Algo Bellman–Ford arbitrage

L'algorithme **Bellman–Ford** sert aussi à résoudre le problème du **plus court chemin** dans un graphe pondéré.

Contrairement à Dijkstra, **il accepte les poids négatifs**, et il permet même de **détecter les cycles de poids négatif**.

Soit un graphe avec une source (point de départ) noté `S`.

---

1. Initialisation

- `dist[S] = 0`
- Pour tout autre nœud `v` : `dist[v] = +∞`
- `prev[v] = null` (permet de reconstruire le chemin)

---

2. Relaxation des arêtes (répétée V−1 fois)

L'idée de Bellman-Ford :

Si un chemin plus court existe, il doit être découvert en au plus V−1 détentes ("relaxations"), car un chemin simple ne peut pas contenir plus de `V-1` arêtes.

Donc :

> Pour i allant de 1 à V−1
>
> Pour chaque arête `(u → v)` de poids `w(u,v)` :
>
> - On calcule : `alt = dist[u] + w(u,v)`
> - Si `alt < dist[v]` alors :
> - `dist[v] = alt`
> - `prev[v] = u`

Cette étape va systématiquement essayer **d'améliorer** tous les chemins, et ce **V−1 fois**, jusqu'à stabilisation.

Contrairement à Dijkstra, on ne choisit pas un "minimum temporaire".

On parcourt **toutes les arêtes**, encore et encore, pour propager les meilleurs chemins.

---

3. Détection d'un cycle de poids négatif

Après les `V-1` relaxations, on fait **une dernière passe** :

> Pour chaque arête (u → v) de poids w(u,v) :
>
> Si `dist[u] + w(u,v) < dist[v]`
>
> → alors **un cycle de poids négatif existe** (le coût pourrait diminuer infiniment).

Dans ce cas, l'algorithme **signale un cycle négatif**, et il n'y a pas de solution de plus court chemin (le coût peut devenir arbitrairement petit).

---

4. Rekonstruire un chemin (optionnel)

Comme pour Dijkstra :

Pour obtenir le chemin vers un nœud `T` :

- on part de `T`
- on remonte `prev[T]`, `prev[prev[T]]`, ... jusqu'à `S`
- puis on inverse la liste obtenue

  → On obtient le plus court chemin.

Complexité de Bellman–Ford

Bellman-Ford n'utilise pas de file de priorité.

Il est basé uniquement sur la relaxation répétée de toutes les arêtes.

1. Nombre d'itérations

- Relaxer toutes les arêtes V−1 fois → **O(V)**
- Pendant chaque relaxation, on parcourt toutes les E arêtes → **O(E)**Total : **O(V × E)**

