



Université Hassan II de Casablanca
École Nationale Supérieure d'Électricité et de Mécanique

RAPPORT DE PROJET

PROBLÈME DU VOYAGEUR DE COMMERCE

(Traveling Salesman Problem - TSP)

Implémentation en Langage C Algorithmes : Dijkstra & Brute Force

Encadré par :

Pr. BOUKHDIR KHALID

Réalisé par :

ANDAME AMINE
AHMED RIDA BOURAYA

Table des Matières

Table des Matières.....	2
Introduction	3
Objectifs du Projet.....	3
Structure du Projet	3
Représentation du Graphe	4
Structure de données	4
Carte des villes marocaines.....	5
Algorithme de Dijkstra	6
Principe de l'algorithme.....	6
Étapes de l'algorithme	6
Algorithme TSP - Brute Force	8
Le problème du voyageur de commerce	8
L'approche Brute Force	8
Génération des permutations	8
Résultats et Tests.....	9
Test de l'algorithme de Dijkstra.....	9
Résultat du TSP.....	9
Conclusion	11

Introduction

Le problème du voyageur de commerce (TSP - Traveling Salesman Problem) est l'un des problèmes d'optimisation combinatoire les plus célèbres en informatique. Il consiste à trouver le plus court chemin permettant de visiter un ensemble de villes exactement une fois et de revenir à la ville de départ.

Ce projet implémente une solution au TSP en utilisant deux algorithmes fondamentaux :

- **L'algorithme de Dijkstra** pour calculer les plus courts chemins entre toutes les paires de villes
- **L'algorithme Brute Force** pour trouver le tour optimal en testant toutes les permutations possibles

L'application est testée sur un réseau de 10 villes marocaines avec des distances réalistes entre elles.

Objectifs du Projet

Les objectifs principaux de ce projet sont :

1. **Représenter un réseau de villes** sous forme de graphe avec une matrice d'adjacence
2. **Implémenter l'algorithme de Dijkstra** pour trouver le plus court chemin entre deux villes
3. **Résoudre le problème TSP** par l'approche Brute Force (force brute)
4. **Tester et valider** les algorithmes sur un cas concret de 10 villes marocaines

Structure du Projet

Le projet est organisé en plusieurs fichiers pour une meilleure modularité :

Fichier	Description
graph.h	En-tête : Structures et prototypes du graphe
graph.c	Implémentation du graphe et de l'algorithme de Dijkstra
tsp.h	En-tête : Structures et prototypes TSP
tsp.c	Implémentation de l'algorithme TSP Brute Force
main.c	Programme principal avec les tests et le menu

