Optimisation Multicritères:
Voyageurs de commerce bi-critères: Coûts/Durée

Valentin Hesters 20201346

19 Novembre 2024

Sommaire

le dossier du code se trouve à cette adresse :
https://github.com/suprawall/MTSP

Pour lancer le code il s'uffit d'executer *main.py*
la variable globale NB_BLOCKS gère la taille du graph.

1 Implémentation

1.1 Génération du graphe

Un seul type de graphe est considéré dans cette implémentation: le graphe en blocs. Le code source se trouve dans le fichier *graph_init.py*. Le graphe se compose d'un bloc initiale de 4 noeuds reliés par 6 arrêtes. Les blocs suivants sont ajoutés à la "droite" de celui-ci, ils se composent de 2 noeuds et 4 arrêtes. La taille du graphe généré est définie par la variable globale NB_BLOCKS dans le fichier *main.py*. Ainsi, chaque graphe considéré dispose de:
$4 + 2 * (NB\_BLOCKS - 1)$ noeuds, et $6 + 4 * (NB\_BLOCKS - 1)$ arcs.
Le chemin a minimiser est celui entre le premier noeud et l'avant dernier.

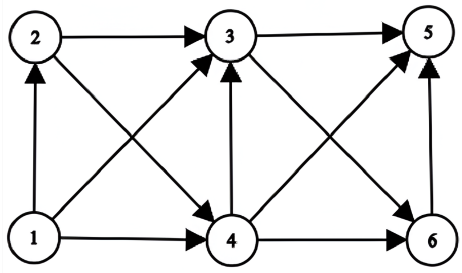

Figure 1: *graphe en block: chemin entre 1 et 5*

les poids sont ajoutés à chaque arcs sous forme de tuple (coût, durée). L'intervalle de valeurs pour chacun des critères est définie par deux variable global dans main.py. Dans la partie expérimentale de ce rapport, chaque coût et chaque durée est compris entre 1 et 20. Pour avoir une distinction nette entre des chemins efficaces pour minimiser le coût et d'autres pour minimiser la durée, les poids sont répartis comme ceci: pour chaque arcs on choisit au hasard entre le coût et la durée. Si le coût a été choisi alors il aura une valeur aléatoire entre 1 et le plafond (ici 20), et la durée aura une valeur aléatoire entre 1 et le plafond/3 (ici 20/3 arrondi à 6). Ainsi la durée a plus de chance d'être bien inférieur au coût, mais la probabilité n'est pas de 0 non plus, ce qui assure une distribution plus au moins naturelle dans le graphe tout en étant assuré d'avoir une distinction entre les solutions qui minimise le coût et les solutions qui minimise la durée sur des grands graphes.

1.2 pseudo-code

Le code pour effectuer l'algorithme epsilon contrainte via la relaxation lagrangienne se trouve dans le fichier *execute_lagrangienne.py*. Dans un premier, on cherche à trouver la solution qui minimise le coût et celle qui minimise la durée. Pour ce faire on utilise le solveur python Pupl `https://coin-or.github.io/pulp/`. La fonction: "def solveur(G, weights, critere):" du fichier *execute_brutforce.py* permet de spécifier quel critère on souhaite minimiser (0 pour le coût et 1 pour la durée). On associe à chaque arcs une variable binaire pour savoir si l'arc est gardé dans la solution. On l'a multiplie avec le poid de cet arc sur le critère concerné. Ensuite on définie le faite que: le noeud source doit avoir un flux sortant de 1, le noeud cible un flux entrant de 1 et que chaque noeud doit avoir un flux entrant et sortant égal.
Ensuite on récupère le couple de poid associé à la solution via la fonction "get_weight_path" à la ligne 23. La premiere contrainte est donc la durée de la solution qui minimise le coût. A partir de là on peut lancer l'algorithme.

le code principal se trouve dans la fonction "execute_lagrangienne" à la ligne 114 de *execute_lagrangienne.py*. Il se compose de 2 boucles: une pour epsilon contrainte et une pour la relaxation lagrangienne.

