

MAHIER Loïc  
JEHANNO Clément

groupe 601B

# Rapport de projet <sup>\*</sup>: Recherche opérationnelle

---

<sup>\*</sup>rapport réalisé sous L<sup>A</sup>T<sub>E</sub>X

## Sommaire

---

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Partie algorithmique</b>	<b>3</b>
2.1	Algorithme d'énumération des regroupements possible . . . . .	3
2.2	Algorithme d'énumération des tournées . . . . .	5
<b>3</b>	<b>Structures de données et bibliothèques du c++</b>	<b>7</b>
<b>4</b>	<b>Analyse des résultats</b>	<b>8</b>
<b>5</b>	<b>Améliorations</b>	<b>9</b>
<b>6</b>	<b>Conclusion</b>	<b>10</b>

## 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 leurs masters doivent mettre en place une stratégie optimale pour récupérer 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 procédé de deux manières : premièrement, afin de récupérer tous les regroupements possibles par notre drone (possible signifie : 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 librairie 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 return subset

```

---

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 vecteur. En parallèle, nous utilisons un vecteur qui stocke le poids des éléments ajoutés à notre vecteur de vecteur intermédiaire *subsetTemp*. Cette partie est extrêmement importante car on ajoute la valeur ( $i + 1$ ) à *subsetTemp* uniquement si la capacité du drone est supérieure ou égale à celle du regroupement calculer, si c'est le cas on ajoute alors le vecteur créé dans le vecteur de vecteur *subset* que l'on va retourner.

Ainsi, admettons que le regroupement  $\{1, 2\}$  dépasse la capacité du drone, celui-ci ne sera pas ajouté au vecteur de vecteur *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 inférieure ou égale à la capacité du drone 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 qu'on connaît 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 chemin le plus court 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
6   v.push_back(-10)
7   subset.push_back(v)
8 end
9 return subset

```

---

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 bibliothèque *algorithm*) qui, pour un vecteur donné, son début et sa fin, va calculer une permutation possible. Donc tant qu'il existe des permutations on les calcule avec notre *do while*.

Chaque permutation calculée est donc ajoutée dans notre vecteur de vecteur *subset* mais il y a aussi un petit 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 sert juste à séparer les regroupements, ainsi, dans notre vecteur nous aurons quelque chose de semblable à : "permutation1Regroupement1 ; permutation2Regroupement1 ; -10 ; permutation1 Regroupement2". On saura, par convention qu'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  $regroupement[c][0] \neq (-10)$  then
4           $s += p \rightarrow distancier[0][regroupement[c][0]]$ 
5          for  $i$  allant de 0 à  $regroupement[c].size()$  do
6               $s += p \rightarrow distancier[regroupement[c][i]]$ 
6                   $[regroupement[c][i + 1]]$ 
7          end
8           $s += p \rightarrow distancier[regroupement[c].back()][0]$ 
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 return subset

```

---

Ainsi, tant qu'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) cela signifie qu'on est dans un regroupement donc on regarde ce qui 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  $i+1$ . 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. Notre  $-10$  sert de balise pour distinguer deux regroupements mais pas les permutations, on parcourt 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 vecteur *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 changé au cours de l'exécution de la fonction.

### 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, notamment pour stocker les lieux et leurs capacités. Un vecteur de vecteur pour stocker la matrice de distance et un enfin un entier contenant la capacité maximale du drone.

Aussi, nous avons utilisé plusieurs bibliothèques du `C++`, notamment 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 qu'on stock nos permutations et donc comme il calcule énormément de permutations il excède la capacité en mémoire de nos ordinateurs.



On constate globalement que le temps requis par notre glpk semble plus important que celui de génération des permutations. Cependant ceci est vrai pour des petites instances, nous pensons que pour des instances de grande taille (B en l'occurrence), le traitement des permutations sera beaucoup plus imposant que celui de 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 cela restera raisonnable pour lui.

## 5. Améliorations

---

Ainsi, le principal problème de notre programme concerne notre calcul des permutations les plus courtes. Nous utilisons la fonction *next\_permutation* pour générer toutes les permutations 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 permutations 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 permutation, et ne pas le faire après qu'elles aient toutes été générées. On supprimerait ainsi la fonction *permutationsLesPlusPetites* et on gagnerait en complexité spatiale et temporelle. On ne devrait dès lors plus rencontrer d'erreur pour résoudre de grand jeux de données.

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'exemple.dat et plusieurs minutes pour les fichiers VRPB. Ainsi l'une des priorités serait d'améliorer le temps de calcul, il faudrait pour cela utiliser des "threads" 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 en effet eu quelques difficultés avec glpk afin de bien comprendre comment fonctionne le solveur. Aussi après que nous ayons compris d'où provenait ce problème de dépassement mémoire, nous n'avons pas eu le temps de le 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 aussi en  $O(n + m)$  avec  $n$  le nombre de regroupements et  $m$  le nombre de permutations. On parcourt chaque regroupement et on fait la somme de chaque permutation pour trouver la plus petite.

Pour conclure, nous avons eu du mal avec l'aspect algorithmique mais glpk s'est avéré bien plus difficile puisque finalement nous pensons avoir réussi à fournir un code assez optimal en ce qui concerne les algorithmes (si on effectue la modification décrite dans les améliorations) tandis que glpk nous a pris beaucoup plus de temps à comprendre et à correctement mettre en place.