République Démocratique et Populaire d'Algérie
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique
Département de Système Informatique
Spécialité : Big Data

T.P. : Δ-Problème du voyageur de commerce

Membres du groupe :

– BENDAHGANE Sami
– MOKRANI Sarra

Travail présenté à :

Pr.AIDER Méziane

Année académique : 2024/2025

Table des matières

1 Introduction

Dans ce rapport, nous étudions l'approximabilité du Δ-Problème du voyageur de commerce (TSP triangulaire), défini sur un graphe complet $G = (V, E)$ dont les arêtes respectent l'inégalité triangulaire. L'algorithme proposé repose sur la construction d'un arbre couvrant minimum, la duplication de ses arêtes, la recherche d'un parcours eulérien, puis la transformation de ce parcours en cycle hamiltonien.

2 Complexité et NP-difficulté du TSP

Le problème du voyageur de commerce (TSP) consiste à trouver un cycle hamiltonien de coût minimal dans un graphe pondéré complet. Ce problème est bien connu pour sa difficulté : il appartient à la classe des problèmes $\mathcal{NP}$-difficiles (NP-hard), même dans des cas restreints.

NP-difficulté du TSP général

Dans sa version générale (sans contrainte d'inégalité triangulaire), le TSP est $\mathcal{NP}$-difficile car:

- Il n'existe pas d'algorithme connu permettant de résoudre ce problème en temps polynomial pour tous les cas.
- Même la simple vérification qu'un cycle donné est optimal requiert de comparer tous les cycles possibles, ce qui est exponentiel.
- Ce travail repose sur l'hypothèse largement acceptée que $P \neq NP$. Ainsi, étant donné que le problème du voyageur de commerce est $\mathcal{NP}$-difficile, il est peu probable qu'un algorithme exact en temps polynomial existe. Cela justifie l'approche par approximation utilisée ici, qui garantit une solution en temps raisonnable, tout en offrant une borne sur la qualité du résultat obtenu.

TSP triangulaire et approximabilité

Cependant, lorsque les poids des arêtes respectent l'inégalité triangulaire ($\forall u, v, w \in V$, $d(u, v) \leq d(u, w) + d(w, v)$), le problème admet de meilleurs résultats d'approximation. Cette condition garantit que passer directement d'un sommet à un autre ne coûte jamais plus que de faire un détour.

Ainsi, bien que le TSP triangulaire reste NP-difficile, il devient **approximable** avec un facteur garanti. L'algorithme décrit dans ce rapport, inspiré de l'algorithme de Christofides (dans une version simplifiée), fournit une solution de coût au plus deux fois supérieur à l'optimal.

Ce compromis entre qualité et efficacité est essentiel dans la résolution pratique de problèmes NP-difficiles.

3 Description de l'algorithme

L'algorithme se déroule en quatre étapes principales :

- Construction d'un arbre couvrant minimum T du graphe G.
- Duplication de chaque arête de T pour obtenir un graphe eulérien H.
- Recherche d'un parcours eulérien dans H.
- Transformation du parcours eulérien en cycle hamiltonien en évitant de revisiter un sommet déjà marqué.

4 Définitions mathématiques des algorithmes utilisés

4.1 Algorithme de Prim

L'algorithme de Prim construit un arbre couvrant minimum en partant d'un sommet initial et en ajoutant à chaque étape l'arête de poids minimal reliant un sommet à l'arbre en cours de construction.

Formellement :

- Initialiser l'arbre avec un sommet quelconque.
- Tant qu'il existe des sommets non inclus dans l'arbre, ajouter l'arête de poids minimal connectant l'arbre à un sommet non inclus.

Complexité pour un graphe dense : $O(n^2)$.

4.2 Algorithme de Kruskal

L'algorithme de Kruskal construit un arbre couvrant minimum en triant toutes les arêtes par poids croissant et en ajoutant les arêtes les plus légères à condition qu'elles ne forment pas de cycle.

Formellement :

- Trier les arêtes par poids croissant.
- Parcourir les arêtes triées et les ajouter à l'arbre si elles relient deux composants différents (structure Union-Find).

Complexité : $O(m \log m)$ avec m le nombre d'arêtes.

4.3 Algorithme de Hierholzer pour parcours eulérien

Hierholzer permet de trouver un parcours eulérien dans un graphe connexe où chaque sommet a un degré pair.

Formellement :

- Partir d'un sommet quelconque et suivre les arêtes disponibles jusqu'à revenir au point de départ.
- Si des sommets du cycle obtenu ont des arêtes non encore explorées, recommencer à partir de ces sommets et fusionner les cycles.

Complexité : $O(m)$.

4.4 Transformation parcours eulérien → cycle hamiltonien

Pendant le parcours eulérien, chaque sommet est visité la première fois qu'il est rencontré, et les répétitions sont ignorées. Complexité : $O(n)$.

