UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

# DEVOIR 3

PAR

JÉRÉMY BOUCHARD (BOUJ08019605)

JEAN-PHILIPPE SAVARD (SAVJ04079609)

ALEXIS VALOTAIRE (VALA09129509)

DEVOIR PRÉSENTÉ À

M. FRANÇOIS LEMIEUX

DANS LE CADRE DU COURS D'ALGORITHMIQUE (8INF433)

*Note :* Le code source LaTeX est disponible à l'URL suivant :
`https://github.com/AlexisCode101/algo-devoir-3`

# Question 1

Nous avons vu en classe un algorithme vorace pour résoudre le problème du retour de la monnaie en un nombre minimum de pièces. Nous avons observé que cet algorithme fonctionne correctement avec de pièces de monnaie canadiennes mais celui-ci ne trouve pas toujours la bonne réponse si, par exemple, des pièces de 12 cents sont ajoutées (e.g. il y a une erreur pour retourner 15 cents). Le problème général du retour de la monnaie peut être résolu exactement en utilisant la programmation dynamique.

Soit n le nombre de pièces distinctes et soit $T[1 \dots n]$ un tableau donnant la valeur de ces pièces (il n'y a aucun intérêt à trier ce tableau). Supposons une quantité illimitée de chaque type de pièces. Soit $L$, une limite sur le montant à obtenir.

**a)** Pour $1 \leq i \leq n$ et $1 \leq j \leq L$, soit $c_{i,j}$ le nombre minimum de pièces pour obtenir le montant $j$ si on se limite aux pièces de type $T[1], T[2], \dots, T[i]$. Si ce montant ne peut être obtenu alors $c_{i,j} = +\infty$. Donnez une équation de récurrence pour $c_{i,j}$ incluant les conditions initiales.

$$c_{i,j} \begin{cases} 1 + c_{i,j}(L - T[i]) & \text{si } L \geq 0 \\ +\infty & \text{ailleurs} \end{cases}$$

**b)** Donnez un algorithme de programmation dynamique pour calculer tous les $c_{n,j}$ où $1 \leq j \leq L$. Votre algorithme ne doit utiliser qu'un seul tableau (à une dimension) de longueur $L$.

**Algorithme 1 :** Trouver $c_{n,j}$

```
   Données : T[1...n], entier n, entier L
   Résultat : Entier c[L]
 1 si L = 0 alors
 2 |  retourner (INT_MAX)
 3 fin
 4 pour j ← 1 à L faire
 5 |  c[j] ← INT_MAX
 6 fin
 7 pour j ← 1 à L faire
 8 |  pour i ← 1 à n faire
 9 |  |  si T[i] ≤ j alors
10 |  |  |  si (j − T[i]) = 0 alors
11 |  |  |  |  temp ← 0
12 |  |  |  fin
13 |  |  |  sinon
14 |  |  |  |  temp ← c[j − T[i]]
15 |  |  |  fin
16 |  |  |  si temp ≠ INT_MAX & (temp + 1) < c[j] alors
17 |  |  |  |  c[j] ← temp + 1
18 |  |  |  fin
19 |  |  fin
20 |  fin
21 fin
```

