

Metaheuristics for Optimization
Series 4: Ant System and Traveling Salesman Problem

Romain Mencattini

7 novembre 2016

Table des matières

1	Le voyageur de commerce	3
2	Algorithme	3
2.1	Greedy	3
2.1.1	Algorithme	3
2.1.2	Mesures	3
2.2	Ant System Séquentiel	4
2.2.1	Algorithme	4
2.2.2	Mesures	5
2.3	Ant System Parallèle	5
2.3.1	Algorithme	5
2.3.2	Mesures	5

1 Le voyageur de commerce

Il s'agit du problème sur lequel nous allons tester l'algorithme du recuit simulé. Détaillons le de manière mathématique.

Nous avons un ensemble de villes : V . Où $v = \{x, y\} \in V$. est une ville qui avec une coordonnée x et une coordonnée y qui appartiennent à \mathbb{N} .

On veut visiter toutes les villes, une et une seule fois. Notre résultat sera une permutation de ces villes : $[v_1, v_2, \dots, v_N]$. Une liste avec N villes. Où v_1 sera la première ville visitée, v_2 la deuxième etc.

On utilise la distance euclidienne :

$$euclidienne(v_1, v_2) = \sqrt{(v_{1_x} - v_{2_x})^2 + (v_{1_y} - v_{2_y})^2}$$

On veut minimiser la distance totale qui est :

$$d_{total} = \sum_{i=2}^N euclidienne(v_{i-1}, v_i)$$

2 Algorithme

Pour ce faire nous allons comparer plusieurs méthodes :

- Greedy
- Ant System séquentiel
- Ant System parallèle

2.1 Greedy

2.1.1 Algorithme

L'algorithme est relativement simple. C'est le même que celui programmé dans le tp3. Il faut :

1. Choisir une ville aléatoire, v_0 .
2. Ajouter la ville la plus proche à notre solution.
3. Répéter l'étape 2. jusqu'à ce qu'on ait vu toutes les villes.

Suivant le choix du développeur, on peut choisir d'ajouter la première ville, à la fin pour avoir une boucle. Cet algorithme est totalement déterministe pour un V fixé et une ville initiale v_0 fixée.

2.1.2 Mesures

Voici les résultats obtenus en moyenne lors de vingt lancers :

File	Mean	Std
./data/cities.dat	31.537453312384304	1.8045784058777552
./data/cities2.dat	3.387658063949946	0.2163689591198703
./data/cities50.dat	6.278780156847236	2.332163876429746e-15
./data/cities60.dat	6.302401870939252	0.047031657572111
./data/cities80.dat	6.2813443585626185	1.2394364910839084e-15
./data/cities100.dat	6.282070156684796	1.8110771949088597e-15

On remarque que l'algorithme trouve toujours la solution optimale pour le cercle. Ce qui est évident. En prenant le plus proche voisins, on trouve notre point à gauche (ou à droite), et ainsi de suite jusqu'à faire le tour du cercle. Il y a une déviation standard assez importante pour les deux premiers fichiers. Cela vient du fait que l'algorithme est déterministe, donc il dépend du point de départ. Ce qui n'est pas le cas des algorithmes fournis.

2.2 Ant System Séquentiel

Les deux versions de cet algorithme (séquentielle et parallèle) sont très proches, nous allons donc expliquer en premier lieu la version séquentielle plus parallèle.

2.2.1 Algorithme

Voici le déroulement de l'algorithme :

1. on choisit une ville v_0 de départ pour la fourmis.
2. tant qu'il existe des villes non visitées, on en choisit une v_i et on l'ajoute.
3. marquer le chemin.

On répète cela pour chacune des m fourmis. Cela correspond à une itération. Une fois l'itération terminée, on met à jour la matrice des phéromones et on reprend les itérations jusqu'à en avoir exécuté t_{max} .

Nous allons maintenant expliquer les points cités. Pour le choix de la fourmis, nous les avons répartis une par ville, (en utilisant le modulo). Il faut donc que n le nombre de villes, soit égal à m le nombre de fourmis. C'est dans ces conditions que les meilleurs résultats sont atteints.

