Techniques Algorithmiques et Programmation

Voyageur de commerce

On rappelle la définition du problème :

Voyageur de commerce

Instance: Un ensemble $V = \{v_0, \ldots, v_{n-1}\}$ de points et une distance d sur V.
Question: Trouver une tournée de longueur minimum passant par tous les points de V, c'est-à-dire une permutation σ des indices des éléments de V telle que $\sum_{i=0}^{n-1} d(v_{\sigma(i)}, v_{\sigma(i+1 \bmod n)})$ est minimum.

Dans la suite on supposera que $V \subset \mathbb{R}^2$ est un ensemble de n points du plan et que d est la distance euclidienne entre deux points. L'objectif des TP va être de programmer et de tester les performances de plusieurs algorithmes sur certaines instances du problème.

On utilisera la structure de donnée `point` défini par `struct{ double x,y; }`.

Question 1. *Donnez le code* `C` *de la fonction* `dist(A,B)` *donnant la distance entre les points* `A` *et* `B`*.*

On représentera une permutation σ de $\{0, \ldots, n-1\}$ par un tableau `P` de taille n tel que `P[i]` $= \sigma(i)$. Par exemple, `int P[]={3,1,2,0};` représentera une permutation pour $n = 4$ qui inverse le premier avec le dernier élément, ce qui définit la tournée v_3, v_1, v_2, v_0, v_3. Dit autrement, `P` représente l'ordre de visite des points de V dans la tournée.

Question 2. *Écrivez la fonction* `value(point *V,int n,int *P)` *qui renvoie la longueur de la tournée des* `n` *points de* `V` *selon la permutation* `P`*.*

1 Approche « Brute-Force »

Les permutations peuvent être rangées par ordre lexicographique de la plus petite (dans notre exemple `P={0,1,2,3}`) à la plus grande (`P={3,2,1,0}`). On suppose donnée la fonction `bool NextPermutation(int *P,int n)` qui calcule, en mettant à jour `P`, la permutation suivant immédiatement `P` dans l'ordre lexicographique. De plus, la fonction renvoie `false` si et seulement si, à l'appel de la fonction, la permutation `P` correspond à la plus grande permutation. Vous pourrez utiliser, mais c'est pas nécessaire, la constante `DBL_MAX` (définie dans `<float.h>`) qui définit la plus grande valeur pouvant être représentée par une variable de type `double`.

Question 3. *Écrivez la fonction* `double tsp_brute_force(point *V,int n,int *Q)` *qui, à partir d'un ensemble* `V` *de* `n` *points, renvoie la permutation* `Q` *minimisant la longueur de la tournée selon l'approche exhaustive (ou « Brute-Force ») ainsi que la longueur de cette tournée.*

On va maintenant optimiser la procédure précédente en `tsp_brute_force_opt()`. La première optimisation consiste à fixer l'un des points de la permutation, le premier ou le dernier (à voir ce qui est le plus pratique). Cela permet de gagner un facteur n sur le nombre de permutations à tester.

La seconde consiste à arrêter pendant le calcul de `value()` l'évaluation dès que la longueur courante atteint ou dépasse celle de la meilleure tournée déjà obtenue, disons w. Dans cette optimisation, n'oubliez pas le retour au point de départ, c'est-à-dire qu'une tournée partielle utilisant les $i+1$ premiers points $v_{\sigma(0)}, \ldots, v_{\sigma(i)}$ possédera $i+1$ arêtes et aura pour longueur $w_i =$

$\left(\sum_{j=0}^{i-1} d(v_{\sigma(j)}, v_{\sigma(j+1)})\right) + d(v_{\sigma(i)}, v_{\sigma(0)})$. L'observation est que si la longueur de la tournée partielle $w_i \geqslant w$, alors on peut directement passer à la plus grande permutation de préfixe $\sigma(0), \ldots, \sigma(i)$.

