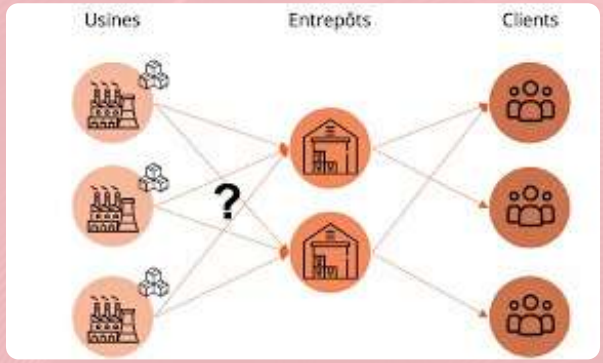
Recherche des chemins les plus courts : Floyd-Warshall

Présenté par TUNG Alexandre, ADHAR Mazien, DUMAS Eloi, KADIRI Ghali, JEYAKANTHAN Matusun

Pourquoi utiliser la théorie des graphes?



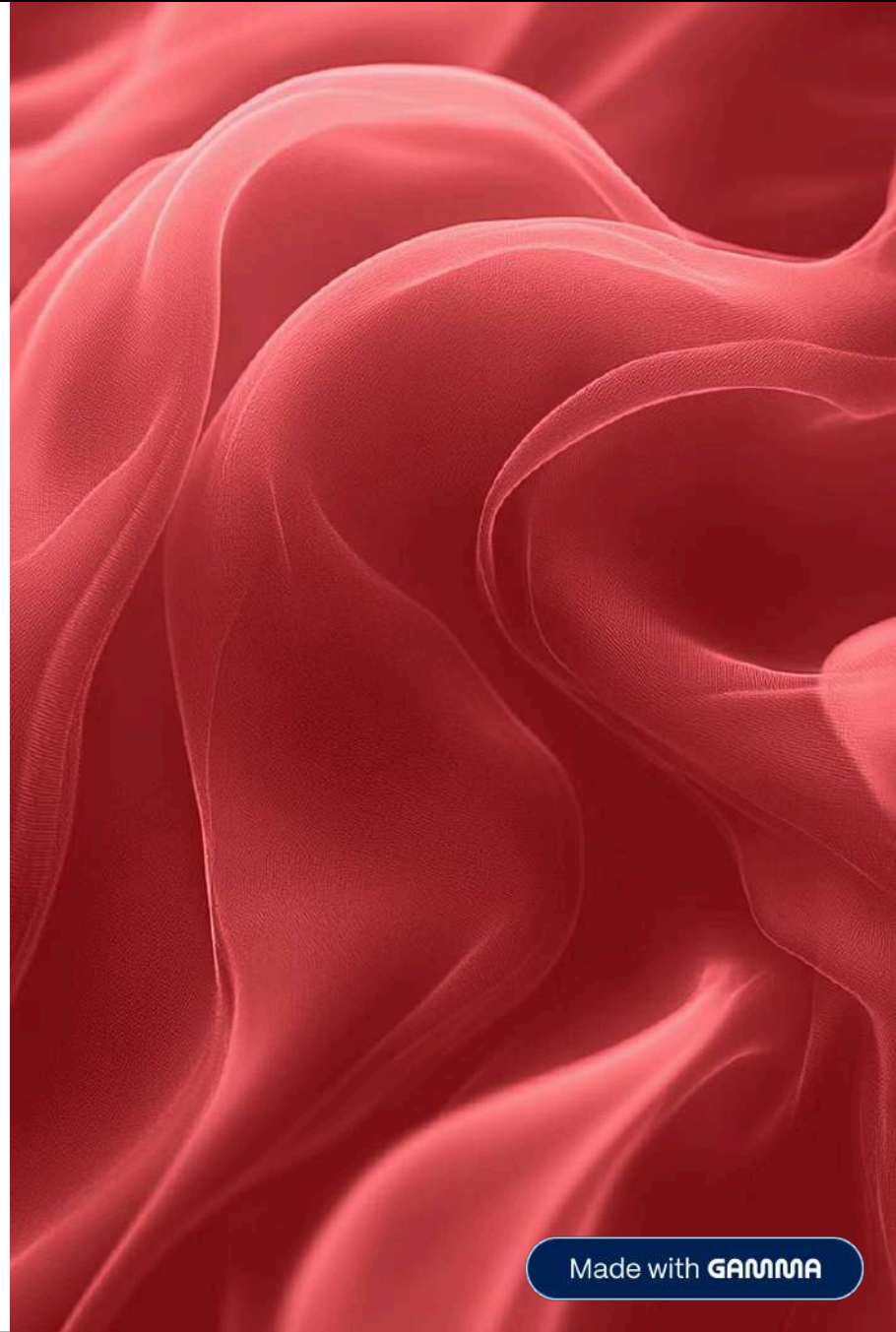
Réseaux de transport

Analyse de transactions financières

Ce qu'on va étudié

- **Implémentation et analyse technique**
  - Modélisation des données et gestion dynamique de la mémoire
  - Stabilité numérique et représentation de l'infini
  - Analyse Algorithmique et complexité
  - La détection des circuits Absorbants

- **Construction et analyse du cas réel (FOREX)**
  - Le trading haute fréquence et l'arbitrage de devises
  - Le circuit Absorbant
  - Contre-exemple et validité des chemins



Implémentation et analyse technique

1- Modélisation des données et gestion dynamique de la mémoire

Pourquoi pas de tableaux statiques ?

- Cahier des charges : **aucune limite au nombre de sommets**.
- Tableaux statiques = **taille fixe**, donc :
  - Limite arbitraire du nombre de sommets.
  - **Gaspillage mémoire** sur petits graphes.

Solution retenue : Allocation dynamique

- Utilisation de **malloc** pour allouer la mémoire **au moment de l'exécution**.
- Adaptation exacte à la taille du graphe : **N × N**.

Structure du graphe

- Graphe modélisé avec une **structure C (struct Graphe)**.
- Matrices **L** et **P** représentées par :
  - **int **** → pointeurs de pointeurs.

Implémentation et analyse technique

1- Modélisation des données et gestion dynamique de la mémoire

