Rapport projet d'Algorithmique des graphes

Résolution du problème du voyageur de commerce

Tristan GROULT, Louis GUERAIN, Kibandou BIGNANG

4 janvier 2026

Sommaire

1 Interface utilisateur **3**

2 Notice d'utilisation **3**

3 Résultats de l'étude statistique **3**
3.1 Complexité théorique 4
3.2 Analyse des performances 4

4 Algorithme **6**
4.1 Point le Plus Proche (PPP) 6
4.2 Point le Plus Proche Optimisé (OptPPP) 11
4.3 Parcours préfixe d'un ACM à partir de Prim (OptPrim) 13
4.4 OptPrim + Décroisé (OptOptPrim) 22
4.5 Évaluation et séparation progressive ("branch and bound") (HDS) 22

1 Interface utilisateur

Notre interface utilisateur est disponible publiquement à l'adresse suivante : `https://graph.grltt.fr`

2 Notice d'utilisation

Ce projet dépend des bibliothèques `numpy` et `matplotlib` ainsi que d'autres modules standards. Il dépend aussi du module `flask` pour l'application web.

```
pip install numpy # (utilisé partout)
pip install matplotlib # (utilisé partout)
pip install python-dotenv # (pour web app)
pip install flask # (pour web app)
```

Ou plus simplement :

```
pip install -r requirement.txt
```

Pour lancer le programme en ligne de commande et tester les algorithmes, utilisez :

```
python3 pvc_main.py
```

ou lancer l'exécution de la cellule sous Pyzo. Les valeurs suivantes sont modifiables pour tester de manière différente les algorithmes implémentés.

```
number_of_tests = 30
number_of_sommet = 10
```

Il est possible de consulter les graphiques générés dans le dossier `/img`

Les algorithmes implémentés sont disponibles dans le fichier `pvc_algos.py`. Ils sont chacun disposés dans une cellule Pyzo distincte pour une meilleure lisibilité. Ces algorithmes sont détaillés dans la section suivante et leur code est aussi donné comme demandé dans le sujet du projet.

L'application web est lancée via le fichier `app.py`. Il est possible de configurer les valeurs maximales des sommets et des temps de "timeout" grâce au fichier `.env` à la racine du projet avec les variables d'environnement suivantes :

```
MAX_RANDOM_POINTS=1000
GLOBAL_TIMEOUT=30 #seconds
SECRET_KEY=your-secret-key-here-change-in-production

# Stats page limits
MAX_STATS_TESTS=100
MAX_STATS_VERTICES=200
```

Le serveur web peut être lancé avec la commande :

```
python3 app.py
```

3 Résultats de l'étude statistique

Nous avons comparé les performances des cinq algorithmes implémentés pour résoudre le problème du voyageur de commerce : PPP, OptPPP, OptPrim, OptOptPrim et HDS. L'étude porte sur le temps d'exécution et la qualité des solutions (longueur du cycle).

3.1 Complexité théorique

L'analyse des algorithmes nous permet d'établir les ordres de grandeur suivants pour un graphe de n sommets :

- **PPP (Point le Plus Proche)** : $O(n^3)$. L'insertion itérative demande de parcourir l'ensemble des points restants à chaque étape.
- **OptPPP** : Complexité de PPP augmentée par la phase de décroisement (recherche locale).
- **OptPrim** : $O(n^2 \log n)$. L'algorithme de Prim est implémenté avec un tas binaire. Sur un graphe complet ($m \approx n^2$ arêtes), le coût des mises à jour des priorités dans le tas domine, donnant une complexité de $O(m \log n)$ soit $O(n^2 \log n)$.
- **OptOptPrim** : Complexité d'OptPrim augmentée par la phase de décroisement.
- **HDS (Heuristique Déterministe Séquentielle)** : Algorithme de type *Branch and Bound*. Sa complexité est exponentielle dans le pire des cas, car il explore l'arbre des solutions possibles pour trouver l'optimum exact.

3.2 Analyse des performances

Vitesse d'exécution :

- **OptPrim** se distingue comme l'algorithme le plus rapide, cohérent avec sa complexité théorique plus faible.
- **PPP** est sensiblement plus lent que l'approche basée sur Prim.
- L'ajout de l'étape de **décroisement** (dans OptPPP et OptOptPrim) augmente naturellement le temps de calcul par rapport aux versions de base, mais permet d'affiner la solution.
- **HDS** est extrêmement coûteux en temps dès que le nombre de sommets augmente, le limitant aux petites instances, contrairement aux autres heuristiques qui passent à l'échelle.

Qualité des solutions :

- **HDS** fournit toujours la solution optimale (le cycle le plus court possible), servant de référence absolue.
- Parmi les heuristiques constructives pures, **PPP** tend à donner de meilleurs résultats initiaux que **OptPrim**.
- L'application du **décroisement** est cruciale : elle améliore considérablement la qualité des solutions. **OptOptPrim** et **OptPPP** produisent des cycles de longueurs comparables et très compétitifs.

Conclusion : L'algorithme **OptOptPrim** présente souvent le meilleur compromis global : il bénéficie de la rapidité de construction de l'arbre de Prim et de l'efficacité du décroisement pour produire d'excellentes solutions en un temps très court. HDS reste l'outil de choix uniquement lorsque l'exactitude prime sur le temps de calcul pour de petits graphes.