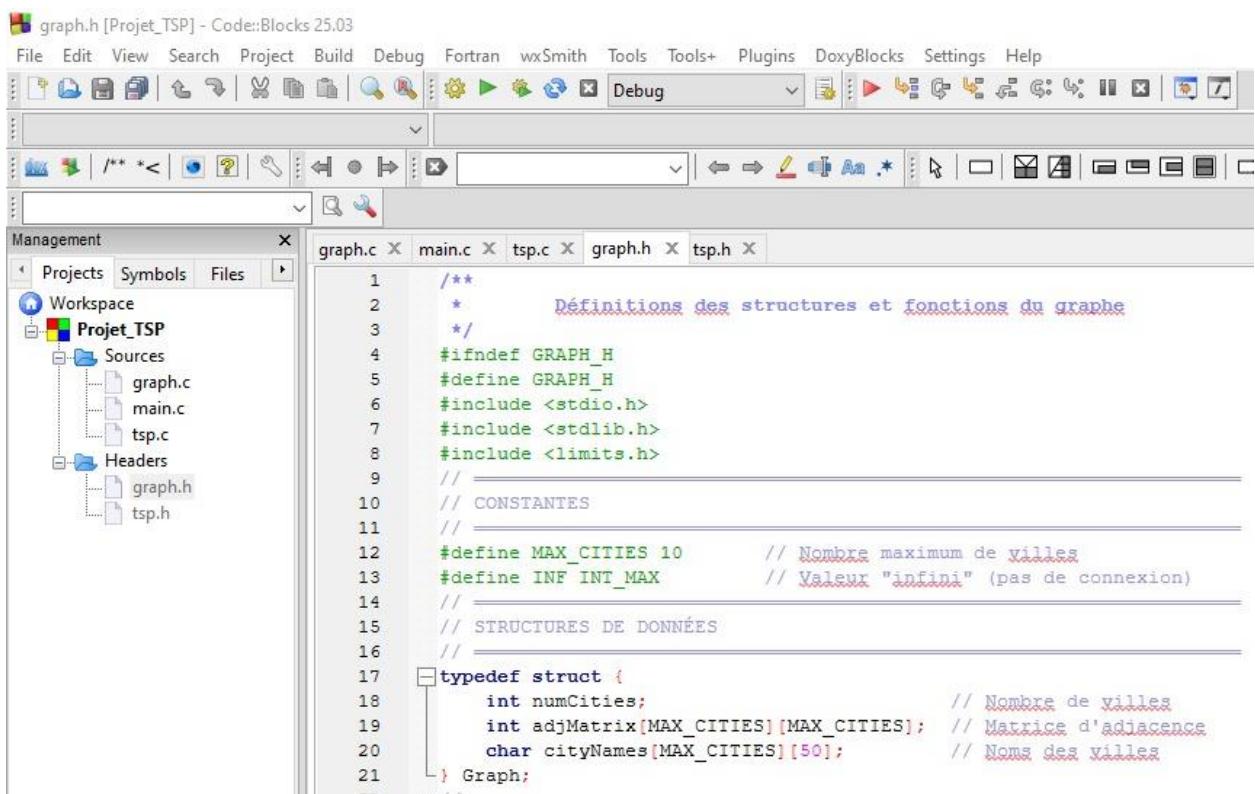
Représentation du Graphe

Structure de données

Le graphe est représenté par une **matrice d'adjacence**. Cette structure permet de stocker les distances entre chaque paire de villes.

La structure Graph contient :

- **numCities** : le nombre de villes dans le graphe
- **adjMatrix[i][j]** : la distance entre la ville i et la ville j
- **cityNames[]** : les noms des villes pour l'affichage



The screenshot shows the Code::Blocks IDE interface with the graph.h file open in the main editor window. The file contains the definition of the Graph structure. The code is as follows:

```

1  /**
2   *      Définitions des structures et fonctions du graphe
3   */
4  #ifndef GRAPH_H
5  #define GRAPH_H
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <limits.h>
9  // ==
10 // CONSTANTES
11 // ==
12 #define MAX_CITIES 10           // Nombre maximum de villes
13 #define INF INT_MAX            // Valeur "infini" (pas de connexion)
14 // ==
15 // STRUCTURES DE DONNÉES
16 // ==
17 typedef struct {
18     int numCities;                // Nombre de villes
19     int adjMatrix[MAX_CITIES][MAX_CITIES]; // Matrice d'adjacence
20     char cityNames[MAX_CITIES][50];    // Noms des villes
21 } Graph;
22

```

Figure 1 : Définition de la structure Graph dans graph.h

Carte des villes marocaines

Notre graphe modélise 10 villes marocaines avec leurs connexions routières :

Index	Ville	Index	Ville
0	Casablanca (Départ)	5	Agadir
1	Rabat	6	Meknes
2	Marrakech	7	Oujda
3	Fes	8	Tetouan
4	Tanger	9	El Jadida

