Rapport de Projet: Le voyageur de commerce

Benjamin Saint-Sever, Caitlin Dagg, Manon Pintault

18 avril 2014

Table des matières

Chapitre 1

Introduction

Le but de ce projet est 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 un ville déjà visitée). Dans le cadre de ce projet, l ?ensemble des villes est donnée sous forme de matrice de distance.

Chapitre 2

Vie du projet

2.1 Logiciel de gestion de version

Afin de pouvoir réalisé 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 partagé les tâches chaque semaine. Par exemple, certains travaillaient un algorithme, l ?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 au différents algorithmes pour améliorer ceux-ci et régler quelques soucis rencontrés .

Chapitre 3

Présentation des algorithmes

3.1 Nearest Neighbour

```
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

```
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 1
    detruireMatrice(m);
    return TabVisite;
}
```

3.3 Brute Force

```
/*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

Chapitre 4

Conclusion