

Résolution du Problème du Voyageur de Commerce

Mohamed Elakef Zenagui

30 décembre 2025

Table des matières

1	Introduction	3
2	Structures de données algorithmiques	3
2.1	Structures générales	3
2.2	Structures pour Prim	3
2.3	Structures pour DFS	3
2.4	Traduction en Python	4
3	Algorithme du Point le Plus Proche (PPP)	6
4	Amélioration de la stratégie du Point le Plus Proche	6
5	Stratégie par l'arbre couvrant de poids minimum	8
5.1	Parcours en profondeur (DFS) avec backtracking	8
5.2	Algorithme de Prim efficace	9
5.3	PVCPrim	10
6	Algorithme exact HDS	10
6.1	Heuristique de la demi-somme	10
6.2	Algorithme HDS	11
7	Étude statistique des heuristiques	12
7.1	Résultats numériques	13
7.2	Représentation graphique	13
7.3	Analyse des résultats	13
8	Analyse de la complexité	14
8.1	Heuristique du Plus Proche Point (PPP)	14
8.2	Amélioration OptPPP	14
8.3	Approximation par arbre couvrant minimum (PVCPrim)	15
8.4	Algorithme exact HDS (Branch and Bound)	15
8.5	Synthèse des complexités	15
9	Conclusion	15

Table des figures

1	Histogrammes des longueurs des cycles obtenus par les 4 algorithmes.	13
2	Comparaison des 3 méthodes approximative avec HDS.	14

Liste des tableaux

1	Longueurs moyennes et gains des algorithmes sur 100 essais.	13
2	Complexité temporelle des algorithmes étudiés.	15

1 Introduction

Le problème du voyageur de commerce (*Travelling Salesman Problem*, TSP) consiste à déterminer une tournée hamiltonienne de coût minimal passant exactement une fois par chaque sommet d'un graphe complet valué. Ce problème est NP-difficile.

Dans ce projet, nous étudions et implémentons plusieurs approches :

- des heuristiques constructives (PPP),
- des heuristiques d'amélioration (OptPPP),
- une approximation basée sur un arbre couvrant minimum (PVCPrim),
- un algorithme exact par *Branch and Bound* (HDS).

2 Structures de données algorithmiques

2.1 Structures générales

CONSTANTE *NOMBRE_POINTS* = 10

CONSTANTE *NOMBRE_ESSAIS* = 100

TYPE **Arrete** = enregistrement

sommet : entier

poids : réel

suiv : ↑ Arrete

FIN

TYPE **GrapheM** = enregistrement

n : entier

M : Tableau[1..n][1..n] de réel

FIN

TYPE **cycle** = ↑ Arrete

2.2 Structures pour Prim

TYPE **GrapheD** = enregistrement

n : entier

L : Tableau[1..n] de ↑ Arrete

cle : Tableau[1..n] de réel

π : Tableau[1..n] d'entier

NoeudTas : Tableau[1..n] d'entier

FIN

TYPE **Tas** = enregistrement

dern : entier

Tab : Tableau[1..n] d'entier

FIN

2.3 Structures pour DFS

TYPE **Cellule** = enregistrement

sommet : entier

suiv : \uparrow Cellule
FIN

TYPE **GrapheTL** = enregistrement
n : entier
L : Tableau[1..n] de \uparrow Cellule
FIN

TYPE **ETAT** = (BLANC, GRIS, NOIR)

2.4 Traduction en Python

Les structures de données algorithmiques présentées précédemment peuvent être traduites de manière naturelle en Python, en utilisant des structures standards telles que les listes, dictionnaires et files de priorité.

Graphe matriciel La structure **GrapheM**, représentant un graphe complet valué par une matrice de distances, peut être implémentée en Python par une liste de listes ou une matrice NumPy :

```
import numpy as np
M = np.zeros((n, n), dtype=float)
```

Arêtes et cycles Le type **Arrete**, utilisé pour représenter un cycle sous forme de liste chaînée, peut être remplacé en Python par une simple liste d'entiers contenant l'ordre de visite des sommets :

```
cycle = [1, 3, 5, 2, 4, 1]
```

Marquage des sommets Les tableaux de booléens utilisés pour marquer les sommets visités dans PPP ou HDS sont implémentés à l'aide de listes Python :

```
marked = [False] * n
```

