

# Report

October 13, 2017

```
In [1]: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

Nom: Grégory Bonaert

Section: INFO

Matricule: 000430665

## 1 Alignement de séquences

### 1.1 Introduction

L'objectif de ce projet est d'implémenter un algorithme d'alignement global (*Needleman-Wunsch*) et d'alignement local (*Smith-Waterman*) entre séquences. Ceci nous permettra d'évaluer la similarité entre séquences d'acides aminés et aussi de trouver les séquences les plus similaires.

Pour vérifier nos résultats, on va comparer notre implémentation avec le logiciel LALIGN, et essayer de comprendre nos résultats via la site Uniprot (qui fournit des informations sur les protéines liées à chacune des séquences).

## 2 L'alignement de séquences

Une approche pour comparer des séquences d'acides (et donc des protéines) est d'examiner leur similarité en essayant de voir quelles séquences "s'alignent" le mieux. En d'autres mots, est-il facile de faire correspondre les acides aminés entre les 2 séquences?

En quantifiant la difficulté d'alignement, on peut donc quantifier la similarité entre séquences d'acides aminés, et donc comparer un grande quantité de protéines entre elles de façon algorithmique.

## 3 Alignement global

L'alignement global cherche à trouver le meilleur alignement entre 2 séquences (d'acides aminés dans notre cas). Pour trouver l'alignement global optimal, nous allons implémenter l'algorithme de *Needleman-Wunsch* avec une pénalité affine.

### 3.1 L'algorithme de Needleman-Wunsch

Cet algorithme se base sur l'idée que trouver le meilleur alignement est un cas spécial du problème de la recherche de la distance d'édition entre les 2 séquences. On utilise donc une approche similaire pour résoudre les 2 problèmes.

#### 3.1.1 La distance d'édition

La distance d'édition est la séquence d'insertions, de substitutions et d'effacements nécessaires pour transformer un string A en un string B à un moindre coût.

Prenons par exemple les strings "ABCDAAA" et "BCEEAAA". Si chaque action (effacer, insérer, substituer) a un coût de 1, alors la meilleure distance d'édition est 3, car il faut:

1. Effacer le premier A
2. Remplacer le D par un E
3. Insérer un E avant le AAA

ABCD-AAA | | . | | | -BCEEAAA

Un espace en bas correspond à un effacement. Un espace en haut correspond à une insertion. Un point entre les lettres correspond à une substitution.

#### 3.1.2 Le lien avec l'alignement

Trouver le meilleur alignement correspond à trouver l'alignement qui minimise la distance d'édition. L'alignement est un cas spécial car des actions différentes peuvent avoir des coûts différents. Par exemple:

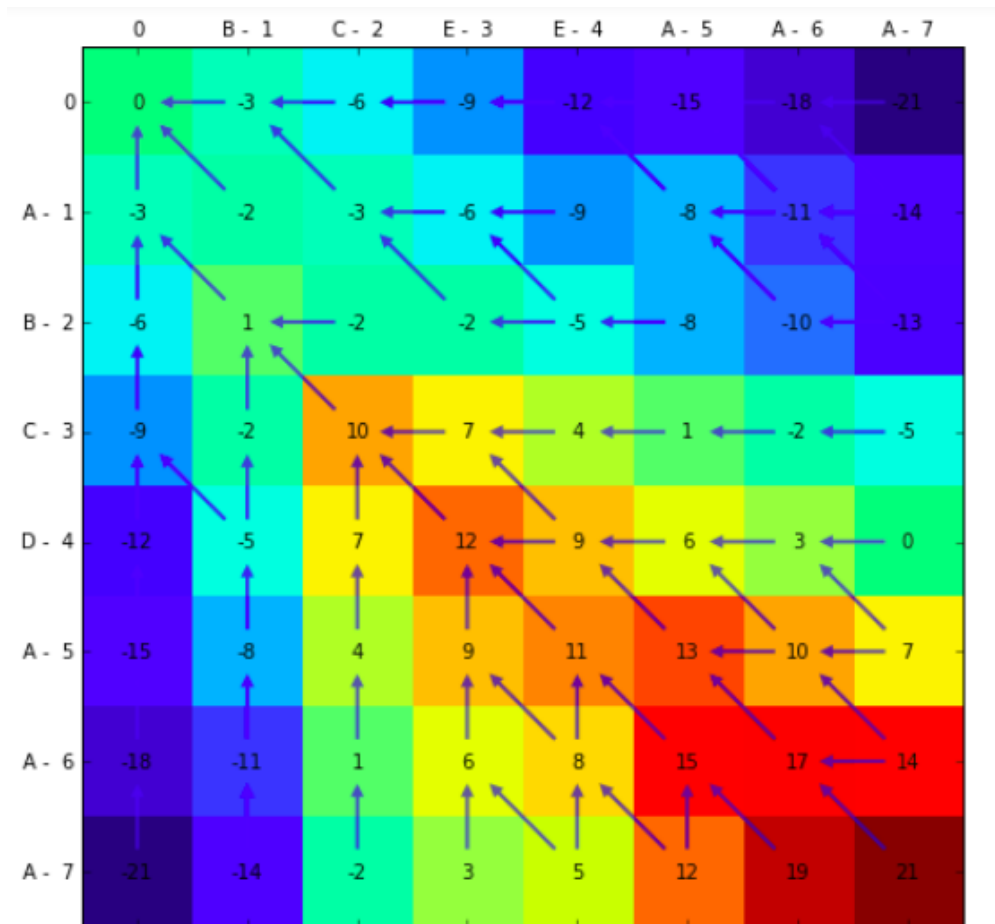
- Effacer ou insérer un acide aminé a un coût 5
- Remplacer l'acide aminé R par C a un coût de 3
- Remplacer l'acide aminé W par V a un coût de 7
- Et ainsi de suite

On peut donc s'inspirer du concept de distance d'édition pour concevoir notre solution.

### 3.2 Résumé de l'algorithme

L'algorithme de Needleman-Wunsch est assez simple:

1. On construit une matrice qui contient les distances d'édicions minimales entre des morceaux chaque fois plus grands des séquences, en mémorisant les actions faites pour aboutir à cette distance d'édition minimale
2. Une fois que la matrice est créée, on retrace les actions pour voir quel alignement minimise la distance d'édition et maximise la similarité
3. On crée l'alignement optimal à partir de cette suite d'actions



Matrice des scores voulue

### 3.2.1 Un exemple

Le premier pas est de construire une *matrice des scores*. Les scores représentent la distance d'édition entre des parties des deux séquences.

Soient les 2 séquences "ABCDAAA" et "BCEEAAA". Voici la matrice des score qu'on souhaite construire:

#### Comment interpréter cette matrice?

Examinons par exemple la case (2,2). Ce score correspond au meilleur moyen de faire correspondre "AB" à "BC". Les actions à faire sont de descendre, d'aller en diagonale puis d'aller à droite. Ceci correspond à

1. Effacer A de la première séquence (en d'autres mots, le faire correspondre à un espace)
2. Faire correspondre B à B (en d'autres mots, substituer B par B)
3. Effacer E de la deuxième séquence (en d'autres mots, le faire correspondre à un espace)

On aboutit donc à l'alignement suivant:

```
AB_  
|  
_BC
```

Comment interpréter les valeurs numériques des scores? Si le score est très bas, il faut beaucoup éditer, donc les séquences ne sont pas similaires. Si le score est élevé, le coût d'édition est bas et donc les séquences sont similaires.

**Par une approche similaire, on peut reconstruire le meilleur alignement en retraçant le meilleur chemin allant de la case du coin en bas à droite au coin en haut à gauche.**

### 3.2.2 Étape 1: calculer la matrice des scores

Pour calculer la matrice des scores, on utilise une approche de programmation dynamique. Ainsi, pour arriver à une certaine case X, on a soit:

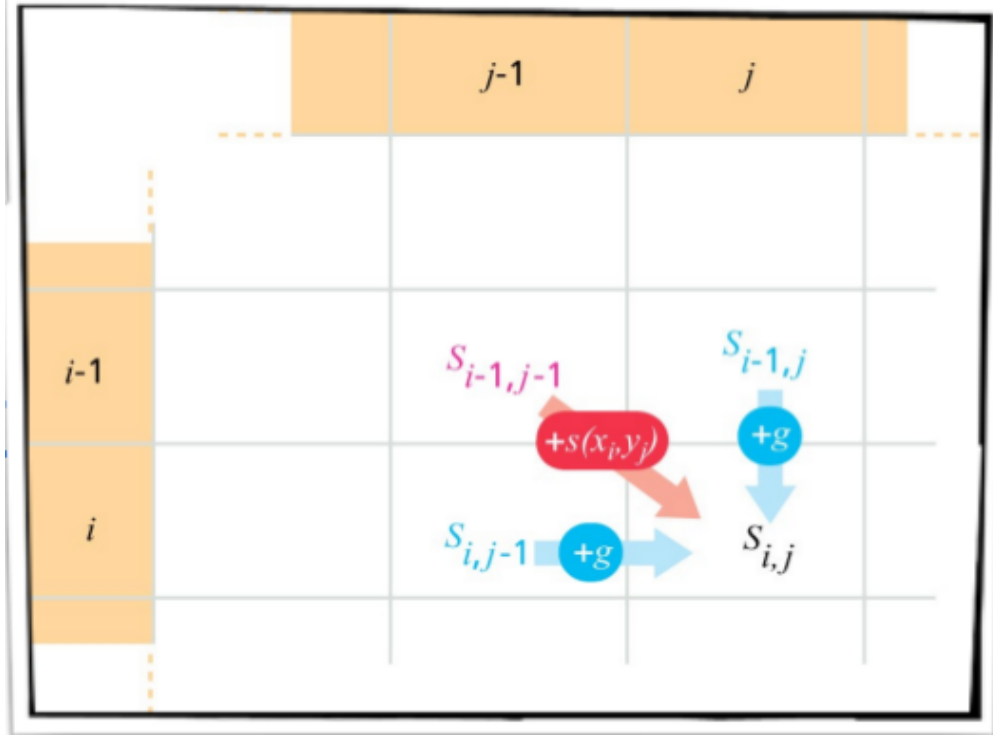
1. Fait correspondre la lettre précédente de la séquence 1 à un espace (on vient de la case du haut)
2. Fait correspondre la lettre précédente de la séquence 2 à un espace (on vient de la case de gauche)
3. Fait correspondre la lettre précédente de la séquence 1 avec la lettre précédente de la séquence 2 (on vient de la case en diagonale)

Pour chacune des trois possibilités, on prend le coût de la case précédente et on y ajoute le coût de la transformation vers la case courante. Pour cela, on doit connaître le coût de la correspondance entre 2 acides aminés (donné dans les matrices de substitution PAM et BLOSUM, par exemple) et celui entre un acide aminé et un espace.

Par exemple, on pourrait choisir une pénalité constante de 4 et comme matrice de substitution BLOSUM62 (qui indique par exemple que le coût de substitution de l'acide aminé R par C est -3).

Le score final sera le maximum de ces 3 scores:

$$\max(S(i-1, j) + G, S(i, j-1) + G, S(i-1, j-1) + \text{coutRemplacement}(i, j))$$



Algorithme sans pénalité affine

La première ligne et colonne de la matrice sont un cas spécial, car la formule précédente n'a pas de sens. En effet, il n'y a pas de voisin à gauche ou en diagonale pour la première colonne. Similairement, il n'y a pas de voisin en haut ou en diagonale pour la première ligne. Heureusement, ces 2 cas correspondent à une séquence d'espaces et on peut facilement réviser notre formule:

$$S(i, j) = \max \begin{cases} i * G & \text{si } i \geq 0 \text{ et } j = 0 \\ j * G & \text{si } i = 0 \text{ et } j \geq 0 \\ \max(S(i-1, j) + G, S(i, j-1) + G, S(i-1, j-1) + \text{coutRemplacement}(i, j)) & \text{ailleurs} \end{cases}$$

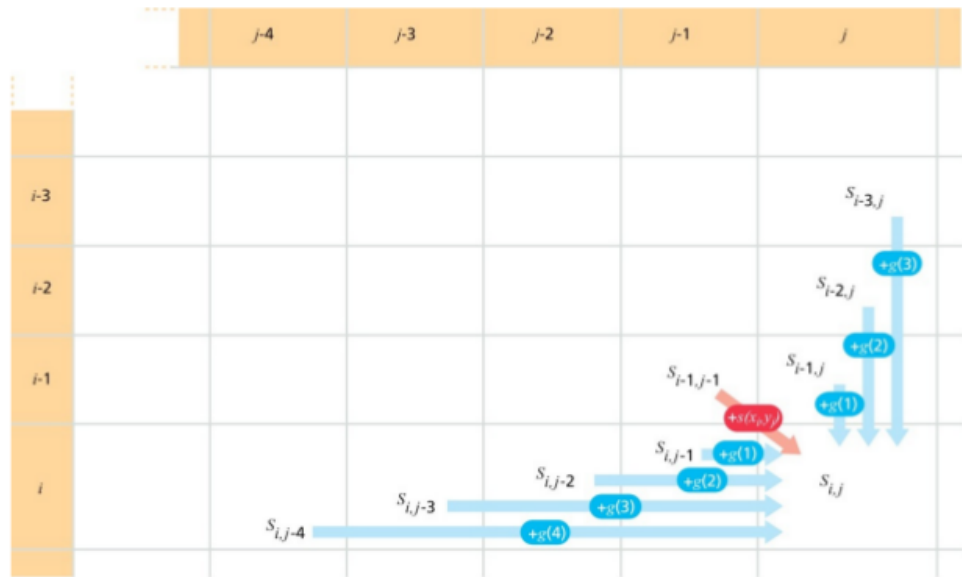
### 3.3 Pénalité affine

Lorsque la pénalité est affine, cela vaut dire que commencer une séquence d'espaces a un coût différent que de la prolonger. Par exemple, le 1er espace peut avoir un coût de 10 ( $I = 10$ ) alors que les espaces qui le suivent n'ont qu'un coût de 2 ( $E = 2$ ).

Pour calculer le score d'une case, il faut tester toutes les séquences d'espaces venant du haut et de la gauche (vu que le coût des espaces n'est plus linéaire) plutôt que juste tester le chemin via la case directement à gauche ou en haut.

La façon la plus simple de calculer ce score est de calculer les scores venant de chacun des voisins (dont tous ceux à gauche et en haut) et de prendre le maximum. Malheureusement, cette approche augmente la complexité de l'algorithme parce qu'il peut y avoir un grand nombre de voisins.

Si  $n$  et  $m$  sont les tailles des 2 séquences, avec  $n > m$ , alors la complexité de cette approche est  $O(mn^2)$ , parce pour une case  $X$ , il y a approximativement  $\frac{n}{2}$  voisins qui mènent à la case via des



Algorithme global naïf avec pénalité affine

espaces et qu'il faut considérer pour calculer le score.

### 3.3.1 Une meilleure approche

Il est possible de garder la complexité de  $O(n * m)$ . L'idée est simple. Soit  $V(i, j)$  le meilleur score que l'on peut obtenir en arrivant à la case  $(i, j)$  de la gauche. Comment peut-t-on le calculer?

Si on vient de la gauche pour aboutir à la case courante, alors on a soit:

#### 1. Commencé une séquence d'espaces à partir de la case directement à gauche

Dans ce cas, le score est très facile à calculer. C'est simplement le score de la case à gauche  $(i-1, j)$  + le coût de commencer une nouvelle séquence d'espaces:  $S(i-1, j) - I$ .

#### 2. Continué une séquence d'espaces (qui a commencé avant la case directement à gauche)

Dans ce cas, on continue une séquence d'espaces. On sait que le coût pour passer de la case  $(i-1, j)$  à la case  $(i, j)$  est de  $E$ , parce qu'on prolonge une séquence d'espaces. On sait aussi que la séquence d'espaces a commencé à gauche de  $(i-1, j)$ . Il ne reste donc plus qu'à découvrir quel est le meilleur score que l'on peut obtenir en arrivant en  $(i-1, j)$  de la gauche.

Heureusement, on connaît ce coût! C'est  $V(i-1, j)$ .

On en déduit que le meilleur score obtenu en prolongeant une séquence d'espaces est donc  $V(i-1, j) - E$ .

**Mettant le tout ensemble** Comme  $V(i, j)$  est le meilleur score que l'on peut obtenir en arrivant de la gauche à la case  $(i, j)$ , il faut donc prendre le score maximum entre ces 2 possibilités. On obtient donc l'expression suivante pour  $V(i, j)$ :

$$V(i, j) = \max \begin{cases} S(i-1, j) - I \\ V(i-1, j) - E \end{cases}$$

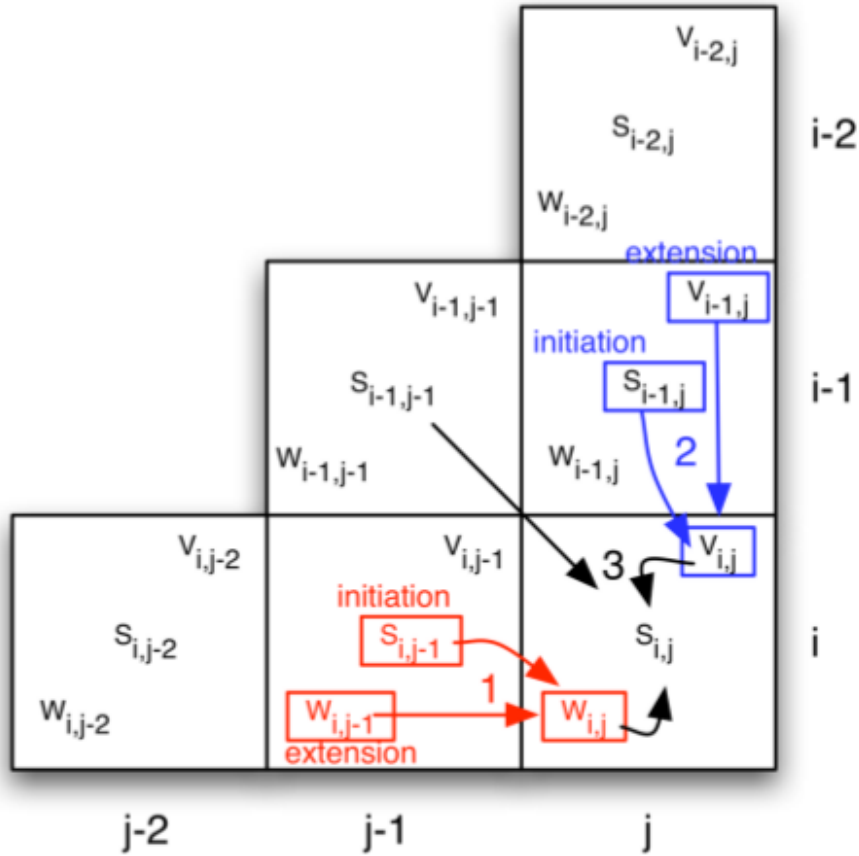
Par un raisonnement similaire, on peut calculer  $W(i, j)$ , qui est le meilleur score que l'on peut obtenir en arrivant du haut à la case  $(i, j)$

$$W(i, j) = \max \begin{cases} S(i, j-1) - I \\ W(i, j-1) - E \end{cases}$$

Avec toutes ces informations, il est donc facile de calculer  $S(i, j)$ . On peut y aboutir soit via une case à gauche, soit via une case en haut, soit via la case directement avant en diagonale. Le score gardé est le maximum de ces 3 possibilités.

$$S(i, j) = \max \begin{cases} S(i-1, j-1) + \text{coutCorrespondance}(i, j) \\ V(i, j) \\ W(i, j) \end{cases}$$

L'image suivant illustre cette approche:



Algorithme global avec pénalité affine efficace

Si on utilise 2 matrices supplémentaires (V et W), on peut donc calculer la valeur de chaque case en  $O(1)$ , car il n'y a que 5 valeurs à calculer au maximum. Vu qu'il y a  $n * m$  cases, cette approche permet donc d'avoir de nouveau une complexité de  $O(n * m)$ .

Pour les mêmes raisons qu'avant, la première colonne et la première ligne sont un cas spécial:  
Soit  $(i, j)$  un case. Si  $j = 0$  et  $i > 0$  alors

$$S(i, j) = I + (i - 1) * E$$

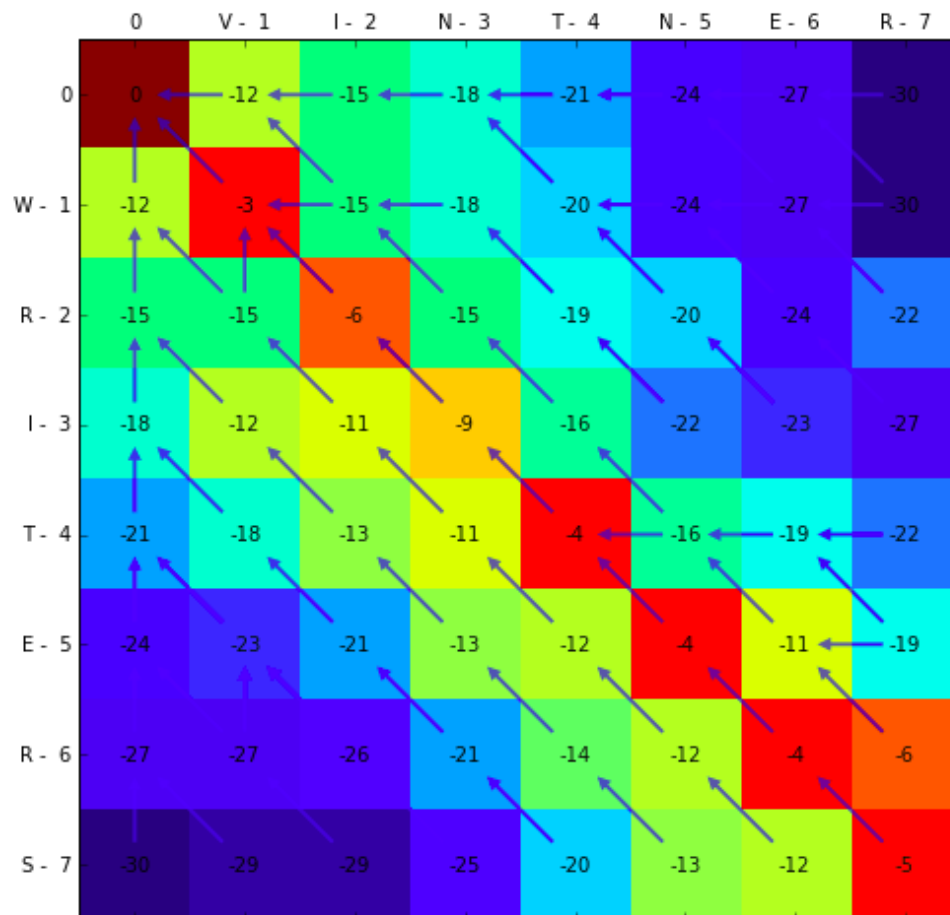
et si  $i = 0$  et  $j > 0$  alors

$$S(i, j) = I + (j - 1) * E$$

On a aussi que  $V(0, j) = V(i, 0) = W(0, j) = W(i, 0) = -\infty$

### 3.3.2 Étape 2: calculer le meilleur chemin

Imaginons qu'on ait calculé la matrice des scores, ayant obtenu ceci:



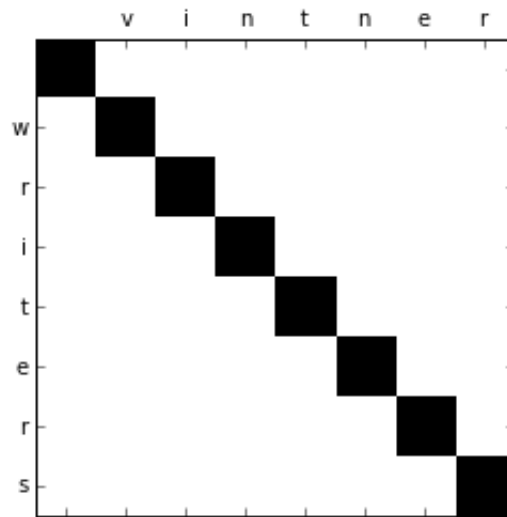
Exemple de matrice des scores

Dans cette matrice, il y a des flèches entre des cases. Lorsqu'on calcule le score d'une cas, on prend le maximum des scores calculés via ses 3 voisins. Pour chaque voisin X qui permet



d'obtenir ce score maximal, on ajoute une flèche entre X et Y. On retient donc quelle action (insérer, substituer, effacer) a mené à la case courante.

On peut utiliser ces flèches pour trouver les chemins optimaux entre la dernière case et la première. Dans cet exemple, on peut voir que le meilleur chemin est la diagonale de la matrice.



Chemin trouvé dans l'exemple

Dans certains cas, il y a plusieurs flèches entrant dans une cellule. Dans ce cas, il y a plusieurs chemins et donc plusieurs alignements globaux optimaux.

### 3.3.3 Étape 3: recréer l'alignement à partir du chemin

Une fois qu'on a obtenu les chemins, il est très facile de recréer l'alignement optimal, car chaque déplacement a un sens précis:

1. Si on descend, on a fait correspondre l'acide aminé courant de la 1ere séquence avec un espace (action: effacer)
2. Si on va à droite, on a fait correspondre l'acide aminé courant de la 2eme séquence avec un espace (action: insérer)
3. Si on va en diagonale, on a fait correspondre l'acide aminé courant de la 1ere séquence avec l'acide aminé courant de la 2ème séquence (action: substituer)

Avec ces 3 règles, on peut facilement recréer l'alignement et donc d'obtenir un résultat comme ceci:

```
writers
|...
vintner
```

Alignement de l'exemple

### 3.3.4 Partie 1: créer les Abstract Data Types

Nous commençons par créer une classe Sequence, qui est l'ADT qui représente une séquence d'acides aminés et toutes les opérations qu'on peut exécuter sur une séquence (exemple: retourner un acide aminé à une certaine position ou visualiser la séquence en format FASTA).

```
In [2]: class Sequence:
    def __init__(self, acids, annotation=None):
        self.acids = acids.upper()
        self.annotation = annotation

    def __getitem__(self, position):
        """ Renvoie l'acide aminé à la position donnée. Syntaxe: sequence[position] """
        return self.acids[position]

    def toFASTAFormat(self):
        return ">" + self.annotation + "\n" + self.acids

    def __repr__(self):
        return "Annotation: %s \nAcids: %s" % (self.annotation, self.acids)

    def length(self):
        return len(self.acids)
```

Nous créons aussi une classe Score, qui est un ADT qui représente une matrice de substitution et les opérations qu'on peut exécuter sur cette matrice.

```
In [3]: class Score:
    def __init__(self):
        self.letters = set()
        self.costs = {}

    def getNumLetters(self):
        return len(self.letters)

    def getLetters(self):
        return self.letters

    def getCost(self, a, b):
        return self.costs[(a, b)]

    def addCost(self, a, b, cost):
        self.letters.add(a)
        self.letters.add(b)
        self.costs[(a.upper(), b.upper())] = cost
```

### 3.4 Parser

Nous devons créer des objets des classes Sequence et Score à partir des fichiers fournis. Il faut donc créer un parser pour les fichiers des séquences (FASTA) et pour les fichiers avec les matrices de substitution (BLOSUM et PAM).

### 3.5 FASTA

Les séquences d'acides aminés sont codées dans le format FASTA. Ce format est très simple. Chaque séquence est encodée par une annotation et ensuite la liste d'acides aminés. Voici un exemple:

```
>sp|Q13526|PIN1_HUMAN Peptidyl-prolyl cis-trans isomerase NIMA-interacting
1 OS=Homo sapiens GN=PIN1 PE=1 SV=1 MADEEKLPPGWEKRMSRSS-
GRVYYFNHITNASQWERPSGNSSSGGKNGQGEPARVRCSHL LVKHSQSRRPSS-
WRQEKITRTKEEALELINGYIQKIKSGEEDFESLASQFSDCSSAKARG DLGAFS-
RGQMOKPFEDASFALRTGEMSGPVFTDSGIHILRTE
```

La première ligne (qui commence par >) est une annotation contenant des informations sur la séquence. Ensuite viennent une série de lignes contenant la séquence d'acides aminés (avec un maximum de 80 caractères par ligne, en général, pour faciliter la lisibilité).

Un fichier FASTA est constitué d'une série de séquences formatées de cette façon.

```
In [4]: def parseFASTASequencesFromFile(filename):
        with open(filename, 'r') as f:
            return parseFASTASequences(f)

        def parseFASTASequences(lines):
            sequences = []
            annotation, acids = "", ""
            for line in lines:
                line = line.strip()

                justEndedPreviousSequence = (len(line) == 0 or line.startswith(">"))
                if justEndedPreviousSequence:
                    sequences.append(Sequence(acids, annotation))

                    if line.startswith(">"):
                        annotation = line[1:]
                        acids = ""
                    else:
                        acids += line.upper()

            return sequences
```

Appliquons ces parsers aux fichiers donnés:

```
In [5]: sequencesFileNames = ['protein-sequences.fasta', 'WW-sequence.fasta']
        proteinSequences, wwSequences = [parseFASTASequencesFromFile(filename) for
        sequence = proteinSequences[0]
```

Faisons quelques tests pour vérifier notre fonction:

```
In [6]: print("La première sequence est: \n")
        print(sequence)
```

La première sequence est:

Annotation: sp|P46937|YAP1\_HUMAN Transcriptional coactivator YAP1 OS=Homo sapiens C  
Acids: MDPGQQPPPPQAPQGQGQPPSQPPQGQGPPSGPGQPAPAATQAAPQAPPAGHQIVHVRGDSETDLEALFNAVMNPF

```
In [7]: print("\nLe premier acide aminé de la séquence est: " + sequence[0])
```

Le premier acide aminé de la séquence est: M

### 3.5.1 Matrices de substitution

Les matrices de substitution sont aussi dans un format simple et intuitif. Le fichier commence par quelques lignes de commentaires (commencent par #) et puis est suivi de la grille des coûts de substitution.

Voici un exemple simplifié:

```
# Ma matrice
# Il y 5 acides aminés dans cet exemple
A  R  N  D  C
A  4 -1 -2 -2  0 -
R -1  5  0 -2 -3
N -2  0  6  1 -3
D -2 -2  1  6 -3
C  0 -3 -3 -3  9
```

```
In [8]: def parseSubstitutionMatrixFromFile(filename):
        with open(filename, 'r') as f:
            return parseSubstitutionMatrix(f)

        def parseSubstitutionMatrix(lines):
            score = Score()

            for line in lines:
                if line.startswith('#') or line.strip() == "": # Skip comments
                    continue
                elif line.startswith(' '): # First line with letters
                    letters = line.strip().upper().split()
                else:
                    letterA, *costs = line.strip().upper().split()
                    for (letterB, cost) in zip(letters, costs):
                        score.addCost(letterA.upper(), letterB, int(cost))

            return score
```

Essayons d'appliquer ce parser à nos fichiers:

```
In [9]: matricesFileNames = ['blosum62.txt', 'blosum80.txt', 'pam60.txt', 'pam120.txt']
        blosum62, blosum80, pam60, pam120 = [parseSubstitutionMatrixFromFile(fileName) for fileName in matricesFileNames]

        print("\nLe coût de substitution de R et C dans Blosum62 devrait être -3 et est -3")
        letters = blosum62.getLetters()
        print("Il y a %d colonnes dans Blosum62: %s" % (len(letters), ''.join(sorted(letters))))
```

Le coût de substitution de R et C dans Blosum62 devrait être -3 et est -3  
Il y a 24 colonnes dans Blosum62: \*ABCDEFGHIKLMNPQRSTUVWXYZ

### 3.5.2 Partie 2: coder l'algorithme de Needleman-Wunsch

```
In [30]: def printMatrix(matrix):
        for line in matrix:
            print(line)

        def makeMatrix(width, height, val=0):
            if type(val) in (int, str, bool):
                return [[val] * width for _ in range(height)]
            elif val == []:
                return [[[ ] for _ in range(width)] for _ in range(height)]
            else:
                return [[val for _ in range(width)] for _ in range(height)]

        def getValue(matrix, x, y):
            if x < 0 or y < 0:
                return float('-inf')
            else:
                return matrix[y][x]

        def getSolutionsFromArrowGridHelper(previous, k, x, y, isGlobal=True):
            completePaths = []
            incompletePaths = [(x, y)]
            numPaths = 1

            while len(incompletePaths) > 0 and len(completePaths) <= k:
                path = incompletePaths.pop()
                lastX, lastY = path[0]

                for (i, previousCoords) in enumerate(previous[lastY][lastX]):
                    if i > 0: # 2+ paths from this cell
                        numPaths += 1
                        if numPaths > k: # No need to consider new paths if we already have k
                            break
```

