

# IMT4204 Intrusion Detection in Physical and Virtual Networks

## Project reports:

### Report #1, task 2:

The classical approximate search algorithm is an algorithm for finding patterns in a given search text. What separates this from exact search, is that the approximate search algorithm allows for up to  $k$  errors. This means that the algorithm is also able to locate patterns that does not exactly match the search text.

Expressed concisely, it is an algorithm that takes a search pattern, a search text and an integer  $k$  as input, and returns the ending positions where a “match” is found. “Match” meaning that the edit distance between the pattern and some part the search text is  $\leq k$ , where edit distance is defined to be the number of insertion, deletions and/or substitutions that are needed to transform one string into the other.

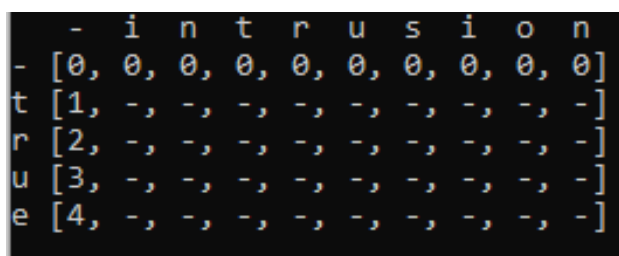
As an example is the edit distance between “cat” and “hat” 1 (substitute “c” with “h”) and the edit distance between “cat” and “hats” is 2 (substitute “c” with “h” and delete “s” from “hats”). Or formulated concisely:  $ed(cat, hat)=1$  and  $ed(cat, hats)=2$ .

One common way to implement this algorithm is to use dynamic programming, and the remainder of this report will discuss more precisely how the algorithm operates using the dynamic programming approach.

When implementing the algorithm using the dynamic programming approach, the key element is a matrix containing edit distances, and the operation of the algorithm mainly considers how to fill this matrix with the appropriate values. This is easiest shown with an example:

Assume we have the search pattern “true”, the search pattern “intrusion” and  $k = 1$ . Something worth noting is that if  $k=0$ , the algorithm performs an exact search (no errors allowed).

The first step is to initialize the matrix. As defined in [1], the initialized matrix should look as follows:



-	i	n	t	r	u	s	i	o	n
-	[0,	0,	0,	0,	0,	0,	0,	0,	0]
t	[1,	-,	-,	-,	-,	-,	-,	-,	-]
r	[2,	-,	-,	-,	-,	-,	-,	-,	-]
u	[3,	-,	-,	-,	-,	-,	-,	-,	-]
e	[4,	-,	-,	-,	-,	-,	-,	-,	-]

The column of consecutive increasing numbers contains the edit distances between the empty string (denoted by “-”) and the pattern. It can be read that  $ed(-, t)=1$  and  $ed(-, true)=4$ . In a traditional matrix of edit distances, the top row would also contain consecutive increasing numbers, but when used in approximate search, it is set to all zeros. This is because it is not known in advance where in the search text the pattern will be found and the calculation of edit distances can therefore start at any position.

After the initialization, the algorithm fills the matrix using the following update formula:

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

Formula 1.

Where  $M_{i,j}$  is the current cell (located at row  $i$  and column  $j$ ) in the matrix,  $x_i$  is the pattern symbol at index  $i$  and  $y_j$  is the search text symbol at index  $j$ .

Based on this formula, it is seen that a cell's value is determined by merely the values of the cells directly above, to the left and above the left.

As an example, let us calculate  $M_{1,1}$ . We have that  $x_i = x_1 = 't'$  and  $y_j = y_1 = 'i'$ . Because  $'i' \neq 't'$ , we need to use the bottom line of the update formula:

$$M_{i,j} = 1 + \min(M_{i-1,j-1}, M_{i-1,j}, M_{i,j-1}) = 1 + \min(0,0,1) = 1 + 0 = 1$$

The matrix now looks like this:

	-	i	n	t	r	u	s	i	o	n
-	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]									
t	[1, 1, -, -, -, -, -, -, -, -, -]									
r	[2, -, -, -, -, -, -, -, -, -, -]									
u	[3, -, -, -, -, -, -, -, -, -, -]									
e	[4, -, -, -, -, -, -, -, -, -, -]									

The update formula should then be performed on all the remaining cells, and the final matrix will then look like this:

	-	i	n	t	r	u	s	i	o	n
-	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]									
t	[1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1]									
r	[2, 2, 2, 1, 0, 1, 2, 2, 2, 2, 2]									
u	[3, 3, 3, 2, 1, 0, 1, 2, 3, 3, 3]									
e	[4, 4, 4, 3, 2, 1, 1, 2, 3, 4, 4]									