5 Analyse de la complexité

5.1 Construction de l'arbre couvrant minimum (MST)

Graphe complet avec n sommets $\Rightarrow O(n^2)$ arêtes.

- Kruskal avec Union-Find : $O(m \log m) = O(n^2 \log n)$.
- Prim avec tableau simple : $O(n^2)$ car graphe dense.

Donc : $O(n^2)$ pour construire le MST avec Prim.

5.2 Construction du graphe eulérien H

Chaque arête du MST est copiée une deuxième fois :

- Nombre d'arêtes du MST : $n-1$, donc $2(n-1)$ après duplication.
- Temps pour doubler les arêtes : $O(n)$.

5.3 Trouver un parcours eulérien

- Le graphe H est eulérien (degré pair partout).
- Recherche par DFS ou Hierholzer sur $2n-2$ arêtes : $O(n)$.

5.4 Transformer en cycle hamiltonien

- On saute les sommets déjà visités.
- Passage exactement une fois sur chaque sommet : $O(n)$.

5.5 Complexité globale

Le traitement est dominé par la construction du MST : Complexité totale : $\boxed{O(n^2)}$.

6 Preuves et Résultats

Problème 1. *Montrer que le cycle hamiltonien construit est une tournée de longueur au plus égale au double de la longueur d'une tournée optimale.*

Solution :

- Étape 1 : Construire un arbre couvrant minimum T.
- Étape 2 : Doubler chaque arête de T pour obtenir un graphe eulérien H.
- Étape 3 : Trouver un parcours eulérien dans H.
- Étape 4 : Transformer ce parcours en cycle hamiltonien en sautant les sommets déjà visités.

Analyses de longueurs :

- L'arbre couvrant T a une longueur $\ell(T)$.
- Le parcours eulérien couvre toutes les arêtes de H, soit une longueur $2 \times \ell(T)$.
- Grâce à l'inégalité triangulaire, sauter des sommets n'augmente pas la longueur totale.

Ainsi :

$$\text{longueur finale} \leq 2 \times \ell(T)$$

Et comme :

$$\ell(T) \leq \ell(OPT)$$

on a :

$$\text{longueur finale} \leq 2 \times \ell(OPT)$$

Théorème 1. *La tournée construite est au plus deux fois plus longue que la tournée optimale.*

7 Conséquences

Problème 2. *Que peut-on déduire de ce qui précède ?*

Réponse : On peut déduire que l'algorithme est une **2-approximation** pour le problème du voyageur de commerce triangulaire. Cela signifie que :

$$\text{Solution Algorithme} \leq 2 \times \text{Solution Optimale}$$

Ce résultat est important car il montre que pour le TSP triangulaire, il existe un algorithme polynomial qui garantit une tournée proche de l'optimum, avec un facteur d'approximation de 2, acceptable dans de nombreuses applications pratiques.

8 Procédure Simul pour la génération de matrices de distances aléatoires

Dans le cadre de l'étude du Δ-Problème du voyageur de commerce, il est essentiel de disposer de données réalistes pour simuler des scénarios et tester les performances des algorithmes. La procédure **Simul** permet de générer des matrices de distances entre les villes d'un graphe complet tout en respectant l'inégalité triangulaire. Cette approche garantit que les instances générées sont valides et peuvent être utilisées dans les simulations des algorithmes de résolution.

8.1 Objectif de la procédure Simul

L'objectif principal de la procédure **Simul** est de générer une matrice de distances aléatoires entre n sommets (villes) d'un graphe complet, tout en respectant l'inégalité triangulaire. L'inégalité triangulaire stipule que pour trois villes u, v, et w, la distance directe entre u et w ne doit pas être supérieure à la somme des distances entre u et v, et entre v et w. En d'autres termes :

$$d(u, w) \leq d(u, v) + d(v, w)$$

Cette contrainte est essentielle pour garantir la validité des simulations et des tests d'algorithmes sur des instances réalistes du problème du voyageur de commerce.

8.2 Structure de la procédure Simul

La procédure **Simul** se divise en plusieurs étapes :

- **Initialisation de la matrice des distances :** La matrice des distances est initialisée sous forme de matrice carrée $n \times n$, où n est le nombre de sommets (villes). Les éléments de la diagonale sont initialisés à 0 (la distance entre une ville et elle-même est nulle). Les autres éléments sont initialement définis à zéro.
- **Génération des distances aléatoires :** La fonction `rand()` génère des valeurs aléatoires dans l'intervalle $[1, \texttt{maxDistance}]$ pour chaque paire de villes (i, j), avec $i \neq j$. La matrice est ensuite rendue symétrique : pour chaque valeur $\texttt{distances}[i][j]$, la valeur $\texttt{distances}[j][i]$ est affectée de la même valeur.
- **Respect de l'inégalité triangulaire :** Après la génération initiale des distances, la procédure ajuste les distances pour garantir que l'inégalité triangulaire est respectée pour chaque triplet de villes (u, v, w). Si la distance directe entre u et w est plus grande que la somme des distances entre u et v, et entre v et w, la distance entre u et w est ajustée pour satisfaire l'inégalité triangulaire.