```

        # Ceci gère le cas où on fait de l'alignement local, que l'on
        if not isGlobal and previousCoords is None:
            completePaths.append(path)
            continue

        newX, newY = previousCoords
        isComplete = (newX == 0 and newY == 0)

        if isComplete:
            completePaths.append([(newX, newY)] + path)
        else:
            incompletePaths.append([(newX, newY)] + path)

    return completePaths

def getSolutionsFromArrowGrid(previous, k, x=None, y=None, isGlobal=True):
    # Crée jusqu'à k chemins à partir de la matrice des fleches et d'une p
    if x is None or y is None:
        x, y = len(previous[0]) - 1, len(previous) - 1

    return getSolutionsFromArrowGridHelper(previous, k, x, y, isGlobal)

def getPathScores(paths, s):
    # Calcule le score associé à chacun des chemins
    pathScores = {}
    for path in paths:
        x, y = path[-1]
        pathScores[(x, y)] = s[y][x]

    return pathScores

def getBestGlobalAlignments(sequence1, sequence2, score, startGap=4, keep
    """
    Finds the best (at most k) global alignments between 2 sequences.
    k: the maximum number of alignment returned (note: less than k aligne
    """

    height, width = sequence1.length() + 1, sequence2.length() + 1
    v, w, s = [makeMatrix(width, height) for _ in range(3)]
    previous = makeMatrix(width, height, [])

    # Initialize first row
    for i in range(1, width):
        s[0][i] = -(startGap + (i - 1) * keepGap)
        v[0][i] = float('-inf')
        previous[0][i] = [(i - 1, 0)]

    # Initialize first column

```

```

for i in range(1, height):
    s[i][0] = -(startGap + (i - 1) * keepGap)
    w[i][0] = float('-inf')
    previous[i][0] = [(0, i - 1)]

# Compute the score matrix
for x in range(1, width):
    for y in range(1, height):
        v[y][x] = max(s[y - 1][x] - startGap, v[y - 1][x] - keepGap)
        w[y][x] = max(s[y][x - 1] - startGap, w[y][x - 1] - keepGap)
        replacementCost = score.getCost(sequence1[y - 1], sequence2[x])

        coords = [(x, y - 1), (x - 1, y), (x - 1, y - 1)]
        costs = [v[y][x], w[y][x], s[y - 1][x - 1] + replacementCost]

        s[y][x] = bestScore = max(costs)
        previous[y][x] = []
        for coord, cost in zip(coords, costs):
            if cost == bestScore:
                previous[y][x].append(coord)

score = s[height - 1][width - 1]

# We could improve the performance here, by either returning the paths
# making it a lazy operation
paths = getSolutionsFromArrowGrid(previous, k)
pathScores = getPathScores(paths, s)

return s, paths, score, previous, pathScores

```

### 3.6 Fonctions de visualisation

On définit quelques fonctions pour visualiser les chemins et la matrice des scores, ce qui sera très utile pour comprendre les actions de l'algorithme et les chemins créés.

```

In [11]: from pprint import pprint
def showPath(path, matrix=None, seq1=None, seq2=None):
    x, y = path[-1]
    if matrix is None:
        height, width = x + 1, y + 1
    else:
        height, width = len(matrix), len(matrix[0])

    mat = makeMatrix(width, height)
    for x, y in path:
        mat[y][x] = 1

```

```

plt.matshow(mat, cmap='Greys')

# Axis Labels
addSequenceLabelsToAxis(seq1, seq2, withNumbers=False)

plt.show()

def addSequenceLabelsToAxis(seq1, seq2, withNumbers=True):
    def getLabels(acids):
        if withNumbers:
            return ['0'] + ['%s - %d' % (acid.upper(), i) for (i, acid) in enumerate(acids)]
        else:
            return ' ' + acids

    if seq1 is not None:
        plt.yticks(range(seq1.length() + 1), getLabels(seq1.acids))

    if seq2 is not None:
        plt.xticks(range(seq2.length() + 1), getLabels(seq2.acids))

def printColoredMatrix(matrix, arrowMatrix=None, seq1=None, seq2=None, figsize=None):
    matrix = np.array(matrix)
    fig, ax = plt.subplots(figsize=figsize)
    ax.matshow(matrix)

    # Values
    for (i, line) in enumerate(matrix):
        for (j, val) in enumerate(line):
            ax.text(j, i, str(val), va='center', ha='center')

    # Axis Labels
    addSequenceLabelsToAxis(seq1, seq2)

    # Arrows
    arrowprops = dict(facecolor=arrowColor, alpha=0.6, lw=0,
                      shrink=0.2, width=2, headwidth=7, headlength=7)

    if arrowMatrix is None:
        plt.show()
        return

    for (y, line) in enumerate(arrowMatrix):
        for (x, previousCells) in enumerate(line):
            for coords in previousCells:
                if coords is not None:
                    ax.annotate("", xy=coords, xytext=(x, y), arrowprops=arrowprops)

    plt.show()

```



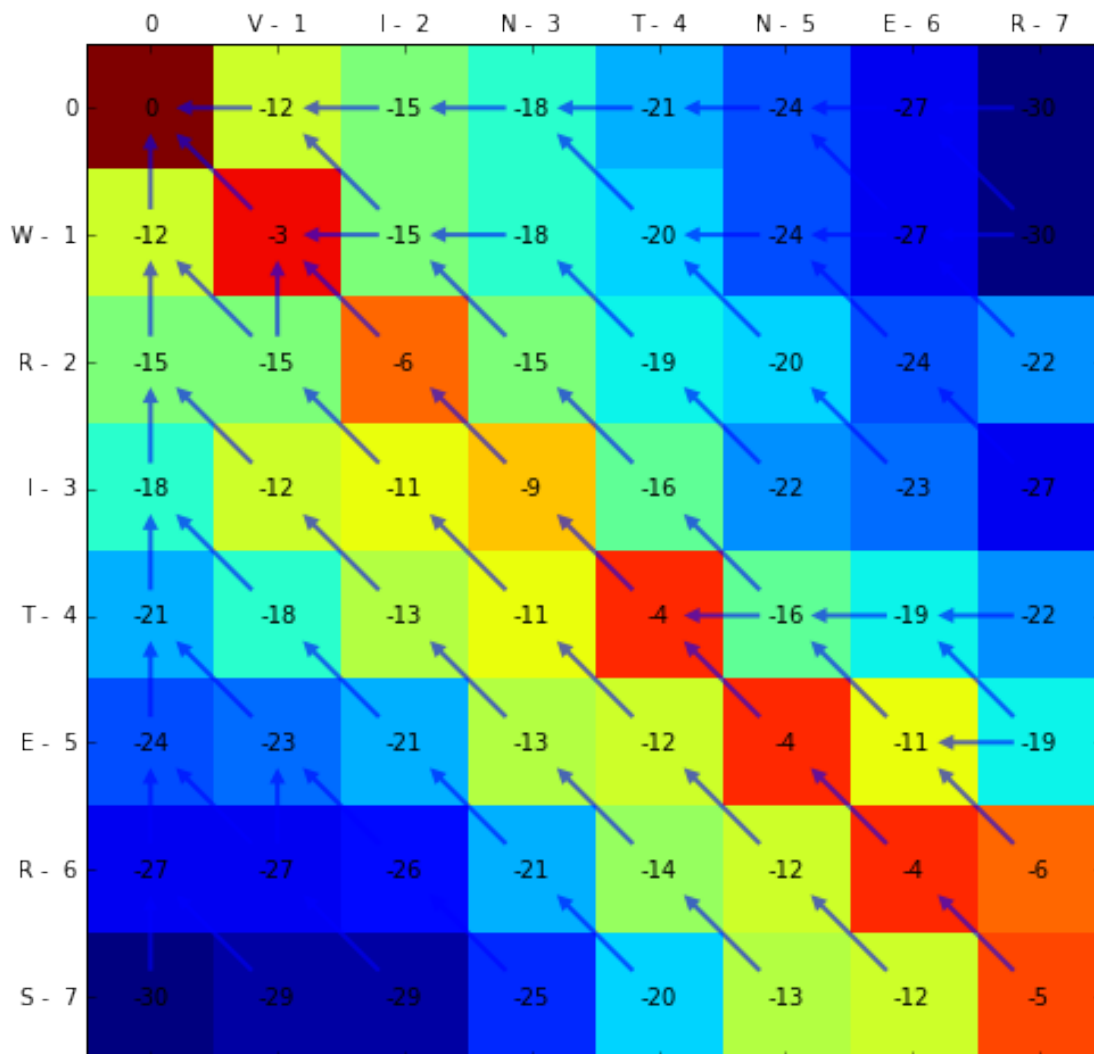
### 3.7 Alignement global: un exemple

Pour comprendre ce que fait l'algorithme, appliquons le à un exemple. On va aligner la séquence "writers" avec la séquence "vintner", voir l'alignement obtenu et visualiser la matrice de scores et le chemin.

```
In [12]: a = Sequence('writers')
         b = Sequence('vintner')

         scoreMatrix, paths, score, arrowMatrix, pathScores = getBestGlobalAlignement(a, b)
         print("The score matrix is\n")
         printColoredMatrix(scoreMatrix, arrowMatrix, seq1=a, seq2=b)
         print(score)
```

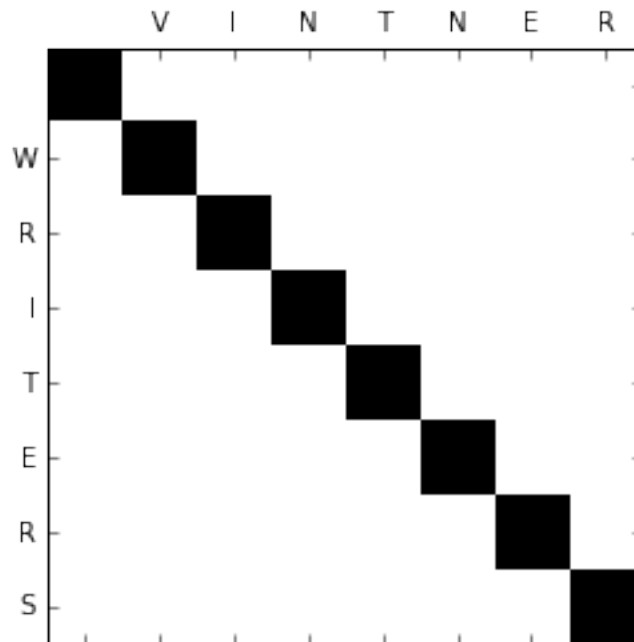
The score matrix is



-5

```
In [13]: print("Number of paths found: %d\n" % len(paths))
         for path in paths:
             showPath(path, seq1=a, seq2=b)
         print(path)
```

Number of paths found: 1



```
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7)]
```

```
In [14]: print("\nThe arrow matrix is\n")
         printMatrix(arrowMatrix)
```

The arrow matrix is

```
[[[]], [(0, 0)], [(1, 0)], [(2, 0)], [(3, 0)], [(4, 0)], [(5, 0)], [(6, 0)]]
[[[(0, 0)], [(0, 0)], [(1, 1), (1, 0)], [(2, 1)], [(3, 0)], [(4, 1)], [(5, 1), (5, 0)]]
```

```

[[ (0, 1)], [(1, 1), (0, 1)], [(1, 1)], [(2, 1)], [(3, 1)], [(4, 1)], [(5, 1)], [(6,
[(0, 2)], [(0, 2)], [(1, 2)], [(2, 2)], [(3, 2)], [(4, 2)], [(5, 2)], [(6, 2)]]
[(0, 3)], [(0, 3)], [(1, 3)], [(2, 3)], [(3, 3)], [(4, 4), (4, 3)], [(5, 4)], [(6,
[(0, 4)], [(0, 4)], [(1, 4)], [(2, 4)], [(3, 4)], [(4, 4)], [(5, 4)], [(6, 5), (6,
[(0, 5)], [(1, 5), (0, 5)], [(1, 5)], [(2, 5)], [(3, 5)], [(4, 5)], [(5, 5)], [(6,
[(0, 6)], [(0, 6)], [(1, 6)], [(2, 6)], [(3, 6)], [(4, 6)], [(5, 6)], [(6, 6)]]

```

## 4 Retrouver les alignements

Une fois qu'on a fait retrouvé le chemin optimal, on peut créer l'alignement qui y correspond.

```

In [15]: def printAlignments(path, sequence1, sequence2, subMatrix):
    text1, correspondance, text2 = "", "", ""
    firstCoord, *remainingCoords = path
    prevX, prevY = firstCoord

    for (i, (x, y)) in enumerate(remainingCoords):
        if x == prevX and y == prevY + 1: # Down
            text1 += sequence1[y - 1]
            text2 += "_"
            correspondance += " "
        elif x == prevX + 1 and y == prevY: # Right
            text1 += "_"
            text2 += sequence2[x - 1]
            correspondance += " "
        elif x == prevX + 1 and y == prevY + 1: # Diagonal
            text1 += sequence1[y - 1]
            text2 += sequence2[x - 1]

            letter1, letter2 = text1[-1], text2[-1]
            if letter1 == letter2:
                correspondance += "|"
            elif subMatrix.getCost(letter1, letter2) >= 3:
                correspondance += ":"
            elif subMatrix.getCost(letter1, letter2) > -2:
                correspondance += "."
            else:
                correspondance += " "

        prevX, prevY = x, y

    if i % 80 == 0 and i > 0:
        print('\n'.join([text1, correspondance, text2]))
        print()
        text1, correspondance, text2 = "", "", ""

```

```

print('\n'.join([text1, correspondance, text2]))

def printAlignementResult(seq1, seq2, result, subMatrix, showScore = True,
                           scoreMatrix, paths, score, arrowMatrix, pathScores = result
                           if showScore:
                               print("The best score is %d\n" % score)

                               print(seq1, end="\n\n")
                               print(seq2, end="\n\n")
                               print("Alignements found: %d (may be artificially limited)\n" % len(paths))
                               for (i, path) in enumerate(paths, start=1):
                                   if len(paths) > 0:
                                       print("Alignement number %d with score %d: \n" % (i, pathScores[i]))

                                   if showLength:
                                       print("The length of the alignment is %d" % len(path))

                                   startX, startY = path[0]
                                   endX, endY = path[-1]
                                   print("Sequence 1: %d-%d" % (startY + 1, endY))
                                   print("Sequence 2: %d-%d" % (startX + 1, endX))

                                   printAlignments(path, seq1, seq2, subMatrix)
                                   print()

```

Maintenant qu'on a défini la fonction, regardons l'alignement créé pour l'exemple.

```

In [16]: def compareSequences(seq1, seq2, subMatrix=blosum62, startGap=4, keepGap=1,
                               result = getBestGlobalAlignements(seq1, seq2, subMatrix, startGap, keepGap,
                               scoreMatrix, score, previous, arrowMatrix, pathScores = result
                               printAlignementResult(seq1, seq2, result, subMatrix, showScore)
                               #printColoredMatrix(scoreMatrix, arrowMatrix)

                               compareSequences(a, b, startGap=12, keepGap=3)

```

The best score is -5

Annotation: None  
Acids: WRITERS

Annotation: None  
Acids: VINTNER

Alignements found: 1 (may be artificially limited)

Alignement number 1 with score -5:

```
Sequence 1: 1-7
Sequence 2: 1-7
WRITERS
|...
VINTNER
```

Prenons des séquences réelles (présentes dans le fichier *ww-sequences.fasta*) et comparons-les.

```
In [17]: ww1, ww2 = wwSequences[0], wwSequences[1]
         compareSequences(ww1, ww2)
```

The best score is 77

```
Annotation: sp|P46937|171-204
Acids: VPLPAGWEMAKTSSGQRYFLNHIDQTTTWQDPRK
```

```
Annotation: sp|P46934|610-643
Acids: SPLPPGWEERQDILGRITYYVNHESRRTQWKRPPTP
```

Alignements found: 1 (may be artificially limited)

Alignement number 1 with score 77:

```
Sequence 1: 1-34
Sequence 2: 1-34
VPLPAGWEMAKTSSGQRYFLNHIDQTTTWQDPRK
| | | . | | |   . . .   | . . | : . | |   . . . | . | .   | . .
SPLPPGWEERQDILGRITYYVNHESRRTQWKRPPTP
```

On peut regarder quelle paires de séquences du fichier *ww-sequences.fasta* ont les meilleurs alignements.

