MAHIER Loïc
JEHANNO Clément

groupe 601B

Rapport de projet [*]: Recherche opérationnelle

[*]rapport réalisé sous LaTeX

Sommaire

1 Introduction **3**

2 Partie algorithmique **3**
2.1 Algorithme d'énumération des regroupements possible 3
2.2 Algorithme d'énumération des tournées 5

3 Structures de données et bibliothèques du c++ **7**

4 Analyse des résultats **8**

5 Améliorations **9**

6 Conclusion **10**

1. Introduction

Trumpland étant ravagée il est de notre devoir de survivre face à l'adversité du milieu. Suite à l'écatombe deux pseudos informaticiens tirés de leurs études avant même de commencer leur masters doivent mettre en place une stratégie optimale pour récupérér de l'eau afin de vivre. Le problème est le suivant : un drone possédant une capacité d'eau maximale doit aller visiter plusieurs lieux de pompage et ramener toute l'eau en effectuant un chemin minimum.

Dans le cadre de notre projet nous avons procédés de deux manières : premièrement, afin de récupérer tous les regroupements possibles par notre drone (nous entendons par possible qui ne dépasse pas la capacité d'eau que peut porter le drone) nous avons fait un algorithme qui calcule tous les regroupements possibles. Ensuite, en connaissant ces regroupements nous avons cherché la permutation qui nous permettait de parcourir un regroupement de la manière la plus courte possible.

Notre deuxième approche fut la manière dont nous avons résolu le problème, une fois toutes nos données récupérées il faut les traiter, pour se faire nous sommes passés par la bibliothèque glpk en c. Notre code est un mélange de c et de c++ car certaines libraires du c++ (next_permutation, vector, etc.) nous sont plus familières mais le besoin du c s'est fait ressentir pour l'utilisation de la libraire glpk.

2. Partie algorithmique

2.1 Algorithme d'énumération des regroupements possible

Ce premier algorithme nous permet de générer tous nos regroupements possibles. Il nous permet donc de générer tous les sous-ensembles de lieux qui ne dépassent pas la capacité du drone. Par exemple avec le jeu de données du fichier exemple.dat, le parcours des lieux 1,2,4,5 sera exclu de nos regroupements car ici le volume d'eau est de 12 ce qui dépasse la capacité du drone qui est de 10.

ensembleDesPartiesPossibles

input : Donnees *p
output : vector<vector<int> >
Data : vector<vector<int» subset, vector<int> poid, vector<int> empty

```
1  poid.push_back(0)
2  subset.push_back(empty)
3  for i allant de 0 à p->lieux.size() do
4      vector < vector < int >> subsetTemp = subset
5      vector < int > poids = poid
6      for j allant de 0 à subsetTemp.size() do
7          poids[j]+ = p− > capacite[i]
8          if poids[j] <= p -> capaciteDrone then
9              subsetTemp[j].push_back(i + 1)
10             subset.push_back(subsetTemp[j])
11             poid.push_back(poids[j])
12         end
13     end
14 end
15 subset.erase(subset.begin())
16 returnsubset
```

Pour faire cet algorithme, nous avons raisonné de la façon suivante : on ajoute tous les regroupements de taille 1, puis tous ceux de taille 2, tous ceux de taille 3, etc. Ainsi on utilise un vecteur de vecteur intermédiaire, *subsetTemp* auquel nous allons ajouter des valeurs (i + 1) à chaque tour de boucle et cela sans jamais effacer la valeur précédente puisque l'on utilise la méthode *push_back*() propre au vector. En paralèlle, nous utilisons un vecteur qui stocke le poids des éléments ajouté à notre vecteur de vecteur intermédiaire *sbsetTemp*. Cette partie est extrèmement importante car on ajoute la valeur (i + 1) à *subsetTemp* uniquement si la capacité du drône est supérieur ou égale à celle du regroupement calculer, si c'est le cas on ajoute alors le vecteur créé dans le vector de vecteur *subset* que l'on va retourner.

Ainsi, admettons que le regroupement $\{1, 2\}$ dépasse la capacité du drône, celui ne sera pas ajouté au vector de vector *subsetTemp*. Et de ce fait on ne le réutilisera pas ensuite, en effet on n'essaiera donc pas dans les prochains tours de boucles de lui ajouter 3, 4 ou les deux. Cette conditionnelle sur le poids nous permet donc de ne garder que les regroupements avec une capacité inférieur ou égale à la capacité du drône et aussi au fur et à mesure de réduire le nombre de regroupement que l'on peut calculer.

2.2 Algorithme d'énumération des tournées

Une fois que on connait les regroupements possibles il faut savoir quelles sont les tournées, autrement dit, l'ordre dans lequel on parcourt chaque regroupement de sorte à faire le moins de chemin possible.

