

Stratégies et Techniques de Résolution de Problèmes

Chapitre II-2 : Stratégies et Algorithmes

S7 - ECL - 2A - MI

2021-2022

Alexandre Saidi

I- Méta-Stratégies générales de résolution

Soit un ensemble de variables

$$X = \{X_1, \dots, X_n\}$$

et leurs domaines

$$D = \{D_1, \dots, D_n\}.$$

I.1- Les techniques qui regardent en arrière

1. **Générer-Tester** (le plus inefficace) :

Ne considère pas les contraintes lors d'affectation des variables.

Choisit $X_n = d_n \in D_n$ tel que $\{X_1, \dots, X_n\}$ satisfasse les contraintes

≠ trop tard pour se rendre compte des mauvais choix !

2. **Retour arrière** (BackTrack):

Restreint le choix de $d_{k+1} \in D_{k+1}$ pour la variable X_{k+1} (suivant les contraintes)

On essaie une valeur pour X_{k+1} en vérifiant les contraintes avec les valeurs (actuelles) de $X_1 \dots X_k$.

⇒ Il y a quelques variantes de Retour arrière (intelligent, etc.)

I.2- Les techniques qui regardent en avant

3. **Forward Checking** (BT + Arc consistency)

En plus de BT, le choix de d_{k+1} laisse une chance à $X_{k+2} \dots X_n$

Dans une forme partielle, on anticipe seulement sur X_{k+2} .

Les vérifications sont faites entre la **dernière** variable instanciée et les restantes.

Arc consistency : vérification de consistance (vérité des tests) des arcs.

4. Look Ahead (BT + Path consistency)

En plus de (FC), on vérifie la **satisfiabilité** d'une solution possible pour les autres variables (deux à deux) :





on vérifie qu'il y aura non seulement une chance pour toute variable $X_{k+1} \dots X_n$ sachant $X_1 \dots X_k$, mais qu'en plus, $X_{k+1} \dots X_n$ se laissent **deux à deux** une chance possible et satisfaisante.

On remarque que les variables non encore instanciées sont testées 2-à-2 :

→ ce qui ne garantit pas qu'elles seront toutes compatibles.

De fait, dans certains problèmes simples, *Look Ahead* résout la totalité du problème dès les premières instanciations !

1.3- Illustration : exemple N-reines

	Q ₁	Q ₂	Q ₃	Q ₄
1				
2				
3				
4				

Placer 4 reines telles qu'elles ne s'attaquent pas (sur une ligne, colonne et diagonale)

Q_i = le numéro de ligne d'une reine dans la colonne i , $1 \leq i \leq 4$

Les contraintes (*in extenso* pour la clarté) :

$Q_1, Q_2, Q_3, Q_4 \in \{1, 2, 3, 4\}$

$Q_1 \neq Q_2, Q_1 \neq Q_3, Q_1 \neq Q_4,$

$Q_2 \neq Q_3, Q_2 \neq Q_4,$

$Q_3 \neq Q_4,$

$Q_1 \neq Q_2 - 1, Q_1 \neq Q_2 + 1, Q_1 \neq Q_3 - 2, Q_1 \neq Q_3 + 2,$

$Q_1 \neq Q_4 - 3, Q_1 \neq Q_4 + 3,$

$Q_2 \neq Q_3 - 1, Q_2 \neq Q_3 + 1, Q_2 \neq Q_4 - 2, Q_2 \neq Q_4 + 2,$

$Q_3 \neq Q_4 - 1, Q_3 \neq Q_4 + 1$

	Q_1	Q_2	Q_3	Q_4
1				
2				
3				
4				

Méthode Générer-tester :

Au total, 256 évaluations :

64 échecs avec **Q1=1** ($4 \times 4 \times 4 = 64$ possibilités pour Q2, Q3 et Q4)

....

un total de **115 évaluations** pour trouver la première solution.

Méthode Retour arrière :

...

	Q ₁	Q ₂	Q ₃	Q ₄
1				
2				
3				
4				

Méthode Forward Checking (FC) : une couleur par Q_i .

De gauche à droite et du haut vers le bas :

$Q_1=1$ permet d'éliminer les cases bleues (et laisse $\{3, 4\}$ à Q_2)

puis $Q_2=3$ élimine les cases bordeaux (ne laisse rien à Q_3);

= on défait $Q_2=3$

puis $Q_2=4$ et $Q_3=2$ ne laisse pas de chance à Q_4 .

On défait $Q_1=1$

	Q_1	Q_2	Q_3	Q_4
1	●			
2				
3				
4				

	Q_1	Q_2	Q_3	Q_4
1	●			
2				
3		●		
4				

	Q_1	Q_2	Q_3	Q_4
1	●			
2				
3				
4		●		

	Q_1	Q_2	Q_3	Q_4
1	●			
2				
3			●	
4		●		

Suite FC après le placement de Q2, un examen indiv. de Q3 puis Q4 (anticipation)

	Q ₁	Q ₂	Q ₃	Q ₄
1				
2				
3				
4				

	Q ₁	Q ₂	Q ₃	Q ₄
1				
2				
3				
4				


	Q ₁	Q ₂	Q ₃	Q ₄
1				
2				
3				
4				



	Q ₁	Q ₂	Q ₃	Q ₄
1				
2				
3				
4				

Méthode Look Ahead (LA): les couleurs impriment l'étendu des Q_i .



Q1 placé en $\langle 1,1 \rangle$: cela laisse une chance 2-à-2 aux autres

Le placement de Q2 en $\langle 3,2 \rangle$ n'aboutira pas pour les mêmes raisons que dans la stratégie CF.

	Q1	Q2	Q3	Q4
1				
2				
3				
4				

	Q1	Q2	Q3	Q4
1				
2				
3				
4				

- On revient en arrière et on place Q2 e, $\langle 4,2 \rangle$.
- Mais Q2 en $\langle 4,2 \rangle$ ne laisse pas une chance (2-à-2) à

	Q1	Q2	Q3	Q4
1				
2				
3				
4				

Q3 et Q4 : leur seule possibilité est incompatible :

↪ **Échec** de placement de Q2 si Q1 est en $\langle 1,1 \rangle$: on revient en arrière.

../..

On passe à la 2e possibilité de Q1 : Q1 en $\langle 2,1 \rangle$ laisse à Q2 la case $\langle 2,4 \rangle$.

Après le placement de Q2 en $\langle 4,2 \rangle$: on vérifie une chance individuelle pour Q3 et Q4

+ une chance de respecter les contraintes (2 à 2) entre Q3 et Q4.

	Q1	Q2	Q3	Q4
1				
2				
3				
4				

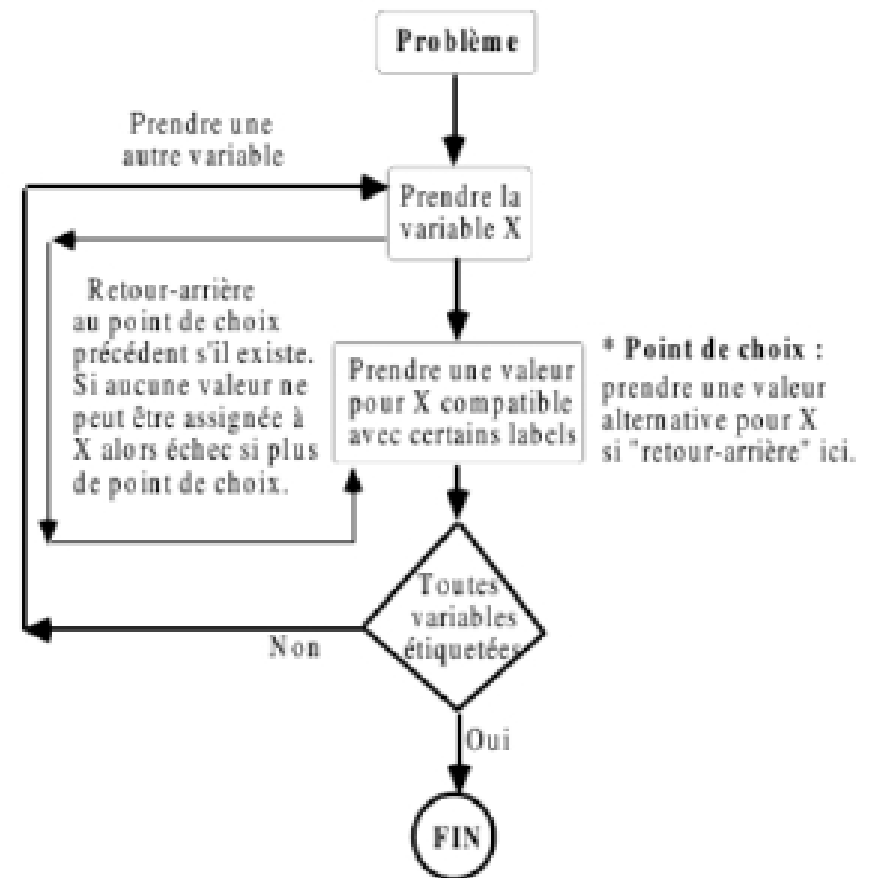
	Q1	Q2	Q3	Q4
1				
2				
3				
4				

	Q1	Q2	Q3	Q4
1				
2				
3				
4				

II- Aspects pratiques et pratiques du Back-tracking

Des exemples d'illustration de BT sont :

- Problème de N-reines
- La somme des sous-séquences (une version simplifiée du problème Sac-à-dos)
- Coloration de graphes
- Problème de circuit Hamiltonien
- Sac à dos généralisé
- Parcours du Cavalier (BE)
- Etc.



Contrôle de l'algorithme de retour-arrière chronologique

II.1- Exemple-1 : placements sous conditions (N-reines)

Représentation :

On décide de consacrer une colonne par pion (simplification des conditions).

Un tableau *Colonne*[1..N] d'entiers donnera la solution où *Colonne*[*i*] contiendra le numéro de la ligne où le pion est placé.

L'algorithme suivant donnera toutes les solutions au problème de N_reines.

```
Procédure N_reines(Colonne: tableau [1..N] d'entier, indice: index)
  Si Prometteur(Colonne, indice) Alors
    Si (indice = N) Alors renvoyer Colonnes[1..N]      // Une solution
    Sinon
      Pour j=1..N                                     // Donnera toutes les solutions
        Colonne[indice+1] = j;
        Si N_reines(Colonne, indice+1) // Pour toutes les solutions, ne conserver de ces 2 lignes que N_reines(Colonne, indice+1)
        Alors renvoyer Colonnes[1..N]
      FinSi                                           // Pas besoin de modifier la case Colonne[indice+1], sa valeur sera modifiée dans cette boucle.
    Fin Pour
  Fin Si
Fin Si
```

La fonction *Prometteur* vérifie les conditions de placement

```
Bool Prometteur(Colonne: tableau [1..N] d'entier, i:index)
  k=1;
  est_prometteur = vrai;
  Tant que k < i ET est_prometteur
    Si (Colonne[i] = Colonne[k]) OU (abs(Colonne[i] - Colonne[k]) = i - k)
      Alors
        est_prometteur = faux
      Fin Si
    k = k+1
  Fin Tant que
  Retourner est_prometteur;
Fin Prometteur
```

Rappel : le tableau *Colonne* contiendra les No lignes où des pions seront placés.

```
// Initialiser la tableau Colonne à (p.ex. -1). Non obligatoire car l'indice 0 permettra à Prometteur de ne pas tester  
Appel:  N=taille(Colonne);  
        afficher(N_reines(Colonne, 0));
```

N.B. : Si on décide d'utiliser FC, la fonction *Prometteur* vérifiera si l'occupation d'une case laisse un voisin possible à cette case.

N.B. : voir aussi le BE sur le Cavalier et les heuristiques utilisées.

