Optimisation d'utilité sur un graphe

Dans ce projet, nous sommes donnés des listes de routes sous la forme d'un triplet d'entier représentant le départ, l'arrivée et la distannce. Nous sommes également donnés une liste de camions avec chacun une distance maximale de trajet et un coût. On considère que l'offre de camion est illimitée.

On imagine qu'une entreprise a une liste de trajet qu'elle veut effectuer avec pour chacun le départ, la destination et l'utilité qu'elle gagne de ce trajet, typiquement le profit. En fonction d'un budget N, on cherche alors à déterminer quelles trajets emprunter et quels camion sélectionner pour maximiser l'utilité.

On doit alors traiter plusieurs sous problèmes.

Dans un premier temps, on doit déterminer pour chaque trajet la distance totale qu'il représente puis le camion le plus adapté à le réaliser et enfin le coût du trajet. Cela revient à déterminer le chemin le plus court dans un arbre pondéré positif, puis à sélectionner le camion le moins couteux qui peut couvrir cette distance.

Dans un deuxième, il faut déterminer la liste de trajets optimale à emprunter pour maximiser l'utilité. Comme une solution optimale déterministe pourait avoir une complexité temporelle trop importante, on aura notamment recours à des heuristiques pour l'approcher.

Notation : **V** : nombre de sommets ; **E** : nombre d'arêtes ; **T** : nombre de trajets intéressants

Dans un premier temps, on s'intéresse à des problèmes de programation usuels sur les graphes dont notamment :

— Décomposition en composantes connexes
— Plus court(s) chemin(s) entre deux noeuds (DFS)
— Construction de l'arbre minimal couvrant (kruskal)
— Adaptation des fonctions à l'arbre minimal couvrant

Fonction *get_path_with_power*

Complexité Temporelle : O(E)

On garde en mémoire les arêtes déjà visitées dans un dictionaire. Ainsi on ne fera au plus que E appels récursifs.
En effet, si par l'absurde emprunter une arête déjà visitée permettait de trouver un chemin admissible, alors lors d'un appel récursif précédent, on l'aurait déjà trouvé et la fonction se serait terminée.

— Vérifier qu'une valeur est une clé d'un dictionaire est en coût constant O(1)
— Ajouter une valeur à un dictionnaire est en coût constant O(1)
— Comparer deux valeurs auxquelles on a accès en coût constant O(1) est en coût constant

Le corps de la boucle "for n in neighbors" est donc constitué de l'appel récursif et d'opérations en coût constant O(1). Les opérations en coût constant O(1) vont être effectuées au plus O(E) fois au total. En effet, à chaque fois que l'on arrive sur un sommet, on effectue ces opérations en cout O(1) pour tous ces voisins, c'est-à-dire le nombre d'arêtes connectées à ce sommet.

Au total, grâce à la mémorisation des arêtes visitées, on retrouve la complexité anoncée.

Complexité Spatiale : O(E+V)

On utilise un dictionnaire de taille en mémoire O(E). On effectue, grace au dictionnaire, au plus E appels récursifs. La longueur de la liste renvoyée est au plus V. La complexité spatiale est donc O(E+V)

Complexité de la fonction *min_power*

Complexité Temporelle : O(E log(E))

On itère sur les noeuds mais le nombre d'ajouts de power à la liste au total est 2×E car chaque arête est vue deux fois. La construction de cette liste prend donc O(E) opérations élémentaires.

Ensuite, transformer la liste en ensemble puis à nouveau en liste prend un nombre d'opérations de l'ordre de grandeur de la longueur de la liste, i.e. 2xE, cela prend donc O(E) opérations ici.

Trier la liste prend O(E log(E)) opérations élémentaires.

La recherche dichotomique prend un nombre d'itération de l'ordre de grandeur le log la longueur de la liste sur laquelle on l'effectue, ici O(log(E)) donc. Or dans chaque itération, on appelle la fonction *get_path_with_power* qui prend un nombre d'opérations élémentaires O(E). Le nombre d'opération élémentaires effectuées pour la recherche dichotomique est donc O(E log(E))

Au total, la complexité temporelle est donc : O(E + E log(E) + O(E) log(E)) = O(E log(E)). L'implémentation est sous-optimale puisque l'on sait que chaque arête est prise en compte deux fois, il suffirait de répertorier dans un dictionnaire les arêtes déjà croisée.