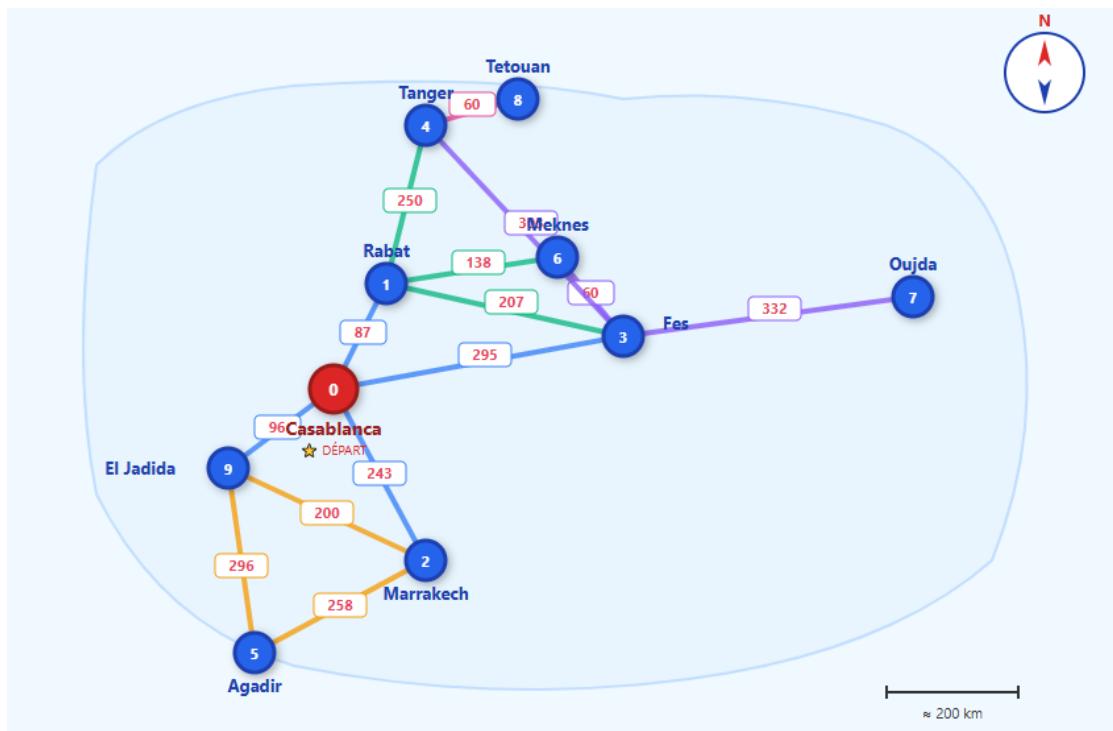


Figure 2 :

Graphe des connexions entre les 10 villes marocaines

Algorithme de Dijkstra

Principe de l'algorithme

L'algorithme de Dijkstra permet de trouver le **plus court chemin** entre une ville source et toutes les autres villes du graphe. Il fonctionne sur des graphes à poids positifs.

Étapes de l'algorithme

5. **Initialisation** : Mettre la distance de la source à 0, toutes les autres à l'infini (INF)
6. **Sélection** : Choisir la ville non visitée avec la plus petite distance
7. **Mise à jour** : Pour chaque voisin, si un chemin plus court est trouvé, mettre à jour sa distance
8. **Répétition** : Répéter jusqu'à ce que toutes les villes soient visitées

```

174     int dijkstra(Graph* graph, int src, int dest, int* path, int* pathLength) {
175         // Vérifications
176         if (graph == NULL || src < 0 || src >= graph->numCities ||
177             dest < 0 || dest >= graph->numCities) {
178             return INF;
179         }
180         int numCities = graph->numCities;
181         int dist[MAX_CITIES];           // Distance minimale depuis la source
182         int visited[MAX_CITIES];       // Sommets déjà traités
183         int parent[MAX_CITIES];        // Prédécesseur dans le chemin
184         // ÉTAPE 1 : Initialisation
185         for (int i = 0; i < numCities; i++) {
186             dist[i] = INF;
187             visited[i] = 0;
188             parent[i] = -1;
189         }
190         dist[src] = 0;
191         // ÉTAPE 2 : Boucle principale
192         for (int count = 0; count < numCities - 1; count++) {
193             // Trouver le sommet non visité avec distance minimale
194             int u = findMinDistance(dist, visited, numCities);
195             if (u == -1) break;
196             visited[u] = 1;
197             // Mettre à jour les distances des voisins
198             for (int v = 0; v < numCities; v++) {
199                 if (!visited[v] &&
```

```
200         graph->adjMatrix[u][v] != INF &&
201         dist[u] != INF &&
202         dist[u] + graph->adjMatrix[u][v] < dist[v]) {
203             dist[v] = dist[u] + graph->adjMatrix[u][v];
204             parent[v] = u;
205         }
206     }
207 }
208 // ÉTAPE 3 : Reconstruction du chemin
209 if (path != NULL && pathLength != NULL && dist[dest] != INF) {
210     int len = 0;
211     int current = dest;
212     while (current != -1) {
213         len++;
214         current = parent[current];
215     }
216     *pathLength = len;
217     current = dest;
218     for (int i = len - 1; i >= 0; i--) {
219         path[i] = current;
220         current = parent[current];
221     }
222 } else if (pathLength != NULL) {
223     *pathLength = 0;
224 }
225 return dist[dest];
```