Avantages de l'allocation dynamique

- **Aucune limite** de taille du graphe.
- **Flexibilité totale** : on alloue seulement ce qu'il faut.
- Optimisation mémoire sur grands graphes.

Gestion propre de la mémoire

- Utilisation de **free** après chaque test.
- Évite les **fuites mémoire** et libère l'espace inutilisé.

Implémentation et analyse technique

2- Stabilité numérique et représentation de l'infini

Problème posé par Floyd-Warshall

- L'opération de relaxation : **L[i][k] + L[k][j]**.
- Risque de dépassement si on utilise **INT_MAX** comme "infini".

Pourquoi INT_MAX est dangereux ?

- **INT_MAX + INT_MAX** dépasse la capacité d'un entier.
- Résultat devient **négatif** → fausse la logique du minimum.
- Provoque des **erreurs de calcul** dans Floyd-Warshall.

Solution adoptée : constante INF

- Définition d'une valeur : **INF = 10^9**.
- Assez grande pour représenter l'infini dans nos graphes.
- Assez petite pour que **INF + INF** reste dans les limites du système.

Avantages

- Évite les **overflow**.
- Garantit la **stabilité numérique** de l'algorithme.
- Assure des résultats **fiables**.

Implémentation et analyse technique

3- Analyse Algorithmique et complexité

Principe utilisé

- Méthode : **programmation dynamique**.
- Décomposition du problème en **sous-problèmes successifs**.
- Utilisation d'un **sommet pivot k** à chaque itération.

Complexité temporelle

- Complexité : **$O(N^3)$**.
- Due à **3 boucles imbriquées** :
  - boucle sur k (pivot),
  - boucle sur i (départ),
  - boucle sur j (arrivée).

