IMT4204 Individual Project

Candidate 10049
Fall 2020

November 26, 2020

1 Task 2 – Approximate search algorithm using the classical dynamic programming approach

1.1 Introduction

The approximate search algorithm is a technique to find strings that match a pattern approximately in a text, allowing up to k errors. One definition of the approximate search algorithm problem is; given a pattern string p in a text T, find substrings s in T, which has edit distance, *ed(p, s) <= k* The edit distance is the minimum number of edit operations needed to transform one string into another, and can be divided into three elementary operations − substitute, delete and insert [1]. The oldest way of solving the approximate search algorithm is with the classical dynamic approach. This method may not be the most efficient a time complexity of $O(mn)$, but it is very adaptable to more complex distance functions. In the classical dynamic approach, the problem is solved using a matrix of partial edit distances, utilizing the fact that the solution to the problem depends upon its subproblems.

1.2 Computing edit distance

The first step to compute the edit distance between string x and string y, ed(x, y), with dynamic programming, is to fill a *n+1*-by-*m+1* matrix w, where n and m corresponds to the length of the strings x and y. In this matrix, $w_i,_j$ is $ed(x_{1...i}, y_{1...j})$ the minimum number of operations needed to match $x_{1...i}$ to $y_{1...j}$. There are two different ways to compute these values.

$$w_{0,0} \leftarrow 0 \tag{1}$$
$$w_{i,j} \leftarrow min(w_{i-1,j-1} + \delta(x_i, y_j), w_{i-1,j} + 1, w_{i,j-1} + 1)$$

Since this approach utilizes dynamic programming and subproblems, it is assumed that all edit distances between shorter strings already have been computed. This implies that when looking at two strings of length i and j, $ed(x_{1...i-1}, y_{1...j-1})$ is already computed. Following equation 1, the matrix starts with a 0 for the first element, $w_{0,0}$, since this is the edit distance between two empty strings. The rest of the matrix is computed using the second line, where $\delta(x_i, y_j) = 0$ if $x_i = y_j$ and 1 otherwise. The equation reflects the three different edit operations that can occur; **substitution**, **insertion** and **deletion**. To explain the three operations, we start by looking at the last characters, x_i and y_j. If these are the same, they are a match and does not have to be considered ($\delta(x_i, y_j) = 0$). The only operation needed is to convert $x_{1...i-1}$ into $y_{1...j-1}$, which has the cost of $w_{i-1,j-1}$.

For **substitution**, it means that x_i can be substituted with by y_i and converted $x_{1...i-1}$ into $y_{1...j-1}$ at cost of $w_{i-1,j-1} + 1$. This can be seen as going diagonal in the matrix. For **deletion**, x_i can be deleted, and the string $x_{1...i-1}$ can be converted into $y_{1...j}$ at cost of $w_{i-1,j} + 1$. This can be seen as going horizontal in the matrix. For **insertion**, y_j can be inserted at the end of $x_{1...i}$, and $x_{1...i}$ can be converted into $y_{1...j-1}$ at cost of $w_{i,j-1} + 1$. This can be seen as going vertical in the matrix. Figure 1 shows how these operations will be calculated in the matrix.

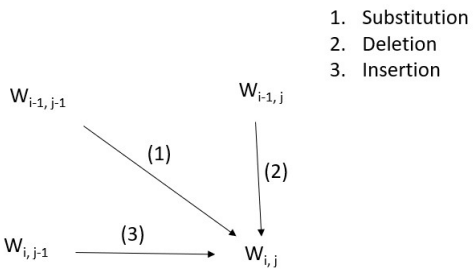

Figure 1: Edit operations diagonal, vertical or horizontal in the partial edit distance matrix

The second way of computing the partial edit distance matrix is shown below in equation 2.

$$w_{i,j} = \begin{cases} w_{i-1,j-1} & \text{if } x_i = y_j, \\ 1 + min(w_{i-1,j-1}, w_{i-1,j}, w_{i,j-1}) & \text{otherwise} \end{cases} \tag{2}$$

This method is faster to implement, and the computing itself is equivalent to equation 1 because neighboring cells in the matrix does not differ with more than 1. This means that if two characters are the same, we get $\delta(x_i, y_i) = 0$ and $w_{i-1,j-1}$ can not be larger than $w_{i-1,j} + 1$ or $w_{i,j-1} + 1$. To initialize this matrix, the first column and row needs to be filled with $0 \rightarrow i$ and $0 \rightarrow j$, where i and j is the length of the two strings used. An example of a partial edit distance matrix is shown in figure 2.

