Projet RP
Arbre de Steiner et recherche local

Loic URIEN et Victor RACINE

2018

Table des matières

Chapitre 1

Introduction

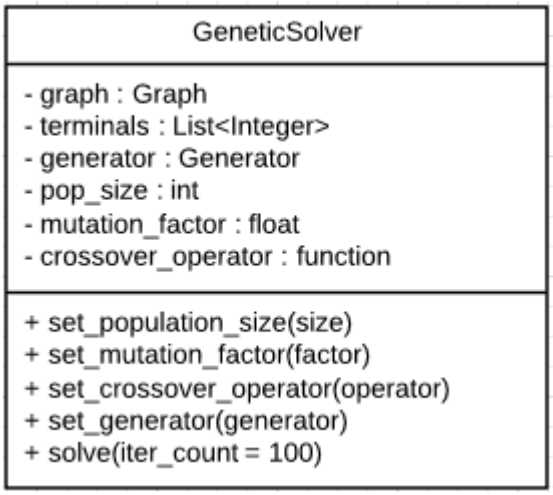
Dans le cadre de ce projet, nous nous sommes intéressé au problème des arbres de Steiner, en particulier à la détermination, à partir d'un graphe donné, d'un arbre de Steiner de poids minimum. Par soucis de brièveté, nous ne rappellerons pas la formalisation du problème de l'arbre de Steiner de poids minimum ici.
Ce problème étant NP-difficile, nous nous sommes assurés de mettre au point des méthodes approchées, a priori sans garantie de performances, qui permettront de déterminer une solution approchée au problème donné dans des temps raisonnables. Dans un premier temps nous avons implémenté une méthode de résolution par algorithme génétique simple que nous avons ensuite agrémentée de plusieurs heuristiques afin d'améliorer la vitesse de convergence. Nous avons ensuite comparé les performances de cette méthode avec une méthode de recherche locale simple avec redémarrage. L'ordre des chapitres suit l'ordre dans lequel l'équipe a implémenté ces différents algorithmes.

Chapitre 2

Algorithme génétique

2.1 Présentation globale de l'algorithme

Dans cette section nous allons présenter l'algorithme génétique implémenté pour ce projet. La classe centrale se chargeant de l'exécution de l'algorithme génétique est la classe *GeneticSolver*. Ci-dessous, l'architecture globale de cette classe :



Nous avons adopté les conventions Java pour cette représentation ainsi que toutes les autres, bien que le code soit implémenté en python. La classe solveur prend en paramètre un graph et une liste de sommets terminaux et la méthode *solve*() se charge d'exécuter l'algorithme. L'algorithme génétique peut être généralisé comme suit :

Algorithm 1: Algorithme génétique basique

Data: $G = (V, E)$, $T \subseteq V$, $iter_count \in \mathbb{N}$
Result: une solution au problème de l'arbre de Steiner de poids minimal

```
population ← generator.generate(population_size)
for i ∈ range(iter_count) do
    for individual ∈ population do
        g ← compute_graph(individual)
        compute_fitness(g)
    end
    population ← replacement_operator(population)
end
return individual with best fitness (minimum in our case)
```

Nous détaillerons le remplacement de la population dans la prochaine section. Dans sa forme la plus basique, l'algorithme opère un remplacement total de la population, avec croisement en un point et génération de la population purement aléatoire.

2.2 Croisement et remplacement de la population

Les croisements et remplacements de population sont les parties centrales de l'algorithme génétique. En ce qui concerne les croisement, la classe *GeneticCrossoverOperators* agit comme un utilitaire se chargeant de regrouper toutes les fonctions de croisement. Dans le cadre de ce projet nous en avons implémenté deux :

— croisement en un point (one point crossover) qui, étant donné deux parents, sélectionne aléatoirement un point c dans le code génétique et créé deux enfants en collant les bouts de code génétique des parents

— croisement en deux points (two points crossover) qui, étant donné deux parents, sélectionne aléatoirement deux points $c1$ et $c2$ dans le code génétique et créé deux enfants en prenant le code génétique d'un parent hors de l'intervalle $[c1, c2]$ et le code génétique de l'autre parent à l'interieur de cet intervalle

Après expérimentation, le croisement en deux points s'est révélé généralement plus efficace, donnant une meilleure vitesse de convergence, il semblerait cependant que changer de méthode de croisement influe relativement peu sur la solution obtenue. Nous avons ici donné les algorithmes des deux méthodes :

Algorithm 2: Croisement en un point

Data: $p1$ et $p2$ deux parents sous forme de code génétique
Result: deux enfants $c1$ et $c2$

$crossover\_point \leftarrow random\_index(p1)$
$c1 \leftarrow p1[0 : crossover\_point] + p2[crossover\_point : len(p2)]$
$c2 \leftarrow p2[0 : crossover\_point] + p1[crossover\_point : len(p1)]$
$return\ c1, c2$

Algorithm 3: Croisement en deux points

Data: $p1$ et $p2$ deux parents sous forme de code génétique
Result: deux enfants $c1$ et $c2$

