Rapport de projet OS01

TRAN Quoc Nhat Han & Adrien WARTELLE

6 décembre 2018

Sommaire

1 Calcul d'un arbre de longueur minimale

Basé sur le principe de Prim, nous proposons 3 algorithmes que nous appellons respectivement *Naive*, *Improved* et *Best*.

Pour la consistence entre le code et le rapport, nous conservons les noms de variables (qui sont anglais dans le code actuel).

1.1 Structures de données principales

1.1.1 `Point`

Un point est défini un couple de réels positifs (x, y) majorés par `maxX` et `maxY` respectivement.

De même, nous définissons le racine (`root`) avec les coordonnées $(0, 0)$.

Les points entrées sont lus et enregistrés au tableau `inputPoints`. Le racine est mis par défaut à `inputPoints[0]`.

Désormais, tous les autres variables dans le programme utiliseront la position du point dans `inputPoints` au lieu de recopier les valeurs de coordonnées, afin d'économiser la mémoire.

1.1.2 `dist2` : Table de distances

En général, dans 3 approches, nous calculons leurs distances et stockons en avance dans un tableau `dist2` afin d'éviter de recalculer les distances entre les points à chaque itération.

`dist2[i][j]` est la distance *en carré* entre l'i-ième point et l'j-ième point. Notons que l'opération racine est couteuse et $0 < x < y \Leftrightarrow x^2 < y^2$. Par conséquence, comparer deux distances de couples (i_1, j_1) et (i_2, j_2) revient à comparaison de `dist2[i1][j1]` et `dist2[i2][j2]`.

1.1.3 `network` : L'arbre à construire

Cet arbre consite d'un registre des points ajoutés (`vertex`) accompagné avec le compteur `n`, d'un registre des branches créés (`edge`) accompagné avec le compteur `m`, et un compteur de longueur total (`totalLength`).

1.2 Algorithmes implémentés

N dénote le nombre de points entrées.

Cet arbre devra contient exactement N branche. Chaque algorithme propose une approche différente à la recherche de nouvelle minimale branche.

1.2.1 Naive

L'algorithme *Naive* cherche franchement, à chaque itération, dans le tableau `dist2` le distance (i, j) le plus court tel que $i \in network$ et $j \notin network$, puis il ajoute j et sa liaison (i, j) au `network`.

La détection de l'attache à l'arbre est faite par le tableau booléan `inTree` : `inTree[i]` vaut `true` si l'i-ième point appartient à `network`, et sinon `false`.

Algorithme 1 : Algorithme Naive

```
// L'ajout de la branche i-ième
pour i allant de 1 à N faire
    // Initialisation
    newEdgeStart ← 0;
    newEdgeEnd ← −1;
    dist2Min ← maxX^2 + maxY^2;
    pour j allant de 0 à N faire
        // Recherche d'un point de début
        si j ∈ network alors
            // Recherche d'un point de fin
            pour k allant de 0 à N faire
                si k ≠ j et k ∉ network alors
                    si dist2[j][k] < dist2Min alors
                        newEdgeStart ← j;
                        newEdgeEnd ← k;
                        dist2Min = dist2[j][k];
                    fin
                fin
            fin
        fin
        Ajouter newEdgeEnd au network;
        Ajouter la branche (newEdgeStart, newEdgeEnd) au network;
    fin
fin
```

Comme nous avons 3 boucles embriquées parcourant de 1 ou 0 à N, la complexité d'algorithm est donc $O(N^3)$.

1.2.2 Improved

Amélioré de *Naive*, *Improved* utilise le tableau `orderedPoints` pour stocker les points entrées dans l'ordre d'ajoute au `network`. C'est-à-dire, si `network` possède déjà n points, le point `orderedPoints[i]`-ième est ajouté au `network` ($i = \overline{0, n-1}$), la reste est dehors.

