
17       $h = \text{la fonction après l'intersection } x$  ;
18       $f = f \cup \{g, h\}$ ;
19       $\mathcal{C} = \mathcal{C} \cup \{x\} \cup \text{points de cassure associés à } g \text{ et } h \text{ sur } I$ ;
20    fin
21    sinon
22       $g = \max(f_1, f_2) \text{ sur } I$  ;
23       $f = f \cup \{g\}$ ;
24       $\mathcal{C} = \mathcal{C} \cup \text{points de cassure associés à } g \text{ sur } I$ ;
25    fin
26  fin
27  return  $f, \mathcal{C}$  ( de gauche à droite)
28 end

```

NB: Les résultats de cet algorithme sont des ensemble au sens d'inclusion (Set en Python par exemple) puisque dans la ligne (19) dans le cas ou on a g, h déjà dans f on les ajoute pas .Et de meme pour \mathcal{C} .

Dans l'algorithme on utilise souvent la notation points de cassure associé à une fonction sur I (soit g par exemple) ; on veut simplement dire par ceci qu'on prend les points qui sont vérifiés par g et aussi sont une des extrémités de I

Dans la ligne (10) on souhaite avoir tout les intervals possibles entre les points d'intersections (en $O(n^2)$).

Exemple : Si $C_1 = \{1, 2\}$ et $C_2 = \{3\}$ on aura :

$intervals = \{] - \infty, 1],] - \infty, 2],] - \infty, 3], [1, 3], [2, 3], [1, +\infty[, [2, +\infty[, [3, +\infty[\}$

I-2) Analyse de complexité

On divise le probleme en deux sous-problemes et pour le cout de combinaison des deux résultats on doit faire une intersection pour chaque element de l'ensemble \mathcal{F}_1 et l'ensemble \mathcal{F}_2 pour chaque interval I donc en $O(n^2)$ d'ou la relation de récurrence suivante :

$$T(n) = 2 \times T\left(\frac{n}{2}\right) + n^2 \text{ Avec : } T(1) = 1$$

$$T(n) = 2 \times (2 \times T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2) + n^2$$

...

$$T(n) = 2^i \times T\left(\frac{n}{2^i}\right) + n^2 \sum_{j=0}^{i-1} \left(\frac{2}{2^2}\right)^j \text{ Jusqu'à } i = \log_2(n)$$

$$T(n) = 2^{\log_2(n)} \times T(1) + n^2 \sum_{j=0}^{\log_2(n)-1} \left(\frac{1}{2}\right)^j$$

$$T(n) = n^{\log_2(2)} \times T(1) + n^2 \times \frac{1 - \frac{1}{2}^{\log_2(n)}}{1 - \frac{1}{2}}$$

$$T(n) = n + 2n^2 = \Theta(n^2)$$

Vérifions par le **Master Theorem**:

On a

$$T(n) = 2 \times T\left(\frac{n}{2}\right) + n^2$$

On a $a = 2$ et $b = 2$ et $f(n) = n^2$

$\log_b(a) = 1$ et donc on constate que $f(n) = \Omega(n^{1+\epsilon})$ pour $\epsilon = \frac{1}{2}$

On se trouve dans le cas 3 du théoreme Maitre.

$a \times f\left(\frac{n}{b}\right) = 2 \times \frac{n^2}{4} \leq c \times n^2$ pour tout $0 \leq c \leq 1$ et $n \geq n_0$ Donc $c \geq \frac{1}{2}$.

Pour $c = \frac{1}{2}$ la condition est vérifiée à partir de $n_0 = 0$. Et donc on a bien :

$$T(n) = \Theta(f(n)) = \Theta(n^2)$$

Remarque : (autre approche possible) Pour une solution de UEP soit (y_i, \dots, y_j) tels que $y_k = m_k x + b_k$ on a les pentes evolue en croissant de gauche à droite (de y_i à y_j) et donc on donne en entrée de l'algorithme la liste des fonctions triée par ordre croissant de pente. en $O(n \log(n))$ et donc on peut regner la partie gauche avec celle de droite en moins d'étapes. Notre solution est donc plus couteuse que cette solution.

II-1)-Algorithme de programmation dynamique pour UEP

Idée de l'algorithme : Dans cet algorithme, si l'entrée est une liste \mathcal{F} de fonctions linéaires de taille n supposons qu'on la trie de gauche à droite au début de l'algorithme (par ordre croissant de pente), nous calculons l'enveloppe supérieure des k premières fonctions pour $k \in \{1, 2, \dots, n\}$, et on retourne la dernière enveloppe calculée sur toutes les n fonctions.