La vérification serait en temps constant O(1) donc cela ne changerait pas la complexité de l'algorithme mais cela éviterait d'avoir recours à la fonction *set* qui est couteuse et ainsi évitable. Nous avons pris le parti d'un code plus clair, puisque de toute manière cette version de min_power ne suffira pas pour les plus gros graphes et sera re-implémentée plus tard dans le projet.

Complexité Spatiale : O(E log(E))

La liste (que l'on trie en place dans l'implémentation), est de longueur E. Cependant chaque itération de *get_path_with_power* prend une place en mémoire de l'ordre de O(E). On fait un nombre O(log(E)) d'itérations de *get_path_with_power*. Le reste des variables prend une place constante en mémoire O(1).

Au total l'algorithme utilise une place en mémoire de l'ordre de O(E log(E))

(N.B. Il était possible d'éviter de dresser la liste *powers* et de directement procéder par dichotomie sur power jusqu'à obtenir une précision 10^{-16}. Nous avons préféré une solution théoriquement exacte puisque de toute manière cette implémentation ne sera pas assez efficace pour les gros graphes. Cette autre solution peut être plus rapide en pratique toutefois.)

Un arbre possédant V sommets possède exactement V-1 arrêtes.

Démonstration : Un arbre est une structure récursive, il serait possible de procéder par induction structurelle. Cependant l'énoncé adopte le point de vue d'un graphe connexe acyclique, on prend donc la preuve selon ce point de vue.

Etape 1 : Pour un graphe connexe, $E \geq V - 1$

Pour V = 1, cette proposition est directement vraie.

Soit $n \geq 1$ et soit G un graphe connexe à $n+1$ noeuds. Supposons que tous les graphe connexe à n noeuds possèdent au moins $n-1$ arêtes.

S'il existe un noeud de degré 1 dans G, en le supprimant avec l'arête qui le lie au graphe on obtient un graphe connexe à n noeuds. Ce sous graphe possède alors par hypothèse au moins $n-1$ arêtes et donc G en possédait au moins n. Sinon tous les noeuds sont de degré au moins deux. Alors $2n = \sum_{n \in G} 2 \leq \sum_{n \in G} deg(n) = 2 * E$ et donc G possède bien au moins n arêtes.

Etape 2 : Pour un graphe acyclique donc $E \leq V - 1$

.

***Lemme* :** Si tous les noeuds d'un graphe sont de degré au moins 2, alors il existe un cycle

Dém : On note $v_1, ..., v_n$ les sommets d'un tel graphe. On construit un chemin récursivement de la manière suivante : $v_{i_0} = v_1$; $v_{i_{k+1}}$ est un voisin de v_{i_k} ; $v_{i_{k+1}} \in \{v_{i_0}, ..., v_{i_k}\}$ jusqu'à ce que ne soit plus possible. Comme le graphe est fini, ce processus s'arrête, on note N le nombre d'étapes de construction. v_{i_N} possède alors un voisin dans $\{v_{i_0}, ..., v_{i_{N-1}}\}$ sinon on pourrait continuer à construire la suite (on suppose $N \geq 2$ le cas non trivial). On a ainsi construit un cycle.

On prouve maintenant la propriété par récurrence.
Un graphe acyclique à 1 noeud a bien au plus 0 arêtes.

Soit G un graphe acyclique de taille $n+1 \geq 2$, on suppose que tout graphe acyclique de taille n possède au plus $n-1$ arêtes. Par le lemme, G possède un noeud de degré 0 ou 1. En le supprimant ainsi que l'éventuelle arête connectée à lui, on obtient un graphe acyclique de taille n. On applique l'hypothèse de récurrence à ce graphe qui possède alors au plus $n-1$ arête, on déduit ensuite que G possédait au plus n arêtes.

Conclusion :
Un arbre de taille V est un graphe connexe acyclique et possède donc exactement $V - 1$ arêtes.