- **Retour de la matrice de distances :** Une fois les ajustements effectués, la matrice des distances est renvoyée, représentant une instance valide du problème du voyageur de commerce.

Choix du langage C++

Le langage C++ a été choisi pour la réalisation de ce projet pour plusieurs raisons techniques et pédagogiques. Tout d'abord, le C++ est un langage compilé, réputé pour sa rapidité d'exécution et son efficacité en gestion de la mémoire, ce qui est particulièrement pertinent pour des algorithmes traitant des graphes ou nécessitant de nombreuses opérations sur des structures de données comme les arbres couvrants ou les circuits eulériens.

Ensuite, le C++ dispose d'une riche bibliothèque standard (`STL`) qui facilite la manipulation des structures de données telles que les vecteurs, les ensembles, ou encore les algorithmes de tri, tout en laissant la possibilité d'implémenter manuellement des structures optimisées comme l'union-find (structure indispensable pour l'algorithme de Kruskal).

D'un point de vue pédagogique, ce projet constitue également une opportunité d'approfondir des concepts fondamentaux de la programmation en C++ (gestion dynamique, récursivité, complexité algorithmique), tout en appliquant des notions avancées d'algorithmique sur un problème classique et difficile, le TSP (Travelling Salesman Problem).

Enfin, le langage C++ permet une bonne portabilité et une intégration aisée avec des outils de visualisation ou d'analyse en ligne de commande, ce qui a facilité l'expérimentation sur différentes tailles d'instances (n = 10, 20, 50).

8.3 Détails de l'implémentation

Voici le code en C++ de la procédure **Simul**, ainsi que les fonctions associées pour afficher la matrice et exécuter le programme principal :

```cpp
// Génère un graphe complet respectant l'inégalité triangulaire
vector< vector<int> > Simul(int n, int maxDistance = 100) {
    vector< vector<int> > distances(n, vector<int>(n, 0));
    // Initialisation aléatoire
    srand(time(0));
    // Remplir la matrice de distances avec des valeurs aléatoires
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            distances[i][j] = rand() % maxDistance + 1; // distance entre 1 et maxDistance
            distances[j][i] = distances[i][j]; // symétrie
        }
    }
    // Ajustement pour respecter l'inégalité triangulaire
    for (int u = 0; u < n; ++u) {
        for (int v = 0; v < n; ++v) {
            for (int w = 0; w < n; ++w) {
                if (u != v && v != w && u != w) {
                    distances[u][w] = min(distances[u][w], distances[u][v] + distances[v][w]);
                    distances[w][u] = distances[u][w]; // symétrie
                }
            }
        }
    }
    return distances;
}
```

(a) Procédure Simul

```cpp
// Fonction pour afficher la matrice
void afficherDistances(const vector< vector<int> >& distances) {
    int n = distances.size();
    cout << "Matrice des distances :" << endl;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cout << setw(4) << distances[i][j] << " ";
        }
        cout << endl;
    }
}
```

(b) Fonction pour afficher la matrice

```cpp
int main() {
    int n;
    cout << "Entrez le nombre de sommets : ";
    cin >> n;

    vector< vector<int> > distances = Simul(n);

    afficherDistances(distances);

    return 0;
}
```

(c) Fonction `main()`

Figure 1: Implémentation complète en C++ de la procédure Simul et des fonctions associées

8.4 Explication du code

- **Matrice des distances :** La matrice `distances` est initialisée comme une matrice carrée de taille $n \times n$, avec des valeurs 0 sur la diagonale (la distance entre une ville et elle-même).

- **Génération aléatoire des distances :** La fonction `rand()` génère des distances comprises entre 1 et `maxDistance` pour chaque paire de villes (i, j) avec $i < j$, en assurant que la matrice est symétrique en affectant `distances`[j][i] = `distances`[i][j].
- **Ajustement de l'inégalité triangulaire :** Le troisième bloc du code parcourt tous les triplets de villes (u, v, w) et ajuste les distances directes pour garantir que l'inégalité triangulaire soit respectée : $d(u, w) \leq d(u, v) + d(v, w)$.
- **Retour de la matrice :** Après avoir ajusté la matrice, celle-ci est renvoyée, représentant une instance valide du Δ-Problème du voyageur de commerce.

