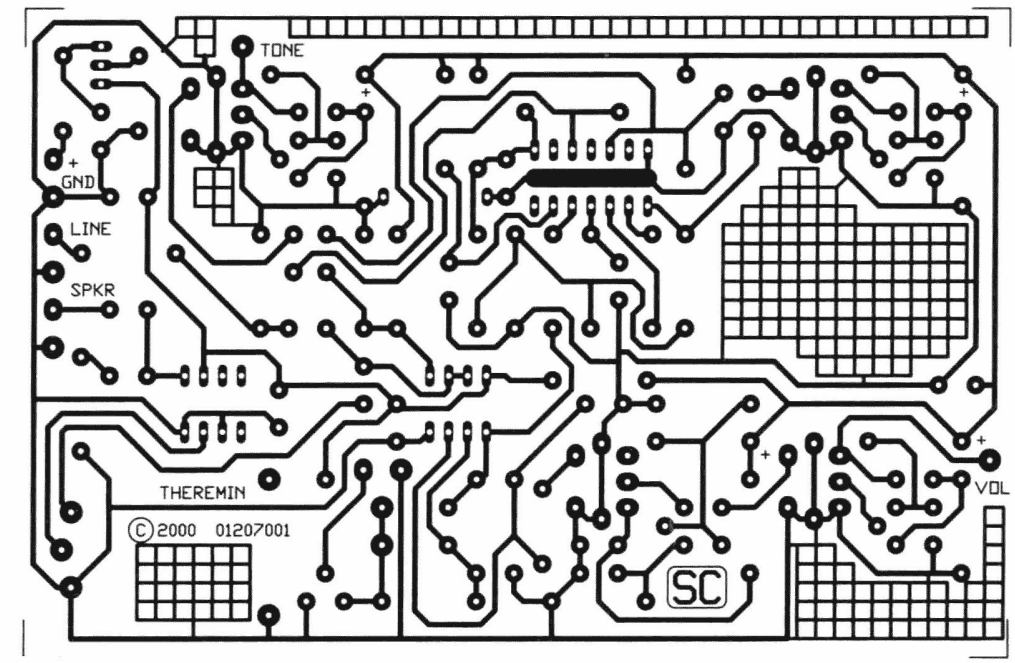
Licence Info. 2e année
Algorithmique & langage C

Projet de fin de semestre 3
Le problème du "voyageur de commerce"

Après une présentation générale du projet et la description du travail à réaliser, vous trouverez la partie intitulée "Cahier des charges, Spécification" qui précise exactement ce que l'on attend de vous. Soyez-y particuliérement attentif !

1 Description du problème au moyen d'une application pratique :

L'industrie de l'électronique utilise des chaînes de fabrications automatisées pour réaliser les cartes électroniques comme les cartes mères d'ordinateurs. Après le gravage du circuit proprement dite il y a une phase de perçage des trous permettant d'enficher les composants, puis de les souder en place. Nous allons nous intéresser à la phase de perçage. Il y a souvent plusieurs centaines de trous à percer par une perceuse automatiques se dépaçant de trou en trou. Les industriels ont cherché à minimiser le temps de déplacement à vide de la tête de perçage (time is money !). Ce temps est lié à la distance séparant les trous. Ce problème est similaire au problème mathématique dit du "voyageur de commerce" (tsp : travelling salesman problem).



1.1 Le problème du "voyageur de commerce"

Etant donnés n points (des " villes ") et les distances les séparant, trouver un chemin de longueur totale minimale qui passe exactement une fois par chaque point et revienne au point de départ (une tournée). En effet, selon l'ordre dans lequel on visite les villes, on ne parcours pas la même distance totale. C'est un problème d'optimisation combinatoire qui consiste à trouver la meilleure solution parmi un ensemble de choix possibles.

Ce problème peut servir tel quel à l'optimisation de trajectoires de machines-outils : par exemple, pour minimiser le temps total que met une fraiseuse à commande numérique pour percer n points dans une plaque de tôle ou pour percer les trous des composants d'un circuit électronique comme dans le cas qui nous intéresse.

Ce problème est plus compliqué qu'il n'y paraît et on ne connaît pas de méthode de résolution permettant d'obtenir des solutions exactes en un temps raisonnable pour de grandes instances (grand nombre de villes) du problème. Pour ces grandes instances, on devra donc souvent se contenter de solutions approchées, car on se retrouve face à une explosion combinatoire : le nombre de chemins possibles passant par 69 villes est déjà un nombre d'une longueur de 100 chiffres. Pour comparaison, un nombre d'une longueur de 80 chiffres permettrait déjà de représenter le nombre d'atomes dans tout l'univers connu.

