Résoudre des problèmes complexes en s'inspirant de la nature : l'exemple de l'algorithme de recherche gravitationnel

Yann Bourreau

Septembre 2023

1 Introduction

La puissance de calcul des ordinateurs a considérablement augmenté au cours des dernières décennies, permettant d'utiliser des programmes informatiques pour résoudre des problèmes de plus en plus complexes, comme la reconnaissance des formes ou l'analyse de génome. Il est aussi devenu fréquent pour beaucoup d'entre nous d'utiliser un programme informatique pour résoudre des problèmes de la vie quotidienne, comme trouver le meilleur chemin d'un point A à un point B avec un service de cartographie sur son téléphone mobile.

Une limite importante pour automatiser une tâche en écrivant un programme informatique est le temps d'exécution du programme, c'est-à-dire le temps qui s'écoule entre son lancement et l'obtention d'un résultat. En effet, il faut que le temps d'exécution soit raisonnable par rapport à l'application. Les services de cartographie ne seraient pas si populaires s'il fallait de longues minutes pour calculer le meilleur chemin allant de A à B !

Pour mesurer le temps d'exécution d'un programme, on pourrait tout simplement le chronométrer. Mais cette mesure du temps d'exécution dépendrait de la puissance de l'ordinateur utilisé et serait donc peu pertinente pour quelqu'un utilisant un autre ordinateur, plus puissant ou moins puissant. Une autre approche est donc utilisée par les chercheurs en informatique, qui consiste à analyser la complexité des algorithmes.

Un algorithme est une suite d'instructions et d'opérations permettant de résoudre un problème. L'algorithme est ensuite traduit dans un langage de programmation pour pouvoir être exécuté sur un ordinateur. Pour mesurer le temps d'exécution d'un programme, on va alors évaluer la *complexité de l'algorithme*. Pour cela, on mesure le

nombre d'actions qu'il effectue, ce qui dépend de la quantité de données du problème que l'on cherche à résoudre.

Certains problèmes présentent des temps de résolution par des algorithmes très semblables. On va alors les regrouper dans ce qu'on appelle des *classes de complexité*.

Certains problèmes peuvent être résolus en temps linéaire par un algorithme, ce qui signifie que le temps d'exécution croît de manière linéaire avec la quantité de données du problème ; on parle de classe LINEAR. Par exemple, pour sommer les éléments d'un tableau, il faut examiner chacun de ses éléments et donc le temps d'exécution croît linéairement avec le nombre d'éléments du tableau.

La classe P contient les problèmes que l'on peut résoudre en temps polynomial, c'est-à-dire dont le temps d'exécution est à peu près proportionnel à n^k, où n représente le nombre de données du problème et k est une constante (on dit plus précisément que ces algorithmes ont un temps d'exécution en $O(n^k)$). Par exemple, vérifier s'il existe un chemin reliant toute paire de sommets d'un graphe (c'est-à-dire si le graphe est connexe) est un problème que l'on peut résoudre en temps polynomial. On considère que les problèmes de cette classe P peuvent être résolus efficacement.

La classe NP contient quant à elle les problèmes que l'on résout en temps non déterministe polynomial. Les algorithmes de cette classe sont dits *non déterministes* parce que, s'il y a un choix à faire durant l'exécution, on va effectuer tous les choix. Prenons l'exemple d'un algorithme conçu pour réussir à sortir d'un labyrinthe. A une intersection, s'il y a deux chemins possibles, un algorithme déterministe va choisir l'un des deux chemins pour continuer. Si ce chemin n'amène pas à la solution, l'algorithme reviendra sur ses pas et essaiera l'autre solution. A cette même intersection, un algorithme non déterministe va lui explorer les deux solutions en même temps.

On peut remarquer que ces trois classes sont imbriquées, avec LINEAR $\subset$ P $\subset$ NP. En effet, tout algorithme en temps linéaire est un algorithme en temps polynomial et un algorithme déterministe est un cas particulier d'algorithme non déterministe. Une question non résolue de l'informatique est de savoir si NP $\subset$ P (ce qui impliquerait que NP = P). Il est admis que ce n'est pas le cas.

