Optimisation dans les graphes

Projet : Evaluation de performances d'algorithmes d'optimisation dans les graphes

L'Université Paris-Saclay
UFR SCIENCES d'Orsay

Rédigé par
Rassim Hadj-lazib
Rafik BENNACER

2024/2025

1 Compréhension du problème

Le problème à résoudre concerne l'optimisation des échanges de devises dans un marché forex. L'objectif est de trouver la meilleure séquence d'échanges entre différentes devises pour maximiser le profit, en partant d'une devise initiale et en y revenant après un nombre donné d'échanges.

1.1 Énoncé du problème

Étant donné :
- Un ensemble de devises (Euro, Dollar, Livre, Franc Suisse)
- Une matrice de taux de change entre ces devises
- Un nombre maximal d'échanges autorisés

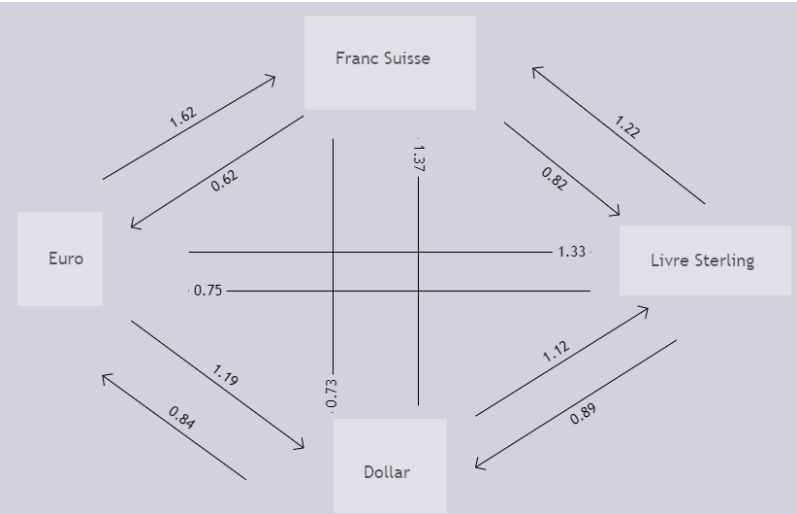
Trouver la séquence d'échanges qui maximise le profit en commençant et terminant par la même devise (généralement l'Euro).

1.2 Modélisation sous forme de graphe

Ce problème se prête naturellement à une modélisation sous forme de graphe pour plusieurs raisons :

Représentation des devises : Chaque devise peut être représentée par un nœud dans le graphe.
Taux de change comme arêtes : Les taux de change entre les devises peuvent être modélisés comme des arêtes pondérées entre les nœuds.
Séquence d'échanges comme chemin : Une séquence d'échanges correspond à un chemin dans le graphe.
Optimisation du profit Trouver le chemin qui maximise le produit des poids des arêtes traversées.

Cette modélisation permet d'appliquer des algorithmes de théorie des graphes pour résoudre efficacement le problème.La matrice par défaut sous forme de graphe ci-dessous.



2 Algorithmes mis en œuvre

Entrée : p (nombre d'échanges), $matrix$ (taux de change entre devises)
Sortie : $meilleur\_profit$, $meilleur\_chemin$

Dijkstra modifié
Initialiser le dictionnaire de correspondance pour les devises :

$$dictionary = \{0 : \text{Euro}, 1 : \text{Dollar}, 2 : \text{Pound}, 3 : \text{Swiss Franc}\}$$

Initialiser les chemins de départ pour chaque devise :

$$chemins = [[\text{Euro}], [\text{Euro}], [\text{Euro}], [\text{Euro}]]$$

Déterminer le nombre de devises :

$$n = \text{longueur}(matrix)$$

Calculer les profits pour $k = 1$:

$$\lambda[j] = matrix[0][j], \quad \forall j \in \{0, 1, \ldots, n-1\}$$

Ajouter la liste λ à la liste complète des profits : $list\_all\_lambda$. Pour chaque $k \in \{2, \ldots, p\}$:

(a) (b)
— — Copier la liste précédente de profits λ_{k-1}.
— Initialiser une nouvelle liste de profits λ pour k et une copie des chemins.
— Pour chaque devise $j \in \{0, 1, \ldots, n-1\}$:
  — Trouver la devise précédente i qui maximise le produit :

$$tmp = \max_i(matrix[i][j] \times \lambda_{k-1}[i])$$

  — Calculer le nouveau profit pour la devise j :

$$\lambda[j] = matrix[i][j] \times \lambda_{k-1}[i]$$

  — Mettre à jour le chemin optimal :

$$chemins[j] = chemins[i] \cup \{dictionary[i]\}$$

— Ajouter la nouvelle liste λ à la liste complète $list\_all\_lambda$.
— Retourner le meilleur profit final et le chemin associé pour revenir à la devise initiale :

$$\lambda[0], chemins[0] \cup \{dictionary[0]\}$$

Programmation dynamique

Initialiser la table DP $dp[i][k][j]$ pour n devises et $p+1$ échanges

Initialiser $path[i][k][j]$ pour suivre le meilleur chemin

Fixer le cas de base :

$$dp[i][0][0] = matrix[i][0], \quad \forall i$$

Pour chaque $k \in \{1, 2, ..., p\}$:

— — Pour chaque devise i (devise de départ) :

— Pour chaque devise j (devise d'arrivée) :

— Initialiser $max\_profit = 0$ et $meilleur\_prev = None$

— Pour chaque devise précédente $prev \in \{0, 1, ..., n-1\}$:

$$profit = dp[prev][k-1][i] \times matrix[i][j]$$

— Si $profit > max\_profit$, alors :

$$max\_profit = profit$$

$$meilleur\_prev = prev$$

— Enregistrer les résultats dans la table :

$$dp[i][k][j] = max\_profit$$

$$path[i][k][j] = meilleur\_prev$$

— Reconstruire le meilleur chemin :

— Initialiser $best\_path = [0]$

— Initialiser $current = 0$

— Pour chaque k allant de p à 1 :

$$current = path[current][k][best\_path[-1]]$$

— Ajouter $current$ au chemin :

$$best\_path.append(current)$$

— Inverser le chemin :

$$best\_path.reverse()$$

— Le chemin final est donné par :

$$best\_path = best\_path[1:]$$

— Retourner $best\_profit$ et $best\_path$

Brute Force
Déterminer le nombre de devises :

$$n = \text{longueur}(matrix)$$

Générer tous les chemins possibles ayant exactement $p+1$ étapes, avec les conditions :

$$\text{chemins} = \{path \mid path[0] = 0 \,\text{et}\, path[-1] = 0\}, \quad path \in \{0, 1, \ldots, n-1\}^{p+1}$$

Initialiser $max\_product = -1$ et $meilleur\_chemin = None$
Pour chaque $path$ dans chemins :
— — Initialiser $product = 1$
— Pour chaque $i \in \{0, 1, \ldots, p-1\}$:

$$product = product \times matrix[path[i]][path[i+1]]$$

— Si $product > max\_product$:
— Mettre à jour $max\_product = product$
— Mettre à jour $meilleur\_chemin = path$
— Retourner le meilleur produit :

$$\text{retourner } max\_product \,\text{et}\, meilleur\_chemin$$

3 Comparaison des complexités

n est le nombre de devises et p le nombre d'échanges.

3.1 Dijkstra modifié :

- Complexité Temporelle : $O(n^2 \cdot p)$.
- Complexité Spatiale : $O(p \cdot n)$.

3.2 Programmation dynamique :

- Complexité Temporelle : $O(n^3 \cdot p)$.
- Complexité Spatiale : $O(n^2 \cdot p)$.

3.3 Brute Force :

- Complexité Temporelle : $O\left(p \cdot n^{p+1}\right)$.
- Complexité Spatiale : $O(n^{p+1} \cdot p)$.

4 Comparaison des Performances des Algorithmes en Fonction du Nombre d'Échanges

Ce graphique compare les temps d'exécution de trois algorithmes en fonction du nombre d'échanges. L'algorithme de force brute, avec une complexité exponentielle, devient rapidement le plus lent. La programmation dynamique reste stable, mais légèrement plus lente que Dijkstra. L'algorithme de Dijkstra, avec sa complexité quadratique, est le plus rapide et efficace, particulièrement pour un grand nombre d'échanges.

