Projet : Un problème de tournée de drones dans un contexte de déconfinement

PREKA Bruno, ZELLE Lars Yannick (681C)

31 mars 2021

1 Introduction

Dans le contexte sanitaire actuel, un scénario fictif suggère l'isolement *total* des personnes atteintes de la Covid-19 dans leur domicile. Des drones sont alors réquisitionnés pour livrer des courses à domicile.

Dans chaque zone géographique, un unique dépôt (qu'on notera 1 par la suite) stocke des ressources alimentaires et médicamenteuses. Plusieurs contraintes se dessinent dans ce problème appartenant à la classe des problèmes de tournées de véhicules. La contrainte garantissant que chaque client est visité une seule fois, ainsi que celle de la charge supportée par un drone, y interviennent. Le but du problème est de minimiser la distance parcourue par l'ensemble des drones.

On résout ce problème à travers deux méthodes possibles :

1. Dite *exacte*, la méthode assimile le problème à un problème d'optimisation combinatoire, mais qui peut demander un temps d'exécution très important aux yeux des décideurs ;
2. *Approchée*, celle-ci est une variante de la méthode de Clark et Wright.

1.1 Structures de données générales

1.1.1 Données du problème

A partir de la fonction `lecture_donnees`, on traduit les instances numériques fournies par la structure de données `donnees` dont les champs sont dédiés au nombre de clients `nbClients` (y compris le dépôt), à la `capacité` du drone, à la `demande` des clients et au distancier `distance`. On retrouvera régulièrement les types suivants :

- `nbClients` $=: n$ et `capacite` sont des entiers (on utilisera le type `Int64`)
- `demande` est un vecteur d'entiers `Vector{Int64}` : la demande du client i est renseigné dans la case `demande[`i`]`, ce qui facilitera son accès.
- `distance` est une matrice à coefficients entiers `Matrix{Int64}`. Ce type est équivalent à `Array{Int64,2}`, ce qui sera utile pour manipuler des sous-matrices (restrictions de matrice).

1.1.2 Ensemble des regroupements possibles : `S` ou $\mathcal{S}$

On rappelle que l'ensemble des regroupements possibles est défini par :

$$\mathcal{S} := \left\{ S \subseteq [\![2, n]\!] : \sum_{j \in S} d_j \leq capacite \right\}$$

Il est clair que la manipulation d'ensembles (type `Set`) est simple, mais elle demande des temps d'exécution plus importants. Il est donc plus que nécessaire de recourir à un "vecteur de vecteurs" au lieu d'un "ensemble d'ensembles", d'où le type `Vector{Vector{Int64}}` pour l'ensemble des regroupements.

1.1.3 Ensemble d'ensembles d'indices de tournées visitant le i-ème client :
`AllS_i` $= \{S_i \subseteq [\![1, |\mathcal{S}|]\!] \;:\; \forall j \in S_i,\; i \in tournee_j\}_{i \in [\![2,n]\!]}$

La contrainte garantissant que chaque client soit visité une seule fois est donnée par :

$$\sum_{j \in S_i} x_j = 1, i \in [\![2, n]\!],$$

où $S_i \subseteq [\![1, |\mathcal{S}|]\!]$ est le sous-ensemble d'indices des tournées contenant le client i. Pour les mêmes raisons que ce qui précède, on opte pour un vecteur de vecteurs d'entiers, au lieu d'un ensemble d'ensembles d'entiers, d'où le type `Vector{Vector{Int64}}`.

Pour un client i, l'accès à l'ensemble des indices des tournées le desservant se fait par `AllS_i[i]` $= S_i$.