Algorithme 2 : Algorithme Improved

```
// L'ajout de la branche i-ième
pour i allant de 1 à N faire
    // Initialisation
    newEdgeStart ← 0;
    newEdgeEnd ← −1;
    dist2Min ← maxX^2 + maxY^2;
    pour j allant de 0 à network.n − 1 faire
        // Grace à la définition de orderedPoints
        // Recherche d'un point de début est réduit à
        // orderedPoints[0] jusqu'au
        // orderedPoints[network.n - 1]
        inNode ← orderedPoints[j];
        si j ∈ network alors
            // Recherche d'un point de fin est réduit à
            // orderedPoints[network.n] jusqu'au
            // orderedPoints[N]
            pour k allant de network.n à N faire
                outNode ← orderedPoints[k];
                si dist2[inNode][outNode] < dist2Min alors
                    newEdgeStart ← j;
                    newEdgeEnd ← k;
                    dist2Min = dist2[inNode][outNode];
                fin
            fin
        fin
        Ajouter orderedPoints[newEdgeEnd] au network;
        Ajouter orderedPoints[newEdgeEnd] au fin des points
         intérieurs de orderedPoints;
        Ajouter la branche
         (orderedPoints[newEdgeStart], orderedPoints[newEdgeEnd])
         au network;
    fin
fin
```

Bien que le nombre d'éléments à parcourir de 2 boucles intérieuses a diminué 2 fois, la complexité de cet algorithme reste $O(N^3)$.

1.2.3 Best

La version meilleure utilise le tableau `nearestNetworkNeighbor` indiquant le voisin le plus proche du point j-ième. Le tableau est mis à jour de manière sélective pendant chaque itération, grâce à la variable `newestNode`, qui stocke le point récemment ajouté.

Algorithme 3 : Algorithme Best

```
// L'ajout de la branche i-ième
pour i allant de 1 à N faire
    // Initialisation
    newEdgeStart ← 0;
    newEdgeEnd ← −1;
    dist2Min ← maxX^2 + maxY^2;
    pour j allant de 0 à N faire
        // Recherche d'un point de fin
        si j ∉ network alors
            // Mettre à jour le voisin le plus proche de j
            // en comparant le newestNode et
               nearestNetworkNeighbor[j]
            si dist2[newestNode][j] < dist2[nearestNetworkNeighbor[j]][j]
             alors
                nearestNetworkNeighbor[j] = newestNode;
            fin
            // Chercher la branche la plus courte
            si dist2[nearestNetworkNeighbor[j]][j] < dist2Min alors
                dist2Min = dist2[nearestNetworkNeighbor[j]][j];
                newEdgeStart = nearestNetworkNeighbor[j];
                newEdgeEnd = j;
            fin
        fin
        Ajouter newEdgeEnd au network;
        Ajouter la branche (newEdgeStart, newEdgeEnd) au network;
    fin
fin
```

A l'aide de `newestNode` et `nearestNetworkNeighbor`, nous économisons un boucle, résultant la complexité de l'ordre $O(N^2)$, qui est le meilleur possible selon les calculs démontrés de Prim.

1.3 Exécution, mesure et visualisation

En utilisant *MersenneTwister.h*, nous générons des jeux de données de tailles différentes : $N = 5, 10, 100, 200, 500, 1000, 10000$.

Spec de l'ordinateur à tester : Lenovo Y520, Intel(R) Core(TM) i7-7700HQ, CPU@2.8GHz(8CPUs), RAM 8192MB.

