Rapport de projet OS01

TRAN Quoc Nhat Han & Adrien WARTELLE

10 décembre 2018

Sommaire

1 Calcul d'un arbre de longueur minimale

Basé sur le principe de Prim, nous avons programmé 3 version de l'algorithme respectivement dénommée *Normal* (Naïf), *Improved* et *Best*. Les 3 algorithmes (dans les fichiers "SST_*.cpp") utilisent le fichier prototype "Network.h" qui contient les structures de données pour construire et utiliser un graphe avec notamment la lecture des fichiers de données et l'écriture des résultats. Pour générer les points d'entrées, on utilise le code de "InputGenerator.cpp" qui utilise le générateur aléatoire de "MersenneTwister.h".

Pour garder la consistance entre le code et le rapport, nous avons conservé 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 par un couple de réels positifs (x, y) majorés par `maxX` et `maxY` respectivement.

De même, nous définissons le racine (`root`, l'usine) de l'arbre avec les coordonnées $(0, 0)$.

Les points entrées (les maisons) sont lus et enregistrés dans le tableau `inputPoints`. Le racine est mis par défaut à `inputPoints[0]`.

Toutes les autres variables dans le programme utilisent les positions des points dans `inputPoints` au lieu de recopier les valeurs de coordonnées, afin d'économiser de la mémoire.

1.1.2 `dist2` : Table de distances

Pour tous les couples de points d'entrées, 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 *au carré* entre le i-ième point et le 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 comparer de `dist2[i1][j1]` et `dist2[i2][j2]`.

1.1.3 `network` : L'arbre à construire

Ce graphe consiste d'un registre des points ajoutés (`nodes`) accompagné du compteur `n`, d'un registre des branches créés (`edge`) accompagné du compteur `m`, et un compteur de longueur total (`totalLength`) (somme des longueurs/coûts de chaque arbre).

1.2 Algorithmes implémentés

N dénote le nombre de points entrés.

Cet arbre devra contenir exactement N branches (avec au total $N+1$ points). Chaque algorithme propose une approche différente pour la recherche d'une nouvelle branche de longueur (coût) minimale.

1.2.1 Normal (naïf)

L'algorithme de base cherche, 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`. Il est naïf puisque qu'il reparcourt entièrement le tableau des distances (au lieu de garder en mémoire des informations sur les distances minimales) à chaque itération.

La détection de l'attachement à l'arbre est faite grâce au tableau booléen `inTree` : `inTree[i]` vaut `true` si le i-ième point appartient à `network`, 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 de l'algorithme est donc $O(N^3)$. Les variables newEdgeStart et newEdgeEnd correspondent aux extremitées de l'arc à ajouter (de longueur minimale dist2Min) pour chaque itération.

1.2.2 Improved

La version *Improved* améliore la version naïve en utilisant le tableau `orderedPoints` pour stocker les points entrés dans l'ordre d'ajout au `network`. C'est-à-dire, si `network` possède déjà n points, les points en position `orderedPoints[i]` pour $n - 1 < i < N$ (indicage de 0 à N-1) sont encore à ajouter au `network`. On évite ainsi toutes les itérations de recherche de longueur minimale pour les couples de points déjà dans l'arbre et ceux en dehors de l'arbre.

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 des 2 boucles intérieures a diminué de 2 fois, la complexité de cet algorithme reste $O(N^3)$.

1.2.3 Best

La version "best" (meilleure) utilise le tableau `nearestNetworkNeighbor` indiquant le voisin le plus proche pour chaque point. Le tableau est mis à jour à chaque itération, grâce à la variable `newestNode`, qui stocke le point récemment ajouté. On remplace ainsi le parcours du tableau des distances (de taille N^2) par celui de `nearestNetworkNeighbor` (de taille N), ce qui réduit d'un ordre la complexité.

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 une boucle, résultant en une complexité de $O(N^2)$, qui est la meilleure possible selon les calculs démontrés de Prim.

1.3 Exécution, mesure et visualisation

En utilisant *MersenneTwister.h* (et "InputGenerator.cpp"), nous avons généré des jeux de données de tailles différentes : $N = 5, 10, 100, 200, 500, 1000, 10000$.

