* 1. Find in the text T all occurences of the substrings which Hamming distance form the pattern P is at most *k*. Apply the dynamic programming approach.

a) *T* = ccacbaabccaccbcabccc, *P* = abcba, *k* = 2,

b) *T* = 00011101100010101011110, *P* = 110010, *k* = 3.

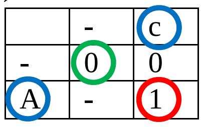
Solution:

Create a table and initialize it as follows:

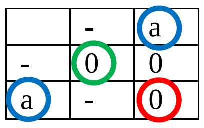
		c	c	a	c	b	A	a	b	c	c	a	c	c	b	c	a	b	c	c	c
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	-																				
b	-	-																			
c	-	-	-																		
b	-	-	-	-																	
a	-	-	-	-	-																

The character '-' represents undefined value. Fill the table row by row using the following rules:

(A)



(B)



Case (A) - The value in a red circle in a i-th row and j-th column is computed as the value in the top-left neighbor (green circle) increased by 1 if the characters in the i-th row and j-th column (blue circles) differ.

Case (B) - If the these characters are the same, copy the value from the green circle to the red circle without a change.

Filled table:

		c	c	a	c	b	a	a	b	c	c	a	c	c	b	c	a	b	c	c	c
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	-	1	1	0	1	1	0	0	1	1	1	0	1	1	1	1	0	1	1	1	1
b	-	-	2	2	1	1	2	1	0	2	2	2	1	2	1	2	2	0	2	2	2
c	-	-	-	3	2	2	2	3	2	0	2	3	2	1	3	1	3	3	0	2	2
b	-	-	-	-	4	2	3	3	3	3	1	3	4	3	1	4	2	3	4	1	3
a	-	-	-	-	-	5	$\textcircled{2}$	3	4	4	4	$\textcircled{1}$	4	5	4	$\textcircled{2}$	4	3	4	5	$\textcircled{2}$

In the last row we find values not greater than k = 2 (violet circles). They correspond to end-positions of substrings of length 5 whose Hamming distance to the pattern *abcba* is at most k = 2. One such an substring is highlighted by the orrange rectangle.

* 2. Find in the text T all occurences of the substrings which Levenshtein distance form the pattern P is at most *k*. Apply the dynamic programming approach.

a) *T* = aacacacbaabbbcbbcacc, *P* = cbbba, *k* = 3.

b) *T* = 0100111010000101010111100 , *P* = 11100, *k* = 1

Solution:

The algorithm is similar. It slightly differs in the initialization, values computation and the result interpretation.

Initialize the table as follows:

		c	c	a	c	b	a	a	b	c	c	a	c	c	b	c	a	b	c	c	c
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1																				
b	2																				
c	3																				
b	4																				
a	5																				

Values of the table are computed by the exactly same rules as in the case of computing Levenshtein distance for two words, namely...???

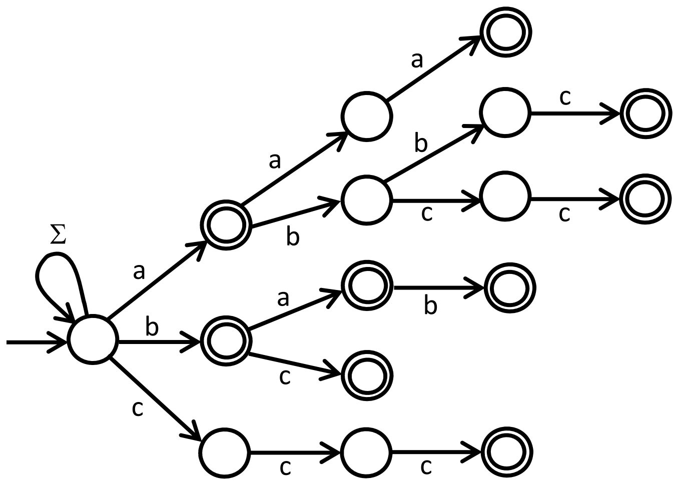
Filled table:

		c	c	a	c	b	a	a	b	c	c	a	c	c	b	c	a	b	c	c	c
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	1	1	0	1	1	0	0	1	1	1	0	1	1	1	1	0	1	1	1	1
b	2	2	2	1	1	1	1	1	0	1	2	1	1	2	1	2	1	0	1	2	2
c	3	2	2	2	1	2	2	2	1	0	1	2	1	1	2	1	2	1	0	1	2
b	4	3	3	3	2	1	2	3	2	1	1	2	2	2	1	2	2	2	1	1	2
a	5	4	4	3	3	2	1	2	3	2	2	1	2	3	2	2	2	3	2	2	2

The last row contains only 3 values greater than k = 3 (see the red crosses). This means that all the other positions are ends of substrings whose Levenshtein distance from ‘abcba’ is at most k = 3. For example the highlighted substring has Levenshtein distance 1.

* 3. Construct a non-deterministic automaton which detects each word of the set M in a text over the alphabet A .
a) A = {a, b, c}, M = {a, b, ba, bc, aaa, bab, ccc, abbc, abcc } .
b) A = {0, 1}, M = {10, 11, 101, 111, 1011, 1101, 10001, 10011, 10111, 11101, 11111 }.

Solution:
The automaton has a tree structure where the root is the initial state. Each word in M adds a chain of states and transions which starts in the root. Transitions are labeled by the characters of the word. The last state of the chain is marked as accepting. If some words from M have the same prefix then they share states and transions representing this prefix. Finally, a loop labeled by the input alphabet is added to the initial state.

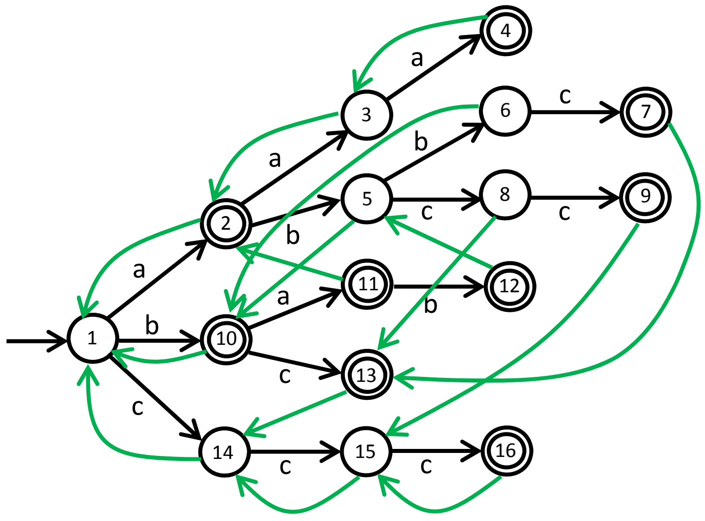


* 4. Construct a deterministic automaton which detects each word of the set M in a text over the alphabet A. M and A are the same as in the previous problem.

Solution: Deterministic dictionary automaton is based on the non-deterministic automaton in the previous assignment. States remain the same. The loop in the initial state is removed. Moreover, we have to define transitions

