Projet Final C++ et Modélisation Mathématique dans GUSEK

OS01 : Modélisation et Programmation Avances

Rim CHATTI, David MORA MEZA

07 janvier 2023

CHATTI Rim, MORA MEZA David

rim.chatti@utt.fr, david.mora_meza@utt.fr

Sommaire

Partie 1 - C++

1.1. Introduction

Dans cette première partie, on va étudier les algorithmes de géométrie (computational geometry) appliquée aux polygones avec langage C++. On va d'abord déclarer une classe polygon et puis réaliser les méthodes associées. Ensuite, on générera aléatoirement des données et analyser les résultats. Enfin, on fers un résumé pour cette partie.

1.2. Première partie

Procédures et fonctions à réaliser pour des polygones quelconques, avec $n >= 3$:

- **Fonction load** :

Cette fonction est conçue pour lire un fichier contenant des coordonnées X-Y et les stocker dans un vecteur 2D. Le fichier est spécifié par le paramètre route, qui indique l'emplacement du fichier sur le disque.

Le fichier contient un nombre entier n sur la première ligne, suivi de n paires de valeurs doubles sur les lignes suivantes.

La fonction commence par ouvrir le fichier en utilisant l'objet fstream F et en lisant la valeur de n. Elle crée ensuite deux vecteurs, X et Y, chacun de taille n+1. Elle lit n paires de valeurs doubles à partir du fichier et stocke la première valeur de chaque paire dans l'élément correspondant de X et la seconde valeur dans l'élément correspondant de Y.

Après avoir lu toutes les valeurs, la fonction ferme le fichier et définit le dernier élément de X et Y comme étant égal au premier élément de chaque vecteur. Elle crée ensuite un nouveau vecteur 2D P en combinant X et Y et renvoie ce vecteur.

Donc, Le vecteur P contient bien les coordonnées X et Y des sommets du polygone, respectivement dans les éléments P[0] et P[1].

P[0] contient le vecteur X, qui stocke les valeurs d'abscisse lues à partir du fichier. P[1] contient le vecteur Y, qui stocke les valeurs d'ordonnée.

Pseudocode de la fonction load:

```
FONCTION load(route)
    déclarer n
    ouvrir le fichier à l'emplacement route
    lire la première valeur du fichier et l'assigner à n
    déclarer un vecteur 2D à P
    déclarer deux vecteurs 1D X et Y, de taille n+1
    i=0
    TANT QUE i < n FAIRE
        lire la i-ème valeur de X du fichier et l'assigner à X[i]
        lire la i-ème valeur de Y du fichier et l'assigner à Y[i]
        incrémenter i
    FIN TANT QUE
    fermer le fichier
    assigner X[n] à X[0]
    assigner Y[n] à Y[0]
    ajouter X à P
    ajouter Y à P
FIN FONCTION
```

- **Fonction Clean** :

Cette fonction serve à éliminer les points qui se trouvent sur des segments de pente constante.

La fonction déclare un vecteur de doubles appelé m qui va stocker les pentes des segments du polygone. Pour ceci on utilise la formule de la pente : pente = (y2 - y1) / (x2 - x1)

La valeur de la pente du premier segment est ajoutée à la fin de m en utilisant la fonction push_back.

```
FONCTION clean(P)
    déclarer un vecteur 1D m de taille P[0].size() - 1
    POUR i allant de 1 à P[0].size() FAIRE
        m[i-1] = (P[0][i] - P[0][i-1]) / (P[1][i] - P[1][i-1])
    FIN POUR
    ajouter m[0] à la fin de m
    déclarer deux vecteurs 1D X et Y
    POUR i allant de 1 à la taille de m FAIRE
        SI m[i] est différent de m[i-1] ALORS
            ajouter P[0][i] à X
            ajouter P[1][i] à Y
        FIN SI
    FIN POUR
    ajouter X[0] à la fin de X
    ajouter Y[0] à la fin de Y
    déclarer un vecteur 2D PN
    ajouter X à PN
    ajouter Y à PN
FIN FONCTION
```

- Fonction Same

Elle prend en entrée deux vecteurs de vecteurs de doubles appelés P et Q qui représentent deux polygones et retourne un booléen indiquant si ces deux polygones sont identiques ou non.