1. On initialise un tableau de parametre contenant des tuples: (coût, durée - contrainte epsilon), pour toutes les solutions de la frontière de Pareto.
2. On calcule les fonctions de lagrange de chaqu'un des élements dans le tableau paramètre
3. On trouve le λ qui maximise les minimums des fonctions dans le tableau paramètre.
4. On calcule les nouveaux poids pour chaque arcs (coût + λ * durée), et on cherche la solution optimale à ce problème mono-critère via le solveur Pulp.
5. On l'ajoute au tableau paramètre et on boucle depuis l'étape 2 jusqu'à convergence
6. Une fois sortis de cette boucle on prend la dernière solution on l'ajoute à la frontiere de Pareto et on définie la prochaine contrainte epsilon comme étant la durée de cette solution
7. On boucle depuis l'étape 1 jusqu'à ce que la durée de la solution trouvée soit égale à la durée de la solution qui minimise la durée

Le dossier se compose aussi d'un fichier *execute_brutforce.py*, qui permet d'executer la méthode epsilon-contrainte de manière brut force, c'est à dire qu'à chaque itération on trouve la nouvelle solution via le solveur Pulp directement. On utilise cette fois la fonction "solveur_containte" de la ligne 73 pour chercher le chemin qui minimise le coût tout en ayant une contrainte sur la durée. J'ai implémenté cette méthode, qui donne donc un ensemble complet Pareto optimale, pour pouvoir la comparer avec les résultats obtenus via l'algorithme.

2 Résultats expérimentaux

Le graphique le plus intéressant est celui qui compare les frontières de Pareto obtenue via l'algorithme lagrangien et via brutforce. Voici en exemple sur un petit graphe de 30 blocs, pour que tous les points du brutforce soit encore distinguables.