Le code Python de cette version :


```
N=5 # On va jsq indice N    (indice 0 non utilisé)
def N_reines(Lst_num_Colonne, indice) :
    if Prometteur_reuse_this(Lst_num_Colonne, indice) :
        if indice == N :
            return Lst_num_Colonne
        else :
            for j in range(1,N+1) :
                Lst_num_Colonne[indice+1]= j
                if N_reines(Lst_num_Colonne, indice+1) :
                    return Lst_num_Colonne

def Prometteur(Lst_num_Colonne, indice) :
    k=1 ; est_prometteur = True
    while k < indice and est_prometteur : # quand indice=1 : on n'a encore rien fait
        if Lst_num_Colonne[indice] == Lst_num_Colonne[k] or \
            abs(Lst_num_Colonne[indice] - Lst_num_Colonne[k]) == indice - k :
            est_prometteur = False
        k += 1
    return est_prometteur

if __name__ == "__main__" :
    Lst_num_Colonne=[-1 for i in range(N+1)]
    # Lst_num_Colonne[1]=1
    print(N_reines(Lst_num_Colonne, 0)) # On aura [-1, 1, 3, 5, 2, 4]
```

Une version AES de ce même algorithme (en Python) :

```
def N_reines_AES(Lst_num_Colonne, indice_a_traiter) :
    if indice_a_traiter == len(Lst_num_Colonne) : return Lst_num_Colonne
    for val_colonne in range(1,len(Lst_num_Colonne)) :
        Lst_num_Colonne[indice_a_traiter] = val_colonne
        if Prometteur(Lst_num_Colonne, indice_a_traiter) :
            if N_reines_AES(Lst_num_Colonne, indice_a_traiter+1) :
                return Lst_num_Colonne
            else : Lst_num_Colonne[indice_a_traiter] = -1
    return None

def Prometteur(Lst_num_Colonne, indice) : // la même que la précédente.
    k=1 ; est_prometteur = True
    while k < indice and est_prometteur :
        if Lst_num_Colonne[indice] == Lst_num_Colonne[k] or \
            abs(Lst_num_Colonne[indice] - Lst_num_Colonne[k]) == indice - k :
            est_prometteur = False
        k += 1
    return est_prometteur

# -----
if __name__ == "__main__" :
    Taille=5
    Lst_num_Colonne=[-1 for i in range(Taille+1)]
    print(N_reines_AES(Lst_num_Colonne, 1))
    # On obtient [-1, 1, 3, 5, 2, 4]          sachant que l'indice 0 n'est pas utilisé (on commence à 1)
```

II.1.1- Complexité de N reines (en nombre d'états visités)

D'une manière générale (stratégie générer-tester), si l'on dessine un arbre (espace d'états) où les nœuds représentent les états successifs (en partant de $i=0$), on aura :

- 1 nœud ($i=0$) à la racine
- N nœuds au niveau 1
- N^2 nœud au niveau 2.
- N^N au niveau N

Le total des nœuds de l'espace d'états $= 1+N+N^2+\dots+N^N = \frac{N^{N+1}-1}{N-1} = O(N^N)$.

Par exemple : pour $N=8$, il y aura un total de 19 173 961 nœuds (combinaisons / états possibles).

Remarque : on peut considérer les nœuds prometteurs en évitant de placer un pion sur une colonne déjà occupée à l'aide de la structure de données choisie (ici, le tableau *Colonne*).

Par exemple, pour $N=8$, on aura la série $1 + 8 + 8 \times 7 + 8 \times 7 \times 6 + \dots + 8!$

Pour N généralisé, on aura la série :

$$1 + N + N(N-1) + N(N-1)(N-2) + \dots + N! = e \cdot N! = O(N!)$$

Rappel : le terme **BT** vaut *back-trcking* (retour arrière) à la manière de l'algorithme AES.

Une **comparaison** des complexités montre le gain important, pour N grand.

- Soit *Algo1*: un algo de parcours en profondeur de l'espace **sans BT** : $O(N^N)$
- *Algo2*: ci-dessus : génère $N!$ Candidats qui placent chaque pion à une ligne et colonne différente.

N	<i>nœuds vérifiés par Algo1</i>	<i>nœuds vérifiés par Algo2</i>	<i>nœuds vérifiés par un BT simple</i>	<i>nbr nœuds prometteurs trouvés par BT</i>
4	341	24	61	17
8	19 173 961	40320	15 721	2057
12	$9.73 * 10^{12}$	$4.79 * 10^8$	$1.01 * 10^7$	$8.56 * 10^5$
14	$1.2 * 10^{16}$	$8.72 * 10^{10}$	$3.78 * 10^8$	$2.74 * 10^7$

Le tableau montre l'efficacité de la méthode **BT + Prometteur**.

Exercice : une technique inspirée de la programmation sous contraintes

Les stratégies FC et LA peuvent être implantées par une technique de propagation de contraintes.

→ L'algorithme N-reines peut profiter de cette technique.

Dans ce cas, lorsqu'un pion est placé, on élimine, pour les autres pions les valeurs qu'ils ne pourront pas prendre.

Par exemple, en plaçant un pion en $\langle 1,1 \rangle$, on peut immédiatement éliminer les cases sur la ligne 1, colonne 1 et la diagonale.

De plus, cette technique autorise l'utilisation des méta-Stratégies telles que Look-Ahead.

Ne pas oublier d'implanter ces principes avec la possibilité de retour arrière.

Exercice : réaliser cette stratégie.

II.2- Exemple-2 : le pb. de la somme des sous ensembles

Cet exemple REPRIS dans la section "Programmation Dynamique".

Un cas simplifié du problème de **Sac-à-dos**.

Énoncé : un "voleur" muni d'un sas-à-dos vole des objets de valeur dans une maison.

- Chaque objet a un poids unique et génère un profit.
- Le but du voleur est de prendre un maximum d'objets sans dépasser la capacité de son sac tout en maximisant le profit.
- Ici : on considère le **même profit pour chaque objet**.

Ce cas simplifié est connu sous le nom du

problème de la somme des sous ensembles.

Remarque : ce problème est différent du problème des sommes du chapitre 1.

Rappelons que dans la version du chapitre-1, la séquence de somme maximum considérée devait être contiguë.

Pour établir une équivalence, il aura fallu trouver la séquence la plus courte / longue avec une somme sous contrainte (e.g. la somme restant inférieure à une certaine limite).

Revenons à notre problème : soit **N** le nombre d'objets et par **W** le poids maximum que le sac à dos pourra supporter sans se déchirer.

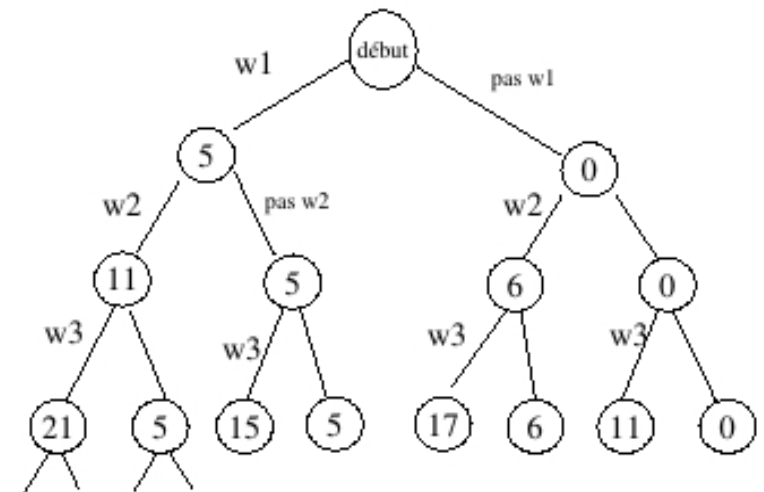
Exemple (avec son espace d'états = arbre en face) :

$N=5$ et $W=21$ avec les objets de poids :

$w_1= 5, w_2 = 6, w_3 = 10, w_4 = 11, w_5 = 16$

⇒ **Trois solutions possibles** :

$\{w_1, w_2, w_3\}, \{w_1, w_5\}$ et $\{w_3, w_4\}$



Un algorithme BT classique examinera toutes les combinaisons possibles (tous sous ensembles : complexité $O(N!)$)

Améliorations : on s'inspire de la PrD et du principe du **A_star** :

Stratégie : une sorte de **propagation de contraintes** pour tenir compte de :

- 1) Soit **Poids_so_far** = la somme des poids des objets déjà ramassés ($w_1 \dots w_i$)

Si *Poids_so_far + w_{i+1} > W* , l'objet w_{i+1} ne sera pas sélectionné

On dira que le nœud w_{i+1} n'est pas Prometteur. Pas la peine d'aller plus loin.

- 2) Soit **Total_poids_restants** = la somme des poids des objets non encore sélectionnés

Après avoir inclus un nœud (w_j) :

Si *Poids_so_far + Total_poids_restants < W* alors on sait par avance que

le nœud actuel ne sera pas Prometteur

(car on ne pourra jamais être égal à W ; ce qui est le but). ../..

Une dernière information peut être également utile :

- 3) Si l'ajout du poids de l'objet w_j est tel que

$$Poids_so_far = W$$

il est évident qu'aucun autre nœud (état suivant l'actuel état) ne doit être recherché (car poids uniques).

Ces trois cas peuvent être illustrés sur l'espace d'état précédent.

L'algorithme **Somme_des_sous_ensembles** suivant utilisera :

N : nombre d'objets

w_i : le poids de l'objet i (par le tableau $w[1..N]$)

Poids_so_far : la somme des poids des objets déjà ramassés

Inclus : tableau de booléens tel que

$Inclus[i]$ = vrai veut dire : w_i est ramassé.

Total_poids_restant : la somme des poids des objets restants

W : le poids maximum à atteindre

Rappel : tous les objets donnent le même profit.

L'algorithme (**BT optimisé qui est une sorte de A^***) suivant donnera **toutes les solutions** pour la séquence $w[1 .. i]$ telle que la somme des $w_j = W$.

Procédure Somme_des_sous_ensembles(i: index; Poids_so_far, Total_poids_restants: entier)

Si Prometteur(i)

Alors

Si Poids_so_far = W

Alors émettre la solution Inclus[1..i]

Sinon

Inclus[i+1] = vrai

Somme_des_sous_ensembles(i+1, Poids_so_far + w[i+1], Total_poids_restants - w[i+1])

Inclus[i+1] = faux // pour essayer d'autres solutions

Somme_des_sous_ensembles(i+1, Poids_so_far, Total_poids_restants - w[i+1])

Fin Si

Fin Si

Fin Somme_des_sous_ensembles

Bool Prometteur(i: index) =

Si (Poids_so_far + Total_poids_restants >= W) && (Poids_so_far = W Ou Poids_so_far + w[i+1] <= W)

Alors retourner Vrai

Sinon retourner Faux

Fin Si

Fin Prometteur

Appel: Somme_des_sous_ensembles(0, 0, Total_tous_les_objets)

où Total_tous_les_objets = la sommes de tous les poids w_i

Complexité :

le nombre de nœuds dans l'espace d'états recherché par l'algorithme est

$$1 + 2 + 2^2 + \dots + 2^N = 2^{N+1} - 1 = O(2^N)$$

- La complexité reste la même si nous voulons **une seule solution**.
- Le cas pire : si la somme de tous les w_i , $i=1..N < W$ mais $w_n=W$
--> Ce qui nécessite de rechercher **tout** l'espace de recherche.

III- Approches de conception d'algorithmes : stratégies

- Nous avons vu les (méta) Stratégies générales (*regard en arrière et en avant*)

Ces principes sont applicables dans toutes stratégies

- Stratégies **Greedy** ou gloutonne : on avance aveuglement étape par étape.

- Dans les parcours de graphes (entre autres), nous avons vu :

Le principe de **Back-Tracking**

../..