$crossover\_point1 \leftarrow random\_index(p1)$
$crossover\_point2 \leftarrow random\_index(p1)$
$c1 \leftarrow p1[0 : crossover\_point1] + p2[crossover\_point1 : crossover\_point2] + p1[crossover\_point2 + 1 : len(p1)]$
$c1 \leftarrow p2[0 : crossover\_point1] + p1[crossover\_point1 : crossover\_point2] + p2[crossover\_point2 + 1 : len(p2)]$
$return\ c1, c2$

En ce qui concerne le remplacement de population, nous avons bien vite abandonné la méthode de remplacement total de la population, celle-ci étant observablement relativement mauvaise, en faveur d'un remplacement élitiste de la population par jeu de roulette.

2.3 Génération de la population initiale

Dans notre programme, le solveur est paramétrable et accepte un générateur en entré. La seule exigence est que ce générateur ait une méthode $generate(size)$ qui prend en entrée la taille de la population à générer. Dans notre code nous avons les générateurs suivants :

- *PopulationGenerator*, un générateur purement aléatoire paramétrisable (le générateur aléatoire peut suivre une autre loi que la loi uniforme). La génération s'effectue de manière assez simple : pour chaque gêne on génère un nombre aléatoire et à partir d'un certain palier, on affecte 1 à ce gêne, 0 sinon
- *ShortestPathPopulationGenerator*, un générateur basé sur l'heuristique du plus court chemin (voir section 3.1)
- *ShortestSpanningTreePopulationGenerator*, un générateur basé sur l'heuristique de l'arbre couvrant de poids minimal (voir section 3.2)
- *MixedPopulationGenerator*, un générateur dont le seul but est de "mixer" les générateurs précédents. Il accepte en entrée une liste de générateurs et un tuple contenant les proportions de ces générateurs. La génération

s'effectue selon la formule suivante :

$$\sum_{i \in 0 \ldots len(list_gen)} list\_gen[i].generate(size * proportions[i])$$

Ce dernier générateur est particulièrement utile pour les instances de grande taille, lorsque l'on veut pouvoir garder la puissance des heuristiques sans perdre trop de temps à générer toute la population heuristiquement. Dans nos tests nous avons donc tenté de générer une partie de la population de manière heuristique et une autre de manière purement aléatoire sur les instances de grandes tailles. Pour plus d'informations sur les résultats, veuillez vous référer à la section sur les résultats numériques.

Chapitre 3

Heuristiques de construction

Dans cette section nous commentons rapidement nos observations sur les différentes heuristiques proposées dans le sujet. Nous ne donnons pas de détails d'implémentation ni ne redonnons la définition des heuristiques, celles-ci étant explicitées dans l'énoncé du projet.

3.1 Heuristique du plus court chemin

Cette heuristique produit globalement d'excellents résultats même sur les instances les plus grandes. Ceci dit elle a un très gros défaut : la construction du graphe complet des chemins de poids minimal est très couteuse ! Etant donné que nous devons lancer un algorithme de Dijkstra pour chaque noeud terminal de complexité $O(|E|+|V|\log|V|)$, nous avons une complexité $O(|T|(|E|+|V|\log|V|))$. Pour les instances de grande taille ayant de plus un grand nombre de noeuds terminaux, cette méthode prendra un temps considérable. Cette méthode donc, bien qu'efficace, est très sensible au nombre de noeuds terminaux et est donc un assez mauvais candidat pour les instances assez grandes avec un nombre de noeuds terminaux conséquents. Ceci étant, dans nos test nous avons préféré cette méthode lors de l'utilisation de la recherche locale, celle-ci étant très sensible à l'initialisation.

3.2 Heuristique de l'arbre couvrant minimal

Cette heuristique, bien que moins efficace que l'heuristique du plus court chemin, montre des performances acceptables pour des instances assez grandes, ce qui en fait un bon candidat pour compléter une population aléatoire de manière intelligente en gardant un temps d'exécution raisonnable. Dans nos tests nous l'avons souvent utilisé pour créer 20% de la population initiale même lorsque l'instance étudiée était grande.

Chapitre 4

Recherche locale

Notre algorithme de recherche locale est excessivement simple et se déroule de la manière suivante :

- on génère une solution admissible, le plus souvent par heuristique de construction
- on énumère les voisins et on choisit, s'il existe, un voisin admissible ayant un meilleur score
- tant qu'il existe encore des voisins admissibles de meilleur score on recommence

Voici la formalisation algorithmique de la méthode :

