Steiner Problem

Sébastien Patte

November 2020

Contents

1 Description / Modélisation du problème

Le problème de Steiner consiste à trouver, pour certains points de l'espace donnés, un ensemble de chemins qui relie tous ces points tels que la distance cumulée des chemins est minimale. Ce problème a des applications pour la conception de circuits électronique, dans les télécommunications, ...

Pour modéliser ce problème on utilise un graphe $G = (V, E)$ avec $T \subset V$ un sous-ensemble de sommets qu'on veut relier entre eux qu'on appelle les sommets terminaux.

On doit ensuite trouver une solution sous la forme d'un graphe $G' = (V', E')$ qui est un sous-graphe de G tel que $T \subset V'$. Comme on doit trouver une solution qui minimise les distances entre les terminaux, on évalue les solutions avec une fonction $eval(G', T) = \sum_{e \in E'} weight(e)$ c'est à dire une fonction qui renvoie le poids total du sous-graphe solution. L'objectif est alors de trouver une solution G' qui minimise $eval(G', T)$.

2 Complexité

2.1 Le problème de Steiner est NP

Pour vérifier si une solution au problème de Steiner est correcte il faut :

- **Vérifier que la solution est un arbre :**
  Comme nos graphes on des arêtes non-dirigées il faut prouver qu'il n'y a pas de cycles, on peut le faire avec un Depth-first Search qui a la complexité $O(|V'| + |E'|)$
- **Vérifier que l'arbre atteint tout les terminaux :**
  Pour chaque terminal, on vérifie qu'il est dans l'ensemble des sommets V', ce qui a une complexité constante. Au total cette étape a donc une complexité de $O(|T|)$

Au total la complexité de l'algorithme de vérification d'une solution est de $O(|V'|+|E'|+|T|)$ Ces vérifications se font en un temps mois que polynomial donc le problème de Steiner est NP.

2.2 Le problème de Steiner est NP-complet

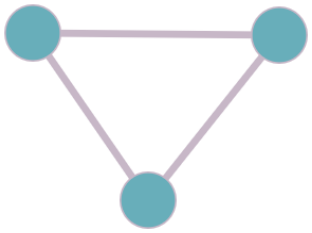
Pour montrer que le problème du Steiner Tree Problem (STP) est NP-complet on va utiliser le problème de Vertex Cover (VC). On sait que VC est un problème NP-complet donc si on arrive a trouver une réduction de VC en STP alors STP est NP-complet.

2.2.1 Vertex Cover

Soit un graphe $G = (V, E)$, on cherche un sous-ensemble de sommets $S \subset V$ tel que $|S| = k$ qui couvre toutes les arêtes de G. Le problème VC prend donc en paramètre un graphe G et un entier k

2.2.2 Réduction

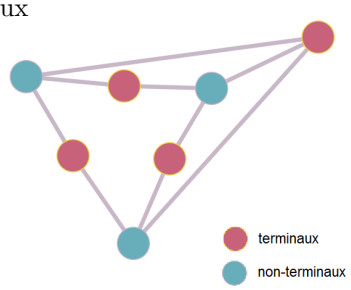
On part d'un graphe G qu'on considère pour le problème de Vertex Cover, comme celui ci-contre par exemple.



1. Pour chaque arête du graphe G, on la divise en 2 pour relier ces 2 parties par un nouveau noeud terminal.

2. Un ajoute un noeud terminal qu'on relie a tout les noeuds non-terminaux

On a alors un graphe qui G' correspond a un problème de Steiner (comme celui ci-contre pour garder le même exemple). Et on a une solution au problème de Steiner avec $|E| + k$ arêtes pour G', si et seulement si il existe une solution au problème VC avec k sommets pour G.
On a donc trouvé une réduction de Vertex Cover en problème de Steiner. Or Vertex Cover est NP-complet, donc le problème de Steiner est aussi NP-complet.



3 Approximation

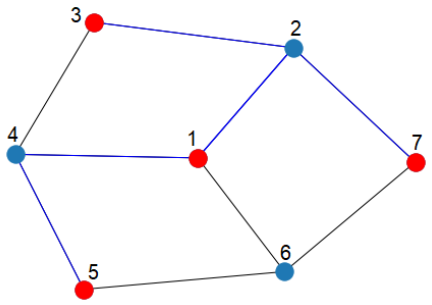
Le problème de Steiner est NP-complet mais on peut en faire une approximation avec un arbre couvrant en un temps polynomial.