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	**0**	1	2	3	4	5	6	7	8
n	2	1	**0**	1	2	3	4	5	6	7
n	3	2	1	**0**	1	2	3	4	5	6
u	4	3	2	1	**1**	2	3	4	5	6
a	5	4	3	2	2	**1**	2	3	4	5
l	6	5	4	3	3	2	**1**	**2**	**3**	**4**

Figure 2: Partial edit distance matrix for X = "annual" and Y = "annealing"

The figure also shows the optimal path, which is the values marked in bold. The last value (4) represents the minimal number of operations needed to transform to entire word "annealing" to "annual". You can back-trace the values from the bottom right 4, to the upper-left 0, and find the exact edit operations done on every character. The most optimal answer is the bottom row 1, which is the transformation from "anneal" to "annual". A mapping, or alignment between all the characters of optimal path, can be seen in figure 3

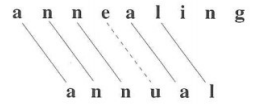

Figure 3: The mapping and edit operations done to transform "annealing" to "annual". The dashed line represents a substitution.

1.3 Text search allowing up to *k* errors

There are many similarities between searching for a pattern p in a text T and computing edit distances. The difference is that an occurrence could begin at any position in the text. By setting the first row to zeroes ($w_{0,j}, \forall j \in 0...n$), this is achievable because the empty pattern appears with zero errors at any position in the text. The reason for that is that it matches with any text substrings of length zero.

The bottom row in the partial edit matrix will contain all the different solutions to the string match, and by finding values that is below the error limit, k, we can accomplish to allow up to k errors. Figure 4 shows an example of a partial edit distance matrix with first row set to 0, allowing up to k errors.

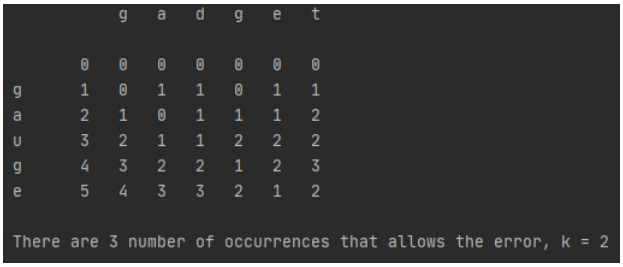

Figure 4: Allowing up to k = 2 errors, there are three solutions to X = "gauge" and Y = "gadget"

1.4 Time and space complexity

As mentioned in section 1.1, approximate search algorithm with classical dynamic programming is not very effective, taking $O(mn)$ time. There are other, better solutions to the algorithm when it comes to pure time complexity, but this will not be the focus now. When working with a n-by-m matrix, it is not enough with the time complexity of $O(mn)$, but it also needs the space complexity of $O(mn)$ space. However, it is possible to reduce this space complexity to $O(m)$. To compute the partial edit distance matrix needed to find the solution, it is enough to know the values from the last column. This can be seen in equation 1 and equation 2, where the "oldest" element we work with is always the previous column. To compute $w_{*,j}$, we need the values from $w_{*,j-1}$. Instead of building the whole matrix, we can maintain a column C of that we uses to compute the values needed. This column is updated every time j iterates. Introducing the column C, setting $x = p$ and $y = T$, we can now update equation 2 into equation 3:

$$C'_i = \begin{cases} C_{i-1} & \text{if } p_i = t_j, \\ 1 + min(C_{i-1}, C'_{i-1}, C_i) & \text{otherwise} \end{cases} \tag{3}$$