Le problème du "voyageur de commerce" a été étudié de puis lontemps et on dispose d'une grande varité d'algorithmes donnant le plus souvent des solutions approchées mais calculables en un temps raisonnable.

Remarque : Pour une application de perçage il faudrait supposer que les tournées du problème commence au point 0 de cordonnées $(0, 0)$ qui est la place au repos de la tête de perçage. Nous négligerons ce point pour garder la généralité du problème.

1.2 L'algorithme "brute force"

Puisqu'il s'agit de trouver l'ordre de visite des villes minimisant le trajet total, une idée serait de générer toutes les possibilités et de retenir celle qui donne la distance minimale.
Le nombre de solution est le nombre de permutation de l'ensemble des villes, soit $n!$ pour n villes. Par exemple pour 27 villes on a un nombre de permutations à tester de :
10888869450418352160768000000.

A moins d'avoir accès à un super calculateur du top ten mondial [1], un tel algorithme demande des années pour trouver la solution optimale. Il n'est donc pas exploitable pour l'industrie électronique, néanmoins il peut être utile pour trouver la solution optimale d'une instance de petite taille (de l'ordre de 10 à 20 villes) pour mettre au point le code C implémentant les méthodes présentées ci-après. Pour éviter de saturer la mémoire de la machine, nous utiliserons la méthode qui, à partir d'une permutation quelconque, génère la permutation suivante dans l'ordre lexicographique (celui du dictionnaire) et dont nous calculerons la longueur avant de continuer (look for "next permutation" on english Wikipédia).

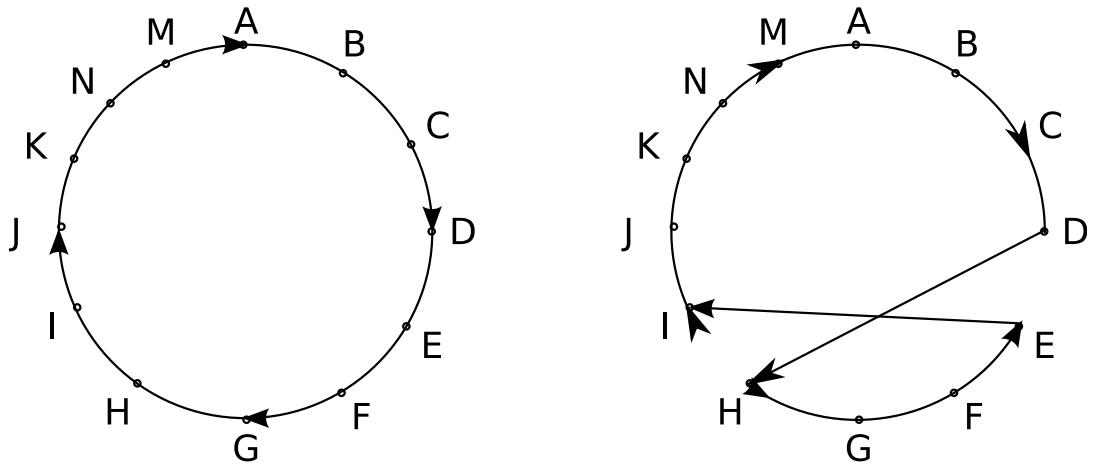
1. http ://www.top500.org/list/2012/06/100

2 Algorithmes de recherche locale

Imaginons que toutes les solutions potentielles soient des points répartis sur le sol. Une métode de recherche locale va partir d'un point (une solution) souvent pris au hasard. Dans notre cas il s'agit de générer une tournée en prenant un chemin aléatoire de ville en ville, ce qu'on appelle le "random walk". Puis on va considérer les points (ie tournées) voisins (selon un critère particulier) et prendre parmis eux celui qui donne la solution la plus intéressante ou seulement une solution meilleure. On recommence le processus à partir de ce nouveau point jusqu'à atteindre une limite de temps ou de nombre de tests fixée à l'avance. Nous allons considérer deux variantes simples d'une telle méthode.

2.1 La 2-optimisation

La 2-optimisation (ou 2-opt) est une méthode pour trouver un voisin d'une tournée. Elle consiste à éliminer deux trajets (arêtes) non consécutifs dans la tournée et reconnecter la tournée d'une manière différente.