Question 4. *L'observation suppose-t-elle l'inégalité triangulaire ?*

Question 5. *Écrivez une fonction* `double value_opt(V,n,P,w)` *qui renvoie :*

- *la longueur de la tournée si elle est plus petite que* `w` *; ou bien*
- `-k` *si* `k` *est le nombre d'arêtes de la première tournée partielle qui dépasse* `w`*.*

(NB : Attention à l'indice `k` *renvoyé ! On peut encore améliorer la fonction en considérant la distance de retour* $d(v_{\sigma(i)}, v_{\sigma(n-1)}) + d(v_{\sigma(n-1)}, v_{\sigma(0)})$ *à la place de* $d(v_{\sigma(i)}, v_{\sigma(0)})$*. Pourquoi ?)*

Question 6. *Écrivez une fonction* `MaxPermutation(P,n,k)` *qui renvoie dans* `P` *la plus grande permutation, dans l'ordre lexicographique, dont le préfixe de taille* `k` *est celui de* `P`*. (Vous pourrez supposer que* `P` *était la plus petite des permutations à posséder ce préfixe là.) Quelle la complexité de votre fonction ? En déduire le code de* `double tsp_brute_force_opt(point *V,int n,int *Q)`*.*

2 En TP

Téléchargez (*Enregistrer la cible du lien sous ...*) les fichiers correspondant au TP à partir de la page de l'UE disponible ci-après, et mettez tout dans le même répertoire :

`http://dept-info.labri.fr/~gavoille/UE-TAP/`

Vous aurez surtout à éditer `tsp_brute_force.c`, à compiler avec `make tsp_main` ou `make -B tsp_main`, et lancer l'exécution avec `./tsp_main`. Implémentez et testez `tsp_brute_force()` puis la version optimisée `tsp_brute_force_opt()`.

Commencez par un petit nombre de points (par défaut $n = 10$) puis testez les diverses optimisations en augmentant n que vous pouvez passer dans la ligne de commande. Pour une animation, ajoutez l'instruction `drawTour(V,n,P);` au cœur de vos fonctions `tsp_brute_force` permettant de visualiser la tournée courante donnée par `P`. Si nécessaire, utilisez la même graine dans le générateur aléatoire, avec `srandom()`, de façon à pouvoir comparer vos expériences (commentez/décommentez des parties du `main()`). Vous devriez observer un gain d'un facteur de 20 à 100 pour les versions optimisées. Pour pouvoir quitter la fenêtre pendant l'exécution, appuyez sur `'q'` mais aussi transformez le `while` de vos fonctions `tsp_brute_force` en `while(NextPermutation(...) && running)`.

Autres optimisations. On peut utiliser la fonction `hypot()` de `math.h` pour simplifier le calcul de la distance (voir l'aide en ligne `man hypot`), mais en terme de temps de calcul le gain n'est pas évident. À tester. Vous pouvez aussi éviter le `%n` dans `value()`, ce qui permet de gagner environ 15%. Ensuite, vous pouvez réécrire `value()` (ou `value_opt()`) de sorte à pré-calculer dans une table `D[][]` toutes les distances plutôt que de faire appel à `dist()`. Il ne faut le faire qu'une fois bien sûr. Votre nouvelle fonction devrait alors ressembler à :

```
static double value(point *V, int n, int *P){
  static double **D=NULL; // initialisé à la compilation
  if(D==NULL){            // ne sera exécuté qu'une seule fois
    D=malloc(n*sizeof(double*));
    for(int i=0;i<n;i++){
      D[i]=malloc(n*sizeof(double));
      for(...) ...;
```

Vous devriez gagner un facteur 2. Comment faire pour libérer la table `D` ?

On pourrait aussi gagner un facteur deux supplémentaire en remarquant que la fonction `tsp_brute_force()` (tout comme la variante `tsp_brute_force()_opt`) visite deux fois trop de tournées. En effet, tourner dans un sens ou dans l'autre fera une tournée de même longueur alors que les deux tournées seront considérées comme différentes par `tsp_brute_force()`. Comment changer la boucle principale pour ne visiter qu'au plus $(n-1)!/2$ tournées ?