**c)** Analysez le temps d'exécution de votre algorithme en fonction de $n$ et de $L$.

Table 1: Analyse du temps d'exécution de l'algorithme

| Ligne | Temps/Exécution | Nombre d'exécutions | Ordre |
|---|---|---|---|
| 1 | $c_1$ | 1 | $O(1)$ |
| 2 | $c_2$ | 1 | $O(1)$ |
| 3 | $c_3$ | 1 | $O(1)$ |
| 4 | $c_4$ | $[L]+1$ | $O(L)$ |
| 5 | $c_5$ | $[L]$ | $O(L)$ |
| 6 | $c_6$ | 1 | $O(1)$ |
| 7 | $c_7$ | $[L]+1$ | $O(L)$ |
| 8 | $c_8$ | $[L]\big([n]+1\big)$ | $O(Ln)$ |
| 9 | $c_9$ | $[L][n]$ | $O(Ln)$ |
| 10 | $c_10$ | $[L][n]$ | $O(Ln)$ |
| 11 | $c_11$ | $[L][n]$ | $O(Ln)$ |
| 12 | $c_{12}$ | $[L][n]$ | $O(Ln)$ |
| 13 | $c_{13}$ | $[L][n]$ | $O(Ln)$ |
| 14 | $c_{14}$ | $[L][n]$ | $O(Ln)$ |
| 15 | $c_{15}$ | $[L][n]$ | $O(Ln)$ |
| 16 | $c_{16}$ | $[L][n]$ | $O(Ln)$ |
| 17 | $c_{17}$ | $[L][n]$ | $O(Ln)$ |
| 18 | $c_{18}$ | $[L][n]$ | $O(Ln)$ |
| 19 | $c_{19}$ | $[L][n]$ | $O(Ln)$ |
| 20 | $c_{20}$ | $[L][n]$ | $O(Ln)$ |
| 21 | $c_{21}$ | $[L]$ | $O(L)$ |

À l'aide du tableau ci-dessus, nous pouvons conclure que le temps d'exécution de notre algorithme est de l'ordre de $O(Ln)$.

# Question 2

Donnez toutes les étapes de l'algorithme de Floyd pour calculer le plus court chemin entre toutes les paires de noeuds du graphe de la figure 1. Donnez la matrice obtenue à chaque itération.

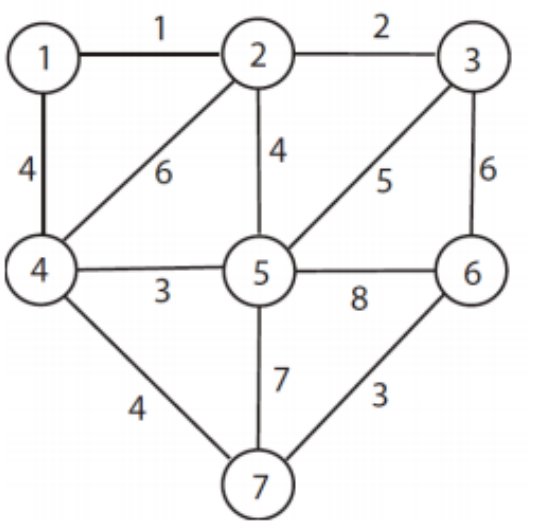


Figure 1: Graphe

La matrice $D_0$ qui suit est la matrice d'adjacence du graphe précédent:

$$D_0 = \begin{bmatrix} 0 & 1 & \infty & 4 & \infty & \infty & \infty \\ 1 & 0 & 2 & 6 & 4 & \infty & \infty \\ \infty & 2 & 0 & \infty & 5 & 6 & \infty \\ 4 & 6 & \infty & 0 & 3 & \infty & 4 \\ \infty & 4 & 5 & 3 & 0 & 8 & 7 \\ \infty & \infty & 6 & \infty & 8 & 0 & 3 \\ \infty & \infty & \infty & 4 & 7 & 3 & 0 \end{bmatrix}$$

Pour la première itération de l'algorithme de Floyd, nous savons que la diagonale de 0, la colonne 1 et la ligne 1 seront semblables à celles de la matrice $D_0$. Ensuite, pour construire le reste de la matrice nous utilisons la formule:

$$D_{\text{k}}[i,j] = Min(D_{\text{k-1}}[i.j].D_{\text{k-1}}[i,k] + D_{\text{k-1}}[k,j])$$

Ce qui donne la matrice suivante, où les éléments [2,4] et [4,2] ont été modifiés:

$$D_1 = \begin{bmatrix} 0 & 1 & \infty & 4 & \infty & \infty & \infty \\ 1 & 0 & 2 & \boxed{5} & 4 & \infty & \infty \\ \infty & 2 & 0 & \infty & 5 & 6 & \infty \\ 4 & \boxed{5} & \infty & 0 & 3 & \infty & 4 \\ \infty & 4 & 5 & 3 & 0 & 8 & 7 \\ \infty & \infty & 6 & \infty & 8 & 0 & 3 \\ \infty & \infty & \infty & 4 & 7 & 3 & 0 \end{bmatrix}$$