1.1.4 Association tournée-distance minimale :
`l` $= \{(tournee_j, l_j) \;:\; \forall j \in [\![1, |\mathcal{S}|]\!],\; tournee_j \in \mathcal{S} \;\wedge\; l_j = \min_{\mathrm{TSP}} \mathrm{dist}(tournee_j)\}$

La fonction objectif (à minimiser) s'écrit :

$$z = \sum_{j=1}^{|\mathcal{S}|} l_j x_j,$$

où l_j désigne la distance minimale du regroupement $j \in [\![1, |\mathcal{S}|]\!]$, calculée par la fonction `solveTSPExact` de résolution du problème de voyageur de commerce (dit TSP, pour *Travelling Salesman Problem*). Ainsi, une telle expression requiert de lier cette distance à son regroupement via un accès immédiat. C'est pourquoi, le vecteur `l`, constitué de couples $(\mathit{tournée}, \mathit{distance}_{\min})$, intervient. Il est de type `Vector{Tuple{Vector{Int64},Int64}}`.

Enfin, en termes d'accès, on a : $\forall j \in [\![1, |\mathcal{S}|]\!]$, $l_j =$ `l[j][2]`.

2 Résolution exacte

2.1 Fonction d'énumération des regroupements possibles des clients

On appelle cette fonction `getSubsets_recursive(P,S,capacite,demande,index,d,distances)`, où :

- `P` et `toadd` sont les vecteurs des indices à ajouter pour compléter un regroupement, `toadd` étant une copie de `P` ;
- `S` est l'ensemble des regroupements ;
- `capacite` désigne la capacité du drone ;
- `demande` est le vecteur de la demande des clients ;
- `index` est un curseur ;
- `d` est l'accumulateur courant de la demande pour respecter la condition de $\mathcal{S}$;
- `distances` est le distancier.

Derrière cette fonction, l'algorithme consiste à itérer (via `index`) sur toutes les demandes des clients et à vérifier si la demande du client $i \in [\![2, n]\!]$, accumulée à `d`, est conforme à la capacité. Ainsi, pour le client i :

1. Si les demandes sommées sont en-dessous du seuil, on sauvegarde l'ensemble `P` des clients à ajouter dans `toadd` et on ajoute le client suivant $i + 1$ (à ajouter) à `toadd`. Sinon, rien n'est fait et on passe à l'étape 4.
2. On construit l'ensemble `S` $= \mathcal{S}$ en concaténant/réunissant `S` et `[toadd]`.
3. On répète le processus sur `S` et le client suivant $i + 1$ et on compare le nouvel ensemble `Snew` de regroupements : si `Snew` accueille de nouveaux regroupements, alors on les ajoute à `S`.

4. Une fois ce processus récursif terminé sur ce sous-groupe de clients $\{i, i+1, ..., n\}$, on recommence pour le sous-groupe $\{i+1, ..., n\}$.

Par conséquent, il en résulte l'ensemble des regroupements attendu, à savoir $\mathcal{S} =$ `S`.

La fonction `getSubsets(capacite,demande,distances)` vient encapsuler la fonction récursive par l'appel `getSubsets_recursive([],[],capacite,demande,1,0,distances)`, avec `index` $= 1$ et la distance `d` initialisée à 0.

2.2 Fonction déterminant la tournée et sa distance minimale dans un regroupement `Si`

On appelle cette fonction `determineShortestCycle(Si,d)`, où :

- `Si` est un regroupement de clients à visiter (de type `Vector{Int64}`) ;
- `d` est le distancier (`Matrix{Int64}`).

Puisque `Si` $\subseteq [\![2, n]\!]$, il n'inclut pas le dépôt 1. Ainsi, la réunion avec $\{1\}$ est primordiale pour passer par la fonction `solveTSPExact(d)` de résolution du TSP.

De plus, il faut restreindre `d` au regroupement donné, d'où une sous-matrice `newd = d[Si,Si]`. Une telle opération est permise par l'équivalence entre les types `Matrix{.}` et `Array{.,2}`, comme évoqué précédemment.

`solveTSPExact` retourne alors un couple ($tourn\acute{e}e, dist_{\min}$). Néanmoins, elle ne raisonne pas sur l'ensemble d'indices `Si`, mais bien sur $[\![1, |$`Si`$|]\!]$.