3.1 Fonctionnement et complexité

1. On calcule le plus court chemin pour chaque paire de terminaux qu'on stocke dans un dictionnaire sous forme de listes d'arêtes. Et on construit un graphe G_T le graphe complet des plus courts chemins entre les terminaux. Pour cela on utilise l'algorithme de Dijkstra ce qui donne une complexité de $\Theta(|V|^2 \cdot log|V|)$.
2. On construit $A = (T, V_A)$ l'arbre couvrant de poids minimum de G_T, avec une complexité de $\Theta(|T|^2 \cdot log|T|)$
3. Pour chaque arête de A on récupère la liste d'arêtes correspondante, ce se fait en $O(|V_A|^2)$ car on a déjà stocké les plus courts chemins correspondant à chaque arête. Puis on fusionne ces listes d'arêtes, la liste résultante est alors celle des arêtes de G qu'on utilise dans notre solution.

La complexité totale de l'algorithme d'approximation est alors de $O(|V|^2 \cdot log|V| + |T|^2 \cdot log|T| + |V_A|^2)$ soit une complexité polynomiale.

Une solution est modélisée par une liste d'arêtes, cette solution représente un sous-graphe de G dans lequel on utilise que les arêtes listées dans *sol* et qui doit être sous forme d'arbre (c-a-d un graphe connecté sans cycle). Pour évaluer cette solution, on construit le sous-graphe de G correspondant, on vérifie que c'est un arbre, et on renvoie son poids total. Par exemple sur l'image ci-contre, représentant une solution pour le graphe "test.stp" fourni, les arêtes en bleu correspondent aux arêtes utilisées par la solution $[(1,2),(2,3),(1,4),(4,5),(2,7)]$. Ici la solution est bien un arbre reliant les noeuds terminaux (en rouge) et comme le poids des arêtes est 1 alors la solution a la valeur 5 ce qui correspond bien a une solution optimale pour ce graphe.



3.2 Rapport d'approximation

On considère un arbre de Steiner optimal T^* :

En faisant un depth-first search sur T^* on obtient un chemin que l'on appellera C. Ce chemin passe exactement 2 fois par chaque arête de T^* donc le poids total de ce chemin est $w(C) = 2 \cdot w(T^*)$.

Ensuite on adapte ce chemin au graphe des plus courts chemins entre les terminaux, G_T. Pour cela on construit un chemin C' qui correspond a C où on a retiré tout les noeuds non-terminaux. C' correspond alors a un chemin dans G_T et comme les arêtes de G_T sont des plus courts chemins, chacune d'entre elles ne peut pas avoir de poids plus grand que leur équivalent dans C. On a alors $w(C') \leq w(C) = 2 \cdot w(T^*)$.

Puis on retire les noeuds dupliqués dans C' pour obtenir un nouveau chemin C''. Comme on a juste retiré des arêtes si nécessaire, on a alors $w(C'') \leq w(C') \leq w(C) = 2 \cdot w(T^*)$.

Alors C'' est un arbre couvrant de G_T. Et comme A est l'arbre couvrant minimal de G_T, on a $w(A) \leq w(C'') \leq 2 \cdot w(T^*)$.

Finalement le graphe solution G_{sol} qu'on construit à partir de G et A, ne peut pas avoir de poids total supérieur à celui de A car on remplace les arêtes de A par des arêtes qui ont exactement le même poids. Par contre le poids peut être inférieur car une arête du graphe solution peut être utilisée par plusieurs arêtes de A. On a alors $w(G_{sol}) \leq w(A) \leq 2 \cdot w(T^*)$ et par extension $w(T^*) \leq w(G_{sol}) \leq w(A) \leq 2 \cdot w(T^*)$.

Donc la valeur de la solution renvoyée par l'algorithme d'approximation est toujours située entre la valeur optimale et son double. Cet algorithme est alors une **2-approximation**.