Tas de priorité La structure **Tas** utilisée dans HDS peut être implémentée avec le module `heapq` de Python :

```
import heapq
heap = []
heapq.heappush(heap, (borne, cout, cycle))
borne, cout, cycle = heapq.heappop(heap)
```

Pour l'algorithme de Prim, on peut utiliser un tas spécialisé pour optimiser les mises à jour de priorité :

```
from src.heap import Heap
H = Heap()
d = np.ones(self.taille_) * np.inf
d[source] = 0
pi = -np.ones(self.taille_, dtype=int)

tas = zip(set(range(self.taille_)), d)
```

```
H.init_heap(tas)
```

```
curr_vertex, curr_priority = H.dequeue()  
H.update_priority(vertex, d[vertex])
```

Classe PVC La classe PVC implémente le cœur des algorithmes pour le Problème du Voyageur de Commerce (PVC) en programmation orientée objet. Les attributs principaux sont :

- `sommets_` : *set* contenant les indices des sommets.
- `arretes_` : liste des arêtes du graphe.
- `taille_` : nombre de sommets.
- `M` : matrice des distances (numpy array).
- `adj_dict_` : dictionnaire d'adjacence du graphe.

```
class PVC:  
    def __init__(self):  
        # Structure interne  
        self.sommets_ = set()  
        self.arretes_ = []  
        self.taille_ = 0  
  
        # Représentations  
        self.M = None  
        self.adj_dict_ = None
```

Classe PVC_points La classe `PVC_points` hérite de `PVC` et ajoute des fonctionnalités pour manipuler des points dans le plan, générer la matrice des distances et encapsuler les algorithmes PPP, OptPPP, PVCPrim et HDS.

Attributs principaux :

- `points_` : liste des points (x, y) .
- `idx_map_` : dictionnaire associant chaque point à son indice dans la matrice de distances.
- `inv_` : dictionnaire inversant les indices pour récupérer les points.
- `seed_` : indice du point de départ pour certaines heuristiques.

Méthodes principales :

- `charger_de_liste(points)` : charge une liste de points, crée la matrice des distances et initialise les dictionnaires d'indices.
- `charger_de_fichier(fichier)` : lit un fichier texte contenant des points et appelle `charger_de_liste`.
- `euclidean_distance(P, Q)` : calcule la distance euclidienne entre deux points.
- `PPP()`, `OptPPP()`, `PVCPrim()`, `HDS()` : méthodes encapsulant les algorithmes étudiés.

Exemple d'utilisation :

```
from src.PVC_points import PVC_points  
  
pvc = PVC_points()  
points = PVC_points.generer_points(10) # Génère 10 points aléatoires  
pvc.charger_de_liste(points)  
  
cycle_ppp = pvc.PPP()      # Heuristique du plus proche point  
cycle_opt = pvc.OptPPP()   # Amélioration par décroisement  
cycle_prim = pvc.PVCPrim() # Approximation par arbre couvrant
```

Cette approche POO centralise les données et simplifie l'utilisation des différentes heuristiques tout en conservant la logique algorithmique.

3 Algorithme du Point le Plus Proche (PPP)

L'algorithme PPP construit un cycle hamiltonien en partant d'un point initial et en ajoutant successivement le point non inclus le plus proche du cycle courant, jusqu'à ce que tous les points soient inclus.

Input: G : GrapheM, s : Entier (point de départ)
Output: c : *cycle* (cycle hamiltonien)
Var u, v : Entier
Var min_dist : Réel
 $marked$: Tableau $[1..n]$ de Booléen
 $c \leftarrow \emptyset$
ajouter(c, s)
for $i \leftarrow 1$ **to** $G.n$ **do**
 | $marked[i] \leftarrow Faux$
end
 $marked[s] \leftarrow Vrai$
while $|c| \neq G.n$ **do**
 | $min_dist \leftarrow \inf$
 | **for** $i \leftarrow 1$ **to** $G.n$ **do**
 | | **if** *not* $marked[i]$ **then**
 | | | $p \leftarrow c$
 | | | **while** $p \neq NIL$ **do**
 | | | | **if** $G.M[i, p \uparrow sommet] < min_dist$ **then**
 | | | | | $min_dist \leftarrow G.M[i, p \uparrow sommet]$
 | | | | | $u \leftarrow i$
 | | | | | $v \leftarrow p \uparrow sommet$
 | | | | **end**
 | | | $p \leftarrow p \uparrow suiv$
 | | **end**
 | **end**
 | **end**
 | ajouter(c, v, u) // Ajouter u dans le cycle c après v
 | $marked[u] \leftarrow Vrai$
end
ajouter(c, s) // Fermer le cycle en revenant au point de départ
Retour c

