POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Devoir 2 - Alimentation de fours à pizza

2015144 Saikali, David
2226903 Djian, Benjamin
Département de Génie Informatique Génie Logiciel

Trimestre Hiver 2024

NB : Ce document reprend les notations définit dans l'énoncé du devoir.

1 Construction de la solution initiale

Pour construire notre solution initiale, on utilise un algorithme glouton à plusieurs étapes. Cet algorithme prend en paramètre une liste de livraison que l'on souhaite pour chaque cycle. On commence l'algorithme en ordonnant les pizzerias selon le nombre de cycle restant avant que leur niveau de charbon descendent plus bas que le minimum. Ceci nous permet de toujours livrer aux cas les plus urgents. À cette liste de livraisons nécessaires, on ajoute les livraisons forcées qu'on a choisi d'avance. Une livraison nécessaire est une livraison qui tombera sous son seuil minimal lors de ce cycle. L'opération la plus longue de cette partie est de trier les pizzerias ce qui se fait en $\mathcal{O}(n \cdot log(n))$ où n est le nombre de pizzerias.

Une fois que l'on a les livraisons que l'on veut faire, on les sépare entre les différents camions. Pour ce faire, on commence par trier les camions selon leur distance à la pizzeria que l'on essaye de placer. Pour calculer la distance d'un camion à une pizzeria, on utilise le dernier arrêt du camion ou la mine s'il y est encore. On itère ensuite sur les camions en ordre décroissant et on assigne la livraison au premier camion qui peut le faire sans excéder sa limite. Pour s'assurer d'avoir un solution valide, on commence par insérer les livraisons nécessaires et on insère ensuite les livraisons forcées. L'opération la plus couteuse est encore une fois un triage, mais des camions cette fois-ci. Par contre, puisqu'on doit faire ce tri pour chaque livraison, la complexité de cette partie est de $\mathcal{O}(n \cdot m \cdot log(m))$ en pire cas où m est le nombre de camions. En effet, en pire cas, il faut faire une livraison à toutes les pizzerias, et pour chacune il faut trier les camions.

La dernière étape de notre construction de solution est d'ajuster la quantité de charbon des livraisons. Les livraisons nécessaires commence avec un volume équivalent au minimum requis et les livraisons forcées commencent avec un minimum arbitraire que l'on a choisi ce qui empêche de les ignorer. L'idée est d'ajuster le volume qu'on livre pour respecter les différentes contraintes, mais aussi de livrer le plus que l'on peut aux différentes pizzerias de manière intelligente. On commence par trier les pizzerias qui ne seront pas pleines après notre livraison par leur distance à la mine. On veut donner le plus de charbon aux pizzerias éloignées pour éviter d'y retourner souvent. Une fois qu'on a cette liste triée, on la parcours en ajoutant à la livraison ce que consomme la pizzeria en un cycle. On limite toujours les livraisons pour ne pas excéder la limite de la pizzeria ni la capacité du camion. On continue d'itérer sur les pizzerias non-pleines tant qu'il y a de l'espace dans le camion et qu'on a des pizzerias non-pleines (on les enlève au fur et à mesure qu'on itère). La complexité temporelle de cette partie est plus difficile à déterminer, mais on montre plus bas ce qu'on a fait:

$$\mathcal{O}(m \cdot (n + n \cdot log(n) + n \cdot (empty\_space/cycle\_consumption))) \tag{1}$$

$$\mathcal{O}(m \cdot n \cdot (1 + log(n) + (empty\_space/cycle\_consumption))) \tag{2}$$

$$\mathcal{O}(m \cdot n \cdot (log(n) + (empty\_space/cycle\_consumption))) \tag{3}$$

Dans le terme final, *empty_space* est l'espace restant de le camion lorsque l'on commence à itérer et *cycle_consumption* est la somme de ce que chaque pizzeria dans la liste consomme en un cycle.

Pour notre implémentation, une solution pour une instance donnée correspond à des livraisons forcées et on construit une solution valide à partir de ces livraisons. Ceci nous permet de faire une recherche locale en changeant les livraisons forcées.

