به نام خدا

پروژه پردازش زبان

سید شایان دانشور

9726523

استاد درس: دکتر دوست

تیر 1400

سوال اول) روش ارایه شده در درس برای محاسبه مینیمم فاصله ویرایشی، روش Levenshtein Distance است که در حالت بهتر آن هزینه های حذف و اضافه کردن کاراکتر 1 و هزینه جایگزینی 2 می باشد.

توضیح الگوریتم: این یک الگوریتم از شاخه الگوریتم های برنامه ریزی پویا یا Dynamic Programming است. به این صورت عمل می کند که با در نظر گرفتن زیر مسئله هایی از سوال، سعی میکند سوال را حل کند.

پس این الگوریتم دو رشته ورودی X و Y با طول های n و m به عنوان ورودی گرفته و ماتریسی با اندازه (n+1) در (m+1) و در ابتدا فرض میکند، رشته X که رشته مبدا باشد و نیز رشته Y که رشته مقصد است، خالی باشد، هزینه حساب می شود که مسلما برابر صفر است، سپس در هر مرحله طول رشته مقصد یکی زیاد شده و هزینه حساب می شود و در سطر اول ماتریس نوشته می شود (جلوتر در شکل نشان داده شده است) در حال حاظر هزینه برای تبدیل رشته خالی به هر طولی از رشته مقصد را داریم، حال در سطر بعدی طول رشته مبدا را 1 گرفته (کاراکتر اول) و از طول 0 برای رشته مقصد شروع میکنیم، حال با توجه به اینکه محاسبه هزینه کمینه کمی مشکل می شود از دستور برنامه ریزی پویا برای این الگوریتم استفاده میکنیم که ببینیم رشته جدید با حذف یا اضافه و یا جایگزینی بهترین نتیجه را میگیرد، همین روال را برای همه سطر های ماتریس می رویم در نهایت کمترین هزینه در خانه آخر ماتریس (پایین چپ) تولید می شود.

برای اینکه بفهمیم چه عملیاتی در هر مرحله باید انجام دهیم، باید ببینیم که در هر خانه ماتریس، آن خانه چطوری تولید شده است (ناشی از حذف یا اضافه و یا جایگزینی بوده است)، و از روی آن از انتهای رشته شروع به تغییر دادن رشته می کنیم تا رشته بدست آید.

رابطه برنامه ریزی پویا برای این مسئله:

$$D[i,j] = \min \begin{cases} D[i-1,j] + \text{del-cost}(source[i]) \\ D[i,j-1] + \text{ins-cost}(target[j]) \\ D[i-1,j-1] + \text{sub-cost}(source[i], target[j]) \end{cases}$$

توضیح رابطه: این به این معنی است که هنگام پر کردن هر خانه (غیر از سطر و ستون اول که بدیهی است چون در یک طرف رشته خالی داریم و هزینه اش همان تعداد کاراکتر ها تا آن جا می شود)، باید به خانه های بالا و چپ و بالا چپ آن خانه نگاه کنیم اگر به بالا چپ نگاه کنیم باید ببینیم که آیا دو کاراکتر فعلی در رشته مبدا و مقصد یکسان اند یا خیر، اگر نبودند همان هزینه خانه ی بالا چپ را در نظر میگیریم و اگر متفاوت بودند همان هزینه به علاوه هزینه جایگزینی (که 2 گرفتیم) را در نظر میگیرم، سپس به خانه بالایی نگاه میکنیم و مقدار آن به علاوه هزینه حذف را در نظر گرفته و سپس به خانه راست نگاه میکنیم و مقدار آن را به علاوه هزینه اضافه کردن میکنیم، هر کدام ازین سه مقدار کمتر بود، باید آن را در ماتریس قرار دهیم که وقتی چنین کاری انجام می دهیم، در واقع مشخص می کنیم که در آن لحظه، بهترین و کم هزینه ترین عمل برای تغییر چه عملی است. واضح است که جهت را نیز می توان بعدا با دیدن جدول طبق همین رابطه بدست آورد.

سوال دوم) فایل پروژه پیوست شده است، با توجه به اینکه کلیت الگوریتم توضیح داده شده، توابع مهم کد را توضیح میدهیم:

کلاس Direction: از این کلاس (enum) برای مشخص کردن جهت وکاراکتر استفاده می شود.

```python
class Direction(enum.Enum):
    Left = 1
    Up = 2
    UpLeft = 3

    @staticmethod
    def get_char(dir_value: int) -> str:
        if dir_value == Direction.Left.value:
            return '←'
        elif dir_value == Direction.Up.value:
            return '↑'
        elif dir_value == Direction.UpLeft.value:
            return '↖'
        else:
            return ''
```