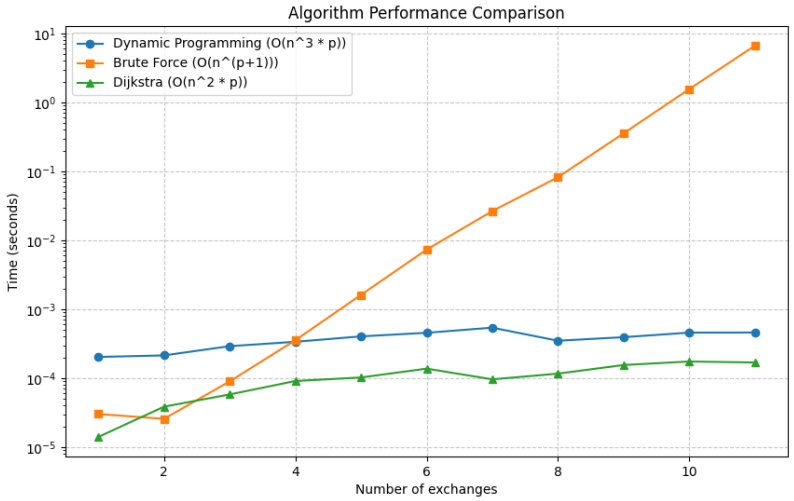


FIGURE 1 – Comparaison des Performances des Algorithmes

5 Prédiction du Temps d'Exécution

```
def predict_execution_time(p, algorithm):
    n = 4  # Number of currencies
    base_time = 1e-6  # Base time for one operation (1 microsecond)

    if algorithm == "dijkstra":
        operations = n**2 * p
    elif algorithm == "brute_force":
        operations = n**(p+1)
    elif algorithm == "dynamic":
        operations = n**3 * p
    else:
        raise ValueError("Unknown algorithm")

    predicted_seconds = operations * base_time

    return predicted_seconds
```

Cette fonction $predict\_execution\_time(p, algorithm)$ calcule le temps d'exécution prédit pour un algorithme donné en fonction de la complexité de cet algorithme et du nombre d'échanges p.

6 Critique des algorithmes et améliorations possibles

6.1 Brute Force :

Avantages : Simple à implémenter, garantit de trouver la solution optimale.

Inconvénients : Extrêmement inefficace pour de grandes valeurs de n ou p.

Améliorations possibles :
Utiliser des techniques de parallélisation pour répartir les calculs.
Implémenter des heuristiques pour élaguer les chemins manifestement sous-optimaux.

6.2 Programmation dynamique

Avantages : Efficace pour des valeurs modérées de n et p, garantit la solution optimale.

Inconvénients : Utilisation importante de mémoire, peut être lent pour de très grandes valeurs de p.

Améliorations possibles :
- Optimisation de l'espace en ne gardant que les deux dernières itérations en mémoire.
- Utiliser des techniques de mémoïzation pour éviter les calculs redondants.

6.3 Dijkstra modifié :

Avantages : Plus efficace en termes de temps et d'espace que les deux précédents pour de grandes valeurs de p.

Améliorations possibles :
- Adapter l'algorithme pour prendre en compte les cycles négatifs (qui représenteraient des opportunités d'arbitrage).
- Utiliser une structure de données plus efficace comme pour améliorer la complexité.

7 Pistes de réflexion et autres modélisations :

7.1 Modélisation stochastique :

- Considérer les taux de change comme des variables aléatoires suivant une distribution probabiliste.

7.2 Optimisation multi-objectifs :

- Introduire d'autres facteurs comme le risque ou les frais de transaction.
- Utiliser des algorithmes d'optimisation multi-objectifs pour trouver un équilibre entre profit et risque.

7.3 Apprentissage par renforcement :

-Modéliser le problème comme un environnement où un agent apprend à prendre des décisions optimales au fil du temps.
- Utiliser des algorithmes comme Q-learning ou les réseaux de neurones profonds pour apprendre des stratégies optimales.

7.4 Analyse en temps réel :

- Adapter les algorithmes pour fonctionner sur des flux de données en temps réel.
- Implémenter des techniques de fenêtre glissante pour ajuster la stratégie en fonction des dernières données.

7.5 Modélisation par graphe temporel :

- Représenter l'évolution des taux de change dans le temps comme un graphe dynamique.
- Appliquer des algorithmes de chemins optimaux dans les graphes temporels pour trouver les meilleures stratégies sur différentes échelles de temps.

Ces approches alternatives pourraient offrir des perspectives intéressantes sur le problème et potentiellement conduire à des stratégies plus robustes et adaptatives dans un environnement forex réel et dynamique.