For a string with length m, the text positions $C_m < k$ are now reported as ending positions. This means that if everything you are after is to see if you can match a string with $k = 2$ errors, you could get the algorithm to end when one of the elements in the row C_m is $<= 2$

1.5 The trade-off between space complexity problem and the usability of the edit distance matrix

As explained in section 1.4, there is possible to implement a version of this matrix, reducing the space complexity from $O(mn)$ to $O(m)$, on the cost of not loading in the whole partial edit distance matrix to memory. In some cases, you might need to be able to back-trace to find the optimal path mentioned in section 1.2. One might need to be able to see the edit operations on every character in the string, something which is only possible while having the entire matrix in memory. This implementation is therefore a trade-off between usability and space complexity, which depends on what the implementer needs the algorithm for.

2 Task 4 – Bit-vector realization of the dynamic programming-based approximate search algorithm

2.1 Introduction

As seen in section 1, the approximate search algorithm using classical dynamic programming approach completes in *O(mn)* time, with the best case of O(m) space. Gene Myers discovered a way of improving the time complexity, using bit-parallelism, and calculate horizontal and vertical deltas without the need to iterate through the whole partial edit distance matrix [2]. The time complexity of Myers' algorithm is *O([mn]/w)*, which is approximately equal to *O(n)* assuming the pattern is less than the size of the computer word, or $m <= w$. The computer word, w, is typically 32 or 64 on most current machines, depending on the operative system running.

2.2 Background research and partial edit distance matrix properties

The partial edit distance matrix used in the algorithm from section 1 is filled following equation 2. Figure 5 shows the matrix for the pattern "annual" and the text "annealing". Myers' algorithm never actually calculate this matrix, only the deltas needed to compute the last row. This is achieved by applying the properties between diagonal and adjacent cells in the matrix. The reference used in this section denotes the dynamical program matrix as D instead of w.

		a	n	n	e	a	l	i	n	g
	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	1	0	1	1	1	1
n	2	1	0	1	2	1	1	2	1	2
n	3	2	1	0	1	2	2	2	2	2
u	4	3	2	1	1	2	3	3	3	3
a	5	4	3	2	2	1	2	3	4	4
l	6	5	4	3	3	2	1	2	3	4

Figure 5: Dynamic programming matrix for x = "annual" and y = "annealing"

$$\begin{aligned} \textbf{The diagonal property: } & D_{i,j} - D_{i-1,j-1} = 0 \text{ or } 1. \\ \textbf{The adjacency property: } & D_{i,j} - D_{i,j-1} = \textit{-1, 0 or 1, and} \\ & D_{i,j} - D_{i-1,j} = \textit{-1, 0 or 1.} \end{aligned} \tag{4}$$

It is possible to verify the properties in equation 4 using the partial edit distance matrix in figure 5.

The last row in the partial edit distance matrix, $D_{m,j}$, shows the different occurrences of the pattern in the text, and by returning j when $D_{m,j} <= k, j \in (1...n)$, it is possible to find and return the first match allowed by an error, k, before every element in the entire matrix is iterated through. A consequence to the diagonal property is that if $D_{i,j} > k$, then $D_{i+r,j+r} > k$ for $r > 0$. In other words, there is no need to compute the rest of the diagonal if this is the case. Also, if q is the lowest cell (highest i value) of the *(j-i)*th column that has a value $<= k$, only the cells $D_{p,j}, D_{2,j}, ..., D_{q+1,j}$ of the jth column needs to be calculated. This is because we know that the cells $D_{i,j} > k$ for $i > q + 1$, which means that these cells can not contribute to find an occurrence. When filling the rest of the cells, the unknown values can then be ignored, since they will not give us a match that is approved by k.

2.3 Myers' bit-vector algorithm

In pseudo-code, we will use python-like notation for bit operations. **"&"** denotes bitwise **AND**, **"|"** bitwise **OR**, **"^"** bitwise **XOR** and **"<<"** /**">>"** for shifting a bit-vector to the left/right.

