AAIA 3IF - TP TSP Branch & Bound

Florian Rascoussier, Janvier 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.

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.

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.

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.

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]
```

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 | e ∈ [1, ..., 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 :

```c
/**
 * @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;
    }
}
```

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
```

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);
```

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
```

Partie 3 : Recherche du plus court-circuit hamiltonien

On ajoute une variable globale :

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

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;
        }
    }
    [...]
}
```

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
```

Comparez ces temps d'exécution à ceux du programme utilisant un principe de programmation dynamique.

TODO: Répondre, faire le TD en question.

Partie 4 : Propagation de contraintes

Montrez qu'une conséquence de cette propriété est que la solution optimale ne peut pas contenir deux arêtes qui se croisent.

Preuve par l'absurde.

On suppose `Copti`, chemin hamiltonien optimal de longueur la plus faible. On suppose également qu'il existe deux indices `i` et `j` (avec `0 ≤ i < j ≤ n`) tels que les arcs `(vi, vi+1)` et `(vj , vj+1)` se croisent, ces arcs étant des arcs de `Copti`.

On transforme `Copti` en `Copti'` prime tel que `Copti'` est un chemin hamiltonien avec les mêmes arcs que `Copti` à la différence des arcs `(vi, vi+1)` et `(vj , vj+1)`, remplacés par les arcs `(vi, vj)` et `(vi+1, vj+1)`.

D'après la propriété `Longueur(Copti') ⩽ Longueur(Copti)`. Contradiction.

On déduit que la solution optimale `Copti` ne peut pas contenir deux arêtes qui se croisent.

Nouvelle modification du code pour éliminer ces cas de la recherche.

Soit i un entier tel que i appartient à {0, …, k - 1}, avec k le nombre de sommets visités dans le chemin à l'étape k. À l'étape k, on cherche à ajouter un k+1ème sommet.

Pour vérifier que l'on n'ajoute pas d'arc qui croise les arcs déjà présents dans le chemin à une étape donnée, on vérifie que la longueur additionnée `Longeur(Vi → Vi+1) + Longueur(Vk → Vk+1) ⩽ Longueur(Vi → Vk) + Longueur(Vi+1 → Vk+1)`.

On modifie ainsi le code de la fonction `permut`. Dans la bouche `for`, on ajoute :

```
// constraint: no crossing edges
        if (hasCrossingEdges(visited, nbVisited, notVisited[i], cost))
            continue;
```

On définit ensuite la fonction `hasCrossingEdges` qui détecte si le nœud que l'on cherche à ajouter va créer un croisement d'arc.

```
bool hasCrossingEdges(int visited[], int nbVisited, int newVertex, int** cost) {
    if (nbVisited ⩽ 3) {
        return false; // no crossing edges with 3 vertices or less
```

```
    }
    // Longueur(Vk → Vk+1)
    int costLastToNewVectex = cost[newVertex][visited[nbVisited-1]];
    for (int i = 0; i < (nbVisited-2); i++) {
        // Longueur(Vi → Vi+1)
        int costItoIPlus1 = cost[visited[i]][visited[i+1]];
        // Longueur(Vi → Vk)
        int costItoLast = cost[visited[i]][visited[nbVisited-1]];
        // Longueur(Vi+1 → Vk+1)
        int costIPlus1toNewVertex = cost[newVertex][visited[i+1]];
        if (
            (costItoLast +  costIPlus1toNewVertex)
            < (costItoIPlus1 + costLastToNewVectex)
        ) {
            return true;
        }
    }
    return false;
}
```

On peut alors effectuer des calculs sur des valeurs de `n` plus élevées :

```
$ for i in 14 16 18 20 22 24; do ./bin/main $i; done
best cost: 69382
n=14 nbCalls=151861 time=0.010s
best cost: 70310
n=16 nbCalls=978013 time=0.051s
best cost: 75456
n=18 nbCalls=6065015 time=0.341s
best cost: 81292
n=20 nbCalls=32165315 time=2.010s
best cost: 82447
n=22 nbCalls=174134399 time=11.980s
best cost: 83193
n=24 nbCalls=1032192967 time=77.010s
```