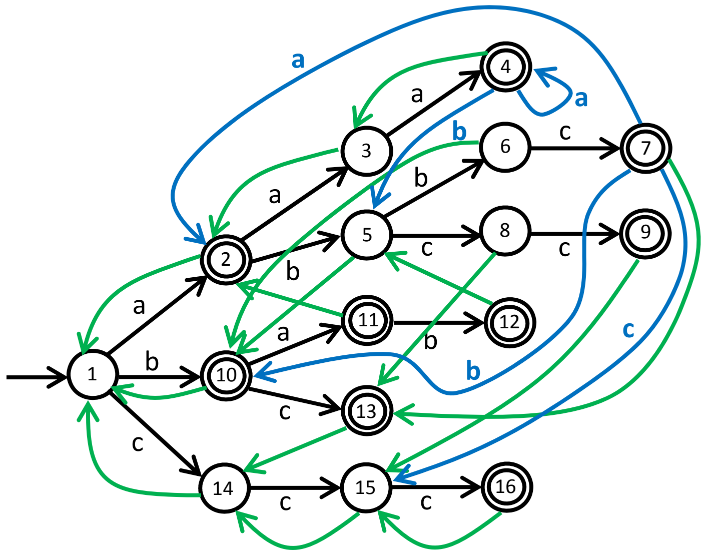
for each state and each input symbol (in our case, the input symbols are ‘a’, ‘b’ and ‘c’). We proceed by the following algorithm:
First of all, for each state (reachable from the initial state by reading a word *u*) we find a state reachable in the non-determinisitc automaton by reading the longest proper suffix of word *u*.

All such states are marked by the green edges in the image bellow.



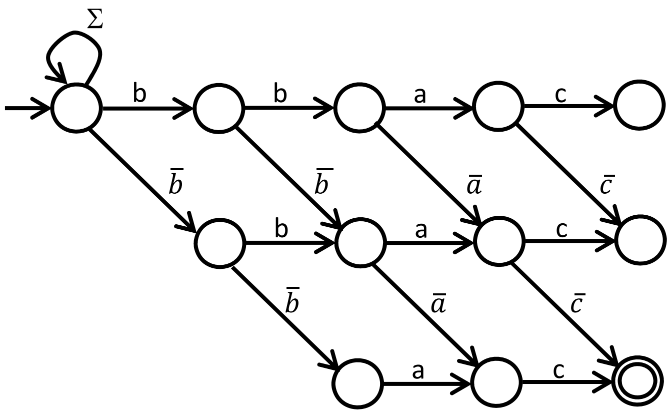
For example, state 7 is reachable from the intial state after reading abbc. The longest proper suffix of abbc is bbc, however, there is no path from the initial state for this suffix. The second longests proper suffix is bc. This word is represented by state 13 since state 13 is reached from the initial state by reading bc. From this reason, there is the green edge going from state 7 to state 13. Another example: state 8 represents word abc whose longest proper suffix is bc, hence there is a green edge going from state 8 to state 13.

We are ready now to define all the needed transions for the determinisitc automaton. Let σ denote the transion function of the nondeterministic automaton and let δ denote the transition function of the constructed determinisitc automaton. Assume we have a state *s* and a character *d*, for which a transition from *s* has not been defined yet. To determine value δ(*s*,*d*), we follow the green edge from *s* (assume it goes to state *p*). If σ(*p*,*d*)=*r*, we define δ(*s*,*d*):=*r*. If σ(*p*,*d*) is empty (i.e. there is no transition from *p* after reading *d*), we follow the green edge from p (assume it goes to a state *t*). Then we again check whether σ(*t*,*d*) is non-empty. If so, we define δ(*s*,*d*):= σ(*t*,*d*). Otherwise we continue in following the green edges. If we arrive to the initial state without defining δ(*s*,*d*), then δ(*s*,*d*) equals the initial state.
Example: Consider state 7 and character *c*, for which there is no transition from state 7 in the nondeterministic automaton. We follow the green edge which goes to state 13. Neither this state has a transion for *c*, thus we follow the green edge to state 14. Here we have a transition to state 15. From this reason, we define δ(*7*,*c*)=15.
Few other transitions added to the deterministic automaton are illustrated by the blue edges in the image bellow.

