COURS : PROGRAMMATION DYNAMIQUE
= DISTANCE DE LEVENSHTEIN =

Notre cinquième étude de cas concerne le problème du calcul de la distance d'édition entre deux chaînes de caractères (aussi appelée distance de Levenshtein). Nous allons construire la solution du problème par programmation dynamique.

I) DÉFINITION DU PROBLÈME

Une instance du problème est spécifiée par deux chaînes de caractères : une chaîne source S_1 de longueur n et une chaîne cible S_2 de longueur m. La tâche de l'algorithme est de trouver le nombre minimal d'opérations élémentaires nécessaires pour transformer S_1 en S_2.

On autorise trois opérations, chacune de coût 1 (garder un caractère à un coût nul) :

- Insertion d'un caractère dans S_1 (pour se rapprocher de S_2) :
  - Ex : si S_1 = "CHAT" et S_2 = "CHATS", on peut insérer 'S' dans S_1.
- Suppression d'un caractère de S_1 :
  - Ex : si S_1 = "CHATS" et S_2 = "CHAT", on peut supprimer 'S' de S_1.
- Substitution d'un caractère de S_1 par un autre :
  - Ex: si S_1 = "CHAT" et S_2 = "CHOT", on peut remplacer 'A' par 'O'.

La distance d'édition d(S_1, S_2) est le coût minimal (donc le nombre minimal d'opérations) pour transformer S_1 en S_2.

Problème de la distance d'édition (Levenshtein)

Entrée : Deux chaînes de caractères S_1 et S_2.

Sortie : Un entier d(S_1, S_2), égal au nombre minimal d'insertions, suppressions et de substitutions transformant S_1 en S_2.

Exemple : Soit le calcul de la distance entre S_1 = "CHIEN" (n=5) et S_2 = "NICHE" (m=5).
Une solution optimale (distance 4) serait :

- Insertion de 'N' $\rightarrow$ S_1 = "**N**CHIEN" (coût +1)
- Insertion de 'I' $\rightarrow$ S_1 = "**NI**CHIEN" (coût +1)
- Garder 'C' puis grader 'H' $\rightarrow$ S_1 = "**NICH**IEN" (coût 0)
- Suppression de 'I' $\rightarrow$ S_1 = "**NICHE**N" (coût +1)
- Suppression de 'N' $\rightarrow$ S_1 = "**NICHE**" (coût +1)

La distance d'édition a de nombreuses applications pratiques :

- Correction orthographique : suggérer des mots proches d'un mot mal orthographié ;
- Bio-informatique : comparer des séquences ADN ou protéiques pour détecter des mutations ;
- Détection de plagiat : mesurer la similarité entre deux textes ;
- Reconnaissance vocale : corriger les erreurs de transcription.

Pour résoudre ce problème de manière exhaustive (par brute force), il faudrait tout d'abord lister toutes les séquences possibles d'opérations, puis pour chacune, vérifier si elle transforme bien S_1 en S_2, et enfin garder la plus courte.

La pire façon de transformer S_1 en S_2 est de supprimer tous les caractères de S_1 (n opérations) et d'insérer tous les caractères de S_2 dans S_1 (m opérations). Le nombre maximum d'étapes d'une séquence de transformation est donc bornée par (n + m). À chaque étape, on peut effectuer jusqu'à 3 types d'opérations. Dans le pire des cas, le nombre de chemins à explorer est de de $O(3^{m+n})$. C'est un problème exponentiel qui demande une approche plus efficace.

II) SOUS-STRUCTURE OPTIMALE ET RELATION DE RÉCURRENCE

Pour appliquer la programmation dynamique, on doit identifier des sous-problèmes pertinents et une relation de récurrence entre eux, issue de la structure d'une solution optimale.

II.1. Sous-structure optimale

Considérons une instance du problème avec une chaîne source S_1 et une chaîne cible S_2. Regardons la dernière opération effectuée permettant d'obtenir une transformation optimale de $S_1[1..n]$ en $S_2[1..m]$.