Relation de récurrence

- À chaque étape k :
  - **$L_{ij}(k) = \min(L_{ij}(k-1), L_{ik}(k-1) + L_{kj}(k-1))$**

Implémentation et analyse technique

3- Analyse Algorithmique et complexité

Mise à jour des prédécesseurs

- Si un chemin plus court est trouvé :
  - $P_{ij} = P_{kj}$
- Permet de **reconstruire l'itinéraire final**.

Suivi de la convergence

- Le programme affiche **les matrices L et P à chaque k**.
- Permet de visualiser l'évolution vers la solution.
- Fonctionnalité exigée par le cahier des charges.

Implémentation et analyse technique

4- Gestion des anomales : La détection des circuits absorbants

Pourquoi c'est un problème ?

- Un **cycle de coût négatif** ⇒ le coût peut descendre **vers −∞**.
- Le **chemin minimal devient indéfini** → résultat impossible à interpréter.

Réaction du programme

- **Blocage** de la reconstruction des chemins via la matrice P.
- Envoi d'un **message d'indétermination** à l'utilisateur.
- Évite la production de résultats incohérents.

Méthode de détection

- Vérification **après la convergence** de Floyd-Warshall.
- Analyse de la **diagonale** de la matrice L.
- Condition :
  - **Si L[i][i] < 0 → cycle négatif détecté**

Avantage

- Gestion **sécurisée** et **robuste** des cas impossibles.
- Empêche la propagation d'erreurs → **solution résiliente**.

Construction et analyse du cas réel (FOREX)

1- Contexte : Le trading haute fréquence et l'arbitrage de devises

Domaine d’application

- Trading Haute Fréquence (**THF**) : besoin d’analyse **instantanée**.
- Objectif : détecter les **opportunités d’arbitrage** entre devises.

Modélisation du marché

- **10 devises majeures** (EUR, USD, JPY, GBP, MYR, ...).
- **Sommets** = devises.
- **Arcs** = transactions possibles en temps réel (taux de change).
- **Poids des arcs** :
  - valeur transformée → coût / gain de l’échange ;
  - **poids négatif = transaction profitable**.

Construction et analyse du cas réel (FOREX)

1- Contexte : Le trading haute fréquence et l'arbitrage de devises

Utilisation de Floyd-Warshall

- Analyse **simultanée de toutes les paires** (10 × 10).
- Détection d'un **circuit absorbant** :
  - correspond exactement à une **opportunité d'arbitrage**.
  - permet un **profit sans risque** grâce à une boucle d'échanges gagnante.

Intérêt de l'approche

- Modélisation robuste pour un marché complexe.
- Permet une **détection rapide et exhaustive** des opportunités.

Construction et analyse du cas réel (FOREX)

2- Analyse du cas critique : Le circuit absorbant

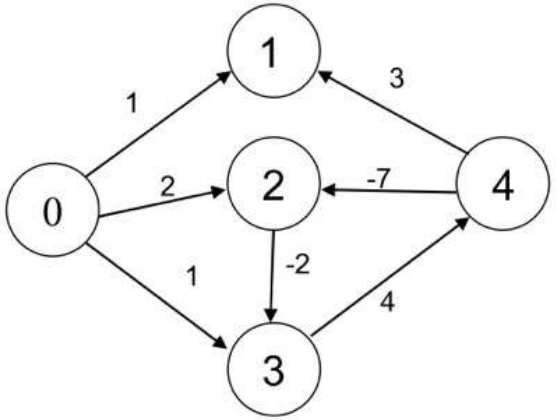
Indicateurs observés

- Dans la **matrice finale L (étape 4)** :
  - **L[2][2] = −5**
  - **L[4][4] = −10**
- Valeurs négatives sur la diagonale ⇒ **preuve d'un cycle négatif**.

Interprétation

- Condition : **L[i][i] < 0** ⇒ plus court chemin **indéfini**.
- Dans le contexte du marché : cela indique une **opportunité d'arbitrage** (profit sans risque).

