Mathématiques discrètes - TP1

Le problème du voyageur de commerce

par **Merlin Nimier-David & Robin Ricard**, Binôme **B3145**

1. Résolution exacte

```
// Partie commune

void pvc_exact_recursif(int nbVilles, double ** distances, t_cycle * chemin, t_cycle * meilleur, bo
{
  if (chemin->poids > meilleur->poids && bb)
  {
    return;
  }
  else if (chemin->taille == nbVilles)
  {
    double trajetRetour = distances[chemin->c[chemin->taille -1]][0];
    if (chemin->poids + trajetRetour < meilleur->poids)
    {
      chemin->poids += trajetRetour;
      recopier_chemin(chemin, meilleur);
      chemin->poids -= trajetRetour;
    }
  }
  else
  {
    for(int i = 0; i < nbVilles; ++i)
    {
      if(!est_dans_chemin(chemin, i))
      {
        double trajet = distances[chemin->c[chemin->taille - 1]][i];
        chemin->taille++;
        chemin->poids += trajet;
        chemin->c[chemin->taille-1] = i;

        // Appel récursif
        pvc_exact_recursif(nbVilles, distances, chemin, meilleur, bb);

        // "Dépiler" la dernière ville parcourue pour commencer une nouvelle branche
        chemin->taille--;
        chemin->poids -= trajet;
      }
    }
  }
}
```

1. PVC exact naif

```
t_cycle pvc_exact(int nbVilles, double ** distances)
{
```

```
    t_cycle meilleur, courant;
    // Solution actuelle : un chemin de poids infini
    meilleur.taille = 0;
    meilleur.poids = DOUBLE_MAX;
    // Par convention, on commence par la ville 0
    courant.taille = 1;
    courant.poids = 0;
    courant.c[0] = 0;
    pvc_exact_recursif(nbVilles, distances, &courant, &meilleur, false);
    return meilleur;
}
```

2. PVC exact Branch & Bound

```
t_cycle pvc_exact_branch_and_bound(int nbVilles, double ** distances)
{
    t_cycle meilleur, courant;
    // Solution actuelle : un chemin de poids infini
    meilleur.taille = 0;
    meilleur.poids = DOUBLE_MAX;
    // Par convention, on commence par la ville 0
    courant.taille = 1;
    courant.poids = 0;
    courant.c[0] = 0;
    pvc_exact_recursif(nbVilles, distances, &courant, &meilleur, true);
    return meilleur;
}
```

3. Temps d'execution

Noeuds	Exact naif	Branch & bound
5	9285ns	15459ns
10	152ms	30ms
12	20s	468ms

2. Approximation : plus proches voisins

1. PVC approche PPV

```
t_cycle pvc_approche_ppv(int nbVilles, double ** distances)
{
  // Par convention, on commence par la ville 0
  t_cycle courant;
  courant.poids = 0;
  courant.c[0] = 0;

  // Choix glouton : on choisit toujours la ville la plus proche
  for (courant.taille = 1; courant.taille <= nbVilles; ++courant.taille)
  {
    int ville;
    double d, trajet = DOUBLE_MAX;
    // Trouver la ville la plus proche qui n'est pas dans le chemin
    for (int j = 1; j < nbVilles; ++j) {
      d = distances[courant.c[courant.taille-1]][j];
```

```
      if (d < trajet && !est_dans_chemin(&courant, j)) {
        ville = j;
        trajet = d;
      }
    }

    if(trajet != DOUBLE_MAX)
      courant.poids += trajet;
    courant.c[courant.taille] = ville;
  }

  courant.poids += distances[0][courant.c[courant.taille -1]];
  return courant;
}
```

2. Complexité

On a un algorithme contenant deux boucles imbriquées parcourant chacune les `n` noeuds. L'appel à `est_dans_chemin` a une complexité linéaire en la taille du chemin, qui est majorée par le nombre de villes. On retrouve une complexité en `O(n^3)`.

3. Approximation : Algorithme Polynomial

1. Calcul de l'arbre couvrant minimum par l'algorithme de Kruskal

```
void minimum_spanning_tree(int n, double ** edges, int * parents)
{
  // Initialisation
  unsigned int height[n];
  for (unsigned int i = 0; i < n; ++i) {
    parents[i] = -1;
    height[i] = 0;
  }

  // Examiner chaque arrête dans l'ordre de coût croissant
  unsigned int i, j, rootI, rootJ;
  unsigned int h = 0, size = 1;
  while (size < n)
  {
    i = edges[h][0];
    j = edges[h][1];
    rootI = get_root(parents, i);
    rootJ = get_root(parents, j);
    // S'il s'agit d'une arrête reliant deux arbres non-reliés
    if (rootI != rootJ)
    {
      // Relier les arbres
      if (height[rootI] > height[rootJ]) {
        // Placer le sous-arbre J sous i
        parents[rootJ] = i;
      }
```

```
      else {
        // Placer le sous-arbre I sous j
        parents[rootI] = j;
        if (height[rootI] == height[rootJ])
          height[rootJ]++;
      }

      size++;
    }
    h++;
  }
}
```

2. Conversion de l'ACM en cycle euclidien

À partir d'un ACM, il suffit de double chaque arrête pour obtenir un cycle euclidien. Les villes peuvent être répétées jusqu'à `N` fois, mais elles sont visitées dans l'ordre dicté par l'ACM.

