ITD – TP-TSP 2018

Unterreiner – Lebtahi – Lelouch

Rapport

Git : https://github.com/lebta/Projet-ITD---Unterreiner-Lelouch-Lebtahi

Objectif : résoudre un problème de TSP sur un ensemble d'instances connues en un minimum de temps, selon un ou plusieurs algorithmes choisis par nos soins

Algorithmes sélectionnés :

1. Algorithme du plus proche voisin
2. Résolution grâce un arbre couvrant de poids c minimal (Algorithme TwiceAroundTheTree)
3. Algoritme d'optimisation d'une solution existante : two opt

I. Présentation des algorithmes implémentées

1. Le plus proche voisin

i. Principe de l'algorithme

On choisit un point de départ arbitrairement, en l'occurrence 0.
A chaque itération, on sélectionne comme point i+1 le point le plus proche de i .

ii. Mise en place

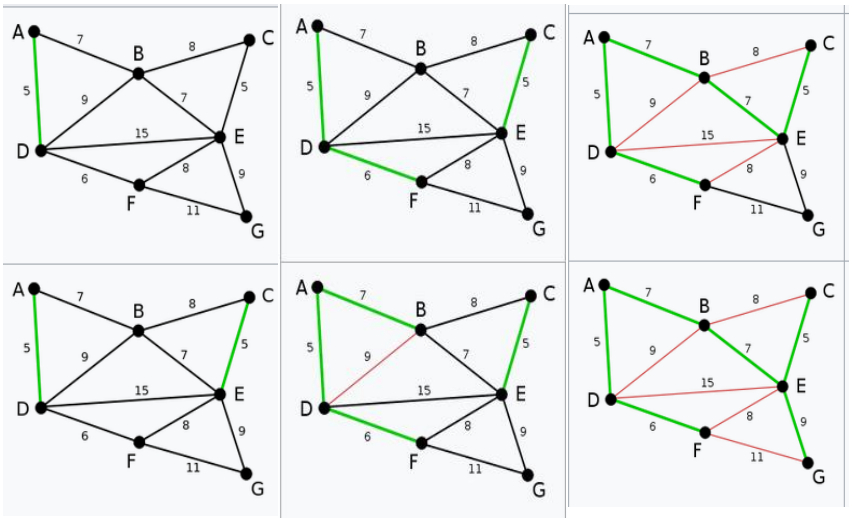
Nous avons choisi de créer une liste d'entier pour stocker les villes déjà visitées.
A chaque itération, on sélectionne le dernier point de cette liste et on cherche le point le plus proche grâce à la fonction getDistances de la classe instance.

2. Resolution à l'aide d'un arbre couvrant de poids minimal

i. Construction de l'arbre (algorihme de Kruskal)

On commence par trier tous les arcs du graphes (les couples de points) par ordre de poids croissant. Pour cela on crée une classe arc qui a pour variables d'instance 2points (2 int), la distance entre les deux points(un long) et une instance du problème.
On implements comparable et on code compareTo afin de pouvoir utiliser la fonction sort sur une liste d'arcs.

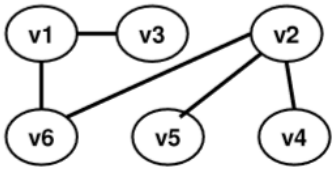
On utilise ensuite l'algorithme de Kruskal à partir de cette liste triée.
On prend les arcs dans l'ordre, on ajoute un arc à l'arbre si l'ajout de celui-ci ne conduit pas à la formation d'un cycle fermé.
En pratique on tient une liste des sous-arbres de l'arbre en construction. On rajoute un arc seulement si les 2 points le constituant n'appartiennent pas déjà à 2 sous-arbres différents (1point seul ne forme pas un sous-arbre).



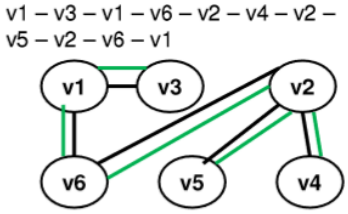
Exemple de construction d'un arbre par l'algorithme de Kruskal

ii. Réolution à partir de l'arbre (aglorithme TwiceAroundTheTree)

On part d'un arbre de points.
Nous allons faire le « tour » de cet arbre, c'est-à-dire passer exactement 2 fois par chaque branches (cf schéma).



Arbre



Tour issu de l'arbre

Pour ce faire, on commence par définir le degré d'un point : c'est le nombre de points auquel il est relié par une branche de l'arbre. Parcourir chaque branche de fois revient à passer par chaque point degré fois, c'est ce point de vue que nous allons adopter.
Nous allons cependant voir le degré comme une variable dynamique : on initialise le degré selon la définition ci-dessus et nous le décrémentons à chaque fois que nous passsons par un point.

On part d'un point arbitraire, en l'occurrence 0.
A l'itération i, on choisit i+1 selon les conditions suivantes :

- i+1 doit être relié à i par une branche de l'arbre
- le degré de i+1 doit être supérieur à 0.
- On ne doit pas être déjà passé par la branche i – i+1 sauf si tout les autres voisins de i sont de degré 0.

