Rapport de projet : Résolution du Problème du Voyageur de Commerce

Mohamed Elakef Zenagui

1 Introduction générale

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 / OptPrim),
— un algorithme exact par *Branch and Bound* (HDS).

2 Structures de données algorithmiques

2.1 Structures générales

CONSTANTE **n** = 10

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
$c \leftarrow \emptyset$
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 :

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

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*, *pi*, P, S, P^*, S^* ; Entiers i_p, i_s
Var v : Entier
p : Acellule
$couleurs[origine] \leftarrow GRIS$
$i_p \leftarrow i_p + 1$
$P[origine] \leftarrow i_p$
$P^*[i_p] \leftarrow origine$
$p \leftarrow G.L[origine]$
while $p \neq NIL$ **do**
  $v \leftarrow p \uparrow sommet$
  if $couleurs[v] = BLANC$ **then**
    $pi[v] \leftarrow origine$
    Visiter_Profondeur(G, v, couleurs, pi, P, S, P^*, S^*, i_p, i_s)
  end
  $p \leftarrow p \uparrow suiv$
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 *pi*, P, S, P^*, S^*
Var$i, i_p, i_s : Entier$
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 : ↑ Arrete
T.dern ← G.n
for i ← 1 to G.n do
    G.cle[i] ← +∞
    G.π[i] ← −1
    G.NoeudTas[i] ← i
    T.Tab[i] ← i
end
G.cle[r] ← 0
while T.dern ≥ 1 do
    t ← T.Tab[1]
    Echange(1, T.dern, G, T)
    T.dern ← T.dern − 1
    Verslebas(1, G, T)
    p ← G.L[t]
    while p ≠ NIL do
        if G.NoeudTas[p ↑ sommet] ≤ T.dern et p ↑ poids < G.cle[p ↑ sommet] then
            G.cle[p ↑ sommet] ← p ↑ poids
            G.π[p ↑ sommet] ← t
            Verslehaut(G.NoeudTas[p ↑ sommet], G, T)
        end
        p ← p ↑ suiv
    end
end
```

Algorithm 5: Prim efficace

5.3 OptPrim

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, pi, 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: OptPrim : 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.

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 :↑ 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 ≠ NIL 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 points aléatoires 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.

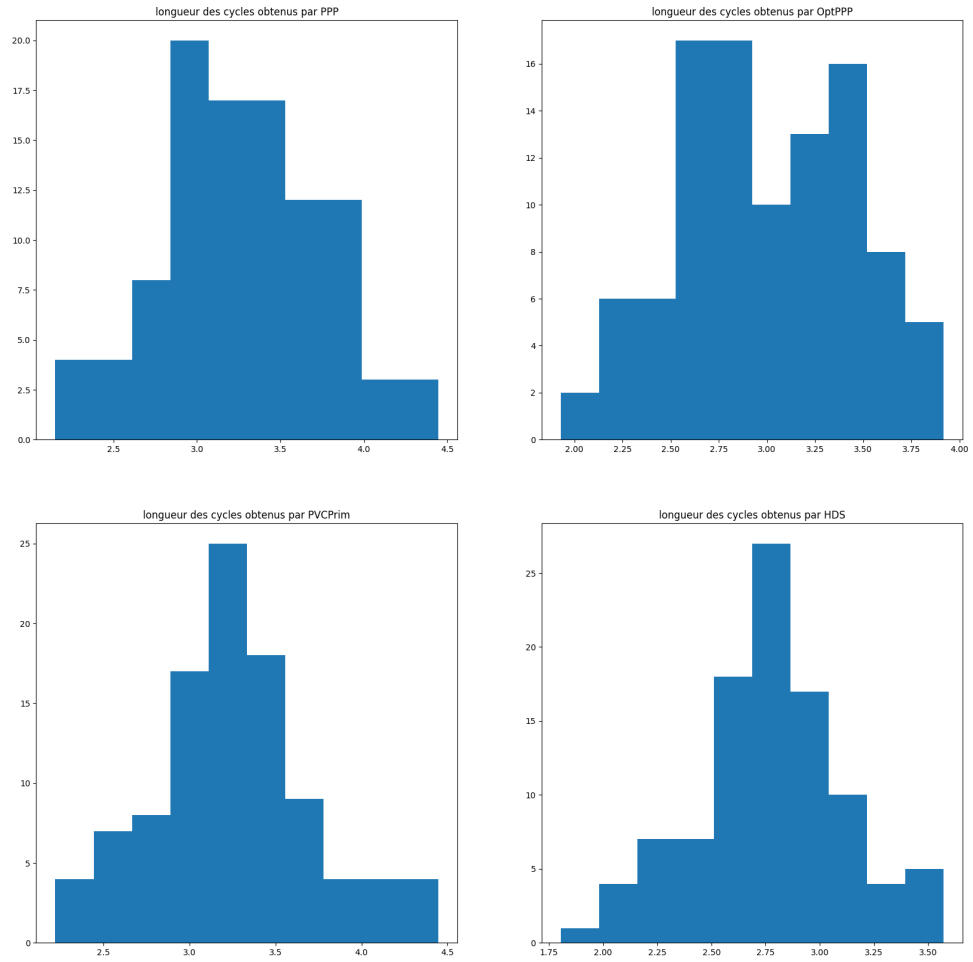

FIGURE 1 – Histogrammes des longueurs des cycles obtenus par les 4 algorithmes.

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 encore de réduire la longueur du cycle obtenu par **OptPPP**.
— 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.

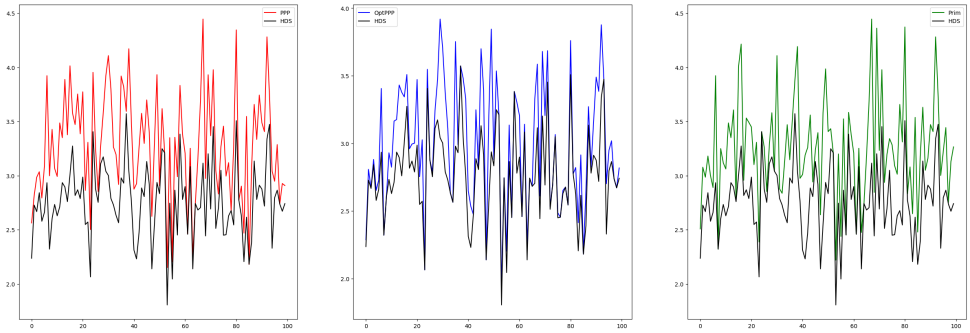

FIGURE 2 – Comparaison des 3 méthodes approximative avec HDS.

8 Conclusion générale

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.