PHALAVANDISHVILI Demetre - JAMET Félix - Gr 601B

Projet RO : BladeFlyer II - conquest of water

Test pour une capacité de 55

Introduction

Le projet consiste à trouver la meilleure solution pour collecter de l'eau parmis différents points de pompage en utilisant un drone capable de récupérer l'eau des points de pompage.
Notre travail consiste à générer tous les regroupements possible entre les points pompage en les filtrant de manière à ce que la quantité totale d'eau qu'ils représentent ne dépasse pas la capacité de transport du drone.
Il s'agit ensuite de déterminer les regroupements à choisir de manière à minimiser le trajet parcouru et à ne passer par un seul point, ce qui correspond à un problème de partitionnement d'ensemble.

Analyse du problème

Posons tout d'abord ce problème sous la forme d'un programme linéaire :

Paramètres

- Re, l'ensemble des regroupements réalisables

Rappel des notations du sujet

- $n \in \mathbb{N}$, le nombre de lieux
- d_i, la quantité d'eau disponible au point i, avec $i \in \{1, \dots, n\}$
- $Ca \in \mathbb{N}$, la capacité du drone
- l_r, la longueur de la plus courte tournée visitant chaque point de pompage du regroupement r

Définition

Un regroupement r est dit *réalisable* si $\sum_{i \in r} d_i \leq Ca$.

Autrement dit, si sa quantité d'eau totale peut être transportée par le drone.

Variables de décison

$$x_r = \begin{cases} 1 \ si \ on \ choisit \ le \ regroupement \ r \\ 0 \ sinon \end{cases}$$

Avec $r \in Re$

Fonction objectif

Minimiser la distance parcourue par le drone

$$min \sum_{r \in Re} l_r * x_r$$

ce qui donne $\sum_{i=1}^{27} l_i * x_i$ sur l'exemple donné dans le sujet, car dans cet exemple, $Re = \{1, \dots, 27\}$

Contraintes

- Le point de pompage i doit être visité une et une seule fois.
  $\forall i \in \{1, \dots, n\}, \sum_{r \in Re_i} x_i = 1$
  Avec Re_i l'ensemble des regroupements réalisables contenant i
- Contrainte sur l'exemple du sujet, pour le point de pompage 1 :
  $\sum_{i=1}^{13} x_i = 1$
  En l'occurence, $Re_1 = \{1, \dots, 13\}$
- Contrainte d'integrité :
  $x_i \in \{0, 1\}, \forall i \in Re$

Description des algorithmes

Génération des sous-ensembles

Il s'agit ici de générer les sous-ensembles des tournées réalisables, c'est à dire celles dont la quantité d'eau ne dépasse pas la capacité du drone.

Le principe utilisé ici est de générer les regroupements de taille 1 et de s'en servir pour générer ceux de taille 2 puis ceux de taille 3, et ainsi de suite.

Posons p le nombre de points de pompage ($p = n - 1$).

Initialisation

La première étape consiste simplement à générer les tournées ne contenant qu'un seul point de pompage.
On se retrouve donc avec p vecteurs de taille 1.

Hérédité

Genération

On remarque que les regroupements de taille t sont des préfixes des regroupements de taille $t + 1$.
On se sert de cette propriété pour construire l'ensemble des regroupements.

Soit r un regroupement de taille t.
On construit S_r, l'ensemble des regroupements de taille $t + 1$ et de préfixe r comme suit :

$S_r \leftarrow \{\}$

$\forall\, i \,\in\, [\, max(r) + 1, \,\dots,\, p\,],\; S_r \,\leftarrow\, S_r \,\cup\, \{\, r \,\cup\, \{i\}\}$

On applique ce procédé à tous les regroupements de taille t pour construire tous les regroupements de tailles $t + 1$.
On peut ainsi construire tous les regroupements de taille allant de 1 à n à partir des regroupements de taille 1.

Filtrage

On remarque que cet algorithme génère tous les regroupements, même ceux non réalisables.
On peut résoudre ce problème en testant si l'instance est réalisable avant de l'ajouter à la solution partielle.
L'avantage de cette approche est que les regroupements ayant un préfixe non réalisable ne sont pas envisagés, économisant ainsi du temps de calcul.