Exemple de statictiques obtentues :

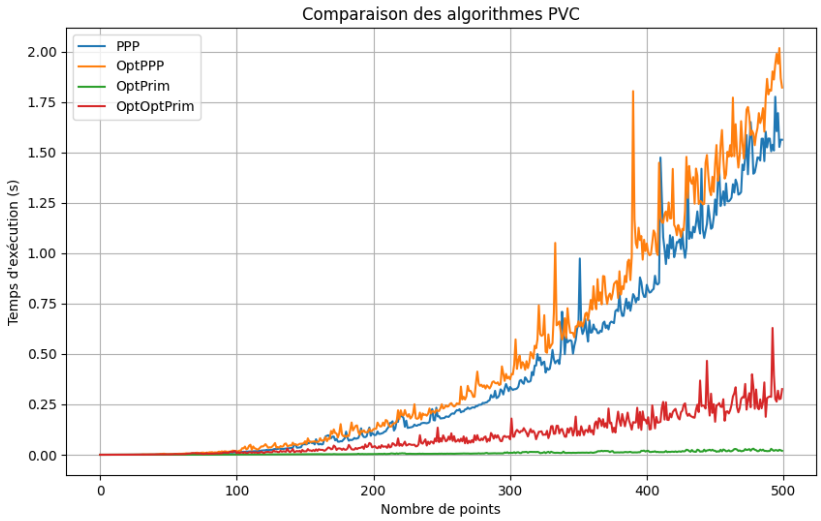

FIGURE 1 – Comparaison des temps d'exécution des algorithmes (sans HDS)

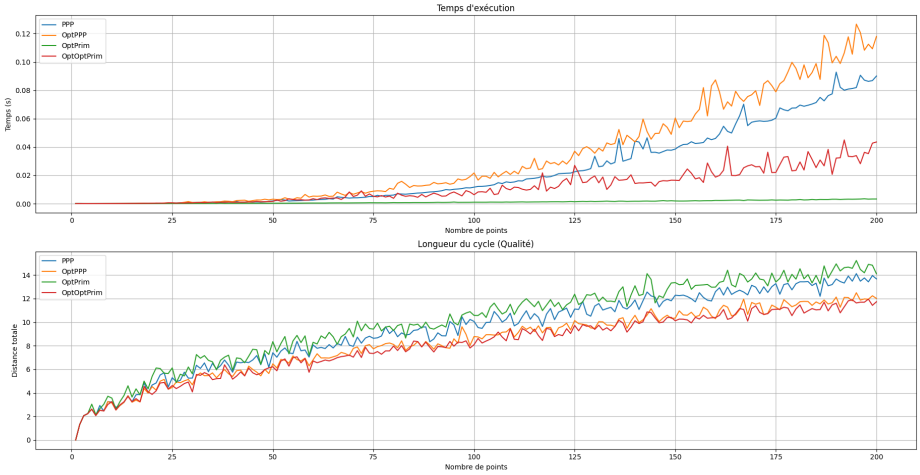

FIGURE 2 – Comparaison de la qualité des solutions (longueur des cycles)(sans HDS)

4 Algorithme

4.1 Point le Plus Proche (PPP)

Algorithme : Procédure PPP (Point le Plus Proche)
Description : Construit un cycle hamiltonien en insérant itérativement le point le plus proche du cycle actuel.
Vars :
- c : Liste d'entiers (Cycle hamiltonien)
- l : Réel (Longueur du cycle)
- Q_i, Q_j : Entiers (Indices des points sélectionnés)

Entrées :
- L : Liste de points $P_1, \ldots, P_n$
- Q : Un point de départ choisi dans L
- D : La matrice des distances $n \times n$

Sorties :
- c : Un cycle hamiltonien
- l : La longueur du cycle

Début :

1. $c \leftarrow$ cycle trivial composé du seul point Q // *Initialisation du cycle*
2. $l \leftarrow 0$
3. Retirer Q de L
4. Tant que L n'est pas vide faire : // *Boucle principale d'insertion*
5. $(Q_i, Q_j) \leftarrow$ `TrouverCoupleMin`(L, c, D) // *Trouver le couple minimisant la distance*
6. $(c, l) \leftarrow$ `InsérerDansCycle`(c, Q_i, Q_j, D, l) // *Insérer et mettre à jour*
7. Retirer Q_i de L
8. Retourner c, l

Algorithme : InsérerDansCycle
Description : Insère Q_i dans c soit entre Q_j et son successeur, soit entre son prédécesseur et Q_j, selon le coût minimal, et met à jour la longueur du cycle.
Vars :
— Q_{next}, Q_{prev} : Entiers (Indices des voisins de Q_j)
— $\text{coût}_{\text{droite}}, \text{coût}_{\text{gauche}}$: Réels (Coûts d'insertion)
Entrées :
— c : Cycle partiel
— Q_i : Point à insérer
— Q_j : Point d'ancrage dans c
— D : Matrice des distances
— l : Longueur actuelle du cycle
Sorties : c (cycle mis à jour), l (nouvelle longueur)
Début :

1. Soit Q_{next} le successeur de Q_j dans c
2. Soit Q_{prev} le prédécesseur de Q_j dans c
3. $\text{coût}_{\text{droite}} \leftarrow D[Q_j, Q_i] + D[Q_i, Q_{next}] - D[Q_j, Q_{next}]$ // *Coût insertion droite*
4. $\text{coût}_{\text{gauche}} \leftarrow D[Q_{prev}, Q_i] + D[Q_i, Q_j] - D[Q_{prev}, Q_j]$ // *Coût insertion gauche*
5. Si $\text{coût}_{\text{droite}} \leq \text{coût}_{\text{gauche}}$ alors : // *Choix du meilleur coût*
6. Insérer Q_i entre Q_j et Q_{next} dans c
7. $l \leftarrow l + \text{coût}_{\text{droite}}$
8. Sinon :
9. Insérer Q_i entre Q_{prev} et Q_j dans c
10. $l \leftarrow l + \text{coût}_{\text{gauche}}$
11. Retourner c, l