Supprimons les trajets DE et HI. On reconnecte D avec H et E avec I.

2.2 Recherche locale random

La méthode consiste à calculer une tournée de départ en choisissant les villes aux hasard (phase d'initialisation). Puis on choisit au hasard également deux trajets non consécutifs et on fait une 2-opt. On garde la nouvelle tournée si elle est meilleure. Le processus et répété à tartir de la tournée précédente un nombre de fois fixé à l'avance.

2.3 Recherche locale systématique

Pour chaque ville d'une tournée de départ choisie au hasard, faire une 2-opt avec toutes les autres villes non adjacentes. Garder la meilleure tournée de l'ensemble. Répéter ces étapes un nombre de fois fixé à l'avance et garde la meilleure tournée de tout le processus.

3 Algorithmes génétiques

Les algorithmes génétiques s'inspirent de la biologie et calculent des solutions approchées.

3.1 Principe général de la méthode

- créer une population initiale de N individus (tournées) initialisées au hasard (random walk);
repeat
  while *le nombre de croisement n'est pas atteint* **do**
    - sélectionner au hasard deux individus;
    - faire un croisement qui donne une tournée fille;
    - avec une probabilité p faire muter la fille;
    - remplacer un individu de la population par la fille (au hasard ou le moins performant);
  end
until *le nombre fixé de générations ou la stabilité est atteint*;

Paramètres possibles :
Nombre d'individus : 20
Nombre de croisements par génération : de 1 à la moitié de la population.
Taux de mutation : 0.3
Nombre de générations : 200

Bien sûr l'algorithme présenté n'est pas le plus performant mais il permet de bien comprendre la méthode.

3.2 Le croisement DPX (distance preserving crossover

Etant donné deux tournée dites parent-1 et parent-2, on initialise la tournée fille en copiant le parent-1. Puis tous les trajets qui ne sont pas en commun avec le parent-2 sont détruites. Les morceaux déconnectés de tournée sont recombinées par la méthode suivante : si le trajet (i, j) a été détruit, alors si k est le plus proche voisin de i parmis les extrémités des autres fragments, on ajoute le trajet (i, k) (si ce trajet n'est contenu dans aucun des deux parents) et tous les trajets du fragment de k en le renversant si nécessaire.

Exemple :
Avec $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ l'ensemble des villes :

Parent 1 = (5 3 9 1 2 8 0 6 7 4)
Parent 2 = (1 2 5 3 9 4 8 6 0 7)

Fille = (5 3 9)(1 2)(8)(0 6)(7)(4)

Trajets détruits : (9,1), (2,8), (8,0), (6,7), (7,4)

Après reconnexion, Fille = (5 3 9 8 7 2 1 4 6 0)

Avec ppv(9)=8, ppv(8)=7, ppv(7)=2, ppv(1)=4, ppv(4)=6 et ppv = plus proche voisin.

3.3 La mutation

Nous nous contenterons, comme mutation, de faire une 2-opt afin de simplifier un peu la mise en oeuvre de cette méthode. Aucun contrôle ne sera effectué sur la tournée obtenue.

4 Travail à réaliser

Les données du problème seront lues dans un fichier. Ces fichiers viennent de la bibliothèque de problèmes TSPLIB. Le format (simplifié) des fichiers est donné en annexe. Les structures de données seront des tableaux statiques déclarés avec une constante de taille maximale et encapsulés dans des `struct` avec leur dimension de travail. On pourra utiliser des types numériques `long` et `double` éventuellement en `unsigned` si cela vous semble utile.

Si vous avez une certaine maîtrise du C, vous pouvez utiliser des tableaux et structures de données dynamiques. Ce plus sera "pour la gloire" car cela n'est pas prévu dans la notation.

Pour contrôler le programme on utilisera les balises suivantes :
-v : (verbose) le programme affiche des informations intermédiaires (tournée et longueur à chaque étape) sinon il n'affiche que le résultat final.
-f <nom de fichier d'entrée>
-bf : déclanche le brute force
-lsr <nombre d'essais> : déclanche la recherche locale random avec le nombre d'essais indiqué
-lsnr <nombre d'essais> : déclanche la recherche locale systématique avec le nombre d'essais indiqué.
-ga <nombre de générations> <taux de mutation> : déclanche l'algorithme génétique avec les paramètres indiqués.

Il y aura un contrôle des erreurs possibles.