2 Recherche locale

Notre recherche local commence en construisant une solution à partir d'une liste de livraisons forcées. Ensuite, on fait du "simulated annealing", en acceptant des solutions dégradantes pour essayer d'en retrouver une meilleure plus tard. Pour éviter de passer trop de temps à dégrader une solution sans rien trouver, si on ne trouve pas une meilleure solution après 10 itérations dégradantes, on fait une relance aléatoire à partir d'une liste de livraisons forcées aléatoire. Ceci nous donne assez de diversification pour explorer un espace intéressant du problème. Notre algorithme glouton de construction nous donne une très bonne intensification. On commence toujours avec une liste de livraisons forcées vide.

Notre voisinnage est assez simple, on enlève une livraison parmi celles que l'on force ou on en ajoute une. On génère tous les voisins et on les mélange pour éviter de passer dans le même ordre à chaque fois. Une fois qu'on a nos voisins, on accepte le premier qui amméliore la solution ou une solution qui l'empire avec une probabilité qui décroit plus la solution est pire. On utilise la formule vue en classe

pour le seuil d'acceptation. Ce voisinage est connecté comme le montre l'algorithme plus bas et est de taille $T \cdot n$ puisque pour chaque cycle on peut forcer une livraison à une pizzeria.

```
solution: list[set[int]] = random_forced_deliveries_per_day()
bestSolution: list[set[int]] = optimal_forced_deliveries_per_day()

for int t in range(T) {
    # Remove all the wrong forced deliveries
    for int id in solution[t] {
        if not id in bestSolution[t] {
            solution[t].remove(id)
        }
    }
    # Add all the missing ones
    for int id in bestSolution[t] {
        if not id in solution[t] {
            solution[t].add(id)
        }
    }
}
```

Notre algorithme de construction assure que les solutions générées seront toujours valides.

Pour évaluer la qualité d'un voisin, on ne fait que comparer le coût final de la solution et, comme mentionné plus haut, on garde le premier qui améliore ou le premier qui détériore avec une probabilité basée sur la qualité de la solution.

Le "simulated annealing" et les relances aléatoires nous permettent d'éviter les minimas locaux et on s'approche généralement d'une bonne solution. Comme le dernier devoir, on utilise l'intuition humaine pour guider la recherche (intensification) et des techniques de diversification pour éviter de rester pris lorsque cette intuition se révèle à être non-optimale.

3 Résultats

Pour chaque instance on précise le temps d'exécution maximal autorisé (en secondes) pour la résolution dans le code.

Pour chaque instance (A,B,C,D,E) le score de la meilleure solution trouvée dans le temps imparti se trouve dans le tableau suivant :

Instances	A	B	C	D	E
Temps de recherche	5min	5min	5min	10min	10min
Coût minimal trouvé	**1641.47**	**2480.17**	**2072.88**	**3568.61**	**4848.48**

Toutes les instances se situent dans les bornes de l'énoncé, et les plus petites atteignent même la borne inférieur (très proche). Si on enlevait la partie de notre algorithme de construction qui balançait les volumes de livraisons, on trouve la borne inférieur pour le A, le B et le C. Par contre, le coût pour les solutions D et E augmente de beaucoup.

Des limitations à l'implémentation sont décrites dans la partie suivante. Elle propose des pistes d'explication de pourquoi l'implémentation n'atteint pas l'optimal pour toutes les instances.

4 Limitations

La première grosse limitation de notre implémentation est l'algorithme de construction qui repose sur des algorithmes gloutons basés sur des heuristiques intuitives, mais qui ne sont pas nécessairement optimales. On ne peut pas assurer qu'on construit la meilleure solution possible à partir de ces livraisons forcées, mais les résultats sont bons. Dans un monde parfait où on avait plus de temps, on devrait faire plus de recherche pour voir comment améliorer cette construction.

La seconde grosse limitation est notre représentation du problème. En limitant notre représentation à des livraisons forcées, on peut rapidement explorer l'espace et générer des voisins, mais il est possible que ce voisinnage manque de précision, laissant trop à la discrétion de l'algorithme de construction (les deux problèmes sont liés l'un à l'autre).

Avec plus de temps, on aurait aimé changer cette représentation et ajuster l'algorithme de construction pour étudier l'effet sur les résultats.