Puis, on répète jusqu'à ce qu'on est parcouru les 7 lignes/colonnes, ce qui donne les résultats suivants (tous les éléments sous-lignés ont été modifiés par rapport à la matrice précédente):

$$D_2 = \begin{bmatrix} 0 & 1 & \boxed{3} & 4 & \boxed{5} & \infty & \infty \\ 1 & 0 & 2 & 5 & 4 & \infty & \infty \\ \boxed{3} & 2 & 0 & \boxed{7} & 5 & 6 & \infty \\ 4 & 5 & \boxed{7} & 0 & 3 & \infty & 4 \\ \boxed{5} & 4 & 5 & 3 & 0 & 8 & 7 \\ \infty & \infty & 6 & \infty & 8 & 0 & 3 \\ \infty & \infty & \infty & 4 & 7 & 3 & 0 \end{bmatrix}$$

$$D_3 = \begin{bmatrix} 0 & 1 & 3 & 4 & 5 & \boxed{9} & \infty \\ 1 & 0 & 2 & 5 & 4 & \boxed{8} & \infty \\ 3 & 2 & 0 & 7 & 5 & 6 & \infty \\ 4 & 5 & 7 & 0 & 3 & \boxed{13} & 4 \\ 5 & 4 & 5 & 3 & 0 & 8 & 7 \\ \boxed{9} & \boxed{8} & 6 & \boxed{13} & 8 & 0 & 3 \\ \infty & \infty & \infty & 4 & 7 & 3 & 0 \end{bmatrix}$$

$$D_4 = \begin{bmatrix} 0 & 1 & 3 & 4 & 5 & 9 & \boxed{8} \\ 1 & 0 & 2 & 5 & 4 & 8 & \boxed{9} \\ 3 & 2 & 0 & 7 & 5 & 6 & \boxed{11} \\ 4 & 5 & 7 & 0 & 3 & 13 & 4 \\ 5 & 4 & 5 & 3 & 0 & 8 & 7 \\ 9 & 8 & 6 & 13 & 8 & 0 & 3 \\ \boxed{8} & \boxed{9} & \boxed{11} & 4 & 7 & 3 & 0 \end{bmatrix}$$

$$D_5 = \begin{bmatrix} 0 & 1 & 3 & 4 & 5 & 9 & 8 \\ 1 & 0 & 2 & 5 & 4 & 8 & 9 \\ 3 & 2 & 0 & 7 & 5 & 6 & 11 \\ 4 & 5 & 7 & 0 & 3 & \boxed{11} & 4 \\ 5 & 4 & 5 & 3 & 0 & 8 & 7 \\ 9 & 8 & 6 & \boxed{11} & 8 & 0 & 3 \\ 8 & 9 & 11 & 4 & 7 & 3 & 0 \end{bmatrix}$$

$$D_6 = \begin{bmatrix} 0 & 1 & 3 & 4 & 5 & 9 & 8 \\ 1 & 0 & 2 & 5 & 4 & 8 & 9 \\ 3 & 2 & 0 & 7 & 5 & 6 & \boxed{9} \\ 4 & 5 & 7 & 0 & 3 & 11 & 4 \\ 5 & 4 & 5 & 3 & 0 & 8 & 7 \\ 9 & 8 & 6 & 11 & 8 & 0 & 3 \\ 8 & 9 & \boxed{9} & 4 & 7 & 3 & 0 \end{bmatrix}$$

$$D_7 = \begin{bmatrix} 0 & 1 & 3 & 4 & 5 & 9 & 8 \\ 1 & 0 & 2 & 5 & 4 & 8 & 9 \\ 3 & 2 & 0 & 7 & 5 & 6 & 9 \\ 4 & 5 & 7 & 0 & 3 & \boxed{7} & 4 \\ 5 & 4 & 5 & 3 & 0 & 8 & 7 \\ 9 & 8 & 6 & \boxed{7} & 8 & 0 & 3 \\ 8 & 9 & 9 & 4 & 7 & 3 & 0 \end{bmatrix}$$