Partie 5 : Implémentation de `bound` simple

On implémente la fonction `bound`, Cette fonction d'évaluation calcule une borne inférieure de la longueur du plus court chemin allant du dernier sommet visité jusqu'à 0 en passant par chaque sommet non visité exactement une fois.

```
int bound(
    int visited[], int nbVisited,
    int notVisited[], int nbNotVisited,
    int** cost
) {
```

```
    int sum = 0;
    // get l, lenght of the smallest edge from the last
    // visited vertex to one of the remaining unvisited
    // vertices
    int l = INT_MAX;
    for (int i; i < nbNotVisited; i++) {
        if (cost[visited[nbVisited-1]][notVisited[i]] < l) {
            l = cost[visited[nbVisited-1]][notVisited[i]];
        }
    }
    sum += l;

    // Now, for every remaining unvisited vertex, we determine
    // l' the length of the smallest edge from this vertex to
    // one of the remaining unvisited vertices, or to the end
    // of the tour (vertex 0).
    for (int i = 0; i < nbNotVisited; i++) {
        int lPrime = INT_MAX;
        // remaining unvisited vertices
        for (int j = 0; j < nbNotVisited; j++) {
            if (cost[notVisited[i]][notVisited[j]] < lPrime) {
                lPrime = cost[notVisited[i]][notVisited[j]];
            }
        }
        // vertex 0
        if (cost[notVisited[i]][0] < lPrime) {
            lPrime = cost[notVisited[i]][0];
        }
        sum += lPrime;
    }

    return sum;
}
```

On peut alors intégrer cette fonction afin de limiter le nombre de récursions dans `permut`. Mais d'abord, il nous fait un moyen de conserver la longueur du parcours en cours. On pourrait recalculer cette valeur à chaque fois, mais ça serait peu efficace. On préfèrera la passer comme un paramètre. On modifie la fonction `permut` avec un nouveau paramètre `int costVisited`. Au premier appel de `permut`, cette valeur sera de 0 puisqu'aucun sommet n'a été visité.

On peut alors ajouter, dans la boucle `for` et après la vérification de non-croisement, le code suivant :

```
// constraint: bound
        int nextCost = costVisited + cost[visited[nbVisited-1]][notVisited[i]];
        int boundedCost = nextCost + bound(
```

```
            visited, nbVisited,
            notVisited, nbNotVisited,
            cost
        );
        if (boundedCost > bestCost)
            continue;
```

Cette amélioration nous permet de faire des calculs avec des valeurs de n toujours plus grand :

```
$ for i in 4 20 22 24 26 28; do ./bin/main $i; done
n=4, bestCost=31319, nbCalls=8, time=0.000s
n=20, bestCost=84729, nbCalls=819954, time=0.164s
n=22, bestCost=85149, nbCalls=3507653, time=0.815s
n=24, bestCost=84078, nbCalls=15779770, time=3.788s
n=26, bestCost=88141, nbCalls=29627046, time=7.679s
n=28, bestCost=89676, nbCalls=88340120, time=27.548s
```

Partie 6 : Implémentation d'une fonction `bound` plus sophistiquée

MST = Minimum Spanning Tree, Arbre Couvrant Minimum.

On peut améliorer la fonction `bound` précédente en calculant une meilleure approximation de la borne inférieur. Pour cela, on se base sur un calcul d'arbre couvrant minimal. Dans l'idée, il s'agit vraiment de faire le même calcul que dans la partie précédente, l'amélioration ici se situe au niveau du fait que, plutôt que de calculer une limite incluant le sommet 0 de fin pour chaque sommet non visité, ici, on aura un meilleur encadrement puisqu'un seul sommet pourra être connecté à l'arrivée 0, au lieu de potentiellement tous avant.

Voici l'implémentation de l'algorithme de Prim, pour notre situation. On note que l'on n'a pas besoin de maintenir l'ensemble E des arcs de l'arbre couvrant minimum, puisqu'on ne s'intéresse qu'à la longueur de l'arborescence (MST) construite par l'algorithme. On notera aussi qu'on n'a pas besoin de passer la structure du graphe de base, puisque celui-ci est complet. Il faut néanmoins passer en paramètre de la fonction le nombre de nœuds du graphe de base malgré qu'on soit en train de calculer le MST du sous-graphe des nœuds non visités, afin de pouvoir initialiser la taille des tableaux correctement.

```
int costPrimMST(
    int baseGraphNbVertices,
    int vertices[],
    int nbVertices,
    int** cost
) {
    int s0 = vertices[0];

    // initializations
```

```
bool* isVisited = (bool*) malloc(baseGraphNbVertices*sizeof(bool));
int* minCostfrom = (int*) malloc(baseGraphNbVertices*sizeof(int));
int* predecesor = (int*) malloc(baseGraphNbVertices*sizeof(int));
// for (int i = 0; i < baseGraphNbVertices; i++) {
//     minCostfrom[i] = INT_MAX;
//     predecesor[i] = -1;
//     isVisited[i] = false;
// }
isVisited[s0] = true;
int nbVisited = 1;
for (int i = 1; i < nbVertices; i++) {
    isVisited[vertices[i]] = false;
    minCostfrom[vertices[i]] = cost[s0][vertices[i]];
    predecesor[vertices[i]] = s0;
}

while (nbVisited < nbVertices) {
    // get vertex with minimum cost
    int minCost = INT_MAX;
    int sMinCost;
    for (int i = 1; i < nbVertices; i++) {
        if (minCostfrom[vertices[i]] < minCost) {
            minCost = minCostfrom[vertices[i]];
            sMinCost = vertices[i];
        }
    }

    isVisited[sMinCost] = true;
    nbVisited++;

    for(int i = 1; i < nbVertices; i++) {
        if (isVisited[vertices[i]] == false &
            cost[sMinCost][vertices[i]] < minCostfrom[vertices[i]]
        ) {
            predecesor[vertices[i]] = sMinCost;
            minCostfrom[vertices[i]] = cost[sMinCost][vertices[i]];
        }
    }
}

// compute sum of costs (sum of all the predecesor arborescence's costs)
int sum = 0;
for (int i = 1; i < nbVertices; i++) {
    sum += cost[predecesor[vertices[i]]][vertices[i]];
}
```

```
    // clean up
    free(isVisited);
    free(minCostfrom);
    free(predecesor);

    return sum;
}
```