Soit r un regroupement de taille t.
On construit R_r, l'ensemble des regroupements réalisables de taille $t+1$ et de préfixe r comme suit :

$S_r \leftarrow \{\}$

$$\forall\, i \in [\, max(r)+1, \ldots, p\,],\ S_r \leftarrow \begin{cases} S_r \cup \{r \cup \{i\}\}\ si\ (\sum_{j \in r} d_j) + d_i \leq Ca \\ S_r\ sinon \end{cases}$$

Calcul du plus court chemin

La méthode utilisée pour calculer le plus court chemin d'un regroupement donné consiste simplement à parcourir toutes les permutations et calculer pour chacune la distance parcourue.

Description des classes et en-têtes utilisés

regroupement

Cette classe représente un ensemble de points de pompage.

Attributs

Attribut	Type	Description
lieux_	vecteur d'entiers	indice des points de pompage du regroupement
quantite_	entier signé	quantité d'eau du regroupement
distance_	entier signé	distance du plus cours chemin passant par tous les points de pompage

Méthodes

`void add(unsigned int point, unsigned int quantité)` :
Ajoute le point de pompage d'indice *point* et de quantité *quantite* dans le regroupement.

donnees

Cette classe permet de lire les données présentes dans le dossier *data*.
Son constructeur et son destructeur reprennent respectivement le code des fonctions *lecture_data* et *free_data*.

Attributs

Attribut	Type	Description
nblieux_	entier	nombre de lieux
capacite_	entier	capacité du drone
demande_	tableau d'entiers	capacité des lieux de pompage
C_	matrice d'entiers	distancier

Méthodes

`std::vecteur<regroupements> generer_regroupements() const` :
Construit tous les regroupements réalisables, sans calculer le plus court chemin.

`unsigned int distance( std::vector<unsigned int> lieux ) const` :
Calcule la distance du parcours dans l'ordre de *lieux*, en commençant et en terminant par le point 0.

`void init_distance(regroupement& rgrp) const`
Initialise l'attribut distance_ et réordonne les lieux_ de *rgrp* selon le plus court chemin.

probleme

Cette classe représente l'instance du problème à résoudre.

Attributs

Attribut	Type	Description
regroupements_	vecteur de regroupements	ensemble des regroupements réalisables
regroupements_contenant_	vecteur de vecteur d'entiers	regroupements[i] = indice des regroupements contenant le point de pompage i

Méthodes

`void init_regroupements_contenant()` :
Parcours chaque regroupement et ajoute son indice i à regroupements_contenant_$[j], \forall j \in regroupements_[i]$.

chrono

Cette classe reprends le code et les fonctionnalités des fonctions *chrono_start*, *chrono_stop* et *chrono_ms*, en ajoutant cependant la possibilité d'obtenir le temps directement en secondes, grâce à la méthode *to_sec*. Elle nous permet donc de déterminer le temps de calcul des étapes cruciales de la résolution du problème.

glpkwrapper

Cette classe est utilisée pour résoudre le problème de partitionnement.

Méthodes

`void construit_couts()` :
Initialise les couts de chaque tournée à la valeur du plus court chemin du regroupement représentant cette tournée en parcourant la distance le plus court de la regroupement.

`void construit_taille_contr()` :
Initialise les coefficients de la matrice creuse.

`void def_probleme()` :
Avec la fonction *glp_set_col_bnds*, on précise que les variables de décision comme sont entières, avec la valeur *GLP_DB* et puis en utilisant *glp_set_col_kind* on prcecise que ces sont des variables binaires en utilisant mots-clés *GLP_BV*. Puis en utilisant la fonction de *glpk*, *glp_set_obj_coef*, on attribut les couts *couts* à chaque variable de decision.

Puis avec la procedure *glp_load_matrix* de *glpk* on charge *nb_creux_*, *ia_*, *ja_*, *ar_* précedemment définis.

`void resoudre_probleme()` :
Au début de cette procédure on appelle la fonction *def_probleme*, puis on utilise les fonctions fournies par glpk pour résoudre le problème.
glp_write_lp nous permet d'écrire le problème dans un fichier *.lp* et permet de simplifier le debugguage en cas d'erreur lors de l'insertion des données.
glp_simplex permet de résoudre le problème en utilisant l'algorithme du simplexe.
glp_intopt permet d'obtenir une solution optimale en relaxant les variables de décision.