La fonction vérifie si les deux polygones ont le même nombre de points en comparant la taille des vecteurs P[0] et Q[0]. Si ces deux tailles sont différentes, la fonction retourne false et affiche un message indiquant que les polygones sont différents

Ensuite Pour chaque point, la fonction vérifie si le point correspondant dans le polygone Q est le même en comparant les abscisses et les ordonnées des points. Cela permet de

comparer tous les points du polygone Q au polygone P, quelle que soit l'ordre dans lequel ils sont présentés.

```
FONCTION same(P, Q)
    déclarer n égal à la taille de P[0] - 1
    SI la taille de P[0] est différente de la taille de Q[0] ALORS
        renvoyer faux
    FIN SI
    POUR j allant de 0 à n-1 FAIRE
        déclarer S comme étant vrai
        POUR i allant de 0 à n FAIRE
            SI i+j < n ALORS
            SI P[0][i] est différent de Q[0][i+j] OU P[1][i] est différent de Q[1][i+j] ALORS
                    assigner faux à S
                FIN SI
            SINON
        SI P[0][i] est différent de Q[0][i+j-n] OU P[1][i] est différent de Q[1][i+j-n] ALORS
                    assigner faux à S
                FIN SI
            FIN SI
        FIN POUR
        SI S est vrai ALORS
            afficher "The two polygons are the same"
```

- Int_Ext()

Prend en entrée un vecteur de vecteurs de doubles appelé P qui représente un polygone et un vecteur de doubles appelé punt qui représente un point. La fonction retourne un booléen indiquant si le point est à l'intérieur ou à l'extérieur du polygone.

Elle sert à vérifier si un point est à l'intérieur ou à l'extérieur d'un polygone. Pour cela,

elle parcourt chaque côté du polygone et vérifie s'il y a une intersection entre ce côté et une droite verticale passant par le point à tester. Si le nombre d'intersections est pair, alors le point est à l'extérieur du polygone, sinon il est à l'intérieur.

```
FONCTION Int_Ext(P, point)
    n = P[0] - 1
    maxY = P[1][0]
    POUR i de 1 à n FAIRE
        maxY = Max(maxY, P[1][i])
    FIN POUR
    x = point[0]
    cpt = 0
    POUR i de 1 n FAIRE
        M =   (P[1][i] - P[1][i-1]) / (P[0][i] - P[0][i-1])
        y = m * (x - P[0][i]) + P[1][i]
        SI (x - P[0][i]) * (x - P[0][i-1]) < 0 ET (y - point[1]) * (y - maxY - 1) < 0 ALORS
            cpt++
        FIN SI
    FIN POUR
    SI cpt %2 != de 0 ALORS
        retourner vrai
    SINON
        Retourner faux
    FIN SI
FIN FONCTION
```

- Périmètre:

Le périmètre du polygone est calculé en parcourant chaque point du polygone, en utilisant une boucle for qui parcourt tous les éléments de P[0] à partir de l'indice 1 jusqu'à la

fin de ce vecteur. Pour chaque point, le périmètre est calculé en utilisant la formule suivante :

Périmètre = côté_1 + côté_2 + ... + côté_n

"côté_i" est la longueur du i-ème côté du polygone, qui peut être calculé en utilisant la formule de la `distance entre deux points.`

```
FONCTION perimeter(P)
    per=0.0
    POUR i de 1 à la taille de P[0] FAIRE
        per += racine_carré((P[0][i] - P[0][i-1])^2 + puissance(P[1][i] - P[1][i-1])^2)
    FIN POUR
FIN FONCTION
```

En effet :

pow(P[0][i]-P[0][i-1], 2) : cette fonction calcule la différence entre les abscisses du point courant et du point précédent, et la met au carré.

pow(P[1][i]-P[1][i-1], 2) : cette fonction calcule la différence entre les ordonnées du point courant et du point précédent, et la met au carré.

pow(pow(P[0][i]-P[0][i-1], 2) + pow(P[1][i]-P[1][i-1], 2) , 0.5) : cette fonction calcule la racine carrée de la somme des carrés des différences entre les abscisses et les ordonnées du point courant et du point précédent. Cette formule correspond à la formule de la distance euclidienne entre deux points ((x2 - x1)^2 + (y2 - y1)^2).

- **Fonction Régulier**

Elle sert à vérifier si un polygone est régulier ou non. Un polygone est dit régulier s'il a tous ses côtés de même longueur et tous ses angles intérieurs de même mesure.
La fonction parcourt chaque côté du polygone à l'aide d'une boucle for qui démarre à 1 et s'arrête à n-1. Pour chaque itération de la boucle, elle calcule les coordonnées des points formant le côté et les points précédant et suivant le côté. Elle utilise ces coordonnées pour calculer la longueur des côtés précédant et suivant et l'angle formé par ces deux côtés.