Pour calculer l'enveloppe de k fonctions lorsque $k \geq 2$, on prend l'enveloppe précédemment calculée de $k - 1$ fonctions et le point de cassure le plus à droite notant le (c_d^k) parmi les points d'intersection entre la k^{eme} fonction et les fonction pris dans l'enveloppe $k - 1$.

Car : En procédant de gauche à droite la pente augmente et donc le seul point qui pourra etre pris est celui qui se trouve tout à la droite et donc dans l'enveloppe à l'itération k contient :

1. Les fonctions qui participent à cette intersection dans c_d^k
2. Les fonctions de l'enveloppe $k - 1$ qui ont des points de cassure à gauche de c_d^k (car ceux qui se trouvent à droite nous n'intéressent pas).
3. Les points de cassure seront donc ceux qui se trouvent à gauche de c_d^k et c_d^k lui-meme.

Comment gérer les intersections à l'étape k :

On utilise la structure H qui est présentée sous forme d'une structure (clé , valeur) :

1. clé : le point de cassure
2. valeur : les fonctions qui s'intersectent dans ce point

elle est essentiellement utile pour éviter de recalculer à chaque fois les intersections entre les fonctions pour un point de cassure c_i donné $i \in \{1, 2, \dots, n\}$.

NB : En utilisant cette structure on passe de $O(n^2)$ à $O(n)$ pour chaque itération.

Mise à jour de H

Lors de l'appel **Mise_a_jour**(H , intersections , c_d^k) on parcourt tous les éléments (key , value) de H et on garde que ceux qui vérifient la condition suivante: key se trouve à gauche de c_d^k . Et à la fin on ajoute (c_d^k , intersections).

Algorithme 4 : UEP – programmation dynamique

```

1 Input : Soit un ensemble  $\mathcal{F}$  de  $n$  fonctions linéaires  $\mathcal{F} = \{y_1, \dots, y_n\}$ 
   et  $y_i = ax_i + b_i$ .
2 Output : Une fonction  $f$  donnée par  $f(x) = \max_{i \leq n}(y_i)$  ( elle sera
   représentée par un ensemble des fonctions contribuant aux points de
   cassures) et  $\mathcal{C}$  un ensemble de points de cassure.
3 begin
4   si (  $n < 1$  ) alors
5     return  $\mathcal{F}, \emptyset$ 
6   fin
7   Trier  $\mathcal{F}$  de gauche à droite;
8   envelopes = [ ];
9    $f, \mathcal{C} = \mathcal{F}[0], [ ]$ ;
10  envelopes = envelopes  $\cup [ f, \mathcal{C} ]$ ;
11   $H = \emptyset$  // Le role H est déjà expliqué dans le paragraphe avant;
12  pour chaque  $f_k \in \mathcal{F}[1 : n]$  faire
13     $f'_k, \mathcal{C}'_k = \text{envelopes}[-1]$  // Dernière enveloppe calculée ;
14    intersections =  $f'_k \cap \{f_k\}$  // les intersections entre les fonctions
      de la dernière enveloppe et notre fct ;
15    si ( intersections  $\neq \emptyset$  ) alors
16       $c_d^k$  = le point le plus à droite de intersections;
17      intersect_en_ $c_d^k$  = { Les fonctions de  $f'_k$  qui participent à
        cette intersection dans  $c_d^k + f_k$  } ;
18       $f = \text{intersect\_en\_}c_d^k \cup H[c_i]$  tels que  $c_i$  à gauche de  $c_d^k$  pour
        tout  $i$  ;
19       $\mathcal{C} = \{\text{les points dans } \mathcal{C}'_k \text{ qui sont à gauche de } c_d^k\} \cup \{c_d^k\}$ ;
20      Mise_a_jour( $H, \text{intersect\_en\_}c_d^k, c_d^k$ );
21    fin
22    sinon
23       $f = f'_i$  // Dans ce cas la fonction précédente nous intéresse;
24       $\mathcal{C} = \mathcal{C}'_i$  // Les points de cassure ne changent pas;
25    fin
26    envelopes = envelopes  $\cup \{f, \mathcal{C}\}$ ;
27  fin
28  return envelopes[-1] // Dernière enveloppe
29 end

```

Relation De récurrence :

$$UEP(k) = \begin{cases} \max(UEP(k-1), Maj(k-1) \cup \mathcal{F}[k]), & \text{si } k \neq 1 \\ \mathcal{F}[0] & \text{si } k = 1 \end{cases}$$

Explication de la relation :

Supposons on souhaite calculer l'enveloppe supérieure à l'itération k soit $UEP(k)$ alors la formule $\max(UEP(k-1), Maj(k-1) \cup \mathcal{F}[k])$ veut simplement dire soit

l'enveloppe supérieure est celle de l'itération $k - 1$ et donc on prends pas de fonction à l'itération k (on se contente des $k - 1$ fonctions.