Spec de l'ordinateur de test : 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)

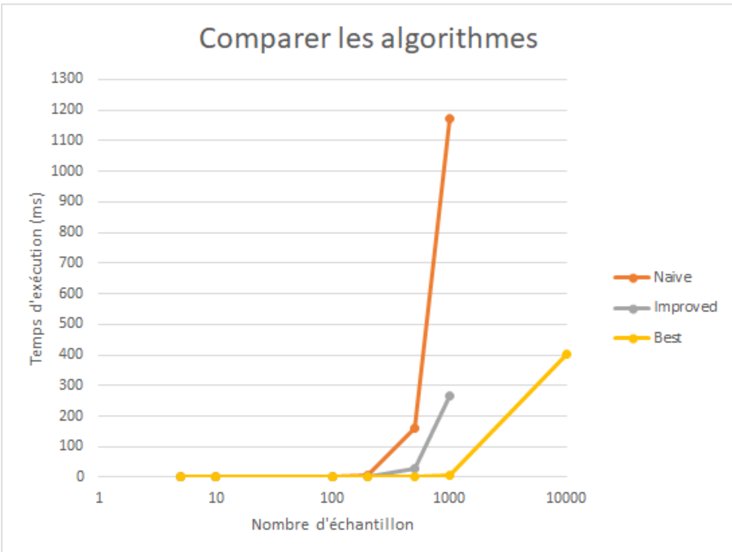

FIGURE 1 – Graphe de temps d'exécution

Nous voyons clairement que l'algorithme *Best* donne la solution le plus rapidement quand le volume des données est énorme. Nous ajoutons ici quelques visualisations (figures 2, 3, 4) de jeux données et le réseau trouvé (le code de visualisation, écrit en MATLAB dans le fichier plot.m est en annexe).