Algorithm 4: Recherche locale

Data: $G = (V, E)$, $T \subseteq V$, max_time
Result: G' une solution approchée au problème

```
1  all_best ← ∅
2  all_best_fitness ← ∅
3  while current_time ≤ max_time do
4      initial ← heuristic_generator.generate(1)
5      current ← initial
6      while current ≠ initial do
7          neighbours ← compute_neighbours(initial)
8          current_best_fitness ← compute_fitness(initial)
9          for individual ∈ neighbours do
10             fitness ← compute_fitness(individual)
11             if (is_admissible(individual)&&fitness <
                current_best_fitness) || all_best = ∅ then
12                 current_best_fitness
13             end
14         end
15         if current_best = ∅||current_best_fitness < all_best_fitness
            then
16             all_best ← current
17             all_best_fitness ← current_best_fitness
18         end
19     end
20 end
21 return current
```

Chapitre 5

Résultats numériques

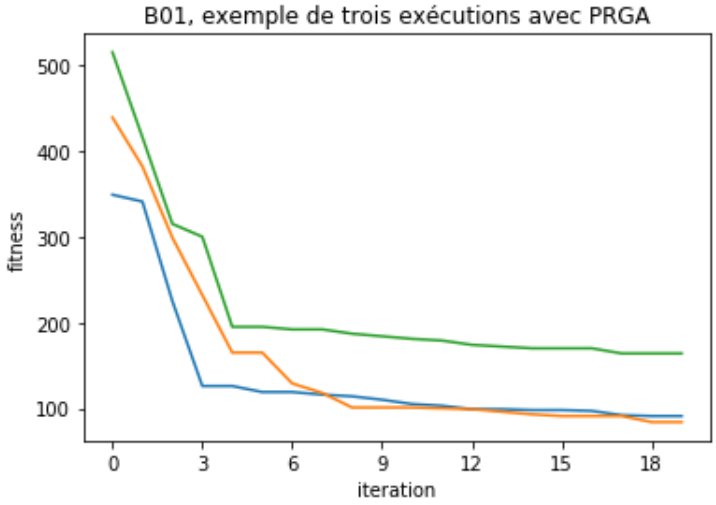
Pour tous les tests dans la suite, la machine utilisée est un Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz tournant sous Windows 7. Dans les tableaux de résultats, les différentes machines utilisées seront indiquées.

5.1 Algorithme génétique

Ici nous présenterons nos résultats pour l'algorithme génétique utilisant une génération de population purement aléatoire, suivant une loi uniforme. Nous référencerons dans la suite cet algorithme en tant que $PRGA$. Tous les résultats sont donnés avec un solveur générant une population de 100 individus et ayant un facteur de mutation de 0.01. Le générateur de population est entièrement uniforme, a un "jitter" de 0.4, un "threshold" de 0.35 et une pénalité de 3. Nous avons en général tenté de conserver le nombre d'itération au minimum. L'opérateur de croisement est l'opérateur de croisement en deux points. Nous pouvons observer ci-dessous les résultats de trois exécutions pour les instances $B01$ et $B02$

Notre algorithme $PRGA$ naïf donne de très bons résultats, sur ces instances de petite taille, ceci étant il échoue assez vite lorsque la taille de l'instance augmente ou simplement la topologie du graphe change. Ici nous avons essayé de trouver une solution pour l'instance $B10$:
L'algorithme met non seulement longtemps à converger, mais il se retrouve de plus très souvent dans un minimum local sans jamais en sortir (augmenter le taux de mutation aide certaines fois, mais la convergence reste excessivement lente). Nous proposons ainsi de tester deux autres algorithmes :

— Un algorithme utilisant l'heuristique de l'arbre couvrant de poids minimum pour générer toute la population initiale. Appelons cet algorithme $STGA$
— Un algorithme générant la population entière en utilisant l'heuristique