Cela fait que les problèmes de la classe NP sont "difficiles" à résoudre. En effet, il n'existe pas d'algorithme qui résout un problème NP en temps polynomial dans tous les cas.

Cependant, pour certains problèmes NP, il existe des algorithmes permettant de les résoudre quasiment en temps polynomial : dans un grand nombre de cas, ils s'exécutent en temps polynomial et dans un petit nombre de cas, en temps exponentiel (donc très lentement). C'est le cas du problème SAT ou problème de satisfaisabilité booléenne, un problème qui appartient à la classe NP et qui a donné lieu à beaucoup

de recherches en informatique. Des programmes s'appelant des solveurs SAT ont été conçus spécifiquement pour ce problème, qu'ils résolvent très rapidement.

Pour d'autres problèmes NP, il n'existe pas d'algorithme permettant de les résoudre de manière exacte quasiment en temps polynomial. Dans ce cas, une autre approche consiste à ne pas chercher de solution exacte au problème mais plutôt une solution approchée. On parle d'*algorithmes d'approximation*. Ces algorithmes permettent de résoudre des problèmes NP de manière approchée mais en temps polynomial dans tous les cas.

1.1 Algorithmes d'approximation

Plus précisément, un algorithme d'approximation est une méthode permettant de trouver une solution approchée à un problème d'optimisation. L'algorithme se caractérise par un facteur d'approximation $\rho > 1$.Si on note z_* l'optimum global du problème d'optimisation, un algorithme d'approximation de facteur $/\rho$ renvoie un résultat dont la valeur est comprise entre :

- z_*/ρ et z_* s'il s'agit d'un problème de maximisation
- z_* et $z_* * \rho$ s'il s'agit d'un problème de minimisation

Par exemple, l'algorithme de Christofides [1] est un algorithme d'approximation pour le célèbre problème du voyageur de commerce quand on se place dans un repère cartésien : quel est le plus court chemin passant une et une seule fois par n villes distantes ? L'algorithme de Christofides ne donne pas la solution optimale à ce problème mais une approximation, avec un parcours au plus une fois et demi plus long que le parcours optimal, donc avec un facteur d'approximation $\rho = 3/2$.

Cependant, tous les problèmes de la classe NP ne peuvent pas être résolus par un algorithme d'approximation. Reprenons l'exemple précédent du problème du voyageur de commerce mais cette fois-ci, considérons qu'on ne se place pas dans un repère cartésien. Dans ce cas, la distance pour aller de A à C peut être supérieure à la distance de A à B plus la distance de B à C. Si c'est le cas, alors il n'existe pas d'algorithme d'approximation permettant de résoudre le problème du voyageur de commerce. C'est-à-dire, pour tout ρ, il n'existe pas d'algorithme permettant d'obtenir un parcours au plus ρ fois plus long que le trajet optimal.

1.2 Algorithmes heuristiques et métaheuristiques

Pour ce type de problèmes complexes, une autre approche consiste à utiliser des algorithmes heuristiques. Un algorithme heuristique est une méthode de résolution pour un problème qui ne conduit pas toujours à la solution exacte mais fournit souvent une

solution approchée, sans que l'on soit capable de déterminer la qualité de cette solution comme pour un algorithme d'approximation.

Par exemple, les algorithmes gloutons (*greedy algorithms*) sont des algorithmes heuristiques. Un algorithme glouton recherche une solution à un problème étape par étape en réalisant le meilleur choix possible à chaque itération. Par exemple, dans le cas du problème du voyageur de commerce, à chaque étape du voyage, l'algorithme glouton sélectionne la ville la plus proche par laquelle le voyageur n'est pas encore passé. On comprend bien que cela ne conduit pas toujours à la solution optimale (le plus court chemin), juste à une solution approchée.

