C'est difficile de résoudre PVC en 60 secondes.

Pierre-Emmanuel Devin Dolunay Moustafa

Avril 2025

Contents

1 Introduction

Le problème du voyageur de commerce est un problème NP-complet, ce qui signifie que trouver une solution exacte nécessiterait un temps de calcul excessivement long. Pour contourner cette difficulté, il est préférable d'utiliser des algorithmes de recherche basés sur des méthodes aléatoires. Ces algorithmes

permettent d'obtenir une plus grande diversité de solutions dans un délai plus court.

Les algorithmes que nous avons utilisés et comparés dans notre étude sont les suivants : l'optimisation par colonies de fourmis, un algorithme évolutionnaire, et GRASP (Greedy Randomized Adaptive Search Procedures). Chacun de ces algorithmes a été amélioré grâce à l'algorithme d'optimisation pour le problème du voyageur de commerce connu sous le nom de *2-OPT*.

2 Presentation du probleme

Le problème du voyageur de commerce (PVC) est l'un des problèmes les plus étudiés en optimisation combinatoire et en informatique théorique. Il peut être formulé de manière simple mais possède une complexité computationnelle élevée ce qui rend le défit de le résoudre rapidement intéressant.

Le PVC peut être décrit comme suit : étant donné un ensemble de villes et les distances entre chaque paire de villes, le but est de trouver le plus court chemin possible permettant de visiter chaque ville exactement une fois et de revenir à la ville de départ. Mathématiquement, si nous avons n villes, nous pouvons représenter ce problème par un graphe complet pondéré où chaque ville est un sommet et chaque distance entre deux villes est une arête pondérée.

Ce problème est classé comme un problème NP-complet. Cela signifie qu'il n'existe pas d'algorithme connu capable de résoudre toutes les instances du problème en un temps polynomial. À mesure que le nombre de villes augmente, le temps nécessaire pour trouver la solution optimale croît de manière exponentielle. Cette complexité rend le PVC particulièrement difficile à résoudre pour de grandes instances.

3 2-OPT

Algorithm 1: Algorithme 2-opt pour optimiser des cycles hamiltoniens

```
1  Fonction 2-OPT(cycleHamiltonien):
       Data: cycleHamiltonien: tableau d'entiers
       Result: cycleHamiltonien optimisé
2      amélioration ← vrai
3      nbAmélioration← 0
4      while amélioration et nbAmélioration < 10 000 do
5          amélioration ← faux
6          for i ← 0 to taille de cycleHamiltonien −1 do
7              for j ← 0 to taille de cycleHamiltonien −1 do
8                  if i − 1 ≤ j ≤ i + 1 then
9                      on passe à l'itération suivante
10                 end
11                 a ← min i, j
12                 b ← max i, j
13                 xi ← sommet numéro a de cycleHamiltonien
14                 xi1 ← sommet numéro a + 1 de cycleHamiltonien
15                 xj ← sommet numéro b de cycleHamiltonien
16                 xj1 ← sommet numéro b + 1 de cycleHamiltonien
17                 if distance(xi, xj) + distance(xi1, xj1) ¡ distance(xi, xi1)
                     + distance(xj, xj1) then
18                     Inverser touts les sommets entre a + 1 et b
19                     amélioration ← vrai
20                     nbAmélioration ← nbAmélioration +1
21                 end
22             end
23         end
24     end
25     return hamiltonianCycle
```

L'algorithme $2-OPT$ est une méthode d'optimisation locale couramment utilisée pour améliorer les solutions du problème du voyageur de commerce. Son objectif principal est de "décroiser" les routes dans un cycle hamiltonien, c'est-à-dire de réduire la longueur totale du trajet en éliminant les intersections inutiles.

Cet algorithme est appliqué sur toutes nos optimisations pour permettre d'obtenir de meilleurs résultats plus rapidement. Il est particulièrement efficace pour améliorer les solutions initiales générées par d'autres algorithmes, tels que les algorithmes évolutionnaires, GRASP, et l'optimisation par colonies de fourmis (ACO). En décroisant les routes, $2-OPT$ réduit la longueur totale du trajet et améliore ainsi la qualité des solutions générées.

La limitation principale de $2-OPT$ est que l'algorithme est déterministe, donc il n'y a aucun espoir d'avoir de meilleur résultats, c'est pour cela que nous allons le combiner avec les algorithmes suivants.