Pour choisir parmi les villes, nous générons un tableau de probabilités (associée une probabilité à chaque ville), et nous choisissons la ville suivante en suivant ces probabilités. Voici comme elles sont générées :

$$p_{ij}^k(t) = \begin{cases} \frac{(\tau_{ij}(t))^\alpha (\eta_{ij})^\beta}{\sum_{l \in J} (\tau_{il}(t))^\alpha (\eta_{il})^\beta} & \text{si } j \in J \\ 0 & \text{sinon} \end{cases}$$

Avec k , la $k^{\text{ième}}$ fourmis, i la ville i , j la ville j , et α, β des paramètres, et t l'itération courante. On va donc favoriser avec τ les chemins avec beaucoup de phéromones ; et avec

η les villes les plus proches. Où τ est la matrice des phéromones, et η la matrice de l'inverse des distances entre villes. Donc $\eta_{ij} = \frac{1}{d_{ij}}$

Chaque fourmis va marquer le chemin de cette manière :

$$\Delta\tau_{ij}^k(t) = \begin{cases} \frac{Q}{L^k(t)} & \text{si la fourmis } k \text{ utilise l'arc}(i, j) \text{ dans son parcours} \\ 0 & \text{sinon} \end{cases}$$

Où Q est un paramètre. $L^k(t)$ est la longueur du chemin de la fourmis k à l'itération t . Mettre à jour la matrice des phéromones, soit τ , se fait comme suit :

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum_{k=1}^m \Delta\tau_{ij}^k(t)$$

2.2.2 Mesures

Pour la version séquentielle, voici les résultats, avec $t_{max} = 50$ et $\rho = 0.1$, toujours vingt itérations :

File	Mean	Std
cities.dat	27.86711241242947	0.7484163226353873
cities2.dat	3.0318633694064667	0.04990194967224049
cities50.dat	6.3283035636850675	0
cities60.dat	6.2759356854396	0
cities80.dat	6.348320460878663	0.007431478876196535
cities100.dat	6.282070156684795	9.362222582871203e-16

On remarque que les résultats sont meilleurs que le Greedy. Ce qui est une bonne chose. L'AS trouve le maximum global du cercle, et fait de meilleurs résultats pour *cities.dat* et *cities2.dat*.

2.3 Ant System Parallèle

2.3.1 Algorithme

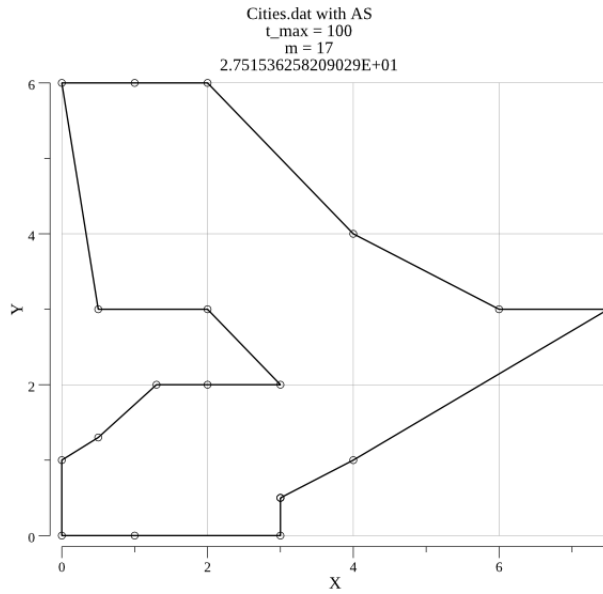
La version parallèle est la même que la version séquentielle, sauf qu'on parallélise les m fourmis lors des itérations. En utilisant les facilités du langage Golang. Il faut ajouter des verrous lors de l'écriture du $\Delta\tau$, et un mécanisme de synchronisation à la fin d'une itération. Une fois cela fait, on obtient les mêmes résultats que la version séquentielle. Cependant en plus rapide, il est donc possible d'augmenter t_{max} dans la version parallèle.