Un réajustement des indices doit conserver la cohérence des indices dans les tournées. Pour ce faire, on sait que le premier indice traité par `solveTSPExact` correspond au premier de `Si`, idem pour le deuxième, ainsi de suite... Puis, dans l'ordre, on a les correspondances suivantes :

- le premier client k_1 de *tournée* $\longrightarrow$ client d'indice k_1 dans `Si` ;
- ...
- le i-ème client k_i de *tournée* $\longrightarrow$ client numéro `Si[`k_i`]`
- ...

Ces correspondances donnent lieu à la "nouvelle" tournée, nommée `newtournee`, qui est retournée.

3 Résolution approchée (brève)

4 Analyse des résultats issus des instances numériques (A et B)

Les tests ont été réalisés sur une configuration Ubuntu, épaulée d'un processeur double-coeur (et 4 threads) Intel Core i3 cadencé à 2,3 GHz et de 12 GB de RAM.

4.1 Résolution exacte

A l'aide de la macro `@time` de Julia, on a mesuré les temps d'exécution (en secondes) des fonctions principales et les allocations mémoires. Par ailleurs, on a également récupéré le nombre de regroupements pour chaque instance et les tailles minimale et maximale des regroupements. Ces résultats sont donnés par les tableaux suivants :

4.1.1 Instances A

Instance A									
Taille de l'instance (nbClients)	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**
Nombre de regroupements |S|	**79**	**240**	**694**	**2948**	**9126**	**4195**	**17626**	**26984**	**44145**
Taille du plus petit regroupement	1	1	1	1	1	1	1	1	1
Taille du plus grand regroupement	**3**	**4**	**4**	**5**	**6**	**4**	**6**	**6**	**5**
Temps CPU pour l'algo. de partitionneme	0,000136	0,000359	0,004501	0,036589	0,162674	0,07205	0,374912	0,740604	2,017109
Temps CPU pour TSP	0,019212	0,066306	0,31166	1,335688	5,010185	1,819831	9,526452	14,621288	23,656863
Temps CPU pour la résolution du PL	0,001653	0,002669	0,006005	0,04848	1,101997	1,531643	0,808347	5,49762	7,501472
Temps CPU total (A)	**0,02335**	**0,092535**	**0,348139**	**1,432477**	**6,326426**	**3,441912**	**10,828685**	**21,028694**	**33,373689**
Allocation mémoire totale	5,026 MiB	19,57 MiB	74,926 MiB	431,179 MiB	1,728 GiB	594,72 MiB	3,965 GiB	7,045 GiB	15,384 GiB

On note tout d'abord que, pour `nbClients` $= 50$, on a l'ordre de grandeur du temps total CPU attendu : ≈ 30 secondes.

Globalement, la fonction de résolution du problème TSP prend le plus de temps à s'exécuter, devant les autres fonctions principales.

En outre, l'allocation mémoire totale qui est très importante, au détriment du temps d'exécution. Cette allocation pourrait être améliorée par un code plus optimisé, une modification du typage (limiter la taille d'encodage d'un entier par exemple), et une limitation des déclarations de variables. Ceci étant dit, il ne faut pas oublier que le programme linéaire (qu'on écrira "PL" par la suite) donne lieu à autant de variables x_j qu'il y a de tournées/regroupements.