Algorithm 1: Algorithme PPP

4 Amélioration de la stratégie du Point le Plus Proche

Une amélioration possible du coût du cycle obtenu par la procédure PPP consiste à **décroiser les arêtes qui se croisent**. Soit (i, j) un couple d'entiers dans l'intervalle $[1, n]$ tel que $j \geq i + 2$, et soit le cycle :

$$c = (PL[1], \dots, PL[i], PL[i + 1], \dots, PL[j], PL[j + 1], \dots, PL[n]).$$

Si le décroisement des arêtes $(PL[i], PL[i + 1])$ et $(PL[j], PL[j + 1])$ réduit la longueur totale du cycle, on transforme c en :

$$\bar{c} = (PL[1], \dots, PL[i], PL[j], \dots, PL[i + 1], PL[j + 1], \dots, PL[n]).$$

Input: c : cycle obtenu par PPP
Output: c : cycle amélioré
Var amelioration : Booléen
 a, b, d, e : Entier
 p, q, r, s : \uparrow *Arrete*
 $amelioration \leftarrow Vrai$
while $amelioration$ **do**
 $amelioration \leftarrow Faux$
 $p \leftarrow c$
 $q \leftarrow p \uparrow suiv$
 while $q \uparrow suiv \neq NIL$ **do**
 $r \leftarrow q \uparrow suiv$
 $s \leftarrow r \uparrow suiv$
 while $s \neq NIL$ **do**
 $a \leftarrow p \uparrow sommet, b \leftarrow q \uparrow sommet$
 $d \leftarrow r \uparrow sommet, e \leftarrow s \uparrow sommet$
 if $G.M[a, d] + G.M[b, e] < G.M[a, b] + G.M[d, e]$ **then**
 // Décroisement avantageux
 // Inverser le segment de c compris entre les cellules q et r
 inverser(c, q, r)
 $amelioration \leftarrow Vrai$
 end
 $r \leftarrow s$
 $s \leftarrow s \uparrow suiv$
 end
 $p \leftarrow q$
 $q \leftarrow q \uparrow suiv$
 end
end
Retour c

Algorithm 2: Algorithme OptPPP

Cet algorithme répète les opérations de décroisement tant qu'il existe des couples d'arêtes croisées dont le remplacement réduit la longueur du cycle. Il permet ainsi d'améliorer efficacement la solution initiale fournie par PPP.

5 Stratégie par l'arbre couvrant de poids minimum

5.1 Parcours en profondeur (DFS) avec backtracking

Input: GrapheTL G , Entier $origine$
Output: Tableaux $couleurs$, π , P , S , P^* , S^* ; Entiers i_p , i_s
Var v : Entier
 $p : \uparrow \text{Cellule}$
 $couleurs[origine] \leftarrow GRIS$
 $i_p \leftarrow i_p + 1$
 $P[origine] \leftarrow i_p$
 $P^*[i_p] \leftarrow origine$
 $p \leftarrow G.L[origine]$
while $p \neq NIL$ **do**
 $v \leftarrow p \uparrow \text{sommet}$
 if $couleurs[v] = BLANC$ **then**
 $\pi[v] \leftarrow origine$
 Visiter_Profondeur($G, v, couleurs, \pi, P, S, P^*, S^*, i_p, i_s$)
 end
 $p \leftarrow p \uparrow \text{suiv}$
end
 $couleurs[origine] \leftarrow NOIR$
 $i_s \leftarrow i_s + 1$
 $S[origine] \leftarrow i_s$
 $S^*[i_s] \leftarrow origine$

Algorithm 3: Visite en profondeur

Input: GrapheTL G
Output: Tableaux π , P , S , P^* , S^*
Vari, i_p, i_s : Entier
 $couleurs$: Tableau[1.. $G.n$] de COULEUR
for $i \leftarrow 1$ **to** $G.n$ **do**
 $couleurs[i] \leftarrow BLANC$
 $\pi[i] \leftarrow -1$
 $P[i] \leftarrow 0$
 $S[i] \leftarrow 0$
 $P^*[i] \leftarrow 0$
 $S^*[i] \leftarrow 0$
end
 $i_p \leftarrow 0$
 $i_s \leftarrow 0$
for $i \leftarrow 1$ **to** $G.n$ **do**
 if $couleurs[i] = BLANC$ **then**
 Visiter_Profondeur($G, i, couleurs, \pi, P, S, P^*, S^*, i_p, i_s$)
 end
end

