ordre. D'abord les distances vers les cellules de la première ligne et première colonne sont faciles à calculer, car il existe un unique chemin vers ces cellules. Ensuite on calcule les distances vers les cellules (i, j) avec $1 \leq i \leq n, 1 \leq j \leq m$ dans l'ordre lexicographique. Ainsi la distance de $(0, 0)$ vers (i, j) est le minimum sur trois alternatives, qui est déterminé en temps constant, car les distances vers les prédécesseurs ont déjà été calculées.

Variantes Plusieurs problèmes classiques se réduisent à ce simple problème, comme décrit dans les sections suivantes.

3.2 Distance d'édition de Levenshtein

```
entrée:  AUDI, LADA
sortie:  LA-DA
         -AUDI
         3 opér. : suppression L, insertion U, substitution A en I
```

Définition Étant donné deux chaînes de caractères x, y, on veut savoir combien d'opérations (insertion, suppression ou substitution) sont requises pour transformer x en y. Cette distance est utilisée par exemple dans la commande unix *diff* qui affiche un nombre minimal d'opérations ligne par ligne entre deux fichiers donnés.

Algorithme en $O(nm)$ pour $n = |x|, m = |y|$ par programmation dynamique, voir figure 3.2. On va calculer un tableau $A[i, j]$ qui est la distance entre le préfixe de x de longueur i et le préfixe de y de longueur j. Pour commencer, on peut initialiser $A[0, j] = j$ et $A[i, 0] = i$. Puis en général quand $i, j \geqslant 1$, on a trois possibilités sur les dernières lettres des préfixes. Soit x_i est supprimé. Soit y_j est inséré (à la fin). Soit x_i est remplacé par y_j (s'il ne sont pas déjà égaux). Ceci donne la formule de récurrence suivante, où *match* est une fonction booléenne qui renvoie 1 si ses arguments sont identiques :

$$A[i, j] = \min \begin{cases} A[i-1, j-1] + \text{match}(x_i, y_j) \\ A[i, j-1] + 1 \\ A[i-1, j] + 1. \end{cases}$$

Cette fonction code le coût d'une substitution de lettre, qui peut ainsi être ajusté. Par exemple le coût pourrait dépendre de la distance des lettres sur le clavier.

Séquence des opérations Pour calculer en plus de la distance d'édition la suite des opérations pour transformer x en y, on peut procéder comme pour la recherche du plus court chemin dans une grille. En explorant les distances des prédécesseurs on détermine un choix qui réalise une distance minimale vers un sommet. Ainsi on peut remonter le chemin du sommet (n, m) à $(0, 0)$, et produire le long du chemin la liste des opérations d'une solution optimale. Pour finir, il suffit d'inverser cette liste.

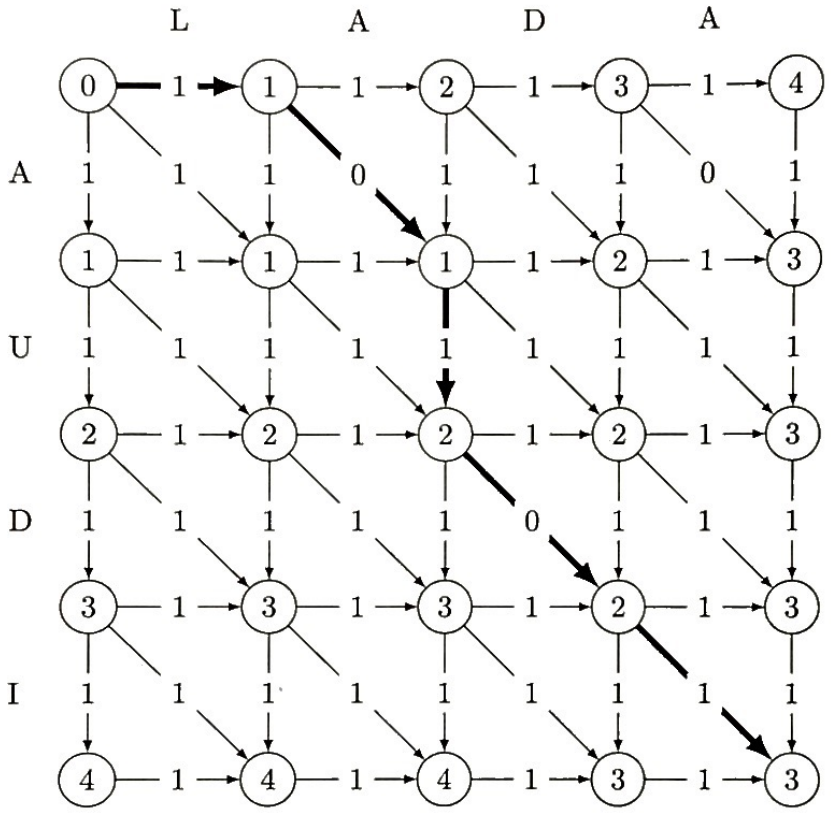

FIGURE 3.2 – Plus court chemin dans une grille orientée, déterminant la distance d'édition entre deux mots.

Détails d'implémentation Notez que dans la description du programme dynamique les indices du tableau commencent à 0, et ceux des chaînes à 1. À tenir compte dans une implémentation.

```
def levenshtein(x, y):
    n = len(x)
    m = len(y)
    #                          initialisation ligne 0 et colonne 0
    A = [[i + j for j in range(m + 1)] for i in range(n + 1)]
    for i in range(n):
        for j in range(m):
            A[i + 1][j + 1] = min(A[i][j + 1] + 1,              # insertion
                                  A[i + 1][j] + 1,              # suppress.
                                  A[i][j] + int(x[i] != y[j]))  # substitut.
    return A[n][m]
```

Pour aller plus loin Des algorithmes plus performants en pratique ont été proposés. Par exemple si une borne supérieure s est connue sur la distance d'édition, alors on peut restreindre le programme dynamique ci-dessus, aux entrées de A à distance au plus s de la diagonale, et obtenir une complexité $O(s \min\{n, m\})$ [28].