Les algorithmes heuristiques ne fournissent pas (toujours) la solution exacte à un problème mais ils présentent des avantages importants. Pour certains problèmes, il peut s'avérer très complexe, voire impossible, de concevoir un algorithme efficace. Dans ce cas, les algorithmes heuristiques ont l'avantage de fournir au moins une solution approchée. Ces algorithmes peuvent aussi être très performants en pratique. Pour toutes ces raisons, les algorithmes heuristiques ont de nombreuses applications industrielles, par exemple pour l'optimisation des réseaux d'énergie [2].

Récemment, les chercheurs en informatique se sont intéressés à un nouveau type d'algorithme heuristique : les algorithmes *métaheuristiques*. Certains problèmes sont très complexes à résoudre en raison du grand nombre et de la variété des paramètres, ainsi que la grande quantité d'entrées. Les données peuvent parfois aussi être incomplètes. Ces problèmes sont si complexes que les algorithmes heuristiques classiques échouent à trouver une solution. Une solution que les chercheurs ont trouvé consiste à ajouter des éléments provenant des algorithmes d'approximation aux algorithmes heuristiques.

On distingue généralement les méthodes de recherche locale et les méthodes avec une population de solutions. Les méthodes de recherche locale (*single-point search algorithms*), comme le recuit simulé, partent d'une solution donnée, qui peut être prise au hasard, puis cherchent à l'améliorer à chaque itération en lui substituant une solution relativement proche. Pour les méthodes avec une population de solutions (*population-based algorithms*), on démarre d'un ensemble de solutions (une population) que l'algorithme fait progressivement converger vers une meilleure solution.

Parmi les algorithmes métaheuristiques avec population de solutions, de nombreux algorithmes sont dits inspirés de la nature (*nature-inspired*). Par exemple, les algorithmes génétiques s'inspirent du processus de sélection naturelle. D'autres s'inspirent du comportement des fourmis ou des abeilles.

Enfin, certains algorithmes s'inspirent de phénomène physique, comme la gravitation. C'est le cas de l'algorithme de recherche gravitationnel (*Gravitational Search*

Algorithm, GSA), que nous allons présenter plus précisément dans la partie suivante.

2 Algorithme de recherche gravitationnel

Comme son nom l'indique, l'algorithme de recherche gravitationnel (*Gravitational Search Algorithm*, GSA) est inspirée de la gravitation. Il a été proposé en 2009 par les chercheurs Rashedi, Nezamabadi-pour et Saryazdi [3]. Il a rapidement attiré l'attention des autres chercheurs en informatique, avec plus de 7000 citations en septembre 2023 selon Google Scholar et de nombreuses versions améliorées de l'algorithme publiées dans les revues d'informatique. Des applications industrielles ont été aussi développées, par exemple, pour les systèmes d'énergie, les réseaux de communication électronique ou la détection d'attaques sur les réseaux informatiques [4].

2.1 Loi de gravitation

La gravitation est l'une des forces fondamentales à l'œuvre dans l'Univers. Elle fait que deux particules ayant une masse s'attirent. Par exemple, deux corps de masses M_1 et M_2 s'attirent mutuellement, avec une force proportionnelle (en valeur absolue) au produit de leurs masses et inversement proportionnelle au carré de la distance R qui les sépare. Plus exactement, cette force vaut (en valeur absolue) :

$$F = G\frac{M_1 M_2}{R^2},$$

où G représente la constante gravitationnelle.

Cette force nous permet d'obtenir la formule de l'accélération d'un élément de masse M :

$$a = \frac{F}{M}.$$

Cette accélération est la cause du déplacement des masses dans l'espace.

L'idée de l'algorithme de recherche gravitationnel est d'appliquer la force gravitationnelle à des solutions possibles pour un problème d'optimisation, en donnant une masse plus élevée aux solutions qui s'approchent le plus de l'optimum.