Algorithm 4: DFS principal (backtracking)

5.2 Algorithme de Prim efficace

Input: G : GrapheD, r : Entier
Output: π : Tableau[1..n] d'Entier
Var t, i : Entier
 $p : \uparrow$ Arrete
 $T.dern \leftarrow G.n$
for $i \leftarrow 1$ **to** $G.n$ **do**
 $G.cle[i] \leftarrow +\infty$
 $G.\pi[i] \leftarrow -1$
 $G.NoeudTas[i] \leftarrow i$
 $T.Tab[i] \leftarrow i$
end
 $G.cle[r] \leftarrow 0$
while $T.dern \geq 1$ **do**
 $t \leftarrow T.Tab[1]$
 Echange(1, $T.dern$, G, T)
 $T.dern \leftarrow T.dern - 1$
 Verslebas(1, G, T)
 $p \leftarrow G.L[t]$
 while $p \neq NIL$ **do**
 if $G.NoeudTas[p \uparrow sommet] \leq T.dern$ **et** $p \uparrow poids < G.cle[p \uparrow sommet]$ **then**
 $G.cle[p \uparrow sommet] \leftarrow p \uparrow poids$
 $G.\pi[p \uparrow sommet] \leftarrow t$
 Verslehaut($G.NoeudTas[p \uparrow sommet]$, G, T)
 end
 $p \leftarrow p \uparrow suiv$
 end
end

Algorithm 5: Prim efficace

5.3 PVCPPrim

Input: G : Graphe complet valué, Q : Entier

Output: C : Cycle hamiltonien

Var π, P, P^*, S, S^* : Tableau[1..n] d'Entier

T : GrapheTL

i : Entier

$\pi \leftarrow \mathbf{Prim}(G, Q)$

$C \leftarrow \emptyset$

for $i \leftarrow 1$ **to** n **do**

if $\pi[i] > -1$ **then**

 ajouter($T.L[i]$, $\pi[i]$)

 ajouter($T.L[\pi[i]]$, i)

end

end

backtrack(T, π, P, S, P^*, S^*)

$C \leftarrow \emptyset$

for $i \leftarrow 1$ **to** n **do**

 ajouter($C, P^*[i]$)

end

ajouter($C, P^*[1]$)

Retour C

Algorithm 6: PVCPPrim : approximation du PVC par arbre couvrant

6 Algorithme exact HDS

6.1 Heuristique de la demi-somme

L'heuristique de la demi-somme fournit une borne inférieure du coût restant à parcourir. Pour chaque sommet on considère la demi-somme des deux plus petites arêtes incidentes admissibles. Cette heuristique fournit une borne inférieure admissible, car chaque sommet du cycle final devra être incident à exactement deux arêtes.

Input: GrapheM G , \uparrow Arrete c
Output: Entier borne
Var k, i : Entier
 distances : Tableau[1..n] de réels
 p, q, r : \uparrow Arrete
 $borne \leftarrow 0$
 $k \leftarrow |c|$
for $i \leftarrow 1$ **to** $G.n$ **do**
 if $k = 1$ **or not** $i \in sommets(c)$ **then**
 distances $\leftarrow tri(G.M[i])$
 borne $\leftarrow borne + distances[2] + distances[3]$
 end
end
 $p \leftarrow c$
 $q \leftarrow p \uparrow suiv$
while $q \neq NIL$ **do**
 if $p = c$ **ou** $q \uparrow suiv = NIL$ **then**
 borne $\leftarrow borne + G.M[p \uparrow sommet, q \uparrow sommet]$
 if $q \uparrow suiv = NIL$ **then**
 distances $\leftarrow tri(G.M[q \uparrow sommet])$
 end
 else
 distances $\leftarrow tri(G.M[p \uparrow sommet])$
 end
 if distances[2] = $G.M[p \uparrow sommet, q \uparrow sommet]$ **then**
 borne $\leftarrow borne + distances[3]$
 end
 else
 borne $\leftarrow borne + distances[2]$
 end
 end
 else
 $r \leftarrow q \uparrow suiv$
 borne $\leftarrow borne + G.M[p \uparrow sommet, q \uparrow sommet] + G.M[q \uparrow sommet, r \uparrow sommet]$
 end
 $p \leftarrow q$
 $q \leftarrow q \uparrow suiv$
end
Retour $\frac{borne}{2}$