Algorithme : TrouverCoupleMin
Description : Trouve le couple (Q_i, Q_j) tel que $Q_i \in L$, $Q_j \in c$ et la distance $D[Q_i, Q_j]$ est minimale.
Vars :
— d_{min} : Réel (Distance minimale trouvée)
— u, v : Entiers (Indices de parcours)
Entrées :
— L : Liste des points non encore visités
— c : Le cycle partiel actuel
— D : La matrice des distances $n \times n$
Sorties : Q_i (point sélectionné dans L), Q_j (point d'ancrage dans c)
Début :

1. $d_{min} \leftarrow \infty$
2. $Q_i \leftarrow$ vide
3. $Q_j \leftarrow$ vide
4. Pour chaque point u appartenant à L faire : *// Parcourir les points non visités*
5. Pour chaque point v appartenant à c faire : *// Parcourir les points du cycle*
6. Si $D[u, v] < d_{min}$ alors : *// Vérifier si distance minimale*
7. $d_{min} \leftarrow D[u, v]$
8. $Q_i \leftarrow u$
9. $Q_j \leftarrow v$
10. Retourner Q_i, Q_j

Implémentation Python :

```python
def trouver_couple_min(L, c, D):
    """
    Trouve le couple (Q_i, Q_j) tel que Q_i dans L, Q_j dans c et D[Q_i,
    Q_j] est minimale.
    Args:
        L (list): Liste des indices des points non encore visités
        c (list): Liste des indices du cycle partiel actuel
        D (np.ndarray): Matrice des distances n x n
    Returns:
        tuple: (Q_i, Q_j) indices
    """
    # Initialisation des variables pour suivre la distance minimale et
   les indices correspondants
    d_min = float('inf')
    Q_i = None
    Q_j = None
    # Parcourir tous les couples (u, v) avec u dans L et v dans c
    for u in L:
        for v in c:
            # Vérifier si la distance D[u, v] est inférieure à la
   distance minimale actuelle
            if D[u, v] < d_min:
                # Mettre à jour la distance minimale et les indices
   correspondants
                d_min = D[u, v]
```

```
                Q_i = u
                Q_j = v

    return Q_i, Q_j

def inserer_dans_cycle(c, Q_i, Q_j, D, l):
    """
    Insère Q_i dans c entre Q_j et son point adjacent, met à jour la
   longueur du cycle.
    Args:
        c (list): Liste des indices du cycle partiel
        Q_i (int): Indice du point à insérer
        Q_j (int): Indice du point d'ancrage dans c
        D (np.ndarray): Matrice des distances
        l (float): Longueur actuelle du cycle
    Returns:
        tuple: (c, l) cycle mis à jour et nouvelle longueur
    """
    # Taille du cycle
    n = len(c)
    # Index de Q_j dans le cycle
    idx_j = c.index(Q_j)

    # Insertion entre Q_j et le suivant (j+1)
    idx_next = (idx_j + 1) % n
    Q_next = c[idx_next]
    cout_droite = D[Q_j, Q_i] + D[Q_i, Q_next] - D[Q_j, Q_next]

    # Insertion entre le précédent (j-1) et Q_j
    idx_prev = (idx_j - 1) % n
    Q_prev = c[idx_prev]
    cout_gauche = D[Q_prev, Q_i] + D[Q_i, Q_j] - D[Q_prev, Q_j]

    # Choix du meilleur cout d'insertion
    if cout_droite <= cout_gauche:
        c.insert(idx_j + 1, Q_i)
        l += cout_droite
    else:
        c.insert(idx_j, Q_i)
        l += cout_gauche

    return c, l

def ppp(D, Q=0):
    """
    Procédure PPP : construit un cycle hamiltonien en insérant ité
   rativement le point le plus proche du cycle actuel.
    Args:
        D (np.ndarray): Matrice des distances.
        Q (int, optional): Indice du point de départ. Si None, prend 0.
    Returns:
        tuple: (c, l) cycle hamiltonien (liste d'indices), longueur du
   cycle
    """
    # taille du graphe
    n = len(D)
    if n == 0:
        return [], 0.0
```

```
    # Liste des points non encore visités
    L = list(range(n))
    # Initialisation du cycle avec le point de départ Q
    c = [Q]
    # Longueur initiale du cycle
    l = 0.0
    # Retirer Q de la liste des points non visités
    L.remove(Q)

    # Boucle principale d'insertion tant qu'il reste des points à insé
rer
    while L != []:
        # trouver le couple (Q_i, Q_j) minimisant la distance
        Q_i, Q_j = trouver_couple_min(L, c, D)
        # insérer Q_i dans le cycle c et mettre à jour la longueur
        c, l = inserer_dans_cycle(c, Q_i, Q_j, D, l)
        # retirer Q_i de L
        L.remove(Q_i)

    return c, l
```

Listing 1 – Implémentation de PPP

4.2 Point le Plus Proche Optimisé (OptPPP)

Algorithme : Procédure Décroise
Description : Améliore un cycle en décroisant les arêtes tant que cela réduit la distance totale.
Vars :
- n : Entier (Nombre de points)
- amélioration : Booléen (Indicateur de modification)
- i, j : Entiers (Indices de boucle)
- u, v, x, y : Entiers (Indices des points des arêtes)
- $d_{actuelle}, d_{nouvelle}$: Réels (Distances comparées)
- p, q : Entiers (Indices pour l'inversion)

Entrées :
- c : Cycle hamiltonien initial
- D : Matrice des distances

Sorties : c (cycle optimisé), l (longueur)
Début :