When the matrix is full, any matches can be found in the cells where  $M_{n,j} \leq k$ , with  $n$  being the length of the pattern. In this case there are two such cells:  $M_{4,5}=1$  and  $M_{4,6}=1$ . The result can then be interpreted as the pattern "true" was found in the search text "intrusion" at index 5 and 6, with an edit distance of 1. By finding the trace of minimal edit distances by traversing the matrix in reverse direction, starting from the result cells  $M_{4,5}$  and  $M_{4,6}$ , one can find the substrings of the search text that caused the match. By the plotting below, it becomes visible that the pattern "true" matched with two substrings, "tru" and "trus", of "intrusion". The tracing is displayed below:

	-	i	n	t	r	u	s	i	o	n
-	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]									
t	[1, 1, 1, 0, 1, 1, 1, 1, 1, 1]									
r	[2, 2, 2, 1, 0, 1, 2, 2, 2, 2]									
u	[3, 3, 3, 2, 1, 0, 1, 2, 3, 3]									
e	[4, 4, 4, 3, 2, 1, 0, 1, 2, 3, 4]									

This algorithm can also be implemented in a more space-efficient way, where it is not necessary to store the entire matrix in memory. If one takes a look at the update formula, one can see that the value of a cell is determined by cells that are in column  $j$  or column  $j-1$ . This means that one can run the algorithm while only keeping two of the matrix's columns in memory. This version is also described in [1]. The initialization consists of defining a column  $C$  where  $C_i = i$ . The next column  $C'$  can then be created using a new, but highly similar, update formula:

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

Formula 2.

Where  $p_i$  is the pattern symbol at index  $i$  and  $t_j$  is the search text symbol at index  $j$ . At this time there should be two columns stored. To continue, one sets  $C=C'$ , before calculating a new  $C'$ . This continues for each character in the search text. If any of the columns have a last element  $\leq k$ , this index is reported as a match.

### User instructions:

An implementation of this algorithm has been created as part of Task 1 in this project. To run the implementation, one simply needs to run the Python-file "classical\_approximate\_search.py" and provide input as described in on-screen instructions. The implementation contains both the storage-efficient and the less storage-efficient (stores the entire matrix in memory) version discussed in this report (follow on-screen instructions to choose which version you want to run).

## Report #2, task 3:

As discussed in Report #1, when implementing the classical approximate search algorithm using dynamic programming, one could either store the entire dynamic programming matrix in memory or only store the two columns necessary for making the calculations. The two approaches have the same time complexity, but by only storing the two necessary columns, the space complexity could be reduced from  $O(mn)$  to  $O(m)$ , where  $m$  is the length of the search pattern and  $n$  is the length of the search string [1].

This might lead one to wonder if there are any situations where it is necessary to keep the entire matrix in memory, even though the algorithm can be implemented without it.

The answer is that the dynamic programming matrix is necessary if one wants to find the exact substring of the search text that caused the match.

To find this substring one needs to find the entire path of minimum edit distances, and this can only be found by storing the dynamic programming matrix and traversing it in reverse. An example of how to find this is shown in the previous report (bottom page 2). This is not possible if only the two current columns are stored.

In a theoretical setting, where memory usage is of no concern, one could also argue that keeping the entire matrix in memory could increase time efficiency. By keeping the matrix in memory, one would not need to reconstruct it each time the same search text was to be compared with the same pattern. This would however probably require too much memory to be considered practical, unless the search text was predictable, and the number of patterns were low.

## Report #3, task 4:

Another algorithm that solves the same problem as in [1], was designed by Myers in 1999 [2]. This algorithm was also described in [3] (with slight modifications). This report will be based on the algorithm as described in [3].

Although this algorithm solves the same problem of approximate search, it has the advantage of being faster. It reduces the time complexity from  $O(mn)$ , as in [1], to  $O(\lceil mn \rceil / w)$ , where  $m$  is the pattern length,  $n$  is the length of the search text and  $w$  is the computer word size. This is due to the use of bit-parallelism.