```
In [31]: import itertools
```

```
def getBestMatches(sequences):
    # Trouve les paires de séquences qui ont le meilleur alignment global
    # et renvoie aussi les résultats associés
    bestScore = float('-inf')
    bestResults = {}

    for seq1, seq2 in itertools.combinations(sequences, 2):
        scoreMatrix, paths, score, arrowMatrix, pathScores = result = getBestMatches(seq1, seq2)
        if score > bestScore:
            bestScore = score
            bestResults = {(seq1, seq2): result}
```

```

        elif score == bestScore:
            bestResults[(seq1, seq2)] = result

    return bestResults, bestScore

bestResults, score = getBestMatches(wwSequences)
print("The best matches have a score of %d" % score)
print("They are between these sequences: ")
i = 1
for (seq1, seq2), result in bestResults.items():
    print("\n----- Sequences %d ----- \n" % i)
    printAlignmentResult(seq1, seq2, result, subMatrix=blosum62, showScore=True)
    i += 1

```

The best matches have a score of 117

They are between these sequences:

----- Sequences 1 -----

Annotation: sp|P46934|610-643

Acids: SPLPPGWEERQDILGRITYYVNHESRRTQWKRPPTP

Annotation: sp|P46934|892-925

Acids: GPLPPGWEERTHTDGRIFYINHNHNIKRTQWEDPRL

Alignements found: 2 (may be artificially limited)

Alignment number 1 with score 117:

Sequence 1: 1-34

Sequence 2: 1-34

SPLPPGWEER\_\_QDILGRITYYVNHESRRTQWKRPPTP

.||||||||| .| ||.:|:|. .||||. |.

GPLPPGWEERTHTD\_\_GRIFYINHNHNIKRTQWEDPRL

Alignment number 2 with score 117:

Sequence 1: 1-34

Sequence 2: 1-34

SPLPPGWEER\_Q\_DILGRITYYVNHESRRTQWKRPPTP

.||||||||| . | ||.:|:|. .||||. |.

GPLPPGWEERTHTD\_\_GRIFYINHNHNIKRTQWEDPRL

## 4.1 Comparaison avec LALIGN

Testons cette paire de séquences optimales avec [LALIGN](#), pour comparer le résultat obtenu: n-w opt: 115 Z-score: 174.3 bits: 34.0 E(1): 8.9e-36 global/local score: 115; 55.6

10 20 30 SPLPPGWEERQ-DILGRTYYVNHESTRRTQWKRPTP ..... : :: ..... :  
GPLPPGWEERTHTD-GRIFYINHNKRTQWEDPRL 10 20 30 Voici donc les informations données par [LALIGN](#):

- **Score:** 115
- **Nombre d'acides aminés identiques dans l'alignement:** 55.6%
- **Nombre d'acides aminés similaires dans l'alignement:** 72.2%

On peut voir que le résultat est très proche. Le score obtenu est légèrement différent (115 plutôt que 117) et l'alignement aussi (seule la position du Q est différente dans le résultat de LALIGN).

LALIGN est un outil avancé, donc la légère différence observée est probablement due à un algorithme légèrement optimisé ou amélioré.

### 4.1.1 Origine des séquences et Uniprot

Une question intéressante est "Est-ce que ces 2 séquences viennent de la même protéine et quelles informations peut-t-on obtenir sur ces séquences?"



Les 2 séquences ont comme annotations:

sp | P46934 | 610-643

et

sp | P46934 | 892-925

On peut donc directement voir qu'elles appartiennent à la même protéine, car elles ont le même identifiant: P46934. La première a comme position 610-643 et la deuxième la position 892-925 dans la protéine. Pour en savoir sur cette protéine, nous allons utiliser [Uniprot](#) pour les comparer.

**Protein** | **E3 ubiquitin-protein ligase NEDD4**  
**Gene** | **NEDD4**  
**Organism** | *Homo sapiens (Human)*  
**Status** |  **Reviewed** - Annotation score:  - Experimental evidence at protein level<sup>i</sup>

Information générale sur la protéine (global)

Plus bas, on peut accéder à l'information sur les domaines:

On peut donc voir que les 2 séquences correspondent aux domaines WW1 et WW4 de la protéine. Vu que ces 2 séquences correspondent à un domaine du même type, il n'est donc pas étonnant qu'elles soient très similaires.

Note: le domaine WW est assez connu en bioinformatique, ayant même [sa propre page Wikipedia](#).

Domains and Repeats

Feature key	Position(s)	Description	Actions	Graphical view	Length
Domain <sup>i</sup>	610 – 643	WW 1 ⓘ PROSITE-ProRule annotation ▾	<a href="#">Add</a> <a href="#">BLAST</a>		34
Domain <sup>i</sup>	767 – 800	WW 2 ⓘ PROSITE-ProRule annotation ▾	<a href="#">Add</a> <a href="#">BLAST</a>		34
Domain <sup>i</sup>	840 – 873	WW 3 ⓘ PROSITE-ProRule annotation ▾	<a href="#">Add</a> <a href="#">BLAST</a>		34
Domain <sup>i</sup>	892 – 925	WW 4 ⓘ PROSITE-ProRule annotation ▾	<a href="#">Add</a> <a href="#">BLAST</a>		34
Domain <sup>i</sup>	984 – 1318	HECT ⓘ PROSITE-ProRule annotation ▾	<a href="#">Add</a> <a href="#">BLAST</a>		335

Region

Feature key	Position(s)	Description	Actions	Graphical view	Length
Region <sup>i</sup>	578 – 981	Mediates interaction with TNIK ⓘ By similarity	<a href="#">Add</a> <a href="#">BLAST</a>		404

Compositional bias

Feature key	Position(s)	Description	Actions	Graphical view	Length
Compositional bias <sup>i</sup>	68 – 215	Ser-rich	<a href="#">Add</a> <a href="#">BLAST</a>		148

Domain<sup>i</sup>

The WW domains mediate interaction with LITAF, RNF11, WBP1, WBP2, TMEPA1, NDFIP1 and PRRG2. ⓘ By similarity

Keywords - Domain<sup>i</sup>

Repeat

Information sur les domaines de la protéine (global)

## 5 Alignement local

L'alignement global cherche le meilleur alignement entre 2 séquences. L'alignement local est plus général, parce qu'il cherche le meilleur alignement entre des parties des 2 séquences, qui peuvent être plus courtes que les séquences. Cela permet de trouver des *morceaux* de séquences qui sont très similaires.

La façon la plus naïve de trouver le meilleur alignement local est d'appliquer l'algorithme d'alignement global à toutes les paires des *morceaux* des 2 séquences et de garder celles qui ont le meilleur score. Bien sûr, ceci est extrêmement peu efficace vu qu'il y a environ  $n^2 * m^2$  paires possibles. Il faut donc trouver une meilleure approche.

### 5.1 L'algorithme de Smith Waterman

Imaginons qu'on appliqué l'alignement global, calculé la matrice des scores et obtenu un chemin dont les cases ont les scores suivants:

$$[0, -2, -4, -5, -2, 1, 2, 5, 8, 10, 7, 6]$$

On peut remarquer plusieurs choses:

1. L'alignement qui se termine à la case de score 10 est meilleur que celui qui continue jusqu'au bout et a un score 6. Il faut donc s'arrêter au maximum du chemin et ne pas garder les cases qui le suivent et font diminuer le score.
2. Certaines cases de cet alignement ont un score négatif. Si on ne éliminait ces acides aminés du chemin (et ceux qui le précédent, vu que le chemin doit être continu), alors le nouvel alignement ne pourrait être que meilleur, car il ne serait pas pénalisé par ces acides aminés dissimilaires.

Ces idées mènent à l'algorithme de *Smith-Waterman*, qui est une variante de l'algorithme de *Needleman-Wunsch*. Voici les différences:

- On n'admet pas de scores négatifs. Si le score calculé est négatif, on le remplace par 0.



- Le retour en arrière (pour trouver le chemin) commence à partir de la case avec le **score maximum de toute la matrice** et on s'arrête dès qu'on trouve un 0.

Si un score est négatif, cela veut dire que les séquences jusqu'à cette position n'ont pas de similarités. Mieux vaut donc mettre le score à 0 pour éliminer l'influence de ce mauvais alignement, et ainsi permettre de trouver les alignement locaux qui commencent à partir de cette case. Ainsi, on ne pénalise pas les séquences qui commencent à une position interne, ce qui permet donc de trouver le meilleur alignement local.

L'algorithme est donc régi par cette nouvelle équation:

$$S(i, j) = \max \begin{cases} S(i-1, j-1) + \text{coutCorrespondance}(i, j) \\ V(i, j) \\ W(i, j) \\ 0 \end{cases}$$

**Corollaire:** la première colonne et ligne ont un score de 0, vu que des scores négatifs ne sont pas tolérés.

### 5.1.1 Recherche du chemin et de l'alignement

La recherche du chemin (retour en arrière) commence à partir de la case avec le score maximum et s'arrête à la première case avec un score de 0.

Le création de l'alignement à partir du chemin ne change pas.

## 5.2 Exemple

On reprend l'exemple antérieur avec les séquences "writers" et "vintner". Voici la matrice des scores qu'on obtient:

Le maximum est la cellule en (7, 6) avec score 11. On crée le chemin à partir de là, suivant les flèches jusqu'à arriver à une case avec un score de 0. On obtient ce chemin:

Ceci correspond à l'alignement suivant

## 5.3 Alignements sous-optimaux

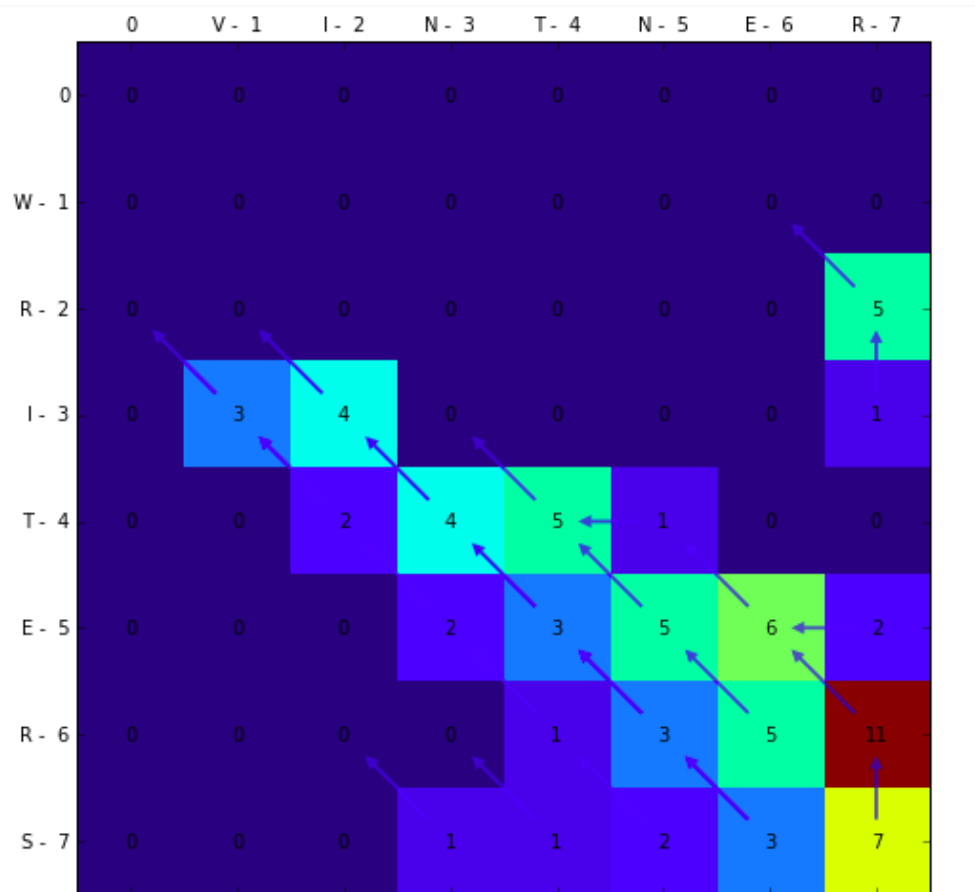
Il se peut qu'on souhaite trouver les 5 meilleurs alignements locaux, mais qu'il n'y ait que 2 alignement locaux optimaux. Lorsqu'il n'y a pas assez d'alignements locaux optimaux, on recherche des alignements sous-optimaux. Mais comment les trouver?

Imaginons qu'on ait trouvé le chemin avec le score maximal, comme dans l'image suivante:

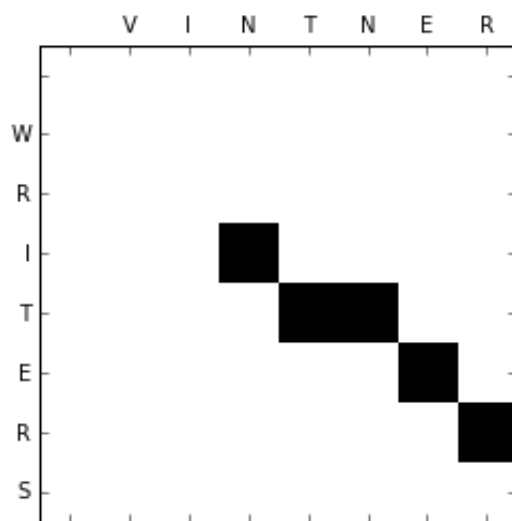
Après avoir trouvé la séquence à score maximal, on met toutes ces cellules à 0 et on recalcule la matrice dans le voisinage du chemin optimal. Ceci élimine des résultats possibles les chemins qui intersectent le chemin optimal qu'on vient de trouver, car les nouveau scores "découpe" ce chemin en deux. De cette façon, on peut trouver des alignements distincts qui peuvent avoir plus d'intérêt, car ils sont assez différents de l'alignement original.

Le reste de l'algorithme ne change pas (création du chemin et de l'alignement).

Si il faut trouver  $n$  alignements, on trouve l'alignement local optimal puis on exécute cette approche  $n - 1$  fois pour trouver  $n - 1$  alignements sous-optimaux. Note: une fois qu'on a mis un score à 0, il reste à 0 pendant le reste de l'exécution de l'algorithme.



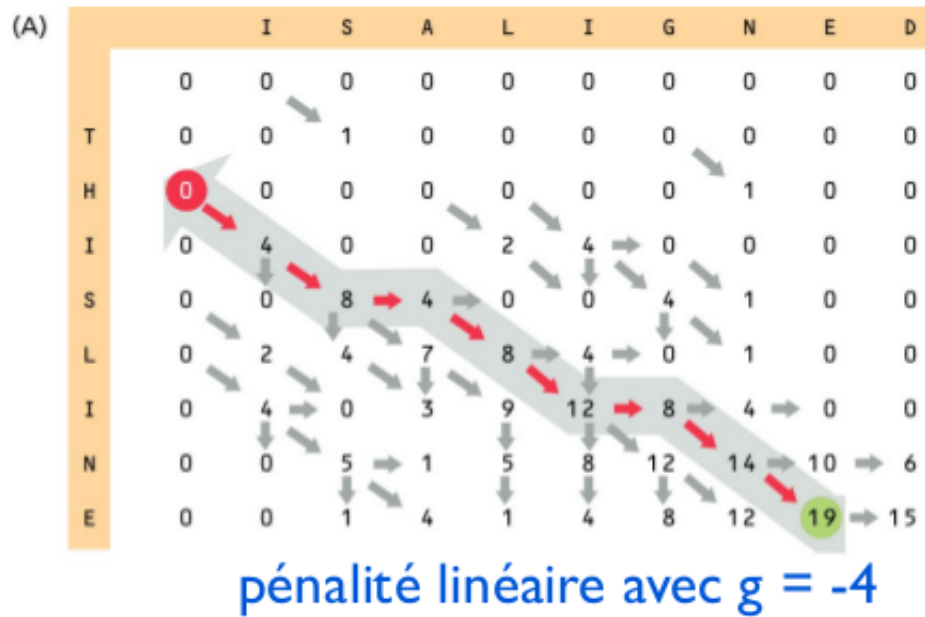
Matrice des scores - alignement local



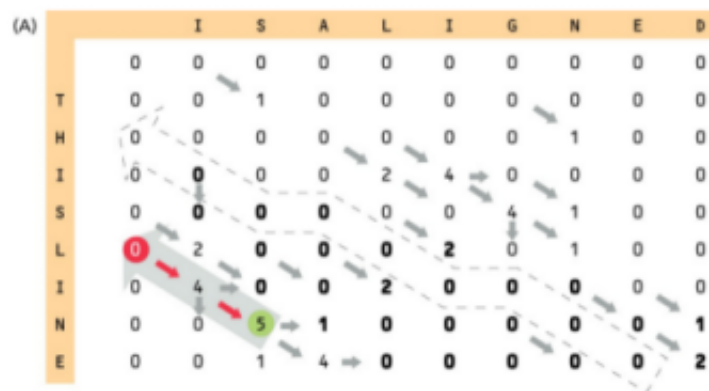
Matrice avec le chemin - alignement local

T<sub>-</sub>ER  
| |  
TNER

Alignement - alignement local



Alignement local optimal



Alignements sous-optimaux

### 5.3.1 Calcul efficace

On peut recalculer la matrice des scores beaucoup plus efficacement, si on sait déduire quand est-ce que des cases ne seront pas affectées par le changement du chemin à 0.

L'article original de Waterman-Eggert nous indique comment le faire:

Take the upper leftmost alignment position  $(i,j)$ .

For the single gap case, recompute each entry  $(k,j)$ ,  $k = i + l, i + 2, \dots$  until the new value  $H_{k,j}^*$  equals the previous  $H_{k,j}$ . Then since  $H_{l,j-1}^* = H_{l,j-1}$ , for all  $l$ , it is clear that  $H_{l,j}^* = H_{j,j}$  for all  $k < 1$ . A similar calculation for the row  $(i,k)$ ,  $j < k$ , allows us to stop whenever the new  $H_{i,k}^*$  equals  $H_{i,k}$ . We proceed by induction. We must go to at least the position that was necessary for the preceding row or column.

To implement our algorithm for the linear gap case, the same procedure is followed as for the single gap case, except that here we stop along a row or column when all three quantities  $H_{i,j}^*$ ,  $E_{i,j}^*$  and  $F_{i,j}^*$  are equal to  $H_{i,j}$ ,  $E_{i,j}$  and  $F_{i,j}$ , respectively.

En d'autres mots, on traite la matrice une ligne et colonne à la fois. On recalcule les valeurs de cette ligne/colonne tant que les nouvelles valeurs de  $S$ ,  $V$  et  $W$  sont différentes. Si dans la ligne/colonne précédente on a recalculé jusqu'à la position  $X$ , on doit aussi recalculer la ligne/colonne suivante jusqu'à cette position.

Par exemple, si on a calculé la première ligne jusqu'à la case avec coordonnée  $x = 5$ , alors on devra recalculer la deuxième ligne au moins aussi jusqu'à la case avec coordonnée  $x = 5$ .

## 5.4 Étape 1: sans alignements sous-optimaux

Nous allons d'abord commencer par implémenter une solution qui ne gère pas les alignements sous-optimaux. Dans la prochaine section, on s'occupera de mettre des valeurs à zéro, recalculer la matrice et trouver de nouveaux alignements (qui seront sous-optimaux).

Pour que cela ait un peu d'intérêt, on trouve tous les alignements locaux optimaux (plutôt que juste le premier).