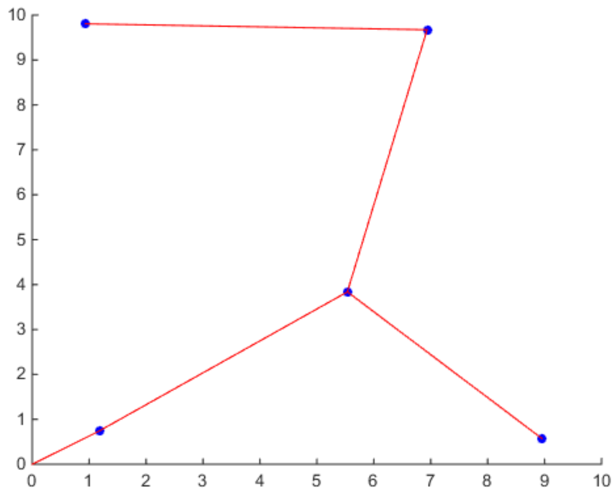

FIGURE 2 – L'arbre le plus court lorsque $N = 5$

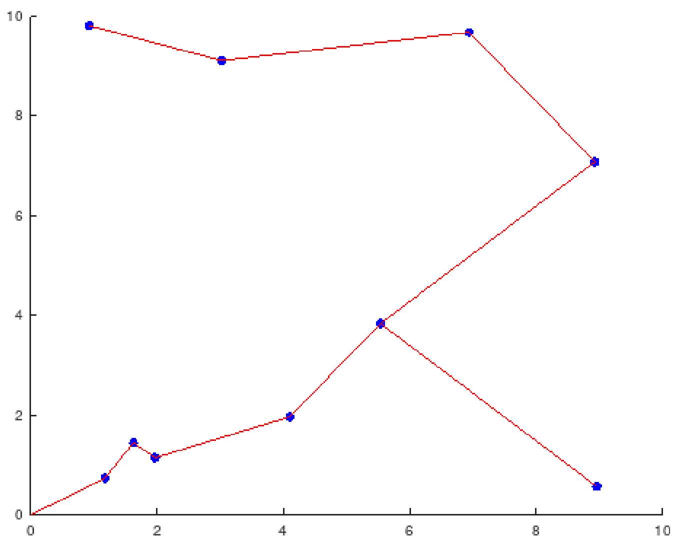

FIGURE 3 – L'arbre le plus court lorsque $N = 10$

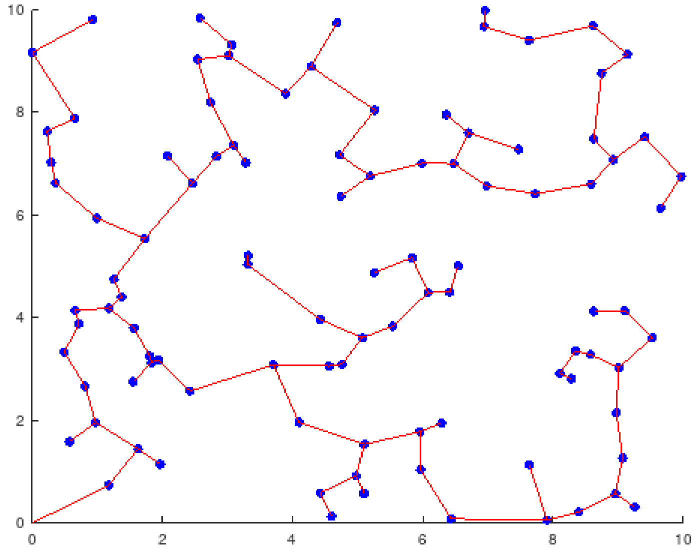

FIGURE 4 – L'arbre le plus court lorsque $N = 100$

2 Plannification de production d'huiles

2.1 Modèle 1

Soient les paramètres :
- V : l'ensemble des huiles végétales brutes.
- H : l'ensemble des huiles hydrogénées brutes.
- $B = V \cup H$: l'ensemble des huiles brutes.
- N : le nombre de mois à planifier.
- p_f : prix de vente du produit final. (euro/t)
- $p_{i,j}$: coût 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 maximal chaque mois pour l'huile i. (t)
- c_s : coût 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 du produit final.
- v_{max} : viscosité maximale du 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 (la fonction objective avec les contraintes)

est décrite de la manière suivante :

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

Sous contraintes :

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

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

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

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

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

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

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

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

La solution sera affichée comme ceci dans le fichier "OilFabrication/OilFab_a.res" :

- Le profit maximal 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).

On obtient un profit maximal après 6 mois de **107843** (euros).

2.2 Modèle 2

Soient les paramètres :

- V : l'ensemble des huiles végétales brutes.
- H : l'ensemble des huiles hydrogénées brutes.
- $B = V \cup H$: l'ensemble des huiles brutes.
- N : le nombre de mois à planifier.
- p_f : prix de vente du produit final. (euro/t)
- $p_{i,j}$: coût 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 maximal chaque mois de l'huile i. (t)
- c_s : coût 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 (la fonction objective avec les contraintes) est décrit de la manière suivante :

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

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 affichée comme ceci dans le fichier "OilFabrication/OilFab_b.res" :
— Le profit maximal après N mois : P (euros).
— Au mois j :
  — Pour l'huile i (si utilisée) :
    — Achater : $a_{i,j}$ tonne(s).
    — Raffiner : $r_{i,j}$ tonne(s).
    — Stocker : $s_{i,j}$ tonne(s).

On obtient un profit maximal après 6 mois de **100279** (euros).

3 Annexe

3.1 Visualisation d'arbre (plot.m)

```
clear;

fin = './input/input_1000_10_10.txt';
##input_5_10_10.txt
##input_10_10_10.txt
##input_100_10_10.txt
##input_200_10_10.txt
##input_500_10_10.txt
##input_1000_10_10.txt
fout = './output/outputV2_input_1000_10_10.txt';
##outputV2_input_5_10_10.txt
##outputV2_input_10_10_10.txt
##outputV2_input_100_10_10.txt
##outputV2_input_200_10_10.txt
##outputV2_input_500_10_10.txt
##outputV2_input_1000_10_10.txt

% read input
f=fopen(fin);
tline = fgetl(f);
tlines = cell(0,1);
while ischar(tline)
    tlines{end+1,1} = tline;
    tline = fgetl(f);
end
fclose(f);

n = sscanf(tlines{1}, '%d');
max = sscanf(tlines{2}, '%f %f');
p = zeros(n, 2);
for i=1:n
    p(i,:) = sscanf(tlines{i+2}, '%f %f');
end;

% read output
f=fopen(fout);
oline = fgetl(f);
olines = cell(0,1);
while ischar(oline)
    olines{end+1,1} = oline;
    oline = fgetl(f);
end
fclose(f);

minL = sscanf(olines{1}, '%f');
disp(minL);
```

```
starts = zeros(n, 2);
ends = zeros(n, 2);
for i=1:n
    l = sscanf(olines{i+1}, '%f %f %f %f');
    starts(i,1)=l(1);
    starts(i,2)=l(2);
    ends(i,1)=l(3);
    ends(i,2)=l(4);
end

% plot
hold on;
axis([0 max(1) 0 max(2)]);
scatter(p(:,1),p(:,2),30,'b','filled');
for i=1:n
    plot([starts(i,1) ends(i,1)],[starts(i,2) ends(i,2)],'r');
end
hold off;
```