2.2 L'algorithme de recherche gravitationnel

Plus précisément, on cherche à optimiser une fonction f, dite *fonction objectif*, définie dans un espace à n dimensions. Le GSA recherche l'optimum de cette fonction en utilisant N individus ou objets. Chaque individu x_i (avec $i \in \{1, \cdots, N\}$) est défini par sa position $(x_i^1, \cdots, x_i^n)$ dans l'espace à n dimensions. Par exemple, si $n = 3$, la

position d'un individu est sa position dans l'espace. Le GSA fonctionne alors de manière itérative. A chaque étape de l'algorithme, la position d'un individu évolue en fonction de la position des autres individus et de leur masse du fait des forces gravitationnelles. La masse d'un individu est liée à la valeur de la fonction objectif pour cet individu.

La première étape du GSA consiste à générer une population initiale de N individus en tirant aléatoirement leur position.

Dans un deuxième temps, l'idée est d'appliquer la force gravitationnelle à ces individus pour les faire se déplacer dans l'espace et s'approcher de la solution optimale. Le GSA s'arrête lorsqu'il atteint un critère de fin.

Plus précisément, une itération se déroule de la manière suivante. Le GSA commence par calculer la valeur de chaque individu dans le problème. On note $f(x_i(t))$ la valeur de l'individu i au temps t.

Avec cette valeur, le GSA calcule les masses M_i de chaque individu i de la façon suivante :

$$M_i(t) = \frac{m_i(t)}{\sum_{i=0}^{N} m_i(t)}$$

avec

$$\begin{aligned} m_i(t) &= \frac{f(x_i(t)) - PIRE}{MEILLEUR - PIRE} \\ MEILLEUR &= \max_{i \in \{1,\dots,N\}} f(x_i(t)) \\ PIRE &= \min_{i \in \{1,\dots,N\}} f(x_i(t)). \end{aligned}$$

Ensuite, le GSA fait diminuer la constante de gravitation. Initialement, les individus sont très éloignés les uns des autres. Il faut donc des forces élevées pour qu'ils se déplacent et explorent l'espace. Dans un second temps, lorsque les individus sont proches les uns des autres, on peut faire baisser l'intensité des forces pour obtenir un résultat plus précis.

Pour éviter que des individus qui seraient éloignés influent négativement sur les autres individus, on retire l'influence des k pires individus, où k évolue linéairement avec le temps en allant de 0 jusqu'à $N-1$. On définit alors $KBEST$ comme l'ensemble des $N-k$ meilleurs individus.

L'algorithme fait alors évoluer la valeur des forces. La force qui s'applique dans la dimension d de l'individu x_i est égale à :

$$F^d(x_i)(t) = \sum_{j=0,\ j \in KBEST}^{N} F_{ij}(t) * random(0,1)$$

avec

$$F_{ij}(t) = G(t) * \frac{M_i(t) * M_j(t)}{R_{ij}(t) + \epsilon} * (x_j^d(t) - x_i^d(t)),$$

où $R_{ij}(t)$ est la distance euclidienne entre i et j.

On remarque trois différences par rapport à la définition de la gravité :

1. Ajout d'un terme ϵ : on ajoute une constante $\epsilon > 0$ proche de zéro qui permet de s'assurer que la force F_{ij} ne supplante pas les autres forces si R_{ij} est très proche de 0.
2. Terme $random(0, 1)$: ce terme permet d'ajouter une composante stochastique à l'algorithme. L'objectif est d'éviter de se faire piéger dans un optimum local.
3. $j \in KBEST$: vu plus haut.

Une fois qu'on a calculé les forces, l'algorithme fait évoluer l'accélération de chaque individu grâce à la formule :

$$a_i^d(t) = \frac{F^d(x_i)(t)}{M_i}.$$

On peut alors mettre à jour la vitesse de chaque individu par la formule :

$$v_i^d(t+1) = random(0, 1) * v_i^d(t) + a_i^d(t),$$

où le terme $random(0, 1)$ est introduit pour donner une caractère stochastique à la recherche de solution. On peut enfin recalculer la position de chaque individu dans l'espace :

$$x_i^d(t+1) = x_i^d(t) + v_i^d(t+1).$$

Pour finir, l'algorithme vérifie si on a atteint le critère de fin. Si oui, il s'arrête. Si non, l'algorithme recommence une itération.