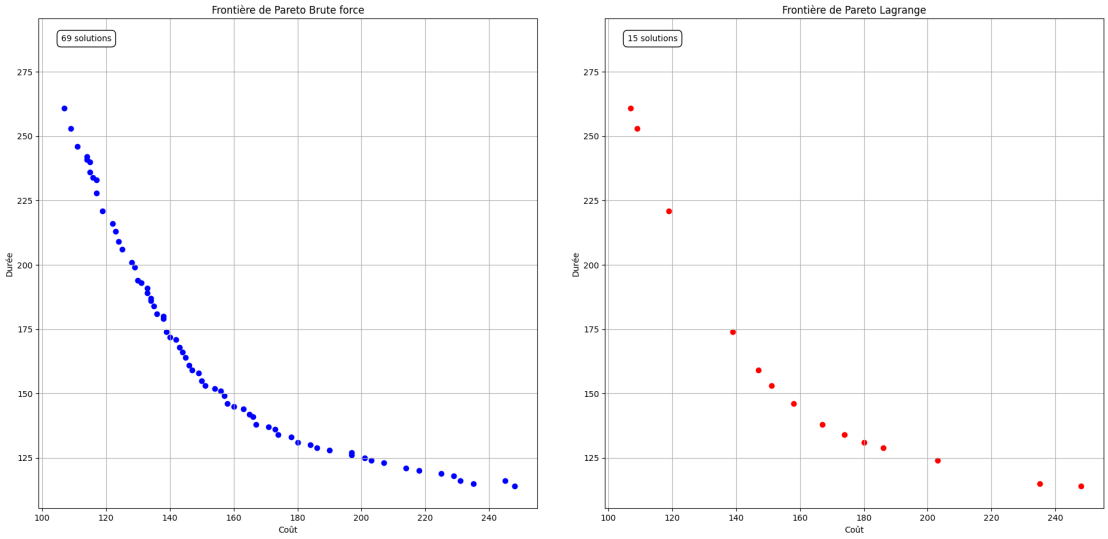

Figure 2: *comparaison sur 30 blocs*

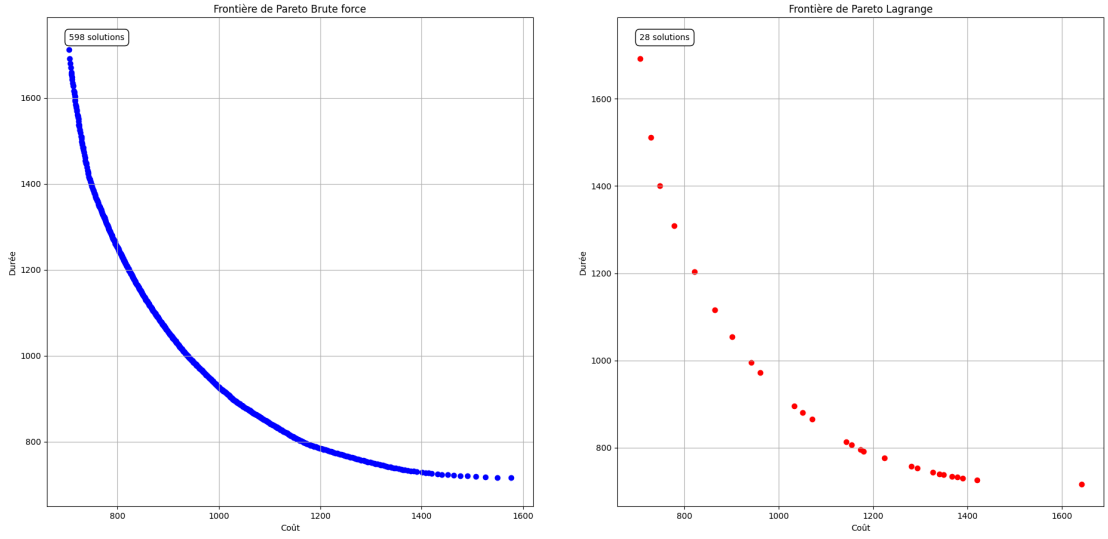

Figure 3: *comparaison sur 200 blocs*

On remarque que les solutions obtenues via l'algorithme ont tendance à être plus concentré vers la fin de la courbe plutôt qu'au début.

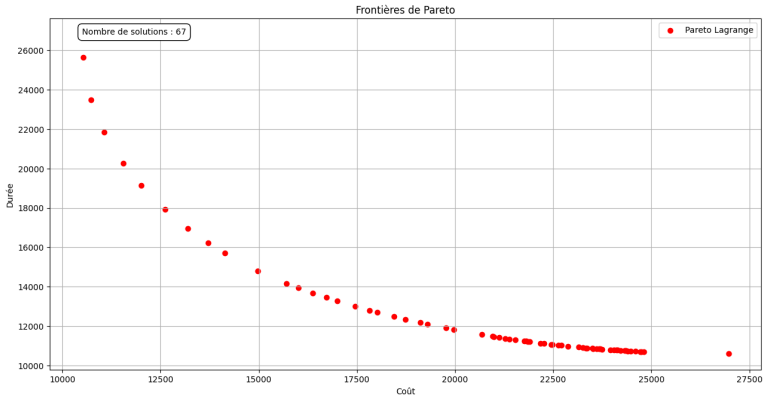

Figure 4: *3000 blocs: 12 002 arcs*

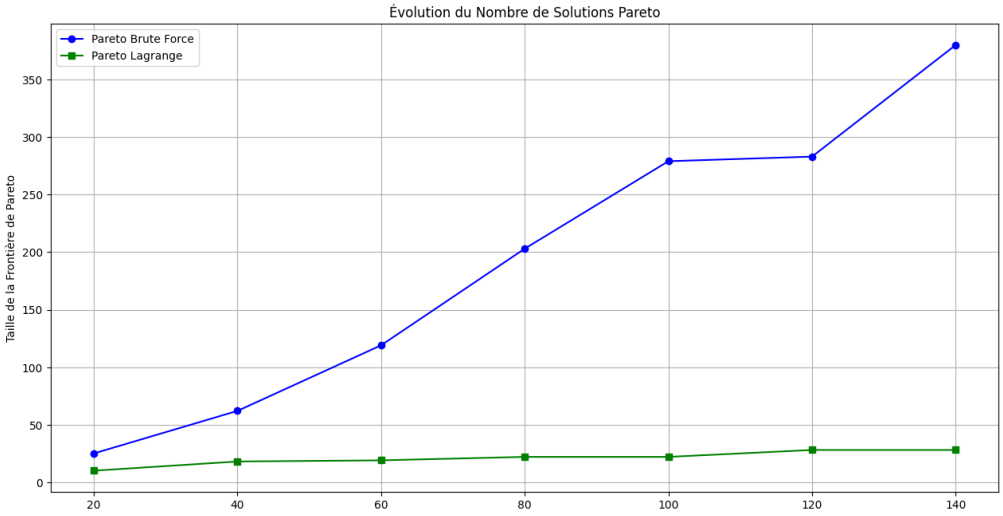

Figure 5: *comparaison nombre de solutions*