Rapport de projet de Recherche Opérationnelle
Mission planning pour une flotte de robots d'exploration

Arnaud Grall et Thomas Minier
groupe 601A

Mercredi 1 Avril 2015

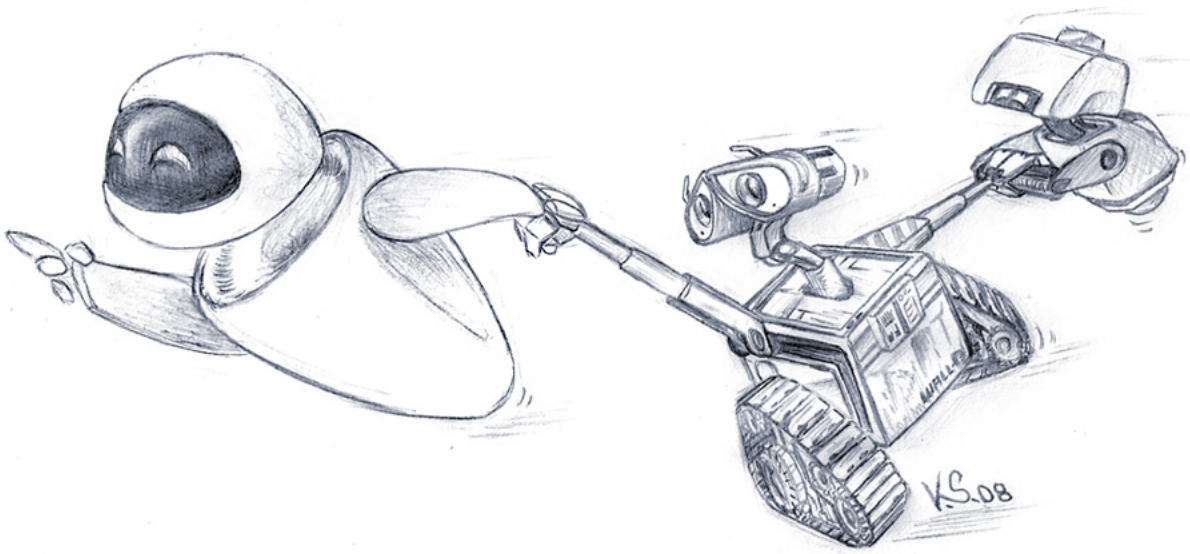

Table des matières

1 Introduction

1.1 Rappel du problème

Dans ce projet, il nous a été demandé de concevoir un programme capable de résoudre une variante du problème mTSP (multitple Traveling Salesman Problem). Pour ce faire, nous utiliserons la bibliothèque GLPK. Nous rappelons que le problème peut se modéliser sous la forme du programme linéaire suivant :

$$Min\ z = \sum_{i-1}^{n}\sum_{j-1}^{n} c_{ij}x_{ij}$$

$$s.c. \sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in \{1,\dots,n\} \tag{1}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \in \{1,\dots,n\} \tag{2}$$

$$\sum_{i,j\in S} x_{ij} \leq |S| - 1 \quad \forall S\ avec\ |S| \leq n-1 \tag{3}$$

$$x_{ij} \in \{0,1\} \quad \forall i,j \in \{1,\dots,n\}$$

Ce problème peut se modéliser aisément dans GLPK. Cependant, ajouter la contrainte (3) telle quelle n'est pas une bonne idée, car elle risque de saturer la mémoire de l'ordinateur sur de grandes instances du problème et de faire exploser le temps de calcul. Nous allons donc tenter de minimiser le nombre de contraintes à ajouter à l'aide d'un algorithme.

1.2 Algorithme de résolution

Notre programme se décompose en plusieurs parties. Tout d'abord, nous commerçons par résoudre le problème en ajoutant seulement les contraintes (1) et (2). Nous obtenons alors une solution non optimale, car elle comporte des cycles. Nous détectons donc ces derniers et gardons en mémoire le plus petit cycle. Si ce cycle est le seul que l'on a trouvé, alors l'algorithme s'arrête. Sinon, nous ajoutons une contrainte pour casser ce sous-cycle et on relance la résolution du problème. On continue jusqu'à ne trouver qu'un seul cycle. On comptera aussi le nombre d'appels à GLPK et le temps nécessaire pour résoudre le problème, qui seront tous les deux affichés en même temps que la solution optimale.