La fonction utilise la formule de produit scalaire pour calculer le produit scalaire des vecteurs formés par les côtés précédant et suivant cela sert à calculer l'angle entre les deux vecteurs, elle utilise également la formule de la racine carrée pour calculer la magnitude (longueur) de chaque vecteur. Enfin, elle utilise la fonction *acos* de la bibliothèque mathématique cmath pour calculer l'angle formé par les deux vecteurs en utilisant la formule *ang = acos(prodP/(mag1*mag2))*.

```
FONCTION regular(P)
    n =P[0] – 1
    POUR i allant de 1 à n-1 FAIRE
        x1 <- P[0][i-1] - P[0][i]
        x2 <- P[0][i+1] - P[0][i]
        y1 <- P[1][i-1] - P[1][i]
        y2 <- P[1][i+1] - P[1][i]
        prodP <- x1 * x2 + y1 * y2
        mag1 <- racine_carrée(puissance(x1, 2) + puissance(y1, 2))
        mag2 <- racine_carrée(puissance(x2, 2) + puissance(y2, 2))
        ang <- acos(prodP / (mag1 * mag2))
        SI mag1 n'est pas égal à mag2 ALORS
            renvoyer FAUX
        FIN SI
        SI i est différent de 1 ALORS
            SI ang n'est pas égal à ang0 ALORS
                renvoyer FAUX
            SINON
                ang0 <- ang
            FIN SI
        SINON
            ang0 <- ang
        FIN SI
```

```
    FIN POUR
FIN FONCTION
```

- **Fonction isConvex**

La fonction isConvex vérifie si un polygone est convexe en utilisant le produit scalaire de deux vecteurs formés par trois points consécutifs du polygone. Si le produit scalaire est négatif, cela signifie que l'angle entre les deux vecteurs est supérieur à 180 degrés, ce qui signifie que le polygone est non convexe. Si le produit scalaire est positif, cela signifie que l'angle est inférieur à 180 degrés et que le polygone est convexe. Si le produit scalaire est égal à zéro, cela signifie que les deux vecteurs sont collinéaires et qu'il n'y a pas d'angle entre eux. La fonction isConvex parcourt tous les points du polygone et vérifie si le produit scalaire est positif ou négatif. Si le produit scalaire est à la fois positif et négatif, cela signifie que le polygone est non convexe et la fonction renvoie false. Sinon, la fonction renvoie true et indique que le polygone est convexe.

```
fonction isConvex(P: vecteur de vecteurs de doubles) -> booléen:
    n <- taille de P[0] - 1
    cp <- 0
    neg <- faux
    pos <- faux

    Pour k allant de 0 à n-1:
        ax <- P[0][k]
        bx <- P[0][k+1]
        cx <- P[0][k+2]
        ay <- P[1][k]
        by <- P[1][k+1]
        cy <- P[1][k+2]
```

```
        cp <- produit vectoriel (ax, ay, bx, by, cx, cy)
        Si cp < 0:
            neg <- vrai
        Sinon si cp > 0:
            pos <- vrai
        Si neg et pos sont vrais:
            Afficher "Polygone non convexe"
            retourner faux
    Afficher "Polygone convexe"
    retourner vrai
```

- isSimple() :

Un polygone est simple si l'intersection de deux côtés non adjacents est vide. Par définition, l'intersection est réduite à un point pour deux côtés adjacents. Dans le cas des polygones simples, on distingue facilement un intérieur et un extérieur.
On vérifie si un polygone est simple en vérifiant s'il y a des intersections entre ses côtés non adjacents. Si aucune intersection n'est détectée, alors le polygone est simple. Si, en revanche, au moins une intersection est détectée, alors le polygone est non simple.
La fonction appelle alors la fonction determinant avec les coordonnées x et y des deux sommets en argument. Si la fonction determinant renvoie 0, cela signifie que les deux sommets ne sont pas situés sur des droites distinctes, donc les deux côtés du polygone ne s'intersectent pas. Dans ce cas, la variable intersection est mise à false. Si la fonction determinant renvoie une valeur non nulle, cela signifie que les deux sommets sont situés sur des droites distinctes, donc les deux côtés du polygone s'intersectent. Dans ce cas, la variable intersection est mise à true.

```
fonction bool isSimple(vecteur <vecteur<double>> P):
    bool intersection
     n = taille de P[0] - 1
    pour k de 0 à n-2:
```

```
        ax = P[0][k]
        bx = P[0][k+2]
        ay = P[1][k]
        by = P[1][k+2]
      si (determinant (ax, ay, bx, by) == 0):
        intersection = faux
      sinon:
        intersection = vrai
  si (!intersection):
    afficher "Le polygone est simple"
  sinon:
    afficher "Le polygone n'est pas simple"
  retourner intersection
```

