

Projet Métaheuristique - Voyageur de commerce
"Genetic-Hill"

Pengxiao SHI
`pengxiao.shi6@etu.univ-lorraine.fr`

Thomas SKORLIC
`thomas.skorlic8@etu.univ-lorraine.fr`

12 Mars 2021

Table des matières

1	Introduction	2
2	Nos Recherches	2
2.1	Grasp	2
2.2	Colonies de fourmis	4
3	Notre algorithme génétique : Genetic-Hill	4
3.1	Initialisation des individus	5
3.2	Multi-Élitisme	7
3.3	Mutations	7
3.4	Sélection par tournoi	8
3.5	Résultat	8
4	Conclusion	10
	Références	10

1 Introduction

Le but de ce projet est de trouver une solution pour le problème bien connu en algorithmique et en optimisation : le voyageur de commerce.

Un voyageur doit passer par plusieurs villes, exactement une fois chacune, puis retourner à sa ville de départ. Son but est de parcourir la distance la plus courte possible.

Afin de trouver une solution satisfaisante pour ce problème, nous cherchons à développer un algorithme métaheuristique ; c'est à dire un algorithme d'optimisation pour résoudre un problème "difficile", pour lesquels on ne peut pas trouver une solution exacte en un temps raisonnable.

Pour le développement de notre projet, nous disposons de trois fichiers contenant des coordonnées de villes. Nous serons évalué sur la solution que nous renvoie notre algorithme sur ces trois instances ainsi que sur une instance que nous ne connaissons pas.

2 Nos Recherches

2.1 Grasp

Nous avons d'abord pensé à utiliser l'algorithme GRASP pour initialiser nos individus de notre algorithme génétique. Cette algorithme consiste à choisir aléatoirement la première ville de départ. Ensuite on choisi la ville suivante avec la probabilité (1) dans la liste RCL (2) :

$$p(v_i) = \begin{cases} \frac{e^{-d(v_i, s_k)}}{\sum_{v_j \notin \{S_1, S_2, \dots, S_k\}} e^{-d(v_j, s_k)}} & \text{Si } v_i \notin \{S_1, S_2, \dots, S_k\} \\ 0 & \text{Sinon} \end{cases} \quad (1)$$

$$RCL = \{v_i \in V | e^{-d(v_i, s_k)} \leq C_{min} + \alpha * (C_{max} - C_{min})\} \quad (2)$$

V est la liste des villes non visitées

$$C_{min} = \min\{e^{-d(v_i, s_k)}\}$$

$$C_{max} = \max\{e^{-d(v_i, s_k)}\}$$

α une constante comprise entre 0 et 1

Algorithm 1: Algorithmme Grasp

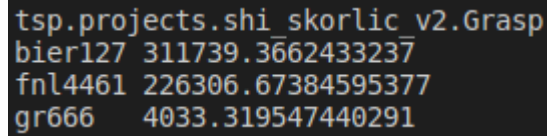
```
1 Fonction Grasp(nbVilles : entier) : tableau
2 begin
    // Initialisation de tableau contenant le chemin final
3   path ← [ ]
4
    // Initialisation de tableau des villes non visitées
5   lesVillesNonVisitees ← [ ]
6   for i de 0 à nbVilles - 1 do
7       | lesVillesNonVisitees.add(i)
8   end
9
    // Choix de la première ville
10   $S_k \leftarrow \text{genererUnNombre}(0, \text{nbVilles} - 1)$ 
11  path.add( $S_k$ )
12  lesVillesNonVisitees.remove( $S_k$ )
13
    // Construction de path: selon  $p(v_i)$  et RCL
14  for i de 1 à nbVilles - 1 do
15      | // Initialisation de RCL
16      | RCL ← [ ]
17
18      // Calcul de  $C_{min}$  et  $C_{max}$ 
19      minMax ← calculerMinMax(lesVillesNonVisitees,  $S_k$ )
20       $C_{min} \leftarrow \text{minMax}[0]$ 
21       $C_{max} \leftarrow \text{minMax}[1]$ 
22
23      // Calcul de RCL
24      for  $v_i \in \text{lesVillesNonVisitees}$  do
25          | candidat ←  $e^{-d(v_i, s_k)}$ 
26          | if  $\text{candidat} \leq C_{min} + \alpha * (C_{max} - C_{min})$  then
27              | RCL.add( $v_i$ )
28          end
29      end
30  end
```

```

28
29     // Piocher une ville dans RCL selon la probabilité
        p( $v_i$ )
30      $S_k \leftarrow$  choisir(lesVillesNonVisitees, RCL,  $S_k$ )
31     path.add( $S_k$ )
32     lesVillesNonVisitees.remove( $S_k$ )
33 end
34
    // retourner le résultat
35 return path
36 end