Le programme détecte un **circuit absorbant** :



Construction et analyse du cas réel (FOREX)

2- Analyse du cas critique : Le circuit absorbant

Interprétation

- Condition : **L[i][i] < 0** ⇒ plus court chemin **indéfini**.
- Dans le contexte du marché : cela indique une **opportunité d'arbitrage** (profit sans risque).

```
=== ETAPE 4 ===
MATRICE L (Poids) :
  0   1  -3  -5  -1
INF   0 INF INF INF
INF   5  -5  -7  -3
INF   7  -3  -5  -1
INF  -2 -12 -14 -10
```

Comportement du programme

- Tentative d'obtenir un chemin impliquant le cycle négatif (ex : **4 → 1**).
- Le programme :
  - **bloque la reconstruction du chemin**,
  - **affiche un message d'indétermination**,
  - empêche un **résultat incohérent**.

```
--- RECHERCHE DE CHEMINS ---

Voulez-vous afficher un chemin ? (1: Oui, 0: Non) :1

Sommet de depart (0 a 4) :4

Sommet d'arrivee (0 a 4) :1

RESULTAT : Circuit absorbant. Chemin indefini.
```

Conclusion

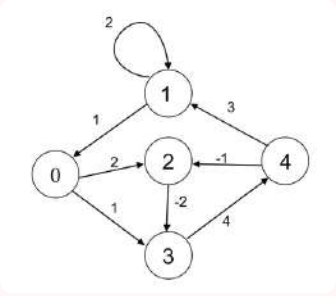
- Le filtre de sécurité fonctionne correctement.
- L'implémentation est **robuste** face aux cas critiques.
- Le système **empêche toute propagation d'erreur**, preuve de sa résilience.

Construction et analyse du cas réel (FOREX)

3- Contre exemple et validité des chemins

Le bon fonctionnement **sans circuit absorbant**,

La capacité à **trouver des chemins corrects**.



Résultats obtenus

- Aucune alerte de sécurité déclenchée.
- La **matrice finale L** présente :
  - **des valeurs ≥ 0 sur toute la diagonale**,
  - preuve qu'**aucun cycle négatif n'existe**.

```
=== ETAPE 4 ===
MATRICE L (Poids) :
  0   7   2   0   4
  1   2   3   1   5
  6   5   0  -2   2
  8   7  10   0   4
  4   3   6   4   0
```

Construction et analyse du cas réel (FOREX)

3- Contre exemple et validité des chemins

Conséquence mathématique

- **L[i][i] ≥ 0** ⇒ pas de circuit absorbant.
- La notion de **plus court chemin reste parfaitement définie**.

Comportement du programme

- Reconstruction **correcte** des chemins optimaux via la matrice **P**.
- Les itinéraires calculés sont **cohérents**, **minimaux** et **fiables**.

Conclusion

- Ce contre-exemple confirme :
  - la **validité** de l'implémentation,
  - la **fiabilité de la reconstruction des itinéraires**,
  - le bon fonctionnement de l'algorithme **en absence d'anomalies**.

```
Voulez-vous afficher un chemin ? (1: Oui, 0: Non) :1

Sommet de depart (0 a 4) :3

Sommet d'arrivee (0 a 4) :1

Chemin minimal (Cout = 7) : 3 -> 4 -> 1

Voulez-vous afficher un chemin ? (1: Oui, 0: Non) :
```

```
Voulez-vous afficher un chemin ? (1: Oui, 0: Non) :1

Sommet de depart (0 a 4) :2

Sommet d'arrivee (0 a 4) :1

Chemin minimal (Cout = 5) : 2 -> 3 -> 4 -> 1

Voulez-vous afficher un chemin ? (1: Oui, 0: Non) :
```

```
Voulez-vous afficher un chemin ? (1: Oui, 0: Non) :1

Sommet de depart (0 a 4) :0

Sommet d'arrivee (0 a 4) :4

Chemin minimal (Cout = 4) : 0 -> 2 -> 3 -> 4

Voulez-vous afficher un chemin ? (1: Oui, 0: Non) :
```