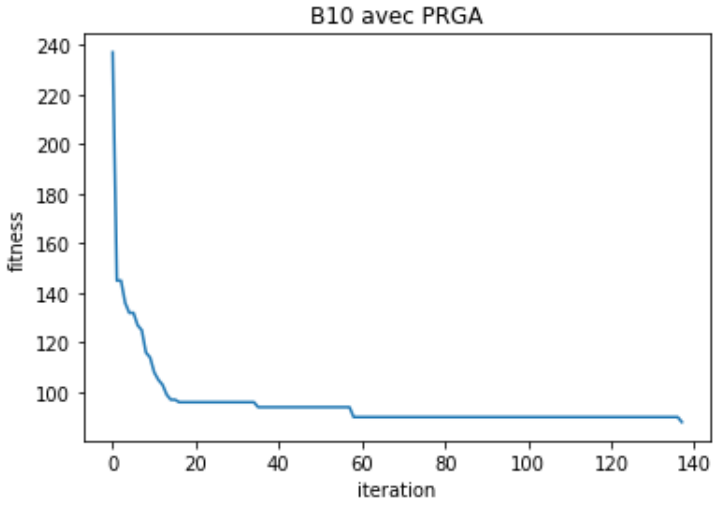


du plus court chemin. Appelons cet algorithme $SPGA$

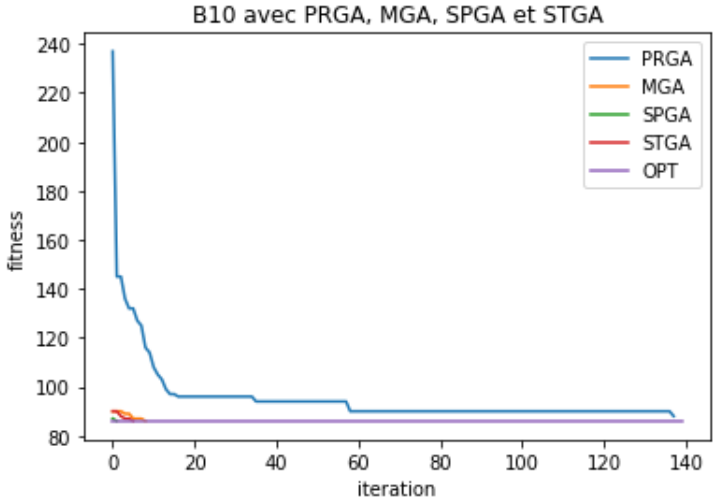
— Un algorithme mixte générant 5% de la population en utilisant l'heuristique du plus court chemin, 15% en utilisant l'heuristique de l'arbre couvrant de poids minimum et les 80% restants à l'aide du générateur aléatoire utilisé pour $PRGA$. Appelons cet algorithme $MGA1$

— Un algorithme autre mixte générant 20% en utilisant l'heuristique de l'arbre couvrant de poids minimum et les 80% restants à l'aide du générateur aléatoire utilisé pour $PRGA$. Appelons cet algorithme $MGA2$

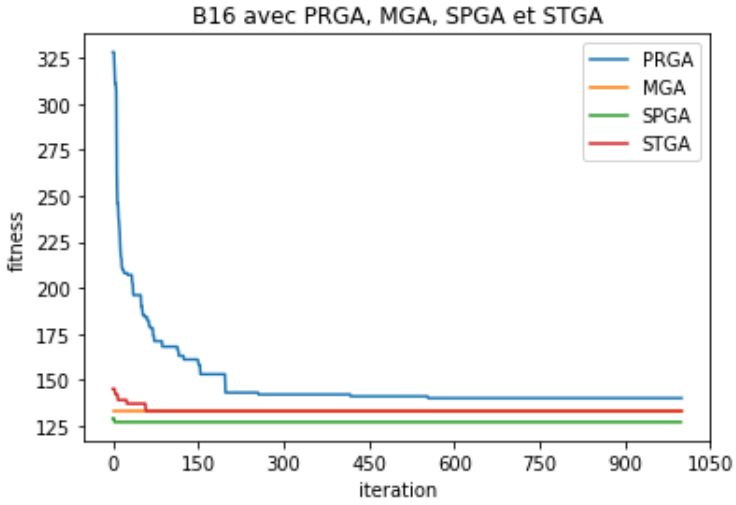
Notons cependant que l'algorithme $SPGA$ n'est plus utilisable dès le set C étant



donné le coût de l'heuristique.



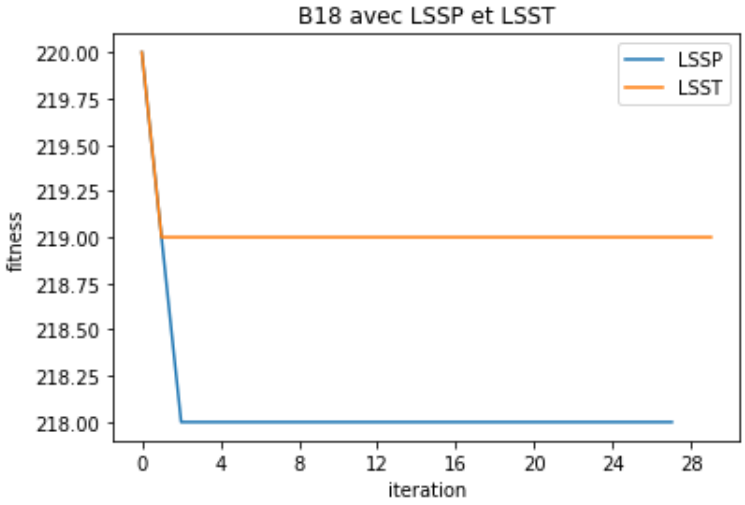
Comme nous pouvons le voir, les algorithmes avec heuristiques sont tellement efficaces que la comparaison avec $PRGA$ est presque inutile. Le résultat est d'autant plus visible lorsque l'on compare les différents algorithmes pour l'instance B16. L'algorithme $PRGA$ met un temps très grand à converger là où les autres algorithmes n'ont que peu de soucis à trouver une solution optimale dans presque tous les cas.