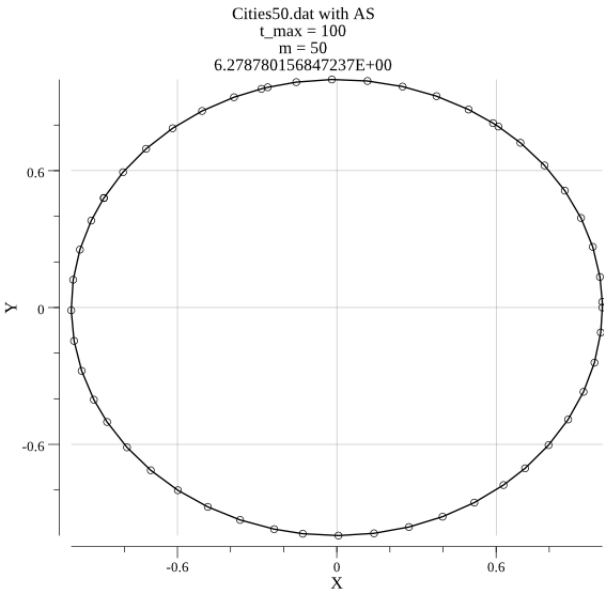
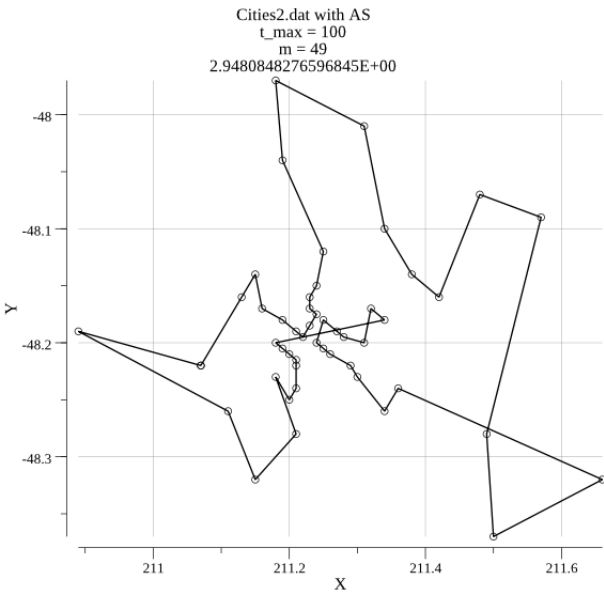
2.3.2 Mesures

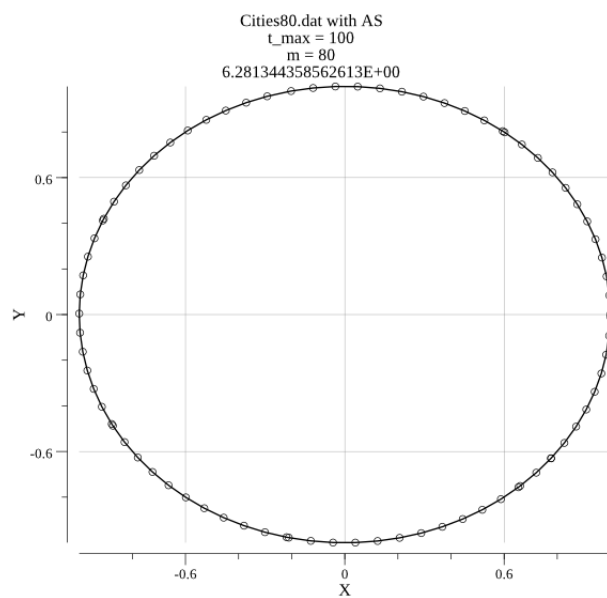
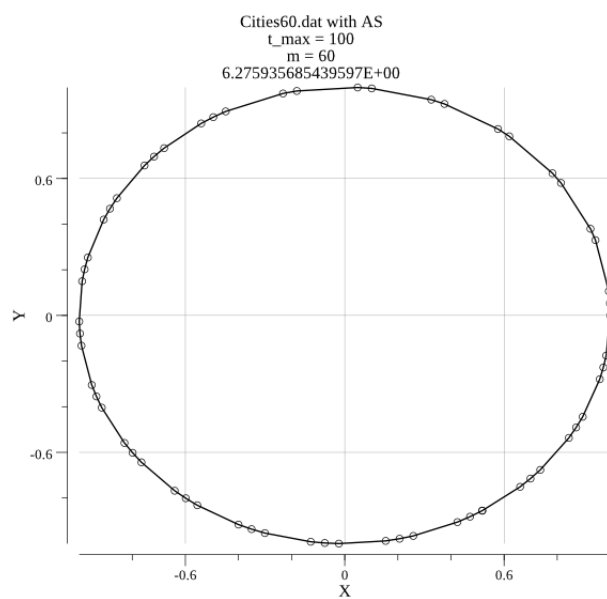
Voici les mesures pour $t_{max} = 100$ et $\rho = 0.1$, toujours vingt itérations :