The main difference between the algorithms is how the dynamic programming matrix is stored. Contrary to how it is stored in [1], Myers' algorithm uses delta vectors to describe the matrix [3]. Each column  $j$  in the matrix is represented by 5 delta vectors ( $D0_j$ ,  $HP_j$ ,  $HN_j$ ,  $VP_j$  and  $VN_j$ ), which together holds all relevant information about the state of the matrix. An example is seen below. The highlighted column's delta vectors would hold the following values:

	-	i	n	t	r	u	s	i	o	n
-	[0,	0,	0,	0,	0,	0,	0,	0,	0,	0]
t	[1,	1,	1,	0,	1,	1,	1,	1,	1,	1]
r	[2,	2,	2,	1,	0,	1,	2,	2,	2,	2]
u	[3,	3,	3,	2,	1,	0,	1,	2,	3,	3]
e	[4,	4,	4,	3,	2,	1,	1,	2,	3,	4]

$D0_4$	$HP_4$	$HN_4$	$VP_4$	$VN_4$
0	1	0	1	0
1	0	1	0	1
1	0	1	1	0
1	0	1	1	0

The formulas for filling these vectors are defined as follows, where  $D[i,j]$  denotes indexing of the dynamic programming matrix [3]:

Diagonal zero delta vector ( $D0_j$ ):  $D0_j[i] = 1$  iff  $D[i,j] = D[i-1,j-1]$ , otherwise 0

Horizontal positive delta vector ( $HP_j$ ):  $HP_j[i] = 1$  iff  $D[i,j] - D[i,j-1] = 1$ , otherwise 0

Horizontal negative delta vector ( $HN_j$ ):  $HN_j[i] = 1$  iff  $D[i,j] - D[i,j-1] = -1$ , otherwise 0

Vertical positive delta vector ( $VP_j$ ):  $VP_j[i] = 1$  iff  $D[i,j] - D[i-1,j] = 1$ , otherwise 0

Vertical negative delta vector ( $VN_j$ ):  $VN_j[i] = 1$  iff  $D[i,j] - D[i-1,j] = -1$ , otherwise 0

These vectors are calculated for each column, which makes it possible to calculate  $D[m,j]$  at each  $j$ , which again makes it possible to check if a match with  $\leq k$  errors is found. The best way to describe to this algorithm in more detail, is by the means of a concrete example:

We use the same example as in Report #1, where the search pattern is "true", the search text is "intrusion" and  $k=1$ . As defined in [3], we also initialize  $VP_0 = 1^m = 1111$ ,  $VN_0 = 0^m = 0000$  and  $D[m,j] = m = 4$ . As stated before,  $m$  is the length of the pattern.

We also initialize an additional variable  $PM_\lambda$ , which stores information about where a character  $\lambda$  is found in the search pattern. As the search pattern is “true”, it will be defined as this (bit-positions grow from right to left):

$PM_t=0001$ ,  $PM_r=0010$ ,  $PM_u=0100$ ,  $PM_e=1000$  and all other  $PM_\lambda=0000$ .

After this initialization, the following four steps are to be performed columnwise for each column  $j$ :

1. Calculate  $D0_j$ .
2. Calculate  $HP_j$  and  $HN_j$ .
3. Calculate  $D[m,j]$ .
4. Calculate  $VP_j$  and  $VN_j$ .

We start with calculating  $D0_j$ :

As described above,  $D0_j[i]=1$  iff  $D[i,j]=D[i-1,j-1]$ . There are three cases that can make this happen, which can be derived the traditional update formula for the dynamic programming matrix (Formula 1).

1. If  $D[i,j-1]=D[i-1,j-1]-1$  (or expressed with delta vectors: if  $VN_{j-1}[i]=1$ ), which happens when  $M_{i,j}=1+M_{i,j-1}$  from Formula 1.
2. If  $PM_\lambda=1$ , which happens when  $M_{i,j}=M_{i-1,j-1}$  from Formula 1.
3. If  $D[i-1,j]=D[i-1,j-1]-1$  (or expressed with delta vectors: if  $HN_j[i-1]=1$ ), which happens when  $M_{i,j}=1+M_{i-1,j}$  from Formula 1.

Any one of these can be fulfilled to make  $D0_j[i]=1$ , so  $D0_j=\text{case1}|\text{case2}|\text{case3}$ . Case 1 and 2 have already been defined ( $VN_{j-1}[i]$  and  $PM_\lambda$ ). Case 3 is a bit more complicated as  $HN_j$  has not been defined yet, but there has been found a solution for this [3]:

One observation is that  $D[i-1,j] = D[i-1,j-1] - 1$  iff  $D0_j[i] = 1 \ \& \ D0_j[i-1] = 1 \ \& \ VP_j[i-1] = 1$ . Based on this and the fact that even if  $VN_j[i-1] = 0$ ,  $D0_j[i-1] = 1$  iff either case 1 or 2 is fulfilled, Hyvrö[3] showed that  $D[i-1,j] = D[i-1,j-1] - 1$  iff  $VP_j[i-1] = 1$  and  $PM_j[i-1] = 1$  or  $D[i-2,j] = D[i-2,j-1] - 1$ .