8.5 Exemple d'exécution

Supposons que nous générions une instance avec $n = 4$ sommets. La matrice des distances générée pourrait ressembler à ceci :

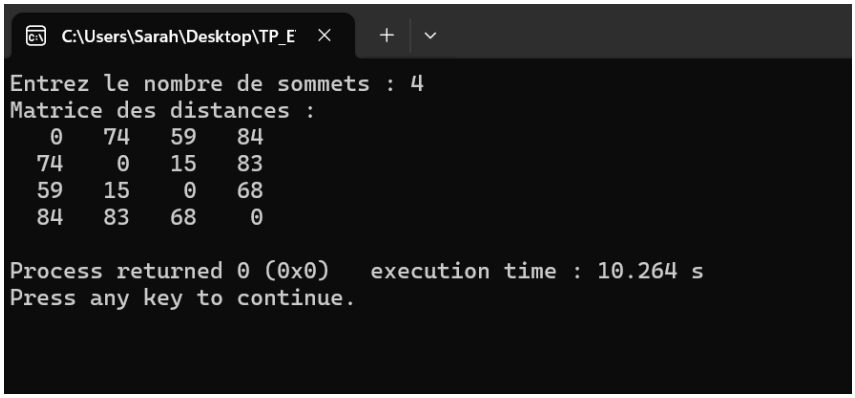

Figure 2: simul exe

L'interprétation de cette matrice est la suivante :

- La distance entre la ville 0 et la ville 1 est 74.
- La distance entre la ville 1 et la ville 2 est 15.
- La matrice est symétrique : la distance entre la ville 0 et la ville 2 est 59, et celle entre la ville 2 et la ville 0 est également 59.
- L'inégalité triangulaire est respectée, comme en témoigne l'ajustement des distances.

8.6 Conclusion

La procédure **Simul** constitue un outil efficace pour générer des instances aléatoires du Δ-Problème du voyageur de commerce. Elle garantit que les distances entre les villes respectent l'inégalité triangulaire, ce qui en fait une méthode fiable pour tester et évaluer des algorithmes de résolution du TSP dans des scénarios réalistes. Grâce à cette procédure, il est possible de réaliser des simulations diverses, en générant des instances adaptées aux tests de performance des algorithmes.

9 Algorithme de Kruskal : Implémentation

9.1 Introduction

L'algorithme de **Kruskal** est une méthode classique utilisée pour résoudre le problème de l'**arbre couvrant de poids minimal (MST)** dans un graphe pondéré. Il consiste à ajouter progres-

sivement les arêtes de poids croissant en évitant la formation de cycles, jusqu'à connecter tous les sommets.

9.2 Objectif de l'implémentation

L'objectif du programme est :

- Générer un graphe non orienté aléatoire.
- Appliquer l'algorithme de Kruskal pour en extraire un arbre couvrant minimal.
- Afficher les arêtes composant l'arbre résultant.

9.3 Structure du code

Représentation d'une arête Une arête est définie par deux sommets u et v et un poids :

```cpp
struct Arete {
    int u, v;
    int poids;
};
```

Figure 3: Enter Caption

Génération du graphe La fonction `genererGraphe()` génère des arêtes aléatoires entre des paires distinctes de sommets, avec des poids compris entre 1 et 100.

```cpp
vector<Arete> genererGraphe(int n, int nbAretes) {
    vector<Arete> graphe;
    srand(time(0));

    while ((int)graphe.size() < nbAretes) {
        int u = rand() % n;
        int v = rand() % n;
        int poids = rand() % 100 + 1;

        if (u != v) {
            Arete a;
            a.u = u;
            a.v = v;
            a.poids = poids;
            graphe.push_back(a);
        }
    }

    return graphe;
}
```

Figure 4: Enter Caption

Structure Union-Find Pour éviter les cycles, deux fonctions de gestion d'ensembles disjoints sont utilisées :

```cpp
int trouver(int x) {
    if (parent[x] != x)
        parent[x] = trouver(parent[x]);
    return parent[x];
}
```

Figure 5: **trouver(x) :** retourne le représentant de l'ensemble contenant x.

```cpp
void unionSets(int a, int b) {
    a = trouver(a);
    b = trouver(b);
    if (a != b)
        parent[b] = a;
}
```

Figure 6: **unionSets(a, b) :** fusionne les ensembles contenant a et b.