- Area

La fonction calcule l'aire du polygone en utilisant la formule du lacet, qui consiste à additionner les produits des coordonnées x et y de chaque paire de sommets adjacents, puis à prendre la valeur absolue de la moitié de la différence entre la somme des produits des coordonnées x et la somme des produits des coordonnées y.

Dans la formule area= abs((a-b)/2);, les variables a et b représentent les sommes des produits des coordonnées x et y des sommets du polygone, respectivement. La différence entre a et b est calculée en soustrayant b à a, puis la valeur absolue de la moitié de cette différence est prise en utilisant la fonction abs. Le résultat est affecté à la variable area, qui représente l'aire du polygone.

```
Area(vector <vector<double>> P) : double
  double area = 0
  entier n = taille(P[0]) - 1
  pour entier i de 0 à n-1 faire
```

```
        area += P[0][i] * P[1][i+1] - P[1][i] * P[0][i+1]
    fin pour
    area += P[0][n] * P[1][0] - P[1][n] * P[0][0]
    retourner abs(area / 2)
fin algorithme
```

1.3. Deuxième partie

Dans cette seconde partie, nous créons la classe **polygon**, du fichier **polygon.cpp.** La classe a plusieurs méthodes, y compris des fonctions d'accès et de mutateur pour les membres (attributs) de données privés, des fonctions pour charger et nettoyer les données, une fonction pour vérifier si deux polygones sont identiques, une fonction pour calculer le périmètre d'un polygone, une fonction pour vérifier si un point est à l'intérieur ou à l'extérieur d'un polygone, une fonction pour calculer la superficie d'un polygone, une fonction pour vérifier si un polygone est simple et une fonction pour vérifier si un polygone est convexe.

En déclarant n, X, et Y comme des membres de données privés de la classe Polygon, ils ne peuvent être directement modifiés ou accédés par des classes ou des fonctions extérieures à la classe. Pour accéder ou modifier ces champs, l'utilisateur doit utiliser les fonctions d'accès (aussi appelées "getters") et de modification (aussi appelées "setters") qui ont été définies dans la classe. Cela permet de contrôler l'accès et la modification de ces champs et de protéger la donnée sous-jacente contre les modifications non autorisées ou des erreurs de programmation. Cela est connu sous le nom de "encapsulation".

Ensuite, nous créons des constructeurs. Les constructeurs sont des fonctions spéciales qui sont utilisées pour créer et initialiser des objets de la classe. Le constructeur avec arguments permet de créer un objet Polygon en spécifiant les valeurs de ses membres de données n, X, et Y. Le constructeur sans arguments, appelé constructeur par défaut, permet de créer un objet Polygon en utilisant des valeurs par défaut pour ses membres de données. Dans ce cas, le membre de données n est initialisé à 0.

En séparant la déclaration de la classe de son implémentation dans deux fichiers distincts, on peut utiliser la classe dans plusieurs programmes en incluant simplement le fichier d'en-tête **polygon.h** qui contient la déclaration de la classe. Le fichier polygon.cpp contient l'implémentation des fonctions de la classe, c'est-à-dire le code qui définit ce que font ces fonctions. Pour utiliser la classe Polygon dans un programme, nous incluons le fichier d'en-tête et ajoutons les deux fichiers au projet.

1.4. Conclusion

Pour conclure, nous avons utilisé nos compétences en programmation C++, nous avons appris à utiliser les classes et les objets C++ pour organiser notre code de manière plus efficace et modulaire. L'utilisation des fichiers .h et .cpp nous a aidé à séparer la définition de notre classe. Cette approche permet également de protéger les données et cela peut aider à prévenir les erreurs et les problèmes de programmation plus tard dans le développement.

Nous avons découvert différentes fonctionnalités de la géométrie algorithmique, comme lire et écrire des données de polygones à partir de fichiers, nettoyer des polygones en enlevant les points superflus, vérifier si deux polygones sont identiques, calculer le périmètre et l'aire d'un polygon, etc.. C'est une compétence précieuse dans de nombreux domaines, comme la robotique, la vision par ordinateur et la géométrie 3D, etc.