4.1.2 Instances B

Instance B						
Taille de l'instance (nbClients)	**10**	**15**	**20**	**25**	**30**	**35**
Nombre de regroupements |S|	**288**	**4509**	**42299**	**62714**	**270307**	***trop long***
Taille du plus petit regroupement	1	1	1	1	1	
Taille du plus grand regroupement	5	7	8	7	9	
Temps CPU pour l'algo. de partitionneme	0,000602	0,03535	2,273813	4,752531	116,453366	
Temps CPU pour TSP	0,170264	2,847467	32,525961	43,311494	217,150977	
Temps CPU pour la résolution du PL	0,005357	0,087734	124,795803	31,052658	119,745037	
Temps CPU total (B)	**0,179404**	**2,989932**	**159,743062**	**79,457142**	**454,479656**	
Allocation mémoire totale	38,806 MiB	927,895 MiB	17,952 GiB	31,013 GiB	397,826 GiB	

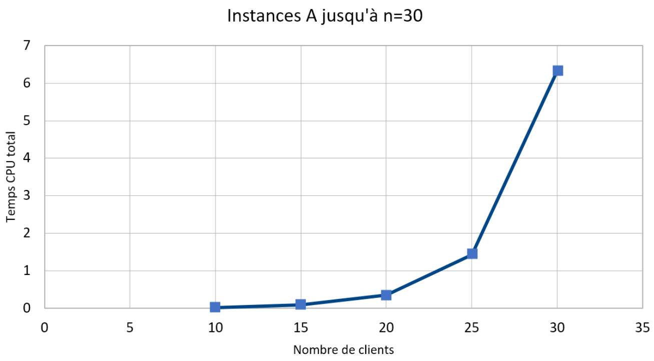
L'exécution sur les instances B prend bien plus de temps. En effet, la capacité y est légèrement supérieure ($capa_A = 30 < capa_B = 33$), et les demandes individuelles ont globalement légèrement diminué. Par exemple, dans `VRPA15.dat` (instance A avec `nbClients` $= 15$), on enregistre une demande moyenne de $9,93$, tandis que dans son homologue B, celle-ci atteint $5,4$.

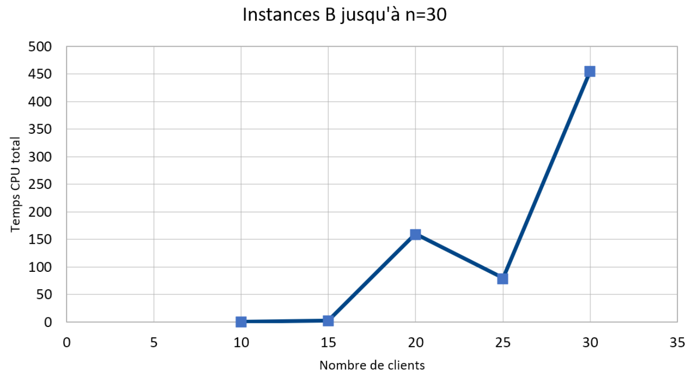
Une telle modification des paramètres implique une accumulation de demandes $\sum_i d_i$ qui s'allonge derrière une capacité plus élevée. D'où davantage de regroupements possibles et de chances d'avoir plus de clients dans un regroupement... C'est pourquoi, la taille maximale d'un regroupement augmente aussi. Elle croit plus vite pour les instances B que pour les instances A.

Enfin, la fonction de partionnement met davantage de temps à s'exécuter : il suffit de regarder la colonne pour `nbClients` $= 30$ pour s'en rendre compte... Une telle augmentation de la taille des données (nombre de regroupements) suggère des traitements plus conséquents dans la fonction `solveTSPExact` ! Qui dit davantage de tournées, dit davantage de variables x_j ! Donc la résolution du PL s'effectue également en un temps bien plus conséquent.

4.1.3 Analyse graphique et rapports de comparaison

Puisque la résolution exacte sur les instances de B de taille (au sens de `nbClients`) supérieure à 30 n'aboutit pas en un temps raisonnable, on se concentre sur les instances A et B de taille au plus 30. Comparons face-à-face les graphiques des temps totaux CPU pour les deux types d'instances :