1. $n \leftarrow$ taille(c)
2. amélioration $\leftarrow$ Vrai
3. Tant que amélioration faire : *// Boucle jusqu'à stabilité*
4. amélioration $\leftarrow$ Faux
5. Pour i allant de 1 à $n-2$ faire : *// Parcourir les paires d'arêtes*
6. Pour j allant de $i+2$ à n faire :
7. $u \leftarrow c[i]$
8. $v \leftarrow c[i+1]$
9. $x \leftarrow c[j]$
10. Si $j = n$ alors :
11. $y \leftarrow c[1]$
12. Sinon :
13. $y \leftarrow c[j+1]$
14. $d_{actuelle} \leftarrow D[u, v] + D[x, y]$
15. $d_{nouvelle} \leftarrow D[u, x] + D[v, y]$
16. Si $d_{nouvelle} < d_{actuelle}$ alors : *// Si amélioration trouvée*
17. $c[i+1 \dots j] \leftarrow$ inverser$(c[i+1 \dots j])$ *// Inversion sous-séquence*
18. amélioration $\leftarrow$ Vrai
19. $l \leftarrow$ CalculerLongueur(c, D) *// Recalcul longueur*
20. Retourner c, l

Algorithme : Procédure OptPPP
Description : Wrapper pour l'algorithme de décroisement.
Vars : Aucune
Entrées :
— c : Cycle hamiltonien initial
— D : Matrice des distances
Sorties : c (cycle optimisé), l (longueur)
Début :
1. Retourner `Décroise`(c, D)

Implémentation Python :

```python
def decroise(c, D):
    """
    Améliore le cycle en décroisant les arêtes.
    Args:
        c (list): Cycle hamiltonien (liste d'indices)
        D (np.ndarray): Matrice des distances
    Returns:
        tuple: (c, l) cycle optimisé et sa nouvelle longueur
    """
    # taille du cycle
    n = len(c)
    amelioration = True
    # Boucle jusqu'à ce qu'il n'y ait plus d'amélioration
    while amelioration:
        amelioration = False
        # Parcourir toutes les paires d'arêtes non adjacentes
        for i in range(n - 2):
            # j doit être au moins i+2 pour éviter les arêtes adjacentes
            for j in range(i + 2, n):
                # u et v sont les extrémités de la première arête
                u = c[i]
                v = c[i + 1]
                # x et y sont les extrémités de la deuxième arête
                x = c[j]
                y = c[(j + 1) % n]
                # calcul des distances actuelles et nouvelles
                d_act = D[u, v] + D[x, y]
                d_nouv = D[u, x] + D[v, y]
                # Si le décroisement améliore la longueur du cycle
                if d_nouv < d_act:
                    # Inversion de la sous-séquence entre i+1 et j
                    c[i+1:j+1] = c[i+1:j+1][::-1]
                    amelioration = True

    # Recalcul de la longueur totale du cycle optimisé
    l = 0.0
    for k in range(n):
        l += D[c[k], c[(k + 1) % n]]

    return c, l

def optppp(D, r0=0):
    """
    Procédure OptPPP : améliore le cycle fourni par PPP en décroisant
```

```
    les arêtes.
    Args:
        D (np.ndarray): Matrice des distances
        r0 (int, optional): Sommet de départ. Par défaut 0.
    Returns:
        tuple: (c, l) cycle optimisé et sa nouvelle longueur
    """
    # Obtenir le cycle initial via PPP
    c, _ = ppp(D, r0)
    # Décroisement
    return decroise(c, D)
```

Listing 2 – Implémentation de OptPPP

4.3 Parcours préfixe d'un ACM à partir de Prim (OptPrim)

Structure de données : GraphePrim
Type GraphePrim = **Enregistrement**
cle : Tableau[1..n] de Réel ;
pred : Tableau[1..n] d'Entier ;
NoeudTas : Tableau[1..n] d'Entier
Tas : Tableau[1..n] d'Entier
Fin ;

Algorithme : Prim (Optimisé)
Description : Construit un arbre couvrant de poids minimum (MST) en utilisant un tas binaire pour sélectionner efficacement le prochain sommet le plus proche.
Vars :
— $GraphePrim$: Structure GraphePrim (Données du graphe et du tas)
— $pres$: Tableau de booléens (Indique si un sommet est dans le tas)
— u, v : Entiers (Indices de sommets)
Entrées :
— D : Matrice des distances $n \times n$
— $r0$: Sommet de départ (par défaut 1)
Sorties :
— $pred$: Liste des prédécesseurs définissant l'arbre couvrant
Début :

1. Initialiser $GraphePrim$ avec n sommets
2. $pres \leftarrow$ Tableau de booléens initialisé à `Vrai` (indique si un sommet est dans le tas)
3. $GraphePrim.cle[r0] \leftarrow 0$ // *Initialiser clé départ*
4. $GraphePrim.echanger(1, GraphePrim.noeudTas[r0])$ // *Placer r0 à la racine*
5. Tant que $GraphePrim.tas$ n'est pas vide faire : // *Boucle principale*
6. $u \leftarrow GraphePrim.tas[1]$ // *Extraire le min*
7. $pres[u] \leftarrow$ `Faux` // *Marquer comme visité*
8. $GraphePrim.echanger(1, \text{taille}(tas))$
9. Retirer le dernier élément de $GraphePrim.tas$
10. Si $GraphePrim.tas$ non vide alors :
11. $GraphePrim.versLeBas(1)$ // *Réajuster le tas*
12. Pour chaque sommet v de 1 à n faire : // *Mettre à jour voisins*
13. Si $pres[v]$ et $D[u, v] < GraphePrim.cle[v]$ alors :
14. $GraphePrim.cle[v] \leftarrow D[u, v]$
15. $GraphePrim.pred[v] \leftarrow u$
16. $GraphePrim.versLeHaut(GraphePrim.noeudTas[v])$ // *Remonter dans le tas*
17. Retourner $GraphePrim.pred$