 Sinon si la fonction $\mathcal{F}[k]$ a une intersection avec $\text{UEP}(k - 1)$. Dans ce cas la plus à droite des intersections nous permet de trouver les fonctions de $\text{UEP}(k - 1)$ qui vérifie notre condition de l'algorithme (**ligne 18**) ($\text{Maj}(k - 1)$ dans notre relation) et on ajoute $\mathcal{F}[k]$ au résultat.

II-2)-Analyse de Complexité

La complexité de notre algorithme est donnée par la relation suivante :

$$\begin{cases} T(n) = T(n - 1) + n \\ T(1) = 1 \end{cases}$$

On peut résoudre par la méthode des remplacements successifs.

$$T(n) = T(n - 2) + n - 1 + n$$

$$T(n) = T(n - 3) + n - 2 + n - 1 + n$$

...

$$T(n) = T(n - k) + (n - (k - 1)) + (n - (k - 2)) + \dots + (n - 1) + n$$

$$T(n) = T(1) + 1 + 2 + 3 + \dots + (n - 1) + n$$

$$T(n) = T(1) + \frac{n(n + 1)}{2} = O(n^2)$$

Donc on a bien notre algorithme s'exécute en $O(n^2)$.

NB : Le tri effectué au début (par ordre croissant de pente) est en $O(n \log(n))$.

Exemple de notre algorithme :

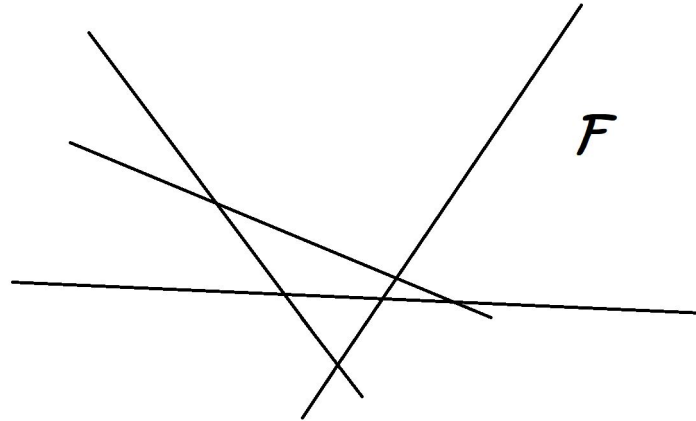


Figure 11: Ensemble des fonctions \mathcal{F}

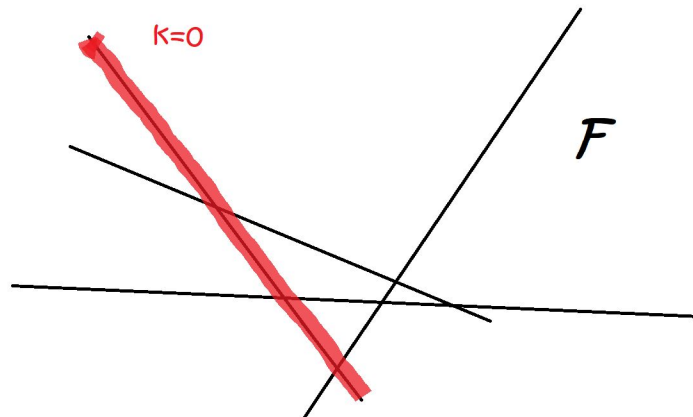


Figure 12: Itération $k=0$

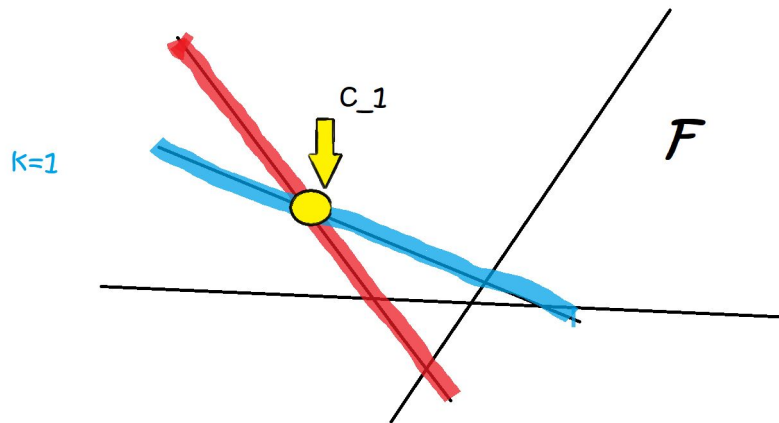


Figure 13: Itération $k=1$

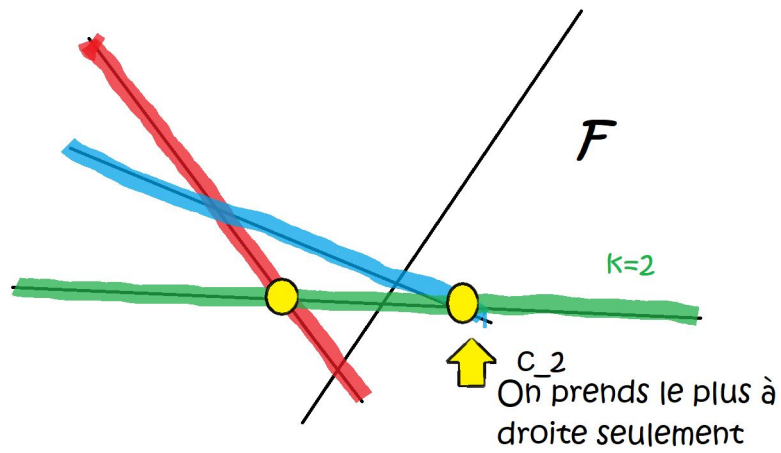


Figure 14: Itération $k=2$

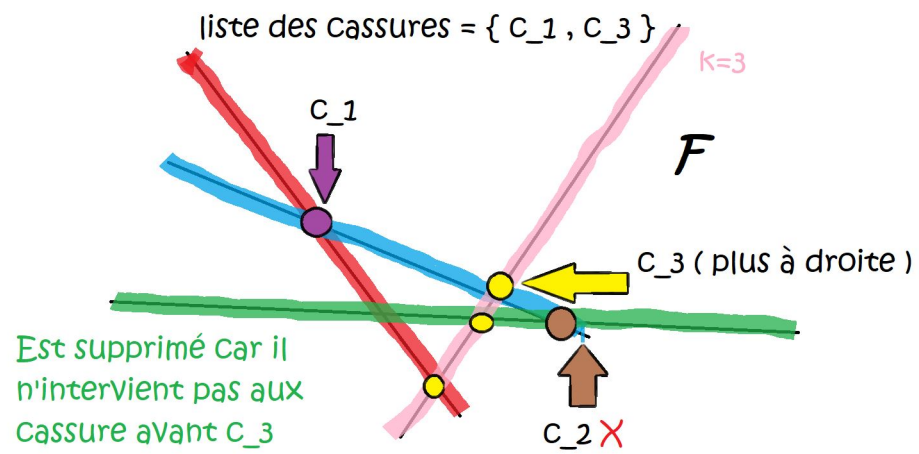


Figure 15: Itération $k=3$

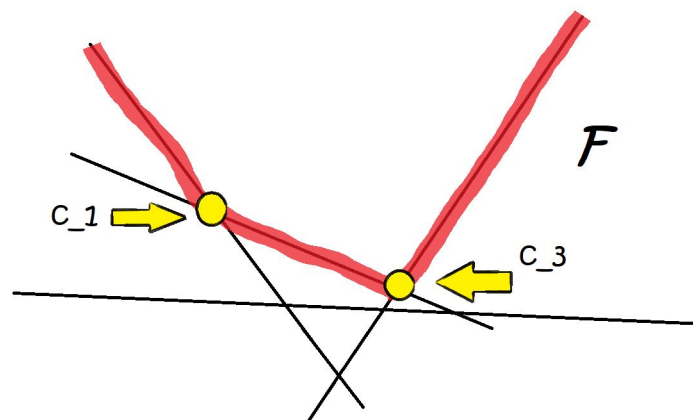


Figure 16: Résultat de l'algorithme