Résolution du Problème du Voyageur de Commerce

Mohamed Elakef Zenagui

28 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) **5**

4 Amélioration de la stratégie du Point le Plus Proche **5**

5 Stratégie par l'arbre couvrant de poids minimum **7**
5.1 Parcours en profondeur (DFS) avec backtracking . . . 7
5.2 Algorithme de Prim efficace . . . 8
5.3 PVCPrim . . . 9

6 Algorithme exact HDS **9**
6.1 Heuristique de la demi-somme . . . 9
6.2 Algorithme HDS . . . 10

7 Étude statistique des heuristiques **11**
7.1 Résultats numériques . . . 12
7.2 Représentation graphique . . . 12
7.3 Analyse des résultats . . . 12

8 Analyse de la complexité **13**
8.1 Heuristique du Plus Proche Point (PPP) . . . 13
8.2 Amélioration OptPPP . . . 13
8.3 Approximation par arbre couvrant minimum (PVCPrim) . . . 13
8.4 Algorithme exact HDS (Branch and Bound) . . . 14
8.5 Synthèse des complexités . . . 14

9 Conclusion **14**

Table des figures

1 Histogrammes des longueurs des cycles obtenus par les 4 algorithmes. 12
2 Comparaison des 3 méthodes approximative avec HDS. 13

Liste des tableaux

1 Longueurs moyennes et gains des algorithmes sur 100 essais. 12
2 Complexité temporelle des algorithmes étudiés. 14

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 : ↑ Cellule
FIN

TYPE **GrapheTL** = enregistrement
n : entier
L : Tableau[1..n] de ↑ 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.

```
G = {
    "n": n,
    "M": [[float] * n for _ in range(n)]
}
```

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 dans PPP ou HDS pour marquer les sommets visités sont implémentés à l'aide de listes Python.

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

Tas de priorité La structure **Tas** utilisée dans les algorithmes Prim et HDS est implémentée efficacement à l'aide du module `heapq` de Python.

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

Cette traduction permet une implémentation simple, lisible et efficace des algorithmes étudiés, tout en conservant leur 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 ← ∅
ajouter(c, s)
for i ← 1 to G.n do
    marked[i] ← Faux
end
marked[s] ← Vrai
while |c| ≠ G.n do
    min_dist ← inf
    for i ← 1 to G.n do
        if not marked[i] then
            p ← c
            while p ≠ NIL do
                if G.M[i, p ↑ sommet] < min_dist then
                    min_dist ← G.M[i, p ↑ sommet]
                    u ← i
                    v ← p ↑ sommet
                end
                p ← p ↑ suiv
            end
        end
    end
    ajouter(c, v, u) // Ajouter u dans le cycle c après v
    marked[u] ← 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 :

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

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 :

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

```
Input: c : cycle obtenu par PPP
Output: c : cycle amélioré
Var amelioration : Booléen
    a, b, d, e : Entier
    p, q, r, s : ↑ Arrete
amelioration ← Vrai
while amelioration do
    amelioration ← Faux
    p ← c
    q ← p ↑ suiv
    while q ↑ suiv ≠ NIL do
        r ← q ↑ suiv
        s ← r ↑ suiv
        while s ≠ NIL do
            a ← p ↑ sommet, b ← q ↑ sommet
            d ← r ↑ sommet, e ← s ↑ 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 ← Vrai
            end
            r ← s
            s ← s ↑ suiv
        end
        p ← q
        q ← q ↑ 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 : ↑ Cellule

```
couleurs[origine] ← GRIS
i_p ← i_p + 1
P[origine] ← i_p
P*[i_p] ← origine
p ← G.L[origine]
while p ≠ NIL do
    v ← p ↑ sommet
    if couleurs[v] = BLANC then
        π[v] ← origine
        Visiter_Profondeur(G, v, couleurs, π, P, S, P*, S*, i_p, i_s)
    end
    p ← p ↑ suiv
end
couleurs[origine] ← NOIR
i_s ← i_s + 1
S[origine] ← i_s
S*[i_s] ← 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 ← 1 to G.n do
    couleurs[i] ← BLANC
    π[i] ← −1
    P[i] ← 0
    S[i] ← 0
    P*[i] ← 0
    S*[i] ← 0
end
i_p ← 0
i_s ← 0
for i ← 1 to G.n do
    if couleurs[i] = BLANC then
        Visiter_Profondeur(G, i, couleurs, π, 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 PVCPrim