On obtient ainsi un tour de l'arbre, c'est-à-dire une liste de point avec des doublons.
Pour obtenir notre solution il suffit de supprimer les doublons en ne gardant que la première occurrence de chaque point.

3. Amélioration d'une solution déjà existante : two opt

On parcourt cherche à améliorer la distance parcourue en échangeant 2 points de 2 arcs diff érents.
On parcourt tout les arcs 2 fois en cherchant des possibilités d'échanges tant que le schéma est améliorable ou que le temps est inférieur à la limite impartie.
Un schéma n'est plus am »liorable lorsqu'aucun échange d'arcs ne peut l'améliorer.
Un schéma est amélioré par 1 échange (k,l) d'un (i,j) et d'un arc(k,l) ssi dist(i,j)+dist(k,l) > dist(i,l)+dist(j,k).

II. Tentative non implémentée : l'algorithme des fourmis

Un nombre choisi de fourmis sont réparties aléatoirement dans l'ensemble des villes de l'instance. L'idée est de leur faire faire parcourir l'ensemble des villes (sans doublon), selon une évolution induite par le nombre de phéromones présents sur les chemins (entre deux villes). Un passage par un chemin représente un ajout d'une phéromone sur le chemin. Ensuite, on récupère le parcours le plus court parmis toutes les fourmis, et on réitère l'opération un nombre de fois voulu, sans remettre à 0 le nombre de phéromones par chemin. A l'issue de ces itérations, on a un parcours se rapprochant du plus cours chemin.

En pratique, une fourmi évolue selon le chemin présentant le plus grand rapport nombre de phéromones/ longueur du chemin. Après un parcours (les fourmis ont parcouru toutes les villes une fois), on met à jour le nombre de phéromones sur les chemins. Ensuite on récupère le meilleur parcourt, on met à jour l'historique des villes parcourues par fourmi, et on réitère l'opération. Il faut par ailleurs, faire évaporer les phéromones entre chaque itération.

Nous n'avons pas eu le temps d'implémenter entièrement cette solution.

IV. Bibliographie

- Arbre de recouvrement (Kruskal) :

http://www.unit.eu/cours/EnsROtice/module_avance_thg_voo6/co/algoKruskal.html

http://www-public.imtbs-tsp.eu/~raffy/Techniques%20Quantitatives/Cours%20TQ/TheorieGraphes/kruskal.htm

- Twice around the tree:

http://www.csl.mtu.edu/cs4321/www/Lectures/Lecture%2028%20-%20Approximation%20Algorithm.htm

http://www.jsums.edu/nmeghanathan/files/2017/01/CSC323-Sp2017-Module-6-P-NP-NP-CompleteProblems-ApproxAlgorithms.pdf?x61976

- 2-opt:

https://ocw.mit.edu/courses/sloan-school-of-management/15-053-optimization-methods-in-management-science-spring-2013/lecture-notes/MIT15_053S13_lec17.pdf

https://wikimonde.com/article/2-opt

https://www2.seas.gwu.edu/~simhaweb/champalg/tsp/tsp.html

- Fourmis :

http://www.i3s.unice.fr/~crescenz/publications/travaux_etude/colonies_fourmis-200605-rapport.pdf
https://fr.wikipedia.org/wiki/Algorithme_de_colonies_de_fourmis

http://www.apprendre-en-ligne.net/info/algo/fourmis.html

Annexe : résultat des expérimentations

- Comparaison Kruskal et Kruskal + 2-opt :

Relevé temps et valeur algorithme Kruskal + twice around the tree, en partant de 0 :

Instance	Valeur	Temps (ms)
d198	18754	172
d657	66379	860
eil10	193	0
eil51	592	31
eil101	895	47
kroA100	27327	63
kroA150	35236	78
kroA200	40357	140
lin318	59729	297
pcb442	64529	484
rat575	9356	672

Relevé temps et valeur algorithme avec amélioration 2-opt :

Instance	Valeur	Temps (ms)
d198	16708	219
d657	54486	1125
eil10	176	0
eil51	454	31
eil101	717	78
kroA100	23180	125
kroA150	28140	156
kroA200	32420	188
lin318	46968	406
pcb442	56020	719
rat575	7415	969

L'algorithme avec amélioration est donc plus long (de 20% en moyenne), mais permet de bien meilleures solutions (-10% des valeurs en moyenne).

- Comparaison PlusProcheVoisin + 2-opt et Kruskal + 2-opt :

Relevé PlusProcheVoisin + 2-opt :

Instance	Valeur	Temps (ms)
d198	17186	62
d657	55269	141
eil10	177	0
eil51	462	0
eil101	741	32
kroA100	24200	16
kroA150	28511	31
kroA200	31549	47
lin318	46625	78
pcb442	54154	125
rat575	7638	156

Le plus proche voisin amélioré est évidemment beaucoup plus rapide que le Kruskal amélioré (-70% en termes de temps moyen), mais apporte des valeurs de distances un peu plus grandes (2% en moyenne)