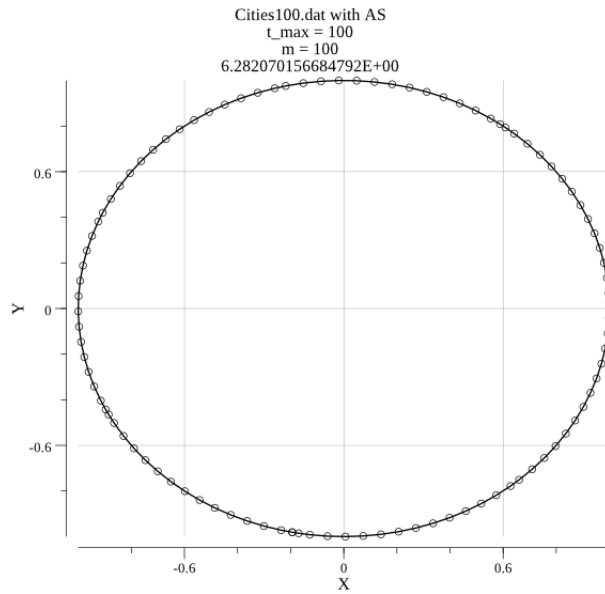
File	Mean	Std
cities.dat	27.51536258209029	0
cities2.dat	2.9352583475370393	0.03282959295749493
cities50.dat	6.278780156847236	9.362222582871203e-16
cities60.dat	6.275935685439598	9.362222582871203e-16
cities80.dat	6.281344358562613	0
cities100.dat	6.282070156684793	9.362222582871203e-16

On remarque que les résultats sont meilleurs que la version à 50 itérations. En augmentant la valeur de t_{max} , on augmente la qualité des résultats. En utilisant la version parallèle, nous avons dessiné des exemples de résultats pour avoir une idée :





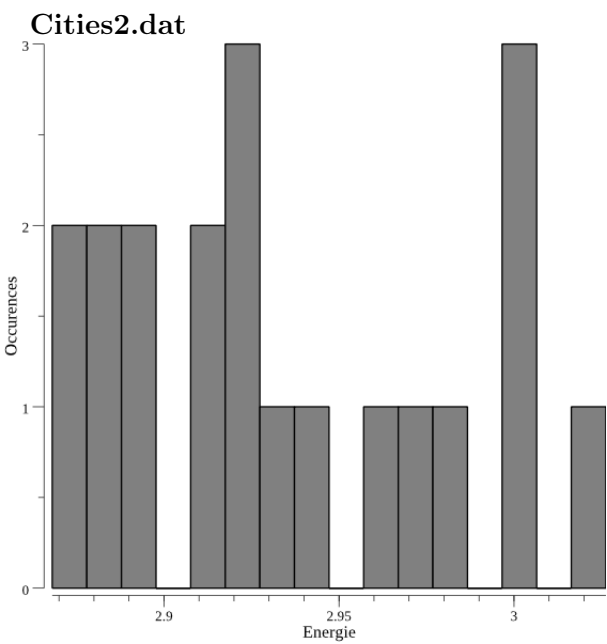
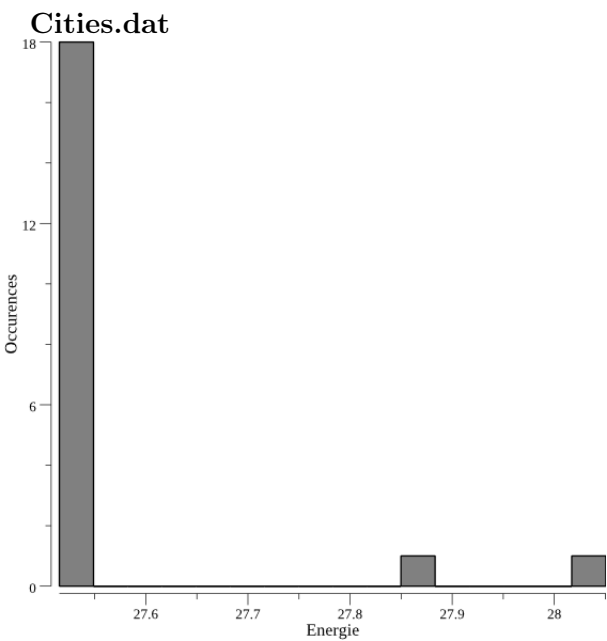