```
In [19]: def findMaximumsInMatrix(matrix):
    maximum = []
    maxVal = float('-inf')
    for (y, line) in enumerate(matrix):
        for (x, value) in enumerate(line):
            if value > maxVal:
                maxVal = value
                maximums = [(x, y)]
            elif value == maxVal:
                maximums.append((x, y))
    return maximums

def findBestLocalPaths(s, previous, pathsToFind):
    # Trouve jusqu'à pathsToFind chemins locaux optimaux
    bestCoords = findMaximumsInMatrix(s)
    x, y = bestCoords[0]
```

```

score = s[y][x]

if score == 0:
    return [], float('-inf')

paths = []
pathsLeft = pathsToFind
for (x, y) in bestCoords:
    newPaths = getSolutionsFromArrowGrid(previous, pathsLeft, x, y, is

    paths.extend(newPaths)
    pathsLeft -= len(newPaths)
    if pathsLeft <= 0:
        break

return paths, score

def computeMatrix(s, v, w, sequence1, sequence2, previous, scoreSub, startGap, keepGap):
    # Calcule la matrice des scores
    height, width = sequence1.length() + 1, sequence2.length() + 1
    for x in range(1, width):
        for y in range(1, height):
            v[y][x] = max(s[y - 1][x] - startGap, v[y - 1][x] - keepGap)
            w[y][x] = max(s[y][x - 1] - startGap, w[y][x - 1] - keepGap)
            replacementCost = scoreSub.getCost(sequence1[y - 1], sequence2[y - 1])

            coords = [(x, y - 1), (x - 1, y), (x - 1, y - 1), None]
            costs = [v[y][x], w[y][x], s[y - 1][x - 1] + replacementCost, s[y][x]]

            s[y][x] = bestScore = max(costs)

            # Arrivé à un 0. Pas de flèches
            if bestScore == 0:
                previous[y][x] = [None]
                continue

            previous[y][x] = []
            for coord, cost in zip(coords, costs):
                if cost == bestScore:
                    previous[y][x].append(coord)

def getBestLocalAlignments(sequence1, sequence2, score, startGap=4, keepGap=4):
    """
    Finds the (up to k) best local alignments (without suboptimal alignments)
    l: the maximum number of alignment returned (note: less than l alignments)
    """

```

```

height, width = sequence1.length() + 1, sequence2.length() + 1
v, w, s = [makeMatrix(width, height) for _ in range(3)]
previous = makeMatrix(width, height, [])

pathsLeft = 1
computeMatrix(s, v, w, sequence1, sequence2, previous, score, startGap)

# We could improve the performance here, by either returning the paths
# making it a lazy operation
paths, bestScore = findBestLocalPaths(s, previous, pathsLeft)
pathsLeft -= len(paths)
pathScores = getPathScores(paths, s)

return s, paths, bestScore, previous, pathScores

```

Observons les résultats de cette fonctions lorsqu'on utilise avec les séquences du premier exemple ("writers" et "vintner")

```

In [20]: s1, s2 = a, b
         result = getBestLocalAlignements(s1, s2, blosum62, startGap=4, keepGap=1,
         printAlignementResult(s1, s2, result, blosum62, showScore = True, showLength=5)

         scoreMatrix, paths, score, arrowMatrix, pathScores = result

```

The best score is 11

Annotation: None  
Acids: WRITERS

Annotation: None  
Acids: VINTNER

Alignements found: 1 (may be artificially limited)

Alignement number 1 with score 11:

The length of the alignement is 5

Sequence 1: 4-6

Sequence 2: 4-7

T\_ER

| ||

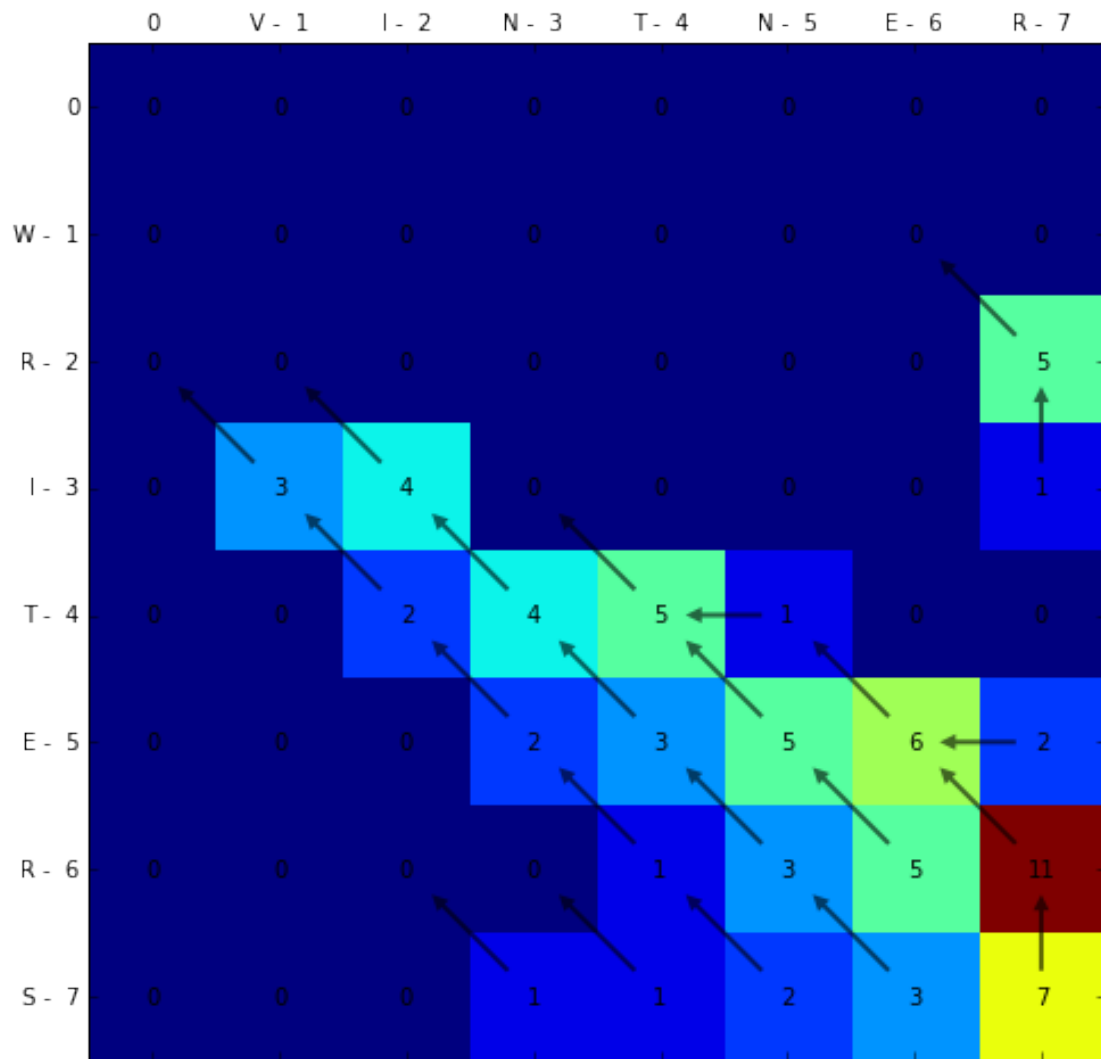
TNER

Visualisons la matrice des scores:

```

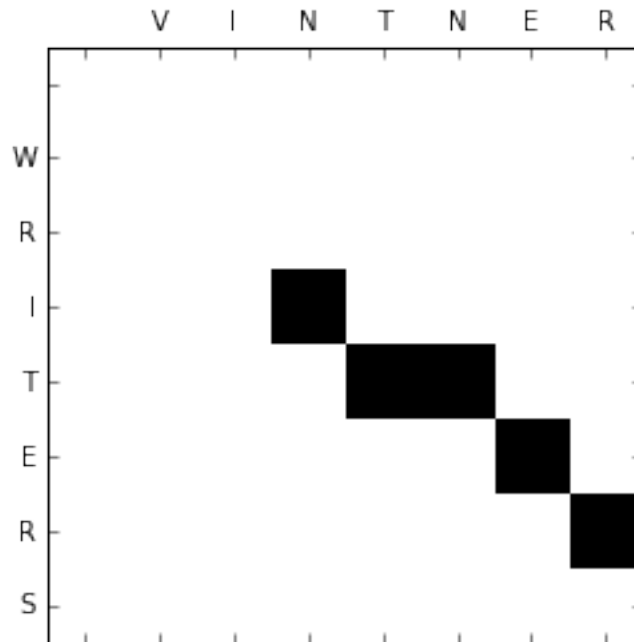
In [21]: printColoredMatrix(scoreMatrix, arrowMatrix, seq1=a, seq2=b, figsize=(8, 8))

```



L'algorithme fonctionne correctement. On peut voir qu'il commence de la case avec score maximum (7, 6) et crée le chemin à partir de là. Visualisons ces chemins:

```
In [22]: for path in paths:
          showPath(path, scoreMatrix, seq1=a, seq2=b)
          print(path)
```



```
[(3, 3), (4, 4), (5, 4), (6, 5), (7, 6)]
```

## 5.5 Étape 2: avec alignements sous-optimaux

```
In [23]: def findMatrixMaxPosAndScore(matrix):
    # Renvoie les coordonnées qui ont le meilleur score dans la matrice, a
    bestCoords = None
    maxVal = float('-inf')
    for (y, line) in enumerate(matrix):
        for (x, value) in enumerate(line):
            if value > maxVal:
                maxVal = value
                bestCoords = (x, y)

    return bestCoords, maxVal

def getBestLocalPath(s, previous):
    # Trouve un meilleur chemin dans la matrice des scores
    (x, y), score = findMatrixMaxPosAndScore(s)

    if score == 0:
        return None, float('-inf')
    else:
        paths = getSolutionsFromArrowGrid(previous, 1, x, y, isGlobal=False)
        path = paths[0]
```



```

        return path, score

def getStartCoords(path, width, height):
    # Renvoie les coordonnées de départ de ligne et de colonne
    # qui permettent d'efficacement recalculer la matrice
    coords = []
    firstX, firstY = path[0]
    lastX, lastY = path[-1]

    size = min((width - 1) - firstX, (height - 1) - firstY) + 1
    for i in range(size):
        coords.append((firstX + i, firstY + i))

    # No coordinate should be equal to 0, because we shouldn't modify the
    if firstX == 0 or firstY == 0:
        coords[0] = (max(1, firstX), max(1, firstY))

    return coords

def computeNewValue(x, y, sequence1, sequence2, s, v, w, previous, scoreSub):
    # Recalcule la valeur d'une cellule de la matrice des scores
    v[y][x] = max(s[y - 1][x] - startGap, v[y - 1][x] - keepGap)
    w[y][x] = max(s[y][x - 1] - startGap, w[y][x - 1] - keepGap)
    replacementCost = scoreSub.getCost(sequence1[y - 1], sequence2[x - 1])

    coords = [(x, y - 1), (x - 1, y), (x - 1, y - 1), None]
    costs = [v[y][x], w[y][x], s[y - 1][x - 1] + replacementCost, 0]

    s[y][x] = bestScore = max(costs)

    # Arrivé à un 0. Pas de flèches
    if bestScore == 0:
        previous[y][x] = [None]
        return

    previous[y][x] = []
    for coord, cost in zip(coords, costs):
        if cost == bestScore:
            previous[y][x].append(coord)

def clearPathsAndRecomputeMatrix(s, v, w, sequence1, sequence2, path, scoreSub):
    # Mettre les valeurs des chemins à 0
    for (x, y) in path:
        s[y][x] = 0
        v[y][x] = 0
        w[y][x] = 0

```

```

        previous[y][x] = [None]
        isInPath[y][x] = True

height, width = len(s), len(s[0])

# Recalculer la matrice efficacemement
prevMaxPosX, prevMaxPosY = float('-inf'), float('-inf')

for (startX, startY) in getStartCoords(path, width, height):
    lastChangedX, lastChangedY = startX, startY
    # Row
    for x in range(startX, width):
        if isInPath[startY][x]:
            continue

        oldV, oldW, oldS = v[startY][x], w[startY][x], s[startY][x]
        computeNewValue(x, startY, sequence1, sequence2, s, v, w, prevMaxPosX, prevMaxPosY)
        newV, newW, newS = v[startY][x], w[startY][x], s[startY][x]

        # Stop if new cells won't change
        if (oldV == newV and oldW == newW and oldS == newS):
            if x >= prevMaxPosX:
                break
            else:
                lastChangedX = x

prevMaxPosX = lastChangedX

# Column
for y in range(startY + 1, height):
    if isInPath[y][startX]:
        continue

    oldV, oldW, oldS = v[y][startX], w[y][startX], s[y][startX]
    computeNewValue(startX, y, sequence1, sequence2, s, v, w, prevMaxPosX, prevMaxPosY)
    newV, newW, newS = v[y][startX], w[y][startX], s[y][startX]

    # Stop if new cells won't change
    if (oldV == newV and oldW == newW and oldS == newS):
        if y >= prevMaxPosY:
            break
        else:
            lastChangedY = y

prevMaxPosY = lastChangedY

def getAllBestLocalAlignements(sequence1, sequence2, scoreSub, startGap=4,

```

```

"""
1: the maximum number of alignment returned (note: less than 1 aligns
"""
height, width = sequence1.length() + 1, sequence2.length() + 1
v, w, s = [makeMatrix(width, height) for _ in range(3)]
previous = makeMatrix(width, height, [])
isInPath = makeMatrix(width, height, False)
pathsLeft = 1
paths = []
pathScores = {}

for i in range(width):
    previous[0][i] = [None]

for i in range(height):
    previous[i][0] = [None]

