TP 3 : Algorithme de Little

Olivier Grunder

L'objectif de ce TP est d'implémenter l'algorithme de Little en C pour résoudre le problème du voyageur de commerce (TSP) et de comparer les performances de différentes méthodes d'optimisation du TSP (solveur/Little principalement)

1 Travail préparatoire

Le fichier « berlin52.tsp » est une instance du problème du voyageur de commerce à 52 villes. Dans un premier temps, nous nous intéresserons qu'au 6 première villes de cette instance (celles qui se trouvent déjà dans le fichier « little.c »).

1. Pour compiler et exécuter le programme, vous pouvez utiliser au choix :
   (a) code : :blocks
   (b) *gcc* en ligne de commande avec le programme « little.c » (la compilation se fait avec la commande « gcc little.c »)

   Le programme doit vous afficher les coordonnées X, Y des 6 premières villes de « berlin52.tsp ».
2. Calculer et afficher la matrice des distances (variable globale « dist ») entre les villes. On posera : dist[i][i] = -1. Dans le programme, les coordonnées de la ville $\boldsymbol{i}$ sont : $\boldsymbol{x_i} = \boldsymbol{coord}[\boldsymbol{i}][\mathbf{0}]$ et $\boldsymbol{y_i} = \boldsymbol{coord}[\boldsymbol{i}][\mathbf{1}]$. Par conséquent la distance de la ville $\boldsymbol{i}$ à la ville $\boldsymbol{j}$ est : $\boldsymbol{dist}[\boldsymbol{i}][\boldsymbol{j}] = \sqrt{(\boldsymbol{x_j} - \boldsymbol{x_i})^{\mathbf{2}} + (\boldsymbol{y_j} - \boldsymbol{y_i})^{\mathbf{2}}}$.
3. Créer une fonction « double build_nearest_neighbour() » qui construit une solution suivant l'heuristique « plus proche voisin » et renvoie l'évaluation de cette solution (c'est à dire la distance du parcours correspondant). L'heuristique « plus proche voisin » consiste à partir d'une ville $\boldsymbol{i}$ à se rendre vers la ville la plus proche et n'ayant pas encore été visitée :
   (a) la première ville visitée est la ville 1
   (b) A l'itération $\boldsymbol{i}$, on choisit la ville la plus proche de la dernière ville choisit (niveau $\boldsymbol{i - 1}$), parmi les villes restantes
   (c) On arrête lorsqu'il ne reste plus de ville disponible

   Ne perdez pas trop de temps à réaliser cette fonction. Au besoin, vous pouvez définir une première solution basique telle que la $\boldsymbol{i^{ième}}$ ville visitée est la ville $\boldsymbol{i}$ dont le code est le suivant :

```
sol[0] = 0 ;
for (i=1; i<NBR_TOWNS; i++)
        sol[i] = i ;
```

2 Algorithme de Little

Ecrire l'algorithme de Little pour trouver la solution optimale du TSP, en écrivant une fonction récursive :

1. La fonction « void little_algorithm(double d0[NBR_TOWNS][NBR_TOWNS], int iteration, double evalNoeudParent) » prendra 3 paramètres : une matrice des distances, l'itération (de 1 jusqu'au nombre de villes) et l'évaluation du sommet parent.
2. Faire apparaitre 1 zero par ligne et par colonne dans la matrice des distances. Modifier la borne minimum du noeud courant en conséquence.
3. Calculez les penalités pour les zéros de la matrice distance et mémoriser la ligne et la colonne du zéro ayant la plus forte pénalité (la pénalité d'un zéro s'obtient en faisant le minimum de sa ligne + le minimum de sa colonne).
   — La ligne de ce zéro correspond à la ville de départ
   — La colonne du zéro correspond à la ville d'arrivée
4. Mettre à jour les 2 tableaux d'entiers « starting_town » et « ending_town » pour mémoriser les segments intéressants du problème (identifiés à l'étape précédente). Par exemple, si en appliquant Little, on s'apercoit à l'itération 3 qu'il est intéressant de partir de la ville 2 et d'aller vers la ville 5, alors on stockera cette information sous la forme :
   starting_town[3] = 2 ;
   ending_town[3] = 5 ;
5. Lorsque le nombre d'itération atteint le nombre de villes du TSP, reconstituer la solution à partir des tableaux « starting_town » et « ending_town » et l'évaluer.
6. Penser également à faire :
   — le test d'arrêt dans le cas où la valeur du noeud courant est plus grande ou égale à la longueur de la meilleure solution trouvée jusqu'à présent.
   — également explorer la branche du non choix starting_town / ending_town, attention dans ce cas il ne faut pas incrémenter le nombre d'itérations, car on ne fait pas de choix explicite...

Attention à la gestion des matrices de distance qui sont modifiées en descendant d'un sommet parent à un sommet fils. Il faut penser à créer des copies des matrices ou reconstituer la matrice modifiée lors des étapes de backtracking, ou de remonter dans l'arborescence.

3 Expérimentation

1. Testez ensuite votre programme sur des jeux de données de dimension faible, et vérifiez que :
   — Berlin52 à 6 villes :
      — borne minimum à la 1ère itération : 1754.406226
      — 1er zéro ayant la pénalité la plus forte : (depart=3 ;arrivee=5)
      — Best solution (2315.15) : 0 1 2 3 5 4
   — Berlin52 à 10 villes :
      — borne minimum à la 1ère itération : 1200.591572
      — 1er zéro ayant la pénalité la plus forte : (depart=1 ;arrivee=6)
      — Best solution (2826.50) : 0 1 6 2 7 8 9 3 5 4
2. Regardez comment évolue le temps de recherche quand la dimension du problème augmente (cf jeu de données “berlin”). On résoudra les problèmes du TSP progressivement de 6 à 52 villes par pas de 5.

3. Comparez les résultats que vous obtenez en terme de solution optimale et de temps de calcul entre votre méthode de Little et le solveur GLPK
4. Essayez d'optimiser l'algorithme pour qu'il trouve la solution plus rapidement, notamment en détectant les sous-tours qui se forment durant l'algorithme de Little et qui ne permettent pas de produire des solutions admissibles (Little+). Comparer à nouveau les temps de calcul entre les 3 méthodes : Solveur GLPK, Little, et Little+
5. Envoyez une archive contenant vos codes sources (programme C Little/Little+ et modèle GMPL) ainsi que votre rapport au format PDF à votre responsable de TP.