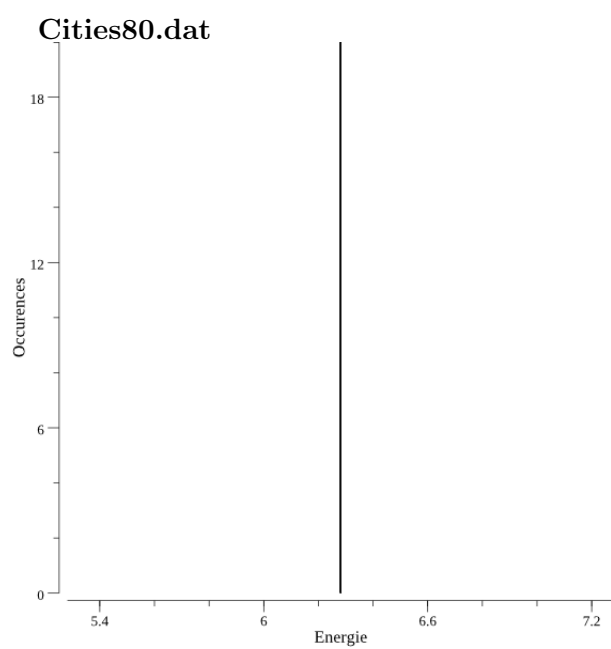
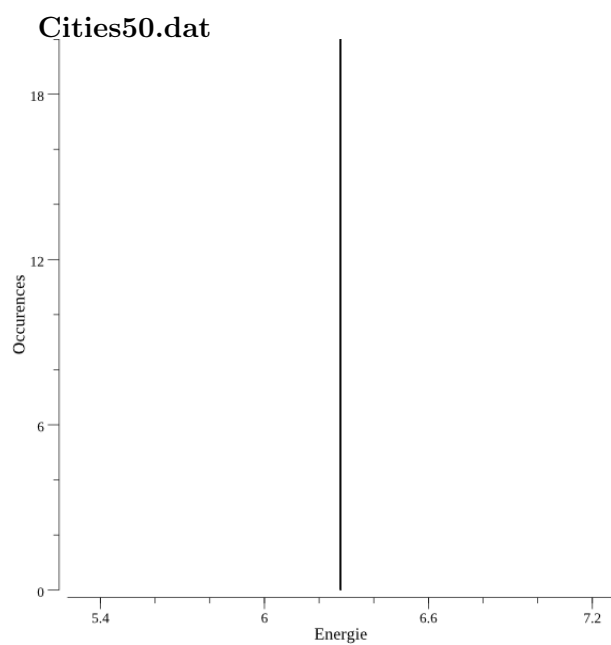


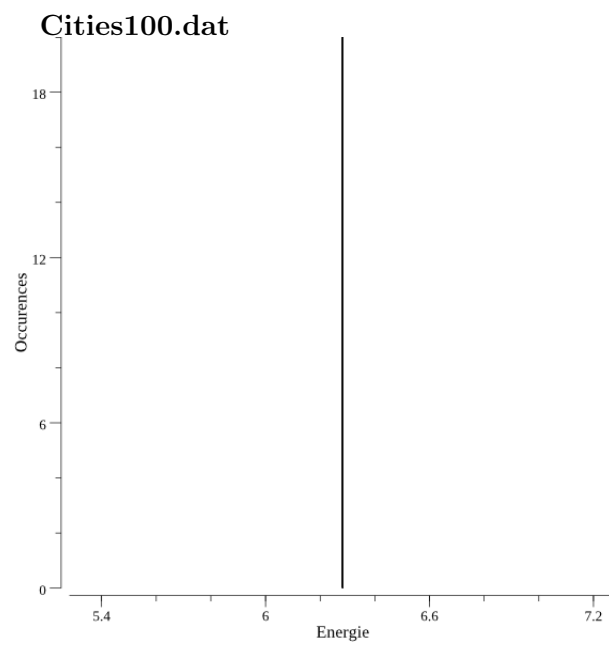
On confirme ce que l'on a compris grâce au tableau de résultats, ci-dessus :

- pour le cercle la solution optimale est trouvée
- l'algorithme se comporte assez bien même lors de paysage très rugueux (cities2.dat)
- il trouve de bonnes valeurs et ce souvent (cities.dat)

Pour appuyer ces résultats, nous allons présenter les histogrammes des valeurs obtenues pour vingt lancers :







Ce qui confirme les résultats.