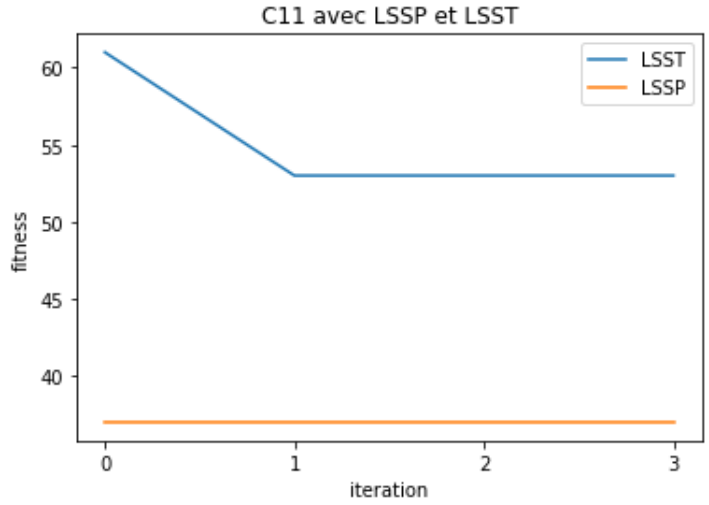
Nous pouvons clairement observer que l'algorithme le plus puissant est l'algorithme $SPGA$, l'heuristique étant la plus puissante il est logique que cette propriété se transcrive sur l'algorithme. Il est aussi le seul à arriver constamment à l'optimum, ceci étant il est aussi relativement lent (voir la section 5.3, comparaison des méthodes).

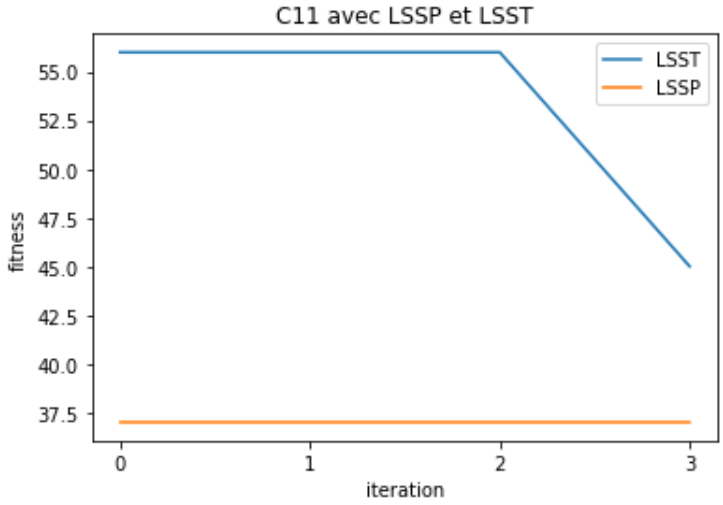
5.2 Recherche locale

Etudier et comparer les performances de la recherche locale est plus simple. Ceci dit, étant donné que nous avons attribué à la méthode un certain temps (ici d'une minute à cinq minutes pour les grandes instances), reporter le temps d'exécution est futile. Dans les tests nous ne prennons donc pas en compte cette donnée. En ce qui concerne la recherche locale, le seul paramètre à faire varier est le générateur de la solution initiale. Nous avons vite abandonné l'approche purement aléatoire et nous sommes concentrés sur l'utilisation des deux heuristiques de construction. Nous pouvons ainsi extraire deux algorithmes, un qui utilise l'heuristique du plus court chemin, que nous appellerons $LSSP$ et un qui utilise l'heuristique de l'arbre couvrant minimal, que nous appellerons $LSST$. Voici quelques résultats sur l'instance B018 :



On remarque que $LSSP$ atteint l'optimum mais que $LSST$ ne l'atteint pas. Même après beaucoup de redémarrages nous n'avons jamais atteint l'optimum avec $LSST$ alors que $LSSP$ l'atteint très souvent. Toutefois, $LSST$ a son utilité étant donné que certaines instances contenant un très grand nombre de noeuds terminaux ne peuvent pas se finir en un temps raisonable avec $LSSP$ mais $LSST$ donne une bonne approximation. Sur la machine de test, l'instance C05 n'est pas soluble en moins de 2 minutes avec $LSSP$, alors que $LSST$ peut donner une solution approximative de "1588" (le meilleur des cas trouvé) qui n'est pas si éloignée de l'optimal "1579". La méthode $LSST$ ne se révèle pas toujours aussi efficace malheureusement, comme on peut le voir sur l'instance C11. L'algorithme $LSSP$ prend du temps mais donne une réponse relativement proche, 37, proche de l'optimal 32, alors que $LSST$ stagne dans un voisinage éloigné de l'optimal. Ci-dessous deux exemples d'exécution de $LSST$ sur l'instance C11 :





Comme nous pouvons le constater, sur cette instance $LSST$ est très peu performante et donne dans certains cas des résultats éloignés de presque 50% de la solution optimale !