1.3 Compilation et exécution du programme

Pour simplifier l'utilisation de notre programme, nous avons défini des règles de compilation et d'exécution dans un Makefile. Pour compiler l'exécutable, il suffit de lancer la commande `make robots`. Pour l'exécuter avec un fichier de données, il suffit d'utiliser la commande `./robots fichier_de_donnees`. Pour résoudre l'exemple, il faut utiliser la commande `make exemple`, et pour résoudre n'importe quel problème de type plat ou relief, il faut utiliser la commande `make fichier_de_donnees`, où i*i* correspond au numéro du fichier. Par exemple, pour résoudre plat10, il suffit d'utiliser la commande `make plat10`. Il est aussi possible de résoudre tous les problèmes plat d'un coup, avec `make run_plats`, ou tous les reliefs, avec `make run_reliefs`.Enfin, pour résoudre tous les problèmes, il suffit d'utiliser la commande `make all` ou simplement `make`.

La totalité des résultats ainsi que le fichier généré par GLPK pour représenter le problème seront stockés dans le répertoire **resultats**.

2 Preuve de l'algorithme

2.1 Preuve de la solution optimale

Après la première résolution de GLPK, on obtient une liste des permutations possibles en fonction des 2 premières contraintes de bases qui sont là pour ignorer les cycles de taille 1. Notre algorithme de recherche de cycles va passer au crible les permutations pour en ressortir un ensemble disjoint de cycles. Dans cette ensemble, on peut avoir soit un unique cycle représentant la solution optimale, soit des cycles de taille 2 ou plus.

Une fois cet ensemble défini, l'algorithme repère le plus petit cycle et le programme ajoute une nouvelle contrainte en fonction de ce cycle. Cette contrainte permet, lors de la résolution d'une nouvelle instance du problème, de ne plus retomber sur une liste des permutations permettant de retrouver un ensemble de cycles disjoints qui nous ferait définir un cycle minimal identique à cette même contrainte.

Nous avons donc à la suite de notre recherche de cycle une solution optimale ou une contrainte à ajouter, ceci sous conditions que notre algorithme nous donne à coup sûr un cycle. Et c'est ce que nous allons déterminer dans la partie suivante.

2.2 Preuve de la terminaison

Nous sauterons la partie déclarations de variables pour plus de lisibilité ainsi que les lignes sans intérêt. Pour plus d'informations, le code source de l'algorithme est disponible en annexe à la page 7. Tous ce qui a été supprimé car non pertinent est symbolisé par "[...]". Le code représente l'algorithme de recherche du cycle minimal.

La première boucle *For* d'initialisation se fait sur le nombre de permutations. Elle termine donc toujours car il n'y a pas de modification interne de la variable d'incrémentation. La deuxième boucle *For* se fait aussi sur le nombre de permutations qui assure que la boucle se terminera toujours d'une manière ou d'une autre (même explication que précédemment). Mais il existe, à l'intérieure de cette deuxième boucle *For*, une boucle *While* qui parcours toutes les permutations jusqu'à retomber sur un élément déjà lu. A moins d'avoir un lieu indéterminé qui débouche sur "rien", la boucle se terminera elle aussi forcément.

Cette deuxième boucle *For* nous permet donc de récupérer un ensemble disjoint de cycles ou un unique cycle de la taille du problème. Cet ensemble est ensuite analysé par notre dernière boucle qui récupère le cycle minimal. Elle se termine elle aussi à coup sûr car même si nous avons 50 000 cycles de même taille, seul le premier sera pris en compte. De même si l'ensemble ne contient qu'un cycle il sera choisit. Et par définition des permutations et du problème, si l'ensemble ne contient qu'un cycle c'est que c'est une solution admissible.

C'est une solution admissible mais aussi optimale car, comme le cycle déterminé par l'algorithme est transformé en contrainte par le programme et ensuite ajoutée au problème avant qu'une autre résolution soit lancée, nous avons toujours un ensemble de contraintes de tailles égales ou de plus en plus importantes. Et comme les contraintes et chaque instance du problème sont analysés les unes après les autres, la première solution admissible du problème sera obligatoirement optimale.