Shifting is assumed to use zero filling. To compute one of the vectors, the logical operation $\sim$, or bitwise complement is needed. Python does not have such a function built in, so this is solved by performing **XOR** with a 1^m bit string.

Preprocessing and initializing

Myers bit-vector algorithm takes advantage of the different properties in the partial edit distance matrix, and computes a match based on different bit-vectors. Instead of storing the actual cell values in the dynamic programming matrix, the differences between the values of adjacent cells (deltas) are stored. Based on the diagonal and adjacency properties, we use the following vectors to represent the partial edit distance matrix:

- **The vertical positive delta vector VP_j:** $VP_j[i] = 1$ iff $D[i,j] - D[i-1,j] = 1$.
- **The vertical negative delta vector VN_j:** $VN_j[i] = 1$ iff $D[i,j] - D[i-1,j] = -1$.
- **The horizontal positive delta vector HP_j:** $HP_j[i] = 1$ iff $D[i,j] - D[i,j-1] = 1$.
- **The horizontal negative delta vector HN_j:** $HN_j[i] = 1$ iff $D[i,j] - D[i,j-1] = -1$.
- **The diagonal zero delta vector $D0_j$:** $D0_j[i] = 1$ iff $D[i,j] = D[i-1,j-1]$.

Figure 6 below, shows how these vectors are computed from a partial edit distance matrix. It is possible to compute the values of the dynamic programming matrix if either both VP_j and VN_j, or HP_j and HN_j are known. This allows us to recover values we might need later, without having to store the matrix.

		t	o	m	o	r	r	o	w
	0	0	0	0	0	0	0	0	0
t	1	0	1	1	1	1	1	1	1
w	2	1	1	2	2	2	2	2	1
o	3	2	1	2	2	3	3	3	2

			j=5		
VP	VN	HP	HN	D0
1	0	0	0	0
1	0	0	0	0
1	0	1	0	0

Figure 6: Dynamic programming matrix for pattern "two" and text "tomorrow". On the right are the vectors mentioned above, based on the column j=5 which is shadowed.

The last bit-vector that we need before starting the search, is the pattern match vector PM_λ, for every character λ in the alphabet:

- **The pattern match vector PM_λ:** $PM_\lambda[i] = 1$ iff $Pat[i] = \lambda$.

This means that PM_λ can be thought of as a python dictionary, where the keys are the characters in the alphabet, and the values are the corresponding bit strings containing the matches for the pattern. Figure 7 shows an example of PM_λ, when the pattern is "annual" and the text is "annealing".

	a	n	u	l	e	i	g
a	1	0	0	0	0	0	0
n	0	1	0	0	0	0	0
n	0	1	0	0	0	0	0
u	0	0	1	0	0	0	0
a	1	0	0	0	0	0	0
l	0	0	0	1	0	0	0

Figure 7: The pattern match vector with pattern = "annual" and text = "annealing", resulting in the alphabet "anuleig". Reading the strings bottom-up gives $PM_a = 010001_2 = 17_{10}$

Myers' algorithm imitates column-wise filling of the partial edit distance matrix, calculating only the cell values needed. These values are the bottom row, or $D_{m,j}$ for $j \in 1...n$, exiting the algorithm and returning the value as fast as a value in the bottom row $< k$ is found. The rest of the cells will only be represented by the delta bit-vectors defined earlier. For the initialization of the different vectors, the following conditions apply:

$$VP_0[i] = 1 \text{ and } VN_0[i] = 0, \text{ according to the cells in } D[i, 0] \text{ for } i \in (1...m).$$
$$HP_0 = HN_0 = D0 = 0$$
$$D[m, 0] = m$$

Searching

The next part of the algorithm is the search in itself. In the search, whenever the algorithm moves between two columns, for an example *j-1* to *j*, it involves five (4+1) steps:

1. Calculate $D0_j$ from the other vectors PM_j, VP_{j-1} and VN_{j-1}.
2. Calculate HP_j and HN_j from the other vectors $D0_j$, VP_{j-1} and VN_{j-1}.
3. Calculate $D[m, j]$ from $D[m, j-1]$, $HP_j[m]$ and $HN_j[m]$.
4. Calculate VP_j and VN_j from $D0_j$, HP_j and HN_j.
5. If $D[m, j] <= k$, stop the algorithm and return j, an approximate occurrence of the pattern is now found in position j.