Algorithme : PredToTree
Description : Convertit la liste des prédécesseurs en une structure d'arbre (dictionnaire d'adjacence).
Vars :
— d : Dictionnaire (Liste d'adjacence)
— $racine$: Entier (Indice de la racine)
— i : Entier (Indice de boucle)
Entrées :
— $pred$: Liste des prédécesseurs
Sorties : d (arbre), $racine$
Début :

1. $d \leftarrow$ Dictionnaire vide
2. $racine \leftarrow -1$
3. Pour i allant de 1 à taille$(pred)$ faire : // *Initialisation*
4. $\quad d[i] \leftarrow$ Liste vide
5. $\quad$ Si $pred[i]$ est `None` ou -1 alors : // *Trouver racine*
6. $\quad\quad racine \leftarrow i$
7. Pour i allant de 1 à taille$(pred)$ faire : // *Remplir adjacence*
8. $\quad$ Si $pred[i]$ n'est pas `None` et $pred[i] \neq -1$ alors :
9. $\quad\quad$ Ajouter i à $d[pred[i]]$ // *Ajouter enfant*
10. Retourner $d, racine$

Algorithme : ParcoursPrefixe
Description : Effectue un parcours en profondeur (DFS) pour obtenir l'ordre de visite des sommets.
Vars :
- $stack$: Pile d'entiers
- p : Tableau d'entiers (Marquage visite)
- p_etoile : Liste d'entiers (Résultat)
- $count$: Entier
- lst_voisin : Liste d'entiers
- rep : Booléen
- $voisin$: Entier

Entrées :
- d : Dictionnaire d'adjacence
- $racine$: Entier

Sorties : p_etoile (liste ordonnée)
Début :

1. $stack \leftarrow [racine]$ // *Initialisation pile*
2. $p \leftarrow$ Tableau de zéros de taille n
3. $p\_etoile \leftarrow [racine]$
4. $p[racine] \leftarrow 1$ // *Marquer racine*
5. $count \leftarrow 2$
6. Tant que $stack$ n'est pas vide faire : // *Parcours*
7. $lst_voisin \leftarrow d[$sommet$(stack)]$
8. $rep \leftarrow$ Faux
9. Pour chaque $voisin$ dans lst_voisin faire : // *Parcourir voisins*
10. Si $p[voisin] == 0$ alors : // *Si non visité*
11. Empiler $voisin$ sur $stack$
12. Ajouter $voisin$ à p_etoile
13. $p[voisin] \leftarrow count$
14. $count \leftarrow count + 1$
15. $rep \leftarrow$ Vrai
16. Sortir de la boucle Pour
17. Si non rep alors : // *Si aucun voisin non visité*
18. Dépiler de $stack$
19. Retourner p_etoile

Algorithme : Procédure OptPrim
Description : Construit un cycle hamiltonien à partir de l'arbre couvrant minimal.
Vars :
- $pred$: Liste d'entiers
- d : Dictionnaire
- $racine$: Entier
- c : Liste d'entiers
- l : Réel

Entrées :
- $points$: Liste de points
- D : Matrice des distances
- $r0$: Sommet de départ

Sorties : c, l
Début :

1. Si D est `None` alors :
2. $\quad D \leftarrow$ `CalculerMatriceDistances`$(points)$
3. $pred \leftarrow$ `Prim`$(D, r0)$ *// Arbre couvrant minimal*
4. $(d, racine) \leftarrow$ `PredToTree`$(pred)$ *// Conversion en arbre*
5. $c \leftarrow$ `ParcoursPrefixe`$(d, racine)$ *// Cycle via parcours*
6. $l \leftarrow$ `CalculerLongueur`(c, D) *// Calcul longueur*
7. Retourner c, l