`void afficher()`:
Cette méthode permet d'afficher la distance minimale à parcourir ainsi que les regroupements correspondant à cette distance.
glp_mip_obj_val permet d'obtenir la valeur de la fonction objective.
glp_mip_col_val cette méthode permet de parcourir une par une toutes les variables de décision.

En-tête *container_overload.hpp*

Cet en-tête contient des surcharges de l'opérateur de redirection sur flux pour les conteneurs std::vector et std::array.

Implémentation des algorithmes en c++

Génération des sous-ensembles

Le code décrit dans cette section provient de la méthode *donnees::generer_regroupements*

Initialisation

```
std::vector<regroupement> result;

for(unsigned int i = 1; i < nblieux_; ++i)//on parcours tous les points de pompage
    if(capacite_ >= demande_[i])//on teste quand même la capacité
        result.emplace_back( std::vector<unsigned int>{i}, demande_[i]);
```

Hérédité

```cpp
unsigned int start = 0;
unsigned int stop = result.size();

for(unsigned int stage = 2; stage < nblieux_; ++stage)
{//génération des sous-ensembles de taille stage
    for(; start < stop; ++start)
    {//parcours de chacun des points de taille stage-1
        for(unsigned int i = result[start].dernier_point() + 1; i < nblieux_;
++i)
        {//création des result de préfixe result[start]
            if(result[start].quantite() + demande_[i] <= capacite_)
            {//filtrage. seuls les regroupements dont la quantité d'eau est
transportable sont ajoutés
                result.push_back( result[start] );
                //le nouveau regroupement est une copie de result[start] ...
                result.back().add(i, demande_[i]);
                // ... auquel on ajoute le point i et la quantité d'eau
correspondante
            }
        }
    }

    start = stop;
    stop = result.size();
}
```

Calcul du plus court chemin

Comme expliqué dans la section *Description des algorithmes*, la méthode de calcul de plus court chemin consiste simplement à parcourir tous les chemins possibles et conserver le plus court.
Pour cela, nous avons utilisé la fonction std::next_permutation permettant de parcourir une à une les permutations d'un conteneur de la bibliothèque standard.

```cpp
void donnees::init_distance(regroupement& rgrp) const
{
    std::vector<unsigned int>& lieux(rgrp.lieux());
    std::vector<unsigned int> ordre_opti(lieux);

    unsigned int min_dist = distance(lieux);
    unsigned int tmp_dist;
    while( std::next_permutation(lieux.begin(), lieux.end()) )
    {//parcours des permutations. le parcours s'arrête lorsque la prochaine permutation est triée.
        tmp_dist = distance(lieux);
        if(tmp_dist < min_dist)
        {
            ordre_opti = lieux;
            min_dist = tmp_dist;
        }
    }
    lieux = ordre_opti;
    rgrp.distance() = min_dist;
}
```

Compilation et execution

Compilation

Ce projet utilise CMake pour générer des fichiers de compilation. Voici comment le compiler sous gnu/linux :

Depuis le dossier *build*

```
cmake ..
make
```

Execution

Toujours depuis le dossier *build*

```
./ro_test data/fichier.dat
```

Analyse des résultats

À l'aide de la classe *chrono*, nous avons pu chronometrer les différentes étapes de la création et de la résolution du problème.
Bien entendu, tous les tests ont été effectués sur la même machine.

Dossier A

La résolution du problème pour les données du dossier A se fait très rapidement : à peine plus de 6 secondes sur une machine moderne pour 50 lieux, comme le montre la capture suivante :

`./ro_test data/A/VRPA50.dat`

```
chargement des donnees du fichier data/A/VRPA50.dat ... 0 secondes

calcul des regroupements réalisables ... 0.006 secondes
nombre de regroupements générés : 44145

calcul des plus courts chemins ... 0.036 secondes

génération du problème ... 0 secondes

résolution du problème avec glpk

Writing problem data to 'trumpland.lp'...

122038 lines were written

[...]

            INTEGER OPTIMAL SOLUTION FOUND

la résolution du problème avec glpk a pris 6.293 secondes

valeur optimale de la fonction objective :
        Z* = 11666

tournees selectionnees :
        [13]
        [1; 49]
        [25; 33]
        [34; 41]
        [21; 4; 36]
        [7; 39; 27]
        [44; 9; 45]
        [18; 10; 23]
        [11; 22; 24]
        [14; 29; 40]
        [16; 47; 38]
        [17; 20; 32]
        [19; 48; 35]
        [30; 28; 43]
        [2; 5; 3; 37]
        [6; 46; 42; 15]
        [26; 12; 8; 31]

temps de calcul total : 6.335
```

On remarque que c'est de loin la résolution du problème de partitionnement d'ensemble qui prends le plus de temps, les autres opérations étant négligeables devant celle-ci.