5.3 Comparaison des méthodes

Nous comparons maintenant les différentes méthodes utilisées. Chaque algorithme a été exécuté au moins trois fois sur l'instance donnée et les résultats sont donnés selon la moyenne des données observées. Notons que les temps obtenus pour les méthodes d'algorithme génétique n'ont au final que peu d'intérêt puisque que l'algorithme peut se retrouver dans un minimum local et continuer à tourner pendant un certain temps. De plus, la population générale ayant un aspect aléatoire, le temps d'exécution le sera aussi dans une certaine mesure, ceci est d'autant plus vrai pour la méthode *PRGA*. Nous ne prennons donc en compte que le temps d'exécution lorsque l'optimum est atteint. Les temps sont donnés en ms. Remarque : pour l'instance B06, l'heuristique du plus court chemin ne permet pas d'atteindre l'optimum la plupart du temps.

TABLE 5.1 – temps d'exécution des algorithmes, instances B de 1 à 6 sur Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz

Instances	B01	B02	B03	B04	B05	B06
PRGA	12143.694	7428.6	1300.7	2435.3	16984.47	2027.116
MGA1	280.01	471.02	654.03	438.02	547.03	16835.96
SPGA	472.13	521.12	856.3	533.4	577.14	24333.45
STGA	434.2	352.43	265.54	324.04	724.04	534

TABLE 5.2 – temps d'exécution des algorithmes, instances B de 7 à 12 sur Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz

Instances	B07	B08	B09	B10	B11	B12
PRGA	25428.33	23663.35	5363.55	123463.55	28171.6	123532.4
MGA1	881.05	1329.07	2435.13	471.02	654.03	438.02
SPGA	14798.8	22154.26	4325.3	521.12	856.3	533.4
STGA	899.05	700.04	667.03	352.43	265.54	324.04

TABLE 5.3 – temps d'exécution des algorithmes, instances B de 13 à 18 sur Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz

Instances	B13	B14	B15	B16	B17	B18
PRGA	21554.53	40511.318	45744.617	>5 min	>5 min	>5 min
MGA1	547.03	16835.96	2553.5	2554.56	5915.3	>5 min
SPGA	577.14	24333.45	45535.5	63393.62	475667.43	>5 min
STGA	724.04	534	24355.6	31973.82	458853.4	>5 min

TABLE 5.4 – distances à l'OPT connu, instances B de 1 à 9 sur Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz

Instances	B01	B02	B03	B04	B05	B06	B07	B08	B09
PRGA	0%	11.6%	0%	12%	0.5%	1.09%	0.3%	0%	1.4%
MGA1	0%	0%	0%	0%	0%	0.3%	0%	0%	0%
SPGA	0%	0%	0%	0%	0%	0.3%	0%	0%	0%
STGA	0%	0%	0%	0%	0%	0%	0%	0%	0%
LSSP	0%	0%	0%	0%	0%	0%	0%	0%	1.2%
LSST	0%	0%	0%	3.4%	1.2%	0%	3.4%	0%	1.34%

TABLE 5.5 – distances à l'OPT connu, instances B de 10 à 18 sur Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz

Instances	B10	B11	B12	B13	B14	B15	B16	B17	B18
PRGA	0.001%	4%	0%	6%	4%	0.6%	6%	0.9%	23.4%
MGA1	0%	0.17%	2.1%	3.1%	5%	2.53	0%	0%	0.14%
SPGA	0%	0%	0%	0%	0%	0%	0%	0%	0.14%
STGA	0%	0%	0%	0%	0%	0%	0%	1%	0.14%
LLSP	0%	0%	3%	0.4%	0.56%	0%	0.4%	0%	0%
LLST	1.3%	4%	3.1%	5.4%	10.4%	1.23%	0.4%	0.32%	0.12%

TABLE 5.6 – distances à l'OPT connu, instances B de 1 à 6 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	b01	b02	b03	b04	b05	b06
PRGA	0.01%	0.04%	0.00%	0.22%	0.05%	0.02%
SPGA	0.00%	0.00%	0.00%	0.00%	0.00%	0.02%
STGA	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
MGA1	0.00%	0.00%	0.00%	0.02%	0.02%	0.02%
MGA2	0.00%	0.00%	0.00%	0.02%	0.02%	0.00%
LSSP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LSST	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

TABLE 5.7 – distances à l'OPT connu, instances B de 7 à 12 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	b07	b08	b09	b10	b11	b12
PRGA	0.07%	0.00%	0.00%	0.05%	0.00%	0.00%
SPGA	0.00%	0.00%	0.00%	0.00%	0.02%	0.00%
STGA	0.00%	0.00%	0.00%	0.00%	0.03%	0.00%
MGA1	0.00%	0.00%	0.00%	0.00%	0.05%	0.00%
MGA2	0.03%	0.03%	0.00%	0.00%	0.02%	0.00%
LSSP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LSST	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

TABLE 5.8 – distances à l'OPT connu, instances B de 13 à 18 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	b13	b14	b15	b16	b17	b18
PRGA	0.25%	4.19%	0.03%	0.02%	0.02%	0.00%
SPGA	0.02%	0.00%	0.00%	0.00%	0.00%	>5min
STGA	0.02%	0.02%	0.01%	0.07%	0.00%	0.00%
MGA1	0.02%	0.00%	0.00%	0.04%	0.01%	0.00%
MGA2	0.07%	0.06%	0.02%	0.12%	0.01%	0.01%
LSSP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LSST	0.02%	0.02%	0.01%	0.05%	0.00%	0.00%