- Il existe d'autres stratégies telles que :
 - Stratégies **Diviser et Régner** : division d'un problème en sous-problèmes.
 - Stratégies **Séparer et Régner** : Séparation (!= division) des objectifs et résolution (indépendante) des sous-objectifs suivi d'un assemblage final.
 - Stratégie Forward-checking & Look-Ahead (cf. **A_star**)
 - Stratégie **Programmation Dynamique** (cf. TC1 première année)
 - **Branch & Bound** (optimisation) :
 - trouver une 1^{re} solution, noter son coût (C) puis chercher d'autres solutions de coûts inférieurs à C .
- La meilleure solution retenue a un coût optimal C^* .

III.1- Stratégies Diviser et Régner

Le **principe de base** est de diviser le problème en sous problèmes, de traiter ces sous problèmes puis d'assembler les résultats. Les sous-problème sont en général indépendants.

III.1.1- Quelques exemples

- Multiplication (voir Annexes)
- Recherche dichotomique dans un tableau ou dans un ABOH
- Tri Fusion (Merge-Sort), Tri Rapide (Quick Sort)
- Calcul de Médiane (50eme centile ou percentile) :
 - la moitié est plus petite que la médiane, l'autre plus grand (v. chapitre 1)
- etc.

Voir le chapitre 1 de ce cours pour le Tri Fusion et le Tri Rapide.

III.2- PrD : Principe

- La Programmation Dynamique (Prd) est similaire à *Diviser-Régner* et découpe le problème P_k en sous problèmes P_j , $j < k$.
- La résolution des sous problèmes P_j a en général besoin d'un faible nombre d'information (par rapport à P_k).
- N.B. : Le terme *PrD* vient de la *théorie du contrôle* où « *programmation* » veut dire que l'on utilise un tableau dans lequel les solutions sont construite
(Algorithm Design & applications : M. Goodrich & al. Wiley 2014).

- **Contrairement** à l'approche *descendante* du "Diviser pour Régner", la technique de la PrD procède ensuite par une approche *Ascendante* (*Bottom-up*) pour reconstituer/calculer la solution.
- **Néanmoins**, une version de la PrD utilisant une approche *descendante* (*Top-Down*) existe également.
- Une caractéristique courante de la PrD est la conservation des solutions intermédiaires.

Un exemple : **approche Ascendante** Fibonacci(N) où on stock Fib(1) ... Fib(N-1).

Autre exemple : le problème du rendu de la monnaie (1A)

- Le stockage intermédiaire n'est pas obligatoire dans PrD.

- Comparer par exemple la *multiplication binaire* (ascendant, voir annexes) et le *calcul de la médiane* (descendant) :

→ On résout les instances simples et petites, on stock les résultats.

Plus tard, lorsque l'on a besoin de ces valeurs déjà calculées, on les utilise (pas de re-calcul) pour calculer les instances plus importantes.

Un exemple : calcul de *Fibonacci* et la version récursive versus l'utilisation d'un tableau qui stock les termes précédents).

→ Ce qui caractérise PrD est l'**approche Ascendante** :

$\text{Fib}(0)$ et $\text{Fib}(1) \rightarrow \text{Fib}(2) \rightarrow \text{Fib}(3) \rightarrow \dots \rightarrow \text{Fib}(N)$

Les étapes de PrD :

1. Définir une propriété **récursive** qui donne la solution à une instance du problème
2. Résoudre une instance du problème de façon descendante en résolvant d'abord les instances plus petites puis assembler ces solutions de façon ascendante.

III.2.1- Quelques exemples PrD

- Coefficient Binomial
- Fib (en partant de 1 et jusqu'à N) est un exemple simple
- Factorielle est un cas trivial de PrD
- Distance d'édition (*Levenshtein*)
- Algorithme de Floyd (chemins les plus courts entre toute paire de nœuds d'un graphe valué)
- Recherche dans un AVL (arbre binaire compacte et équilibré)
- Voyageur de commerce (TSP : *traveler sales Person*)
- Etc

III.3- Exemple 1 - calcul binomial

Coefficient binomial $C_k^n = \begin{bmatrix} n \\ k \end{bmatrix} = \frac{n!}{k! \cdot (n-k)!}$ pour $0 \leq k \leq n$

L'approche PrD de résolution de ce problème établit la **propriété récursive** :

$$C_k^n = \begin{bmatrix} n \\ k \end{bmatrix} = \begin{cases} \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + \begin{bmatrix} n-1 \\ k \end{bmatrix} & \text{pour } 0 < k < n \\ 1 & \text{pour } k = 0 \text{ ou } k = n \end{cases}$$

Ce qui donnera l'algorithme **basique** suivant (voir amélioration dans bin2) :

fonction **bin(n, k: entiers)**

Si $k=0$ OU $n=k$

Alors renvoyer 1

Sinon renvoyer $\text{bin}(n-1, k-1) + \text{bin}(n-1, k)$ // Découpage (en n-1) puis assemblage (addition)

Finsi

Fin Bin;

Amélioration : on utilisera une matrice B pour stocker les calculs intermédiaires :

Le contenu de cette matrice 4×4 pour le calcul de $\text{bin}(4, 2)$:

$\text{bin}(4, 2)$ $B[0][0] = 1$

Calcul de la ligne 1 $B[1][0] = 1$

$$B[1][1] = 1$$

Calcul de la ligne 2 $B[2][0] = 1$

$$B[2][1] = B[1][0] + B[1][1] = 1 + 1 = 2$$

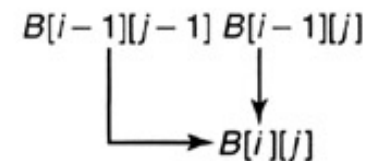
$$B[2][2] = 1$$

Calcul de la ligne 3 $B[3][0] = 1$

$$B[3][1] = B[2][0] + B[2][1] = 1 + 2 = 3$$

$$B[3][2] = B[2][1] + B[2][2] = 2 + 1 = 3$$

	0	1	2	3	4	j	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
i							
n							



Calcul de la ligne 4 $B[4][0] = 1$

$$B[4][1] = B[3][0] + B[3][1] = 1+3 = 4$$

$$B[4][2] = B[3][1] + B[3][2] = 3+3 = 6$$

$$B[4][3] = B[3][2] + B[3][3] = 3+1 = 4$$

$$B[4][4] = B[3][3] + B[3][4] = 1+0 = 1$$

Code Python :

```
import numpy as np
import scipy.special
def bin2 (n : int , k : int) -> int : # version PrD
    B=[[0 for j in range(n+1)] for i in range(n+1)] #matrice (n x n) init 0
    for i in range(n+1) :                          # Découpage ...
        for j in range(min(i, k)+1) :
            if j == 0 or j == i :    B[i][j] = 1
            else : B[i][j] = B[i-1][j-1] + B[i-1][j]    # assemblage
    return B[n][k]

if __name__ == "__main__":
    print(bin2(4,2))
    print("Vérifions : ", scipy.special.binom(4,2))

# On obtient 6 (vérifié)

# N.B. : changer le 2 "for" en "for j in range(min(i,n)+1) : ..." pour avoir bin(n,0..n)
```

	0	1	2	3	4		j	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			

i

n

```

      B[i-1][j-1] B[i-1][j]
        |         |
        +-----> B[i][j]
    
```

Complexité de bin2 :

pour une valeur de i , on note le nombre de passages dans la *boucle j*

Valeur de i	0	1	2	3	4	5	...	$k-1$	k	$k+1$...	n
Nbr. de passages dans la <i>boucle j</i>	1	2	3	4	5	6		k	$k+1$	$k+1$	$k+1$	$k+1$

Le total : $1 + 2 + \dots + k$ jusqu'à $i=k-1$ + $(n-k+1)$ fois $(k+1)$

$$= (2n-k+2)(k+1)/2 = \Theta(nk).$$

Bin2 améliore grandement la version naïve qui utilise la factorielle classique.

N.B. : En s'appuyant sur la version PrD de la factorielle, on obtiendra une complexité $\Omega(nk)$.

Exercice : en quoi l'égalité $C_k^n = C_{n-k}^n$ peut nous aider ?

III.4- Exemple 2 : multiplication de matrices

Rappel : définition des sous-problèmes et condition de **sous-problème optimal** :

On peut caractériser une solution optimale à un sous-problème en termes de solutions à ses sous-problèmes (donc sous-sous-problèmes).

Par exemple, si on doit multiplier une chaîne de matrices $A_0 \dots A_{n-1}$

(1) Les sous-problèmes seront les différents parenthésages **optimaux** de la séquence $A_i \times A_{i+1} \times \dots \times A_j$

→ On découpe donc en sous-problèmes :

pour $A_i \times A_{i+1} \times \dots \times A_j$,

on doit trouver $(A_i \times \dots \times A_k)(A_{k+1} \times \dots \times A_j)$ avec $k \in \{i, i+1, \dots, j-1\}$

Caractérisation des solutions optimales et condition de **sous-structure optimale** :

(2) Quel que soit le choix de k , $(A_i \times \dots \times A_k)$ et $(A_{k+1} \times \dots \times A_j)$ doivent être résolus de façon **optimale** pour avoir un optimum global.

☞ Si cela ne devait pas être le cas, une solution globale optimale aurait une solution non optimale à un de ses sous-problèmes !

Mais cela est impossible car on devrait pouvoir remplacer la solution non optimale par une optimale !

→ Cette observation détermine un problème d'optimisation :

la PrD est un outil d'optimisation.

- Supposons placer k là où on aura un nombre minimal de multiplications et obtenir N_{ij} une solution.

→ N_{ij} sera exprimé en termes de solutions optimales aux sous-problèmes.

III.4.1- Conception de la PrD

Sachant que chaque A_i est une matrice $d_i \times d_{i+1}$, de ci-dessus, on déduit :

$$N_{ij} = \min_{i \leq k < j} \{N_{ik} + N_{k+1j} + d_i d_{k+1} d_{j+1}\} \text{ pour } i : 0..n-1 \text{ avec } N_{ii} = 0 \text{ (une seule matrice)}$$

→ N_{ij} = minimum du nombre de multiplications nécessaires pour chaque sous-expression + le nombre de multiplications pour faire la dernière multiplication de matrices.

Ex. : $A_i \times A_{i+1} \times \dots \times A_j$

$$(A_i \times \dots \times A_k) \quad (A_{k+1} \times \dots \times A_j) \quad \text{avec } k \in \{i, i+1, \dots, j-1\}$$

|←trouver min→| |←trouver min→|

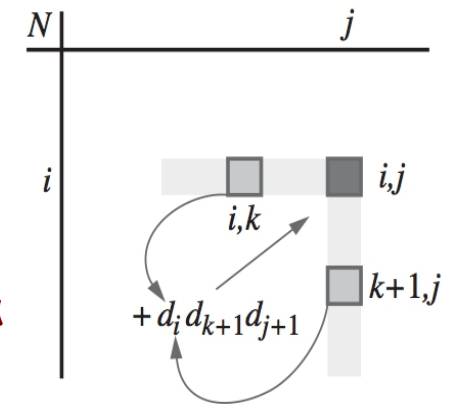
+ |← la multiplication de ces deux matrices →|

L'expression N_{ij} ressemble à celle d'une stratégie *Diviser-Régner* mais seulement en apparence.

- Ici, les sous-problèmes **ne sont pas indépendants** et **partagent des sous-sous-problèmes** qui nous empêchent de couper le problème initial en sous-problèmes indépendants.

On peut calculer les N_{ij} de manière *Ascendante (Bottom-up)* et les stocker dans une table (cf. PrD) :

1. On initialise $N_{ij} = 0$ pour $i = 0..n-1$ puis
 2. L'équation générale de N_{ij} calcule $N_{i+1,j}$ qui ne dépend que de $N_{i,i}$ et de $N_{i+1,i+1}$ qui seront à ce moment-là disponibles.
 3. Avec $N_{i+1,i+1}$, on aura $N_{i+2,i+2}$...
- N_{ij} est donc calculé de manière ascendante et **$N_{0,n-1}$ sera la réponse finale.**



- Ci-dessous l'algorithme dont la complexité est évaluée à $O(n^3)$.