Partie 2 - GUSEK

2.1. Problème 1 – Production Et Distribution De Ciment

Ce problème consiste à la production et le stockage de ciment, dans ce cas nous considérons avoir une usine, où nous produisons et stockons le ciment, et un centre de distribution où il peut être stocké et il peut être vendu. Il est important de noter qu'il existe des restrictions sur la capacité de production, la capacité de stockage et ouverture du centre de distribution. L'horizon de planification est généré pour 2 semaines et sera développé comme un modèle mathématique linéaire et résolu dans le logiciel GUSEK.

2.1.1. Les paramètres et les variables

Les données sont obtenues à partir du problème et résumées ci-dessous, il est possible de montrer rapidement que, étant donné qu'il n'y a pas de ventes le week-end, et que notre objectif sera de minimiser les coûts, nous pouvons réduire les deux derniers jours de notre horizon de planification de 14 à 12 jours.

1) Sets
   - ✓ ***T*** : Set de jours dans la planification, $T=\{1..n\}$
   - ✓ ***T′*** : Set de jours T sans le premier jour, $T'=\{2..n\}$
   - ✓ ***W*** : Set de jours T qui sont dans le weekends, $W=\{6,7\}$

2) Paramètres
   - ✓ ***n*** : nombre de jour = 12
   - ✓ ***C*** : Coût de production = 150 €/tonne
   - ✓ ***CSU*** : Coût de stock dans l'usine = 20 €/jour
   - ✓ ***CSCD*** : Coût de stockage dans le CD = 20 €/jour
   - ✓ ***CT*** : Coût de transport d'usine à CD = 30 €/tonne
   - ✓ ***CFT*** : Coût fixé si on utilise chaque camion = 500 €/jour
   - ✓ ***CAT*** : Coût additionnel pour utiliser plusieurs camions = 700 €/(jour*camion)
   - ✓ ***K*** : Capacité de production = 260 tonnes/jour
   - ✓ ***KSU*** : Capacité de stock dans l'usine = 350 tonnes
   - ✓ ***KSCD*** : Capacité de stockage dans le CD = 300 tonnes
   - ✓ ***Kcam*** : Capacité de chaque camion = 35 tonnes
   - ✓ ***cam*** : Quantité de camions disponible = 3 camions

- ✓ ***SIU*** : Stock initial dans l'usine = 50 tonnes
- ✓ ***SICD*** : Stock initial dans le CD = 20 tonnes
- ✓ $\boldsymbol{D_t}$: Demande de jour *t*

3) Variables
   - ✓ $\boldsymbol{X_t}$: Quantité à faire en le jour *t*
   - ✓ $\boldsymbol{XTR_t}$: Quantité transporté de l'usine à le CD en le jour *t*
   - ✓ $\boldsymbol{SU_t}$: Stock dans l'usine à la fin de jour *t*
   - ✓ $\boldsymbol{SCD_t}$: Stock dans le CD à la fin de jour *t*
   - ✓ $\boldsymbol{V_t}$: Quantité de véhicules utilisé en le jour *t*
   - ✓ $\boldsymbol{VA_t}$: Quantité de véhicules qu'on doit louer en le jour *t*

2.1.2. L'objectif

L'objectif de ce problème est de minimiser le coût total, en général, les coûts sont associés aux coûts de production, de transport et de stockage.

$$Min\ Z = \sum_{t \in T} (C * X_t + CSU * SU_t + CSCD * SCD_t + CT * XTR_t + CFT * V_t + CAT * VA_t) \quad (1)$$

2.1.3. Les contraintes

Les contraintes générées pour respecter les limites du système proposé sont présentées ci-dessous. Les équations *(2)*, *(3)*, *(4)* et *(5)* représentent les limites de la capacité de production, de stockage dans l'usine, de stockage dans le centre de distribution et de chaque camion, respectivement. En revanche, l'équation *(6)* nous permet d'augmenter le nombre de camions dont nous disposons si nous les louons.