He further showed that this can be applied recursively with decreasing  $i$ 's until one arrives at an integer  $h < i-1$ , where  $PM_j[h] = 1$ . The recursion can stop due to the guaranteed condition that  $D[0,j] \neq D[0,j-1] - 1$  (the top row is all 0s). This led to the rule:

$D[i-1,j] = D[i-1,j-1] - 1$  iff  $\exists h: PM_j[h] = 1$  and  $VP_{j-1}[q] = 1$ , for  $q = h \dots i-1$ .

To derive a formula from this, Hyvrö exploited the fact that the way the run of 1s in  $VP_{j-1}[q]$  propagated down diagonal zero-differences (this effect can be spotted between the columns ‘n’ and ‘t’ in the dynamic programming matrix on the previous page) shared some similarity with carrying in integer addition [3]. Thus, he expressed the rule in the following formula:

$D0_j = (((PM_j \ \& \ VP_{j-1}) + VP_{j-1}) \wedge VP_{j-1}) \mid PM_j \mid VN_{j-1}$ .

To return to our example, we would have that:

$$\begin{aligned}
D0_1 &= (((PM_{i'} \& VP_0) + VP_0) \wedge VP_0) \mid PM_{i'} \mid VN_0 \\
&= (((0000 \& 1111) + 1111) \wedge 1111) \mid 0000 \mid 0000 = ((0000+1111) \wedge 1111) \mid 0000 \mid 0000 \\
&= (1111 \wedge 1111) \mid 0000 \mid 0000 = 0000 \mid 0000 \mid 0000 = 0000.
\end{aligned}$$

The next step is then to calculate  $HP_j$  and  $HN_j$ . By looking at the definitions of the delta vectors, we can derive that  $HP_j[i] = 1$  iff  $D[i,j-1] = D[i-1,j-1]-1$  or  $D[i,j] = D[i-1,j-1]$  and  $D[i,j-1] = D[i-1,j-1]$ . Expressed with delta vectors, this can be translated to  $HP_j[i] = 1$  iff  $VN_{j-1}[i] = 1$  or  $D0_j[i] = 0$  and  $VP_{j-1}[i] = 0$  and  $VN_{j-1}[i] = 0$ . This rule can be expressed as the Boolean equation  $HP_j = VN_{j-1} \mid \sim(D0_j \mid VP_{j-1} \mid VN_{j-1})$ . Because  $VN_{j-1} \mid \sim VN_{j-1}$  always is 1 (absorption law), we can finally simplify the expression to  $HP_j = VN_{j-1} \mid \sim(D0_j \mid VP_{j-1})$ .

Following a similar procedure, we can see that  $HN_j[i] = 1$  iff  $D[i,j] = D[i-1,j-1]$  and  $D[i,j-1] = D[i-1,j-1]+1$ . By translating this rule to be expressed with delta vectors, we get that  $HN_j = D0_j \& VP_{j-1}$ .

In our example we can then calculate that  $HP_1 = VN_0 \mid \sim(D0_1 \mid VP_0) = 0000 \mid \sim(0000 \mid 1111) = 0000 \mid \sim(1111) = 0000$ . We can also calculate that  $HN_1 = D0_1 \& VP_0 = 0000 \& 1111 = 0000$ .

The next step is to calculate  $D[m,j]$  and compare it with  $k$  to determine whether a match was found or not. This is easy as we have already obtained  $HP_j$  and  $HN_j$ , and we also know that  $D[m,0] = m = 4$ , as defined during initialization.

By looking at values of  $HP_j[m]$  and  $HN_j[m]$ , we can determine whether  $D[m,j]$  has increased by 1, decreased by 1 or stayed the same. This makes the rules that if  $HP_j[m] = 1$ ,  $D[m,j] = D[m,j-1] + 1$ , if  $HN_j[m] = 1$ ,  $D[m,j] = D[m,j-1] - 1$  and otherwise  $D[m,j] = D[m,j-1]$ . After this, one should check  $D[m,j] \leq k$ , and if true, it should be reported that a match has been found.

In our example, we have that  $HP_1[m] = HN_1[m] = 0$ , so the value of  $D[m,1]$  will be 4.

Finally,  $VP_j$  and  $VN_j$  is to be calculated. This is very similar to the way  $HN_j$  and  $HP_j$  were calculated. By looking at the definition of the delta vectors again, one can derive that  $VP_j[i] = 1$  iff  $HN_j[i-1] = 1$  or  $D0_j[i] = 1$  and  $HP_j[i-1] = 0$ . From this, the formula  $VP_j = (HN_j \ll 1) \mid \sim(D0_j \mid (HP_j \ll 1))$  was defined. The reason  $HP_j$  and  $HN_j$  are shifted is to compensate for the row difference between  $VP_j[i]$  and  $HP_j[i-1]$  and  $HN_j[i-1]$ .