TABLE 5.9 – distances à l'OPT connu, instances C de 1 à 6 sur Intel(R) Core(TM) i5-480M CPU @ 2.66GHz

Instances	c01	c02	c03	c04	c05	c06
PRGA	359.80%	163.72%	33.65%	23.54%	5.13%	195.44%
SPGA	>5min	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	>5min	>5min
MGA	>5min	>5min	>5min	>5min	>5min	>5min
LSSP	0.00%	>5min	>5min	>5min	>5min	>5min
LSST	0.14%	0.14%	0.05%	0.03%	0.01%	0.09%

TABLE 5.10 – distances à l'OPT connu, instances C de 1 à 5 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	c01	c02	c03	c04	c05
PRGA	230.51%	136.47%	18.97%	14.13%	4.48%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	0.15%	0.24%	>5min	0.05%	0.01%
MGA1	0.04%	0.03%	>5min	>5min	>5min
MGA2	0.31%	0.42%	0.09%	0.08%	0.02%
LSSP	0.00%	0.00%	>5min	>5min	>5min
LSST	0.14%	0.17%	0.04%	0.03%	0.01%

TABLE 5.11 – distances à l'OPT connu, instances C de 6 à 10 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	c06	c07	c08	c09	c10
PRGA	105.33%	55.80%	8.92%	6.15%	0.14%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	0.01%
MGA1	0.27%	0.01%	>5min	>5min	>5min
MGA2	0.51%	1.10%	0.23%	0.13%	0.06%
LSSP	0.00%	0.01%	>5min	>5min	>5min
LSST	0.09%	0.40%	0.09%	0.06%	0.01%

TABLE 5.12 – distances à l'OPT connu, instances C de 11 à 15 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	c11	c12	c13	c14	c15
PRGA	12.66%	8.46%	0.98%	0.57%	0.09%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	0.03%
MGA1	>5min	>5min	>5min	>5min	>5min
MGA2	1.22%	0.41%	0.22%	0.16%	0.07%
LSSP	0.06%	>5min	>5min	>5min	>5min
LSST	0.28%	0.33%	0.10%	0.06%	0.02%

TABLE 5.13 – distances à l'OPT connu, instances C de 16 à 20 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	c16	c17	c18	c19	c20
PRGA	12.91%	8.06%	0.74%	0.45%	0.11%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	>5min
MGA1	>5min	>5min	>5min	>5min	>5min
MGA2	2.00%	2.56%	0.74%	0.40%	0.10%
LSSP	>5min	>5min	>5min	>5min	>5min
LSST	0.56%	0.61%	0.11%	0.05%	0.00%

TABLE 5.14 – distances à l'OPT connu, instances D de 1 à 5 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	d01	d02	d03	d04	d05
PRGA	964.56%	477.72%	53.96%	38.93%	16.03%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	>5min
MGA1	>5min	>5min	>5min	>5min	>5min
MGA2	0.60%	0.66%	0.16%	0.11%	0.02%
LSSP	0.02%	>5min	>5min	>5min	>5min
LSST	0.27%	0.40%	0.07%	0.04%	0.01%

TABLE 5.15 – distances à l'OPT connu, instances D de 6 à 10 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	d06	d07	d08	d09	d10
PRGA	547.27%	385.73%	22.44%	17.20%	0.79%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	>5min
MGA1	>5min	>5min	>5min	>5min	>5min
MGA2	0.78%	1.60%	0.25%	0.15%	0.06%
LSSP	0.06%	>5min	>5min	>5min	>5min
LSST	0.42%	0.97%	0.09%	0.05%	0.01%

TABLE 5.16 – distances à l'OPT connu, instances D de 11 à 15 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	d11	d12	d13	d14	d15
PRGA	35.90%	25.48%	1.43%	0.81%	0.21%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	>5min
MGA1	>5min	>5min	>5min	>5min	>5min
MGA2	2.45%	0.71%	0.28%	0.23%	0.09%
LSSP	>5min	>5min	>5min	>5min	>5min
LSST	0.86%	0.57%	0.11%	0.07%	0.02%

TABLE 5.17 – distances à l'OPT connu, instances D de 16 à 20 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	d16	d17	d18	d19	d20
PRGA	>5min	14.65%	0.90%	0.56%	0.20%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	>5min
MGA1	>5min	>5min	>5min	>5min	>5min
MGA2	>5min	>5min	0.78%	0.59%	0.20%
LSSP	>5min	>5min	>5min	>5min	>5min
LSST	>5min	>5min	0.15%	>5min	>5min

TABLE 5.18 – distances à l'OPT connu, instances E de 1 à 5 sur Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz

