Rapport de Projet: Le voyageur de commerce

Benjamin Saint-Sever, Caitlin Dagg, Manon Pintault

19 avril 2014

Table des matières

Chapitre 1

Introduction

Le but de ce projet était de fournir un programme fonctionnant en ligne de commande permettant de calculer des solutions (pas forcément optimales) au problème du voyageur de commerce métrique (c'est-à-dire calculer le meilleur trajet à parcourir pour un ensemble de villes données sans repasser par une ville déjà visitée). Dans le cadre de ce projet, l'ensemble des villes a été donnée sous forme de matrice de distances.

Chapitre 2

Vie du projet

2.1 Logiciel de gestion de version

Afin de pouvoir réaliser ce projet, nous avons choisi d'utiliser un dépôt en ligne : github. Si nous avons opté pour cette solution c'est car il nous permettait de choisir entre l'utilisation de svn ou git. De plus c'est un outil simple d'utilisation et accessible avec une simple connexion internet depuis n'importe quel support (mac, windows, linux).

2.2 organisation

Pendant toute la durée du projet, nous nous sommes partagés les tâches chaque semaine. Par exemple, certains travaillaient un algorithme, un autre s'occupait du makefile, de réfléchir sur le prochain algorithme à faire et le dernier créait des tests et gérait l'organisation des fichiers. Mais généralement nous rajoutions chacun des lignes de codes aux différents algorithmes et modules pour améliorer ceux-ci, ajouter des fonctionnalités et régler quelques soucis rencontrés.

Chapitre 3

Présentation des algorithmes

3.1 Nearest Neighbour

L'algorithme Nearest Neighbour consiste à visiter le point le plus proche non-visité jusqu'à obtenir un parcours complet des points. Cette algorithme ne fournit que rarement le parcours optimal, mais a un complexité en O(n) (contrairement au O(n !) de l'algorithme de brute force), et fournit une distance qui se situe à environ deux fois la distance minimale, ce qui peut le rendre utile. Notre algorithme prends en paramètre une matrice de points, et partant du premier, utilise la matrice des distances et la propriété "visitée/non-visitée" de la structure de donnée des points pour trouver le prochain point. Ce point est ensuite ajouté à la liste des points représentant le parcours (qui est en fait un tableau), et on recommence en cherchant le « nearest neighbour » de ce point. L'algorithme s'appuie sur la fonction "PointLePlusProche" qui prend en paramètre le point de départ et une matrice de point, la fonction renvoi le point le plus proche non visité du point de départ.

```
point *nearestNeighbour(matrice mIn){

    matrice m = cloneMatrice(mIn);
    int nombreDePoints = getDimensionMatrice(m);
    point *ordreDePassage = malloc(sizeof(point)*nombreDePoints);

    //Insertion du point d'origine
    ordreDePassage[0] = clone(getPointIndice(m, 0));
    markVisited(getPointIndice(m, 0));

    int indicePointActuel = 0;

    //Indice du tableau d'ordre de passage
    int indice = 1;

    //Tant que l'on est pas à la fin de la matrice
    while(indice<=nombreDePoints-1){

        indicePointActuel = PointLePlusProche(indicePointActuel,m);

        markVisited(getPointIndice(m, indicePointActuel));
        //Le point le plus proche du pointActuel est inséré
        ordreDePassage[indice] = clone(getPointIndice(m, indicePointActuel));

        // On passe à l'indice suivant
        indice++;
    }

    //On retourne au point d'origine
    //ordreDePassage[indice] = clone(ordreDePassage[0]);
    detruireMatrice(m);

    return ordreDePassage;
}
```

FIGURE 3.1 – Une heuristique simple

3.2 Prim

L'algorithme de Prim, utilise le principe de Minimum Spanning Tree (MST), et consiste à essayer de connecter tous les points, ou noeuds, en suivant les vertices ayant un coût minimum. Nous considérons le premier point de la matrice comme le premier sommet de l'arbre. Nous initialisons donc le tableau de point visité par ce point, et tant qu'il reste des points non visités on appelle la fonction PointLePlusProche pour chaque point visité, il ne reste plus qu'a comparer les distances des points les plus proches, des points visités. La distance minimale indiquera quel sera le prochain point visité. Nous recherchons le point par parcours de la matrice de point et nous effectuons ce parcours NBPoint fois. Donc la complexité de la fonction Prim est en O(n2) avec n le nombre de points.

```
point* prim(matrice mIn){


   matrice m = cloneMatrice(mIn);
   int dim = getDimensionMatrice(m);
   point* TabVisite = malloc(sizeof(point)*dim);
   int current = 0; //Indice du point courant
   int indicePointVisite[dim];
   int nbPointVisite = 1;
   int tmp = 0;
   int min = 32000;

   //Initatiliser le marquage des points sur non visités
   for(int i = 1;i<dim;i++)
       markNoVisited(getPointIndice(m, i));

   //Point d'origine visité
   markVisited(getPointIndice(m, 0));
   TabVisite[0] = getPointIndice(m,0);
   indicePointVisite[0] = 0;


   //Parcours
   //Tant qu'il reste des points à visité
   while(nbPointVisite < dim){

       //Pour tous les points visités
       //On cherche le point le plus proche de chaque point visité et on garde celui qui sera le plus economique (distance)
       for(int i = 0;i<nbPointVisite;i++){
           tmp = PointLePlusProche(indicePointVisite[i], m);
           if (min > getDistanceIndice(m, indicePointVisite[i], tmp)){
               min = getDistanceIndice(m, indicePointVisite[i], tmp);
               current = tmp;
           }
       }
```