Trois cas sont possibles pour traiter les derniers caractères des préfixes $S_1[1..n]$ et $S_2[1..m]$ lors de la dernière étape :

Cas n°1 : Suppression du dernier caractère de S_1

Lors de la dernière étape, on a choisi de supprimer le caractère $S_1[n]$. Rechercher le coût minimal pour transformer $S_1[1..n]$ en $S_2[1..m]$ revient dans ce cas à rechercher le coût minimal pour transformer $S_1[1..n-1]$ en $S_2[1..m]$, en y ajoutant le coût d'une opération de suppression (+1) :

$$coût\ (S_1[1..n], S_2[1..m]) = coût\ (S_1[1..n-1], S_2[1..m]) + 1$$

Exemple : S_1 = "abc" et S_2 = "ab"

- Solution optimale pour transformer $S_1[1..3]$ en $S_2[1..2]$: supprimer 'c'.
- Coût final : coût du sous-problème {$S_1[1..2]$ = "ab", $S_2[1..2]$= "ab"} + coût d'une suppression (+1).

Cas n°2 : Substitution ou alignement (match) des derniers caractères de S_1 et S_2.

Lors de la dernière étape, on a choisi soit de substituer le caractère $S_1[n]$ par $S_2[m]$ (c'est-à-dire de remplacer $S_1[n]$ par $S_2[m]$), soit d'aligner $S_1[n]$ avec $S_2[m]$ (c'est-à-dire de garder $S_1[n]$ tel quel car $S_1[n] = S_2[m]$).

Rechercher le coût optimal de la transformation finale revient dans ces cas à rechercher le coût minimal pour transformer $S_1[1..n-1]$ en $S_2[1..m-1]$, en y ajoutant éventuellement le coût de la substitution (+1) si $S_1[n] \neq S_2[m]$. Dans le cas où $S_1[n] = S_2[m]$, on n'ajoute pas de coût supplémentaire :

$$coût\ (S_1[1..n], S_2[1..m]) = coût\ (S_1[1..n-1], S_2[1..m-1]) + \begin{cases} 0, & si\ S_1[n] == S_2[m] \\ 1, & sinon \end{cases}$$

Exemple : S_1 = "abc" et S_2 = "abd"

- Solution optimale pour transformer $S_1[1..3]$ en $S_2[1..3]$: substituer 'c' par 'd'.
- Coût final : coût du sous-problème {$S_1[1..2]$ = "ab", $S_2[1..2]$= "ab"} + coût d'une substitution (+1).

Exemple : S_1 = "abc" et S_2 = "abc"

- Solution optimale pour transformer $S_1[1..3]$ en $S_2[1..3]$: ne rien faire (match).
- Coût final : coût du sous-problème {$S_1[1..2]$ = "ab", $S_2[1..2]$= "ab"}.

Cas n°3 : Insertion du dernier caractère de S_2.

Lors de la dernière étape, on a choisi d'insérer le caractère $S_2[m]$ afin d'obtenir le coût minimal permettant de transformer $S_1[1..n]$ en $S_2[1..m]$.
Rechercher le coût optimal de la transformation finale revient dans ce cas à rechercher le coût minimal pour transformer $S_1[1..n]$ en $S_2[1..m-1]$, en y ajoutant le coût d'une opération d'insertion (+1) :

$$coût\ (S_1[1..n], S_2[1..m]) = coût\ (S_1[1..n], S_2[1..m-1]) + 1$$

Exemple : S_1 = "ab" et S_2 = "abc"

- Solution optimale pour transformer $S_1[1..2]$ en $S_2[1..3]$: insérer 'c'.
- Coût final : coût du sous-problème {$S_1[1..2]$ = "ab", $S_2[1..2]$= "ab"} + coût d'une insertion (+1).

II.2. Équation de récurrence sur les valeurs optimales

On note $D_{i,j}$ la distance d'édition minimale entre les i premiers caractères de S_1 et les j premiers caractères de S_2.

Récurrence sur la valeur de la solution optimale

Pour tout i ∈ {0..n} et j ∈ {0..m}, les cas de base sont :

- $D_{0,j}$ = j (transformer la chaîne vide en j caractères demande j insertions),
- $D_{i,0}$ = i (transformer i caractères en chaîne vide demande i suppressions).

Pour tout i ∈ {1..n} et j ∈ {1..m} :

$$D_{i,j} = min \begin{cases} D_{i-1,j} + 1 & (cas\ n°1 :\ suppression) \\ D_{i,j-1} + 1 & (cas\ n°3 :\ insertion) \\ D_{i-1,j-1} + \begin{cases} 0, & si\ S_1[i] == S_2[j] \\ 1, & sinon \end{cases} & (cas\ n°2 :\ substitution\ /\ match) \end{cases}$$

III) SOUS-PROBLÈMES ET COMPLEXITÉ

III.1. Définition des sous-problèmes

Ici, les sous-problèmes sont indexés par les paramètres i (longueur du préfixe de S_1, de 0 à n) et j (longueur du préfixe de S_2, de 0 à m). Quand i = 0 ou j = 0, $S_1[1..0]$ et $S_2[1..0]$ sont des chaînes vides.