Complexité de la fonction *kruskal*

Complexité Temporelle : O(V + E log(E))

Nous n'avons pas encore choisi d'orienter le graphe pour avoir un arbre avec une relation de parenté, la conversion sera de toute manière faisable en temps O(V) à l'issue de cette fonction (en parcourant le graphe en largeur). On construit d'abord une liste des arêtes sans répétition grace à un dictionnaire qui garde en mémoire les arêtes déjà rajoutées.
Chaque arête est vue deux fois, mais puisqu'elle est déjà dans le dictionaire la deuxième fois, elle n'est pas ajoutée à la liste.
Ce processus a un coût en opération élémentaire de l'ordre de $O(2 * E + E) = O(E)$.

Trier la liste sur power prend O(V + E log(E)) opérations élémentaires classiquement.

Ensuite on construit le graphe. On itère sur la liste des arêtes. La vérification "Si nodea et nodeb non connectées" s'effectue en temps constant. Ce test suffit car les composantes connexes sont disjointes, et deux éléments dans la même composante renvoient vers un même pointeur de liste. Les listes sont de plus toujours non-vides.

La section de code qui actualise pour les voisins de nodeb la nouvelle composante à laquelle ils appartiennent n'est pas constante pour chaque itération, mais grâce au test, quand il n'y a plus qu'une composante connexe, elle n'est plus effectuée, on déduit qu'au total l'execution de cette partie du code regroupée pour toutes les itérations sur *edges* ne va représenter que O(V) opérations élémentaires.

Complexité Spatiale : O(E+V)

On utilise successivement un dictionnaire de taille E, une liste de taille E, un tri en place (O(1)), tout ceci représente un espace mémoire de l'ordre de O(E). Enfin, on utilise un dictionaire *connected* de taille O(V) ; car il y a V clefs et la somme des longueurs des listes disctinces contenues dedans est également constante égale à V (invariant de boucle immédiat).

Test de la fonction krusal Pour tester l'algorithme implémenter, on considère des graphs de petite taille que sont "network.00" et "network.01", mais on teste aussi "network.10". On remarque alors que l'ensemble de ces 3 tests s'effectuent en seulement 1.471s ce qui, considérant que le dernier graph comporte 200 000 noeuds, est appréciable et vient valider qualitativement les résultats de complexité de la question précédente.

En principe, pour implémenter plus de test et sur des graphes plus importants, le plus simple serait de passer par une visualisation ainsi qu'une vérification que les chemins minimaux du graphe originel sont retrouvés dans l'arbre généré.

Complexité de la fonction *min_power_tree*

Complexité Temporelle : O(V)