FIGURE 3.2 – Un algorithme d'approximation

```
            markVisited(getPointIndice(m, current));

            //Parcour stocke
            TabVisite[nbPointVisite] = getPointIndice(m,curr

            nbPointVisite++;
            indicePointVisite[nbPointVisite-1] = current;
            min = 32000; //Reinitialisation de min

        }
        //TabVisite[dim] = getPointIndice(m, 0);//Retour a l
        detruireMatrice(m);
        return TabVisite;
    }
```

3.3 Brute Force

L'algorithme de Brute Force trouve la solution du problème de voyageur de commerce en calculant la distance totale de chaque parcours possible, et retourne la solution optimale. Notre fonction Brute Force fonctionne en trouvant chaque permutation possible d'une liste de points (en utilisant du swapping et backtracking), en calculant la distance totale avec la fonction overallDistance, et enfin comparant avec la valeur minimum actuel ; si la valeur est plus petite que cette minimum, alors on prends cette valeur comme nouvel minimum, et copie la permutation actuel des points vers la liste « pOut ». La « vraie » fonction Brute Force, appelé « bruteForceRough » est récursif : il prends donc plusieurs paramètres, certains assez obscurs. Pour faciliter l'utilisation de cette fonction, on a crée une fonction « wrapper » (appelé « bruteForce»), qui prend comme unique paramètre une matrice et retourne une liste de points, et fournit aux fonctions récursives les paramètres nécessaires. Cette solution est lente, de complexité O(n !), et n ?est donc utilisable en pratique que dans des cas où n est très petit.

```
/*fonction brute force*/
void bruteForceRough(matrice m, point *pIn, int i, int n, int *min, point *pOut)
{
    int j;
    if(i == n)
    {
        int d = overallDistance(m, pIn);
        if( d < *min)
        {
            *min = d;
            copyList(pIn, pOut, n);
        }
    }
    else
    {
        for(j = i; j < n; j++)
        {
            swap(pIn, i, j);
            bruteForceRough(m, pIn, i+1, n, min, pOut);
            swap(pIn, i, j);
        }
    }

}
```

FIGURE 3.3 – Un algorithme exact par recherche exhaustive

```
point *bruteForce(matrice m)
{
    int n = getDimensionMatrice(m);

    point pList[n];
    copyList(getTableauPointsMatrice(m), pList, n);

    point *pOut = malloc(sizeof(point) * n);
    //copyList(pList, pOut, n);

    int *min = malloc(sizeof(int));
    *min = overallDistance(m, pList);

    bruteForceRough(m, pList, 0, n, min, pOut);

    //copyList(pList, pOut, n);

    free(min);
    return pOut;
}
```

FIGURE 3.4 – Un algorithme exact par recherche exhaustive

3.4 Branch and Bound

L'algorithme Branch and Bound, comme son nom l'indique, est base deux principes : branching, et bounding (séparation et évaluation).

— Branching : on pourrait représenter l'ensemble des solutions du TSP par un arbre (que ce soit planaire, avec chaque noeud représentant les points qu'on peut visiter, ou binaire, avec un noeud représentant l'inclusion d'un point dans le parcours, et l'autre son exclusion). La branching consistera alors à choisir le meilleur noeud, en fonction d'un critère donné.

— Bounding : c'est avec le bounding qu'on choisit sur quel parcours continuer. Dans notre algorithme, ceci consiste à calculer la « lower bound », c'est-à-dire la moins grande distance possible d'un parcours donné. En combinant ces deux principes, on crée donc un algorithme en partant du même principe que brute force, mais qui, en pratique, diminue fortement le temps de recherche de la solution.

Chapitre 4

Conclusion

Nous avons pu mettre en pratique des savoirs acquis au fils des autres cours, par exemple : l'algorithme de permutation utilise des concepts vus en cours d'algorithme de S3, le découpage modulaire vu en cours de programmation aussi du semestre dernier, et l'utilisation de valgrind et des makefile vu en cours d'EDD ce semestre. Néanmoins, nous conseillons aux voyageurs de commerce de contacter Montgomery Scott au lieu des étudiants de L2, ce serait peut-être plus simple ?