Tri des arêtes Les arêtes sont triées par poids croissant à l'aide de la fonction :

```cpp
bool comparerAretes(Arete a, Arete b) {
    return a.poids < b.poids;
}
```

Figure 7: **comparerAretes(a,b)**

Fonction principale de Kruskal Cette fonction trie les arêtes et sélectionne les plus légères qui ne forment pas de cycle pour construire un arbre couvrant minimal.

```cpp
vector<Arete> kruskal(int n, vector<Arete>& aretes) {
    vector<Arete> mst;
    parent.resize(n);
    for (int i = 0; i < n; ++i)
        parent[i] = i;

    sort(aretes.begin(), aretes.end(), comparerAretes);

    for (int i = 0; i < (int)aretes.size(); ++i) {
        Arete& a = aretes[i];
        if (trouver(a.u) != trouver(a.v)) {
            mst.push_back(a);
            unionSets(a.u, a.v);
        }
    }

    return mst;
}
```

Figure 8: Fonction `kruskal()`

Complexité : `O(E log E)`, où `E` est le nombre d'arêtes (tri par poids).

9.4 Exemple d'exécution

```cpp
int main() {
    int n, m;
    cout << "Nombre de sommets : ";
    cin >> n;
    cout << "Nombre d'arêtes : ";
    cin >> m;

    vector<Arete> graphe = genererGraphe(n, m);
    vector<Arete> mst = kruskal(n, graphe);

    cout << "\nArbre couvrant de poids minimal (Kruskal) :\n";
    for (size_t i = 0; i < mst.size(); ++i) {
        cout << mst[i].u << " - " << mst[i].v << " : " << mst[i].poids << endl;
    }

    return 0;
}
```

Figure 9: **main()**

```
Select C:\Users\Sarah\Documents\New folder\kruskal.exe
Nombre de sommets : 5
Nombre d'arÛtes : 7

Arbre couvrant de poids minimal (Kruskal) :
3 - 0 : 18
4 - 0 : 21
2 - 3 : 24
1 - 0 : 63

--------------------------------
Process exited after 20.84 seconds with return value 0
Press any key to continue . . .
```

Figure 10: kruskal.exe

Chaque ligne indique une arête sélectionnée, les sommets connectés et leur poids.

9.5 Conclusion

L'algorithme de Kruskal, implémenté dans ce programme, permet de générer dynamiquement des graphes aléatoires et de construire efficacement un arbre couvrant minimal. Il est particulièrement adapté aux graphes denses et garantit une bonne complexité algorithmique. Des vérifications supplémentaires assureraient la connexité du graphe pour une fiabilité accrue.

10 Transformation d'un parcours eulérien en cycle hamiltonien

10.1 Objectif

À partir de l'arbre couvrant minimal obtenu par Kruskal, on souhaite construire un cycle hamiltonien de coût au plus égal à celui du parcours eulérien associé. Pour cela, on applique une procédure inspirée de l'approximation 2 du TSP.

10.2 Étapes de la transformation

1. Utiliser `Simul()` pour générer une matrice de distances vérifiant l'inégalité triangulaire.
2. Appliquer Kruskal pour extraire un arbre couvrant minimal.
3. Construire un parcours eulérien à l'aide d'un DFS.
4. Transformer le parcours eulérien en cycle hamiltonien en retirant les répétitions de sommets.
5. Calculer la longueur du cycle résultant.

10.3 Implémentation des étapes clés

Génération des distances .

```cpp
// Génère un graphe complet respectant l'inégalité triangulaire
vector< vector<int> > Simul(int n, int maxDistance = 100) {
    vector< vector<int> > distances(n, vector<int>(n, 0));
    // Initialisation aléatoire
    srand(time(0));
    // Remplir la matrice de distances avec des valeurs aléatoires
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            distances[i][j] = rand() % maxDistance + 1; // distance entre 1 et maxDistance
            distances[j][i] = distances[i][j]; // symétrie
        }
    }
    // Ajustement pour respecter l'inégalité triangulaire
    for (int u = 0; u < n; ++u) {
        for (int v = 0; v < n; ++v) {
            for (int w = 0; w < n; ++w) {
                if (u != v && v != w && u != w) {
                    distances[u][w] = min(distances[u][w], distances[u][v] + distances[v][w]);
                    distances[w][u] = distances[u][w]; // symétrie
                }
            }
        }
    }
    return distances;
}
```

Figure 11: Procédure Simul

Construction du parcours .

```cpp
void dfs(int u, const vector<vector<int> >& adj, vector<bool>& visite, vector<int>& parcours) {
    visite[u] = true;
    parcours.push_back(u);
    for (int v : adj[u])
        if (!visite[v])
            dfs(v, adj, visite, parcours);
}
```

Figure 12: Fonction `dfs()`

Transformation en cycle hamiltonien .

```cpp
vector<int> toHamiltonien(const vector<int>& parcours) {
    vector<bool> visite(*max_element(parcours.begin(), parcours.end()) + 1, false);
    vector<int> cycle;

    for (int v : parcours) {
        if (!visite[v]) {
            cycle.push_back(v);
            visite[v] = true;
        }
    }

    if (!cycle.empty())
        cycle.push_back(cycle.front());

    return cycle;
}
```