N	Naive	Improved	Best
5	0,014587	0,003282	0,013128
10	0,026986	0,004011	0,010211
100	2,920300	0,233755	0,050324
200	5,043060	1,77413	0,163373
500	161,429000	28,331	3,527840
1000	1169,760000	265,508000	4,60399
10000	$>$ 60000	$>$60000	402,536000

TABLE 1 – Table de temps d'exécution (ms)

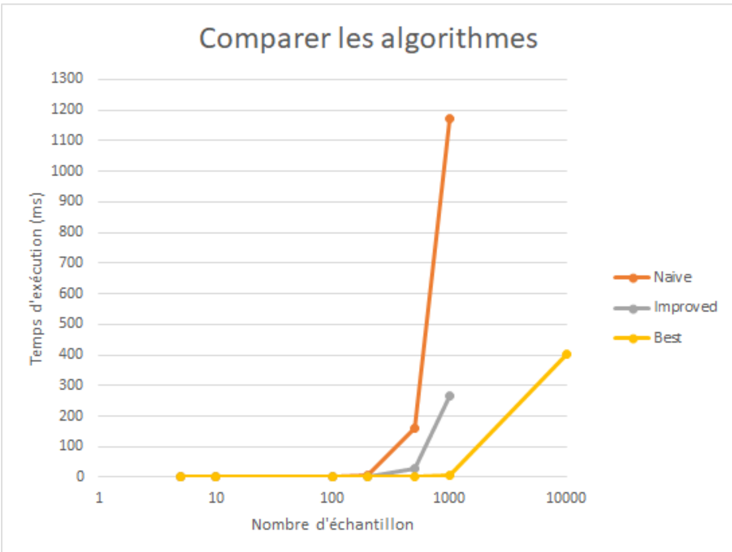

FIGURE 1 – Graphe de temps d'exécution

Nous voyons clairement que l'algorithme *Best* donne la solution le plus vite quand la volume des données est énorme. Nous ajoutons ici quelques visualisations de jeux données et le réseau trouvé (le code de visualiser, écrit en MATLAB, est mis dans l'annexe).