Ensuite, nous trouvons les équations du flux de stockage, en générant les équations *(7)* et *(9)* pour la première période, dans laquelle nous devons tenir compte du stock initial de l'usine et du centre de distribution, puis en utilisant *(8)* et *(10)* pour les périodes suivantes. En outre, l'équation *(11)* est générée afin de représenter le fait que le centre de distribution n'est pas ouvert le week-end, la quantité qui y est transportée doit donc être 0. Enfin, l'équation *(12)* est la déclaration des variables, les quantités étant des nombres réels positifs et le nombre de

véhicules des nombres entiers positifs.

$$X_t \leq K, \qquad \forall t \in T \tag{2}$$

$$SU_t \leq KSU, \qquad \forall t \in T \tag{3}$$

$$SCD_t \leq KSCD, \qquad \forall t \in T \tag{4}$$

$$XTR_t \leq Kcam * V_t, \qquad \forall t \in T \tag{5}$$

$$V_t \leq cam + VA_t, \qquad \forall t \in T \tag{6}$$

$$SU_1 = SIU + X_1 - XTR_1 \tag{7}$$

$$SU_t = SU_{t-1} + X_t - XTR_t, \qquad \forall t \in T' \tag{8}$$

$$SCD_1 = SICD + XTR_1 - D_1 \tag{9}$$

$$SCD_t = SCD_{t-1} + XTR_t - D_t, \qquad \forall t \in T' \tag{10}$$

$$XTR_t = 0, \qquad \forall t \in W \tag{11}$$

$$X_t, XTR_t, SU_t, SCD_t \in R^+, \qquad V_t, VA_t \in Z^+ \tag{12}$$

2.1.4. Analyse des résultats

Après avoir modélisé et résolu ce problème dans le GUSEK, les résultats sont obtenus comme ci-dessous :

Processeur : **Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz**

Temps de calcul ≈ **0.0 s**

Coût total : **468200 €**

Les plans détaillés de la production, du transport et du stockage sont présentés dans le tableau 1 comme ci-dessous. Il est possible de voir que la production de ciment s'arrête le samedi, mais reprend le dimanche, même s'il n'est pas possible de transporter le ciment vers le centre de distribution ces jours-là. En outre, il y a une grande différence entre le stockage à l'usine et au centre de distribution. Étant donné le coût élevé du stockage du ciment au centre de distribution, il est préférable de transporter le plus grand nombre de jours possible, quitte à louer des véhicules supplémentaires pour répondre à la demande de transport.

t	Xt	XTRt	SUt	SCDt	Vt	VAt
1	210	95	165	15	3	0
2	260	245	180	10	7	4
3	260	280	160	10	8	5
4	260	420	0	0	12	9
5	140	140	0	0	4	1
6	0	0	0	0	0	0
7	65	0	65	0	0	0
8	260	170	155	0	5	2
9	260	415	0	5	12	9
10	105	105	0	0	3	0
11	240	240	0	0	7	4
12	130	130	0	0	4	1

Tableau 1 : Plan de production du ciment

2.2. Problème 2 – Approvisionnement En Biomasse D'une Centrale Electrique

Ce problème consiste en la production d'énergie à partir de la biomasse qui doit être achetée et transportée jusqu'à notre centrale électrique. Pour ce problème, nous devons satisfaire la demande d'énergie de notre région et, en outre, nous devons respecter les conditions de chacun de nos fournisseurs, en ayant 2 types de biomasse différents, qui sont fournis à des prix différents selon le fournisseur, et qui ont également une efficacité différente. En outre, nous devons tenir compte de la distance entre chaque fournisseur et notre centrale électrique. En bref, nous devons minimiser les coûts d'achat, de transport, de main-d'œuvre et de stockage.

2.2.1. Les paramètres et les variables

Les données sont obtenues à partir du problème et résumées ci-dessous. Il est possible de réduire l'ensemble des fournisseurs de 4 à 2 s'ils sont classés en fonction du type de biomasse qu'ils produisent. Il est important de noter qu'avec ce changement, lorsqu'on se réfère à un certain fournisseur, il doit toujours être accompagné de sa biomasse respective afin de le différencier, par exemple, le fournisseur 1 pour la biomasse bois est différent du fournisseur 1 pour la biomasse paille.