2. Dernière passe pour les cycles négatifs

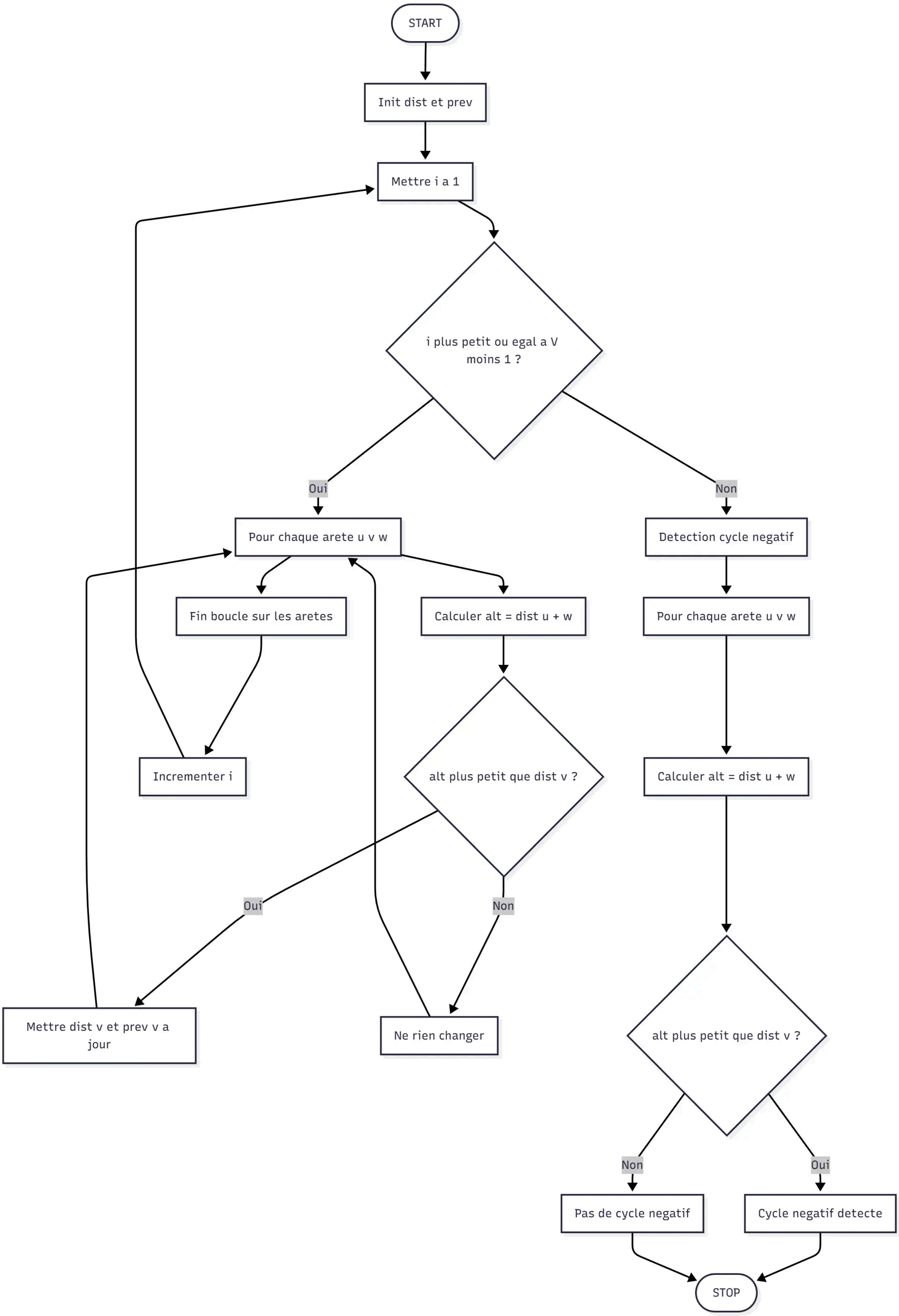
Coût : **O(E)**, déjà dominé par O(VE).

Complexité finale :

O(V × E)

(Plus lent que Dijkstra, mais plus général, car il accepte les poids négatifs.)

Flowshart:


START
Init dist et prev
Mettre i a 1
i plus petit ou egal a V moins 1 ?
Oui
Non
Pour chaque arete u v w
Detection cycle negatif
Fin boucle sur les aretes
Calculer alt = dist u + w
Pour chaque arete u v w
alt plus petit que dist v ?
Incrementer i
Calculer alt = dist u + w
Oui
Non
Mettre dist v et prev v a jour
Ne rien changer
alt plus petit que dist v ?
Non
Oui
Pas de cycle negatif
Cycle negatif detecte
STOP

Apres un peu de recherche j’ai trouvé une utilisation qui me plaisait fortement de l’algo de **Bellman–Ford : l’ utilisation en banque pour faire de l’arbitrage de devises**

```
// Exemple de taux (arbitrage réel)
addEdge(&g, 0, 1, 1.10);    // EUR → USD
addEdge(&g, 1, 2, 150.0);   // USD → JPY
addEdge(&g, 2, 0, 0.0068);  // JPY → EUR
```

j’ai donc mis des exemples de devise et le but sera de détecter des cycle négatif:
résultat de l’algo:

```
ludo@LudoMusk:~/git/SchoolProjet/Dijstra_Metro/Bellman-Ford_arbitrage_entre_devises$ ./arbitrage

 Arbitrage détecté ! Cycle négatif trouvé.
Cycle d'arbitrage potentiel : 1 → 0 → 2 → 1
```