Similar for  $VN_j$ , we see that  $VN_j[i] = 1$  iff  $D0_j[i] = 1$  and  $HP_j[i-1] = 1$ . Again, we need to shift  $HP_j$  (due to row difference between  $i$  and  $i-1$ ), and we end up with the formula  $VN_j = D0_j \& (HP_j \ll 1)$ .

To return to our example, we have that  $VP_1 = (HN_1 \ll 1) \mid \sim(D0_1 \mid (HP_1 \ll 1)) = (0000 \ll 1) \mid \sim(0000 \mid 0000 \ll 1) = 0000 \mid \sim(0000 \mid 0000) = 0000 \mid 1111 = 1111$  and  $VN_1 = D0_1 \& (HP_1 \ll 1) = 0000 \& (0000 \ll 1) = 0000 \& 0000 = 0000$ .

This means that after the first character 'i' in the search text, the delta vectors contain the following:

D0 <sub>1</sub>	HP <sub>1</sub>	HN <sub>1</sub>	VP <sub>1</sub>	VN <sub>1</sub>
0	0	0	1	0
0	0	0	1	0
0	0	0	1	0
0	0	0	1	0

To summarize, another iteration of the algorithm will be shown in its entirety. Assume that the algorithm has finished processing the characters 'i', 'n', 't' and 'r'. The next character is then 'u', meaning j=5.

The delta vector before processing 'u' are:

D0 <sub>4</sub>	HP <sub>4</sub>	HN <sub>4</sub>	VP <sub>4</sub>	VN <sub>4</sub>
0	1	0	1	0
1	0	1	0	1
1	0	1	1	0
1	0	1	1	0

We also know that  $D[m,4] = 2$  and, again, remember that bits grows from right to left.

Step 1, calculate D0<sub>5</sub>:

$$D0_5 = (((PM_{u'} \& VP_4) + VP_4) \wedge VP_4) \mid PM_{u'} \mid VN_4 = (((0100 \& 1101) + 1101) \wedge 1101) \mid 0100 \mid 0010 = ((0100 + 1101) \wedge 1101) \mid 0100 \mid 0010 = (0001 \wedge 1101) \mid 0100 \mid 0010 = 1100 \mid 0100 \mid 0010 = 1110$$

Step 2, calculate HP<sub>5</sub> and HN<sub>5</sub>:

$$HP_5 = VN_4 \mid \sim(D0_5 \mid VP_4) = 0010 \mid \sim(1110 \mid 1101) = 0010 \mid \sim(1111) = 0010$$

$$HN_5 = D0_5 \& VP_4 = 1110 \& 1101 = 1100$$

Step 3, calculate D[m,j]:

$$HN_5[m] = 1, \text{ so } D[m,5] = D[m,4] - 1 = 2 - 1 = 1. \text{ } 1 \leq k \text{ is true, so a match is reported.}$$

Step 4, calculate VP<sub>5</sub> and VN<sub>5</sub>:

$$VP_5 = (HN_5 \ll 1) \mid \sim(D0_5 \mid (HP_5 \ll 1)) = (1100 \ll 1) \mid \sim(1110 \mid (0010 \ll 1)) = 1000 \mid \sim(1110 \mid 0100) = 1000 \mid \sim(1110) = 1001$$

$$VN_5 = D0_5 \& (HP_5 \ll 1) = 1110 \& (0010 \ll 1) = 1110 \& 0100 = 0100$$

The delta vectors are now:



D0 <sub>5</sub>	HP <sub>5</sub>	HN <sub>5</sub>	VP <sub>5</sub>	VN <sub>5</sub>
0	0	0	1	0
1	1	0	0	0
1	0	1	0	1
1	0	1	1	0

### User instructions:

An implementation of this algorithm has been created as part of Task 4 in this project. To run the implementation, one simply needs to run the Python-file “bit\_vector\_approximate\_search.py” and provide input as described in on-screen instructions.

### References:

- [1] Navarro G. and Raffinot M., Flexible Pattern Matching in Strings, Cambridge University Press, 2002
- [2] Myers G., A fast bit-vector algorithm for approximate string matching based on dynamic programming, Journal of the ACM, Vol. 46, No. 3, May 1999, pp. 395-415
- [3] Hyvrö H., Explaining and extending the bit-parallel approximate string matching algorithm of Myers, Technical Report A-2001-10, University of Tampere, Finland, 2001.