Figure 3 : Implémentation de l'algorithme de Dijkstra

Algorithme TSP - Brute Force

Le problème du voyageur de commerce

Le problème TSP consiste à trouver le tour le plus court qui :

- Part d'une ville de départ (Casablanca)
- Visite chaque ville exactement UNE FOIS
- Retourne à la ville de départ

L'approche Brute Force

L'approche Brute Force consiste à tester **TOUTES** les permutations possibles des villes et à garder celle avec la distance minimale.

Pour n villes, le nombre de permutations à tester est $(n-1)!$ car on fixe la ville de départ.

Nombre de villes	Permutations	Temps estimé
5	24	< 1 ms
10	362,880	< 1 sec
15	87 milliards	Plusieurs heures

Tableau 1 : Croissance factorielle du nombre de permutations

Génération des permutations

Les permutations sont générées de manière **récursive** par la fonction `permute()`. Cette fonction utilise la technique du **backtracking** (retour sur trace).

```

51 static void permute(int* cities, int start, int end) {
52     // CAS DE BASE : permutation complète
53     if (start == end) {
54         int distance = calculateTourDistance(cities, globalNumCities, globalDistMatrix);
55         if (distance < bestDistance) {
56             bestDistance = distance;
57             for (int i = 0; i < globalNumCities; i++) {
58                 bestTour[i] = cities[i];
59             }
60             bestTour[globalNumCities] = cities[0];
61         }
62     } else {
63         // CAS RÉCURSIF : générer les permutations
64         for (int i = start; i <= end; i++) {
65             swap(&cities[start], &cities[i]);
66             permute(cities, start + 1, end);
67             swap(&cities[start], &cities[i]); // Backtrack
68         }
69     }
70 }
```

Figure 4 : Fonction récursive de génération des permutations

Résultats et Tests

Test de l'algorithme de Dijkstra

Nous avons testé Dijkstra sur deux trajets :

Trajet	Distance	Chemin optimal
Casablanca → Oujda	617 km	Casa → Rabat → Meknes → Fes → Oujda
Agadir → Tetouan	789 km	Agadir → El Jadida → Casa → Rabat → Tanger → Tetouan

Résultat du TSP

Après avoir testé les **362,880 permutations**, l'algorithme a trouvé le tour optimal suivant :

DISTANCE TOTALE MINIMALE : 2,602 km

Tour : Casablanca → Rabat → Tanger → Tetouan → Oujda → Fes → Meknes → Marrakech → Agadir → El Jadida → Casablanca