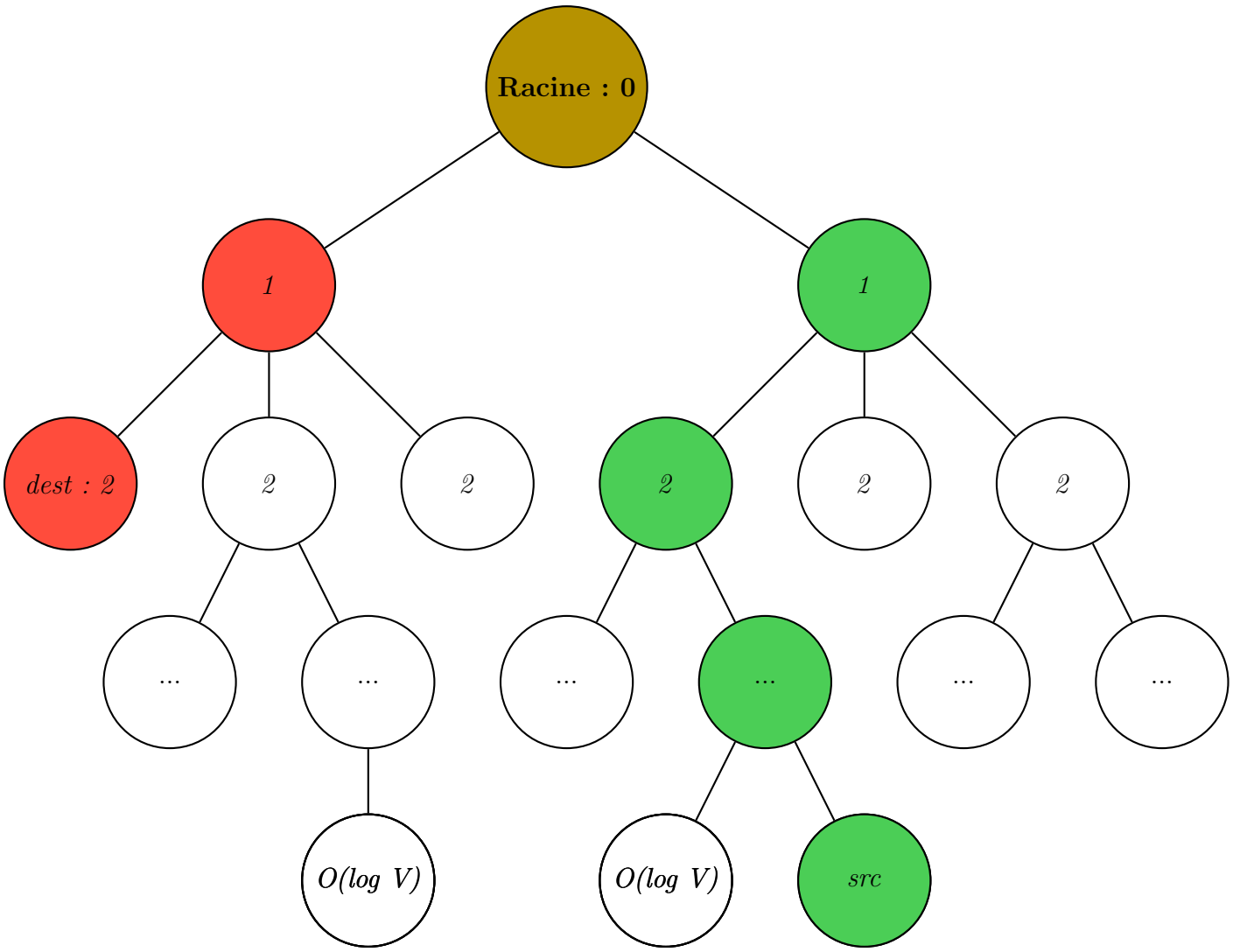
On considère que le graph passé en argument a déjà été converti en arbre couvrant minimal par la fonction kruskal.
La fonction *min_power_tree* est ainsi simplifiée.
On parcourt en profondeur récursivement le graphe, on garde en mémoire le pouvoir minimal du chemin en le passant dans les arguments.
Grace au dictionnaire *seen*, on ne passe qu'une fois par chaque noeuds, et donc par chaque arête.
Cela fait une complexité O(V).

Problème : Ce n'est pas assez bien pour de grands graphes avec beaucoup de routes !
Par exemple, pour V = 100 000 et 100 000 routes, cela fait un ordre de grandeur de 10 milliards d'opérations élémentaires.
Comme ce n'est qu'un ordre de grandeur et que cela peut être plusieurs fois cette quantité dans l'implémentation, en réalité on peut arriver facilement plusieurs dizaines de milliard d'opérations élémentaires.

Si l'on considère que Python peut effectuer environ 1 million d'opérations élémentaires par seconde approximativement, alors l'exécution prendrait une trentaine de minutes.
On retrouve ce résultat sur network.2 !

En réalité, la structure d'arbre permet de faire bien mieux.
Si l'arbre est bien équilibré, par un preprocessing des ancêtres, on peut arriver à une complexité O(log(V)), et alors l'exécution de *min_power_tree_optimised* prendrait un temps $T \times log(V)$. Pour les graphes proposés dans input, cela donnerait une implémentation de l'ordre de la minute, comme le suggère la séance 3.