Le programme à écrire comporte plusieurs volets :
- **détection des balises :** voir ci-dessus.
- **lecture de l'instance dans un fichier :** (voir annexe).
- **structures de données .**
- **algorithmes de calcul :**
  - **brute force** recherche exhaustive pour trouver la solution optimale. On limitera cette partie à des instances de moins de 10 villes.
  - **recherches locales** random et systématique.
  - **algorithmes génétiques**

5 Cahier des charges, spécification

6 Notation

La notation se fait en deux étapes :

- Etape 1 : Catégories de programmes :
  Le programme que vous allez rendre sera classé dans l'une des catégories ci-dessous et sera noté par rapport à la note maximale associée.

  – **Version basique : 12 points maximum**

    Analyse syntaxique de la ligne de commande : on détecte les balises saisies et on s'assure que l'appel est correct. Si c'est le cas, on lance le calcul correspondant.
    Lors de l'analyse syntaxique, on écrira **obligatoirement** un sous-programme qui pour un tableau t de chaînes de caractères et une chaîne de caractères ch renvoit un entier égal à :
    - la position de ch dans t (i.e l'indice de la case contenant ch)
    - -1 si ch n'est pas présente dans t.

    Ce sous-programme doit être spécifié et prouvé à l'aide de la technique vue en cours. La spécification et la preuve seront fournies dans un fichier au format pdf.
    – Structure de donnée pour les instances : nom, nombre de villes, coordonées des points, matrice des distances.
    – Structure de données pour une tournée : nombre de ville, longueur, liste des villes constituant la tournée (dans l'ordre). Les villes auront pour nom le numéro correspondant dans le fichier.
    – Sous programme renvoyant une tournée selon le "random walk".
    – Sous programme d'affichage d'une instance.
    – Sous programme d'affichage d'une tournée : liste des villes et longueur totale.
    – Sous programme de calcul des distances et initialisation de la matrice. Cette matrice est symétrique puisque `distance(i,j)=distance(j,i)`, et a une diagonale principale nulle (`distance(i,i)=0`), aussi on ne stockera que les valeurs utiles dans un tableau linéaire. Les distances entre i et j seront trouvées à l'aide de la formule adhéquate.
    – Sous programme d'affichage de la matrice des distances, sous sa forme réelle (tableau linéaire), et sous sa forme matricielle carrée.
    – algorithme "brute force" qui utilisera un sous programme de calcul de la permutation suivante (Wikipédia est votre ami).

  – **Version intermédiaire : 16 points maximum**

    On ajoute :
    – sous programme 2-opt.
    – recherche locale "random".
    – recherche locale systématique.

  – **Version complète : 20 points maximum**

– sous programme DPX. On pourra avoir besoin d'une structure d'ensemble avec les actions : appartient, ajouter, supprimer, vide, pour stocker et traiter les sommets candidats.
– sous programme mutation qui appelle 2-opt.
– algorithme génétique.

Attention : Pour obtenir les points relatifs à une version, il faut que les versions inférieures aient été traitées dans leur intégralité.

- Etape 2 : Détermination de la note finale :
  Trois critères guideront la notation dans chaque catégorie :

  1. Respect du cahier des charges : 20% de la note finale
     Tout manquement à la spécification présentée ci-avant diminuera votre note de 10%. En effet la validité de votre programme sera testée automatiquement ; si tout fonctionne correctement, parfait ! si tel n'est pas le cas nous devrons regarder en détails pourquoi, et même si votre programme effectue les opérations requises, votre note sera diminuée.
  2. Compilation, exécution : 50% de la note finale
     Le programme compile sans warning et s'exécute sans problèmes.

  3. Qualité du code : 30% de la note finale
     Seront pris en compte :
     · la pertinence des types utilisés,
     · l'efficacité des algorithmes,
     · la simplicité, l'efficacité du code C,
     · la lisibilité du code : indentation, identificateurs explicites,
     · le respect des règles de codage,
     · la pertinence des commentaires : en particulier chaque déclaration de fonction sera précédée d'un cartouche en trois parties :
       Description des paramètres du sous programme avec les mentions : in, out, ou in-out.
       Descrition de l'objet du sous programme.
       Description des résultats : valeur retournée et paramètres de sortie.
     · la modularité : pertinence du découpage en modules .h