Il ne reste plus qu'à intégrer cette nouvelle fonction dans la fonction `bound` :

```
int bound(
    int visited[], int nbVisited,
    int notVisited[], int nbNotVisited,
    int** cost
) {
    int sum = 0;
    // get l, lenght of the smallest edge from the last
    // visited vertex to one of the remaining unvisited
    // vertices.
    // Same for lToZero, from any unvisited vertex to 0.
    int lFromLast, lToZero = INT_MAX;
    for (int i; i < nbNotVisited; i++) {
        if (cost[visited[nbVisited-1]][notVisited[i]] < lFromLast) {
            lFromLast = cost[visited[nbVisited-1]][notVisited[i]];
        }
        if (cost[notVisited[i]][0] < lToZero) {
            lToZero = cost[notVisited[i]][0];
        }
    }
    sum += lFromLast + lToZero;

    // Now, for every remaining unvisited vertex, we compute
    // the value of the minimum spanning tree (MST) of the
    // remaining unvisited vertices, and add it to the sum.
    sum += costPrimMST(
        nbNotVisited + nbVisited, // n
        notVisited, nbNotVisited, cost
    );

    return sum;
}
```

Cette amélioration permet de continuer à augmenter la taille de `n` calculable "rapidement".

```
$ for i in 4 22 24 26 28 30; do ./bin/main $i; done
n=4, bestCost=31319, nbCalls=8, time=0.000s
n=22, bestCost=85149, nbCalls=487196, time=0.136s
```

```
n=24, bestCost=91278, nbCalls=3295095, time=0.892s
n=26, bestCost=90943, nbCalls=11584310, time=3.424s
n=28, bestCost=93669, nbCalls=29870273, time=10.189s
n=30, bestCost=95952, nbCalls=158631849, time=57.491s
```

Note sur la non-utilisation d'une file de priorité.

Considérons l'algorithme d'un point de vue théorique. D'après le cours de Christine Solnon :

Soient `n` le nombre de sommets et `p` le nombre d'arêtes du graphe sur lequel on veut calculer le MST. L'algorithme passe `n - 1` fois dans la boucle `while`. À chaque passage, il faut chercher le sommet de l'ensemble des sommets pas encore visité ayant la plus petite valeur du tableau `c` (`minCostfrom` dans mon implémentation) puis parcourir toutes les arêtes adjacentes à ce sommet. Si ces sommets non visités sont mémorisés dans un tableau ou une liste, la complexité est `O(n2)`. Si on utilise une file de priorité (implémentation en tas binaire), alors la complexité est `O(p.log(n))`. En effet, si l'accès se fait en temps constant, il faut aussi compter la mise à jour du tas binaire à chaque fois que l'on modifie le tableau `c`. Comme il y a au plus `p` mises à jour de `c` (une par arête), la complexité de l'algorithme Prim dans ce cas est bien `O(p log n)`.

Malheureusement, comme le graphe qui nous intéresse est complet, c'est-à-dire que chaque sommet est connecté à tous les autres sommets, alors on a `p = n(n - 1)/2`. Dans ce cas, on aurait `O(n^2.log n)` qui est pire que `O(n^2)`. D'où le fait que l'on n'utilisera pas de file de priorité ici.

Partie 7 : Ajout d'une heuristique d'ordre

On peut encore améliorer nos résultats simplement, en guidant la récusion dans `permut` pour que soient explorés en premier les plus courts circuits. En effet, cela permettra alors d'élaguer les chemins moins intéressants encore plus rapidement, et donc d'accélérer le processus de recherche.

Pour cela, on dispose d'une *heuristique d'ordre* simple: Visiter d'abord les sommets les plus proches du dernier sommet visité.

