AAIA TP 3IF - PageRank

Florian Rascoussier Romain Fontaine Christine Solnon

5 février 2024

Introduction

Dans ce TP, nous nous intéresseront à un problème bien connu en Informatique, et à une famille d'algorithme non moins célèbre pour s'y attaquer.

0.1 Le Problème du Voyageur de Commerce (TSP)

Le Travelling Salesman Problem (TSP) ou Problème du Voyageur de Commerce est un problème fondamental en optimisation combinatoire et en théorie des graphes.

Le problème se formule de la manière suivante :

> Étant donné une liste de villes et les distances entre chaque paire de villes, quel est le chemin le plus court possible qui visite chaque ville exactement une fois et revient à la ville d'origine ?

Ce problème est notoire non seulement pour sa simplicité conceptuelle, mais aussi pour sa difficulté computationnelle, car il appartient à la classe des problèmes NP-complets. Cela signifie que, pour un grand nombre de villes, il est extrêmement difficile de calculer la solution exacte en un temps raisonnable. Le TSP a des applications pratiques dans de nombreux domaines tels que la planification d'itinéraires, la logistique et même la conception de circuits électroniques.

0.2 L'Algorithme Branch & Bound

Parmi les stratégies algorithmiques développées pour traiter le TSP, l'algorithme Branch & Bound se distingue par son efficacité. Cette méthode adopte une approche systématique pour examiner l'ensemble des solutions potentielles, tout en éliminant progressivement les options non viables. L'algorithme se décompose en deux étapes clés :

- Branching (Branche) : Fractionnement du problème en sous-problèmes plus petits.
- Bounding (Limite) : Évaluation des limites inférieures et supérieures des solutions possibles dans chaque sous-problème pour élaguer les branches non prometteuses.

0.3 Objectifs du TP

Ce TP vise à appliquer les principes théoriques de l'algorithme Branch & Bound à une instance concrète du TSP. À travers ce travail, les objectifs pédagogiques sont multiples :

- Explorer les défis inhérents à la résolution de problèmes NP-complets.
- Implémenter une stratégie d'optimisation avancée et analyser son efficacité.
- Évaluer les performances et les limites de l'algorithme Branch & Bound dans le cadre spécifique du TSP.

Ce travail pratique offre ainsi une opportunité d'approfondissement dans le domaine de l'optimisation combinatoire et une expérience pratique significative dans la résolution d'un problème algorithmique classique.

0.4 A savoir avant de commencer

> La première règle du Fight Club est : il est interdit de parler du Fight Club. La deuxième règle du Fight Club est : il est interdit de parler du Fight Club. ref, Film complet

- Avant de passer d'une partie à l'autre, il est **impératif** que les résultats obtenus soient **identiques** à ceux présentés dans le sujet. En cas de différence, **ne pas continuer** et trouver l'erreur.
- Faire bien attention à l'ordre des opérations, surtout au moment d'ajouter du code dans une fonction d'une partie précédente.
- Ne passez pas votre temps à modifier le template du TP. Passez directement à la lecture du sujet et à la programmation. La compréhension des fichiers externes tels que le `Makefile` n'est pas l'objectif du TP.
- N'oublier pas de documenter vos réponses.
- Essayer de respecter les principes du Clean Code et veillez à la compréhensibilité de votre code, commentaires et notes.

Table des Matières

Table des figures

Codes et programmes

1 Partie 1 : compléter `permut`

Soit `n` un entier naturel positif. On considère un graphe de `n` sommets tel que chacun de ses sommets est connecté à tous les autres (graphe complet). On cherche à calculer l'ensemble des circuits hamiltoniens, c'est-à-dire des chemins partant d'un sommet initial, passant par tous les sommets du graphe et retournant sur le sommet de départ. Ci-dessous, un exemple d'ensemble de chemins pour `n = 4`, avec `0` le sommet de départ. Les sommets sont représentés par des nombres, de `0` à `n - 1` :

```
    [0, 1, 3, 2, 0]
    [0, 1, 2, 3, 0]
    [0, 2, 1, 3, 0]
    [0, 2, 3, 1, 0]
    [0, 3, 1, 2, 0]
    [0, 3, 2, 1, 0]
```

Code 1 – Ensemble de permutations possible d'un chemin hamiltonien pour $n = 4$

Par définition, chaque chemin hamiltonien commence et se termine par 0. Naturellement, calculer l'ensemble de ces chemins revient donc à calculer l'ensemble des permutations de l'ensemble $\{e \mid e \in [1, \ldots, n-1]\}$. Or, le nombre de permutations d'un ensemble fini non vide à k éléments est $n!$ (voir cours). Donc pour un graphe de n nœuds, on cherche ses $(n-1)!$ permutations de chemins hamiltonien au départ du nœud 0.