En faisant varier ces deux paramètres sur toutes les valeurs pertinentes, nous obtenons nos sous-problèmes :

Sous-problèmes de la distance de Levenshtein

Calculer $D_{i,j}$, la distance minimale pour transformer le préfixe de longueur i de S_1 en le préfixe de longueur j de S_2.

(Pour chaque i = 0, 1, 2 … n et j = 0, 1, 2 … m)

Le plus grand sous-problème (avec i=n et j=m) est exactement le problème original.

III.2. Schéma de récursion

Le schéma de récursion sur un l'exemple où on cherche à aligner S_1 = "tu" avec S_2 = "toi" est donné sur la figure ci-dessous :

- La notation ["tu","toi"] signifie qu'on cherche la solution optimale au problème qui consiste à aligner S_1 = "tu" avec S_2 = "toi" ;
- La notation D[2][3]=2 signifie que le coût optimal pour aligner les deux premiers caractères de S_1 avec les 3 premiers caractères de S_2 vaut 2 ;
- Dans le cas n°1 (branches de gauche), à partir des cas de base (en vert), on remonte la valeur D[i-1][j] + 1 (coût d'une suppression) ;
- Dans le cas n°2 (branches du milieu), à partir des cas de base (en vert), on remonte la valeur D[i-1][j-1] +1 (si il y a substitution) ou D[i-1][j-1] (si il y a match) ;
- Dans le cas n°3 (branches de droite), à partir des cas de base (en vert), on remonte la valeur D[i][j-1] + 1 (coût d'une insertion).
- Les valeurs D[i][j] prennent le minimum des valeurs remontées.