Complexité de *tree_search*



Construction du chemin de src à dest dans l'arbre couvrant minima (cas idéal)

Pour éviter la complexité linéaire dans la situation du pire cas (e.g. un peigne), on devrait faire une étape de preprocessing des ancêtres d'ordre 2^i pour assurer une complexité $O(log(V))$ dans le pire des cas. Nous n'avons pas eu le temps de développer cette solution comme nous avions déjà les fichiers routes.i.out et avons décidé de passer à la suite.

Plusieurs heuristiques possibles

Bien qu'il y ait de nombreuses approches pour aborder le problème, certains sous-problèmes sont communs et reviennent dans chacune.

Il y a un premier constat : il n'est pas possible de traverser chaque route avec le budget alloué, l'intérêt du problème est de choisir les routes maximisant notre l'utilité totale. On le remarque en essayant une première approche :

Première approche : Brute force

On a jusqu'ici gardé le choix d'algorithmes parfaitement déterministes qui assurent de trouver **le** chemin minimal, **le** power minimal. On peut d'abord chercher à rester dans cette optique et tenter de trouver à coup sûr **le** maximum d'utilité, global donc.

Une manière évidente mais avec un coût extrême serait d'essayer toutes les combinaisons de routes, de calculer l'utilité pour chaque combinaison et de conserver le maximum. Pour N routes, il y a 2^N possibilités, cette solution est donc bien sûr irréalisable.

Une approche plus intelligente serait alors un recours à de la programmation dynamique et une heuristique "diviser pour régner" :
pour des budgets croissants, on calcule le choix optimal de routes et on en déduit le panier optimal pour les budgets suppérieurs à partir de ceux là.
Cependant même avec cette amélioration, la complexité reste $O(BT)$ où B est le budget et T le nombre de routes. Pour ce problème, $B = 25 \times 10^9$ et T oscille autour de 10^6 : cette complexité reste trop importante pour notre problème.

Deuxième approche : Algorithme glouton
Le principe est le suivant :

- On trie la liste des trajets en fonction de leur rentabilité (ici le ratio $\frac{utilité}{coût}$). À chaque route, il est clair que l'on peut associer un camion optimal. Pour le déterminer, on examine les camions et on supprime ceux qui sont inutiles i.e. ceux pour qui il existe un autre camion avec un power plus grand mais coûtant moins cher, cela prend un coût $O(ClogC)$, où C représente le nombre de camions. Ensuite en regardant le power_min de chaque route et par une recherche dans la liste triée des camions, on détermine en $O(Tlog(C))$ l'appariement route - coût. Ce pre-processing est en fait utile pour la plupart des solutions (potentiellement approchées) au problème.

- On cherche à ajouter les routes les plus rentables par ordre décroissant

Cependant, pour les problèmes avec un nombre de routes suffisamment grand par rapport au budget, la solution trouvée n'est pas optimale.
Il pourrait être plus avantageux de laisser une route plus "profitable" pour éviter de finir à la fin avec des routes qui le sont peu. Un entre deux, avec uniquement des routes moyennement

profitables pourrait être un meilleur choix. L'avantage de cette méthode est qu'elle est rapide à implémenter (de complexité quasi linéaire en T, quasi linéaire en C).
La méthode gloutonne peut néanmoins être une solution donnant un résultat plutôt satisfaisant compte tenant de l'effort fourni et sa solution est un bon point de départ pour des méthodes de *local search*.

Troisième approche : Heuristique de *recuit simulé* ou *simulated annealing*

Cette méthode n'assure pas de trouver un extremum global à coup sûr et est basée sur l'heuristique suivante :

- On fait arbitrairement un premier choix de routes respectant le budget (soit de manière aléatoire, soit de manière déterministe).

- On regarde ensuite une configuration voisine : on enlève ou rajoute une route, ou encore les deux opérations simultanées, de manière à toujours respecter le budget. Alors il y a deux situations selon la variation d'utilité ΔU :
  - si $\Delta U > 0$, on confirme le changement.

  - sinon on accepte le changement (négatif) avec probabilité

$$P_{accept}(T,\ \Delta U) = exp(\ \frac{\Delta U}{K\ T}\ )$$

    où
    T représente une température qui va décroitre géométriquement au cours de l'algorithme
    $K \geq 0$ est un hyperparamètre déterminé "à la main".