Algorithm 7: Heuristique de la demi-somme

6.2 Algorithme HDS

L'algorithme HDS explore l'espace des solutions partielles en privilégiant les nœuds possédant la plus petite borne inférieure.

```

Input: GrapheM  $G$ 
Output:  $best\_solution : \uparrow Arrete$ ,  $best\_cost : \text{R  el}$ 
Var  $u, v : \text{Entier}$ 
        $borne, h, new\_cost, cost : \text{R  el}$ 
        $T : \text{Tas}$ 
        $c, new\_c : \uparrow Arrete$ 
 $best\_cost \leftarrow +\infty$ 
 $best\_solution \leftarrow \text{NIL}$ 
 $T \leftarrow \emptyset$ 
 $c \leftarrow \emptyset$ 
ajouter( $c, 1$ )
 $cost \leftarrow 0$ 
 $borne \leftarrow \text{Heuristique\_Demi\_Somme}(G, c)$ 
Entasser( $T, \langle borne, cost, c \rangle$ )
while  $T \neq \emptyset$  do
    D  tasser( $T, \langle borne, cost, c \rangle$ )
    if  $borne < best\_cost$  then
        if  $|c| = G.n$  then
             $total\_cost \leftarrow cost + G.M[\text{dernier}(c), 1]$ 
            if  $total\_cost < best\_cost$  then
                 $best\_cost \leftarrow total\_cost$ 
                 $best\_solution \leftarrow c$ 
            end
        end
    else
         $u \leftarrow \text{dernier}(c)$ 
        for  $v \leftarrow 1$  to  $G.n$  do
            if not  $v \in \text{sommets}(c)$  then
                 $new\_cost \leftarrow cost + G.M[u, v]$ 
                 $new\_c \leftarrow c$ 
                ajouter( $new\_c, v$ )
                 $h \leftarrow \text{Heuristique\_Demi\_Somme}(G, new\_c)$ 
                if  $h < best\_cost$  then
                    Entasser( $T, \langle h, new\_cost, new\_c \rangle$ )
                end
            end
        end
    end
end
end
Retour ( $best\_solution, best\_cost$ )

```

Algorithm 8: Algorithme HDS

7   tude statistique des heuristiques

Pour   valuer la qualit   des heuristiques, nous avons g  n  r   100 ensembles de 10 points tir  s al  atoirement dans $[0, 1]^2$ et appliqu   les algorithmes **PPP**, **OptPPP** et **PVCPrim**. Nous avons calcul   pour chaque algorithme la longueur moyenne du cycle obtenu et les gains relatifs entre les m  thodes.

7.1 Résultats numériques

Les résultats moyens obtenus sont présentés dans le tableau ci-dessous :

Algorithme	Longueur moyenne	Gain / PPP (%)	Gain / OptPPP (%)	Gain / PVCPrim (%)
PPP	3.483	-	-8.65	-1.79
OptPPP	3.193	8.65	-	7.92
PVCPrim	3.422	1.79	-7.92	-
HDS (exact)	2.907	16.22	8.07	14.33

TABLE 1 – Longueurs moyennes et gains des algorithmes sur 100 essais.

7.2 Représentation graphique

La figure ci-dessous montre la répartition des longueurs obtenues par les différentes méthodes.