Figure 13: Transformation en cycle hamiltonien (filtrage des doublons)

Calcul de la longueur du cycle .

```cpp
int calculerLongueurCycle(const vector<int>& cycle, const vector<vector<int> >& distances) {
    int longueur = 0;
    for (size_t i = 0; i < cycle.size() - 1; ++i)
        longueur += distances[cycle[i]][cycle[i + 1]];
    return longueur;
}
```

Figure 14: Calcul de la longueur du cycle hamiltonien

10.4 Exemple d'exécution complète

L'exécution du programme C++ produit, pour différentes tailles de graphes ($n = 10, 20, 50$), un affichage détaillé des différentes étapes de la transformation du graphe initial vers un cycle hamiltonien.

```
C:\Users\Sarah\Desktop\ETU(
========== Test pour n = 10 ==========
Fichier distances gÚnÚrÚ : distances_10.csv
Fichier DOT gÚnÚrÚ : graphe_initial_10.dot
Fichier DOT gÚnÚrÚ : arbre_kruskal_10.dot
Longueur du cycle hamiltonien : 256
Fichier rÚsultats gÚnÚrÚ : resultats_10.txt
Cycle : 0 -> 5 -> 4 -> 8 -> 6 -> 2 -> 1 -> 3 -> 9 -> 7 -> 0 -> 0
Temps d'exÚcution : 0.0080 secondes
========================================

========== Test pour n = 20 ==========
Fichier distances gÚnÚrÚ : distances_20.csv
Fichier DOT gÚnÚrÚ : graphe_initial_20.dot
Fichier DOT gÚnÚrÚ : arbre_kruskal_20.dot
Longueur du cycle hamiltonien : 229
Fichier rÚsultats gÚnÚrÚ : resultats_20.txt
Cycle : 0 -> 5 -> 17 -> 9 -> 1 -> 11 -> 3 -> 16 -> 12 -> 4 -> 6 -> 2 -> 14 -> 8 -> 13 -> 19 -> 18 -> 10 -> 7 -> 15 -> 0
-> 0
Temps d'exÚcution : 0.0090 secondes
========================================

========== Test pour n = 50 ==========
Fichier distances gÚnÚrÚ : distances_50.csv
Fichier DOT gÚnÚrÚ : graphe_initial_50.dot
Fichier DOT gÚnÚrÚ : arbre_kruskal_50.dot
Longueur du cycle hamiltonien : 282
Fichier rÚsultats gÚnÚrÚ : resultats_50.txt
Cycle : 0 -> 41 -> 36 -> 35 -> 47 -> 19 -> 25 -> 22 -> 49 -> 1 -> 30 -> 44 -> 3 -> 34 -> 13 -> 29 -> 6 -> 38 -> 21 -> 11
 -> 8 -> 18 -> 5 -> 4 -> 17 -> 32 -> 40 -> 46 -> 27 -> 7 -> 45 -> 43 -> 48 -> 20 -> 2 -> 9 -> 33 -> 23 -> 16 -> 12 -> 10
 -> 14 -> 26 -> 37 -> 28 -> 39 -> 42 -> 24 -> 15 -> 31 -> 0 -> 0
```

Figure 15: exemple de la derniere execution

À chaque itération (pour chaque valeur de n), le programme :

- Génère une matrice de distances sauvegardée dans un fichier `.csv` :

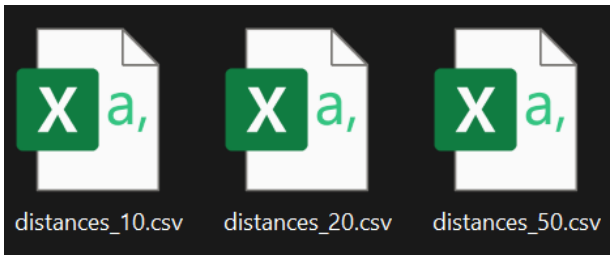

Figure 16: fichiers .csv

- Génère un graphe initial puis un arbre couvrant minimal à l'aide de l'algorithme de Kruskal, sauvegardés au format `.dot` :

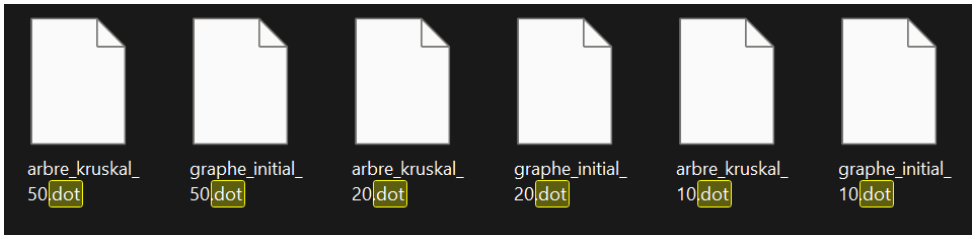

Figure 17: fichiers.dot

- Calcule un cycle hamiltonien approximatif dont la longueur est affichée, ainsi que le cycle parcouru.
- Enregistre les résultats détaillés du cycle obtenu dans un fichier `.txt` :