Fonction ChaineMatrices(d_0, \dots, d_n) :

Entrée : une séquence $d_0 \dots d_n$ d'entiers

Sortie : pour $i, j = 0 \dots n - 1$, le nombre minimum de multiplications $N_{i,j}$ nécessaires pour calculer le produit $A_i \cdot A_{i+1} \cdots A_j$ où A_k est une matrice $d_k \times d_{k+1}$

Début

Pour $i \leftarrow 0$ à $n - 1$:

$N_{i,i} \leftarrow 0$

Pour $b \leftarrow 1$ à $n - 1$:

Pour $i \leftarrow 0$ à $n - b - 1$:

$j \leftarrow i + b$

$N_{i,j} \leftarrow \infty$

Pour $k \leftarrow i$ à $j - 1$:

$N_{i,j} \leftarrow \min(N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1})$

Fin ChaineMatrices

III.5- Résumé des étapes de définition d'un schéma PrD

(I) **Sous-problèmes simples** : le problème initial doit pouvoir être décomposé en sous-problèmes avec une *structure similaire* au problème initial (même formule d'optimalité).

De plus il doit y avoir une façon *simple* de définir les sous-problèmes (avec peu d'indices et de variables).

(II) **Optimalité des sous-problèmes** : une solution optimale au problème initial doit être une **composition simple** des solutions optimales aux sous-problèmes.

→ On ne devrait pas pouvoir trouver une solution optimale globale contenant une solution *non-optimale* aux sous-problèmes. La composition doit rester simple.

(III) Recouvrement des sous-problèmes : les solutions optimales aux sous-problèmes indépendants peuvent contenir des sous-problèmes communs (donc des sous-sous-problèmes communs).

Ces recouvrements permettent d'améliorer l'efficacité de l'algorithme PrD en mémorisant les solutions à ces sous-sous-problèmes (pour ne pas refaire des calculs).

Un bon exemple est $\text{Fib}(N)$ utilisant $\text{Fib}(N-1)$ et $\text{Fib}(N-2)$ déjà calculées et stockées.

On pourra utiliser des tableaux / matrices / etc.

III.6- Exemple 3 : Distance de Levenshtein

La *distance de Levenshtein* est une métrique permettant de calculer une *distance d'édition* entre deux mots en vue d'une (proposition de) correction orthographique.

Cette distance représente le nombre **minimum** d'opérations d'édition (*suppression, remplacement et insertion* de caractères) nécessaires pour rendre identiques les deux chaînes de caractères.

Pour deux chaînes de caractères M et M2, le coût minimal calculé est celui de transformer M1 en M2 en effectuant uniquement les opérations élémentaires.

N.B : on ne modifie que la chaîne M1.

Exemples de la distance (d) :

- pour 2 mots identiques ("Ecole" et « Ecole »), $d = 0$
- pour "Ecole" et "Ekole", $d=1$ (remplacement de 'k' par 'c').
- N.B. : une suppression de 'k' puis une insertion de 'c' donnera $d=2$ qui n'est pas minimale.
- pour la correction de "klavié" vers "clavier", on a $d=3$:
 - un remplacement de 'k' par 'c',
 - + un remplacement de 'é' par 'e' et l'ajout de 'r'.
- Les éditeurs de texte qui proposent de remplacer un mot erroné par un autre proposent en général des mots pris dans un dictionnaire et dans l'ordre de cette distance.
- La **complexité** de ce calcul est le produit des longueurs des 2 mots comparés.

III.6.1- Calcul du cout de Levenshtein

Le cout (PrD) de la distance de Levenshtein :

N.B. : $|ch|$ représente la longueur de la chaîne ch .

$$\begin{aligned}
 \text{Dist_Leven}(a,b) &= \max(|a|, |b|) && \underline{\text{si}} \min(|a|, |b|)=0 && \text{un des mots est vide} \\
 &= \text{Dist_Leven}(a[1:], b[1:]) && \underline{\text{si}} a[0]=b[0] \\
 &= 1 + \min \left(\text{Dist_Leven}(a[1:], b), \right. && \underline{\text{sinon}} \\
 &\quad \text{Dist_Leven}(a, b[1:]), \\
 &\quad \left. \text{Dist_Leven}(a[1:], b[1:]) \right)
 \end{aligned}$$

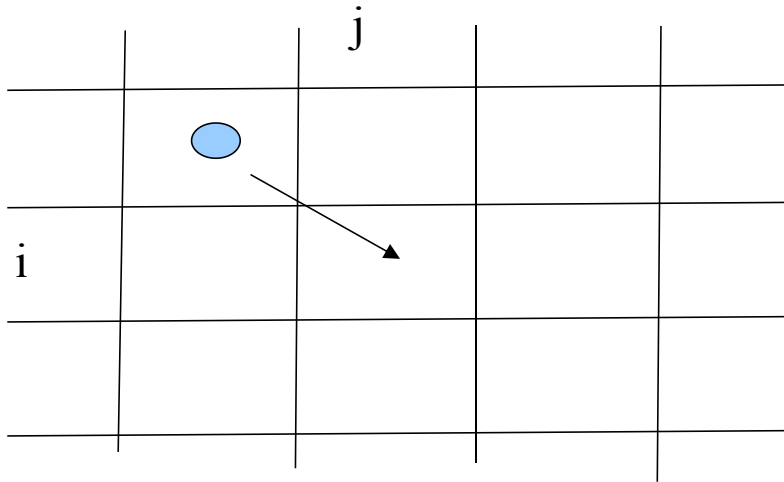
Ce qui donne : ../..

III.6.2- Code du cout de Levenshtein

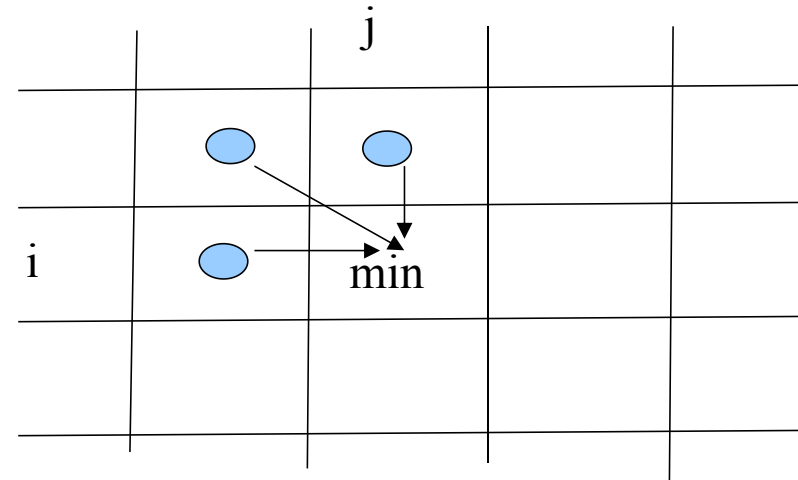
La solution récursive correspondant à la définition précédente donne :

```
def Levenshtein_rec(mot1, mot2) :  
    if mot1=="": return len(mot2)  
    if mot2=="": return len(mot1)  
    if mot1[0] == mot2[0] : return Levenshtein_rec(mot1[1:], mot2[1:])  
    return 1 + min (Levenshtein_rec(mot1[1:], mot2) ,           # sinon  
                    Levenshtein_rec(mot1 , mot2[1:]) ,  
                    Levenshtein_rec(mot1[1:], mot2[1:]))  
  
if __name__ == "__main__":  
    mot1="CHIENS" ; mot2="NICHE"  
    print(Levenshtein_rec(mot1, mot2))  
  
# Réponse : 5
```

Pour comprendre le remplissage de la matrice M , considérons les deux cas de figure de comparaison de 2 lettres (identiques ou différentes) :



$$\text{mot2}[j] = \text{mot1}[i] : \text{mat}[i][j] = \text{mat}[i-1][j-1] + 1$$



$$\text{mot2}[j] \neq \text{mot1}[i] : \text{mat}[i][j] = \min(\text{des 3 cases}) + 1$$

III.6.3- La solution PrD de Levenshtein

La solution PrD avec construction de la matrice :

```
def levenshtein(mot1, mot2):
    M=[[ 0 for j in range(len(mot2)+1)] for i in range(len(mot1)+1)]
    for i in range(len(mot1)+1) : M[i][0]=i      # distance des préfixes du mot1 aux mots vides
    for j in range(len(mot2)+1) : M[0][j]=j      # distance des préfixes du mot2 aux mots vides

    for x in range(1, len(mot1)+1):
        for y in range(1, len(mot2)+1):
            if mot1[x-1] == mot2[y-1]: cout=0
            else : cout=1
            M[x][y] = min( M[x-1][y] + 1,
                          M[x-1][y-1]+cout ,
                          M[x][y-1] + 1
                        )
    print (f'{mot1} en lign et {mot2} en colonne : \n {np.array((M))}')
    return (M[len(mot1)+1 - 1][len(mot2)+1 - 1])

if __name__ == "__main__":
    mot1="CHIENS" ; mot2="NICHE"
    print(levenshtein(mot1, mot2))
```


Illustration (thanks to Wiki pour les 2 figures) :

- Pour mot1="CHIENS" et mot2="NICHE".
- Le coût des substitutions selon la règle :

Soit $\text{Cout}[i, j] = 0$ si $a[i] = b[j]$ et

$\text{Cout}[i, j] = 1$ si $a[i] \neq b[j]$.

- On a donc ici la matrice **Cout** lettre par lettre :

Matrice Cout

	C	H	I	E	N	S
N	1	1	1	1	0	1
I	1	1	0	1	1	1
C	0	1	1	1	1	1
H	1	0	1	1	1	1
E	1	1	1	0	1	1

- La matrice est ensuite remplie par le code Python précédent pour cumuler les couts pour les 2 mots.

Chaque ligne i de cette matrice est telle que tout au long de l'algorithme, $\text{ligne}[i][k]$ contienne la distance de **Levenshtein** entre les k premières lettres de mot1 et les i premières lettres de mot2.

Au début, $i=0$, et la distance entre les k premières lettres de mot1 et la chaîne vide vaut k (car il faut k suppressions pour passer des k premières lettres de mot1 à la chaîne vide).



		C	H	I	E	N	S
↩	0	1	2	3	4	5	6
N	1	1	2	3	4	4	5
I	2	2	2	2	3	4	5
C	3	2	3	3	3	4	5
H	4	3	2	3	4	4	5
E	5	4	3	3	3	4	5

La case toute ne bas à droite donne le cout pour rendre identique ces 2 mots.

- L'intérêt de la stratégie **PrD** est de calculer les distances successives à l'aide de précédentes (en phase ascendante).

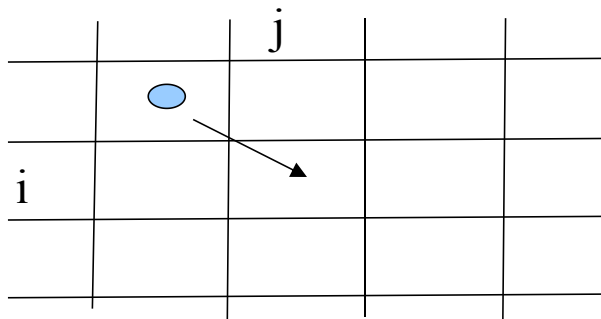
La phase descendante est ici réduite aux initialisations.