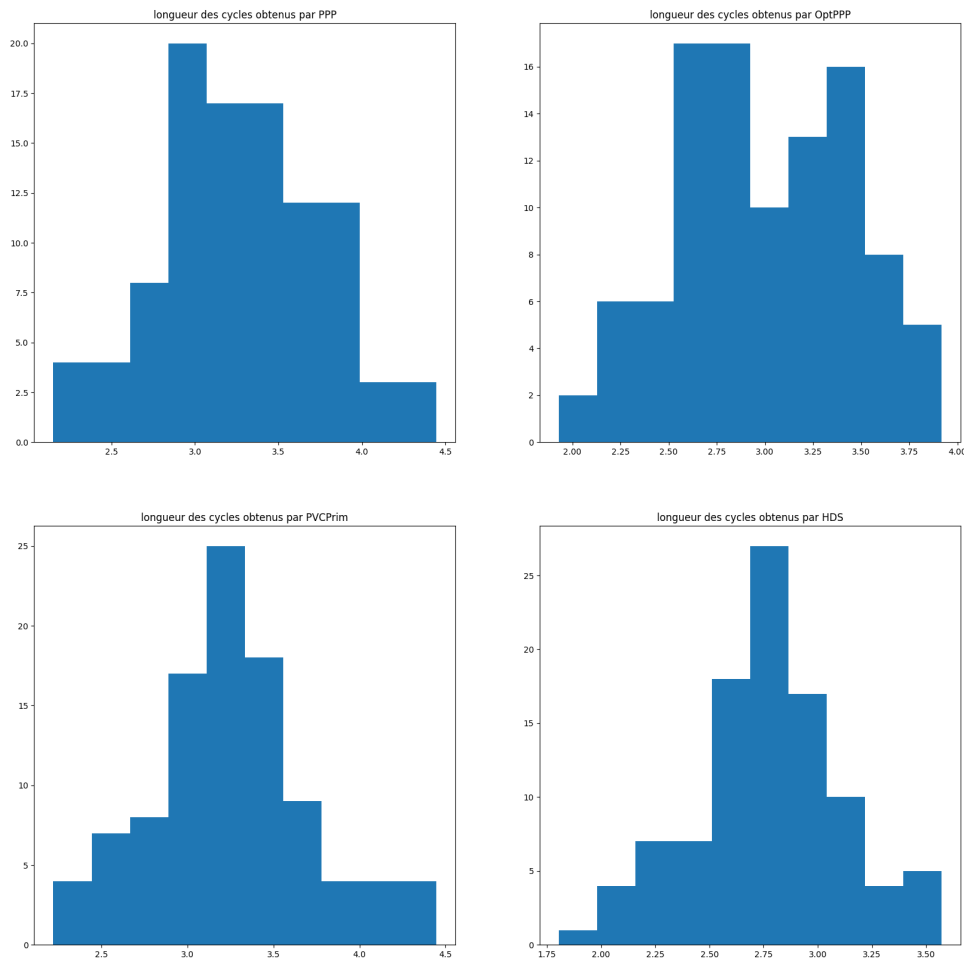


FIGURE 1 – Histogrammes des longueurs des cycles obtenus par les 4 algorithmes.

7.3 Analyse des résultats

- Les résultats numériques montrent que l'algorithme **OptPPP** améliore significativement la solution obtenue par **PPP**. La longueur moyenne du cycle passe de 3.483 à 3.193, ce qui correspond à un gain moyen de **8.65 %** par rapport à **PPP**.
- L'algorithme **PVCPrim** permet également d'améliorer la solution fournie par **PPP**, avec une réduction moyenne de **1.79 %** de la longueur du cycle. Toutefois, les cycles obtenus

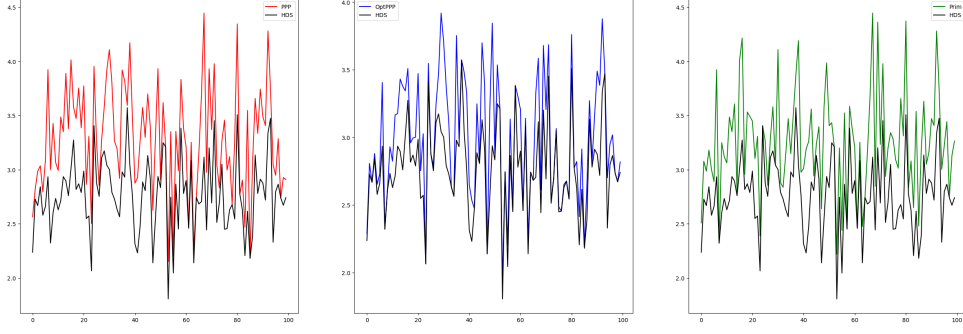


FIGURE 2 – Comparaison des 3 méthodes approximative avec HDS.

par **PVCPrim** restent en moyenne **7.92 % plus longs** que ceux produits par **OptPPP**, ce qui montre que les améliorations locales sont plus efficaces dans ce contexte.