1) Sets
   - ✓ $\boldsymbol{T}$: Set de jours dans la planification, $T = \{1..n\}$
   - ✓ $\boldsymbol{T'}$: Set de jours T sans le premier jour, $T' = \{2..n\}$
   - ✓ $\boldsymbol{B}$: Set de biomasses, B = {bois, paille}
   - ✓ $\boldsymbol{F}$: Set de fournisseurs, F = {1, 2}

2) Paramètres
   - ✓ $\boldsymbol{n}$: nombre de jours = 8
   - ✓ $\boldsymbol{D_t}$: Demande d'électricité dans le jour t (MWh)
   - ✓ $\boldsymbol{TP_b}$: Taux de production d'électricité de biomasse b (MWh/tonne)
   - ✓ $\boldsymbol{SD_{bf}}$: Stock disponible de biomasse b en fournisseur f (tonnes)

- ✓ P_{bf} : Prix d'achat de biomasse b en fournisseur f (€/tonne)
- ✓ DI_{bf} : Distance entre centrale et fournisseur f de biomasse b (km)
- ✓ KC : Capacité du camion (tonnes)
- ✓ VC : Vitesse moyenne du camion (km/h)
- ✓ KH : Capacité de travail du conducteur (h)
- ✓ CH : Coût du conducteur (€/h)
- ✓ CT : Coût de transport (€/km)
- ✓ KV : Quantité maximale de voyages par jour
- ✓ TA : Temps d'attendre au fournisseur pour chaque voyage (min)
- ✓ TD : Temps de chargement pour chaque voyage (min)
- ✓ KS : Capacité de stockage par chaque biomasse dans la centrale (tonnes)
- ✓ SI_b : Stock Initial de biomasse b (tonnes)
- ✓ CS_b : Coût de stockage de biomasse b (€/(tonnes*jour))

3) Variables
   - ✓ X_{bft} : Quantité acheté de biomasse *b* au fournisseur *f* dans le jour *t*
   - ✓ Y_{bt} : Quantité de biomasse *b* brûlées dans le jour *t*
   - ✓ S_{bt} : Stock de biomasse *b* dans le jour *t*
   - ✓ V_{bft} : Quantité de voyages pour biomasse *b* au fournisseur *f* dans le jour *t*
   - ✓ H_t : Quantité de heures travaillées pour le conducteur dans le jour *t*

2.2.2. L'objectif

L'objectif de ce problème est de minimiser le coût total, en général, les coûts sont associés aux coûts d'achat, de transport, de main-d'œuvre et de stockage.

$$Min\ Z = \sum_{b\in B}\sum_{f\in F}\sum_{t\in T}\left(P_{bf} * X_{bft} + 2 * CT * DI_{bf} * V_{bft}\right) + \sum_{b\in B}\sum_{t\in T}(CS_b * S_{bt}) + \sum_{t\in T}(CH * H_t) \quad (13)$$

2.2.3. Les contraintes

Les contraintes générées pour ce problème sont présentées ci-dessous. L'équation *(14)* permet de satisfaire la demande d'électricité, l'équation *(15)* limite la quantité à acheter à

chaque fournisseur en fonction de sa capacité. L'équation *(16)* définit le stockage maximal pour chaque type de biomasse. L'équation *(17)* nous permet de calculer le nombre de voyages nécessaires pour transporter la quantité de biomasse requise, et les équations *(18)* et *(19)* limitent le nombre maximal de voyages et d'heures travaillées par jour. En outre, la contrainte selon laquelle le conducteur ne travaille pas ce jour-là est satisfaite par l'équation *(20)*. Enfin, les équations *(22)* et *(23)* maintiennent le flux de stockage de la biomasse.

L'équation *(21)* calcule le nombre d'heures travaillées par le conducteur, cette restriction est laissée comme supérieure ou égale étant donné que 2 scénarios sont présentés avec les équations *(24)* et *(25)*, où, dans le premier cas (S1), le paiement au travailleur est calculé en fonction du temps travaillé, et dans le second cas (S2) une interprétation est faite que le travailleur est payé pour des heures complètes. Par exemple, si le conducteur a travaillé *5,1 h*, dans le premier cas, il sera payé exactement *5,1 h*, alors que dans le second cas, il devrait être payé pour *6 h*. Si l'on décide de conserver le premier scénario (équation *(24)*), l'équation *(21)* peut être modifiée avec égalité ou non, dans les deux cas elle fonctionne.