3 Correspondance problème - GLPK

La bibliothèque GLPK utilise des indices qui partent de 1. Notre programme utilisant des indices partant de 0, il convient donc d'établir la correspondance entre nos indices et ceux de la matrice utilisée par GLPK. Étant donné une case du distancier à la ligne i et à la colonne j, cela correspondra, dans la matrice de GLP, à :

— $ja = (i-1) * taille\_distancier + j$

— $ia = i$

où ja est l'indice de la colonne de la matrice, et ia est l'indice de la ligne de la matrice.

4 Exécution avec l'exemple

Nous testons notre algorithme sur le fichier **exemple.dat** qui nous a été fourni. Nous rappelons qu'il représente le distancier suivant :

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	0	768	549	657	331	559	250
2	786	0	668	979	593	224	905
3	549	668	0	316	607	472	467
4	657	979	316	0	890	769	400
5	331	593	607	890	0	386	559
6	559	224	472	769	386	0	681
7	250	905	467	400	559	681	0

Nous commençons par charger dans la matrice de GLPK les contraintes (1) et (2), puis nous lançons une première résolution. Nous obtenons alors une solution $z = 2220.000000$ mais avec un certain nombres de cycles. Si nous voyons ces cycles comme une permutation, nous obtenons :

$$\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 5 & 6 & 4 & 3 & 7 & 2 & 1 \end{array}$$

et comme un produit de cycles disjoints, nous obtenons :

$$(157)(26)(34)$$

Nous choisissons donc de casser le plus petit cycle. Il s'agit ici de (26) et il est de taille 2. Nous ajoutons donc la contrainte suivante :

$$x_{26} + x_{62} \leq 1$$

Nous chargons ensuite les bonnes valeurs dans la matrice de GLPK puis nous relançons la résolution du problème. Nous obtenons alors une solution $z = 2335.000000$ ainsi que plusieurs autres cycles. Si nous voyons ces cycles comme une permutation, nous obtenons :

$$\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 7 & 6 & 4 & 3 & 2 & 5 & 1 \end{array}$$

et comme un produit de cycles disjoints, nous obtenons :

$$(17)(265)(34)$$

Nous choisissons donc de casser le plus petit cycle. Il s'agit ici de (17) et il est de taille 2. Nous ajoutons donc la contrainte suivante :

$$x_{17} + x_{71} \leq 1$$

Une fois encore, nous rechargeons les bonnes valeurs dans la matrice de GLPK et nous relançons la résolution du problème. Nous obtenons alors une solution $z = 2575.000000$ ainsi qu'un seul cycle. Si nous le voyons comme une permutation, nous obtenons :

$$\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 7 & 6 & 2 & 3 & 1 & 5 & 4 \end{array}$$

et comme un produit de cycles disjoints, nous obtenons :

$$(1743265)$$

Nous n'avons donc plus qu'un seule cycle, ce qui correspond à notre solution optimale. Le problème aura donc nécessité trois appels à GLPK, aura ajouté deux contraintes au problème et aura pris moins d'une seconde à se résoudre.

5 Résultats des autres instances du problème

5.1 Instances de type plat

Fichier	z	Temps écoulé	NB d'appels à GLPK	NB de contraintes ajoutées
plat10	170.000000	0.003 s	3	2
plat20	201.000000	0.037 s	9	8
plat30	152.000000	0.103 s	12	11
plat40	196.000000	0.667 s	16	15
plat50	215.000000	0.895 s	25	24
plat60	135.000000	1.578 s	23	22
plat70	163.000000	7.461 s	35	34
plat80	187.000000	8.683 s	30	29
plat90	162.000000	19.179 s	38	37
plat100	173.000000	26.577 s	40	39
plat110	165.000000	66.921 s	47	46
plat120	141.000000	106.905 s	53	52
plat130	117.000000	159.256 s	53	52
plat140	142.000000	156.171 s	49	48
plat150	148.000000	169.945 s	58	57

On constate que le temps de calcul et le nombre de contraintes ajoutées croissent très rapidement avec la taille du distancier, ce qui semble plutôt cohérent. En effet, plus il y a de chemins possibles, plus on risque d'obtenir des cycles qu'il faudra casser. On remarque aussi qu'à partir de `plat120`, la croissance ralentit nettement.