4 GRASP

4.1 Algorithme

La procédure de recherche gloutonne adaptive et stochastique, plus communément connue sous le nom de GRASP (Greedy Randomized Adaptive Search Procedure), est une métaheuristique utilisée pour résoudre des problèmes d'optimisation combinatoire, dont le problème du voyageur de commerce. GRASP combine des éléments de recherche gloutonne et de recherche aléatoire pour explorer l'espace des solutions.

L'algorithme est une boucle sur les 60 secondes données de la démarche suivante :

4.1.1 Phase gloutonne stochastique

GRASP commence par construire une solution initiale en utilisant une approche gloutonne stochastique. Contrairement à une méthode purement gloutonne qui choisit toujours l'option la plus avantageuse, GRASP introduit un élément d'aléatoire. À chaque étape de la construction, une ville est ensuite choisie aléatoirement parmi toutes les autres villes, de manière inversement proportionelle à la distance.

4.1.2 Phase de recherche locale

Une fois la solution initiale construite, GRASP utilise une méthode de recherche locale pour l'améliorer. Dans ce contexte, l'algorithme de Hill Climbing est appliqué. Le Hill Climbing consiste à explorer les voisins de la solution actuelle et à se déplacer vers une solution meilleure si elle est trouvée.

4.1.3 Application de 2-OPT

Une fois que la solution générée par le Hill Climbing est stagnante, nous appliquons 2-OPT sur celle-ci. Elle est ensuite enregistrée par le programme préécrit pour être évaluée.

4.2 Performance

La figure 1 représente la moyenne des résultats de l'algorithme GRASP sur 4 ensembles de points en 10x60 secondes.

5 Algorithme évolutionaire

5.1 Algorithme

Les algorithmes évolutionnaires s'inspirent de la théorie de l'évolution de Darwin. Ils intègrent des concepts tels que les individus, l'environnement, la sélection

```
tsp.projects.competitor.grasp.Grasp
bier127 120024.18165033737
gr666   3285.300691656328
pr2048  4094.0
test10  7576.8211310058205
```

Figure 1: Performances de GRASP sur les 4 ensembles de points testés, moyenne de 10×60 secondes.

naturelle, l'hérédité et la mutation. Ces algorithmes sont particulièrement adaptés aux problèmes d'optimisation complexes.

L'algorithme débute avec une population initiale générée aléatoirement. Chaque individu représente une solution potentielle au problème. À chaque itération, les individus sont évalués à l'aide d'une fonction de coût. Les meilleurs d'entre eux sont sélectionnés pour engendrer une nouvelle génération à travers des opérations de croisement et de mutation. Pour améliorer encore les solutions, l'algorithme applique l'heuristique 2-OPT à chacune d'entre elles.

5.2 Exploration et exploitation

L'algorithme équilibre deux mécanismes complémentaires :

- L'exploration : obtenue par la mutation d'individus sélectionnés, permettant de découvrir de nouvelles zones de l'espace de recherche.
- L'exploitation : assurée par le croisement entre plusieurs individus, afin de combiner les meilleures caractéristiques des solutions existantes.

5.2.1 Opérateur de mutations utilisés dans le cadre du PVC :

- échange de deux villes adjacentes $< A, B, C, D, E > \rightarrow < A, B, D, C, E >$
- échange de n villes adjacentes $< A, B, C, D, E > \rightarrow < D, C, B, A, E >$
- échange de 2 villes aléatoire $< A, B, C, D, E > \rightarrow < A, E, C, D, B >$

5.2.2 Opérateur de croisement utilisés dans le cadre du PVC :

- (p_1, p_2) couple de parents. Sélection de villes dans p_1 à leur emplacement, ajout des villes non sélectionnées dans les trous dans l'ordre de P2, $< A, B, C, D, E > + < C, A, B, E, D > \rightarrow < A, B, C, E, D >$

```
tsp.projects.competitor.evol2.Evol2
bier127 134758.26177733685
gr666   3871.4845179806784
pr2048  4704.800632649426
test10  7576.8211310058205
```

Figure 2: Performances de notre algorithme évolutionaire sûr les 4 ensembles de points testés, moyenne de 10×60 secondes.