Un autre exemple : les mots "klavié" vs "clavier"

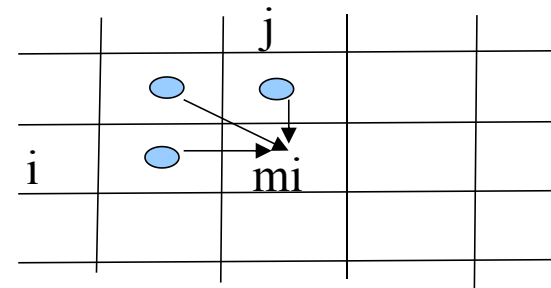
	k	l	a	v	i	é
0	1	2	3	4	5	6
c	1	1	2	3	4	5
l	2	2	1	2	3	4
a	3	3	2	1	2	3
v	4	4	3	2	1	2
i	5	5	4	3	2	1
e	6	6	5	4	3	2
r	7	7	6	5	4	3

← il faudra 3 opérations pour rendre "klavié" identique à "clavier"

Rappel : pour comprendre le remplissage de la matrice M, considérons les deux cas de figure de comparaison de 2 lettres (identiques ou différentes) :



$$\text{mot2}[j] = \text{mot1}[i] : \text{mat}[i][j] = \text{mat}[i-1][j-1] + 1$$



$$\text{mot2}[j] \neq \text{mot1}[i] : \text{mat}[i][j] = \min(\text{des 3 cases}) + 1$$

Variantes de Levenshtein :

- Dans une variante (Daereau-Levenshtein) d'édition, on peut également procéder à **intervertir** (transposer) deux caractères adjacents.
- La distance de **Hamming** est une variante qui ne permet que la substitution.
- La distance de Levenshtein a donné lieu à une variante appliquée en séquençement d'ADN (*Smith-Waterman*).

Le principe de PrD appliqué au calcul de la distance de Levenshtein de 2 mots est de construire une matrice contenant la distance entre tous les préfixes des deux mots.

→ Ainsi, la dernière valeur calculée donnera la distance entre les deux mots.

III.7- Exemple 4 : algorithme de Floyd

Propos : calcul des chemins **entre toutes paires** de nœuds d'un graphe.

Utilisation : trafic aérien (en l'absence de ligne directe), routage (adresses) dans les réseaux et routage de messages dans un réseau global, ...

Ex : dans ce graphe (valué orienté), on cherche le meilleur chemin entre v1 et v3.

On a les **candidats** avec leurs **valeurs**:

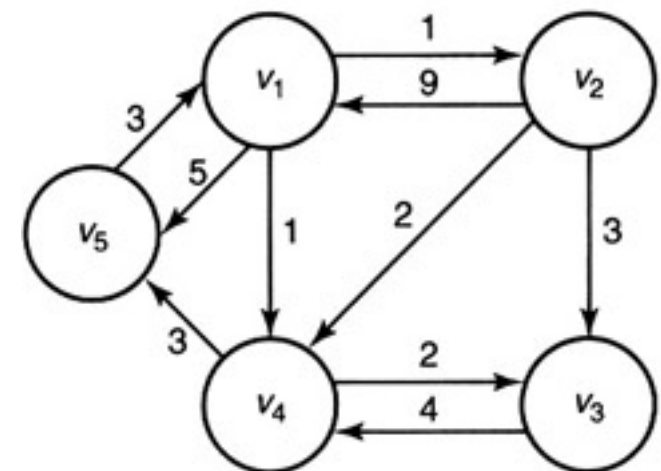
$$\text{longueur}(\langle v1, v2, v3 \rangle) = 1 + 3 = 4$$

$$\text{longueur}(\langle v1, v4, v3 \rangle) = 1 + 2 = 3$$

$$\text{longueur}(\langle v1, v2, v4, v3 \rangle) = 1 + 2 + 2 = 5$$

Le chemin $\langle v1, v4, v3 \rangle$ semble le plus court.

→ Un problème d'optimisation



../..

- La recherche du chemin le plus court est un problème **d'optimisation** :
 - Il existe plusieurs chemins (**candidats**) différents entre deux nœuds,
 - On souhaite calculer le plus court (de **valeur optimale**)
 - Il peut exister plusieurs chemins de longueur minimale entre 2 nœuds.
 - Le choix est dans ce cas aléatoire.
- Un algorithme **basique** peut calculer TOUS les chemins entre TOUS couples de nœuds puis de choisir le plus court.
- Un tel algorithme (pour un graphe fortement connexe) est de **complexité exponentielle** $(N-2)!$ pour N nœuds.
 - > A l'aide de la PrD, on peut obtenir une complexité cubique.

Démarche : on définit une matrice W des poids (distances directes) avec la convention :

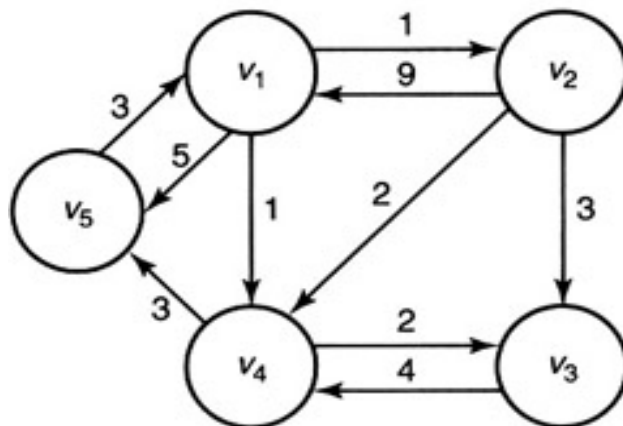
$$W[i][j] = 0 \quad \text{si } i=j$$

$$W[i][j] = \infty \quad \text{si } i \text{ et } j \text{ non directement connectés}$$

$$W[i][j] = \text{la distance (arc) entre } i \text{ et } j$$

Pour le graphe donné, on aura la matrice initiale des distances W et

D = la matrice des distances (chemins) les plus courts.



	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

On notera $D^{(k)}[i][j]$ = la longueur du chemin le *plus court* entre i et j utilisant seulement les nœuds $\{v_1, v_2, \dots, v_k\}$.

Par définition, on a (pour N nœuds) :

- $D^{(0)}[i][j]$ = chemin passant par aucun autre nœud que i et j = la matrice W
- $D^{(n)}[i][j]$ = chemin passant par n'importe quel autre nœud du graphe
= la matrice finale D (meilleurs chemins)

Exemples :

$$D^{(0)}[2][5] = \text{longueur}(\langle v_2, v_5 \rangle) = \infty \quad (\text{pas de chemin direct})$$

chemin le plus court entre v_2-v_5 en passant par $v_0 = \text{direct}$ (v_0 fictif)

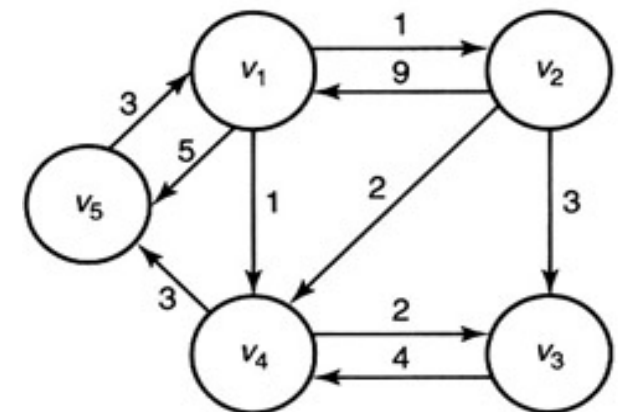
$$D^{(1)}[2][5] = \text{minimum}(\text{longueur}(\langle v_2, v_1, v_5 \rangle), \text{longueur}(\langle v_2, v_5 \rangle))$$

$$= \text{minimum}(14, \infty) = 14 \quad (\text{non concerné par } \langle v_2, v_1, v_4, v_5 \rangle : v_4 \text{ ne "joue" pas encore !})$$

$$D^{(2)}[2][5] = D^{(1)}[2][5] = 14 \quad \text{aucun chemin partant de } v_2 \text{ ne peut repasser par } v_2 \text{ (vrai } \forall \text{ graphe)}$$

$$D^{(3)}[2][5] = D^{(2)}[2][5] = 14 \quad \text{l'inclusion de } v_3 \text{ (seul) n'apporte rien de plus dans ce graphe}$$

→ $\langle v_2, v_3, v_5 \rangle = \langle v_2, v_5 \rangle$, on ne passe pas par v_4 , v_4 ne "joue" pas encore !).



Donc, on obtiendra la matrice D depuis W en procédant (principe de la PrD):

Etape1 : définir une propriété récursive (calcul) pour obtenir $D^{(k)}$ depuis $D^{(k-1)}$.

Etape2 : Résoudre une instance du problème (ascendant) en répétant l'étape (1) pour $k=1 \dots n$. Ce qui crée la séquence : $D^{(0)}$, $D^{(1)}$, $D^{(2)}$, ... , $D^{(n)}$.

Deux cas peuvent se présenter pour l'**étape 1**:

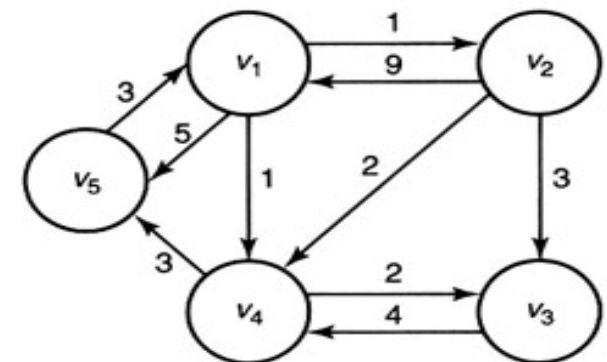
Cas 1 : au moins un des chemins les plus courts de i à j n'utilisera pas v_k dans

$\{v_1, \dots, v_k\}$: $D^{(k)}[i][j] = D^{(k-1)}[i][j]$ (1)

Par exemple, dans le graphe précédent :

$$D^{(5)}[1][3] = D^{(4)}[1][3] = 3$$

Car l'inclusion de v_5 à $\{v_1, v_4, v_3\}$ n'apporte rien au chemin le plus court qui reste $\langle v_1, v_4, v_3 \rangle$.



Cas 2 (de l'étape 1) : les chemins les plus courts allant de v_i à v_j utilisent seulement $\{v_1, v_2, \dots, v_k\}$ utilisent v_k .

Dans ce cas, tout chemin le plus court est décrit dans le schéma ci-contre.

Le chemin le plus court de v_i à v_j utilisant seulement $\{v_1, v_2, \dots, v_k\}$



Sachant que v_k ne peut pas être un nœud intermédiaire du sous chemin v_i à v_k , ce sous chemin utilise seulement les nœuds $\{v_1, v_2, \dots, v_{k-1}\}$.

Le chemin le plus court de v_i à v_k utilisant seulement $\{v_1, v_2, \dots, v_k\}$

Le chemin le plus court de v_k à v_j utilisant seulement $\{v_1, v_2, \dots, v_k\}$

Ce qui implique que le sous-chemin sera de la même longueur que $D^{(k-1)}[i][k]$.

$$D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]. \quad (2)$$

$$\text{Exemple : } D^{(2)}[5][3] = D^{(1)}[5][2] + D^{(1)}[2][3] = 4 + 3 = 7.$$

Sachant que nous serons dans l'un des 2 cas ci-dessus, la valeur de $D^{(k)}[i][j]$ est le minimum de la valeur à droite des équations (1) et (2).

Ce qui veut dire que l'on détermine $D^{(k)}[i][j]$ depuis $D^{(k-1)}[i][j]$ par :

$$D^{(k)}[i][j] = \text{minimum}(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j]) . \quad \text{c-à-d. } D^{(k)}[i][j] = \min(\text{cas1}, \text{cas2})$$

Fin de l'**Etape 1** ci dessus.

Pour l'**étape 2**, on développera la propriété récursive de l'étape 1 pour créer la séquence $D^{(0)}$, $D^{(1)}$, $D^{(2)}$, ..., $D^{(n)}$.

