ISIMA
INSTITUT D'INFORMATIQUE
Université Clermont Auvergne

Compte rendu : Expression parenthésées - The Travelling Salesman - Lumen

Lilian Hiault

29 avril 2019

Table des matières

1 Expressions parenthésées

Introduction

« Expressions parenthésées »est un problème posé sur CodinGame `https://www.codingame.com/training/easy/brackets-extreme-edition`.
On doit vérifier qu'une expression est correctement parenthésée c'est à dire si toutes les parenthèses, crochets, accolades sont ouverts puis fermés dans le bon ordre.

1.1 Vérifier l'ouverture et fermeture des symboles

Une fois qu'on a récupéré l'exxpression sous forme d'une chaîne de caractère, j'ai créé une variable pour chaque symbole : parenthèse, crochet et accolade. Il faut ensuite vérifier chaque caractère un à un et tester si il appartient aux symboles voulus. Si c'est le cas et qu'il s'ouvre alors on incrémente la variable correspondant au symbole. Si c'est une fermeture alors on décrémente la variable.

```
int main()
{
    char expression[2049];
    scanf("%s", expression);
    int parenthese = 0;
    int crochet = 0;
    int accolade = 0;
    char caractere = expression[0];
    int compt = 0;
    int bienParenthesee = 1;
    while (bienParenthesee && caractere != '\0')
    {
      caractere = expression[compt];
      if (caractere == '(')
      {
        parenthese ++;
      }
      else if (caractere == ')')
      {
        parenthese --;
      }
      else if (caractere == '[')
      {
        crochet ++;
      }
      else if (caractere == ']')
      {
        crochet --;
      }
      else if (caractere == '{')
      {
        accolade ++;
      }
      else if (caractere == '}')
      {
        accolade --;
      }
```

1.2 Ordre d'ouverture

En gardant simplement les compteur il est impossible de savoir dans quel ordre on ouvre ou ferme les parenthèses, «)(»est donc considéré comme bien parenthésé.

Pour vérifier qu'on ne ferme pas les parenthèses avant de les ouvrir, à chaque fois que je modifie mes compteurs je vérifie que mon compteur est positif c'est à dire que je n'ai pas fermé plus de parenthèses qu'il n'y en a d'ouvertes. Si l'expression est mal parenthésée alors le booléen correspondant passe à faux et stope la boucle tant que.
Si les symboles sont ouverts puis fermés dans le bon ordre et qu'il y a autant d'ouvertures que de fermetures, alors l'expression est bien parenthésée.

```
  if ((parenthese < 0) || (crochet < 0) || (accolade < 0))
      {
        bienParenthesee = 0;
      }
      compt ++;
    }
    if (bienParenthesee && (parenthese == 0) && (crochet == 0)
       && (accolade == 0))
    {
      printf("true\n");
    }
    else
    {
      printf("false\n");
    }
    return 0;
}
```

Conclusion

L'idée de faire des compteurs m'est venue assez naturellement pour résoudre ce problème et paraissait assez simple. Néanmoins j'ai eu plus de mal à modéliser les différentes situations sous la forme de compteurs comme par exemple pour l'ordre d'ouverture. Je n'ai pas réussi à prendre en compte les différents symboles imbriqués comme par exemple « {(}) »qui est reconnu comme bien parenthésé par mon programme. Malgré cela, mon code à tout de même obtenu 100% de score sur CodinGame.

2 The Travelling Salesman

Introduction

« The Travelling Salesman »est un problème où on doit trouver le chemin le plus court pour visiter chacune des villes données une fois et revenir à la ville de départ. C'est un problème très complexe à résoudre. Dans cet exercice `https://www.codingame.com/training/easy/the-travelling-salesman-problem` il faut visiter la ville la plus proche jusqu'à avoir visité chaque ville une fois puis revenir à celle de départ.

2.1 Calculer la distance entre deux villes

Pour garder l'emplacement des villes, j'ai créé une structure « Coordonnées »qui contient l'abscisse et l'ordonnée de la ville.

```
typedef struct coordonnees
{
  int x;
  int y;
}coord;
```

Pour calculer la distance on utilise la formule $\sqrt{(ville2.x - ville1.x)^2 + (ville2.y - ville1.y)^2}$

```
float distance(coord ville1, coord ville2)
{
  return sqrt(pow(ville2.x-ville1.x, 2) + pow(ville2.y-ville1.y, 2));
}
```