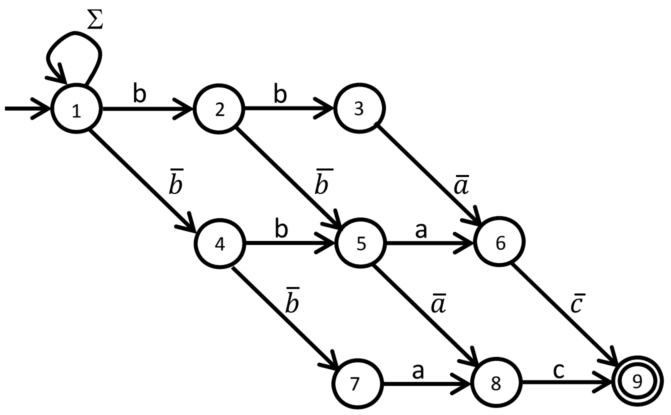


* 5. Use the method of bit parallelism to construct the tables for simulation of the text search automaton which detects in the text *t* all occurences of substrings with Hamming distance *k* from the pattern *p*.
a) *t* = abcbcaaccbbaa, *p* = bbac, *k* = 2,
b) *t* = accbbaaabcba, *p* = acbb, *k* = 2.

We first construct a nondeterministic automaton computing Hamming distance 2 with respect to the given pattern *p*.



We further simplify the automaton by removing unreachable states.



To simulate this automaton using the bit parallelism technique we need bit vectors of length 9. The set of initial states is decribed by vector *I*=<1,0,0,0,0,0,0,0,0>. The table of transitions consists of bit vectors for each state *s* and a character *d* where the bit vector represents states reachable from *s* after reading *d*. For example, for state 1, the table contains vector <1,0,0,1,0,0,0,0,0>, <1,1,0,0,0,0,0,0,0> and <1,0,0,1,0,0,0,0,0> for character *a*, *b* and *c*, respectively. Finally, vector *F*=<0,0,0,0,0,0,0,0,1> represents the set of accepting states.

An example, how to simulate one step of the nondeterministic automaton: Assume that *a* is the first input symbol. After reading it, the set of active states is <1,0,0,1,0,0,0,0,0>. Assume that *c* is the second input character. States reachable from state 1 after reading *c* are represented by <1,0,0,1,0,0,0,0,0>, states reachable from state 4 after reading *c* are represented by <0,0,0,0,0,0,1,0,0>. We compute bitwise OR of these two vectors to obtain the next set of active states, etc.
To check whether a set of active states represented by vector *V* contains an accepting state, we compute *V* AND *F*.

6. We define the reduced Levenshtein distance of strings X and Y to be the minimum number of edit operations which transform X to Y (or Y to X). Only edit operations Insert and Delete are considered in this definition. Describe an algorithm which will apply the Dynamic programming approach to compute the reduced Levenshtein distance between X and Y.

Solution:

We modify the dynamic programming algorithm that computes the standard Levenshtein distance.
There is no change if there are matching characters (green circles), we copy the top-left neighbouring value.

	-	A	T
-	0	1	2
A	1	0	
N	2		

However if the characters do not match, we cannot apply REWRITE, from this reason we ignore the top-left value when computing the next value to fill.

	-	A	T
-	0	1	2
A	1	0	1
N	2		

	-	A	T
-	0	1	2
A	1	0	1
N	2	1	

	-	A	T
-	0	1	2
A	1	0	1
N	2	1	2

7. Describe an algorithm which will detect in a text all substrings which reduced Levenshtein distance from the given pattern is minimum possible. See the definition of the reduced Levenshtein distance in the previous problem.

Solution:
The algorithm is similar to that one described for the assignment 2. The only difference is that we apply the modification prom assignement 7 when filling the table.

8. We say that the Insert-distance of two strings X and Y is exactly *k* if and only if *k* is the minimum number of operations Insert applied to only one of the strings and which will make X and Y identical. If the identity cannot be achieved we define the Insert-distance of X and Y to be positive infinity. Construct a NFA which will accept all strings which Insert-distance from a given pattern is at most *k*.

9. Describe an algorithm based on the Dynamic programming approach which will detect in a text all occurences of strings which Insert-distance from a given pattern *p* is exactly *k*. See the definition of the Insert-distance in the previous problem.

10. Describe the changes which must be applied in the previous problem to obtain a similar algorithm which uses Delete-distance instead of Insert-distance.