Step 5 can be combined with step 3, checking if $D[m, j] <= k$ directly after computing it. This leads to the algorithm not having to complete step 4 if an occurrence is found.

Step 1: $D0_j$

From the properties in equation 4, we can see that there is three different ways that $D0_j[i]$ gets the value 1:

1. $D[i, j-1] = D[i-1, j-1] - 1$, in other words $VN_{j-1}[i] = 1$. Using *D[i,j]= D[i, j-1] + 1* enables the zero-difference to propagate from left.
2. $PM_j[i] = 1$, meaning the zero-difference comes from *Pat[i] = Text[j]*, setting *D[i,j]* to D[i-1,j-1].
3. $D[i-1, j] = D[i-1, j-1] - 1$. Using *D[i,j]= D[i-1, j] + 1* enables the zero-difference to propagate from above.

The first two cases are simple, because $D0_j[i] = 1$ if $VN_{j-1}[i] = 1$ and/or $PM_j[i] = 1$. All we need to do here is **OR**-ing the three vectors together. Case 3 on the other hand, is much more complex. Myers present a nice way of solving this problem, and by using the different properties for the partial edit distance matrix, we can see that case 3 is the same as finding $((PM_j \ \& \ VP_{j-1}) + VP_{j-1})\hat{}VP_{j-1}$. The full explanation and derivation can be read in reference [2], pages 5-7.

By combining the three cases, we can derive the following formula for computing $D0_j$:

$$D0_j = (((PM_j VP_{j-1}) + VP_{j-1})\hat{}VP_{j-1})|PM_j|VN_{j-1} \tag{5}$$

Step 2: HP_j **and** HN_j

From the dynamic programming matrix it can be seen that $HP_j[i] = 1$ iff either *D[i,j-1] = D[i-1, j-1] - 1*, or *D[i,j] = D[i-1, j-1] + 1* and *D[i, j-1] = D[i-1, j-1]*. Looking at the delta bit-vectors,

this means that $HP_j[i] = 1$ iff $VN_{j-1}[i] = 1$ or $D0_j[i] = 0$ and $VP_{j-1}[i] = 0$ and $VN_{j-1} = 0$. This can also be seen in figure 8 (cases a and b). Since $VN_{j-1}[i] = 1$ implicit expresses $VN_{j-1} = 0$, the formula for computing vector $HP_j[i]$ is the following:

$$HP_j = VN_{j-1}| \sim (D0_j|VP_{j-1}) \tag{6}$$

In the same way, we can see that $HN_j[i] = 1$ iff $D[i, j] = D[i - 1, j - 1]$ and $D[i, j - 1] = D[i - 1, j - 1] + 1$, leading to $VN_j[i] = 1$ iff $D0_j[i] = 1$ and $VP_{j-1}[i] = 1$. Figure 8 (case c) shows the example of this. This gives us the following formula:

$$HN_j = D0_j VP_{j-1} \tag{7}$$

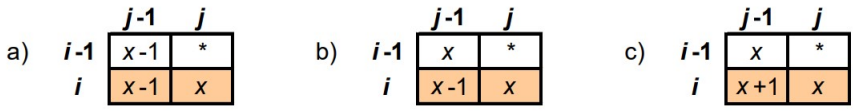

Figure 8: a and b gives the only combinations of HP_j, while c gives the only example of HN_j

Step 3: $D[m, j]$

By using the horizontal vector deltas computed in the last step, it is now possible to calculate the value $D[m, j]$ from $D[m, j - 1]$. We know that if the horizontal delta positive is 1, then $D[m, j]$ must be $D[m, j - 1] + 1$. In similar fashion, if the horizontal delta negative is 1, then $D[m, j]$ must be $D[m, j - 1] - 1$. If both deltas are 0, we know that no change has happened in the row and that $D[m, j] = D[m, j - 1]$. After computing the value $D[m, j]$, we check if it's smaller than the error, k. If this is true, the algorithm exits and returns the position j as the occurrence to the pattern in the text.