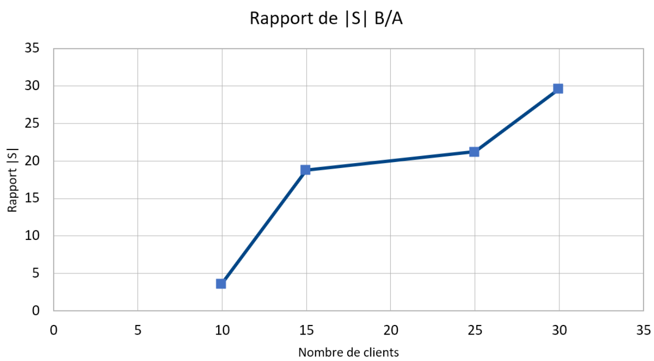
Le point le plus flagrant à souligner est la différence d'échelle de temps entre les deux graphiques. Celle-ci atteste des temps totaux CPU bien plus importants enregistrés pour les instances B.

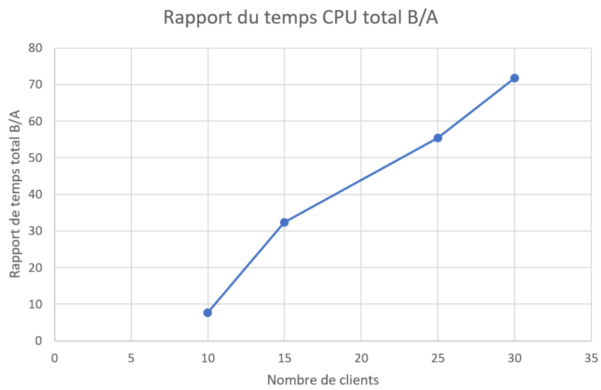
De plus, le temps total CPU pour les instances A semble croître exponentiellement, ce qui n'est pas forcément le cas pour les instances B. En effet, sur le graphique de droite, on note un pic parasite à $\texttt{nbClients} = 20$ ainsi qu'une monotonie et une vitesse de croissance très irrégulières (les pics $\texttt{nbClients} \in \{20, 30\}$ sont très soudains et les suivants doivent être pires). Des arguments précédents sur l'analyse des instances B viennent confirmer une évolution du temps CPU aussi singulière.

Enfin, un tableau des rapports des temps totaux CPU et des tailles $|\mathcal{S}|$ des regroupements entre les instances B et A est donné ci-dessous :

Taille de l'instance (nbClients)	10	15	20	25	30
Rapport de temps total B/A	7,68325482	32,3113633	458,848512	55,4683545	71,8382948
Rapport de |S| B/A	3,64556962	18,7875	60,9495677	21,2734057	29,619439

ainsi que les graphiques témoignant de la croissance de ces rapports (le point parasite $\texttt{nbClients} = 20$ est exclu) :





Selon les données de l'instance, les regroupements peuvent être complètement différents, ce qui explique l'allure peu pertinente du graphique de gauche.

Cependant, à droite, le rapport du temps CPU est quasiment proportionnel (si $\texttt{nbClients} = 10$ est l'origine) au nombre de clients. Ainsi, malgré tout, on peut se permettre d'approcher ce constat par :
$\exists \alpha \in \mathbb{R}_+^* : \forall n = \texttt{nbClients} \in \mathbb{N}$,

$$\frac{t_{\text{total},B}(n)}{t_{\text{total},A}(n)} \approx \alpha \cdot n,$$

ce qui témoigne d'une évolution cohérente et raisonnable du temps total CPU en passant d'un type d'instance à un autre.

4.2 Résolution approchée

4.3 Conclusion

De légères modifications des données (demandes et capacité) entre les instances suffisent à faire gonfler le nombre de regroupements possibles, à étendre les tournées, et ainsi à influer sévèrement les temps CPU, notamment dans la résolution du TSP.

Il est clair que, pour répondre en un temps court aux décideurs, la résolution approchée semble plus appropriée pour certaines instances telles que celles de type B. Voyons dans quelle mesure cette méthode est qualitative et pertinente.

5 Qualité de la solution admissible obtenue par la méthode *approchée*

6 Conclusion