La matrice résultante de l'algorithme de Floyd est donc:

$$D_{\text{resultante}} = \begin{bmatrix} 0 & 1 & 3 & 4 & 5 & 9 & 8 \\ 1 & 0 & 2 & 5 & 4 & 8 & 9 \\ 3 & 2 & 0 & 7 & 5 & 6 & 9 \\ 4 & 5 & 7 & 0 & 3 & 7 & 4 \\ 5 & 4 & 5 & 3 & 0 & 8 & 7 \\ 9 & 8 & 6 & 7 & 8 & 0 & 3 \\ 8 & 9 & 9 & 4 & 7 & 3 & 0 \end{bmatrix}$$

# Question 3

L'algorithme de Floyd que nous avons vu en classe donne la longueur du plus court chemin entre toutes les paires de noeuds mais ne permet pas de retrouver les plus courts chemins. Montrer comment modifier l'algorithme afin qu'il mémorise l'information permettant de retrouver efficacement (temps $O(n)$) le plus court chemin entre deux noeuds quelconque du graphe.

Nous savons que l'algorithme de Floyd est :

$$D_{\text{k}}[i,j] = Min(D_{\text{k-1}}[i.j].D_{\text{k-1}}[i,k] + D_{\text{k-1}}[k,j])$$

Or, afin de mémoriser l'information sur le plus court chemin entre deux noeurs quelconque du graphe, il faut ajouter l'équation suivante :

$$P_k[i,j] = \begin{cases} P_{k-1}[k,j] & \text{si } D_k[i,j] \neq D_{k-1}[i,j] \\ P_{k-1}[i,j] & \text{sinon} \end{cases}$$

Cette matrice contient le plus court chemin entre deux noeuds quelconque. Cependant, afin d'obtenir ce chemin, il faut ajouter cet algorithme à la matrice $P$ ainsi complétée :

**Algorithme 2 :** Trouver le plus court chemin avec Floyd

**Données :** $P[1 \dots n, 1 \dots n], entier\ i, entier\ j$
**Résultat :** $tableau\ d'entier\ chemin$

```
1 si P[i][j] = null alors
2 |   retourner
3 fin
4 tant que i ≠ j faire
5 |   j ← P[i][j]
6 |   chemin.append(j)
7 fin
```

**Note :** la donnée $i$ est le noeud de départ et la donnée $j$ est le noeud de fin

# Question 4

Supposez que vous ayez un damier $n \times n$ et un jeton. Vous devez déplacer le jeton depuis le bord inferieur du damier vers le bord supérieur, en respectant la règle suivante. À chaque étape, vous pouvez placer le jeton sur l'un des trois carrés suivants

- le carré qui est juste au dessus,
- le carré qui est une position plus haut et une position plus à gauche (à condition que le jeton ne soit pas déjà dans la colonne la plus à gauche),
- le carré qui est une position plus haut et une position plus à droite (à condition que le jeton ne soit pas déjà dans la colonne la plus à droite).

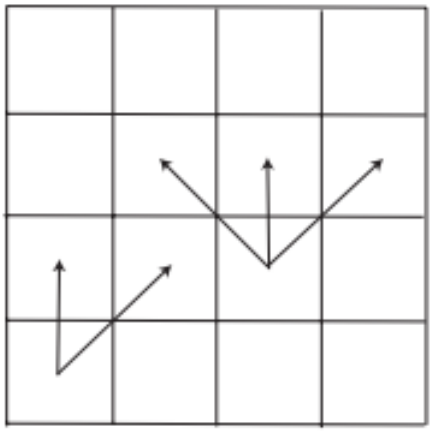

Figure 2: Graphe

La figure 2 illustre les déplacements valides à partir de deux positions dans un damier de dimension $4 \times 4$.