On modifie en conséquence la fonction `permut`. On commence par sortir la boucle `for` dans une sous-fonction `permutLoop`. Cela sera utile pour compartimenter le code, et surtout éviter de renommer le tableau `notVisited` que l'on va devoir réordonner à chaque passage dans `permut`.

```
/**
 * @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 permutLoop(
    int visited[], int nbVisited, int costVisited,
    int notVisited[], int nbNotVisited,
    int** cost
) {
    for (int i = 0; i < nbNotVisited; i++) {
        // constraint: no crossing edges
        if (hasCrossingEdges(visited, nbVisited, notVisited[i], cost))
            continue;

        // constraint: bound
        int nextCost = costVisited + cost[visited[nbVisited-1]][notVisited[i]];
        int boundedCost = nextCost + bound(
            visited, nbVisited,
            notVisited, nbNotVisited,
            cost
        );
        if (boundedCost > bestCost)
            continue;

        // 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, nextCost,
            notVisited, nbNotVisited-1,
            cost
        );

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

À noter que cette fonction utilise `permut` dans son corps. Hors, cette fonction n'est pas encore définie. On doit donc ajouter un fichier *header* pour définir nos fonctions et permettre la compilation sans erreurs. Le fichier `main.c` :

```c
#ifndef MAIN_H
#define MAIN_H

#include <stdbool.h>

int nextRand(int n);
int** createCost(int n);
void genPythonTurtleTour(int visited[], int n);
bool hasCrossingEdges(
    int visited[],
    int nbVisited,
    int newVertex,
    int** cost
);
int costPrimMST(
    int baseGraphNbVertices,
    int vertices[],
    int nbVertices,
    int** cost
);
int bound(
    int visited[], int nbVisited,
    int notVisited[], int nbNotVisited,
    int** cost
);
void permutLoop(
    int visited[], int nbVisited, int costVisited,
    int notVisited[], int nbNotVisited,
    int** cost
);
void permut(
    int visited[], int nbVisited, int costVisited,
    int notVisited[], int nbNotVisited,
    int** cost
);
void printCostMatrix(int** cost, int n);

#endif
```

Dans ce fichier, seule le fait de définir `permut` et `permutLoop` est réellement important.

A l'emplacement où se trouvait le corps de `permutLoop` dans `permut`, on ajoute le code suivant :

```
// reordering notVisited array (a copy of it)
// wrt cost of edge from last visited vertex

// make copies of arrays before sorting in place
int* costsFromLastVisited = (int*) malloc(nbNotVisited*sizeof(int));
for (int i = 0; i < nbNotVisited; i++) {
    costsFromLastVisited[i] = cost[visited[nbVisited-1]][notVisited[i]];
}
int* notVisitedIncrOrder = (int*) malloc(nbNotVisited*sizeof(int));
for (int i = 0; i < nbNotVisited; i++) {
    notVisitedIncrOrder[i] = notVisited[i];
}

quicksortInPlace(
    notVisitedIncrOrder,
    0, // index of first element
    nbNotVisited - 1, // WARN: index of last element, not number of elements
    costsFromLastVisited
);

// recursive call, with reordered notVisited array
permutLoop(
    visited, nbVisited, costVisited,
    notVisitedIncrOrder, nbNotVisited,
    cost
);

// clean up
free(costsFromLastVisited);
free(notVisitedIncrOrder);
```

Ceci permet de trier le tableau de sommets `notVisited` en un tableau trié dans l'ordre croissant des coûts de chaque arc depuis le dernier sommet visité jusqu'à chacun des sommets du tableau. Pour effectuer le tri, on dispose de pas mal de possibilités. J'ai ici choisi l'algorithme `quicksort` parce qu'il est plutôt efficate, élégant et simple à implémenter. Néanmoins, d'autres algorithmes plus performants existent (voir comparaison, voir visualisation).

On obtient les résultats suivants :

```
$ for i in 4 28 30 32 34 36 38 40 45 50 52; do ./bin/main $i; done
n=4, bestCost=31319, nbCalls=6, time=0.000s
n=28, bestCost=90552, nbCalls=11867, time=0.005s
n=30, bestCost=104093, nbCalls=18834, time=0.011s
n=32, bestCost=97563, nbCalls=1655, time=0.001s
n=34, bestCost=98386, nbCalls=3540, time=0.002s
n=36, bestCost=99110, nbCalls=8320, time=0.005s
```

```
n=38, bestCost=111108, nbCalls=1384, time=0.001s
n=40, bestCost=118817, nbCalls=1386, time=0.002s
n=45, bestCost=127100, nbCalls=1392, time=0.002s
n=50, bestCost=129422, nbCalls=138467, time=0.052s
n=52, bestCost=130734, nbCalls=95488710, time=58.517s
```

On observe que le mur de temps de calculs est maintenant atteint pour `n=52`, ce qui est une amélioration significative.