4 Métaheuristique

Une solution est sous la forme d'un tableau de bits (par exemple $[0, 1, 0, 1, 1, 0]$) représentant les sommets de G qu'on utilise dans la solution.

L'algorithme utilise les fonctions suivantes :

- **init(graph, terms) :** renvoie une solution aléatoire
- **voisinage(sol, graph, terms) :** On change un bit au hasard dans sol, puis on a une chance sur $|V|$ de changer un deuxième bit.
- **eval(sol, graph, terms) :** renvoie le poids total du sous-graphe de G désigné par sol + un malus pour chaque terme qui n'est pas relié + un malus par composante connexe quand il y a plusieurs.
- **choix(I, nI) :** A chaque tour de boucle on doit choisir entre la solution actuelle I et la solution voisine nI. Pour cela on choisit nI à tout les coups si $val_I < val_{nI}$. Et sinon on choisit nI avec une probabilité $e^{-(val_{nI} - val_I)/T}$.
- **recuit(G, terms, T_{min}) :** On appelle init() pour avoir une solution initiale et la température T. On parcoure la boucle tant qu'on n'est pas passé en dessous du seuil de température T_{min}. A chaque tour de boucle on diminue T en le multipliant par une valeur λ_T tel que $0 < \lambda_T < 1$. Ensuite on choisit une des 2 solutions avec choix(I,nI).

Algorithm 1 recuit(graph, terms, T_{init}, T_{min})

$I \leftarrow init()$
$T \leftarrow T_{min}$
while $T < T_{min}$ **do**
    $nI \leftarrow voisinage(I, graph, terms)$
    $val_I \leftarrow eval(I)$
    $val_{nI} \leftarrow eval(nI)$
    if $val_{nI} < val_I$ **then**
        $I \leftarrow nI$
    else if $rand() < e^{-\frac{val_{nI} - val_I}{T}}$ **then**
        $I \leftarrow nI$
        $val_I \leftarrow val_{nI}$
    end if
    $T \leftarrow T \times \lambda_T$
end while

Aussi pour éviter que l'algorithme ne choisisse une solution avec une valeur plus grande peu avant la fin (bien que la probabilité soit faible), au long de l'exécution on stocke la valeur d'évaluation la plus basse qu'on a rencontrée et à la fin de la fonction recuit on ajoute cette valeur après toutes les valeurs explorées.

5 Résultats expérimentaux

5.1 Construction des graphiques

Pour dessiner les graphiques on utilise les 2 fonctions **ComputeInstance** et **PlotFromFile**.

ComputeInstance :

La fonction **ComputeInstance** lance la fonction recuit 100 fois et récupère toutes les valeurs explorées pendant l'estimation pour en extraire des moyennes. On choisit 10 points parmi ceux explorés, on fait une moyenne sur les 100 runs pour chacun de ces points, puis on calcule un intervalle de confiance pour chaque point.

Ces résultats qu'on a calculé sont ensuite sauvegardés dans un fichier, le nom du fichier est généré à partir des paramètres du recuit, on écrit T_{init} puis la partie décimale de T_{min} et λ_T. Par exemple pour $T_{init} = 1000$, $T_{min} = 0.1$ et $\lambda_T = 0.99$ le nom du fichier sera "1000_1_9". Et le fichier sera enregistré dans un sous-dossier de "runs/" dont on donne le nom à ComputeInstance en paramètre.

Dans chaque ligne du fichier un écrit la valeur d'un des points de la courbe, la valeur de la borne inférieure de son intervalle de confiance puis la borne supérieure. Ce qui permet de dessiner la courbe ultérieurement et de changer la manière dont on la dessine sans être obligé de relancer toutes les estimations.