Chaque fois que vous vous déplacez du carré $x = (i, j)$ au carré $y \in \{(i-1, j-1), (i-1, j), (i-1, j+1)\}$, vous recevez $p(x, y)$ dollars. On vous donne $p(x, y)$ pour toutes les paires $(x, y)$ correspondant à un déplacement licite de $x$ à $y$. $p(x, y)$ **n'est pas forcément positif**.

Donner un algorithme efficace de programmation dynamique pour déterminer le montant maximum que vous pouvez empocher. Votre algorithme peut partir de n'importe quel carré du bord inférieur et arriver sur n'importe quel carré du bord

supérieur pour maximiser le montant collecté au cours du trajet.

Expliquez comment vous pouvez optimiser l'espace mémoire et évaluez le temps d'exécution et l'espace mémoire de votre algorithme.

Il est possible de facilement constater que ce problème de programmation dynamique présente une structure caractéristique qui le prête bien à ce genre d'algorithme. En effet, la résolution de ce problème se fait en résolvant les sous-problèmes.

Pour comprendre comment résoudre les sous-problèmes, il suffit de comprendre que, pour obtenir le meilleur résultat, nous n'avons qu'à partir de la fin du problème (du rang n jusqu'au rang 1) et de récursivement descendre en trouvant la case précédente qui nous génère le plus grand montant. Ensuite, nous n'avons qu'à refaire cela pour chaque case final et ainsi obtenir le chemin le plus payant.

Voici un démonstatration rapide de cet algorithme en utilisant une gille 6x6.

| | | | | | | |
|---|---|---|---|---|---|---|
| 6 | | | | | | |
| 5 | | | | | | |
| 4 | | | A | | | |
| 3 | | B | C | D | | |
| 2 | | | | | | |
| 1 | | | | | | |
| | 1 | 2 | 3 | 4 | 5 | 6 |

Pour choisir le bon chemin à partir de A, il suffit de choisir la valeur la plus élevée de la fonction $P$ entre B, C ou D. On construit donc la valeur de $P$ pour le point A à partir des valeurs inférieure.
On obtient donc l'équations suivante:

$$P(A) = max(P(B), P(C), P(D)) + C(A))$$

où $C$ est une fonction qui calcule le coût de l'emplacement.

Il est possible aussi de se rendre compte que si le jeton est sur les côtés, la valeur du

du côté gauche du jeton (s'il est du côté gauche) doit être de -infini pour que celui-ci ne choisisse pas ce chemin. Il est donc possible de facilement définir un algorithme en pseudo-code pour réaliser ceci.

**Algorithme 3 :** Fonction F: Trouver le chemin le plus payant

**Données :** $P[1 \dots n, 1 \dots n]$ *entier* $i$, *entier* $j$
**Résultat :** *valeur de* $i$, $j$

```
si  j < 1 ou j > n alors
    retourner −infini
fin
sinon si  i = 1 alors
    retourner C(i, j)
fin
sinon
    retourner max(F(i − 1, j − 1), F(i − 1, j), F(i − 1, j + 1)) + C(i, j)
fin
```

Il est possible d'analyser cet algorithme:

Pour la complexité temporelle, on voit extrêmement facilement que pour chaque boucle récursive, on descend d'un rang. Puisque l'on descend d'un rang à chaque fois, et qu'il y a $n$ rang, et que dans chaque boucle, choisir la prochaine étape ce fait en $O(3)$ (d'ou $O(1)$).

Il est possible de conclure que la notation grand O de cet algorithme est de $O(n*1) = O(n)$.

Pour la complexité spatiale, on voit également facilement que cet algorithme utilise seulement un tableau de $n$ par $n$. On voit donc que la complexité spatiale est de $O(n^2)$.

Il serait possible d'optimiser cet algorithme en conservant la valeur de C entre chaque exécution. L'avantage de cela serait de ne pas recalculer pour chaque fois que l'on veut la valeur maximale d'une nouvelle case. On pourrait également remplacer le tableau en une simple liste, chaînée ou non, si l'on veut que contenir le chemin et sa valeur jusqu'à maintenant. Cela diminue la complexité spatiale du problème.