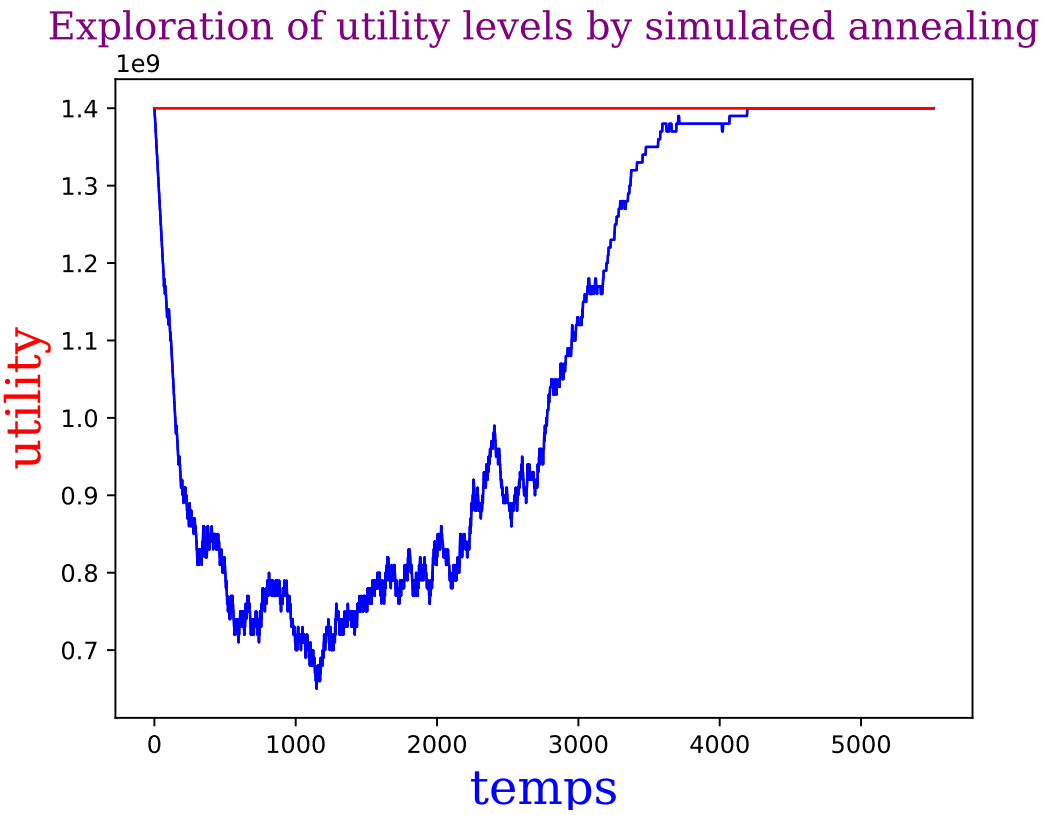
- on itère l'opération jusqu'à ce que $T \leq 10^{-k}$ où l'on choisit k un entier . Le choix de K et de k sont dépendants (cf formule P_{accept}).

La raison qui nous a poussé à choisir cet algorithme est qu'il permet une exploration et que l'on ne risque pas de s'enfermer dans un minimum local comme avec un algorithme glouton. Vers la fin de l'algorithme, une fois que T a décru assez, il est improbable de faire un pas contreproductif et l'état converge vers le minimum local d'un algorithme de glouton.