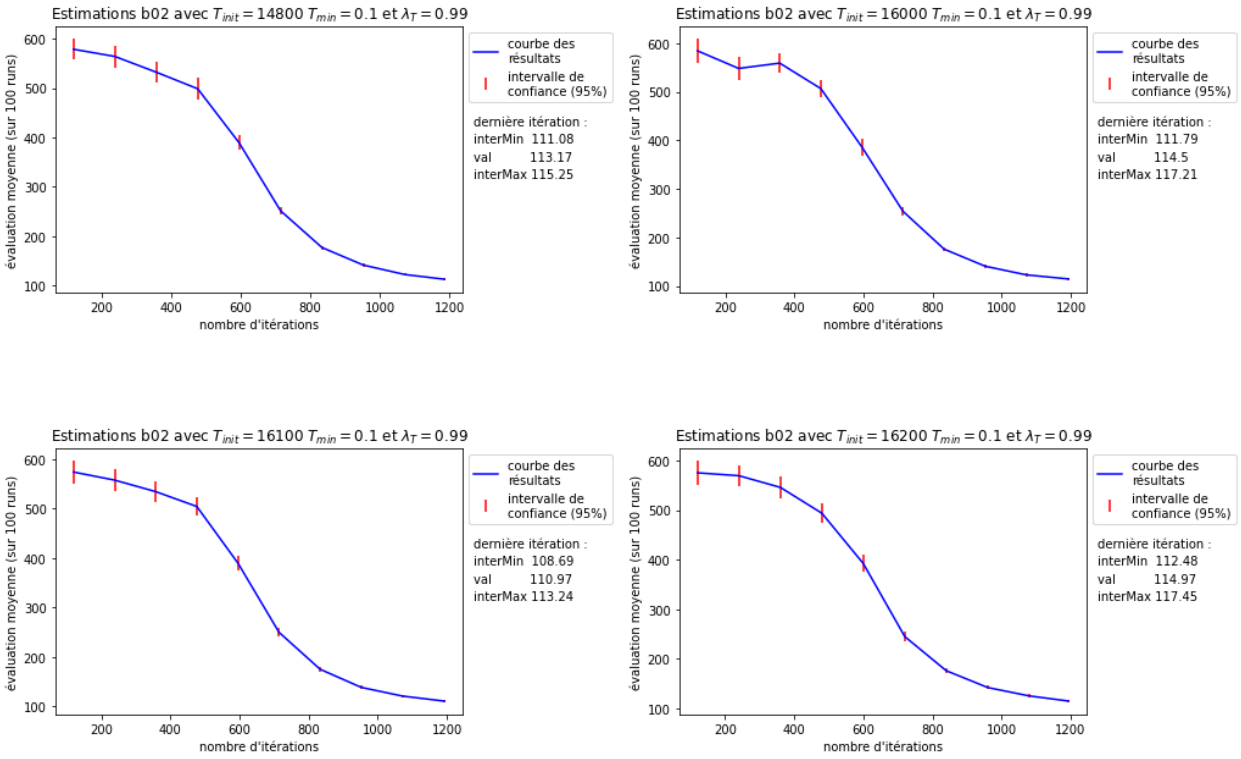
PlotFromFile :

La fonction **PlotFromFile** prend en paramètre T_{init}, T_{min}, λ_T et un nom de sous dossier de "runs/". La fonction va alors chercher un fichier dans ce sous-dossier qui corresponde aux paramètres. Si elle trouve un fichier correspondant elle récupère les données du fichier (les points et intervalles de confiances) et dessine un graphique à partir de ces données avec un titre et des légendes.

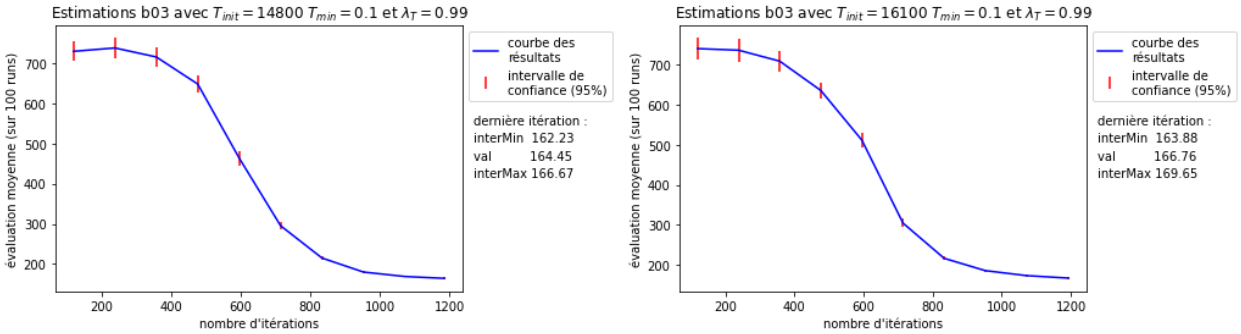
5.2 Choix des paramètres

Pour choisir les paramètres, on va d'abord utiliser le graphe b02.stp fourni par steinlib pour lequel ils indiquent que la solution optimale est 83.