Step 4: VP_j **and** VN_j

The last step is very similar to step 2, being diagonally symmetric. We compute VP_j almost the same way as we computed HP_j, and we have that $VP_j[i] = 1$ iff $HN_j[i - 1] = 1$ or $D0_j[i] = 1$ and $HP_j[i - 1] = 0$. The main difference now, is that the $(i - 1)$th row bits $HN_j[i - 1]$ and $HP_j[i1]$ needs to be aligned with the ith row bit $VP_j[0]$. This can be done shifting the bits one step down (one step to the left). The formula is therefore almost the same as for HP_j, just with bit-shifting:

$$VP_j = (HN_j << 1)| \sim (D0_j|(VP_j << 1)) \tag{8}$$

And for VN_j:

$$VN_j = D0_j(HP_j << 1) \tag{9}$$

Summary and conclusion

The only difference between this algorithm and Myers original algorithm, is that while Hyyrö chose to use one bit-vector $D0_j$ instead the two vectors, XV_j and XH_j which Myers used. The difference between these choices have no particular significance as long as the algorithm is used for sequential approximate string matching with *ed*. The code in figure 9 shows the python code for the preprocessing and initialization of the different delta bit-vectors used in the algorithm. It also shows an example of how to compute the alphabet and the pattern match vector, PM. Figure 10 shows the algorithm looping through step 1-4, with step 5 being combined with step 3.

```python
# Preprocess

m = len(x) + 1
n = len(y) + 1

# Init all vectors
VP = [0] * n
VP[0] = int("1" * len(x), 2)
VN = [0] * n
HP = [0] * n
HN = [0] * n
D0 = [0] * n
PM = {}
D_MJ = len(x)

# Find the alfabet in x+y
for character in str(x + y):
    PM[character] = 0
# Find the match for PM on x
for iteration, character in enumerate(x):
    zerosBefore = (m - iteration) * "0"
    zerosAfter = iteration * "0"
    PM[character] = PM[character] | int(zerosBefore + "1" + zerosAfter, 2)
```

Figure 9: Preprocessing and initialization of the different bit-vectors used in Myers algorithm. This code is written in Python.

```python
# Search
for j, character in enumerate(y):
    j += 1
    # 1. D0_j computed from PM_j, VP_j-1 and VN_j-1
    D0[j] = (((PM[character] & VP[j - 1]) + VP[j - 1]) ^ VP[j - 1]) | PM[character] | VN[j - 1]

    # 2. HP_j and HN_j computed from D0_j, VP_j-1 and VN_j-1
    HP[j] = VN[j - 1] | ((D0[j] | VP[j - 1]) ^ int("1" * m, 2))
    HN[j] = D0[j] & VP[j - 1]

    # 3. D_MJ computed from D_M[j-1] and HP_j[m], HN_j[m]
    if HP[j] & int("1" + "0" * (len(x) - 1), 2):
        D_MJ += 1
    if HN[j] & int("1" + "0" * (len(x) - 1), 2):
        D_MJ -= 1

    # Ends at text position j when D_M[j] <= k
    if D_MJ <= k:
        return j

    # 4. VP_j and VN_j computed from D0_j, HP_j and HN_j
    VP[j] = (HN[j] << 1) | ((D0[j] | (HP[j] << 1)) ^ int("1" * m, 2))
    VN[j] = D0[j] & (HP[j] << 1)
return 0
```

Figure 10: The search part of the algorithm, looping through step 1-4. This code is written in Python.

References

[1] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings: practical online search algorithms for texts and biological sequences*. Cambridge university press, 2002.

[2] Heikki Hyyrö. Explaining and extending the bit-parallel approximate string matching algorithm of myers. Technical report, Citeseer, 2001.