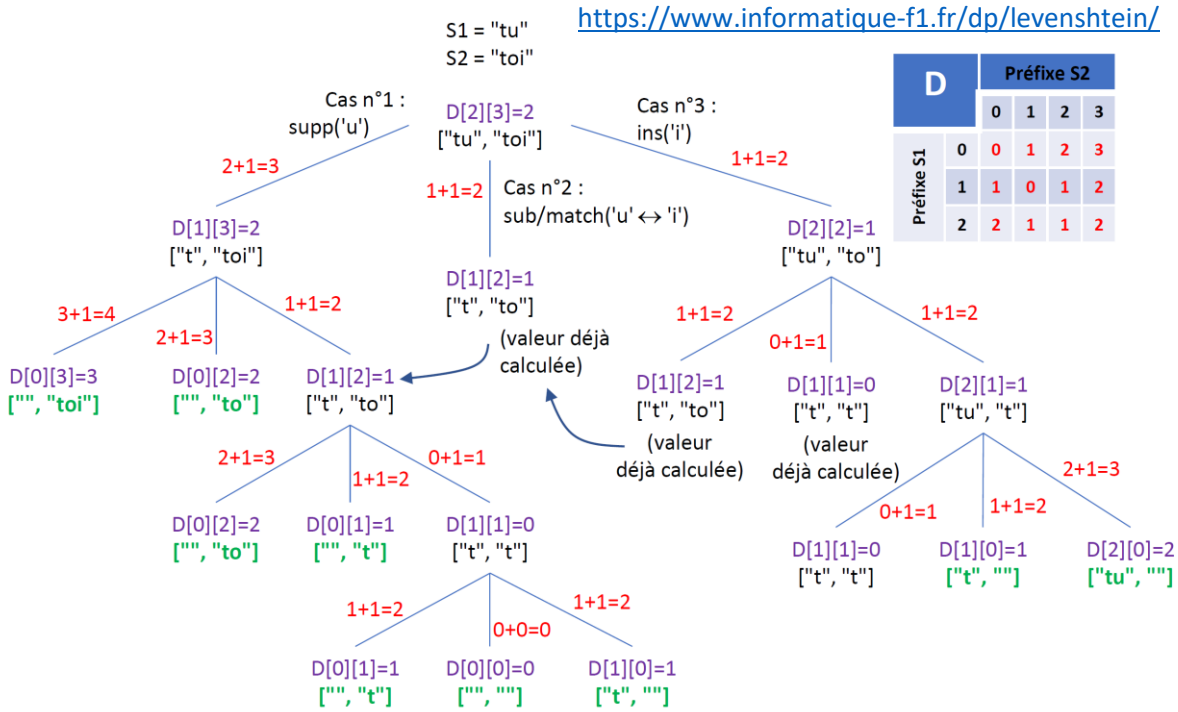

Figure 1 : Schéma de récursion du problème de la distance de Levenshtein

III.3. Complexité sans mémoïsation

À chaque étape, on diminue i ou j (ou les deux). La profondeur de récursion est donc au plus de (n + m). Dans le pire des cas, chaque nœud se ramifie en 3, ce qui donne un arbre de récursion pouvant contenir jusqu'à $O(3^{n+m})$ nœuds.

Le travail local à chaque nœud est en O(1) (comparaisons, additions, calcul d'un minimum), donc l'algorithme récursif sans mémoïsation est exponentiel : $O(3^{n+m})$.

Remarquons cependant qu'il n'y a que (m+1)·(n+1) sous-problèmes distincts. De nombreux appels récursifs portent donc sur les mêmes sous-problèmes, d'où l'intérêt de la mémoïsation.

IV) ALGORITHMES DE PROGRAMMATION DYNAMIQUE

IV.1. Algorithme top-down

On mémorise les valeurs déjà calculées $D_{i,j}$ dans un dictionnaire, afin de ne jamais recalculer deux fois le même sous-problème.

Algorithme top-down (1) pour le calcul des valeurs optimales

Entrée : S1[1, …, n] : Chaîne source
S2[1, …, m] : Chaîne cible

Sortie : distance de Levenshtein d(S1, S2)

```
# Dictionnaire de mémoïsation
D := {}

rec_opt_val_Levenshtein (i, j) :
    # i : longueur du préfixe de S1
    # j : longueur du préfixe de S2

    # Utilise la mémoïsation
    Si (i, j) est dans D :
    |   Retourner D[(i, j)]

    # Cas de base i ==0 et j == 0
    Si i == 0 :
    |   D[(i, j)] := j
    |   Retourner D[(i, j)]
    Si j == 0 :
    |   D[(i, j)] := i
    |   Retourner D[(i, j)]

    # Coût match/substitution
    Si S1[i] == S2[j] :
    |   c := 0
    Sinon
    |   c := 1

    # Trois possibilités (suppression, substitution/match, insertion)
    V1 := rec_opt_val_Levenshtein (i - 1, j) + 1
    V2 := rec_opt_val_Levenshtein (i - 1, j - 1) + c
    V3 := rec_opt_val_Levenshtein (i, j - 1) + 1

    D[(i, j)] := min (V1, V2, V3)

    Retourner D[(i, j)]

# Appel initial
résultat := rec_opt_val_Levenshtein (n, m)
```

On remarque que cet algorithme calcule toujours les valeurs des trois cas, même lorsque $S_1[i] == S_2[j]$. La table va donc être entièrement remplie.

On peut imaginer alors un algorithme top-down qui ne calcule ni le cas n°1 (suppression) ni le cas n°3 (insertion) s'il y a un match :

Algorithme top-down (2) optimisé pour le calcul des valeurs optimales

```
rec_opt_val_Levenshtein (i, j) :
    Si (i, j) est dans D :
    |   Retourner D[(i, j)]

    Si i == 0 :
    |   D[(i, j)] := j
    |   Retourner D[(i, j)]
    Si j == 0 :
    |   D[(i, j)] := i
    |   Retourner D[(i, j)]

    # Match / Substitution
    Si S1[i] == S2[j] :
    |   D[(i, j)] := rec_opt_val_Levenshtein (i - 1, j - 1)
    Sinon
    |   # Trois possibilités (suppression, substitution/match, insertion)
    |   V1 := rec_opt_val_Levenshtein (i - 1, j) + 1
    |   V2 := rec_opt_val_Levenshtein (i - 1, j - 1) + 1
    |   V3 := rec_opt_val_Levenshtein (i, j - 1) + 1
    |   D[(i, j)] := min (V1, V2, V3)

    Retourner D[(i, j)]
```

IV.2. Complexité de l'algorithme top-down

Les états possibles sont les couples (i, j) avec $0 \le i \le n$ et $0 \le j \le m$, soit au plus $(n+1)\cdot(m+1)$, c'est-à-dire $O(n\cdot m)$ sous-problèmes.