# Find best paths
computeMatrix(s, v, w, sequence1, sequence2, previous, scoreSub, start)

newPath, bestScore = getBestLocalPath(s, previous)
pathScores[newPath[-1]] = bestScore
paths.append(newPath)
pathsLeft -= 1

if showMatrix:
    printColoredMatrix(s, previous, seq1=sequence1, seq2=sequence2)

# If we don't find enough paths, keep searching for sub-optimal paths
while pathsLeft > 0:
    clearPathsAndRecomputeMatrix(s, v, w, sequence1, sequence2, newPath)

    if showMatrix:
        printColoredMatrix(s, previous, seq1=sequence1, seq2=sequence2)

    newPath, bestCurrentScore = getBestLocalPath(s, previous)

    # No more good paths
    if newPath is None:
        print("Stopped searching because no good paths")
        break

    pathsLeft -= 1
    paths.append(newPath)
    pathScores[newPath[-1]] = bestCurrentScore

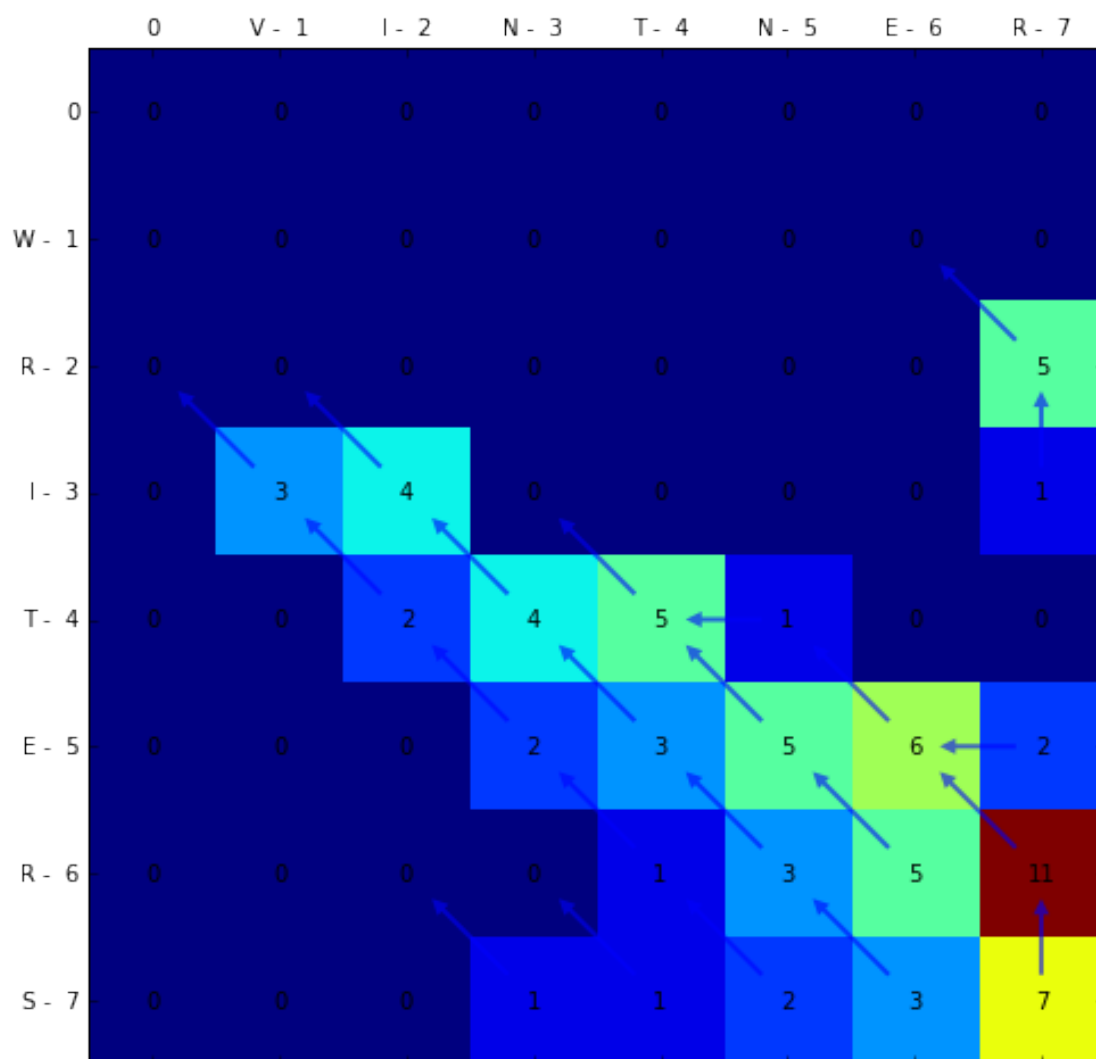
return s, paths, bestScore, previous, pathScores

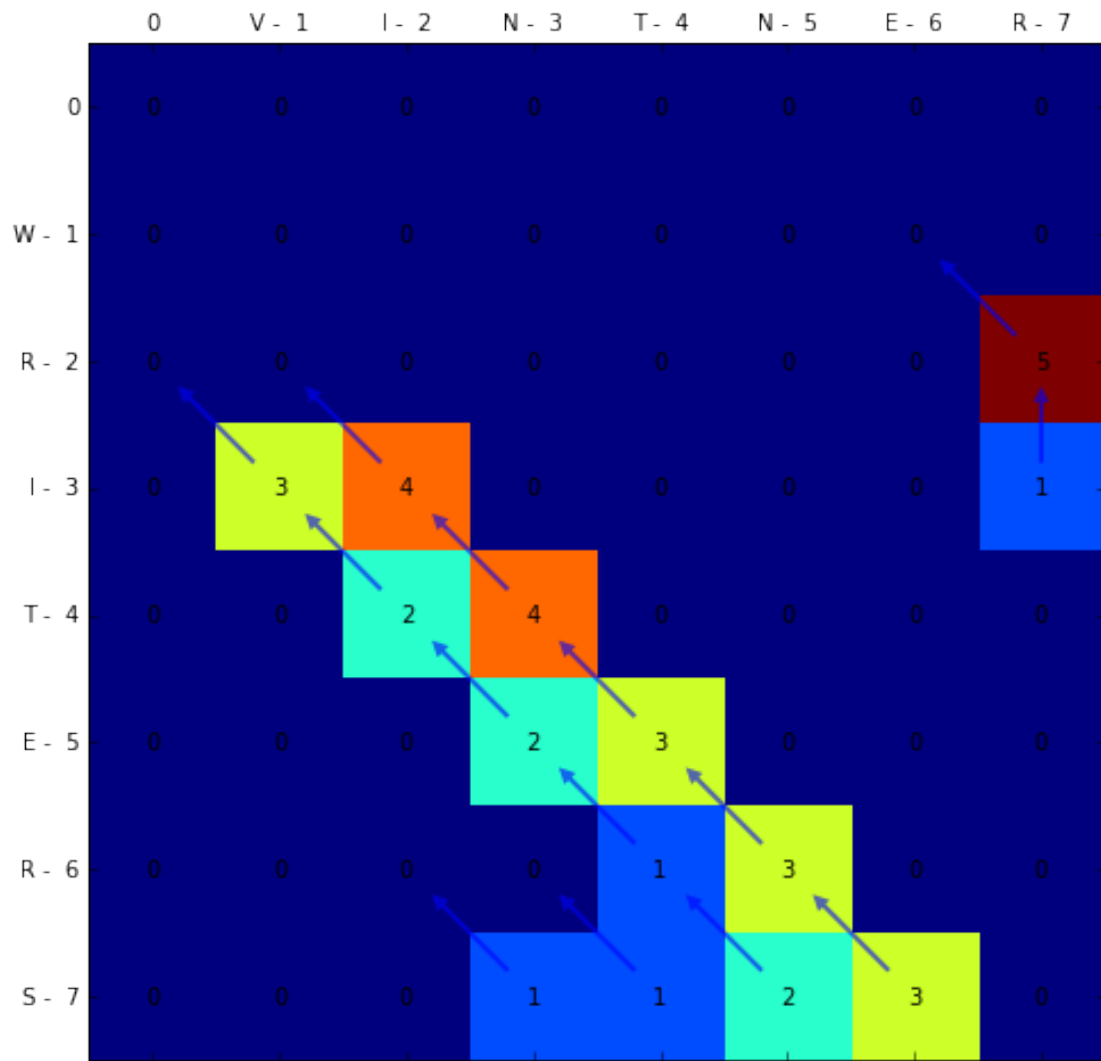
```

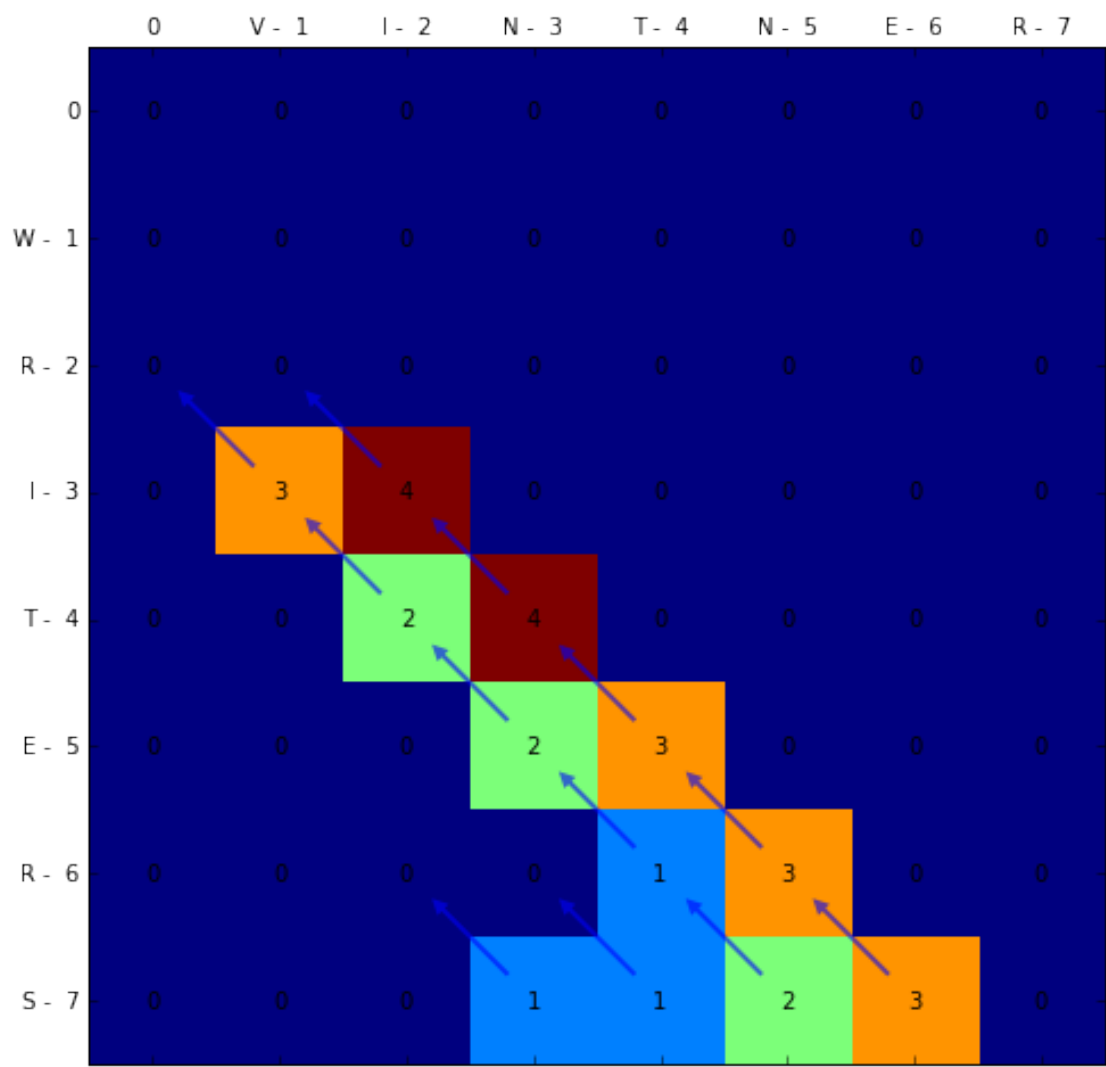
Appliquons cette fonction à notre premier exemple ("writers" et "vintner"). Pour mieux comprendre l'évolution de la matrice des scores, elle est aussi affichée chaque fois qu'elle est recalculée.

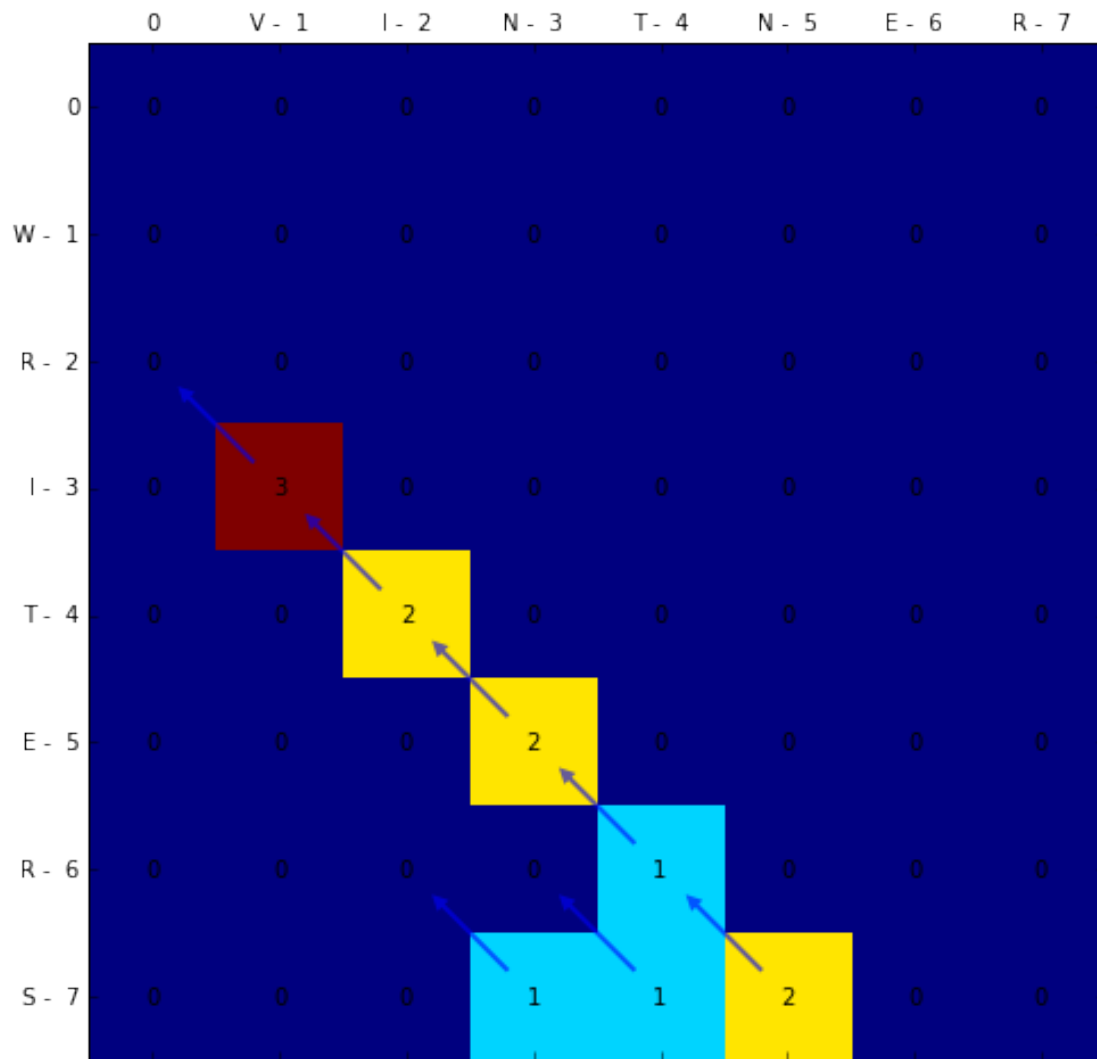
```
In [24]: s1, s2 = a, b
         #s1, s2 = ww1, ww2
         result = getAllBestLocalAlignements(s1, s2, blosum62, startGap=4, keepGap=
         printAlignementResult(s1, s2, result, blosum62, showScore = True, showLeng