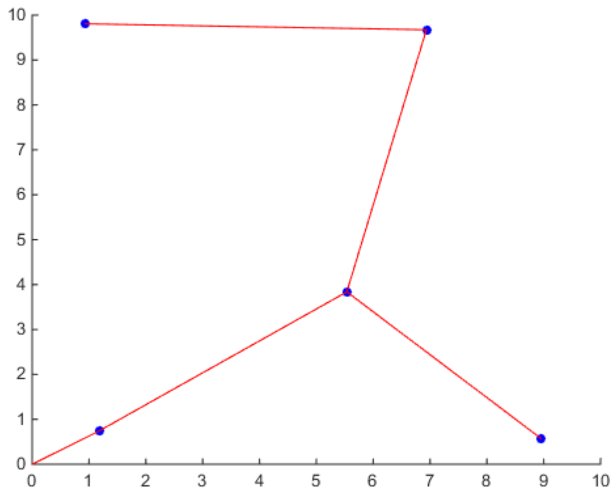

FIGURE 2 – L'arbre le plus court lorsque $N = 5$

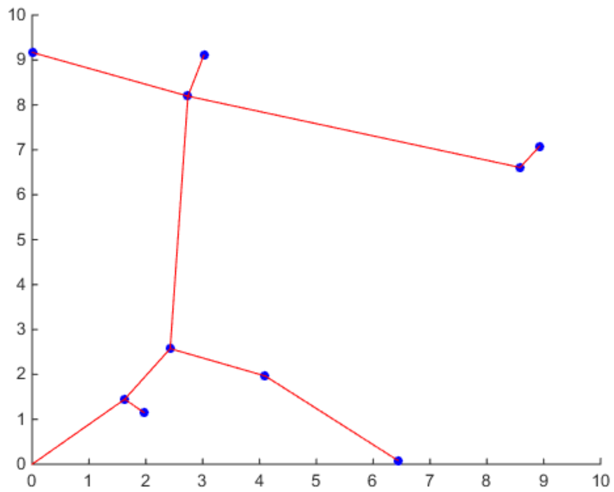

FIGURE 3 – L'arbre le plus court lorsque $N = 10$

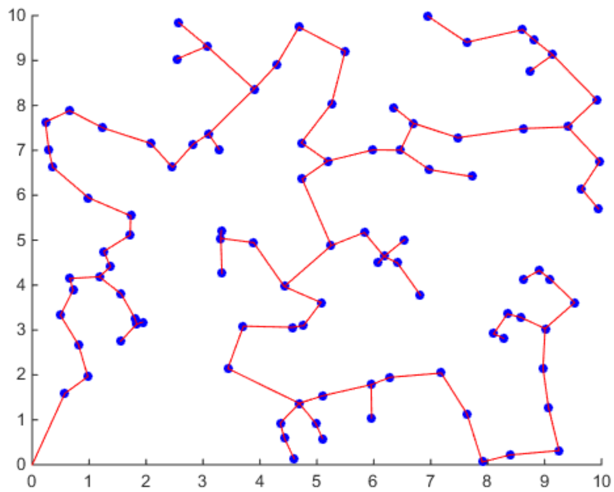

FIGURE 4 – L'arbre le plus court lorsque $N = 100$

2 Plannification de production d'huiles

2.1 Modèle 1

Soient les paramètres :
- V : l'ensemble d'huiles végétales brutes.
- H : l'ensemble d'huiles hydrogénées brutes.
- $B = V \cup H$: l'ensemble d'huiles brutes.
- N : le nombre de mois à plannifier.
- p_f : prix vendu de produit final. (euro/t)
- $p_{i,j}$: cout d'achat de l'huile i au mois j. (euro/t)
- V_{max} : quantité maximale de raffinage d'huiles végétales. (t)
- H_{max} : quantité maximale de raffinage d'huiles hydrogénées. (t)
- SM_i : stock maximale chaque mois de l'huile i. (t)
- c_s : cout de stockage. (euro/t)
- SI_i : stock initial de l'huile i. (t)
- SF_i : stock final de l'huile i. (t)
- v_i : coefficient de viscosité de l'huile i.
- v_{min} : viscosité minimale de produit final.
- v_{max} : viscosité maximale de produit final.

Soient les variables :
- $a_{i,j}$: quantité d'achat de l'huile i au mois j. (t)
- $s_{i,j}$: stock de l'huile i au mois j. (t)
- $r_{i,j}$: quantité de raffinage de l'huile i au mois j. (t)

Soit P le profit de l'entreprise après N mois.

Le système d'équations linéaires est décrit de manière suivant :

$$\text{Maximizer } P = \sum_{j=1}^{N} \sum_{i \in B} r_{i,j} p_f - \sum_{j=1}^{N} \sum_{i \in B} a_{i,j} p_{i,j} - c_s \sum_{j=1}^{N} \sum_{i \in B} s_{i,j}$$

Sachant que :

$$\forall j = \overline{1, N} : \sum_{i \in V} r_{i,j} \leq V_{\max}$$

$$\forall j = \overline{1, N} : \sum_{i \in H} r_{i,j} \leq H_{\max}$$

$$\forall j = \overline{1, N} : \sum_{i \in B} r_{i,j} v_i \leq v_{\max} \sum_{i \in B} r_{i,j}$$

$$\forall j = \overline{1, N} : \sum_{i \in B} r_{i,j} v_i \geq v_{\min} \sum_{i \in B} r_{i,j}$$

$$\forall j = \overline{1, N}, \forall i \in B : s_{i,j} = s_{i,j-1} + a_{i,j} - r_{i,j}$$

$$\forall i \in B : s_{i,0} = SI_i$$

$$\forall i \in B : s_{i,N} = SF_i$$

$$\forall j = \overline{0, N}, \forall i \in B : SM_i \geq s_{i,j} \geq 0; a_{i,j} \geq 0; r_{i,j} \geq 0$$

La solution sera :

- Le profit maximale après N mois : P (euros).
- Au mois j :
  - Pour l'huile i :
    - Achater : $a_{i,j}$ tonne(s).
    - Raffiner : $r_{i,j}$ tonne(s).
    - Stocker : $s_{i,j}$ tonne(s).