2.2 Stocker les données

Pour garder les emplacements des villes, je crée un tableau de coordonnées « villes »dans lequel chaque case correspond à l'abscisse et l'ordonnée de la ville.
Je crée ensuite un tableau d'entiers qui me permet de savoir si j'ai déjà ou non visité une ville. Si c'est le cas alors la case de même indice que la ville prendra 1 sinon 0. Il est initialisé à faux.

```
int main()
{
    int N;
    scanf("%d", &N);
    coord * villes = (coord *) malloc(N*sizeof(coord));
    int * dejaVisite = (int *) malloc(N*sizeof(int));
    int k;
    for (k=0; k<N; k++)
    {
      dejaVisite[k] = 0;
    }
    // Stocke les emplacements des villes ans un tableau
    int i;
    for(i = 0; i < N; i++)
    {
        scanf("%d%d", &villes[i].x, &villes[i].y);
    }
```

2.3 Calculer la distance totale

Le nombre de voyage va être équivalent au nombre de villes à visiter. On applique donc une boucle pour N fois. A chaque itération, la ville de destination sera la plus proche de la ville actuelle. On ajoute donc à la distance totale du trajet la distance entre ces deux villes. Une fois le trajet fait, la ville qu'on vient de quitter est ajouté au tableau des villes déjà visitées. Enfin la ville dans laquelle on vient d'arriver devient la ville actuelle.
Une fois sorti de la boucle, on ajoute le dernier trajet qui consiste à revenir au point de départ.

```
    float sommeDistance = 0;
    int villeEnCours = 0;
    int villeArrivee = 0;
    int j;
    for (j=0; j<N; j++)
    {
      villeArrivee = plusProche(villeEnCours, villes, N, dejaVisite);
      sommeDistance += distance(villes[villeEnCours], villes[villeArrivee]);
      dejaVisite[villeEnCours] = 1;
      villeEnCours = villeArrivee;
    }
    sommeDistance += distance(villes[villeEnCours], villes[0]);
    printf("%d\n", (int) round(sommeDistance));
    free(villes);
    free(dejaVisite);
    return 0;
}
```

2.4 La ville la plus proche

Afin de pouvoir éxecuter ce programme il manque une fonction pour trouver la ville la plus proche par rapport à notre position.
La première étape consiste à trouver une ville non visitée et différente du point de départ pour servir à initialiser la distance minimum. Tant qu'on n'a pas trouvé de ville non visitée qui est à une distance différente de 0 de la ville actuelle (donc une ville différente) on continue d'en chercher une. Une fois qu'on a trouvé une ville on garde l'indice du tableau des villes correspondant.

```
int plusProche(int villeDepart, coord villes[], int taille, int dejaVisite[])
{
  int indice;
  int i=0;
  int trouve = 0;
  float distanceEnCours = 0;
  while(i<taille && trouve==0)
  {
    distanceEnCours = distance(villes[villeDepart], villes[i]);
    if ((distanceEnCours!=0) && (dejaVisite[i] == 0))
    {
      indice = i;
      trouve = 1;
    }
    else
      {
i++;
      }
  }
```

On peut maintenant initialiser la distance de la ville la plus proche à la distance entre cette ville et la ville actuelle.
On teste ensuite pour chaque ville différente de la ville actuelle si elle est plus proche que la ville

la plus proche jusque là. Si c'est le cas on garde son indice. Une fois que l'on a testé chaque ville on peut renvoyer l'indice de la ville la plus proche pour en calculer la distance comme illustré dans le main.

```
  float distancePlusProche = distance(villes[villeDepart], villes[indice]);
  int j=0;
  for(j=i; j<taille; j++)
  {
    distanceEnCours = distance(villes[villeDepart], villes[j]);
    if((distanceEnCours!=0) && (distanceEnCours < distancePlusProche))
    {
      indice = j;
      distancePlusProche = distanceEnCours;
    }
  }
  return indice;
}
```

Conclusion

Le plus difficile dans cet exercice a été de trouver comment initialiser la ville la plus proche avant de tester. De plus un problème avec la librairie math.h nécessite de rajouter un argument lors de la compilation « -lm »à positionner avant la sortie. Enfin j'ai eu un dernier problème : lorsque je compile puis éxecute mon prgramme sur ma machine avec les mêmes données que celles de CodinGame je trouve la bon résultat mais le résultat sur le site est différent et n'est donc pas celui attendu. J'ai bien vérifié que le code et les données sont identiques mais j'ai toujours une réponse différente...

3 Lumen

Introduction

Lumen est un problème disponible sur CodinGame `https://www.codingame.com/training/easy/lumen`. Des bougies sont disposées dans une pièce dont on nous donne les dimensions. Les bougies peuvent avoir une intensité différente et la lumière se propage autour d'elles en fonction de son intensité. Il faut trouver les zones d'ombres dans la pièce.