Implémentation Python :

```python
class GraphePrim:
    def __init__(self, n):
        """
        Initialise la structure de données pour l'algorithme de Prim.
        Args:
            n (int): Nombre de sommets du graphe.
        """
        self.cle = [float('inf')] * n
        self.pred = [None] * n
        self.noeudTas = list(range(n))
        self.tas = list(range(n))

    def echanger(self, i, j):
        """
        Échange deux éléments dans le tas et met à jour leurs positions.
        Args:
            i (int): Indice du premier élément dans le tas.
            j (int): Indice du deuxième élément dans le tas.
        """
        temp = self.tas[i]
        self.tas[i] = self.tas[j]
        self.tas[j] = temp
        self.noeudTas[self.tas[i]] = i
        self.noeudTas[self.tas[j]] = j

    def priorite(self, i):
        """
```

```
        Retourne la priorité (clé) d'un élément à une position donnée
    dans le tas.
        Args:
            i (int): Indice dans le tas.
        Returns:
            float: La valeur de la clé associée.
        """
        return self.cle[self.tas[i]]

    def versLeHaut(self, i):
        """
        Fait remonter un élément dans le tas pour respecter la propriété
     de tas min.
        Args:
            i (int): Indice de l'élément à faire remonter.
        """
        # Remonter tant que l'élément a une priorité inférieure à son
    parent
        while i > 0 and self.priorite(i) < self.priorite((i - 1) // 2):
            # Échanger avec le parent
            self.echanger(i, (i - 1) // 2)
            # Mettre à jour l'indice pour continuer à remonter
            i = (i - 1) // 2

    def versLeBas(self, i):
        """
        Fait descendre un élément dans le tas pour respecter la propriét
    é de tas min.
        Args:
            i (int): Indice de l'élément à faire descendre.
        """
        # Indice du fils gauche
        s = 2 * i + 1
        # Descendre tant que l'élément a une priorité supérieure à l'un
    de ses enfants
        while s < len(self.tas):
            # Trouver l'enfant avec la plus petite priorité en comparant
     les deux fils et si c'est le droit
            if s + 1 < len(self.tas) and self.priorite(s + 1) < self.
    priorite(s):
                # Prendre le fils droit
                s += 1
            # Si l'élément courant a une priorité supérieure à son
    enfant
            if self.priorite(i) > self.priorite(s):
                # echanger et continuer à descendre
                self.echanger(i, s)
                # Mettre à jour l'indice courant de l'elément à
    descendre
                i = s
                # Mettre à jour l'indice avec le fils gauche
                s = 2 * i + 1
            # Sinon, la propriété de tas est respectée
            else:
                break

def prim(D, r0=0):
    """
```

```
    Algorithme de Prim pour trouver l'arbre couvrant de poids minimal.
    Args:
        D (np.ndarray): Matrice des distances
        r0 (int, optional): Sommet de départ. Par défaut 0.
    Returns:
        list: Liste des prédécesseurs (pred[i] est le parent de i dans l
   'arbre)
    """
    n = len(D)
    graphe = GraphePrim(n)
    pres = [True] * n

    # Initialiser la clé du sommet de départ
    graphe.cle[r0] = 0
    # Placer r0 à la racine du tas
    graphe.echanger(0, graphe.noeudTas[r0])

    # Boucle principale de Prim
    while len(graphe.tas) > 0:
        # Extraire le sommet u de clé minimale
        u = graphe.tas[0]
        # Marquer u comme inclus dans l'arbre
        pres[u] = False

        # Retirer u du tas
        graphe.echanger(0, len(graphe.tas) - 1)
        graphe.tas.pop()

        # Reajuster le tas
        if len(graphe.tas) > 0:
            graphe.versLeBas(0)

        # Mettre à jour les clés et prédécesseurs des sommets adjacents
        for v in range(n):
            # Si v est encore dans le tas et l'arête (u, v) est plus
   petite que la clé actuelle
            if pres[v] and D[u, v] < graphe.cle[v]:
                # Mettre à jour la clé et le prédécesseur
                graphe.cle[v] = D[u, v]
                graphe.pred[v] = u
                # Faire remonter v dans le tas
                graphe.versLeHaut(graphe.noeudTas[v])

    return graphe.pred

def pred_to_tree(pred):
    """
    Convertit la liste des prédécesseurs en une structure d'arbre (
   dictionnaire d'adjacence).
    Args:
        pred (list): Liste des prédécesseurs où pred[i] est le parent de
    i.
    Returns:
        tuple: (d, racine) où d est le dictionnaire d'adjacence et
   racine est l'indice de la racine.
    """
    d = {}
    racine = -1
```

```
    # Initialisation du dictionnaire d'adjacence
    for i in range(len(pred)):
        # Chaque nœud a une liste vide au départ
        d[i] = []
        # Trouver la racine
        if pred[i] is None or pred[i] == -1:
            racine = i

    # Remplir le dictionnaire d'adjacence
    for i in range(len(pred)):
        # Ajouter i comme enfant de son prédécesseur
        if pred[i] is not None and pred[i] != -1:
            # Ajouter i à la liste des enfants de pred[i]
            d[pred[i]].append(i)

    return d, racine


def parcours_prefixe(d, racine):
    """
    Effectue un parcours préfixe (DFS) de l'arbre pour obtenir l'ordre
   de visite des sommets.
    Args:
        d (dict): Dictionnaire d'adjacence de l'arbre.
        racine (int): Indice de la racine de l'arbre.
    Returns:
        list: Liste des sommets dans l'ordre du parcours préfixe.
    """
    # Initialisation de la pile avec la racine
    stack = [racine]
    # Marquage des sommets visités en lui attribuant l'ordre de visite
    p = [0] * len(d)
    # Liste du parcours préfixe
    p_etoile = [racine]
    # Marquer la racine comme visitée
    p[racine] = 1
    count = 2

    # Parcours prefixe
    while len(stack) > 0 :
        # Récupérer le sommet au sommet de la pile
        lst_voisin = d[stack[len(stack) -1]]
        rep = False

        # Parcourir les voisins
        for i in range(len(lst_voisin)):
            # voisin courant de i
            voisin = lst_voisin[i]
            # Si le voisin n'a pas été visité
            if p[voisin] ==0 :
                # L'ajouter à la pile
                stack.append(voisin)
                # L'ajouter au parcours préfixe
                p_etoile.append(voisin)
                # Marquer comme visité en lui attribuant l'ordre de
   visite
                p[voisin] = count
                count +=1
```

```
                # Indiquer qu'on a trouvé un voisin non visité
                rep = True
                break

        # Si aucun voisin non visité n'a été trouvé, dépiler
        if not rep :
            stack.pop()

    return p_etoile

def optPrim(D, r0=0):
    """
    Procédure OptPrim : construit un cycle hamiltonien à partir de l'
   arbre couvrant minimal en utilisant un parcours préfixe.
    Args:
        D (np.ndarray): Matrice des distances.
        r0 (int, optional): Sommet de départ pour Prim. Par défaut 0.
    Returns:
        tuple: (c, l) cycle hamiltonien (liste d'indices), longueur du
   cycle
    """
    # Obtenir l'arbre couvrant minimal via Prim
    pred = prim(D, r0)
    # Convertir les prédécesseurs en arbre
    d, racine = pred_to_tree(pred)
    # Obtenir le cycle via un parcours préfixe de l'arbre
    c = parcours_prefixe(d, racine)

    # Calcul de la longueur totale du cycle
    l = 0.0
    n = len(c)
    for k in range(n):
        l += D[c[k], c[(k + 1) % n]]

    return c, l
```