Instances	e01	e02	e03	e04	e05
PRGA	3178.86%	1658.42%	78.94%	60.07%	23.64%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	>5min
MGA1	>5min	>5min	>5min	>5min	>5min
MGA2	0.90%	>5min	>5min	>5min	0.03%
LSSP	>5min	>5min	>5min	>5min	>5min
LSST	0.25%	>5min	>5min	>5min	0.03%

TABLE 5.19 – distances à l'OPT connu, instances E de 1 à 5 sur Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz

Instances	e06	e07	e08	e09	e10
PRGA	2394.74%	1254.77%	49.75%	26.18%	9.03%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	>5min
MGA1	>5min	>5min	>5min	>5min	>5min
MGA2	1.78%	1.24%	0.22%	0.19%	0.08%
LSSP	>5min	>5min	>5min	>5min	>5min
LSST	1.21%	0.88%	0.11%	0.06%	0.02%

TABLE 5.20 – distances à l'OPT connu, instances E de 1 à 5 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	e01	e02	e03	e04	e05
PRGA	3051.01%	1541.00%	83.61%	58.69%	23.75%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	>5min
MGA1	>5min	>5min	>5min	>5min	>5min
MGA2	0.48%	1.36%	0.14%	0.11%	0.05%
LSSP	>5min	>5min	>5min	>5min	>5min
LSST	0.41%	1.04%	0.06%	0.03%	0.01%

TABLE 5.21 – distances à l'OPT connu, instances E de 6 à 10 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	e06	e07	e08	e09	e10
PRGA	2339.75%	1177.05%	47.53%	24.00%	10.25%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	>5min
MGA1	>5min	>5min	>5min	>5min	>5min
MGA2	1.70%	0.91%	0.25%	0.18%	0.08%
LSSP	>5min	>5min	>5min	>5min	>5min
LSST	1.60%	0.83%	0.11%	0.06%	0.02%

TABLE 5.22 – distances à l'OPT connu, instances E de 11 à 15 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	e11	e12	e13	e14	e15
PRGA	183.21%	164.18%	1.61%	0.99%	0.31%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	>5min
MGA1	>5min	>5min	>5min	>5min	>5min
MGA2	1.09%	0.75%	0.31%	0.23%	0.09%
LSSP	>5min	>5min	>5min	>5min	>5min
LSST	0.91%	0.63%	0.14%	0.09%	0.02%

TABLE 5.23 – distances à l'OPT connu, instances E de 16 à 20 sur Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

Instances	e16	e17	e18	e19	e20
PRGA	65.80%	39.04%	1.16%	0.73%	0.25%
SPGA	>5min	>5min	>5min	>5min	>5min
STGA	>5min	>5min	>5min	>5min	>5min
MGA1	>5min	>5min	>5min	>5min	>5min
MGA2	5.73%	3.16%	1.02%	0.70%	>5min
LSSP	>5min	>5min	>5min	>5min	>5min
LSST	0.63%	1.34%	0.93%	1.23%	0.01%

Chapitre 6

Conclusion

Après avoir étudié nos différentes méthodes de manière expérimentale, plusieurs lignes directrices en sortent. Nous pouvons remarquer d'entrée de jeu que la recherche locale est la méthode la moins puissante de toutes. Elle est bien plus dépendante de la qualité de la solution initialement générée que les méthodes par algorithme génétique et n'explore qu'un voisinage très restreint. Ceci étant, elle est beaucoup plus rapide que les algorithmes génétiques et peut éventuellement permettre d'améliorer une heuristique déja puissante pour se rapprocher d'autant plus de la solution optimale. Les méthodes par algorithmes génétiques sont quant à elles plus puissantes et peuvent se rapprocher assez aisément de la solution optimale pour des instances modérément grandes. L'algorithme $SPGA$ est objectivement le meilleur, mais il est beaucoup trop lent dû à son heuristique couteuse sur des instances de grandes tailles, $STGA$ est plus rapide mais globalement moins bon, et $MGA1$ permet dans le cas général d'obtenir un bon compromis entre les deux algorithmes. Plus étonnant sont les performances excellentes de $LSST$ pour les instances de grandes tailles, ce dernier allant jusquà résoudre les instances E(jusqu'à 2500 sommets et 62500 arêtes) dans un temps raisonnable (limite de 5 minutes) avec une erreur le plus souvent en dessous des 1% ! La convergence peut cependant s'avérer lente. Plus étonnant encore sont les performances de $MGA2$ qui sont globalement assez mauvaises, bien plus mauvaises qu'espéré, celui-ci étant presque toujours moins bon que $LSST$ dans la limite des 5 minutes. Une possible amélioration serait d'utiliser le fait que la recherche locale est en générale rapide et peu couteuse afin d'améliorer efficacement l'heuristique de construction et l'utiliser dans la population initiale d'une méthode par algorithme génétique. Il serait peut-être intéressant aussi de détecter une stagnation dans la population d'un algorithme génétique indiquant très certainement un manque de diversité, afin d'éliminer quelques individus au hasard et en regénérer afin de remplacer les individus éliminés et ainsi diversifier le patrimoine génétique de la population actuelle.