5.3 Performance

La figure 2 est les performances de l'algorithme évolutionaire sur les 4 ensembles de points. Nous pouvont voir que l'approche GRASP surpasse l'aproche évolutionnaire sur toutes les instances. Il parvient mieux a exploiter l'espace de recherche en un temps plutôt cours, et l'algorithme évolutionaire doit aussi arriver à des optimums locaux plus lentement. Peut-être que c'est dû au problème qui convient mieux à du Hill Climbing, sachant que les algorithmes évolutionaires sont aussi efficaces dans d'autres dommaines.

6 Optimisation par les colonies de fourmis

6.1 Algorithme

L'optimisation par colonies de fourmis (Ant Colony Optimization, ou ACO) est une métaheuristique inspirée du comportement collectif des fourmis pour trouver des chemins courts entre leur nid et une source de nourriture. Ce principe repose sur l'usage de phéromones : lorsqu'une fourmi trouve un bon chemin, elle le marque, incitant d'autres fourmis à l'emprunter à leur tour. Plus un chemin est emprunté, plus la concentration de phéromones augmente, ce qui renforce son attractivité.

Dans notre implémentation pour le problème du voyageur de commerce, chaque fourmi construit un parcours en choisissant à chaque étape la ville suivante selon une probabilité basée sur deux facteurs :

- La quantité de phéromones sur une arête.
- L'inverse de la distance, pour une préférence pour les trajets cours.

avec des valeurs α de 1 et β de 5.

Dans notre implémentation, l'ACO permet à une quantité de 101 formis, donc chaque itération.

Chaques itération ressemble à :

- Chaque fourmis prend un chemin, selon les conditions précédente.

```
tsp.projects.competitor.aco.ACO
bler127 119234.63072849852
gr666   3282.6723390289494
pr2048  4094.0
test10  7576.8211310058205
```

Figure 3: Performances de notre optimisation par colonie de fourmis sûr 4 ensembles de points testés, moyenne de 10 × 60 secondes.

- Ajouts des phéromones des fourmis de l'itération courrante.
- Évaporation des phéromones (le taux *rho* d'évaporation est de 0.5)

Nous avons aussi ajouté une *Super Fourmis* qui a le chemin le plus cours, et dont les phéromones sont 5 fois plus puissance, ce qui permet d'augmenter le poid de l'exploitation, et de réduire celui de l'exploration.

6.2 Performance

L'approche ACO se montre compétitive sur toutes les instances testées. Elle obtient notamment le meilleur score sur bler127, dépassant légèrement GRASP. Sur les autres cas (gr666, pr2048, test10), ses performances sont très proches, mais légèrement supérieure à celles de GRASP, et supérieures à celles de l'algorithme évolutionnaire.

Nous pouvons donc voir la robustesse de la méthode ACO, particulièrement efficace dans l'exploration de l'espace de solutions grâce à la dynamique des phéromones. La combinaison avec 2-OPT permet également de stabiliser et d'affiner les résultats. Ce bon équilibre entre exploration globale et exploitation locale en fait une méthode très adaptée au PVC, même dans un cadre temporel restreint.

7 Conclusion

Dans ce rapport, nous avons comparé trois approches heuristiques pour résoudre le problème du voyageur de commerce (PVC) dans une limite de temps de 60 secondes : GRASP, un algorithme évolutionnaire, et l'optimisation par colonies de fourmis (ACO). Chacune de ces approches a été renforcée par l'algorithme local 2-OPT pour affiner les solutions proposées.

Les résultats montrent clairement que l'algorithme par colonies de fourmis est celui qui donne les meilleures performances globales. Il se distingue particulièrement sur des instances de grande taille comme bler127, où il obtient le score le plus bas. Grâce à son mécanisme d'exploration guidée par les phéromones et

sa capacité d'exploitation via 2-OPT, ACO parvient à un bon équilibre entre recherche globale et optimisation locale.

L'approche GRASP reste une méthode efficace, en particulier par sa capacité à construire rapidement des solutions de bonne qualité. Toutefois, elle est légèrement surpassée par ACO sur plusieurs instances, ce qui montre que la dynamique collective de la colonie permet une exploration plus fine dans le même laps de temps.

L'algorithme évolutionnaire, quant à lui, a produit des résultats globalement moins bons dans le cadre temporel imposé. Bien qu'il soit robuste et flexible, sa convergence plus lente semble l'avoir pénalisé dans ce contexte. Il reste cependant intéressant dans des situations où l'on dispose de davantage de temps ou pour des problèmes aux structures plus variées.