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
$\pi \leftarrow$ **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: PVCPrim : 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, ↑ Arrete c
Output: Entier borne
Var k, i : Entier
    distances : Tableau[1..n] de réels
    p, q, r : ↑ Arrete
borne ← 0
k ← |c|
for i ← 1 to G.n do
    if k = 1 or not i ∈ sommets(c) then
        distances ← tri(G.M[i])
        borne ← borne + distances[2] + distances[3]
    end
end
p ← c
q ← p ↑ suiv
while q ≠ NIL do
    if p = c ou q ↑ suiv = NIL then
        borne ← borne + G.M[p ↑ sommet, q ↑ sommet]
        if q ↑ suiv = NIL then
            distances ← tri(G.M[q ↑ sommet])
        end
        else
            distances ← tri(G.M[p ↑ sommet])
        end
        if distances[2] = G.M[p ↑ sommet, q ↑ sommet] then
            borne ← borne + distances[3]
        end
        else
            borne ← borne + distances[2]
        end
    end
    else
        r ← q ↑ suiv
        borne ← borne + G.M[p ↑ sommet, q ↑ sommet] + G.M[q ↑ sommet, r ↑ sommet]
    end
    p ← q
    q ← q ↑ suiv
end
Retour 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 :↑ Arrete, best_cost : Réel
Var u, v : Entier
    borne, h, new_cost, cost : Réel
    T : Tas
    c, new_c : ↑ Arrete
best_cost ← +∞
best_solution ← NIL
T ← ∅
c ← ∅
ajouter(c, 1)
cost ← 0
borne ← Heuristique_Demi_Somme(G, c)
Entasser(T, ⟨borne, cost, c⟩)
while T ≠ ∅ do
    Détasser(T, ⟨borne, cost, c⟩)
    if borne < best_cost then
        if |c| = G.n then
            total_cost ← cost + G.M[dernier(c), 1]
            if total_cost < best_cost then
                best_cost ← total_cost
                best_solution ← c
            end
        end
        else
            u ← dernier(c)
            for v ← 1 to G.n do
                if not v ∈ sommets(c) then
                    new_cost ← cost + G.M[u, v]
                    new_c ← c
                    ajouter(new_c, v)
                    h ← Heuristique_Demi_Somme(G, new_c)
                    if h < best_cost then
                        Entasser(T, ⟨h, new_cost, new_c⟩)
                    end
                end
            end
        end
    end
end
Retour (best_solution, best_cost)