Listing 3 – Implémentation de OptPrim

4.4 OptPrim + Décroisé (OptOptPrim)

Algorithme : Procédure OptOptPrim
Description : Combine OptPrim et le décroisement.
Vars :
— c_{init} : Liste d'entiers (Cycle initial)
Entrées :
— $points$: Liste de points
— D : Matrice des distances
— $r0$: Sommet de départ
Sorties : c, l
Début :

1. Si D est `None` alors :
2. $D \leftarrow$ `CalculerMatriceDistances`$(points)$
3. $(c_{init}, \_) \leftarrow$ `OptPrim`$(points, D, r0)$ // *Cycle initial (Prim + DFS)*
4. Retourner `Décroise`(c_{init}, D) // *Amélioration par décroisement*

Implémentation Python :

```python
def optOptPrim(D, r0=0):
    """
    Procédure OptOptPrim : combine OptPrim (MST + DFS) et décroisement.
    Génère un cycle initial avec l'heuristique de l'arbre couvrant
   minimum,
    puis l'améliore avec une recherche locale (décroisement).

    Args:
        D (np.ndarray): Matrice des distances.
        r0 (int, optional): Sommet de départ pour Prim. Par défaut 0.
    Returns:
        tuple: (c, l) cycle hamiltonien optimisé, longueur du cycle
    """
    # Obtenir le cycle initial via Prim + Parcours Préfixe
    c_init, _ = optPrim(D, r0)
    # Décroisement
    return decroise(c_init, D)
```

Listing 4 – Implémentation de OptOptPrim

4.5 Évaluation et séparation progressive ("branch and bound") (HDS)

Structure de données : Noeud
Type Noeud = **Enregistrement**
chemin : Liste d'Entiers ;
cout_actuel : Réel ;
dernier : Entier ;
borne : Réel
Fin ;

Structure de données : Tas
Type Tas = **Enregistrement**
tas : Liste de Noeuds
Fin ;

Algorithme : CalcBorne
Description : Calcule une borne inférieure du coût total pour un nœud donné.
Vars :
- $somme$: Réel
- $debut, fin$: Entiers (Extrémités du chemin partiel)

Entrées :
- $noeud$: Structure Noeud
- D : Matrice des distances

Sorties : $borne$ (Réel)
Début :

1. $somme \leftarrow 0$
2. $debut \leftarrow noeud.chemin[1]$
3. $fin \leftarrow noeud.chemin[$dernier$]$
4. Pour chaque sommet i du graphe faire :
5. Si i est interne au chemin alors : // *Cas 1 : Sommet interne*
6. Ajouter à $somme$ les distances vers ses deux voisins dans le chemin
7. Sinon si i est une extrémité ($debut$ ou fin) alors : // *Cas 2 : Extrémité*
8. Ajouter à $somme$ la distance vers son voisin dans le chemin
9. Ajouter à $somme$ la distance minimale vers un sommet non connecté à i dans le chemin
10. Sinon (i hors du chemin) : // *Cas 3 : Hors chemin*
11. Ajouter à $somme$ les deux plus petites distances incidentes à i
12. Retourner $somme/2$

Algorithme : HDS (Branch and Bound)
Description : Recherche le cycle hamiltonien de coût minimal en explorant l'espace des solutions à l'aide d'un arbre de recherche et d'une fonction d'évaluation (borne).
Vars :
- tas : Structure Tas
- $noeud\_courant, nouveau\_noeud$: Structure Noeud
- c : Liste d'entiers (Meilleur cycle)
- l : Réel (Meilleur coût)
- $cout\_total$: Réel

Entrées :
- D : Matrice des distances

Sorties : c, l
Début :