Avec mémoïsation, chaque (i, j) est calculé au plus une fois et le calcul effectue un travail local en O(1) (l'accès aux caractères des chaînes et en O(1) et aux valeurs stockées dans le dictionnaire également) et fait au plus 3 appels (vers des états plus petits). La complexité en temps est donc de $O(n\cdot m)$.

L'espace nécessaire pour sauvegarder les informations dans le dictionnaire est $O(n\cdot m)$. À chaque étape, on diminue i ou j (ou les deux). La profondeur de récursion est donc au plus de (n + m) et donc la profondeur de la pile est en O(n + m). Le total de l'espace mémoire est donc dominé par le dictionnaire et est de $O(n\cdot m)$.

IV.3. Algorithme bottom-up

L'algorithme bottom-up consiste à remplir progressivement la table des solutions des sous-problèmes en utilisant la relation de récurrence, en partant des cas de base.

Algorithme bottom-up pour le calcul des valeurs optimales

```
Entrée : S1[1, …, n] : Chaîne source
         S2[1, …, m] : Chaîne cible
Sortie : distance de Levenshtein d(S1, S2)

# Dictionnaire de mémoïsation
D := {}

opt_val_Levenshtein (S1, S2) :
    # i : longueur du préfixe de S1
    # j : longueur du préfixe de S2

    # Cas de base
    Pour j allant de 0 à m :
    |   D[(0, j)] := j
    Pour i allant de 0 à n :
    |   D[(i, 0)] := i

    # Remplissage de la table
    Pour i allant de 1 à n :
    |   Pour j allant de 1 à m :
    |   |   Si S1[i] == S2[j] :
    |   |   |   c := 0
    |   |   Sinon :
    |   |   |   c := 1
    D[(i, j)] := min(D[(i - 1, j)] + 1, D[(i - 1, j - 1)] + c, D[(i, j - 1)] + 1)
    Retourner D[(n, m)]
```

Voici deux exemples de tables remplies avec l'algorithme top-down (2) et bottom-up:

S1 = "pomme" ; S2 = "pompe"

Top-down

D		Préfixe S2					
		0	1	2	3	4	5
Préfixe S1	0	0	1	2	3		
	1	1	0	1	2	3	
	2	2	1	0	1	2	
	3	3	2	1	0	1	
	4				1	1	
	5						1

Bottom-up

D		Préfixe S2					
		0	1	2	3	4	5
Préfixe S1	0	0	1	2	3	4	5
	1	1	0	1	2	3	4
	2	2	1	0	1	2	3
	3	3	2	1	0	1	2
	4	4	3	2	1	1	2
	5	5	4	3	2	2	1

Figure 2 : Tables des valeurs optimales top-down (2) (gauche) et bottom-up (droite)

IV.4. Complexité de l'algorithme bottom-up

L'algorithme bottom-up calcule toutes les cases (i,j) de la table, pour $0 \le i \le n$ et $0 \le j \le m$. Il effectue donc exactement $(n+1)\cdot(m+1)$ calculs, chacun en O(1). La complexité en temps et en espace est donc de $O(n\cdot m)$.

V) ALGORITHME DE RECONSTRUCTION

V.1. Principe et algorithme de reconstruction

L'objectif est de reconstruire une suite d'opérations optimales à réaliser dans la chaîne source S1 permettant de transformer S1 en S2. On peut reconstruire cette suite d'opérations en retraçant le chemin depuis D[n][m] jusqu'à D[0][0].

On part du coin inférieur droit (n, m), l'algorithme regarde quel voisin a permis d'obtenir la valeur actuelle D[i][j]. Les opérations à effectuer s'ajoutent à une liste initialement vide.

Algorithme de reconstruction (compatible top-down optimisé de la page 7)

Entrée : S1[1, …, n] : Chaîne source
S2[1, …, m] : Chaîne cible
D = {(i, j) : … }: Dictionnaire / table des valeurs optimales
Sortie : Ops[…] : Liste des opérations à effectuer

```
Reconstruction (S1, S2, D) :
    Ops := []                # Liste de vide pour sauvegarder les opérations à effectuer
    i := n
    j := m

    Tant que i > 0 ou j > 0 :
        # Cas du match (prioritaire)
        Si i > 0 et j > 0 et D[(i, j)] == D[(i - 1, j - 1)]  et S1[i] == S2[j] :
            Ajouter « Garder S1[i] » à la liste Ops
            i := i - 1
            j := j - 1
        # Cas de la substitution
        Sinon si i > 0 et j > 0 et D[(i, j)] == D[(i - 1, j - 1)] + 1 :
            Ajouter « Substituer S1[i] par S2[j] » à la liste Ops
            i := i - 1
            j := j - 1
        # Cas de la suppression
        Sinon si i > 0 et D[(i, j)] == D[(i - 1, j)] + 1 :
            Ajouter « Supprimer S1[i] » à la liste Ops
            i := i - 1
        # Cas de l'insertion
        Sinon si j > 0 et D[(i, j)] == D[(i, j - 1)] + 1 :
            Ajouter « Insérer S2[j] » à la liste Ops
            j := j - 1

    Retourner Ops renversée
```