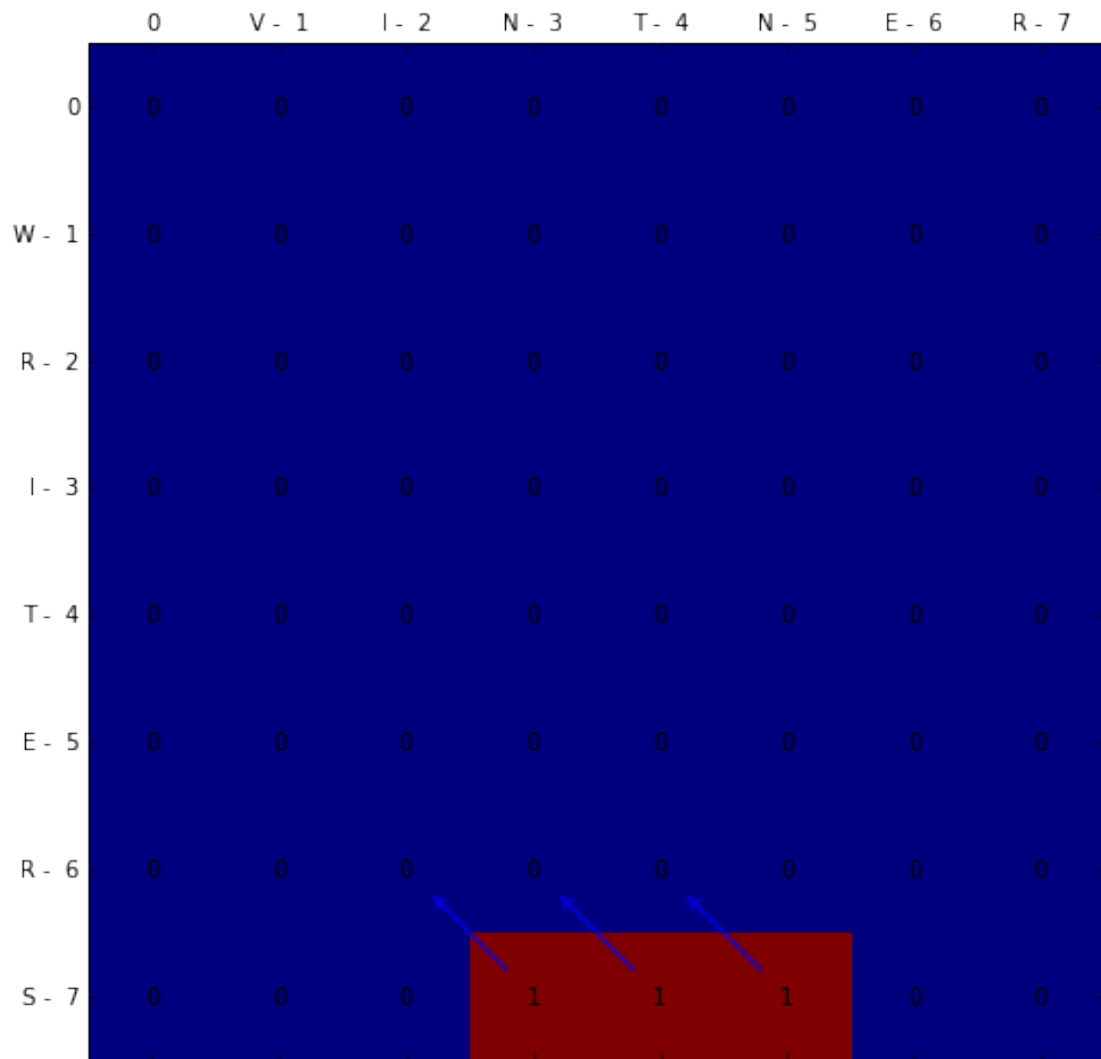
         scoreMatrix, paths, score, arrowMatrix, pathScores = result
```





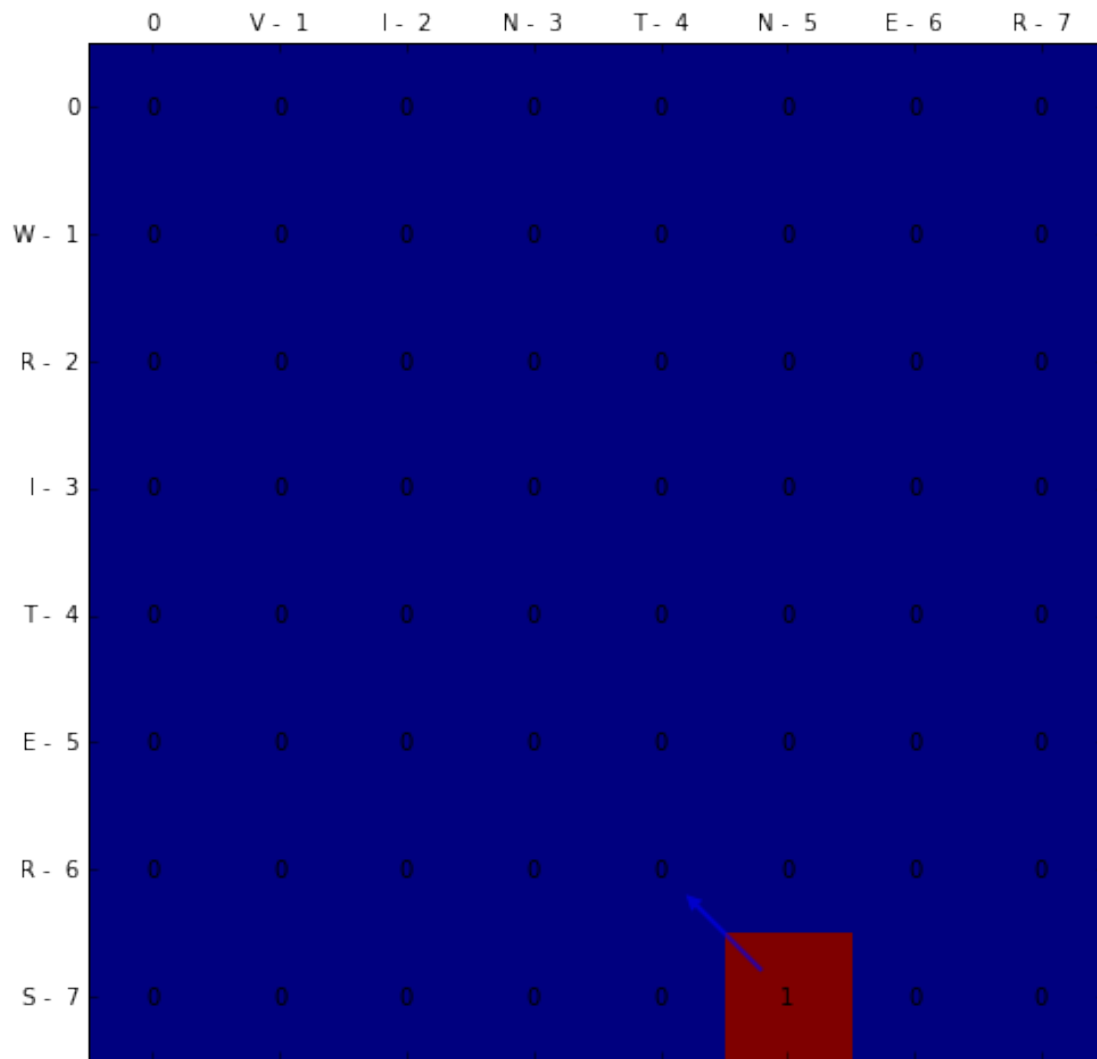








	0	V - 1	I - 2	N - 3	T - 4	N - 5	E - 6	R - 7
0	0	0	0	0	0	0	0	0
W - 1	0	0	0	0	0	0	0	0
R - 2	0	0	0	0	0	0	0	0
I - 3	0	0	0	0	0	0	0	0
T - 4	0	0	0	0	0	0	0	0
E - 5	0	0	0	0	0	0	0	0
R - 6	0	0	0	0	0	0	0	0
S - 7	0	0	0	0	1	1	0	0



	0	V - 1	I - 2	N - 3	T - 4	N - 5	E - 6	R - 7
0	0	0	0	0	0	0	0	0
W - 1	0	0	0	0	0	0	0	0
R - 2	0	0	0	0	0	0	0	0
I - 3	0	0	0	0	0	0	0	0
T - 4	0	0	0	0	0	0	0	0
E - 5	0	0	0	0	0	0	0	0
R - 6	0	0	0	0	0	0	0	0
S - 7	0	0	0	0	0	0	0	0

Stopped searching because no good paths

The best score is 11

Annotation: None

Acids: WRITERS

Annotation: None

Acids: VINTNER

Alignements found: 7 (may be artificially limited)

Alignment number 1 with score 11:

Sequence 1: 4-6

Sequence 2: 4-7

T\_ER

| ||

TNER

Alignement number 2 with score 5:

Sequence 1: 2-2

Sequence 2: 7-7

R

|

R

Alignement number 3 with score 4:

Sequence 1: 3-3

Sequence 2: 2-2

I

|

I

Alignement number 4 with score 3:

Sequence 1: 3-3

Sequence 2: 1-1

I

:

V

Alignement number 5 with score 1:

Sequence 1: 7-7

Sequence 2: 3-3

S

.

N

Alignement number 6 with score 1:

Sequence 1: 7-7

Sequence 2: 4-4

S

.

T

Alignement number 7 with score 1:

Sequence 1: 7-7

Sequence 2: 5-5

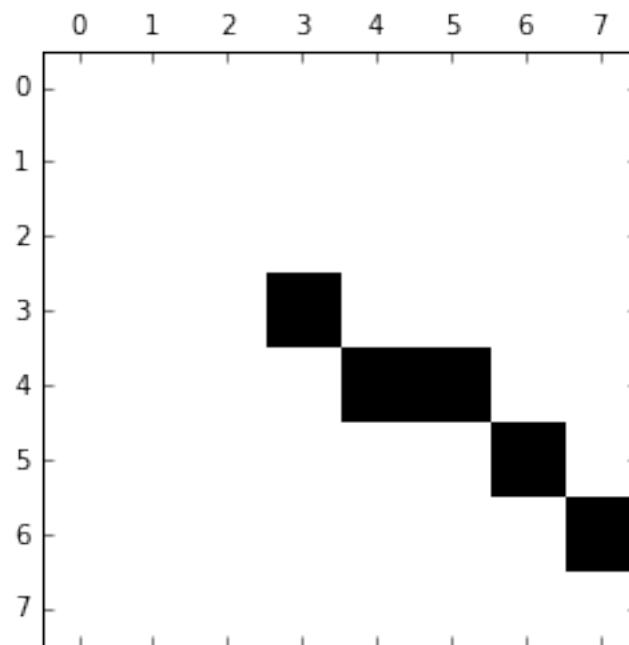
S

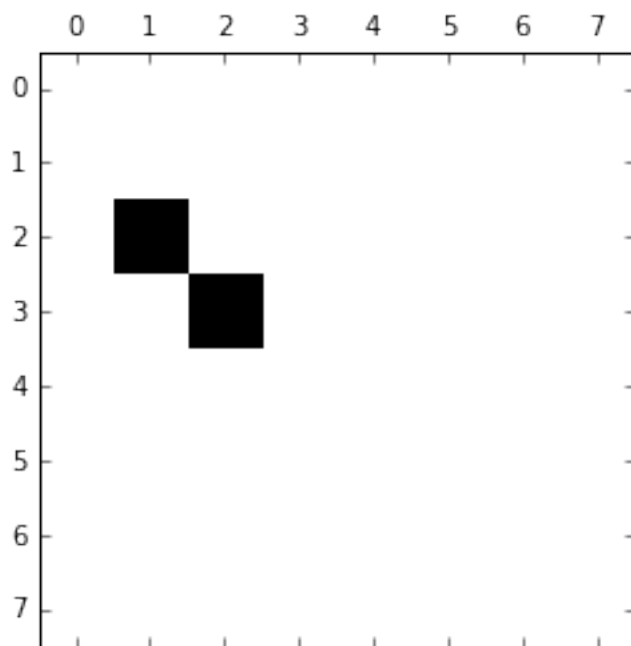
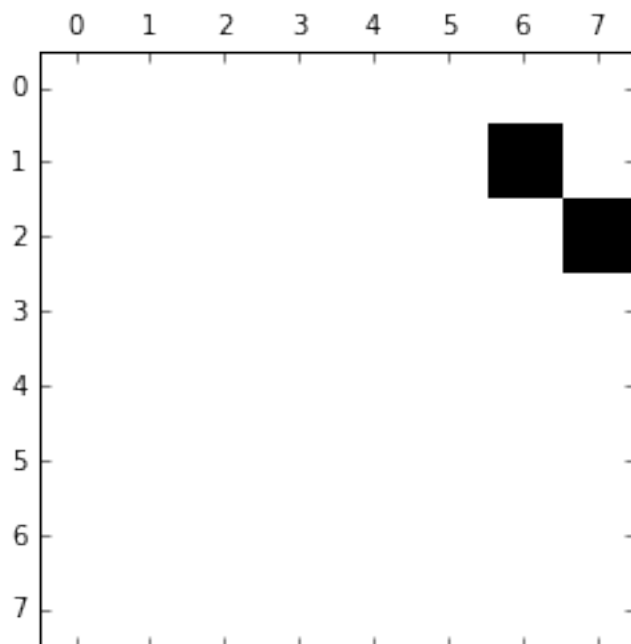
.

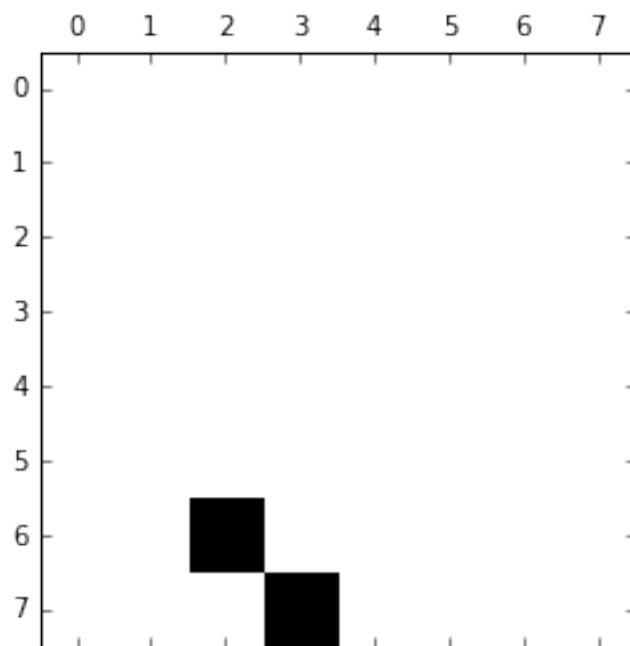
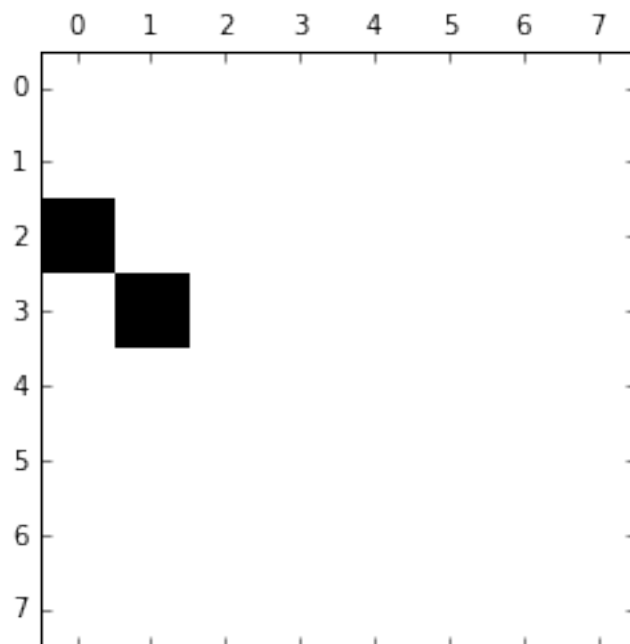
N

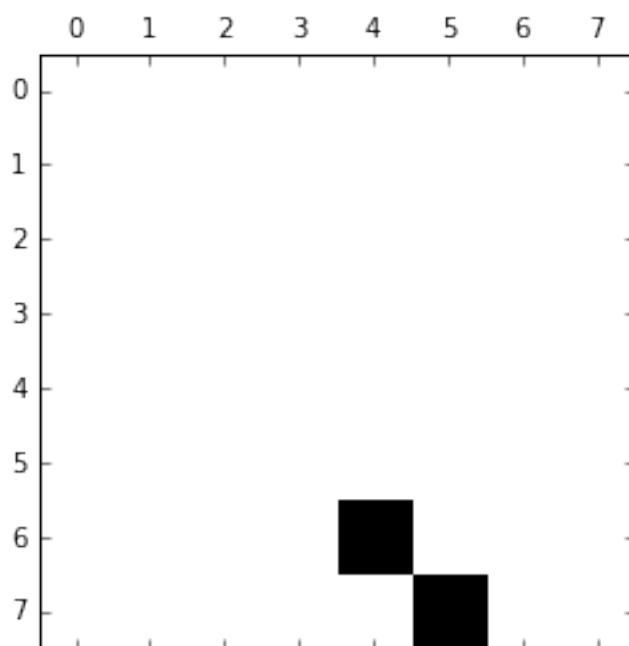
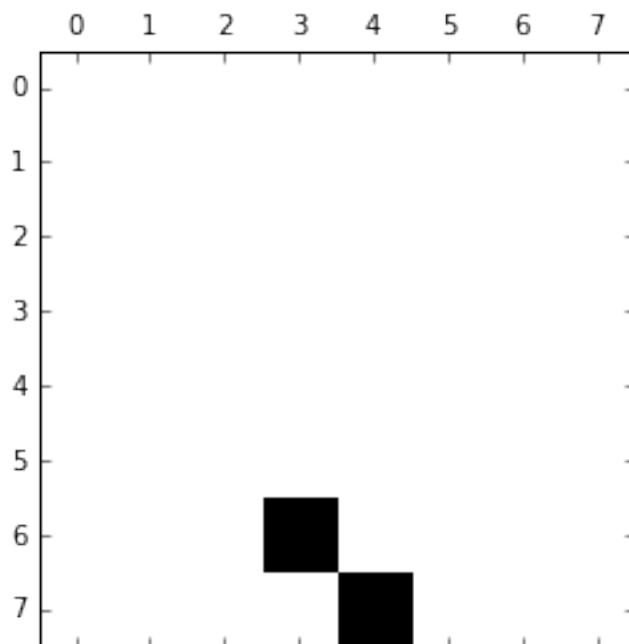
Visualisons les chemins créés:

```
In [26]: for path in paths:
          showPath(path, scoreMatrix)
```









Appliquons maintenant cet algorithme aux 2 premières séquences du fichier *protein-sequences.fasta*.

```
In [27]: p1, p2 = proteinSequences[0], proteinSequences[1]
          #p1, p2 = ww1, ww2
```



```

result = getAllBestLocalAlignments(p1, p2, blosum62, startGap=4, keepGap=
scoreMatrix, paths, score, arrowMatrix, pathScores = result
printAlignementResult(p1, p2, result, blosum62, showScore = True, showLeng

```

The best score is 435

Annotation: sp|P46937|YAP1\_HUMAN Transcriptional coactivator YAP1 OS=Homo sapiens C  
Acids: MDPGQQPPPPQPAPQGQGQPPSQPPQGQGPPSGPGQPAPAATQAAPQAPPAGHQIVHVRGDSETDLEALFNAVMPNPF

Annotation: sp|P46934|NEDD4\_HUMAN E3 ubiquitin-protein ligase NEDD4 OS=Homo sapiens  
Acids: MAQSLRLHFAARRSNTYPLSETSGDDLDSHVHMCFKRPTTRISTSNVVQMKLTPRQTALAPLIKENVQSQERSVPS

Alignements found: 2 (may be artificially limited)

Alignment number 1 with score 435:

The length of the alignment is 616

Sequence 1: 1-504

Sequence 2: 587-1107

```

MDPG___QQPPPQPA_P_QGQGQPPSQ_PPQG___Q___GPP___SGPGQ___PAPA___ATQA___A___P_QAPPA_
..||   .||   ..|   | | |.||. || |   |   |   |   |   |.|.   .|.|   .   . ||. |
LEPGWVLDQP___DAACHLQQQ_QEPSPLPP_GWEERQDILGRTYVYNHESRRTQWKRPTPQDNLTDAENGNIQLQAQRAF

___GHQI___VHVRGDSET___DLEA_LF_N_AVMNPK_TAN___VP___QTVPMRLRKLPDSFFKPPEPKS_HS
. ||   |. | .||.   | || .: | |. .|. .| ||   ... .||. . | |...|...| ||
TTRRQISEETESVDNRESENWEIRED_EATMYSNQAFSPPPSSNLDVPTHAEELNARLTIFGNSAVSQPASSNHS

___R___QAST___DA___GT_AGALTP___QHVR___A___HSS___PASLQLGAVSPGTLTPTGVVSGP_
|   || |   ..   |. | |. |   |. |   .   |. |   |   .|. |...|...|..   .||
SRRGSLQAYTFEEQPTLPVLLPTSSG_LPPGWEEKQDERGRSYYVDHNSRTTTWTKP_TVQ_ATVETSQLTSSQSSAGPQ

___AATP_TAQHLRQSSFEIPDDVPLPAGWEMAKTSSGQRYFLNHIDQTTTWQDPR_KAMLSQMNVTAPTSPFPVQQNMMS
|. |. .|. |...| | |... | |. || |...|. : |...| .|. || |. || |   |   . ...   .||   ... |.
SQASTSDSGQQVTQPS_EIEQGF_LPKGWEVRHAPNGRPFFIDHNTKTTTWEDPRLK_IPAHLR___GKTS___LDTS___ND

ASGPLPDGWEQAMTQ_DGEIYYINHKN_KTTSWLDPRLDPRFAMNQRISSAPVKQPPPLAPQSPQGGVMGGSNSNQQQQ
. ||||. |||. |. ||. |: || | | |. |   |   :   | :. |. .|. |   .. ...
L_GPLPPGWEER_THTDGRIFYINH_NIKRTQWEDPRLE___N___V___A_ITGPA_VPYS___RDYKRKYE

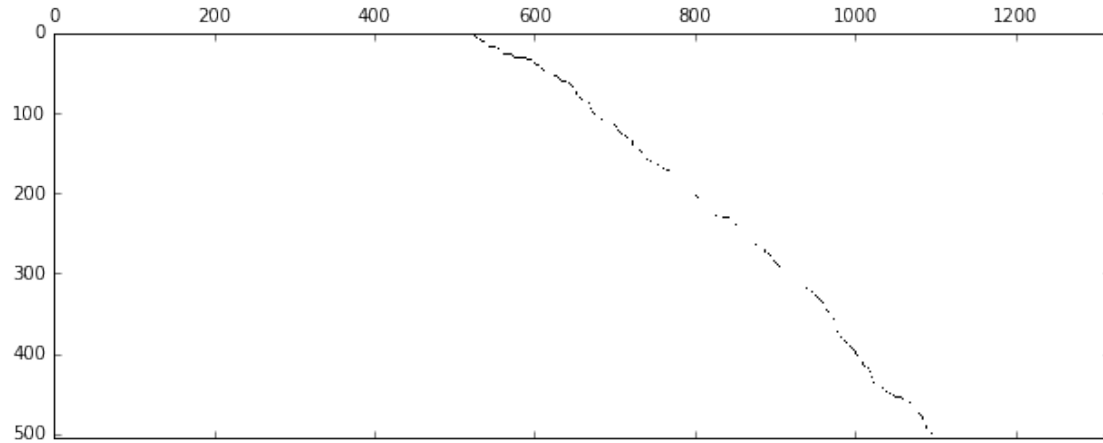
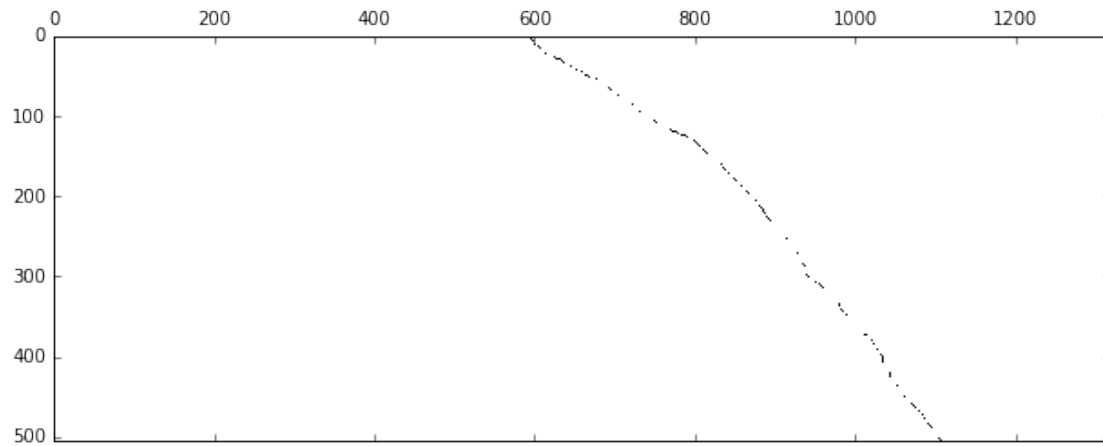
M___R_LQ_QLQME_K_ERLRLKQOELLRQAMRNINPSTANSPKCQELALRSQLPTLEQDGGTQNPVSSPGMSQE___L_
. | |. | ... | | ..|. |...|. |   . . | ... |...| .| ||   .. .. |...|   .
FFRRKLKKQNDIPNKFE_MKLRRATVLEDSYRRI___M_GVKRADF_LKARL_WIEFDG_EKGLDYGGVAREWFFLIS

RTMTTNSSDPFLNSGT_Y_HSRDESTDSGLSMSSYSVPRTPDFFLNSVDEMDTGDTINQSTLPSQQNRFPDYLEAIPGTNV
..| |   |: . |.: .|   .||.   |... .|   ||   | ..|. | |. |. :. :. |. |
KEMF_N___PYY___GLFEYS___ATDN___YTLQINP___NS___G_LCNEDHL_SY___FK_FIGRVAGMAV

DLGTLEGDGMNIEG_EELMPSLQEALSSDILNDMESVLAATKLDKESF___LTWL

```





## 5.6 Trouver le meilleur alignement local

In [29]: `import itertools`

```
def getBestLocalMatches(sequences, subMatrix=blosum62, startGap=4, keepGap=4):
    bestScore = float('-inf')
    bestResults = {}

    for seq1, seq2 in itertools.combinations(sequences, 2):
        result = getAllBestLocalAlignements(seq1, seq2, subMatrix, startGap, keepGap)
        scoreMatrix, paths, score, arrowMatrix, pathScores = result
        if score > bestScore:
            bestScore = score
            bestResults = {(seq1, seq2): result}
        elif score == bestScore:
```



```

Waterman-Eggert score: 190; 79.8 bits; E(1) < 6.5e-19
41.3% identity (69.6% similar) in 92 aa overlap (173-264:842-926)

      180      190      200      210      220      230
P46937 LPAGWEMAKTSSGQRYFLNHIDQTTTWQDPRKAMLSQMNVTAPTSPPVQNNMMNSASGPL
      :: :::: . .: .::: .::: .::: .::: .::: .::: .::: .:::
P46934 LPKGWEVRHAPNGRPFFIDHNTKTTTWEDPRLKIPAHLR--GKTSLDTSNDL-----GPL
              850      860      870      880      890

      240      250      260
P46937 PDGWEQAMTQDGEIYYINHKNKTTSWLDPRLD
      : :::: .::: .::: .::: .::: .:::
P46934 PPGWEERTHTDGRIFYINHNIKRTQWEDPRLE
              900      910      920

```

### Alignement local optimal - LALIGN

On peut donc voir que l'alignement obtenu est le même et que le score est très proche:

- Score: 194 (notre résultat) / 190 (LALIGN)
- Morceau choisi de la séquence 1: 173-264 (notre résultat) / 173-264 (LALIGN)
- Morceau choisi de la séquence 2: 842-926 (notre résultat) / 842-926 (LALIGN)

Cela semble donc indiquer que notre implementation est correcte.

## 5.8 Étude des morceaux et Uniprot

Quand on consulte Uniport pour la [séquence 1](#) et la [séquence 2](#), on peut voir que l'alignement est entre:

- Séquence 1 (173-264): un morceau qui contient le domaine WW1 et WW2

Feature key	Position(s)	Description	Actions	Graphical view	Length
Domain <sup>i</sup>	171 - 204	WW 1 <a href="#">PROSITE-ProRule annotation</a>	<a href="#">Add</a> <a href="#">BLAST</a>		34
Domain <sup>i</sup>	230 - 263	WW 2 <a href="#">PROSITE-ProRule annotation</a>	<a href="#">Add</a> <a href="#">BLAST</a>		34

### Domaines séquence 1

- Séquence 2 (842-926): un morceau qui contient le domaine WW3 et WW4

Feature key	Position(s)	Description	Actions	Graphical view	Length
Domain <sup>i</sup>	610 - 643	WW 1 <a href="#">PROSITE-ProRule annotation</a>	<a href="#">Add</a> <a href="#">BLAST</a>		34
Domain <sup>i</sup>	767 - 800	WW 2 <a href="#">PROSITE-ProRule annotation</a>	<a href="#">Add</a> <a href="#">BLAST</a>		34
Domain <sup>i</sup>	840 - 873	WW 3 <a href="#">PROSITE-ProRule annotation</a>	<a href="#">Add</a> <a href="#">BLAST</a>		34
Domain <sup>i</sup>	892 - 925	WW 4 <a href="#">PROSITE-ProRule annotation</a>	<a href="#">Add</a> <a href="#">BLAST</a>		34
Domain <sup>i</sup>	984 - 1318	HECT <a href="#">PROSITE-ProRule annotation</a>	<a href="#">Add</a> <a href="#">BLAST</a>		335

### Domaines séquence 2

Vu que les 2 morceaux contiennent deux domaines du type WW à courte distance l'un de l'autre (environ 25 acides aminés pour les 2 morceaux), il est donc logique que l'alignement local soit très bon.

Comme le morceau de la séquence 1 contient le domaine WW1 et WW2, on pourrait se dire que le morceau de la séquence 2 devrait contenir WW1 et WW2, plutôt que WW3 et WW4, pour améliorer l'alignement. Ce n'est pas le cas parce qu'on pénalise fortement les espaces, et qu'il y a un grand espace entre le domaine WW1 et WW2 dans la séquence 2 (130 acides aminés).

Dans le morceau 1, WW1 et WW2 sont espacés de 25 acides aminés et dans le morceau 2 WW3 et WW4 sont espacés de 18 acides aminés. Comme les espaces sont courts et en quantité similaire pour les 2 morceaux, l'alignement local est donc très bon.

## 6 Conclusion

Dans ce travail, nous avons implémenté les algorithmes d'alignement global et local. Ils nous permettent de comparer des séquences et de trouver le meilleur alignement parmi une grande quantité de séquences. Ainsi, on peut découvrir quels morceaux d'acides aminés sont les plus similaires entre eux.

Après avoir implémenté ces algorithmes, on voit que nos résultats sont proches de LALIGN et qu'ils sont cohérents avec les informations contenues dans Uniprot. Cela nous permet donc d'avoir confiance dans la solidité de notre implémentation.

## 7 Amélioration possibles

Une amélioration possible pour l'alignement local est de prendre en compte non seulement le score, mais aussi la longueur de l'alignement.

## 8 Notes d'implémentation

Pour des raisons de clarté, j'ai implémenté le code via une série de fonctions. On peut rapidement voir que ceci n'est pas optimal, car certaines fonctions en appellent d'autres en leur passant beaucoup de paramètres. D'autre part, certaines fonctions sont des très proches variantes d'autres fonctions.

Ceci suggère que définir des classes (probablement avec de l'héritage) permettrait de simplifier le code et éviter des répétitions. Malheureusement, une classe doit être définie en une seule cellule Jupyter, et ce n'est donc pas possible d'intercaler du texte en Markdown et les définitions de méthodes de ces classes. Pour des raisons de clarté, je ne l'ai donc pas fait, mais dans du code réel je le ferai.