Annexe chemins contre-exemple

Annexe fonction.h

```c
#ifndef FONCTIONS_H
#define FONCTIONS_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define INF 99999
#define MAX_FILENAME 256

typedef struct {
    int n_sommets;
    int** L;
    int** P;
    int est_charge;
} Graphe;

int** allouer_matrice(int n);
void liberer_matrice(int** mat, int n);

void initialiser_graphe(Graphe* g);
void detruire_graphe(Graphe* g);

int charger_graphe_fichier(char* nom_fichier, Graphe* g);

void afficher_matrices(Graphe* g, int k);
void floyd_warshall(Graphe* g);
int detecter_circuit_absorbant(Graphe* g);
void afficher_chemin_recursif(Graphe* g, int depart, int arrivee);

#endif
```

Annexe fonction.c

```c
#include "fonctions.h"

int** allouer_matrice(int n) {
    if (n <= 0) return NULL;

    int** mat = (int**)malloc(n * sizeof(int*));
    if (mat == NULL) {
        printf("Erreur allocation.\n");
        return NULL;
    }

    for (int i = 0; i < n; i++) {
        mat[i] = (int*)malloc(n * sizeof(int));
        if (mat[i] == NULL) {
            for (int j = 0; j < i; j++) free(mat[j]);
            free(mat);
            return NULL;
        }
    }
    return mat;
}

void liberer_matrice(int** mat, int n) {
    if (mat == NULL) return;
    for (int i = 0; i < n; i++) {
        free(mat[i]);
    }
    free(mat);
}

void initialiser_graphe(Graphe* g) {
    g->n_sommets = 0;
    g->L = NULL;
    g->P = NULL;
    g->est_charge = 0;
}

void detruire_graphe(Graphe* g) {
    if (g->est_charge == 1) {
        liberer_matrice(g->L, g->n_sommets);
        liberer_matrice(g->P, g->n_sommets);
        g->est_charge = 0;
        g->n_sommets = 0;
    }
}
```

```c

int charger_graphe_fichier(char* nom_fichier, Graphe* g) {
    detruire_graphe(g);

    FILE* f = fopen(nom_fichier, "r");
    if (f == NULL) {
        printf("Erreur : Impossible d'ouvrir le fichier '%s'.\n", nom_fichier);
        return 0;
    }

    if (fscanf(f, "%d", &g->n_sommets) != 1) {
        printf("Erreur de format (nombre de sommets).\n");
        fclose(f);
        return 0;
    }

    int n_arcs;
    if (fscanf(f, "%d", &n_arcs) != 1) {
        printf("Erreur de format (nombre d'arcs).\n");
        fclose(f);
        return 0;
    }

    g->L = allouer_matrice(g->n_sommets);
    g->P = allouer_matrice(g->n_sommets);
    g->est_charge = 1;

    for (int i = 0; i < g->n_sommets; i++) {
        for (int j = 0; j < g->n_sommets; j++) {
            if (i == j) {
                g->L[i][j] = 0;
            } else {
                g->L[i][j] = INF;
            }
            g->P[i][j] = -1;
        }
    }
```

