République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Rapport de Travaux Pratiques

Module : TPRO

TP2 : Le Problème du Voyageur de Commerce (PVC)

Comparaison : Force Brute vs Held-Karp

Réalisé par le binôme :

ATTIA Oussama Abderraouf
SRAICH Imene

Encadré par :
M. HADIM Boukhalfa

Année Universitaire : 2025 - 2026

Table des matières

1 Introduction **2**

2 Approche Algorithmique **2**
2.1 Méthode Exacte (Force Brute) 2
2.2 Algorithme de Held-Karp (Programmation Dynamique) 2

3 Implémentation et Code **2**
3.1 Structures et Génération de Graphe 2
3.2 Implémentation Force Brute (Backtracking) 3
3.3 Implémentation Held-Karp 3

4 Analyse des Résultats **4**
4.1 Tableau Comparatif (Sortie Console) 4
4.2 Observation 4

5 Conclusion **4**

1 Introduction

Le Problème du Voyageur de Commerce (PVC), ou Traveling Salesperson Problem (TSP), est un classique de l'algorithmique. Il consiste, pour un ensemble de villes et les distances les séparant, à trouver le plus court chemin qui visite chaque ville exactement une fois et revient au point de départ.

Ce TP a pour objectif d'implémenter et de comparer deux approches pour résoudre ce problème :

1. Une méthode exacte naïve par énumération des permutations (**Force Brute**).
2. Une méthode optimisée par programmation dynamique (**Algorithme de Bellman-Held-Karp**).

Le langage **C** a été choisi pour sa performance dans la manipulation bas niveau de la mémoire (tableaux, matrices, masques de bits).

2 Approche Algorithmique

2.1 Méthode Exacte (Force Brute)

Cette méthode consiste à générer toutes les permutations possibles des villes intermédiaires. Si nous avons N villes numérotées de 0 à $N-1$, et que nous fixons 0 comme ville de départ, il nous reste $(N-1)!$ chemins possibles à explorer.

— **Avantage :** Simple à implémenter (Backtracking).
— **Inconvénient :** Complexité factorielle $O(N!)$. Devient inutilisable pour $N \geq 12$.

2.2 Algorithme de Held-Karp (Programmation Dynamique)

L'algorithme utilise la programmation dynamique avec un masque de bits (bitmask) pour représenter l'ensemble des villes visitées. Soit $dp[mask][j]$ le coût minimal pour visiter l'ensemble de villes défini par le masque $mask$, en finissant par la ville j.

La transition est définie par :

$$dp[mask|(1 \ll k)][k] = \min(dp[mask][j] + dist[j][k])$$

— **Complexité Temporelle :** $O(N^2 \cdot 2^N)$.
— **Complexité Spatiale :** $O(N \cdot 2^N)$.
— **Avantage :** Beaucoup plus rapide que la force brute pour $N \leq 20$.

3 Implémentation et Code

Voici les parties essentielles du code source développé (`tsp_compare.c`).

3.1 Structures et Génération de Graphe

Nous utilisons une matrice d'adjacence pour représenter le graphe complet. Les distances sont générées aléatoirement.

```c
// Generation d'un graphe complet avec poids aleatoires
long long **generate_random_complete_graph(int n) {
    long long **d = alloc_matrix_int(n);
    for (int i=0;i<n;i++){
        for (int j=0;j<n;j++){
            if (i==j) d[i][j]=0;
            else if (j<i) d[i][j]=d[j][i];
            else d[i][j] = 1 + rand()%MAX_DIST;
        }
    }
    return d;
}
```

3.2 Implémentation Force Brute (Backtracking)

Cette fonction récursive explore toutes les permutations.

```c
void backtrack_perm(long long **d, int n, int *perm, int depth) {
    if (depth == n-1) {
        // perm est complet : calcule cout total du cycle
        long long cost = cycle_cost_from_perm(d, n, perm, n-1);
        if (cost < exact_best_cost) {
            exact_best_cost = cost;
            // sauvegarde du meilleur chemin...
        }
        return;
    }
    // essayer toutes les villes non utilisees (1..n-1)
    for (int v=1; v<n; v++){
        if (!used_global[v]) {
            used_global[v] = 1;
            perm[depth] = v;
            backtrack_perm(d, n, perm, depth+1);
            used_global[v] = 0;
        }
    }
}
```

3.3 Implémentation Held-Karp

Utilisation des opérations bit-à-bit pour gérer les ensembles.

```c
long long held_karp(long long **d, int n) {
    int Nmask = 1<<n;
    // Allocation memoire pour DP table...

    // Initialisation
    for (int j=1;j<n;j++){
        int mask = (1<<0) | (1<<j);
        dp[mask*n + j] = d[0][j];
    }

    // Iteration sur les masques
    for (int mask=0; mask<Nmask; mask++){
        if (!(mask & 1)) continue; // Doit contenir 0

        for (int j=1;j<n;j++){
```

```
            if (!(mask & (1<<j))) continue;
            // Tenter d'etendre vers k
            for (int k=1;k<n;k++){
                if (mask & (1<<k)) continue;
                int nmask = mask | (1<<k);
                // Mise a jour du min...
            }
        }
    }
    // ... Retour au depart et choix du min final
}
```

4 Analyse des Résultats

Le programme effectue un benchmark en faisant varier N de 2 à 12. Pour chaque taille, il compare le temps d'exécution et vérifie que les deux algorithmes trouvent bien le même coût minimal.

4.1 Tableau Comparatif (Sortie Console)

Lors de l'exécution, le programme affiche un tableau similaire à celui-ci :

```
 n | exact_time(s) | heldkarp_time(s) | best_cost | note
-------------------------------------------------------------
 4 |   0.000002    |   0.000005       |       120 | exact OK
...
10 |   1.254000    |   0.002100       |       345 | exact OK
11 |       ---     |   0.004500       |       410 | exact skipped
12 |       ---     |   0.010200       |       450 | exact skipped
```

4.2 Observation

On remarque clairement l'explosion combinatoire de la méthode exacte.

— Pour $N = 10$, la méthode exacte prend plus d'une seconde, alors que Held-Karp est quasi-instantané (2ms).

— Au-delà de $N = 10$, nous avons désactivé la méthode exacte (`exact skipped`) car le temps de calcul devient déraisonnable ($11! \approx 40$ millions d'opérations, $12! \approx 479$ millions).

— Held-Karp reste très performant même pour $N = 12$, validant la complexité en $O(N^2 2^N)$.

5 Conclusion

Ce TP nous a permis de confronter la théorie à la pratique. Bien que le problème du voyageur de commerce soit NP-Complet, l'approche par programmation dynamique (Held-Karp) permet de repousser les limites de calcul par rapport à une approche naïve.

L'implémentation en C nous a permis de gérer efficacement la mémoire nécessaire pour la table de programmation dynamique, bien que celle-ci grandisse aussi de manière exponentielle (2^N).