```

Algorithm 8: Algorithme HDS

7 Étude statistique des heuristiques

Pour évaluer la qualité des heuristiques, nous avons généré 100 ensembles de 10 points tirés aléatoirement dans $[0,1]^2$ et appliqué les algorithmes **PPP**, **OptPPP** et **PVCPrim**. Nous avons calculé pour chaque algorithme la longueur moyenne du cycle obtenu et les gains relatifs entre les méthodes.

7.1 Résultats numériques

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

Algorithme	Longueur moyenne	Gain par rapport à PPP (%)	Gain par rapport à OptPPP (%)
PPP	3.483	-	-
OptPPP	3.193	7.95	-
PVCPrim	3.422	-	-9.60
HDS (exact)	2.907	-	-

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.

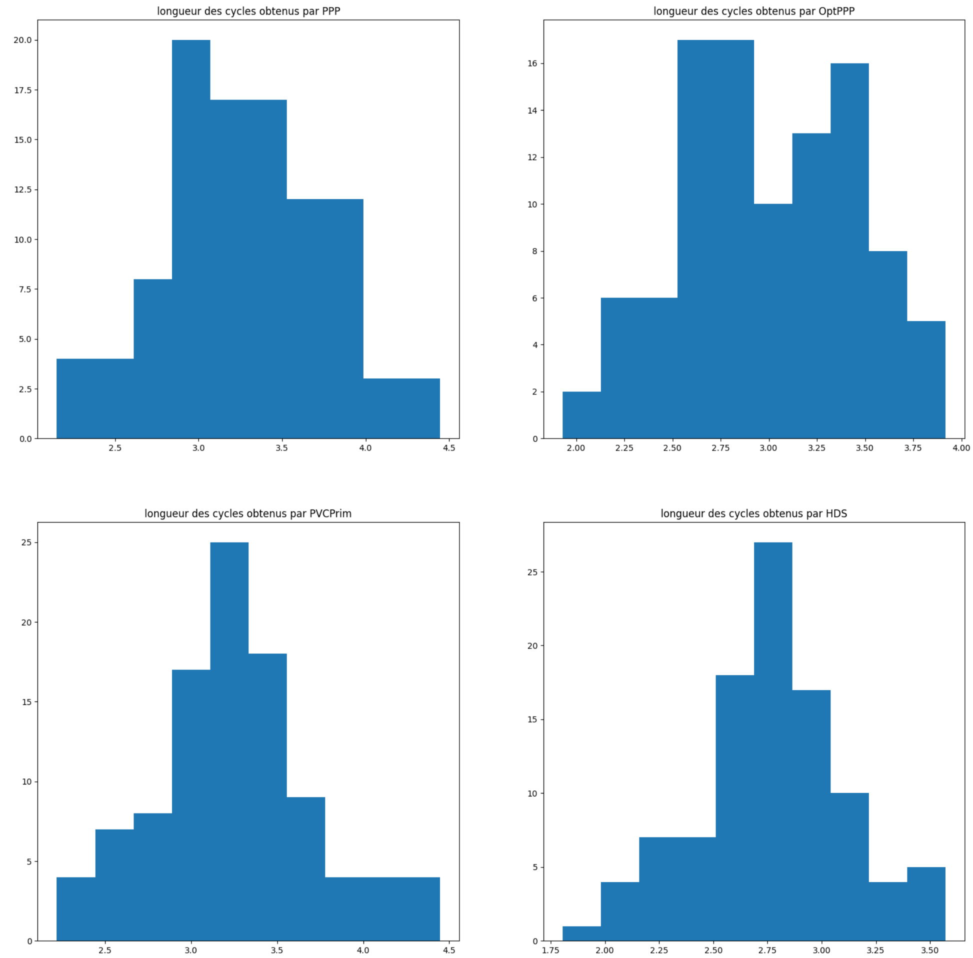

FIGURE 1 – Histogrammes des longueurs des cycles obtenus par les 4 algorithmes.

7.3 Analyse des résultats

- L'amélioration apportée par **OptPPP** par rapport à **PPP** est significative, avec un gain moyen de 7.95 %.
- **PVCPrim** permet de réduire la longueur du cycle obtenu par **PPP** mais elle donne de moins bons résultats que **OptPPP** avec des cycles 1.5 % plus longs.

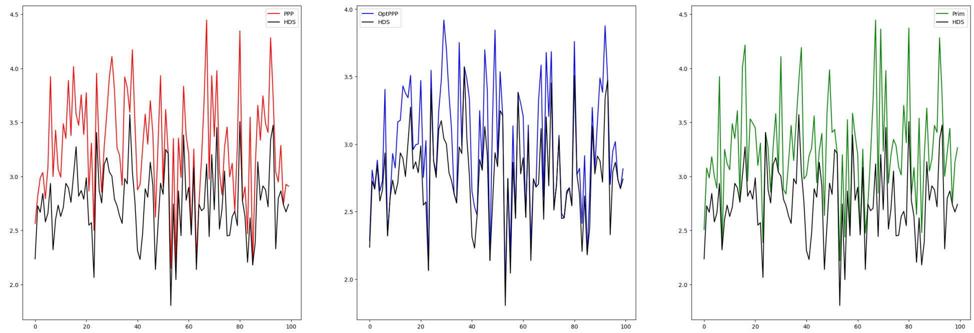

FIGURE 2 – Comparaison des 3 méthodes approximative avec HDS.

— L'algorithme exact **HDS** fournit la solution optimale, ce qui permet de comparer la qualité des heuristiques.
— L'étude statistique confirme que les heuristiques combinées à des améliorations locales offrent un bon compromis entre précision et temps de calcul.

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.

$$\boxed{\text{Complexité temporelle de PPP : } O(n^2)}$$

La complexité spatiale est dominée par le stockage de la matrice des distances, soit $O(n^2)$.

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.

$$\boxed{\text{Complexité temporelle de OptPPP : } O(n^3)}$$

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 $O(n)$.

$$\boxed{\text{Complexité temporelle de PVCPrim : } O(n \log n)}$$

La complexité spatiale est dominée par la structure du tas et la matrice des distances, soit $O(n^2)$.

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.

$$\boxed{\text{Complexité temporelle de HDS : } O(n!)}$$

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.