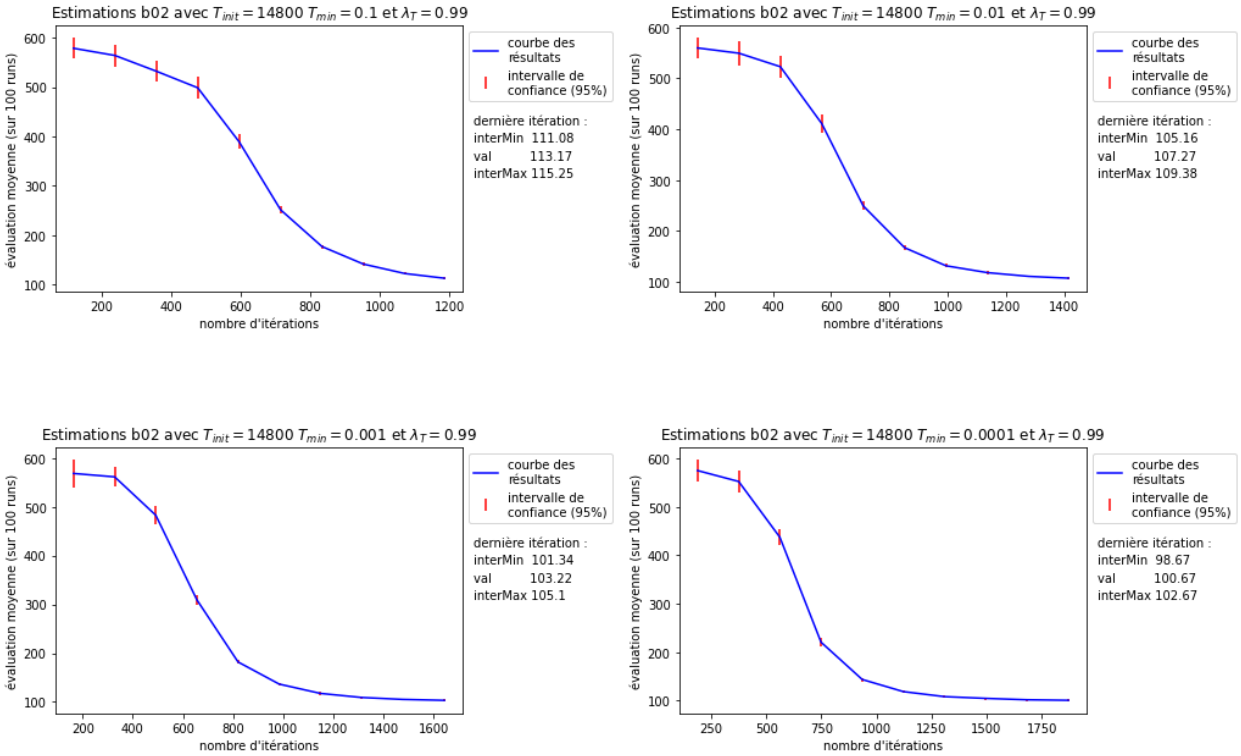
Sur les courbes ci-dessous comparant les valeurs de T_{init} pour le graphe b02, on voit que la courbe où $T_{init} = 16100$ est celle qui donne la moyenne la plus basse à la dernière itération, suivi par $T_{init} = 14800$.