2.2 Modèle 2

Soient les paramètres :

- V : l'ensemble d'huiles végétales brutes.
- H : l'ensemble d'huiles hydrogénées brutes.
- $B = V \cup H$: l'ensemble d'huiles brutes.
- N : le nombre de mois à plannifier.
- p_f : prix vendu de produit final. (euro/t)
- $p_{i,j}$: cout d'achat de l'huile i au mois j. (euro/t)
- V_{max} : quantité maximale de raffinage d'huiles végétales. (t)
- H_{max} : quantité maximale de raffinage d'huiles hydrogénées. (t)
- SM_i : stock maximale chaque mois de l'huile i. (t)
- c_s : cout de stockage. (euro/t)
- SI_i : stock initial de l'huile i. (t)
- SF_i : stock final de l'huile i. (t)
- v_i : coefficient de viscosité de l'huile i.
- v_{min} : viscosité minimale de produit final.
- v_{max} : viscosité maximale de produit final.
- $D = B \times B$: couples de dépendances. $(x, y) \in D$ signifie que si x est utilisée, y doit être utilisée aussi.

— n_{max} : nombre maximal d'huiles utilisées dans un mois.
— u_{min} : la quantité minimale si une huile est utilisée. (t)
— $M = V_{max} + H_{max}$: un grand nombre.
Soient les variables :
— $a_{i,j}$: quantité d'achat de l'huile i au mois j. (t)
— $s_{i,j}$: stock de l'huile i au mois j. (t)
— $r_{i,j}$: quantité de raffinage de l'huile i au mois j. (t)
— $u_{i,j}$: variable binaire, indiquant l'usage de l'huile i au mois j.
Soit P le profit de l'entreprise après N mois.
Le système d'équations linéaires est décrit de manière suivant :

$$\text{Maximizer } P = \sum_{j=1}^{N} \sum_{i \in B} r_{i,j} p_f - \sum_{j=1}^{N} \sum_{i \in B} a_{i,j} p_{i,j} - c_s \sum_{j=1}^{N} \sum_{i \in B} s_{i,j}$$

Sachant que :

$$\forall j = \overline{1, N} : \sum_{i \in V} r_{i,j} \leq V_{\max}$$

$$\forall j = \overline{1, N} : \sum_{i \in H} r_{i,j} \leq H_{\max}$$

$$\forall j = \overline{1, N} : \sum_{i \in B} r_{i,j} v_i \leq v_{\max} \sum_{i \in B} r_{i,j}$$

$$\forall j = \overline{1, N} : \sum_{i \in B} r_{i,j} v_i \geq v_{\min} \sum_{i \in B} r_{i,j}$$

$$\forall j = \overline{1, N}, \forall i \in B : s_{i,j} = s_{i,j-1} + a_{i,j} - r_{i,j}$$

$$\forall i \in B : s_{i,0} = SI_i$$

$$\forall i \in B : s_{i,N} = SF_i$$

$$\forall j = \overline{1, N}, \forall i \in B : u_{\min} * u_{i,j} \leq r_{i,j} \leq M * u_{i,j}$$

$$\forall j = \overline{1, N} : \sum_{i \in B} u_{i,j} \leq n_{\max}$$

$$\forall j = \overline{1, N}, \forall (i_1, i_2) \in D : u_{i_1,j} \leq u_{i_2,j}$$

$$\forall j = \overline{0, N}, \forall i \in B : SM_i \geq s_{i,j} \geq 0; a_{i,j} \geq 0; r_{i,j} \geq 0$$

La solution sera :
— Le profit maximale après N mois : P (euros).
— Au mois j :
  — Pour l'huile i :
    — Achater : $a_{i,j}$ tonne(s).
    — Raffiner : $r_{i,j}$ tonne(s).
    — Stocker : $s_{i,j}$ tonne(s).