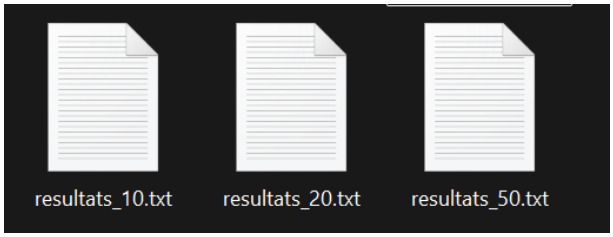

Figure 18: fichiers .txt

Ces fichiers générés peuvent ensuite être utilisés pour :

- Visualiser les graphes via `Graphviz` (fichiers `.dot`),
- Analyser les cycles obtenus (fichiers `.txt`),
- Réutiliser ou comparer les distances (fichiers `.csv`).

Le temps d'exécution pour chaque test est aussi mesuré, illustrant l'efficacité de l'approche même pour un graphe de taille 50.

Voici un exemple de rapport structuré que tu peux inclure dans ton document pour expliquer le processus de génération des graphes à partir des fichiers '.dot', en utilisant Graphviz via un script batch local :

—

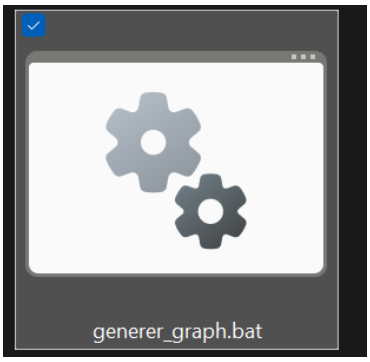

Figure 20: fichier .bat affiche dans le dossier

10.5 Génération des graphes avec Graphviz

Une fois les fichiers `.dot` générés par le programme C++, ceux-ci peuvent être transformés en images exploitables (format PNG) grâce à l'outil Graphviz. Pour automatiser cette étape, un script batch nommé `generer_graph.bat` a été ajouté dans le dossier contenant le projet.

Contenu du script `generer_graph.bat` : .

File Edit View

```
@echo off
echo Génération des images depuis les fichiers DOT...

REM Récupère le chemin absolu du dossier contenant ce script
set SCRIPT_DIR=%~dp0

REM Définit le chemin vers dot.exe à partir du dossier du script
set DOT_PATH=%SCRIPT_DIR%graphviz\bin\dot.exe

IF NOT EXIST "%DOT_PATH%" (
    echo Erreur : dot.exe introuvable à "%DOT_PATH%"
    pause
    exit /b
)

for %%f in (*.dot) do (
    echo Traitement de %%f ...
    "%DOT_PATH%" -Tpng "%%f" -o "%%~nf.png"
)

echo Terminé !
pause
```

Figure 19: `generer_graph.bat` en editeur de texte

Une fois le script `generer_graph.bat` enregistré dans le dossier du projet, celui-ci apparaît sous forme d'une icône caractéristique des fichiers batch sous Windows, comme illustré dans la figure ci-dessous :

Il suffit de double-cliquer sur cette icône pour exécuter automatiquement le script.

Fonctionnement :

- Le script commence par détecter automatiquement le dossier dans lequel il se trouve, afin de retrouver le chemin de `dot.exe` (fourni dans un sous-dossier `graphviz\bin` local au projet).

- Il vérifie que `dot.exe` est bien présent. En cas d'erreur, un message est affiché et le script s'interrompt.
- Pour chaque fichier `.dot` présent dans le dossier courant, une image `.png` est générée avec le même nom (ex. `graphe_initial_10.dot` devient `graphe_initial_10.png`) :

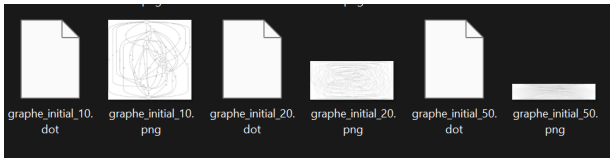

Figure 21: images generees

Avantages de cette approche :

- Aucune installation globale de Graphviz n'est requise : tout est contenu dans le dossier du projet.
- L'exécution est automatisée pour tous les fichiers `.dot`, évitant des conversions manuelles répétitives.