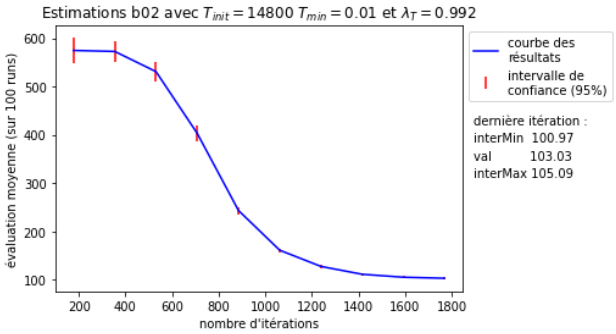
Cependant on voit sur les 2 courbes ci-dessous que pour le graphe b03 avec $T_{init} = 16100$ la moyenne à la dernière itération est significativement plus élevée que pour $T_{init} = 14800$ (peut être car b03 est bien plus grand que b02). On va donc choisir $T_{init} = 14800$ qui est quand même un bon paramètre pour b02 et permet d'améliorer les résultats pour un graphe plus grand comme b03.

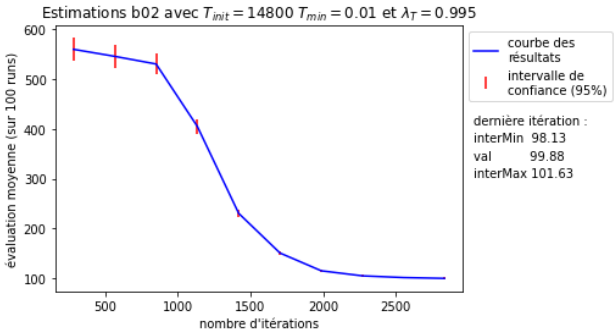


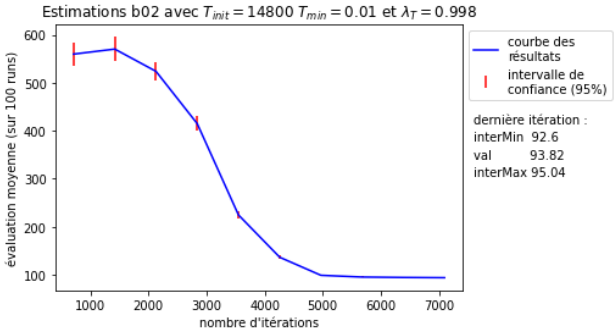
Sur les courbes ci-dessous on observe que plus on baisse T_{min}, plus la courbe se creuse et la moyenne à la dernière itération est faible. Mais entre $T_{min} = 0.1$ et 0.01 on baisse cette moyenne de 6 alors qu'entre $T_{min} = 0.01$ et 0.001 la différence est de 4, et entre 0.001 et 0.0001 la différence est de moins de 3. Donc plus on baisse T_{min} moins l'avantage est visible, et plus le temps d'exécution augmente.

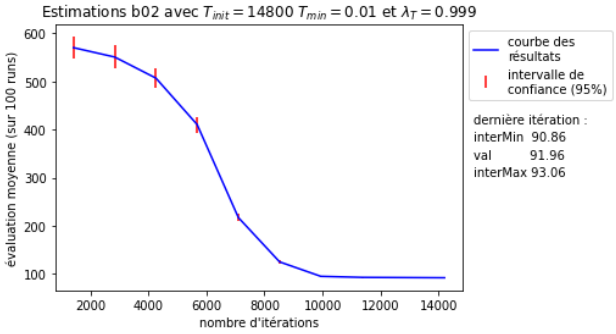


Avec les courbes ci-dessous on constate que plus on augmente λ_T, plus la moyenne à la dernière itération est basse. Cependant la durée d'exécution augmente aussi, pour 100 runs avec $T_{min} = 0.01$ et $\lambda_T = 0.999$ on met environs 20 minutes. C'est pourquoi par la suite on va utiliser un $T_{min} = 0.1$.