Distance Totale Minimale
2,602 km
 sur 362,880 permutations testées

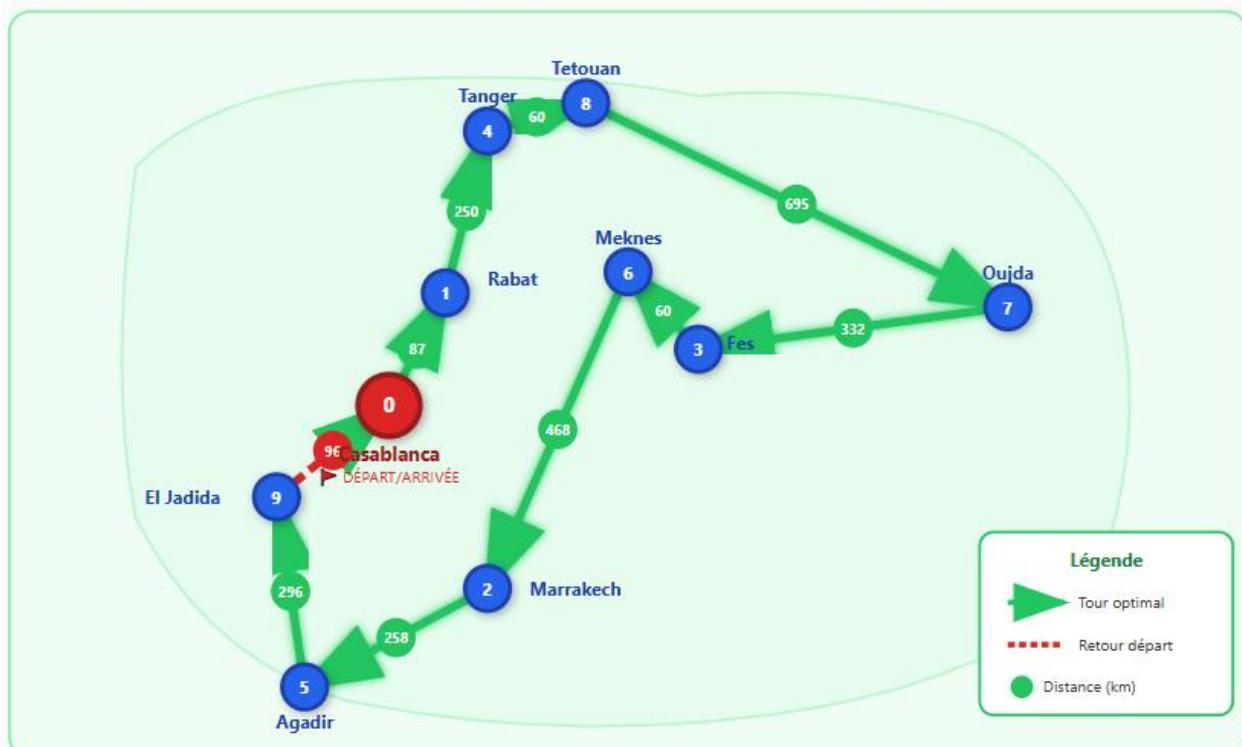


Figure 5 : Visualisation du tour optimal trouvé par l'algorithme

```
+=====+  
|      TEST DE L'ALGORITHME DIJKSTRA      |  
+=====+  
  
Test 1 : Casablanca -> Oujda  
-----  
Distance minimale : 617 km  
Chemin : Casablanca -> Rabat -> Meknes -> Fes -> Oujda  
  
Test 2 : Agadir -> Tetouan  
-----  
Distance minimale : 789 km  
Chemin : Agadir -> El Jadida -> Casablanca -> Rabat -> Tanger -> Tetouan  
  
===== RESULTAT DU TSP =====  
  
Distance totale minimale : 2602  
  
Tour optimal :  
1. Casablanca (ville 0) --->  
2. Rabat (ville 1) --->  
3. Tanger (ville 4) --->  
4. Tetouan (ville 8) --->  
5. Oujda (ville 7) --->  
6. Fes (ville 3) --->  
7. Meknes (ville 6) --->  
8. Marrakech (ville 2) --->  
9. Agadir (ville 5) --->  
10. El Jadida (ville 9) --->  
11. Casablanca (ville 0)
```

Figure 6 : Capture d'écran de l'exécution du programme

Conclusion

Ce projet nous a permis d'implémenter et de comprendre deux algorithmes fondamentaux en informatique :

- **L'algorithme de Dijkstra** : un algorithme glouton efficace pour trouver les plus courts chemins avec une complexité $O(V^2)$
- **L'algorithme Brute Force pour le TSP** : une approche exhaustive qui garantit la solution optimale mais avec une complexité factorielle $O(n!)$

Résultats obtenus :

- Le programme trouve correctement les plus courts chemins entre les villes
- Le tour optimal de 2,602 km a été trouvé parmi 362,880 permutations
- L'exécution est rapide pour 10 villes (moins d'une seconde)

Perspectives d'amélioration :

- Implémenter des algorithmes plus efficaces comme Held-Karp (programmation dynamique)
- Utiliser des heuristiques comme l'algorithme génétique ou le recuit simulé
- Ajouter une interface graphique pour visualiser les résultats