Dans un premier temps il nous faut donc calculer toutes les permutations possibles et ensuite regarder sur ces permutations lesquelles sont les plus petites.

ensembleDesPermutationsPossibles

input : Donnees *p, vector<vector<int> > regroupement
output : vector<vector<int> >
Data : vector<vector<int> > subset, vector<int> levecapush

```
1 for i allant de 0 à regroupement.size() do
2 |   do
3 |   |   subset.push_back(regroupement[i])
4 |   while
  |     next_permutation(regroupement[i].begin(),regroupement[i].end());
5 |   vector < int > v; v.push_back(−10)
6 |   subset.push_back(v)
7 end
8 returnsubset
```

Cette partie du code s'avère assez cruciale. Voici comment nous procédons : pour chaque regroupement possible on fait appel à la fonction next_permutation (fournie par la std) qui, pour un vecteur donné, son début et sa fin, va calculer UNE permutation possible. Donc tant que il existe des permutations on les calcule avec notre do while.

Chaque permutation calculée est donc ajoutée dans notre vecteur subset mais ily a aussi un petite détail technique, ici nous ajoutons un "-10" dans notre vecteur. Pourquoi cet ajout ? Nous allons voir dans notre second algorithme qu'il nous faut séparer chaque regroupement afin de pouvoir traiter nos regroupements un par un. Pour se faire notre -10 sert de balise entre chaque il nous permet juste de séparer, ainsi dans notre vecteur nous aurons quelque chose du style : "permutation1Regroupement1 ; permutation2Regroupement1 ; -10 ; permutation1Regroupement2". On saura, par convention que on change de regroupement.

Nous avons donc récupéré toutes nos permutations possibles pour un regroupement, désormais il faut décider pour un regroupement x quelle est sa permutation qui offre le chemin le plus court.

permutationsLesPlusPetites

input : Donnees *p, vector<vector<int> > regroupement, vector<int> &longueurTournee
output : vector<vector<int> >
Data : vector<vector<int> > subset, unsigned int c = 0, int s = 0, vector<int> tmp, int min = INT_MAX

```
1  while c < regroupement.size() do
2      s = 0
3      if regroupement[c][0] != (-10) then
4          s+ = p- > distancier[0][regroupement[c][0]]
5          for i allant de 0 à regroupement[c].size() do
6              s+ = p- >
                distancier[regroupement[c][i]][regroupement[c][i + 1]]
7          end
8          s+ = p- > distancier[regroupement[c].back()][0]
9          if min > s then
10             min = s
11             tmp = regroupement[c]
12         end
13     else
14         subset.push_back(tmp);
15         longueurTournee.push_back(min);
16         tmp.clear();
17         min = INT_MAX;
18     end
19     ++c
20 end
21 returnsubset
```

Tant que on a pas parcouru tous nos regroupements on continue. la variable s nous servira à stocker la longueur du regroupement et min à stocker la valeur minimum du regroupement. Si on est pas sur un indice qui vaut -10 (c'est à dire une balise) ça veut dire qu'on est dans un regroupement donc on regarde ce qu'il se passe. On ajoute la distance du point 0 au premier point de la permutation. Par exemple sur une permutation $\{1, 2, 3\}$ on ajoute la

distance de 0 à 1. Ensuite pour tous les éléments dans notre regroupement on va ajouter leur distance entre le point i et le point suivant. Ici on va ajouter la distance entre $1 \rightarrow 2$, $2 \rightarrow 3$. Pour finir on rajoute la distance entre le dernier point et notre point 0 (il faut revenir à la base) donc on ajoute ici $3 \rightarrow 0$. Une fois que c'est fait on regarde si cette distance est plus petite que notre minimum car on a fini cette permutation mais on a pas fini ce regroupement. La balise sert de balise pour distinguer deux regroupements mais pas les permutations, on parcours donc chacune des permutations en sommant leur taille, dès qu'on trouve une taille plus petite on garde le vecteur qui nous intéresse et cette taille.

Quand on rencontre un -10 c'est qu'on a fait toutes les permutations du regroupement. Donc on a gardé une valeur minimale, on récupère cette valeur minimale qu'on ajoute au vecteur de vetcteur *subset* et la longueur minimale qu'on met dans notre vecteur $longueurTournee$ afin de savoir pour la tournée i quelle longueur elle fait (la tournée en position i dans notre vecteur subset aura la longueur en position i dans notre vecteur $longueurTournee$). Nous n'oublions pas de réinitialiser les valeurs de notre vecteur et de *min*. Une fois qu'on a fait un tour de boucle on incrémente c pour passer à l'indice suivant.

On renvoie notre vecteur de vecteur *subset* qui contiendra ainsi toutes les permutations les plus petites, le vecteur $longueurTournee$ étant placé en paramètre modifiable il sera incrémenté autotmatiquement.

3. Structures de données et bibliothèques du c++

Nous nous sommes inspirés de la structure de données fournie dans le squelette de code mais nous l'avons très grandement modifié pour d'avantage d'aisance. Ainsi nous avons préféré utiliser des outils du c++ tel que les vecteurs, notemment pour stocker les lieux et leurs capacités. Un vecteurs de vecteurs pour stocker la matrice de distance et un enfin un entier contenant la capacité maximale du drône.