On présente ci-dessous l'algorithme en pseudo code.

Algorithm 1 GSA

1: Définition du problème
2: Génération de la population initiale : N tirages aléatoire $x_i = (x_i^1, ..., x_i^n)$
3: **repeat**
4: Évaluer la valeur des x_i dans le problème
5: Calculer les nouvelles masses de chaque individu x_i : M_i
6: Mettre à jour la constante de gravitation G
7: Calculer les forces entre chaque couple d'individus (x_i, x_j)
8: Mettre à jour l'accélération a_i de chaque individu x_i
9: Mettre à jour la vitesse v_i de chaque individu x_i
10: Mettre à jour la position de chaque individu x_i
11: **until** Condition de fin

2.3 Résultats expérimentaux de la littérature

Les chercheurs qui ont introduit le GSA ont démontré sa supériorité par rapport à d'autres algorithmes d'optimisation lorsque l'on cherche à optimiser certaines fonctions avec beaucoup de dimensions.

Par exemple, le GSA donne de bons résultats par rapport à aux algorithmes PSO (pour Particle Swarm Optimization) et RGA (pour Real Genetic Algorithm). Le tableau ci-dessous montre la distance au minimum global du résultat obtenu par chaque algorithme, pour trois fonctions différentes. Comme on peut le voir, les résultats obtenus par GSA sont meilleurs que ceux de PSO et RGA pour ces fonctions.

Fonction à minimiser	GSA	PSO	RGA
$\sum_{i=1}^{30} x_i^2$	$2.1 * 10^{-10}$	$5.0 * 10^{-2}$	23.45
$\max_{1 \geq i \geq 30}(x_i)$	$8.5 * 10^{-6}$	23.6	11.78
$-20 \exp -0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2} - \exp \frac{1}{n}\sum_{i=1}^{n} \cos(2\pi x_i) + 20 + e$	$1.1 * 10^{-5}$	0.02	2.15

Les algorithmes de métaheuristiques sont notamment très utilisés en ingénierie électrique. Des auteurs ont étudié les performances du GSA dans ce domaine. La figure ci-dessous, tirée de l'article [5], présente les résultats de tests pour optimiser une fonction de perte pour un problème dans lequel on cherche à répartir de façon optimale la puissance électrique. Le tableau montre que les algorithmes les plus performants sont l'algorithme GSA ainsi qu'une de ses variantes, OGSA. Les chiffres du tableau représentent la consommation électrique pour chaque tâche que l'on cherche à optimiser. OGSA a une perte minimale de 4.4984 MW.

Table 2
Comparison of simulation results for IEEE 30-bus test power system with P_{Loss} minimization objective.