Exemples :

$$D^{(2)}[5][4] = \min(D^{(1)}[5][4], D^{(1)}[5][2] + D^{(1)}[2][4])$$

(rappel : $D^0 = W$ = la matrice initiale) :

$$D^{(1)}[2][4] = \min(D^{(0)}[2][4], D^{(0)}[2][1] + D^{(0)}[1][4]) = \min(2, 9+1) = 2$$

$$D^{(1)}[5][2] = \min(D^{(0)}[5][2], D^{(0)}[5][1] + D^{(0)}[1][2]) = \min(\infty, 3+1) = 4$$

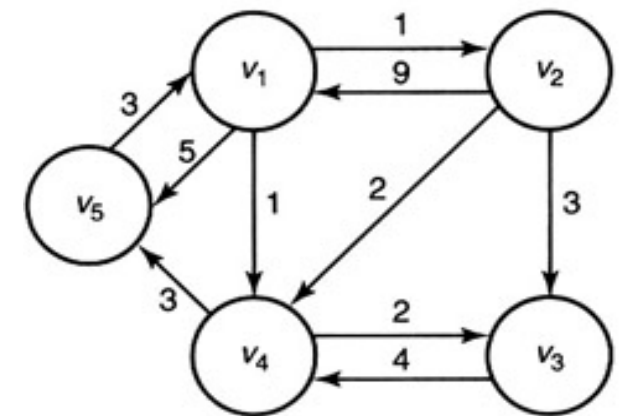
$$D^{(1)}[5][4] = \min(D^{(0)}[5][4], D^{(0)}[5][1] + D^{(0)}[1][4]) = \min(\infty, 3+1) = 4$$

$$D^{(2)}[5][4] = \min(D^{(1)}[5][4], D^{(1)}[5][2] + D^{(1)}[2][4]) = \min(4, 3+2) = 4$$

Après avoir calculé tous les $D^{(2)}$, on continuera jusqu'à $D^{(5)}$.

--> couverture de tous les noeuds.

Ces calculs donneront la matrice D.



	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

III.7.1- Algorithme de Floyd

```
def floyd (n : int , W: matrice, D: matrice) -> None :  
    D=W  
    for k in range(1, n+1):  
        for i in range(1, n+1):  
            for j in range(1, n+1):  
                D[i][j] = min(D[i][j], D[i][k] + D[k][j]);    # même principe utilisé dans Dijkstra (voir plus loin)
```

N.B. : on peut remarquer la trame de l'algorithme de Dijkstra.

La complexité de cet algorithme : $\Theta(n^3)$

La matrice **finale** (D) des chemins les plus courts :

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

III.7.2- Obtention des trajets (coming from)

On enregistre dans la nouvelle matrice $CF[u][v]$ le plus grand numéro du nœud intermédiaire sur le chemin le plus court $u \rightarrow v$.

$CF[u][v]=0$ si ce nœud n'existe pas.

```
def floyd2 (n : int , W: matrice, D: matrice, CF : matrice) -> None :  
    for i in range(1, n+1):  
        for j in range(1, n+1):  
            CF[i] [j] = 0;  
    D = W  
    for k in range(1, n+1):  
        for i in range(1, n+1):  
            for j in range(1, n+1):  
                if D[i][k] + D[k][j] < D[i][j] :  
                    CF[i][j] = k  
                    D[i][j] = D[i][k] + D[k][j]
```

Affichage du chemin :

```
Def affiche_path (u : int , v : int ) :    # trajet pour aller de u  à  v
  if CF[ u ][ v ] != 0 :
    affiche_path (u, CF[u] [v])
    Print( " --> " , CF[ u ][ v ])
    affiche_path (CF[ u ][ v ], v)
```

III.7.3- Programmation Dynamique et optimisation

L'algorithme de **Floyd** permet de déterminer le chemin le plus court et de le calculer.

Comme étudié ci-dessus, la construction de la solution optimale devient donc la 3e étape du développement d'un problème d'optimisation.

Les étapes de PrD appliquée à un problème d'optimisation devient :

1. Définir une propriété récursive qui donne la solution à une instance du pb.
2. Calculer la valeur de la solution optimale de chaque instance
3. Construire la solution optimale de façon ascendante en utilisant les instances plus petites.

La PrD **peut ne pas être adaptée** à un problème d'optimisation si le principe suivant n'est pas respecté :

Le principe d'optimalité : *une solution optimale à un problème contient des solutions optimales à toutes les sous instances de ce problème (et vice versa).*

Appliquons ce principe au problème des chemins les plus courts :

Le chemin le plus court $v_i \rightarrow v_j$ en passant par le nœud intermédiaire v_k est optimale si les chemins partiels $v_i \rightarrow v_k$ et $v_k \rightarrow v_j$ sont optimaux (ici : chemins les plus courts).

Il faudra donc d'abord démontrer l'application du principe d'optimalité avant de faire l'hypothèse d'optimalité de la solution avec PrD. ../..

Contre exemple :

calculer les chemins **les plus longs** sur le graphe ci-contre.

N.B. : on se restreint à ne pas exploiter le circuit (V2,V3) dans ce graphe.

sinon, on peut passer une infinité de fois dans ce cycle.

On a le chemin le plus long optimal entre $v_1 \rightarrow v_4$:

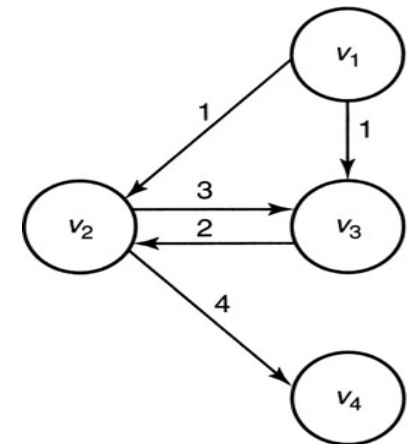
$$\langle v_1, v_3, v_2, v_4 \rangle = 7$$

Cependant, le sous-chemin $v_1 \rightarrow v_3$ **n'est pas optimal** car :

$\text{longueur}(\langle v_1, v_3 \rangle) = 1$ et $\text{longueur}(\langle v_1, v_2, v_3 \rangle) = 4$ (sans circuit)

Or, la solution optimale finale $\langle v_1, v_3, v_2, v_4 \rangle$ ne contient pas $\langle v_1, v_2, v_3 \rangle$.

\Rightarrow Le principe d'optimalité ne s'applique pas ici.



- La solution optimale passera parfois par une optimisation des données.

III.7.4- Variant de Floyd : algorithme de base de Roy-Warshall

Une variante de l'algorithme de Floyd :

Calculer les plus courts chemins comportant au plus k arcs.

On calcule les deux matrices W et D de la même manière :

Init : Matrice W : pour tous les couples x, y on pose

$W[x, y] = \text{valeur}(a)$ s'il existe un arc a entre x et y , ∞ sinon.

Algo : Matrice D (meilleures distances de longueur k arcs) est une évolution de la matrice W suivant le principe suivant :/..

Principe de l'algorithme :

$$D = W$$

Pour $k = 1, 2, 3, \dots, n$ faire

 Pour tout couple de nœuds x, y

 Pour tout sommet z successeur de x

$$D[x, y] = \min(D[x, y], \quad D[x, z] + D[z, y])$$

La preuve de la justesse est apportée par une récurrence sur k .

N. B. : pour obtenir effectivement des trajets, une variante à *Coming_From* est d'utiliser une matrice *suivant* $[x][y]$ qui contiendra le sommet qui suit x dans le trajet qui mène de x à y .

Ce qui donne l'algorithme Roy Warshall de complexité $O(n^3)$:

Les chemins les plus courts de longueur au plus k entre tout couple de nœuds :

Pour k' de 1 à k faire

 Pour i de 1 à n faire

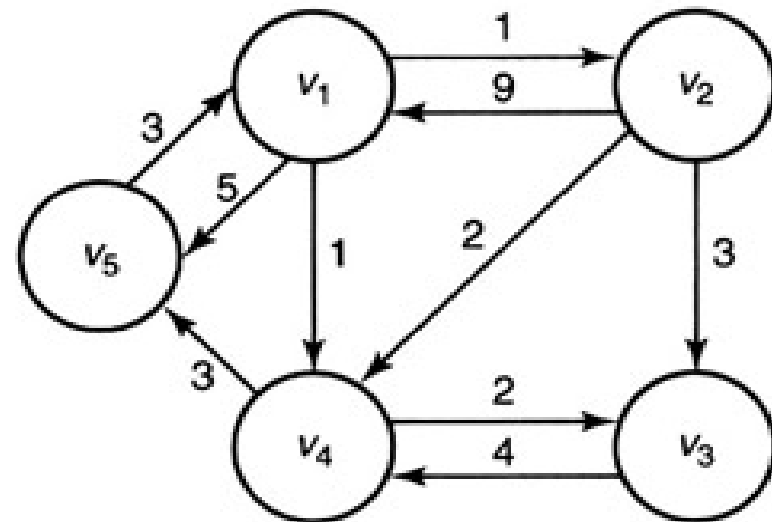
 Pour j de 1 à n faire

 Si $D[i][j] > D[i][k'] + D[k'][j]$

 Alors $D[i][j] = D[i][k'] + D[k'][j]$

Exemple ($k=2$) : la matrice des distances :

	V1	V2	V5	V4	V3
V1	999	1	5	1	4
V2	999	999	999	2	3
V5	3	4	8	4	7
V4	999	999	3	999	2
V3	999	999	999	4	999



III.8- Exemple 5 : le voyageur de commerce (TSP)

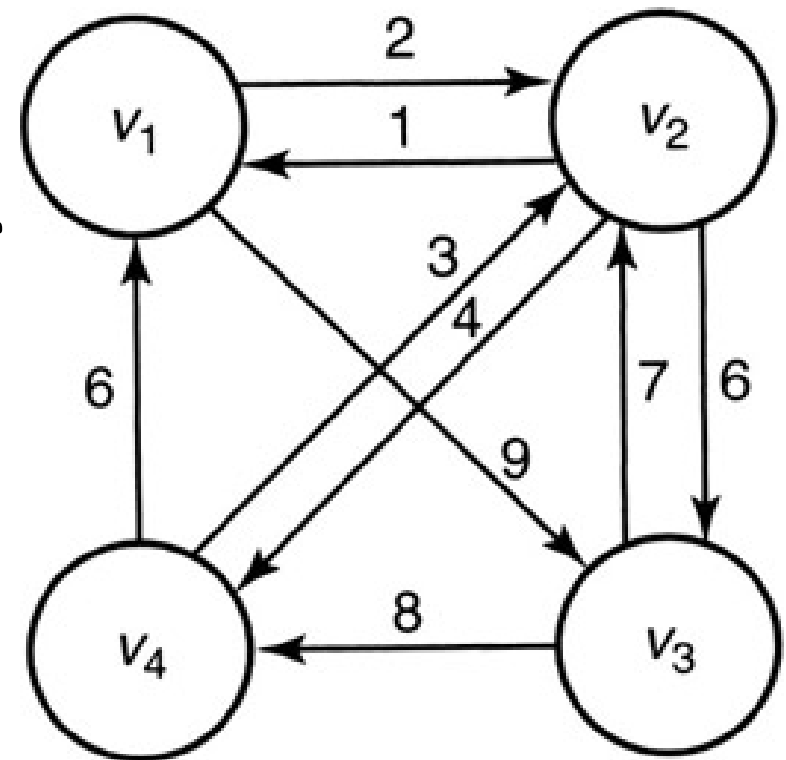
Problème : trouver, dans un graphe de villes et planifier un voyage de **coût** minimal où le voyageur visite chaque nœud (ville) une seule fois et **revient** à sa base (son point de départ).

Dans l'exemple, on a :

$$\text{longueur}(\langle v_1, v_2, v_3, v_4, v_1 \rangle) = 22$$

$$\text{longueur}(\langle v_1, v_3, v_2, v_4, v_1 \rangle) = 26$$

$$\text{longueur}(\langle v_1, v_3, v_4, v_2, v_1 \rangle) = \mathbf{21}$$



⇒ Comme pour le calcul des chemins les plus courts, la **complexité de ce calcul est exponentielle**.

PrD peut-elle être appliquée ici ?