Dossier B

Comme prévu, les données du dossier B sont plus difficiles à résoudre. La plus grande instance que nous avons réussi à résoudre est celle comportant 35 lieux :

`./ro_test data/B/VRPB35.dat`

```
chargement des donnees du fichier data/B/VRPB35.dat ... 0 secondes

calcul des regroupements réalisables ... 0.113 secondes
nombre de regroupements générés : 1954584

calcul des plus courts chemins ... 394.366 secondes

génération du problème ... 0.066 secondes

résolution du problème avec glpk

Writing problem data to 'trumpland.lp'...
6475627 lines were written

[...]

            INTEGER OPTIMAL SOLUTION FOUND

la résolution du problème avec glpk a pris 705.524 secondes

valeur optimale de la fonction objective :
        Z* = 5399

tournees selectionnees :
         [10; 3; 32; 20]
         [18; 28; 23; 12; 25]
         [9; 30; 5; 17; 16; 29]
         [19; 26; 6; 13; 27; 21]
         [8; 33; 11; 14; 24; 34]
         [1; 7; 4; 2; 31; 15; 22]

temps de calcul total : 1100.07
```

Plus court chemin

Même si c'est à nouveau glpk qui a consommé le plus de temps cpu, on remarque que le calcul du plus court chemin a pris un temps non négligeable cette fois ci.
Cela est dû au fait que la méthode utilisée est sensible à l'explosion combinatoire. En effet, à un chemin de n points correspond $n!$ permutations, permutations que notre implémentation parcours entièrement, et ce pour tous les regroupements.
Il serait donc grandement profitable d'améliorer cette partie du programme. Le voyageur de commerce étant un problème difficile, nous n'avons cependant pas cherché à implémenter une solution plus efficace.

Génération des regroupements réalisables

Il est intéressant de remarquer que le calcul des regroupements s'est effectué très rapidement, comparé au plus court chemin.
Bien que la génération de regroupements soit aussi sujette à l'explosion combinatoire, le fait que notre implémentation ne considère pas les regroupements de préfixe non réalisable permet en quelque sorte d'y

échapper.
En quelque sorte car c'est seulement dû à la faible capacité du drone.

Nous avons effectué des tests en nous basant sur le fichier data/B/VRPB35.dat mais en remplaçant la capacité originale de 33 par une capacité de 40, 50 et 55. Les fichiers modifiés sont disponibles dans le dossier data/C.

Test pour une capacité de 40

`./ro_test data/C/40.dat`

```
chargement des donnees du fichier data/C/40.dat ... 0 secondes

calcul des regroupements réalisables ... 0.719 secondes
nombre de regroupements générés : 10976776
```

Avec seulement 7 capacité en plus, le calcul dure près de 7 fois plus longtemps et génère 5 fois plus de regroupements.

Test pour une capacité de 50

`./ro_test data/C/50.dat`

```
chargement des donnees du fichier data/C/50.dat ... 0 secondes

calcul des regroupements réalisables ... 5.493 secondes
nombre de regroupements générés : 83277188
```

Le temps de calcul reste raisonnable mais on remarque qu'il y a cette fois plus de 83 millions de regroupements générés, ce qui nécessite l'utilisation de plus de 8gb de mémoire vive.

Test pour une capacité de 55

`./ro_test data/C/55.dat`

```
chargement des donnees du fichier data/C/55.dat ... 0 secondes

Killed
```

La quantité de mémoire nécessaire dépasse celle de la machine. Cependant, même avec suffisament de mémoire, cette instance aurait demandé beaucoup trop de temps pour être résolue. L'ajout de capacité entraîne la génération d'un grand nombre de regroupements, potentiellement plus longs que les précédents et pour chacun desquels il faut calculer un plus court chemin et rajouter des contraintes.

On peut ainsi se rendre compte qu'il suffit de peu pour passer d'une instance résolvable à une instance dont la résolution est inconcevable.