کلاس MinimumEditDistance : منطق الگوریتم و متد های کمکی دیگر همه در این کلاس پیاده سازی شده اند.

```
class MinimumEditDistance:
    def __init__(self, x: str, y: str, substitution_cost: int = 2,
                 del_cost: int = 1, insertion_cost: int = 1) -> None:
        super().__init__()
        self.y = y
        self.x = x
        self.ins_cost = insertion_cost
        self.del_cost = del_cost
        self.sub_cost = substitution_cost
        self.rows = 1 + len(x)
        self.cols = 1 + len(y)
        self.distance_matrix = np.zeros((self.rows, self.cols))
        self.direction_matrix = np.zeros((self.rows, self.cols, 3))
        self.__calculate_distance()
        self.__calculate_direction()
```

برای محاسبه فاصله کمینه، برنامه به صورت داینامیک روی هزینه های خذف و اضافه و جایگزینی تعریف شده است، به همین دلیل در ورودی سازنده این هزینه ها در کنار رشته مبدا و مقصد دریافت می شوند. پس از دریافت ورودی ها، طول و عرض ماتریس محاسبه شده و ماتریس فاصله و ماتریس جهت ساخته می شود. (چون نمی توان همزمان هم جهت را داشت هم مقدار، مگر با پچیده شدن)، در انتها نیز دو متد calculate_distance و calculate_direction صدا زده می شوند.

متد اول برای محاسبه هزنیه های کمینه و متد دوم برای در آوردن جهت ها می باشد.

جلوتر این دو متد و سایر متد ها توضیح داده می شوند.

متد calculate_distance :

```
def __calculate_distance(self) -> None:
    """
        calculates Minimum Edit Distance of X to Y
        sub,del and ins costs are set in the constructor
    """
    for i in range(0, self.rows):
        self.distance_matrix[i, 0] = i

    for j in range(0, self.cols):
        self.distance_matrix[0, j] = j

    for i in range(1, self.rows):
        for j in range(1, self.cols):
            sub_cost = self.sub_cost
            if self.x[i - 1] == self.y[j - 1]:
                sub_cost = 0
            self.distance_matrix[i, j] = min(
                self.distance_matrix[i - 1, j] + self.del_cost,
                self.distance_matrix[i, j - 1] + self.ins_cost,
                self.distance_matrix[i - 1][j - 1] + sub_cost)
```

در این متد، ابتدا ستون اول و سطر اول که هزینه مشخصی دارند (طول کاراکتر تا آن نقطه) ابتدا در ماتریس پر می شوند، سپس طبق همان فرمولی که پیشتر در سوال 1 گفته شد، وزن ها در آورده می شوند، حالت جایگزینی دو کاراکتر مشابه نیز در نظر گرفته شده و حواسمان به تشابه بوده است. (در if که دیده می شود هندل شده است)

متد کمکی get_min_dist : خانه آخر پایین چپ را که همان هزینه کمینه است را بر میگرداند.

```
def get_min_dist(self):
    return self.distance_matrix[self.rows - 1, self.cols - 1]
```

متد calculate_direction:

```
def __calculate_direction(self) -> None:
    for i in range(0, self.rows):
        index = Direction.Up.value - 1
        self.direction_matrix[i, 0, index] = Direction.Up.value

    for j in range(0, self.cols):
        index = Direction.Left.value - 1
        self.direction_matrix[0, j, index] = Direction.Left.value

    for i in range(1, self.rows):
        for j in range(1, self.cols):
            dist = self.distance_matrix[i][j]
            if self.distance_matrix[i - 1, j] + self.del_cost == dist:
                self.direction_matrix[
                    i, j, Direction.Up.value - 1] = Direction.Up.value
            if self.distance_matrix[i, j - 1] + self.ins_cost == dist:
                self.direction_matrix[
                    i, j, Direction.Left.value - 1] = Direction.Left.value
            if (self.distance_matrix[i - 1, j - 1] == dist or self
                    .distance_matrix[i - 1, j - 1] + self.sub_cost == dist):
                self.direction_matrix[i, j, Direction.UpLeft.value - 1] = \
                    Direction.UpLeft.value
```

در این متد، همه جهت های ممکن در آورده می شود و به یک جهت بسنده نمی شود، به این صورت که در ابتدا جهت ستون اول (همگی به سمت بالا) و جهت سطر اول(همگی به سمت چپ) اند در ماتریس در نظر گرفته شده قرار داده می شود (در این ماتریس همه جهت ها در ابتدا با

صفر نمایش داده می شوند، به این معنی که هیچ جهتی وجود ندارد، عنصر اول یا 0 است یا 1، اگر 1 بود، یعنی جهت رو به چپ است، عنصر دوم یا 0 است یا 2 اگر 2 بود یعنی جهت رو به بالاست و در نهایت عنصر سوم یا 0 است یا 3 که اگر 3 بود یعنی جهت رو به بالاچپ وجود دارد).

سپس برای در آوردن جهت کافی است ببینیم در هر مرحله با اضافه کردن کدام یک از مقادیر رابطه پویا، به مقدار فعلی میرسیم، ممکن است از چندین راه برسیم، به همین دلیل همه را در این قسمت در نظر میگیریم البته که در نهایت فقط یک روند را به عنوان نمونه نشان می دهیم هر چند هزینه بهینه در هر صورت یکسان است.