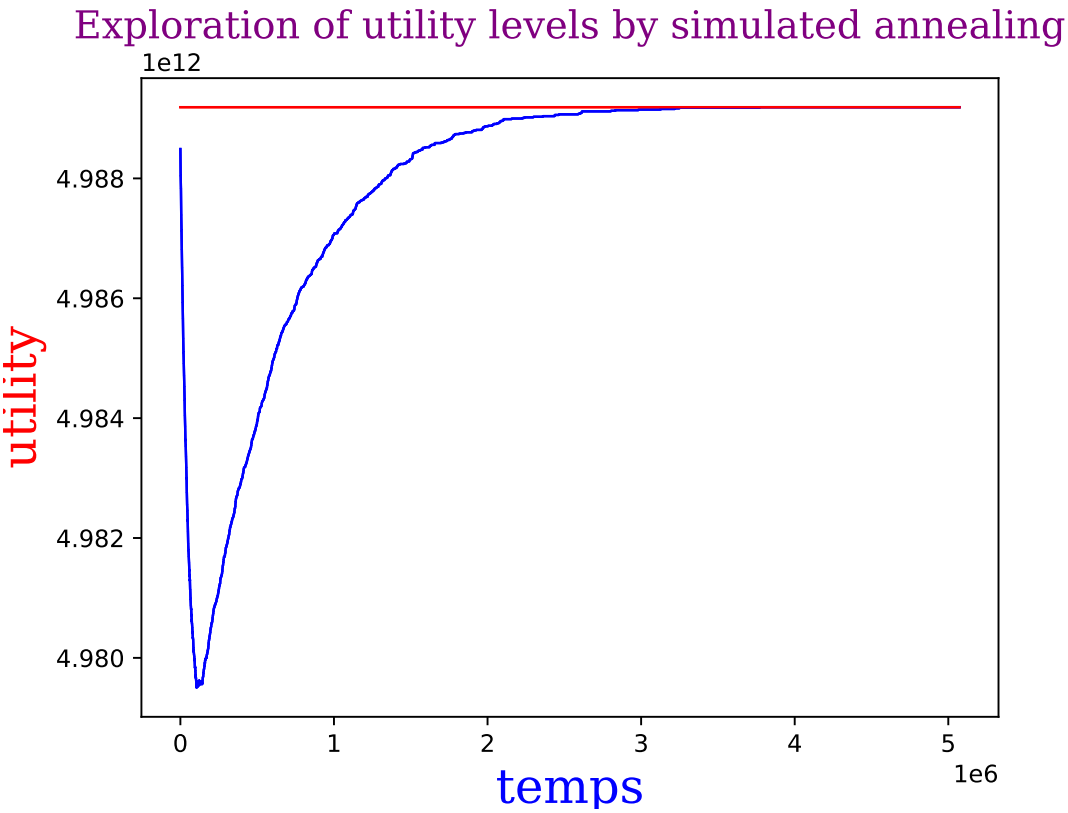
Les deux points qui sont cruciaux pour cet algorithme sont le choix du voisin ainsi que le choix des constantes.

En effet, un voisin doit en théorie être proche de la situation initiale afin d'avoir des variations plutôt controlées et permettre "d'explorer" l'espace.
Cette heuristique s'oppose à une heuristique de type Monte Carlo qui plutôt qu'explorer l'espace, cherche à l'échantilloner. Les hyperparamètres sont grandement dépendants de l'échelle d'utilité et des autres paramètres du problème. C'est un jeu d'ajustement avec la température jusqu'à l'obtention de résultats satisfaisants.

Cet algorithme est fait pour être répété plusieurs fois, et à itération un résultat différent apparait du fait de l'exploration aléatoire des configurations voisines.
On peut tracer les niveaux d'utilités rencontrés lors de l'exploration, une "trajectoire d'exploration", comme par exemple pour le fichier route 1, en partant de la solution d'un algorithme glouton (qui est optimale car le graphe est encore petit) :



Cette méthode produit sur le fichier route 2 une trajectoire du type, en deux secondes seulement, et qui bat l'algorithme glouton (la courbe rouge représente le maximum, le point de départ est celui de l'algorithme glouton) :



Chaque exécution est de l'ordre de la seconde, on peut alors décider prendre comme point de départ la meilleure route trouvée lors d'une recherche précédente et itérer (au lieu de repartir de l'algorithme glouton). On trouve alors une amélioration significative.
Pour le fichier route 4, :

$$\textit{1 itération : } log(U) = 29.238117$$
$$\textit{30 itérations : } log(U) = 29,238314$$

Soit une différence d'utilité de presque 1 milliard ! Les augmentations deviennent de moins en moins significatives, mais il n'est pas impossible que "par chance" de temps à autres, il y ait de gros bons dans les utilités maximales.