Remarque importante : Avec l'algorithme top-down (2) optimisé, certaines valeurs voisines peuvent ne pas exister dans le dictionnaire. En cas de match, seul D[i-1][j-1] a été calculé, tandis que D[i-1][j] et D[i][j-1] n'existent pas. Pour garantir la compatibilité avec les deux approches, l'algorithme de reconstruction doit tester les cas diagonaux (match et substitution) avant les cas de suppression et d'insertion.

On aurait également pu imaginer un algorithme de reconstruction qui vérifie si les clés D[i][j] sont disponibles dans le dictionnaire, et les calcule en appelant la fonction récursive en cas de besoin (à l'image de ce que nous avions fait pour le tableau partitionné d'entiers positifs).

La figure ci-dessous illustre ce principe de reconstruction avec S_1 = "tu" et S_2 = "toi" :

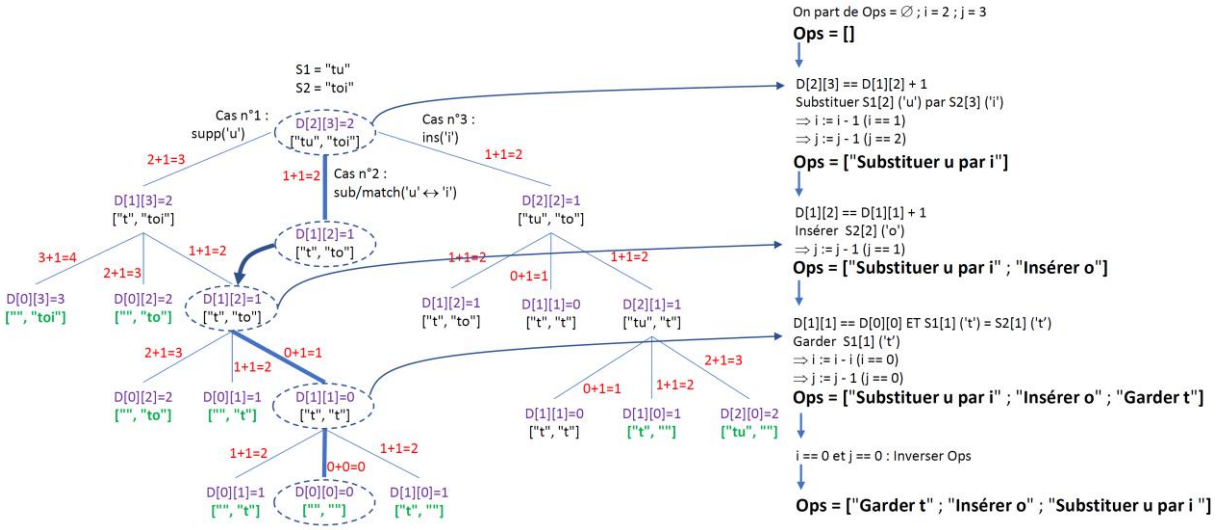

Figure 3 : Principe de reconstruction de la solution optimale

V.2. Complexité finale

La reconstruction parcourt au plus (n + m) étapes (à chaque étape, on diminue i ou j ou les deux), avec un travail local en O(1). La complexité en temps de la reconstruction est donc de O(n + m) et l'espace mémoire utilisé en O(n + m) (si on stocke la liste des opérations).

La complexité totale (calcul + reconstruction) est donc en O (n·m) + O(n + m) = O(n·m).

Si on ne veut que la distance (pas la reconstruction), on peut réduire l'espace mémoire de O(n·m) à O(min(n, m)) en ne gardant que la ligne (ou la colonne) précédente. En revanche, pour reconstruire une suite d'opérations, il faut en général conserver davantage d'informations (table complète).