7 Divers

7.1 Exemples

- invocation de divers algorithmes :
  ```
  ./tsp -v -f ../LIB/essai.txt -bf
  ./tsp -f ../LIB/essai.txt -lsr 200
  ./tsp -v -f ../LIB/essai.txt -ga 300
  ./tsp -v -f ../LIB/essai.txt -bf -lsnr 200 -ga 300
  ```
  Remarque : les résultats pouront être sauvegardés dans un fichier en redirigeant la commande Linux : `./tsp -f ../LIB/essai.txt -bf -lsnr 200 >resultat.txt`

- Messages d'erreur :
  On distinguera 3 types d'erreur :
  Syntaxe de la ligne de commande (nombre d'arguments, erreurs de balise,...) :
  `./tsp` $--> $ ERROR : command line
  `./tsp essai.txt` $--> $ ERROR : balise -f non trouvée
  Fichiers inexistants : `./tsp -f esssai.txt` $--> $ ERROR : fichier non trouvé
  Erreur de paramètre :
  `./tsp -f essai.txt -lsr` $--> $ ERROR : -lsr paramètre manquant
  `./tsp -f essai.txt -lsr -v` $--> $ ERROR : -lsr paramètre non valide

7.2 Codage en C

7.2.1 Détermination de la fin d'un fichier

Pour détecter la fin du fichier à traiter, évitez de détecter directement le caractére EOF. Utilisez plutôt la fonction `feof()` de signature : int feof(FILE* filep) ;
`feof` retourne une valeur différente de zéro si la tête de lecture du fichier référencé par filep¿ est arrivée à la fin du fichier sinon la valeur du résultat est zéro.
`filep` est un pointeur du type `FILE*` qui est relié au nom du fichier à lire.

7.2.2 Déboggage

Les programmes de calculs numériques sont assez fourbes. Il semblent fonctionner mais si on y regarde de près, ils font souvent des calculs faux. Il est donc conseillé d'y regarder de près dès le début en faisant des test méthodiques. Il est même conseillé de garder les tests pour vérifier, en cas de soupçon de bug sur un sous programme, ce qui a été testé et ce qui a été oublié. Dans le même ordre d'idée, il est conseillé d'indiquer les spécifications d'entrée et de sortie de chaque sous programme : intervalle de valeurs, caractéristiques, propriétés.

Remarque : Comme ces algorithmes laissent la place au hasard, il est normal que deux exécutions consécutives du programme avec les mêmes paramètres donnent des résultats différents. Normalement, plus le nombres de répétitions est élevé, meilleure est la solution obtenue. Cependant nous avons donné ici des versions simplifiées qui vont être moins performantes que les versions utilisées en production. Avec les fichiers de tests vous trouverez des fichiers donnant la tournée optimale afin d'évaluer les performances de vos programmes.

A Annexe : la structure des fichiers de données

Nous utiliserons des fichiers de données provenant de la bibliothèque de problèmes TSPLIB. Des fichiers ont été sélectionnés (et simplifiés) pour ce projet. Ils ne représentent pas forcément des problèmes de perçage mais cela n'a aucune importance.
Les fichiers sont des fichiers textes (.txt) et comportent un certain nombres de mots clé. Il nous suffit de détecter les mots clé suivants :

– `NAME` suivit du nom du problème (chaîne alphanumérique),

– `DIMENSION` suivi d'un entier donnant le nombre de villes,

– `DISPLAY_DATA_SECTION` ou `NODE_COORD_SECTION` qui annoncent les lignes de coordonnées des points :

  `<numéro de la ville> <abscisse> <ordonnée>`

  Les numéros de villes sont des entiers et commencent à 1, les coordonnées sont des entiers ou des flottants éventuellement en notation exponentielle.

  Bien sûr il y a autant de lignes que de points. Comme nous sommes dans un fichier texte, les nombres entiers et flottants seront lus caractère par caractère ou sous forme de chaînes et transformés en nombres au moyen de la fonction `atoi` de `string.h`.

– `EOF` **Attention** : il s'agit là d'un mot clé annonçant la fin du fichier et non du caractère `eof`.

Il peut y avoir d'autres mots clés et données dans le fichier. Il suffit de les ignorer car nous ne nous en servirons pas pour ce projet.

Exemple :

```
NAME: essai
TYPE: TSP
DIMENSION: 8
DISPLAY_DATA_SECTION
   1    1150.0  1760.0
   2     630.0  1660.0
   3      40.0  2090.0
   4     750.0  1100.0
   5     750.0  2030.0
   6    1030.0  2070.0
   7    1650.0   650.0
   8    1490.0  1630.0
EOF
```

Sujet proposé par Vincent Dugat