```

Cependant notre algorithme Grasp a donné des résultats très mauvais sur l'instance bier127 particulièrement, donc nous n'avons pas retenu cette méthode pour initialiser nos individus.



```

tsp.projects.shi_skorlic v2.Grasp
bier127 311739.3662433237
fnl4461 226306.67384595377
gr666 4033.319547440291

```

FIGURE 1 – Résultats obtenu par Grasp

2.2 Colonies de fourmis

Après avoir développé l'algorithme que nous rendons aujourd'hui, on s'est renseigné sur les algorithmes de colonies de fourmis pour éventuellement améliorer notre résultat. Nous avons cependant manqué de temps pour terminer notre algorithme qui n'était pas au point donc on s'est reconcentré sur notre algorithme génétique.

3 Notre algorithme génétique : Genetic-Hill

Notre algorithme que nous rendons est basé sur un système de générations. Pour créer une nouvelle génération, nous utilisons des mutations et des élitismes de la population actuelle. Notre population est constituée de

80 individus. D'après nos tests, ce nombre d'individus nous a globalement renvoyé les meilleurs résultats, même si les différences étaient plutôt faibles.

3.1 Initialisation des individus

Pour avoir un algorithme génétique plus efficace, il nous semblait important de ne pas générer les individus de notre population aléatoirement. Nous avons décidé d'initialiser nos individus avec la méthode du Hill-Climbing. Un individu est initialisé par notre méthode du "Hill Climbing", tous les autres sont générés par une mutation de ce premier individu.

Algorithm 2: Algorithme de Hill-Climbing

```
1 Fonction HillClimbing(nbVilles : entier) : tableau
2 begin
3     // Initialisation de tableau contenant le chemin final
4     path  $\leftarrow$  [ ]
5
6     // Initialisation de tableau contenant les villes non
7     // visitées
8     lesVillesNonVisitees  $\leftarrow$  [ ]
9     for  $i$  de 0 à nbVilles - 1 do
10         // Au départ, aucune ville n'est visitée
11         lesVillesNonVisitees.add(i)
12     end
13
14     // Choix aléatoire de la première ville  $S_0$ 
15      $S_0 \leftarrow$  genererUnNombre(0, nbVilles - 1)
16     path[0]  $\leftarrow$   $S_0$ 
17     lesVillesNonVisitees.remove( $S_0$ )
18
19     // Construction de "path": chaque ville  $S_k$  est obtenue
20     // en cherchant, parmi les villes non visitées, la
21     // ville la plus proche de  $S_{k-1}$ 
22     for  $k$  de 1 à nbVilles - 1 do
23          $S_{k-1} \leftarrow$  path[k-1]
24          $S_k \leftarrow$  chercherLaVilleLaPlusProche( $S_{k-1}$ ,
25             lesVillesNonVisitees)
26         path[k]  $\leftarrow$   $S_k$ 
27         lesVillesNonVisitees.remove( $S_k$ )
28     end
29
30     // retourner le path
31     return path
32 end
```

Nous nous sommes ensuite rendu compte que cette méthode renvoyait une évaluation différente selon le point de départ. Par exemple, si nous initialisons la ville de départ à 0 pour chacun des trois fichiers, nous trouvons les évaluations suivantes :

```
tsp.projects.shi_skorlic.HillClimbing2
bier127 147312.7210414805
fnl4461 227156.6074673403
gr666 4110.899863988607
```

FIGURE 2 – La ville de départ est la ville numéro 0

Ensuite, si nous générons la ville de départ aléatoirement pour chacun des trois fichiers, nous trouvons les évaluations suivantes :

```
tsp.projects.shi_skorlic.HillClimbing2
bier127 141056.84124718903
fnl4461 222866.2042405449
gr666 3835.215131467921
```

FIGURE 3 – La ville de départ est aléatoire

On a donc eu l'idée de chercher, pour chacun des 3 fichiers de données, quelle était la ville de départ la plus avantageuse. Nous avons trouvé que pour le fichier de données bier127, la ville de départ la plus intéressante est la ville numéro 116, pour le fichier fnl4461 c'est la ville numéro 4129, et pour le fichier gr666 c'est la ville numéro 74. Pour le fichier de données "surprise", la ville de départ sera donc initialisée aléatoirement.