- L'algorithme exact **HDS** fournit la solution optimale du problème. Il permet d'obtenir une longueur moyenne de 2.907, soit un gain de **16.22 %** par rapport à **PPP**, de **8.07 %** par rapport à **OptPPP** et de **14.33 %** par rapport à **PVCPrim**. Ces résultats confirment la qualité des solutions approchées tout en mettant en évidence l'écart restant avec l'optimum.
- L'étude statistique, confirmée par les représentations graphiques, montre que les heuristiques offrent un compromis pertinent entre qualité de solution et temps de calcul. En particulier, **OptPPP** apparaît comme la meilleure méthode approchée parmi celles étudiées, fournissant des solutions proches de l'optimum pour un coût de calcul raisonnable.

8 Analyse de la complexité

Dans cette section, nous analysons la complexité temporelle des différents algorithmes implémentés pour la résolution du Problème du Voyageur de Commerce (PVC). On note n le nombre de points à visiter.

8.1 Heuristique du Plus Proche Point (PPP)

L'algorithme **PPP** construit le cycle en sélectionnant, à chaque étape, le point non encore visité le plus proche du point courant.

À chaque itération, l'algorithme parcourt l'ensemble des points non visités afin de déterminer le plus proche voisin, ce qui nécessite un temps en $O(n)$. Cette opération est répétée pour chacun des n sommets.

Complexité temporelle de PPP : $O(n^2)$

8.2 Amélioration OptPPP

L'algorithme **OptPPP** améliore la solution fournie par PPP en éliminant les croisements d'arêtes par des opérations locales de type 2-opt.

Chaque amélioration potentielle consiste à tester des paires d'arêtes, ce qui conduit à un nombre de combinaisons en $O(n^2)$. Dans le pire des cas, ces opérations peuvent être répétées $O(n)$ fois jusqu'à stabilisation de la solution.

Complexité temporelle de OptPPP : $O(n^3)$

En pratique, le nombre d'itérations est souvent bien inférieur, ce qui rend l'algorithme utilisable pour des tailles de problèmes modérées.

8.3 Approximation par arbre couvrant minimum (PVCPrim)

L'algorithme **PVCPrim** repose sur la construction d'un arbre couvrant de poids minimum à l'aide de l'algorithme de Prim, puis sur un parcours en profondeur de l'arbre obtenu afin de construire un cycle hamiltonien.

La complexité de l'algorithme de Prim, implémenté à l'aide d'un tas binaire, est en $O(n \log n)$. Le parcours en profondeur s'effectue en temps linéaire $O(n)$.

Complexité temporelle de PVCPrim : $O(n \log n)$

8.4 Algorithme exact HDS (Branch and Bound)

L'algorithme **HDS** repose sur une exploration systématique de l'espace des permutations possibles des sommets, combinée à une stratégie de *Branch and Bound* utilisant une borne inférieure basée sur la demi-somme des arêtes.

Dans le pire des cas, l'algorithme doit explorer l'ensemble des permutations possibles, soit $(n - 1)!$, ce qui conduit à une complexité exponentielle.

Complexité temporelle de HDS : $O(n!)$

Toutefois, en pratique, l'utilisation de bornes efficaces permet de réduire considérablement l'espace de recherche, rendant l'algorithme exploitable pour des instances de petite taille.

8.5 Synthèse des complexités

Algorithme	Complexité temporelle	Nature
PPP	$O(n^2)$	Heuristique
OptPPP	$O(n^3)$	Heuristique améliorée
PVCPrim	$O(n \log n)$	Approximation
HDS	$O(n!)$	Exact

TABLE 2 – Complexité temporelle des algorithmes étudiés.

9 Conclusion

Dans ce projet, nous avons étudié plusieurs approches pour résoudre le problème du voyageur de commerce, un problème d'optimisation combinatoire connu pour sa complexité.

Les heuristiques constructives comme **PPP** permettent d'obtenir rapidement des solutions valides, tandis que **OptPPP** améliore significativement leur qualité grâce à des opérations de décroisement. La stratégie **PVCPrim**, basée sur un arbre couvrant de poids minimum suivi d'un parcours en profondeur, fournit une approximation efficace avec un bon compromis entre coût et temps de calcul.

Enfin, l'algorithme **HDS**, fondé sur le principe du *Branch and Bound* et utilisant l'heuristique de la demi-somme comme borne inférieure, permet d'obtenir une solution optimale au prix d'une complexité exponentielle dans le pire des cas.

L'ensemble de ces méthodes illustre les compromis classiques entre rapidité, précision et complexité, et montre l'intérêt de combiner heuristiques et algorithmes exacts selon la taille et les contraintes du problème à résoudre.