Aussi, nous avons utilisés plusieurs bibliothèques du c++, notemment la bibliothèque *algorithm* qui est fondamentale puisque nous utilisons la fonction $next\_permutation$ pour calculer l'ensemble de nos permutations.

4. Analyse des résultats

Pour l'analyse de nos résultats nous commençons par les données de type A. Voici pour nos données les temps d'exécutions avec d'un côté le temps nécessaire pour calculer nos permutations et de l'autre le temps nécessaire pour effectuer la résolution avec glpk.c

donnees	temps permutations	temps glpk	z
VRPA10.dat	0	0	z = 3656.000000
VRPA15.dat	0.002000	0.002000	z = 4735.000000
VRPA20.dat	0.001000	0.010000	z = 4818.000000
VRPA25.dat	0.017000	0.062000	z = 5932.000000
VRPA30.dat	0.056000	2.590000	z = 7279.000000
VRPA35.dat	0.010000	1.318000	z = 9583.000000
VRPA40.dat	0.096000	1.886000	z = 10893.000000
VRPA45.dat	0.186000	12.945000	z = 11889.000000
VRPA50.dat	0.343000	17.227000	z = 11666.000000

TABLE 1 – Données de type A

Nous observons donc ici que nos temps de calculs des permutations sont relativement rapides par rapport au glpk dès que on commence à travailler sur des instances qui commencent à devenir importantes.

Regardons maintenant nos instances de type B :

donnees	temps permutations	temps glpk	z
VRPB10.dat	0.001000	0.009000	z = 2137.000000
VRPB15.dat	0.273000	0.074000	z = 3080.000000

TABLE 2 – Données de type B

Pour nos données de taille B nous avons un problème de std :bad_alloc qui intervient sur le calcul de nos permutations. Ce qu'il se passe c'est que on stocke nos permutations et donc comme il calcule énormément de permutations il excède la capacité en mémoire de nos ordinateurs.

On constate globbalement que le temps requis par notre glpk semble plus important que celui de génération des permutations. Cepedant ceci est vrai pour des petites instances, nous pensons que pour des instances de grande taille B le traitement des permutations sera beacoup plus imposant que celui du glpk car les données étant beaucoup trop importantes le calcul des permutations sera extrêmement cher tandis que le solveur finalement fera toujours son travail sur certes plus de données mais ça restera raisonnable pour lui.

5. Améliorations

Le principale problème de notre programme concerne notre calcul des permutations les plus courtes. Nous utilisons la fonction *next_permutation* pour générer toutes les permuations dans un vecteur de vecteur et ensuite nous ne gardons que les permutation les plus courtes. Nous rencontrons là un problème de fuite mémoire : notre programme fonctionne très bien pour les fichiers VRPA mais rencontre un problème d'allocation à partir du fichier VRPB20.dat inclus jusqu'au fichier VRPB50.dat. Après recherche à l'aide du programme valgring, il semblerait que pour de très grande instance (à partir de VRPB20.dat donc) le fait que l'on calcul toutes les permuations avant de garder les plus courtes provoquent des fuites mémoire.

Par manque de temps, nous nous contenterons d'expliquer comment résoudre ce problème : il faut dans la boucle *do while* de notre fonction *ensemblesDesPermutationsPossibles* faire le calcul pour ne conserver que la plus petite, et ne pas le faire après qu'elles aient toutes été générées. On supprimerait ainsi la fonction *permutationsLesPlusPetites* et on gagnerait en compléxité spatiale et temporelle.

Enfin l'autre problème majeur de notre programme concerne le temps de calcul qui est exponentiel : en effet on met moins d'une seconde à calculer le résultat pour l'exemble.dat et plusieurs minutes pour les fichier VRPB. Ainsi l'une des priorités serait d'améliorer le temps de calcul, il faudrait pour cela utiliser des "trheads" pour optimiser nos temps de calculs en s'adaptant aux capacités des machines (multi-core, ...).

6. Conclusion

La survie à Trumpland a été bien plus dure que prévue.. Nous avons eu quelques difficultés avec glpk afin de bien comprendre comment fonctionne le solveur et une fois que nous avions compris cette histoire de dépassement mémoire que nous n'avons pas le temps de corriger.. Nous avons donc échoué et risquons de mourir si notre drone est amené à traiter des données B. Plus sérieusement, comme nous l'avons expliqué dans notre partie amélioration il sera possible de remédier à ce problème avec un peu plus de temps.

Le problème donné s'est avéré assez difficile à résoudre algorithmiquement parlant, les algorithmes de génération de regroupements nous ont posés quelques difficultés mais nous sommes arrivés à une version que nous estimons correcte. Pour l'ensemble des parties possibles nous pensons avoir une complexité $O(n^2)$ avec n le nombre de lieux. Le calcul de l'ensemble des permutations quand à lui est plus important