1. $c \leftarrow []$, $l \leftarrow \infty$
2. $tas \leftarrow$ NouveauTas()
3. tas.insere(NouveauNoeud($[0], 0, D$)) // *Initialisation avec racine*
4. Tant que tas n'est pas vide faire :
5. $noeud\_courant \leftarrow tas$.extraire_min() // *Extraire min borne*
6. Si $noeud\_courant.borne \geq l$ alors : // *Élagage*
7. Continuer (élagage)
8. Si taille($noeud\_courant.chemin$) $== n$ alors : // *Chemin complet*
9. $cout\_total \leftarrow noeud\_courant.cout\_actuel + D[noeud\_courant.dernier, noeud\_courant.chemin[1]]$
10. Si $cout\_total < l$ alors : // *Meilleure solution ?*
11. $l \leftarrow cout\_total$
12. $c \leftarrow noeud\_courant.chemin + [noeud\_courant.chemin[1]]$
13. Sinon : // *Brancher*
14. Pour chaque sommet i non dans $noeud\_courant.chemin$ faire :
15. Créer $nouveau\_noeud$ en ajoutant i au chemin
16. tas.insere($nouveau\_noeud$) // *Insérer successeur*
17. Retourner c, l

```python
class Noeud:
    def __init__(self, chemin, cout_actuel, D):
        self.chemin = chemin
        self.cout_actuel = cout_actuel
        self.dernier = chemin[-1]

        self.borne = self.calc_borne(D)

    def calc_borne(self, D):
        """
        Calcule une borne inférieure du coût total pour le noeud courant
    . (demi somme)
        Args:
```

```
            D (np.ndarray): Matrice des distances
        Returns:
            float: Borne inférieure du coût total (demi somme)
        """
        n = len(D)
        some = 0.0
        debut = self.chemin[0]
        fin = self.chemin[-1]

        for i in range(n):
            # Cas 1 : Le sommet est déja completement dans le chemin : A
    ->B->i->C->D
            if i in self.chemin and not i == debut and not i == fin:
                idx = self.chemin.index(i)
                # prendre les deux points adjacents dans le chemin
                some += D[self.chemin[idx - 1], i] + D[i, self.chemin[
    idx + 1]]

            # Cas 2 : Une extrémité du chemin n'est pas dans le chemin :
     i->B->C->D ou A->B->C->i
            elif (i == debut or i == fin) and len(self.chemin) > 1:
                # prendre le point adjacent qui est dans le chemin
                if i == debut:
                    some += D[i, self.chemin[1]]
                else:
                    some += D[self.chemin[-2], i]
                # trouve le minimum parmi les autres
                min_val = float('inf')
                for j in range(n):
                    if min_val > D[i, j] and not i == j:
                        if i == debut and j != self.chemin[1]:
                            min_val = D[i, j]
                        elif i == fin and j != self.chemin[-2]:
                            min_val = D[i, j]
                some += min_val

            # Cas 3 : i non dans le chemin : A->B->C->D     Z
            else:
                sorted_indices = np.argsort(D[i])
                # sorted_indices[0] est i (dist 0), on prend [1] et [2]
                some += D[i, sorted_indices[1]] + D[i, sorted_indices
    [2]]

        borne = some / 2.0
        return borne

    def __lt__(self, other):
        return self.borne < other.borne

class Tas:
    def __init__(self):
        self.tas = []

    def __versLeHaut(self, i):
        """
        Fait remonter un élément dans le tas pour respecter la propriété
     de tas min.
        Args:
```

```
            i (int): Indice de l'élément à faire remonter.
        """
        # Remonter tant que l'élément a une priorité inférieure à son
parent
        while i > 0 and self.priorite(i) < self.priorite((i - 1) // 2):
            # Échanger avec le parent
            self.echanger(i, (i - 1) // 2)
            # Mettre à jour l'indice pour continuer à remonter
            i = (i - 1) // 2

    def __versLeBas(self, i):
        """
        Fait descendre un élément dans le tas pour respecter la propriét
é de tas min.
        Args:
            i (int): Indice de l'élément à faire descendre.
        """
        # Indice du fils gauche
        s = 2 * i + 1
        # Descendre tant que l'élément a une priorité supérieure à l'un
de ses enfants
        while s < len(self.tas):
            # Trouver l'enfant avec la plus petite priorité en comparant
 les deux fils et si c'est le droit
            if s + 1 < len(self.tas) and self.priorite(s + 1) < self.
priorite(s):
                # Prendre le fils droit
                s += 1
            # Si l'élément courant a une priorité supérieure à son
enfant
            if self.priorite(i) > self.priorite(s):
                # echanger et continuer à descendre
                self.echanger(i, s)
                # Mettre à jour l'indice courant de l'elément à
descendre
                i = s
                # Mettre à jour l'indice avec le fils gauche
                s = 2 * i + 1
            # Sinon, la propriété de tas est respectée
            else:
                break

    def insere(self, elem):
        """
        Insère un nouvel élément dans le tas et rétablit la propriété de
 tas min.

        Args:
            elem: L'élément à insérer.
        """
        self.tas.append(elem)
        self.__versLeHaut(len(self.tas) - 1)

    def extraire_min(self):
        """
        Extrait et retourne l'élément de priorité minimale (la racine du
 tas).
        Réorganise le tas après l'extraction.
```

```
        Returns:
            L'élément minimal, ou None si le tas est vide.
        """
        if len(self.tas) == 0:
            return None
        # Extraire le minimum (racine)
        min_noeud = self.tas[0]
        # Remplacer la racine par le dernier élément
        self.tas[0] = self.tas[-1]
        self.tas.pop()
        # Réajuster le tas
        if len(self.tas) > 0:
            self.__versLeBas(0)
        return min_noeud

    def priorite(self, i):
        return self.tas[i].borne

    def echanger(self, i, j):
        self.tas[i], self.tas[j] = self.tas[j], self.tas[i]

def hds(D):
    """
    construire un cycle hamiltonien en utilisant la méthode de Branch
   and Bound (HDS).
    Args:
        D (np.ndarray): Matrice des distances.
    Returns:
        tuple: (c, l) cycle hamiltonien (liste d'indices), longueur du
   cycle
    """
    c = []
    l = float('inf')
    n = len(D)

    if n < 3:
        if n == 0:
            return [], 0
        if n == 1:
            return [0, 0], 0
        if n == 2:
            return [0, 1, 0], D[0, 1] * 2

    tas = Tas()
    # Initialisation du tas avec le noeud racine
    noeud_racine = Noeud([0], 0.0, D)
    tas.insere(noeud_racine)

    while True:
        # Extraire le noeud avec la plus petite borne
        noeud_courant = tas.extraire_min()
        if noeud_courant is None:
            break

        # si la borne théorique est supérieure à la meilleure solution
   trouvée, on ignore ce noeud
        if noeud_courant.borne >= l:
```

```
            continue

        # Si le chemin est complet
        if len(noeud_courant.chemin) == n:
            # Ajouter le coût de retour au point de départ
            cout_total = noeud_courant.cout_actuel + D[noeud_courant.
dernier, noeud_courant.chemin[0]]
            # Si ce coût est inférieur à la meilleure solution trouvée
            if cout_total < l:
                l = cout_total
                c = noeud_courant.chemin + [noeud_courant.chemin[0]]
        else:
            # Générer les successeurs possibles
            for i in range(n):
                if i not in noeud_courant.chemin:
                    nouveau_chemin = noeud_courant.chemin + [i]
                    nouveau_cout = noeud_courant.cout_actuel + D[
noeud_courant.dernier, i]
                    nouveau_noeud = Noeud(nouveau_chemin, nouveau_cout,
D)
                    # Insérer le nouveau noeud dans le tas
                    tas.insere(nouveau_noeud)

    return c, l
```

Listing 5 – Implémentation de HDS