On vérifie son principe :

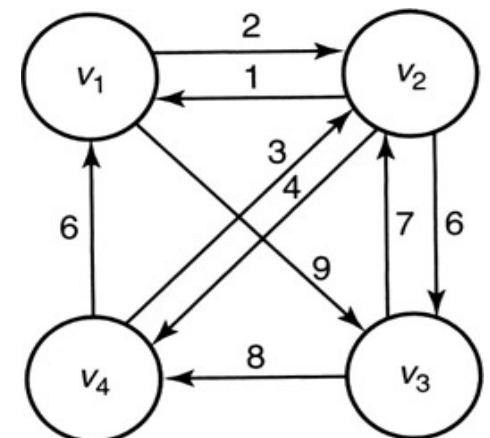
On note que sur la tournée optimale, si v_k est le premier nœud avant v_1 , alors le sous chemin de v_k à v_1 doit être le chemin le plus court de $v_k \rightarrow v_1$ qui passe une fois par tous les autres nœuds.

→ Le principe d'optimalité s'applique et la PrD peut être utilisée.

Pour ce faire, on représente la graphe par sa matrice d'adjacence W (cf. Floyd) :

On peut poser ce problème :

Calculer la matrice D des distances telle que $D[v_i][A] =$ la longueur du chemin le plus court de v_i à v_1 en passant une seule fois par tous les nœuds de l'ensemble A ($V_1 = \text{départ}$)



Pour le graphe ci-dessus, on a $V=\{v_1, v_2, v_3, v_4\}$ (V_1 = départ)

- Si $A=\{v_3\}$, alors $D[v_2][A] = \text{longueur}(\langle v_2, v_3, v_1 \rangle) = 15$

- Si $A=\{v_3, v_4\}$, alors $D[v_2][A] =$

$$\min(\text{longueur}(\langle v_2, v_3, v_4, v_1 \rangle), \text{longueur}(\langle v_2, v_4, v_3, v_1 \rangle)) = \min(20, \infty) = 20$$

Rappel : la valeur ∞ : pas d'arc (direct) entre V_4 et V_3 .

Avec $V-\{v_1, v_j\}$ = tous les nœuds (sauf ces 2 là) et le principe d'optimalité :

longueur de la meilleure tournée =

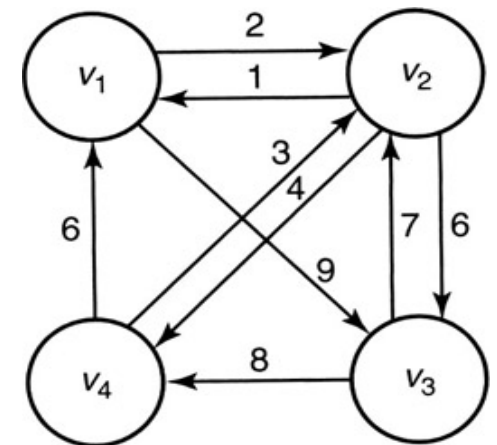
$$\min(W[1][j] + D[v_j][V-\{v_1, v_j\}]) \quad 2 \leq j \leq n$$

Et en général, pour $i \neq 1$ et $v_i \notin A$:

- $D[v_i][A] = \min(W[i][j] + D[v_j][V-\{v_j\}])$

si $A \neq \emptyset$, $j : v_j \in A$, (\emptyset = ensemble vide)

- $D[v_i][\emptyset] = W[i][1]$



On utilisera cette dernière égalité ($D[v_i][\emptyset] = W[i][1]$) pour construire l'algorithme de la PrD de ce problème.

Notons alors que :

$$D[v_2][\emptyset] = W[2][1] = 1$$

$$D[v_3][\emptyset] = \infty$$

$$D[v_4][\emptyset] = 6$$

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

Ensuite, on considère tous les ensembles contenant un élément :

$$\begin{aligned}
 \bullet D[v_3][\{v_2\}] &= \min(W[3][j] + D[v_j][\{v_2\} - \{v_j\}]) \quad j : v_j \in \{v_2\} \\
 &= W[3][2] + D[v_2][\emptyset] = 7+1=8
 \end{aligned}$$

De manière similaire :

$$D[v_4][\{v_2\}] = 3+1 = 4$$

$$D[v_2][\{v_3\}] = 6+\infty = \infty$$

$$D[v_4][\{v_3\}] = \infty + \infty + \infty = \infty$$

$$D[v_2][\{v_4\}] = 4+6 = 10$$

$$D[v_3][\{v_4\}] = 8+6 = 14$$

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

Considérons ensuite les ensembles à 2 éléments :

$$\begin{aligned}
 D[v_4][\{v_2, v_3\}] &= \min(W[4][j] + D[v_j][\{v_2, v_3\} - \{v_j\}]) \quad j : v_j \in \{v_2, v_3\} \\
 &= \min(W[4][2] + D[v_2][\{v_3\}] , W[4][3] + D[v_3][\{v_2\}]) \\
 &= \min(3 + \infty, \infty+8) = \infty
 \end{aligned}$$

et de manière similaire :

$$D[v3][\{v2, v4\}] = \min(7 + 10, 8 + 4) = 12$$

$$D[v2][\{v3, v4\}] = \min(6 + 14, 4 + \infty) = 20$$

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

Et la longueur de la tournée optimale :

$$D[v1][\{v2, v3, v4\}] = \min(W[1][j] + D[vj][\{v2, v3, v4\} - \{vj\}])$$

$$j : vj \in \{v2, v3, v4\}$$

$$= \min(W[1][2] + D[v2][\{v3, v4\}],$$

$$W[1][3] + D[v3][\{v2, v4\}],$$

$$W[1][4] + D[v4][\{v2, v3\}])$$

$$= \min(2 + 20, 9 + 12, \infty + \infty) = 21$$

III.8.1- PrD : Algorithme TSP

```

void travel (int n, int W [] [], int CF [] [], int & minlength)
{ int i, j, k; int D [1 .. n] [sous ensemble de V - {v1}];
  for (i = 2; i <= n; i++)      D [i] [∅] = W[i] [1];
  for (k = 1; k <= n - 2; k++)
    for (tout sous ensemble A ⊆ V - {v1} contenant k nœuds)
      for (i tel que i ≠ 1 & vi ∉ A)
        { D [i] [A] = minimum (W [i] [j] + D [j] [A - {vj}]);           // j : vj ∈ A
          CF[i] [A] = valeur de j qui a donné le minimum;
        }
  D [1] [V - {v1}] = minimum (W[1] [j] + D[j] [V - {v1, vj}]);    2 <= j <= n
  CF[1] [V - {v1}] = valeur de j qui a donné le minimum;
  minlength = D[1] [V - {v1}];
}

```

La matrice CF (Coming-From) donnera la tournée optimale.

La complexité de cet algorithme est $\Theta(n^2 \cdot 2^n)$ /..

A titre d'exemple : pour $n=20$ (nombre de nœuds)

- le calcul avec un algorithme brut-force et une microseconde par opération de base prendra $19!$ microsecondes = **3857 années**
- le calcul avec cet algorithme : $20^2 \cdot 2^{20} = 21$ millions d'accès aux matrices.
- avec $n=60$, même l'algorithme ci-dessus prendra **plusieurs années !**

Le calcul du trajet (CF) pour le graphe :

3

CF[1], {v2, v3, v4}]

4

CF[3], {v2, v4}]

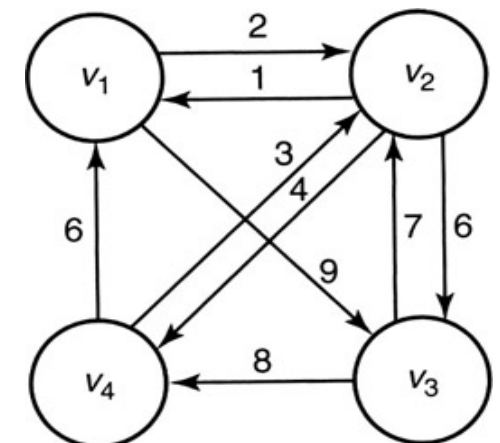
2

CF[4], {v2}]

La tournée optimale est calculée par :

- indexe du premier nœud : $CF[1][\{v2, v3, v4\}] = 3$
- on utilise 3 : $CF[3][\{v2, v4\}] = 4$
- on utilise 4 : $CF[3][\{v2\}] = 2$

Et la tournée optimale : {v1, v3, v4, v2, v1}



III.9- PrD et le problème de Sac à Dos

- Des objets $1, 2, \dots, n$ de poids w_1, w_2, \dots, w_n
- Chacun rapporte un bénéfice b_1, b_2, \dots, b_n
- Trouver le bénéfice maximum réalisable sachant que la charge maximale est P_0

Idée pour P_0 entier pas trop grand :

- On note $B(k, p)$ le bénéfice maximal réalisable avec des objets $1, 2, \dots, k$ et le poids maximal p
- On a pour $k = 1$:

$$\begin{aligned} B(1, p) &= 0 \quad \text{si} \quad p < w_1 && // \text{ si le poids max est inférieur au poids de l'objet (on ne prend pas !)} \\ &= b_1 \quad \text{si} \quad p \geq w_1 \end{aligned}$$

- Pour $k > 1$

$$\begin{aligned} B(k, p) &= B(k - 1, p) \quad \text{si} \quad p < w_k && // \text{ idem mais pour le } k^{\text{ème}} \text{ objet} \\ &= \max[B(k - 1, p), B(k - 1, p - w_k) + b_k] \quad \text{si} \quad p \geq w_k \end{aligned}$$

N.B. : à comparer avec les versions BT et BT-optimisée du problème de Sac à dos.

Exemple (poids total $P_0 = 12$) :

matrice des profits (on fait varier $k:1..8$ et $P_0: 0..12$) :

Objets	1	2	3	4	5	6	7	8
Poids	2	3	5	2	4	6	3	1
Bénéfices	5	8	14	6	13	17	10	4

$B(k, p)$	0	1	2	3	4	5	6	7	8	9	10	11	12
$k = 1$	0	0	5	5	5	5	5	5	5	5	5	5	5

$B(k, p)$	0	1	2	3	4	5	6	7	8	9	10	11	12
$k = 1$	0	0	5	5	5	5	5	5	5	5	5	5	5
$k = 2$	0	0	5	8	8	13	13	13	13	13	13	13	13

$B(k, p)$	0	1	2	3	4	5	6	7	8	9	10	11	12
$k = 1$	0	0	5	5	5	5	5	5	5	5	5	5	5
$k = 2$	0	0	5	8	8	13	13	13	13	13	13	13	13
$k = 3$	0	0	5	8	8	14	14	19	22	22	27	27	27

Suite ... (il y a 8 objets, $k=1..8$)

$B(k, p)$	0	1	2	3	4	5	6	7	8	9	10	11	12
$k = 1$	0	0	5	5	5	5	5	5	5	5	5	5	5
$k = 2$	0	0	5	8	8	13	13	13	13	13	13	13	13
$k = 3$	0	0	5	8	8	14	14	19	22	22	27	27	27
$k = 4$	0	0	6	8	11	14	14	20	22	25	28	28	33
$k = 5$	0	0	6	8	13	14	19	21	24	27	28	33	35
$k = 6$	0	0	6	8	13	14	19	21	24	27	30	33	36
$k = 7$	0	0	6	10	13	16	19	23	24	29	31	34	37
$k = 8$	0	4	6	10	14	17	20	23	27	29	33	35	38

IV- Annexes

IV.1- Algorithmes de recherche et optimisation

Best-first, first-fail (dans une certaine mesure, vu aussi dans TSP), A et A* sont des stratégies d'optimisation de la recherche.

Cas **adversatif** : Un autre exemple : Min-Max / Alpha-Beta

B & B est une méta stratégie de plus haut niveau (ne dépend pas d'adversaire).

L'exemple Morpion permet d'illustrer plusieurs stratégies.