Variable	OGSA	GSA [20]	BBO [40]	DE [16]	CLPSO [41]	PSO [41]	SARGA [42]
Generator voltage							
V_1, p.u.	1.0500	1.071652	1.1000	1.1000	1.1000	1.1000	NR*
V_2, p.u.	1.0410	1.022199	1.0944	1.0931	1.1000	1.1000	NR*
V_5, p.u.	1.0154	1.040094	1.0749	1.0736	1.0795	1.0867	NR*
V_8, p.u.	1.0267	1.050721	1.0768	1.0756	1.1000	1.1000	NR*
V_{11}, p.u.	1.0082	0.977122	1.0999	1.1000	1.1000	1.1000	NR*
V_{13}, p.u.	1.0500	0.967650	1.0999	1.1000	1.1000	1.1000	NR*
Transformer tap ratio							
T_{6-9}	1.0585	1.098450	1.0435	1.0465	0.9154	0.9587	NR*
T_{6-10}	0.9089	0.982481	0.90117	0.9097	0.9000	1.0543	NR*
T_{4-12}	1.0141	1.095909	0.98244	0.9867	0.9000	1.0024	NR*
T_{28-27}	1.0182	1.059339	0.96918	0.9689	0.9397	0.9755	NR*
Capacitor banks							
Q_{C-10}, p.u.	0.0330	1.653790	4.9998	5.0000	4.9265	4.2803	NR*
Q_{C-12}, p.u.	0.0249	4.372261	4.987	5.0000	5.0000	5.0000	NR*
Q_{C-15}, p.u.	0.0177	0.119957	4.9906	5.0000	5.0000	3.0288	NR*
Q_{C-17}, p.u.	0.0500	2.087617	4.997	5.0000	5.0000	4.0365	NR*
Q_{C-20}, p.u.	0.0334	0.357729	4.9901	4.4060	5.0000	2.6697	NR*
Q_{C-21}, p.u.	0.0403	0.260254	4.9946	5.0000	5.0000	3.8894	NR*
Q_{C-23}, p.u.	0.0269	0.000000	3.8753	2.8004	5.0000	0.0000	NR*
Q_{C-24}, p.u.	0.0500	1.383953	4.9867	5.0000	5.0000	3.5879	NR*
Q_{C-29}, p.u.	0.0194	0.000317	2.9098	2.5979	5.0000	2.8415	NR*
P_{Loss}, MW	**4.4984**	4.514310	4.5511	4.5550	4.5615	4.6282	4.57401
TVD, p.u.	0.8085	0.875220	NR*	1.9589	0.4773	1.0883	NR*
L-index, p.u.	0.1407	0.141090	NR*	0.5513	NR*	NR*	NR*
CPU time, s	89.19	94.6938	NR*	NR*	138	130	NR*

NR* means not reported.

FIGURE 1 – Comparaison des algorithmes GSA et OGSA avec d'autres algorithmes d'optimisation pour un problème d'ingénierie électrique.

2.4 Application

Dans cette partie, nous présentons un exemple d'application de l'algorithme GSA. L'algorithme est codé en Python. [1]

La fonction à optimiser est la fonction $f(x) = -x^2 + 400$, où x prend des valeurs dans l'intervalle $[-20, 20]$. L'optimum global à trouver est donc $x = 0$.

L'algorithme est configuré comme suit. Comme critère de fin, on demande à l'algorithme de s'arrêter au bout de 100 itérations, donc quand $t = T = 100$. La constante de gravitation évolue au fil des interactions suivant la fonction $G(t) = G_0 * e^{-\alpha * t/T}$, avec $G_0 = 100$ et $\alpha = 20$. Enfin, on génère $N = 50$ individus.

Les figures ci-dessous représentent la fonction f sur son domaine $[-20, 20]$ ainsi que la valeur des individus x_i en orange, à différentes étapes de l'exécution de l'algorithme.

On remarque que la valeur des individus s'approche de l'optimum au bout de quelques itérations seulement. Lorsque l'algorithme s'arrête (après 100 itérations), on obtient souvent une précision d'au moins 10^{-14}.

Cependant, il est important de noter que la fonction à optimiser est ici très simple, uni-dimensionnelle, ce type d'algorithme étant surtout utilisé pour des problèmes très complexes, multi-dimensionnels.

1. Le code Python ainsi que cet article et une vidéo d'illustration sont disponibles à l'adresse `https://github.com/Boubou14/GSA_python_implementation.git`.

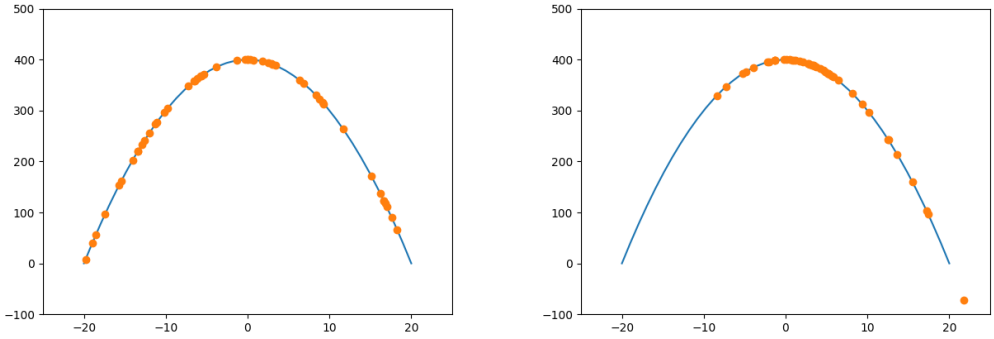

FIGURE 2 – Itération du GSA pour $t = 0$ (gauche) et $t = 5$ (droite).