Annexe fonction.c

```
83
84      int u, v, w;
85      for (int k = 0; k < n_arcs; k++) {
86          if (fscanf(f, "%d %d %d", &u, &v, &w) != 3) {
87              printf("Erreur de lecture de l'arc n %d.\n", k + 1);
88              break;
89          }
90          if (u >= 0 && u < g->n_sommets && v >= 0 && v < g->n_sommets) {
91              g->L[u][v] = w;
92              g->P[u][v] = u;
93          }
94      }
95
96      fclose(f);
97      printf("Graphe charge avec succes (%d sommets, %d arcs).\n", g->n_sommets, n_arcs);
98      return 1;
99  }
100
101 void afficher_matrices(Graphe* g, int k) {
102     printf("\n=== ETAPE %d ===\n", k);
103     printf("MATRICE L (Poids) :\n");
104     for (int i = 0; i < g->n_sommets; i++) {
105         for (int j = 0; j < g->n_sommets; j++) {
106             if (g->L[i][j] == INF) printf("  INF ");
107             else printf("%5d ", g->L[i][j]);
108         }
109         printf("\n");
110     }
111
112     printf("\nMATRICE P (Predecesseurs) :\n");
113     for (int i = 0; i < g->n_sommets; i++) {
114         for (int j = 0; j < g->n_sommets; j++) {
115             printf("%5d ", g->P[i][j]);
116         }
117         printf("\n");
118     }
119 }
```

```
120
121 void floyd_warshall(Graphe* g) {
122     if (g->est_charge == 0) return;
123
124     afficher_matrices(g, -1);
125
126     for (int k = 0; k < g->n_sommets; k++) {
127         for (int i = 0; i < g->n_sommets; i++) {
128             for (int j = 0; j < g->n_sommets; j++) {
129                 if (g->L[i][k] != INF && g->L[k][j] != INF) {
130                     int nouveau_poids = g->L[i][k] + g->L[k][j];
131                     if (nouveau_poids < g->L[i][j]) {
132                         g->L[i][j] = nouveau_poids;
133                         g->P[i][j] = g->P[k][j];
134                     }
135                 }
136             }
137         }
138         afficher_matrices(g, k);
139     }
140 }
141
142 int detecter_circuit_absorbant(Graphe* g) {
143     for (int i = 0; i < g->n_sommets; i++) {
144         if (g->L[i][i] < 0) return 1;
145     }
146     return 0;
147 }
148
149 void afficher_chemin_recursif(Graphe* g, int depart, int arrivee) {
150     if (depart == arrivee) {
151         printf("%d", depart);
152         return;
153     }
154     int pred = g->P[depart][arrivee];
155     if (pred == -1) {
156         printf("(Pas de chemin)");
157         return;
158     }
159     afficher_chemin_recursif(g, depart, pred);
160     printf(" -> %d", arrivee);
161 }
```

Annexe main.c

```c
int main() {
    Graphe g;
    initialiser_graphe(&g);

    int continuer_programme = 1;
    char nom_fichier[MAX_FILENAME];

    printf("--- PROJET GRAPHES : FLOYD-WARSHALL ---\n");

    while (continuer_programme == 1) {

        printf("\nEntrez le nom du fichier (ex: graphe.txt) : ");
        scanf("%s", nom_fichier);

        if (charger_graphe_fichier(nom_fichier, &g) == 1) {

            floyd_warshall(&g);

            int circuit_detecte = detecter_circuit_absorbant(&g);

            if (circuit_detecte == 1) {
                printf("\n/!\\ ALERTE : Circuit absorbant detecte.\n");
                printf("Les resultats ne sont pas garantis.\n");
            }

            printf("\n--- RECHERCHE DE CHEMINS ---\n");

            int continuer_chemin = 1;
            int choix = 0;

            while (continuer_chemin == 1) {
                printf("\nVoulez-vous afficher un chemin ? (1: Oui, 0: Non) : ");
                scanf("%d", &choix);

                if (choix == 0) {
                    continuer_chemin = 0;
                } else {
                    int depart, arrivee;

                    printf("Sommet de depart (0 a %d) : ", g.n_sommets - 1);
                    scanf("%d", &depart);

                    printf("Sommet d'arrivee (0 a %d) : ", g.n_sommets - 1);
                    scanf("%d", &arrivee);

                    if (depart < 0 || depart >= g.n_sommets || arrivee < 0 || arrivee >= g.n_sommets) {
                        printf("Erreur : Sommets invalides.\n");
                    }
                    else if (circuit_detecte == 1) {
                        printf("RESULTAT : Circuit absorbant. Chemin indefini.\n");
                    }
                    else if (g.L[depart][arrivee] == INF) {
                        printf("Resultat : Pas de chemin existant (Infini).\n");
                    }
                    else {
                        printf("Chemin minimal (Cout = %d) : ", g.L[depart][arrivee]);
                        afficher_chemin_recursif(&g, depart, arrivee);
                        printf("\n");
                    }
                }
            }
        }

        detruire_graphe(&g);

        printf("\n----------------------------------------\n");
        printf("Traiter un autre fichier ? (1: Oui, 0: Non) : ");
        scanf("%d", &continuer_programme);
    }

    printf("\nFin du programme.\n");
    return 0;
}
```

Fin