3.2 Multi-Élitisme

Nous avons créé une méthode qui nous permet de garder un nombre N d'individus de la génération actuelle vers la population suivante. Ces individus correspondent aux meilleurs individus de notre population. Nous avons fixé N à 6 (de manière plutôt arbitraire, bien qu'une valeur <4 ou >8 nous donnait des résultats plutôt mauvais).

3.3 Mutations

Nous avons d'abord effectué un premier algorithme de mutation qui se contentait d'inverser les positions de deux villes aléatoires. Ensuite on a effectué un algorithme de mutation qui sélectionne deux villes aléatoirement puis

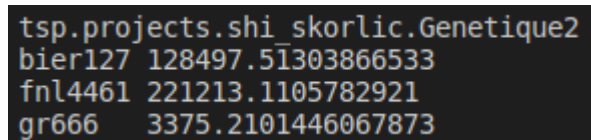
inverse l'ordre de toutes les villes entre ces deux villes. Nous nous sommes rendu compte que le premier algorithme n'apportait pas grand chose étant donné la taille des individus, donc qu'il n'était pas nécessaire de le garder.

3.4 Sélection par tournoi

Afin de choisir quels individus de la population nous allons faire muter, nous avons implémenter une sélection par tournoi. Nous sélectionnons aléatoirement M individus parmi notre population, nous sélectionnons le meilleur parmi les M , puis on le fait muter. Nous avons fixé ce M à 5.

3.5 Résultat

Nous avons décidé de rendre un algorithme qui n'effectue pas de croisements entre les individus, car malgré nos tentatives, cela a toujours dégradé notre solution. Notre algorithme génère une nouvelle population uniquement en effectuant des mutations sur les individus ainsi que de l'élitisme. En ayant pris soin d'initialiser nos individus correctement, nous trouvons des résultats qui nous semblent plutôt satisfaisants.



```
tsp.projects.shi skorlic.Genetique2  
bier127 128497.51303866533  
fnl4461 221213.1105782921  
gr666 3375.2101446067873
```

FIGURE 4 – Résultats obtenus par notre algorithme

Malgré les multiples générations, nous n'arrivons pas à obtenir une nette amélioration sur l'ensemble fnl4461 mais les résultats sur bier127 et gr666 sont bien mieux que ceux obtenus par notre HillClimbing de départ.

Globalement notre algorithme se présente comme ça :

Algorithm 3: Algorithmme Genetic-Hill

```
1 Fonction Genetic-Hill(nbVilles : entier) : tableau
2 begin
    // Étape 1: Initialisation de la population de départ
3   population ← [ ] // population est un tableau de path
4
    // le premier individu est le path donné par
    Hill-Climbing
5   hillClimbing ← HillClimbing(nbVilles)
6   population[0] ← hillClimbing
7
    // les autres individus sont les mutants de
    hillClimbing
8   for i de 1 à nbIndividus - 1 do
9       | population[i] ← mutant(hillClimbing)
10  end
11
    // Étape 2: Création d'une nouvelle génération
12  nouvellePopulation ← [ ]
13
    // Étape 2.1: Multi-Élitisme
14  lesElites ← elitisme(population) // "lesElites" contient les
    "nbElites" élites de la population
15  for i de 0 à nbElites - 1 do
16      | nouvellePopulation[i] ← lesElites[i]
17  end
18
    // Étape 2.2: Mutation
19  for i de nbElites à nbElites+nbMutants - 1 do
20      | meilleurIndividu ← selection(population)
21      | nouvellePopulation[i] ← mutant(meilleurIndividu)
22  end
23  population ← nouvellePopulation
24
    // Étape 3: retourner le chemin
25  return population[0] // car on fait en sort que le
    meilleur path reste en tête
26 end
```

4 Conclusion

Bien que nous ayons déjà étudié théoriquement le problème du voyageur de commerce les années précédentes, nous avons pu mettre en oeuvre une solution grâce à ce projet. Nous avons pu étudier des algorithmes que nous ne connaissions pas comme l'algorithme du recuit simulé, le grasp ou encore la recherche tabou. On pense que l'on aurait pu rendre un algorithme de colonies de fourmies donnant un résultat au moins équivalent ou meilleur que celui-ci si nous avions disposé d'un peu plus de temps car cette algorithme se prête particulièrement bien au problème du voyageur de commerce.

Références

- [1] Les optimums des données : <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf>
- [2] SAILLOT Mathis, Je descent de la colline à cheval, 2020