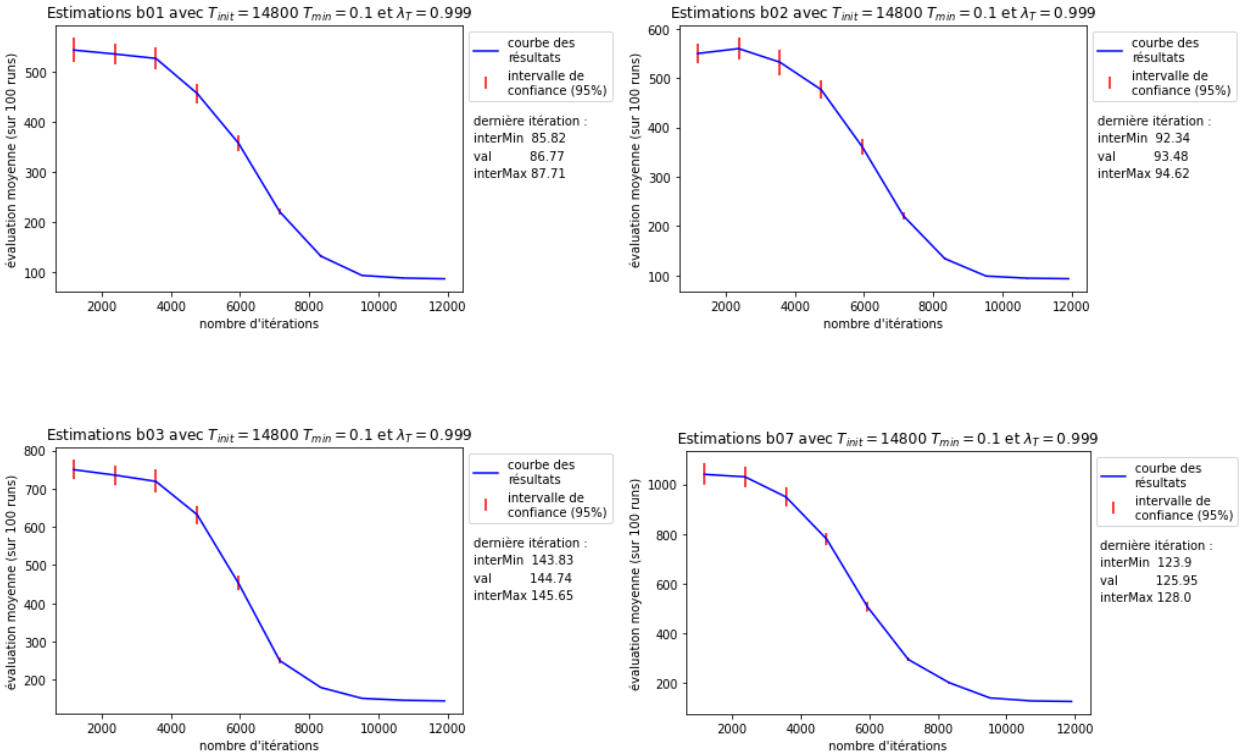








Ci-dessous les résultats pour les graphes b01, b02, b03 et b07 qui on respectivement une valeur optimale de 82, 83, 138 et 111. Même si on n'atteint pas les valeurs optimales dans l'intervalle de confiance, on s'en rapproche quand même fortement, et sur 100 runs on peut quand même espérer trouver la valeur optimale plusieurs fois.



6 Conclusion

L'avantage de l'approximation c'est qu'on peut calculer une solution en un temps polynomial, mais suivant le rapport d'approximation (si on a pu en prouver un), la valeur de la solution renvoyée peut être bien plus grande que la valeur optimale. Par exemple ici avec l'approximation de Steiner on peut avoir une solution dont la valeur va jusqu'à 2 fois la valeur optimale. Cependant si on a un temps et/ou des capacités de calcul limitée et où il n'est pas nécessaire de trouver à tout prix la solution optimale, alors il peut être intéressant d'utiliser un approximation polynomiale.

La métaheuristique a un temps d'exécution plus important et ne nous donne pas de garantie sur les résultats qu'on peut obtenir. Néanmoins elle repose sur de l'aléatoire et explore donc des solutions différentes à chaque exécution. Alors en lançant un bon nombre de fois la métaheuristique on peut espérer trouver une solution optimale. De plus dans notre cas, avec les paramètres qu'on a choisis, l'intervalle de confiance pour la dernière itération est bien plus bas que pour une 2-approximation.

7 Sources

- https://networkx.org/documentation/stable/
- https://en.wikipedia.org/wiki/Steiner_tree_problem
- https://theory.stanford.edu/~trevisan/cs261/lecture01.pdf
- https://theory.stanford.edu/~trevisan/cs261/lecture02.pdf