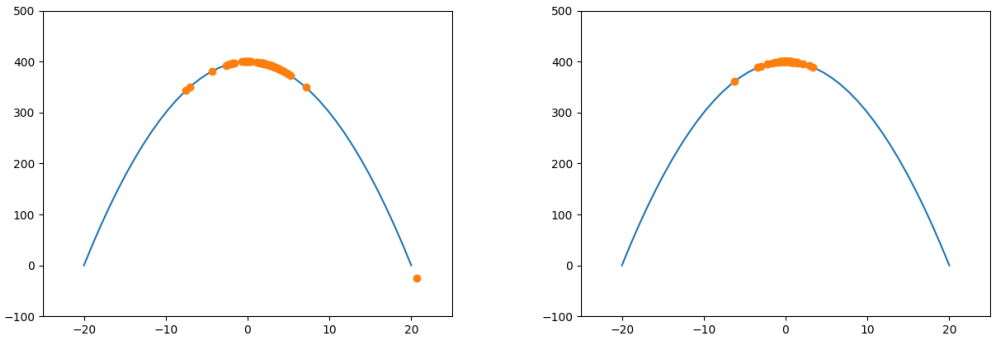

FIGURE 3 – Itération du GSA pour $t = 10$ (gauche) et $t = 15$ (droite).

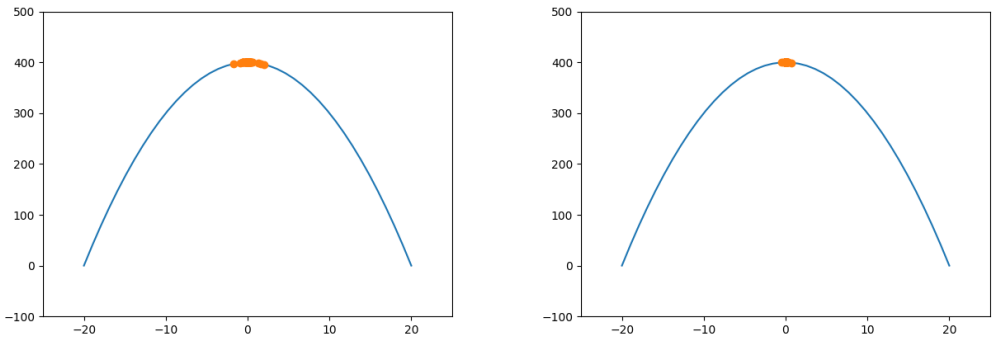

FIGURE 4 – Itération du GSA pour $t = 20$ (gauche) et $t = 25$ (droite).

Références

[1] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Graduate School of Industrial Administration, CMU, Report 388, 2014.

[2] Ming Niu, Can Wan, and Zhao Xu. A review on applications of heuristic opti-

mization algorithms for optimal power flow in modern power systems. *Journal of Modern Power Systems and Clean Energy*, 2 :289–297, 2014.

[3] Esmat Rashedi, Hossein Nezamabadi-pour, and Saeid Saryazdi. GSA : A Gravitational Search Algorithm. *Information Sciences*, 179(13) :2232–2248, 2009. Special Section on High Order Fuzzy Sets.

[4] Esmat Rashedi, Elaheh Rashedi, and Hossein Nezamabadi-pour. A comprehensive survey on gravitational search algorithm. *Swarm and Evolutionary Computation*, 41 :141–158, 2018.

[5] Binod Shaw, V. Mukherjee, and Sakti Ghoshal. Solution of reactive power dispatch of power systems by an opposition-based gravitational search algorithm. *International Journal of Electrical Power and Energy Systems*, 55 :29–40, 02 2014.