متد get_instructions_raw: این متد، تنها مقادیر عددی جهت ها را به کاراکتر جهت تبدیل می کند و در متد دیگری استفاده شده است.

```
def get_instructions_raw(self):
    result = []
    i = self.rows - 1
    j = self.cols - 1
    while not (i == 0 and j == 0):
        dir_left = self.direction_matrix[i, j, Direction.Left.value - 1]
        dir_up = self.direction_matrix[i, j, Direction.Up.value - 1]
        dir_ul = self.direction_matrix[i, j, Direction.UpLeft.value - 1]
        if dir_left != 0:
            result.append(Direction.get_char(dir_left))
            j -= 1
        elif dir_up != 0:
            result.append(Direction.get_char(dir_up))
            i -= 1
        elif dir_ul != 0:
            result.append(Direction.get_char(dir_ul))
            i -= 1
            j -= 1

    return result
```

متد get_instrucitons

```
def get_instructions(self) -> list:
    results = []
    i = self.rows - 1
    j = self.cols - 1
    while not (i == 0 and j == 0):
        dir_left = self.direction_matrix[i, j, Direction.Left.value - 1]
        dir_up = self.direction_matrix[i, j, Direction.Up.value - 1]
        dir_ul = self.direction_matrix[i, j, Direction.UpLeft.value - 1]
        if dir_left != 0:
            results.append(
                f"{Direction.get_char(dir_left)} : Insert {self.y[j - 1]}")
            j -= 1
        elif dir_up != 0:
            results.append(
                f"{Direction.get_char(dir_up)} : Delete {self.x[i - 1]}")
            i -= 1
        elif dir_ul != 0:
            results.append(f"{Direction.get_char(dir_ul)} :"
                           f" Replace {self.x[i - 1]} with {self.y[j - 1]}")
            j -= 1
            i -= 1

    return results
```

این متد مشابه متد قبل است، با این تفاوت که علاوه بر جهت، واضح نیز بیان میکند که منظور از این جهت چه بوده و دقیقا چه کاراکتری باید حذف اضافه و یا با چه کاراکتر دیگری جایگزین شود.

متد print_step_by_step: این متد، از همان متد get_instructions_raw استفاده میکند و پس از در آوردن جهت، روی رشته مبدا، مرحله به مرحله عملیات گفته شده را اعمال می کند در لحظه خروجی می دهد (یک generator function است) و این ها توسط صدا زننده تابع چاپ می شوند.

```python
def print_step_by_step(self):
    ins = self.get_instructions_raw()
    i = self.rows - 1
    j = self.cols - 1
    yield self.x
    for ch in ins:
        if ch == '↖':
            self.x = self.x[:i - 1] + self.y[j - 1] + self.x[i:]
            i -= 1
            j -= 1
        elif ch == '←':
            self.x = self.x[:i] + self.y[j - 1] + self.x[i:]
            j -= 1
        else:
            self.x = self.x[:i - 1] + self.x[i:]
            i -= 1
        yield self.x
```

(برای اجرا نیاز به پایتون 3.6+ دارد) اجرای برنامه در فایل App.py بوده و کافی است اجرا شود.

App.py × MinimumEditDistance.py ×

```python
from MinimumEditDistance import MinimumEditDistance

if __name__ == '__main__':
    x = "shayan"
    y = 'daneshvar'
    med = MinimumEditDistance(x, y)
    # print(f"distance matrix:\n {med.distance_matrix}")
    # print(f"direction matrix:\n {med.direction_matrix} ")
    # print("----------------------------------------")
    print(f"Strings: {x} & {y}")
    print(f"Minimum Edit Distance is: {med.get_min_dist()}")
    print("Instructions Extracted from results:(from the end to the beginning)")
    # print(med.get_instructions_raw())
    for each in med.get_instructions():
        print(each)
    print("Editing Source to Target Step by Step:")
    for each in med.print_step_by_step():
        print(each)
```

نمونه خروجی در کنسول:

```
C:\Users\TOP\AppData\Local\Programs\Python\Python36\python.exe "C:/Users/TOP/Desktop/NLP Project/App.py"
Strings: shayan & daneshvar
Minimum Edit Distance is: 9.0
Instructions Extracted from results:(from the end to the beginning)
← : Insert r
↑ : Delete n
↑ : Delete a
↑ : Delete y
↖ : Replace a with a
← : Insert v
↖ : Replace h with h
↖ : Replace s with s
← : Insert e
← : Insert n
← : Insert a
← : Insert d
Editing Source to Target Step by Step:
shayan
shayanr
shayar
shayr
shar
shar
shvar
shvar
shvar
eshvar
neshvar
aneshvar
daneshvar
```

همانطور که دیده میشود، فاصله ویرایشی کمینه برای تبدیل رشته shayan به daneshvar (و البته بر عکس) 9 است، روند این تبدیل نیز برای یکی از چند روند ممکن نیز اجرا شده و از رشته ورودی به رشته خروجی رسیده ایم. (هر رشته دلخواهی را می توانید وارد کنید و این روند را ببینید !)

پایان