5.2 Instances de type relief

Fichier	z	Temps écoulé	NB d'appels à GLPK	NB de contraintes ajoutées
relief10	198.000000	0.002 s	2	1
relief20	250.000000	0.046 s	6	5
relief30	116.000000	0.009 s	1	0
relief40	190.000000	0.136 s	6	5
relief50	167.000000	0.25 s	6	5
relief60	264.000000	0.241 s	6	5
relief70	115.000000	0.372 s	6	5
relief80	375.000000	0.439 s	5	4
relief90	119.000000	0.746 s	5	4
relief100	233.000000	2.857 s	8	7
relief110	334.000000	1.907 s	8	7
relief120	341.000000	1.844 s	7	6
relief130	304.000000	1.201 s	4	3
relief140	111.000000	1.743 s	7	6
relief150	102.000000	12.374 s	10	9

Là encore, on remarque que le temps de calcul et le nombre de contraintes ajoutées croissent avec la taille du distancier, mais beaucoup plus lentement que précédemment. En effet, la croissance se fait par palier. Les problèmes sont de plus beaucoup plus rapide à résoudre. On peut supposer que la présence de relief augmente le nombre de chemins possibles et réduit donc la probabilité que des cycles se forment.

6 Conclusion

Ce projet nous aura permis de mieux comprendre la bibliothèque GLPK et de nous confronter à un problème de recherche opérationnelle où résoudre un problème en appliquant la méthode su simplexe directement sur un programme linéaire n'est pas forcement une bonne idée. Nous auront vu comment raffiner un problème par un traitement algorithmique pour optimiser sa résolution et nous épargner du temps de calcul et une éventuelle saturation de la mémoire de la machine.

On remarque aussi que notre programme pourrait être améliorer de plusieurs manière. Par exemple, la réallocation de la matrice de GLPK pourrait être faite à l'avance et par grand bloc mémoire, pour éviter la fragmentation. De plus, lors de la détection des cycles, on pourrait arrêter la recherche dès que l'on trouve un cycle de même taille que le distancier, ce qui nous éviterait des itérations superflues.

7 Annexe : algorithme de recherche du cycle minimal

```
cycle_min min_cycles(int * cycles, int taille) {
    //Déclarations des variables utiles
    [...]

    //Initialisation des valeurs des différents tableaux
    for(i = 0; i < taille; i++) {
        deja_lu[i] = false;
        est_debut_cycle[i] = false;
        c[i] = (int *)malloc(taille * sizeof(int));
        taille_cycles[i] = 2; // la taille min d'un cycle est de 2
    }

    //Parcours du tableau des cycles
    int z = 0;
    for(i = 0; i < taille; i++) {
        // si l'arc n'a pas été lu
        if( (! deja_lu[i])) {
            int y = cycles[i];
            //on initialise les deux premières valeurs du cycle
            c[z][0] =  i;
            c[z][1] = cycles[i];
            //on a lu le premier arc
            deja_lu[i] = true;
            deja_lu[y] = true;
            //on note que i est un début de cycle
            est_debut_cycle[i] = true;
            est_debut_cycle[y] = true;
            // on incrémente le nombre de cycles
            nb_cycles++;
            j = 2;
            // tant que l'indice du cycle que l'on va lire n'a pas déjà  été lu
            while(deja_lu[cycles[y]] == false) {
                deja_lu[cycles[y]] = true; // on a lu l'arc
                est_debut_cycle[cycles[y]] = true;
                //on remplit les tableaux
                c[z][j] = cycles[y];
                taille_cycles[z]++;
                // on avance dans le parcours
                y = cycles[y];
                j++;
            }
            z++;
        }
    }
```

[suite page suivante]

```
    //recherche de l'indice du min sur taille_cycles
    int min_c = taille_cycles[0];
    int ind_min = 0;
    for(i = 1; i < nb_cycles; i++) {
        if(est_debut_cycle[i]) {
            if(taille_cycles[i] < min_c) {
                min_c = taille_cycles[i];
                ind_min = i;
            }
        }
    }

    // On a trouvé le plus petit cycle et sa taille
    //Creation de la structure de retour
    cycle_min c_min;
    [...]
    //libération de la mémoire

    return c_min;
}
```