$$\sum_{b \in B} (TP_b * Y_{bt}) = D_t, \qquad \forall t \in T \tag{14}$$

$$\sum_{t \in T} X_{bft} \leq SD_{bf}, \qquad \forall b \in B, \forall f \in F \tag{15}$$

$$S_{bt} \leq KS, \qquad \forall b \in B, \forall t \in T \tag{16}$$

$$X_{bft} \leq KC * V_{bft}, \qquad \forall b \in B, \forall f \in F, \forall t \in T \tag{17}$$

$$\sum_{b \in B} \sum_{f \in F} V_{bft} \leq KV, \qquad \forall t \in T \tag{18}$$

$$H_t \leq KH, \qquad \forall t \in T \tag{19}$$

$$H_5 = 0 \tag{20}$$

$$H_t \geq \sum_{b \in B} \sum_{f \in F} \left(\frac{2}{VC} DI_{bf} + \frac{TA + TD}{60} \right) V_{bft}, \qquad \forall t \in T \tag{21}$$

$$S_{b1} = SI_b + \sum_{f \in F} X_{bf1} - Y_{b1} \quad (22)$$

$$S_{bt} = S_{b,t-1} + \sum_{f \in F} X_{bft} - Y_{bt}, \qquad \forall t \in T' \quad (23)$$

$$X_{bft}, Y_{bt}, S_{bt}, H_t \in R^+, \qquad V_{bft} \in Z^+ \quad (24)$$

$$X_{bft}, Y_{bt}, S_{bt} \in R^+, \qquad V_{bft}, H_t \in Z^+ \quad (25)$$

2.2.4. Analyse des résultats

Après avoir modélisé et résolu les deux scénarios dans GUSEK, les résultats sont obtenus comme ci-dessous, il est évident que la variation du coût entre les deux scénarios est d'environ 46 euros, ce qui, en pourcentage, représente une augmentation du coût de 0,1%. Mais nous pouvons également constater que GUSEK génère la solution 43 % plus rapidement.

Processeur : **Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz**

Temps de calcul S1 : **352.2 s**

Coût total S1 : **26483.1666666667 €**

Temps de calcul S1 : **199.2 s**

Coût total S2 : **26529.5 €**

Les plans d'achats, de production, de stockage et de transport sont présentés dans le tableau 2, 3, 4, 5 et 6 comme ci-dessous. Nous pouvons clairement voir, comment la quantité initiale de biomasse est suffisante pour satisfaire la demande générée pour le premier jour, puisqu'il n'est pas nécessaire d'acheter pour le premier jour, économisant ainsi les coûts de stockage. En outre, on observe une augmentation significative de l'achat de biomasse le quatrième jour, car aucune biomasse ne peut être achetée le cinquième jour.

Biomasse	fournisseur	1	2	3	4	5	6	7	8
bois	1	0,000	0,000	20,000	60,000	0,000	0,000	0,000	0,000
bois	2	0,000	26,667	0,000	30,000	0,000	28,333	30,000	0,000

paille	1	0,000	0,000	52,500	0,000	0,000	27,500	30,000	0,000
paille	2	0,000	0,000	0,000	0,000	0,000	0,000	30,000	60,000

Tableau 2 Quantité acheté de biomasse b au fournisseur f dans le jour t (X_{bft})

Biomasse	1	2	3	4	5	6	7	8
bois	15,000	26,667	5,000	75,000	30,000	26,667	30,000	1,667
paille	10,000	10,000	22,500	0,000	30,000	27,500	60,000	60,000

Tableau 3 Quantité de biomasse b brûlées dans le jour t (Y_{bt})

Biomasse	1	2	3	4	5	6	7	8
bois	0,000	0,000	15,000	30,000	0,000	1,667	1,667	0,000
paille	10,000	0,000	30,000	30,000	0,000	0,000	0,000	0,000

Tableau 4 Stock de biomasse b dans le jour t (S_{bt})

Biomasse	fournisseur	1	2	3	4	5	6	7	8
bois	1	0	0	1	2	0	0	0	0
bois	2	0	1	0	1	0	1	1	0
paille	1	0	0	2	0	0	1	1	0
paille	2	0	0	0	0	0	0	1	2

Tableau 5 Quantité de voyages pour biomasse b au fournisseur f dans le jour t (V_{bft})

Jour	1	2	3	4	5	6	7	8
H(t)	0,000	1,524	3,600	4,000	0,000	2,705	4,086	2,762

Tableau 6 Quantité de heures travaillées pour le conducteur dans le jour t (H_t)

2.3. Conclusion

Les deux problèmes de planification de la production et du transport ont été résolus à l'aide de modèles linéaires qui peuvent être transférés à de nouveaux contextes si nécessaire. Dans le premier problème, nous avions un coût optimal de 468200 euros et dans le second de 26483.17 euros. Nous concluons que l'utilisation de modèles d'optimisation aide à la planification et à la réduction des coûts dans des systèmes complexes qu'il serait impossible de résoudre par une seule personne.