Exemple de graphes générés : pour n=10 :

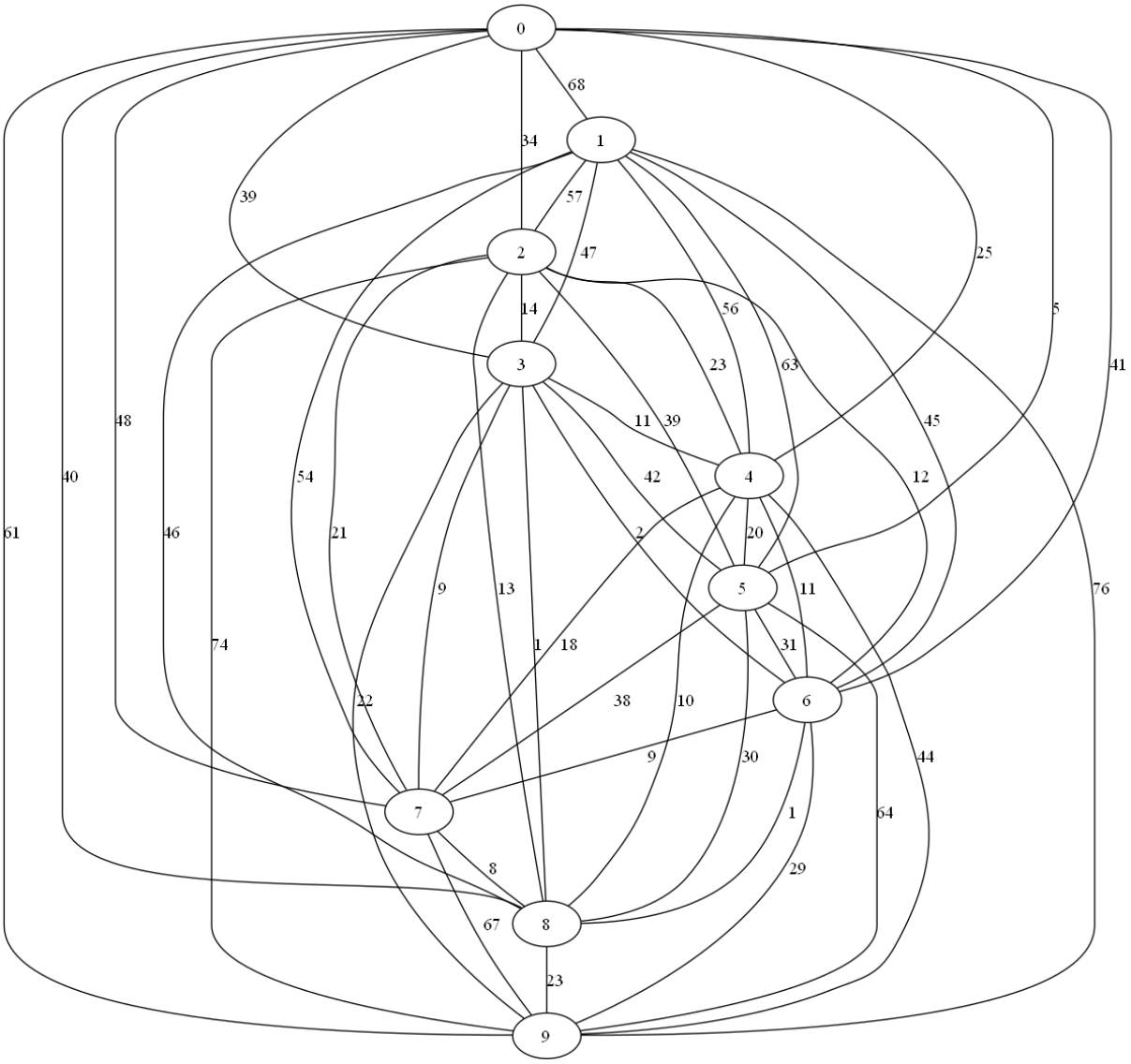

Figure 22: graphe initial pour n=10

11 Conclusion Générale

Ce projet a permis de mettre en œuvre une solution approchée pour le problème du voyageur de commerce (TSP) à partir d'une méthode simple et efficace fondée sur un arbre couvrant minimal. À travers les différentes étapes — génération de matrices de distances, construction d'un arbre via l'algorithme de Kruskal, parcours eulérien par DFS, et transformation en cycle hamiltonien — nous avons pu produire des solutions réalisables avec un coût borné grâce à l'inégalité triangulaire.

L'implémentation en C++ a démontré des performances satisfaisantes même pour des tailles croissantes de graphes, avec une complexité raisonnable. De plus, l'export des résultats en fichiers structurés (`.csv`, `.txt`, `.dot`) et l'intégration d'un script automatisé (`.bat`) pour la génération graphique facilitent considérablement l'analyse et la visualisation des résultats à l'aide de Graphviz.