L'algorithme `permut` calcule l'ensemble des permutations de chemins possible. Ci-dessous, une solution possible d'implémentation :

```
/**
* @brief Loop inside the recursive function that computes all
* the permutations of the vertices of the graph.
*
* @param visited Array of visited vertices
* @param nbVisited Number of visited vertices
* @param costVisited Cost of the visited vertices
* @param notVisited Array of unvisited vertices
* @param nbNotVisited Number of unvisited vertices
* @param cost Cost matrix
*/
void permut(int visited[], int nbVisited, int notVisited[], int nbNotVisited) {
    nbCalls++;
    if (nbNotVisited == 0) {
        genTurtleTour(visited, nbVisited);

        // log in terminal
        for (int i = 0; i < nbVisited; i++) {
            printf("%d ", visited[i]);
        }
        printf("0\n");
    }
    for (int i = 0; i < nbNotVisited; i++) {
        // add notVisited[i] to visited
        visited[nbVisited] = notVisited[i];

        // remove notVisited[i] from notVisited
        // we need to swap notVisited[i] with notVisited[nbNotVisited-1] for
        // easy restoration of the array after the recursive call
```

```
        int tmp = notVisited[i];
        notVisited[i] = notVisited[nbNotVisited-1];
        notVisited[nbNotVisited-1] = tmp;

        // recursive call
        permut(visited, nbVisited+1, notVisited, nbNotVisited-1);

        // backtrack
        notVisited[nbNotVisited-1] = notVisited[i];
        notVisited[i] = tmp;
    }
}
```

Code 2 – Implémentation de l'algorithme `permut`

Une fois la fonction `permut` complétée, on peut alors exécuter le programme. Exemple pour $n = 4$:

```
./bin/main 4
0 1 3 2 0
0 1 2 3 0
0 2 1 3 0
0 2 3 1 0
0 3 1 2 0
0 3 2 1 0
n=4 nbCalls=16 time=0.000s
```

Code 3 – Exécution de l'algorithme `permut` pour $n = 4$

2 Partie 2 : calcul de longueur des circuits hamiltoniens

Dans cette étape, on modifie la fonction `createCost` pour qu'elle renvoie un `int**`, la matrice des coûts.

On peut alors fournir cette matrice à `permut` en modifiant ses paramètres. On ajoute alors dans la *scope* du `if` de cette fonction, le code suivant :

```
    // compute and display cost
    int totalCost = 0;
    for (int i = 0; i < nbVisited-1; i++) {
        totalCost += cost[visited[i]][visited[i+1]];
    }
    totalCost += cost[visited[nbVisited-1]][visited[0]];
    printf(" cost: %d\n", totalCost);
```

Code 4 – Ajout de la matrice des coûts dans la fonction `permut`

Ce code permet de calculer et d'afficher la valeur du chemin. On obtient par exemple, pour $n = 4$:

```
[0, 1, 3, 2, 0] cost: 31319
[0, 1, 2, 3, 0] cost: 32786
[0, 2, 1, 3, 0] cost: 34415
[0, 2, 3, 1, 0] cost: 31319
[0, 3, 1, 2, 0] cost: 34415
```

```
[0, 3, 2, 1, 0] cost: 32786
n=4 nbCalls=16 time=0.000s
```

Code 5 – Exécution de l'algorithme `permut` pour $n = 4$ avec calcul de la longueur

3 Partie 3 : Recherche du plus court-circuit hamiltonien

On ajoute une variable globale :

```
int bestCost = INT_MAX; // Best cost found so far
```

Code 6 – Ajout d'une variable globale pour stocker le coût minimum

On modifie ensuite la fonction `permut` pour qu'elle puisse tenir la variable `bestCost` à jour.

```
void permut(
    int visited[], int nbVisited,
    int notVisited[], int nbNotVisited,
    int** cost
) {
    if (nbNotVisited == 0) {
        [...]

        // update best cost
        if (totalCost < bestCost) {
            bestCost = totalCost;
        }
    }
    [...]
}
```

Code 7 – Mise à jour de la variable `bestCost` dans la fonction `permut`

Cette modification simple permet de maintenir à jour la valeur du meilleur coût rencontré. On obtient par exemple, pour $n = 4$:

```
$ ./bin/main 4 -g
[0, 1, 3, 2, 0] cost: 31319
[0, 1, 2, 3, 0] cost: 32786
[0, 2, 1, 3, 0] cost: 34415
[0, 2, 3, 1, 0] cost: 31319
[0, 3, 1, 2, 0] cost: 34415
[0, 3, 2, 1, 0] cost: 32786
best cost: 31319
n=4 nbCalls=16 time=0.000s
```

Code 8 – Exécution de l'algorithme `permut` pour $n = 4$ avec calcul de la longueur et recherche du plus court chemin

```
$ for i in 8 10 12 14; do ./bin/main $i; done
n = 8; bestCost = 53,591; nbCalls = 13,700; time = 0.000s
n = 10; bestCost = 66,393; nbCalls = 986,410; time = 0.007s
n = 12; bestCost = 67,063; nbCalls = 108,505,112; time = 0.430s
n = 14; bestCost = 69,382; nbCalls = 16,926,797,486; time = 66.658s
```

Code 9 – Exécution de l'algorithme `permut` pour différentes valeurs de n.

4 Partie 4 : Propagation de contraintes

Acronymes

AAIA Algorithmique Avancée pour l'Intelligence Artificielle et les graphes. 2

Bibliographie additionnelle

[1] Christine Solnon. *Première partie : Algorithmique avancée pour les graphes*. AAIA. CITI Lab, INSA Lyon dépt. Informatique. 2016. url : `http://perso.citi-lab.fr/csolnon/supportAlgoGraphes.pdf` (visité le 28/12/2023).