IV.2- La stratégie Min-Max (voir BE)

Stratégie employée dans les jeux entre deux adversaires.

Cas de morpion où on place des 'X' et des 'O'.

1/ Au départ, on développe l'arbre complet (Figure suivante)

2/ On trouve la meilleure case où placer la 1er 'X' en appliquant la fonction *Evaluation_MinMax*(S: état) où S est l'état initial.

3/ En fonction du jeu de l'adversaire qui place son 1er 'O', une bonne partie de l'arbre (7/9) devient inutile (pour cette étape).

On évalue le meilleur coup à jouer par la même fonction que en étape (2)

4/

N.B. : En toute rigueur, on peut développer l'arbre à partir d'un état donné (pas seulement le départ).

Une manière pratique de construire l'arbre Min-Max est de partir des états terminaux (gain, perte ou égalité), de les évaluer par une fonction d'évaluation (+1, -1 et 0 est un exemple) et de remonter ces valeurs niveau par niveau en fonction du type (*Min* ou *Max*) du niveau, jusqu'à la racine.

La Figure ci-dessous montre un exemple partiel de l'arbre Min-Max pour le jeu morpion.

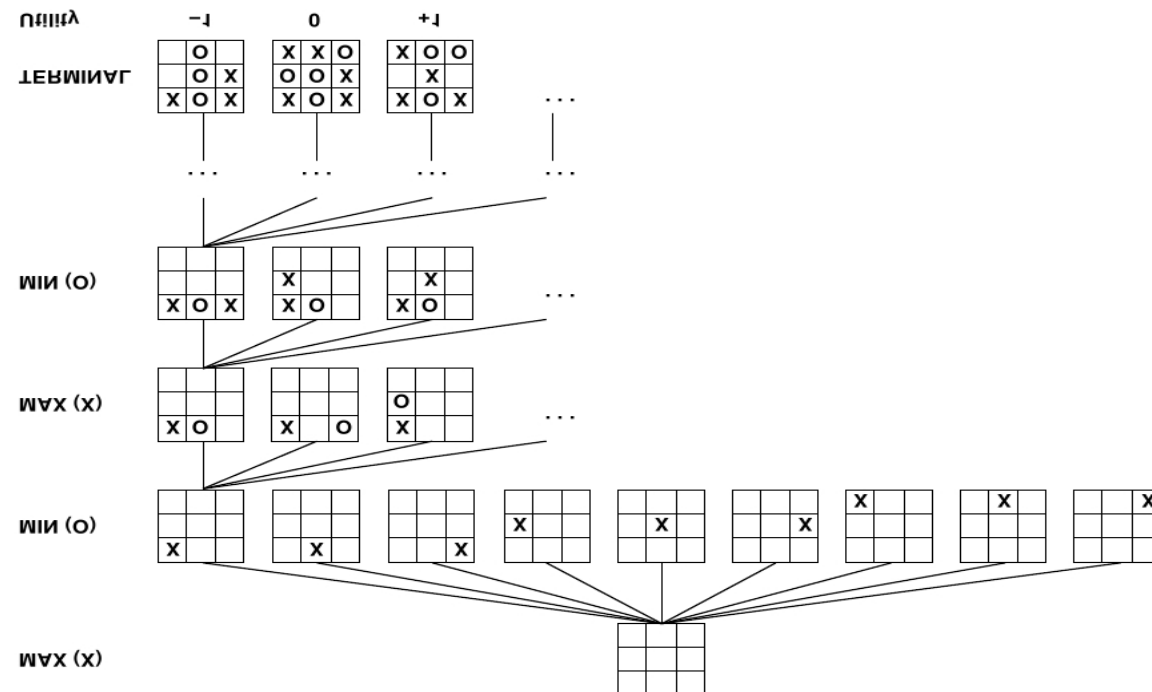


Figure : le nœud racine au sommet est l'état initial et MAX (le **X**) joue le 1er. L'arbre donne les différents choix possibles pour MIN (le **O**) et pour MAX jusqu'à ce que l'un des états terminaux soit atteint.

IV.3- Multiplication de nombres binaires

La stratégie Diviser-et-Régner peut se mettre en place par un schéma récurrent (pour la division, en particulier Dichotomique).

Pour multiplier deux nombres binaires positifs, on peut procéder par cette stratégie (Carl Friedrich Gauss, 1777-1855).

Exemple : soit $X=10110010$ et $Y=0110001$.

Pour les multiplier, on procède par :

XL = la moitié gauche de X = 1011

XR = la moitié droite de X = 0010

YL = la moitié gauche de Y = 0110

YR = la moitié droite de Y = 0001

Pour la taille n = maximum de celles de X et de Y ,
on procède ensuite par 3 multiplications $Mul1$, $Mul2$, $Mul3$ et une addition (Add) :

→ $Mul1 : XL * YL$ ici $1011 * 0110$

→ $Mul2 : XR * YR$ ici $0010 * 0001$

→ $Mul3 : (XL + XR) * (YL + YR)$ ici $1101 * 1001$

→ $Add : Mul1 * 2^n + (Mul3 - Mul1 - Mul2) * 2^{n/2} + Mul2.$

L'algorithme de cette multiplication :

Fonction multiplier(X, Y?: entiers positifs) : renvoie le produit X.Y

Début

$n = \max(\text{taille de } X, \text{taille de } Y)$

Si $(n = 1)$ renvoyer $X*Y$ // cas basique

$x_L, x_R = \text{leftmost } n/2 \text{ bits de } X, \text{rightmost } n/2 \text{ bits de } X$

$y_L, y_R = \text{leftmost } n/2 \text{ bits de } Y, \text{rightmost } n/2 \text{ bits de } Y$

$P1 = \text{multiplier}(x_L, y_L)$

$P2 = \text{multiplier}(x_R, y_R)$

$P3 = \text{multiplier}(x_L + x_R, y_L + y_R)$

renvoyer $P1 \times 2^n + (P3 - P1 - P2) \times 2^{n/2} + P2$

Fin multiplier

Complexité d'un schéma récurrent (stratégie diviser-régner) :

$$T(n) = a T(\lceil n/b \rceil) + O(n^d) \quad \text{où}$$

n : la taille du problème à chaque étape (ici 8 puis 4,2,1),

a : le cout lors de la division (ici $a=3$ car 3 appels récurifs),

b : facteur de division en sous-problèmes (ici $b=2$),

d : le cout de l'assemblage (ici $d=1$).

Dans le cas de la multiplication binaire, $T(n)=3 T(\lceil n/2 \rceil) + O(n)$

⇒ le calcul :

IV.3.1- Calcul de la complexité

On a $T(n) = 3 T(n/2) + n$ Avec $T(0) = 0$ et $T(1) = 1$

Posons $N = 2^k \rightarrow T(2^k) = 3 T(2^{k-1}) + 2^k$

On pose $t_k = T(2^k) : \rightarrow t_k = 3 t_{k-1} + 2^k$

D'où : (1) $t_k - 3 t_{k-1} - 2^k = 0$

Aussi (2) $t_{k-1} - 3 t_{k-2} - 2^{k-1} = 0$

On divise (1) par la constante 2 puis on soustrait les deux pour faire disparaître les constantes :

$$(1) t_k/2 - 3/2 t_{k-1} - 2^{k-1} = 0$$

$$(2) t_{k-1} - 3 t_{k-2} - 2^{k-1} = 0$$

La soustraction donne l'équation de récurrence :

$$(3) 1/2 t_k - 5/2 t_{k-1} + 3 t_{k-2} = 0 \rightarrow t_k - 5 t_{k-1} + 6 t_{k-2} = 0$$

On pose $t_k = r^k$ et on obtient l'équation caractéristique :

$$r^k - 5 r^{k-1} + 6 r^{k-2} = 0$$

$$\text{D'où } r^{k-2} (r^2 - 5 r + 6) = 0 \rightarrow r^{k-2}(r - 2)(r-3) = 0$$

Les racines seront : $r = 0, 2, 3 \rightarrow t_k = c_1 2^k + c_2 3^k$.

L'exploitation des cas d'arrêt donne $c_1 = -1$ et $c_2 = 1$

$$\text{d'où } \rightarrow t_k = 3^k - 2^k$$

Sachant que $N = 2^k$, on a $k = \log_2 N$

$$\rightarrow T(N) = 3^{\log_2 N} - N = N^{\log_2 3} - N = O(N^{\log_2 3}) = O(N^{1.6})$$

N.B. : pour $N=1$, $k=0$ et $T(N) = O(N)$ car $N = 1$ (cas d'arrêt).

A l'autre extrémité, la complexité sera $T(N) = O(N^{\log_2 3})$.

Nota Bene :

Si N varie, ces termes représentent une suite géométrique d'un facteur (raison) de $\log 3$ (≈ 1.6).

La somme des termes d'une telle suite géométrique (N tend vers l'infini) est $N^{1+\log 3}$.

IV.4- TDA Graphe

A titre indicatif.

Sorte Graphe, It % It vaut Itérateur

Utilise : Bool, Élément, Liste

graphe_Vide :	—> Graphe	
est_vide: Graphe	—> Bool	
adjacents: Graphe x It	—> Liste	// Liste d'élément
adjacents: Graphe x Élément	—> Liste	
recherche: Graphe x Élément	—> It	
existe: Graphe x Élément	—> bool	
insère: Graphe x Élément	—> Graphe	
premier: Graphe	-/-> Élément	
premier: Graphe	—> It	
suivant : Graphe x Élément	-/-> Élément	
suivant : Graphe x Élément	—> It	
noeud_courant: Graphe	—> It	
noeud_courant: Graphe	-/-> Élément	
noeud_suivant: Graphe x Élément	—> Élément	
noeud_suivant: Graphe x Élément	—> It	

Les opérateurs sur les Itérateurs (cf. les itérateurs)

Début: Liste \rightarrow It

Next: It \rightarrow It

Pred: It \rightarrow It

Déref: It \rightarrow Élément

Valide: It \rightarrow bool

Pré-conditions et Axiomes :

.....

V- Table des matières

I- Méta-Stratégies générales de résolution	2
I.1- Les techniques qui <i>regardent en arrière</i>	3
I.2- Les techniques qui <i>regardent en avant</i>	4
I.3- Illustration : exemple N-reines	6
II- Aspects pratiques et pratiques du Back-tracking	13
II.1- Exemple-1 : placements sous conditions (N-reines)	14
II.1.1- Complexité de N reines (en nombre d'états visités)	19
II.2- Exemple-2 : le pb. de la somme des sous ensembles	23
III- Approches de conception d'algorithmes : stratégies	31
III.1- Stratégies Diviser et Régner	33
III.1.1- Quelques exemples	34
III.2- PrD : Principe	35
III.2.1- Quelques exemples PrD	39
III.3- Exemple 1 - calcul binomial	40
III.4- Exemple 2 : multiplication de matrices	44
III.4.1- Conception de la PrD	46
III.5- Résumé des étapes de définition d'un schéma PrD	49
III.6- Exemple 3 : Distance de Levenshtein	51
III.6.1- Calcul du cout de Levenshtein	53
III.6.2- Code du cout de Levenshtein	54
III.6.3- La solution PrD de Levenshtein	56
III.7- Exemple 4 : algorithme de Floyd	61
III.7.1- Algorithme de Floyd	68
III.7.2- Obtention des trajets (coming from)	69
III.7.3- Programmation Dynamique et optimisation	71
III.7.4- Variant de Floyd : algorithme de base de Roy-Warshall	74
III.8- Exemple 5 : le voyageur de commerce (TSP)	77
III.8.1- PrD : Algorithme TSP	83

III.9- PrD et le problème de Sac à Dos	85
IV- Annexes	89
IV.1- Algorithmes de recherche et optimisation	89
IV.2- La stratégie Min-Max (voir BE)	90
IV.3- Multiplication de nombres binaires	92
IV.3.1- Calcul de la complexité	96
IV.4- TDA Graphe	99
V- Table des matières	101