3.1 Trouver les zones d'ombre

On crée un tableau d'entiers en deux dimensions dans laquelle les nombres représenterons la luminosité locale dans la pièce. On initialise chaque case à 0.

```
int main()
{
    int taille;
    scanf("%d", &taille);
    int intensite;
    scanf("%d", &intensite); fgetc(stdin);
    int* piece[taille];
    int i;
    for(i=0; i<taille; i++)
    {
     piece[i] = (int *) malloc(taille*sizeof(int));
    }
    for(i=0; i<taille; i++)
      {
       for(j=0; j<taille; j++)
        {
           piece[i][j] = 0;
         }
         }
```

Pour chaque ligne qu'on reçoit on initialise une ligne de notre tableau d'entier : chaque « C »est positionné dans le tableau avec la valeur de l'intensité d'une bougie.
Suite à cela on éclaire les zones autour des bougies en baissant petit à petit l'intensité de la lumière.
Enfin on cherche le nombre de zones d'ombres.

```
    for (i = 0; i < taille; i++) {
        char ligne[501];
        fgets(ligne, 501, stdin);
        initialiseLigne(ligne, i, piece, taille, intensite);
        }
    eclairer(intensite, piece, taille);
    int nbOmbre = ombre(piece, taille);
    printf("%d\n", nbOmbre);
    return 0;
}
```

3.2 Initialiser le tableau avec les bougies

On parcourt la chaîne de caractère entrée deux à deux pour ne pas parcourir les espaces entre chaque caractère. Si un caractères est un « C »alors on positionne aux même endroit sur le plan de la pièce une bougie qui correspond à la valeur de son intensité.

```
void initialiseLigne(char ligne[], int numLigne, int** piece, int taille, int intensite)
{
  int i;
  for(i=0; i<taille*2-1; i+=2)
    {
      if(ligne[i] == 'C')
{
  piece[numLigne][i/2] = intensite;
}
    }
}
```

3.3 Eclairer l'espace autour d'une bougie

On initialise la lumière que l'on cherche avec la valeur de l'intensité d'une bougie. On parcourt chaque case de la pièce dont la valeur vaut 0 (elle n'est pas éclairée). Si elle est dans le voisinage d'une lumière (donc d'une bougie à la première éxecution) alors la case prend la valeur de la lumière-1. Si la valeur de luminosité est différente de 0 alors il y a une lumière dans son entourage. Puis on recommence mais avec une valeur de lumière décrémentée.

```
void eclairer(int intensite, int** piece, int taille)
{
  int lumiere = intensite;
  int i,j;
  while(lumiere > 1)
    {
      for(i=0; i<taille; i++)
{
  for(j=0; j<taille; j++)
    {
   if((piece[i][j] == 0) && (luminosite(lumiere, piece, taille, i, j) != 0))
     {
       piece[i][j] = lumiere-1;
     }
 }
}
      lumiere --;
    }
}
```

3.4 Connaître la luminosité autour d'une case

Pour savoir si une des cases autour d'une cible est d'une luminosité choisie, on teste pour chaque case dans son voisinage si elle est toujours dans le tableau et si sa valeur et celle de la lumière qu'on recherche. Si c'est le cas alors on incrémente un compteur. Si le compteur est non nul c'est qu'il y a une lumière dans sonr voisinage.

```
int luminosite(int lumiere, int** piece, int taille, int i, int j)
// On regarde s'il y a une source de lumière sur les cases voisines
{
  int somme=0;
  if((i>=1) && (piece[i-1][j] == lumiere)){
    somme ++;
  }
  if(i+1<taille && piece[i+1][j] == lumiere){
    somme ++ ;
  }
  if(j-1>=0 &&  piece[i][j-1] == lumiere){
    somme ++;
  }
  if(j+1<taille && piece[i][j+1] == lumiere){
    somme ++;
  }
  if((i-1>=0) && (j+1<taille) && piece[i-1][j+1] == lumiere){
    somme ++;
  }
  if((i-1>=0) && (j-1>=0) && piece[i-1][j-1] == lumiere){
    somme ++;
  }
  if((i+1<taille) && (j+1<taille) && piece[i+1][j+1] == lumiere){
    somme ++;
  }
  if((i+1<taille) && (j-1>=0) &&  piece[i+1][j-1] == lumiere){
    somme ++;
  }
  return somme;
}
```

3.5 Compter les zones d'ombre

Pour chaque case du tableau j'ai testé si sa valeur était 0. Si oui alors j'incrémente le nombre de zones d'ombre.

```
int ombre(int** piece, int taille)
{
  int somme = 0;
  int i,j;
  for(i=0; i<taille; i++)
    {
      for(j=0; j<taille; j++)
{
  if(piece[i][j] == 0)
  {
    somme ++;
  }
}
    }
  return somme;
}
```

Conclusion

J'ai eu beaucoup de mal à éclairer les zones autour de la bougie. J'ai d'abord essayé de prendre une bougie comme point de départ et d'affecter directement les valeurs aux cases autour mais c'était trop compliqué. Puis j'ai essayé de faire un algorithme récursif en partant une fois de plus des bougies mais sans succès. J'ai finalement opté pour une solution plus simple : changer la valeur d'éclairage des cases dans l'ombre par rapport aux cases voisines.