3. Conversion du cycle Euclidien en cycle hamiltonien

Il suffit, en conservant l'ordre, de supprimer tous les nœuds ayant déjà été parcourus.

```
t_cycle pvc_mst_hamiltonian(int n, double ** distances)
{
  t_cycle euclidian = pvc_mst_euclidian(n, distances);
  t_cycle hamiltonian;
  hamiltonian.c[0] = euclidian.c[0];
  hamiltonian.taille = 1;

  unsigned char seen[n];
  for (unsigned int i = 0; i < n; ++i)
    seen[i] = 0;

  seen[hamiltonian.c[0]] = 1;
  unsigned int k = 0;
  double delta;
  while (hamiltonian.taille < n)
  {
    if (!seen[euclidian.c[k]]) {
      hamiltonian.c[hamiltonian.taille] = euclidian.c[k];
      hamiltonian.taille++;

      delta = distances[hamiltonian.c[hamiltonian.taille-1]][hamiltonian.c[hamiltonian.taille]];
      hamiltonian.poids += delta;

      seen[hamiltonian.c[hamiltonian.taille-1]] = 1;
    }
    k++;
  }

  return hamiltonian;
}
```

4. Relation entre `L(A)` et `L(H)`

Notre graphe satisfait l'inégalité triangulaire : pour toutes villes `a, b, c`, on a :

`d(a,c) < d(a,b) + d(b,c)`

Ainsi, lors de l'étape précédente, en supprimant des doublons, on ne peut qu'avoir raccourci le poids du cycle. Pour A le cycle Euclidien et H le cycle hamiltonien, on obtient :

`L(H) <= L(A)`

5. Relation entre `L(A)` et `L(Opt)`

Si on considère un cycle hamiltonien de poids minimum noté `Opt`, on a la relation :

```
L(Opt) <= L(A)
// TODO: trouver une majoration plus serrée
```

6. Résultat obtenu sur les données

Pour le problème de test de `n = 250` villes, nous obtenons un cycle de poids `d = 18.2`. Ce résultat étant moins bon que l'approche des plus proche voisin, il semblerait que notre algorithme souffre d'un problème d'implémentation. L'algorithme de calcul de l'arbre couvrant pose probablement problème.

4. Approximation : Optimisation locale

1. Démonstration : `dcycle` est une distance

`dcycle` est une distance car :

`dcycle(a,b) = dcycle(b,a)` (symétrie)
`dcycle(a,b) = 0` implique que `a` et `b` aient `n` arrêtes en commun. Or le problème est de taille `n` donc les cycles ont au maximum `n` arrêtes. Donc `a = b`, `dcycle` respecte la propriété de séparation.
`dcycle` respecte également l'inégalité triangulaire (à démontrer)

2. Cardinal de `C(a, 2)`

`C(a0, 2)` dénombre les cycles hamiltoniens de taille 5 contenant le nœud `a`. On distingue deux cycles hamiltoniens : le premier avec deux arrêtes croisées et le second sans arrêtes croisées.

3. Cycle Opt 2

```
// Note : Plutôt que de partir du centre a0, on définit juste un index de départ,
// ce qui reste plus simple pour écrire le "désentrelacement".
t_cycle two_opt(int idx, int nbVilles, double ** distances, t_cycle cycle)
{
  // on mesure le gain en entralacant/désentrelacant
  double gain = (
      distances[cycle.c[idx]][cycle.c[idx + 1]] +
```

```
      distances[cycle.c[idx + 2]][cycle.c[idx + 3]]
    ) - (
      distances[cycle.c[idx]][cycle.c[idx + 2]] +
      distances[cycle.c[idx + 1]][cycle.c[idx + 3]]
    );
  if(gain > 0) { // on applique l'opération si le gain est positif
    int buffer = cycle.c[idx + 1];
    cycle.c[idx + 1] = cycle.c[idx + 2];
    cycle.c[idx + 2] = buffer;
    cycle.poids -= gain;
  }
  return cycle;
}
```

4. Utilisation sur les données

```
t_cycle opt_cycle(int nbVilles, double ** distances, t_cycle cycle) {
  if(nbVilles > 4) { // il nous faut assez de villes
    for(int i = 0; i < nbVilles - 5; i ++) {
      cycle = two_opt(i, nbVilles, distances, cycle);
    }
  }
  return cycle;
}
```

5. Analyse

1. Mesures

Approche	Noeuds	Temps d'execution	Poids
Naif	12	20s	3.31
Branch & Bound	12	475ms	3.31
Branch & Bound	15	45s	3.37
PPV	15	0.39ms	4.22
PPV	250	2.07ms	14.73
PPV + Opt2	250	2.21ms	14.56
Minimum spanning tree	250	3.8ms	18.98
Minimum spanning tree + Opt2	250	4.01ms	18.18

2. Conclusion

ACM serait le meilleur algorithme présenté, à la fois rapide, peu consommateur de mémoire et relativement précis, il est le plus efficace pour résoudre ce genre de problème. Cependant, notre implémentation comportant certainement une erreur, nous n'avons pas obtenu les résultats attendus.

Dans tous les cas, pour n'importe quel algorithme de résolution par approximation, il est toujours intéressant d'ajouter une phase d'optimisation qui coûte très peu pour des améliorations non négligeables au vu du temps passé à optimiser.