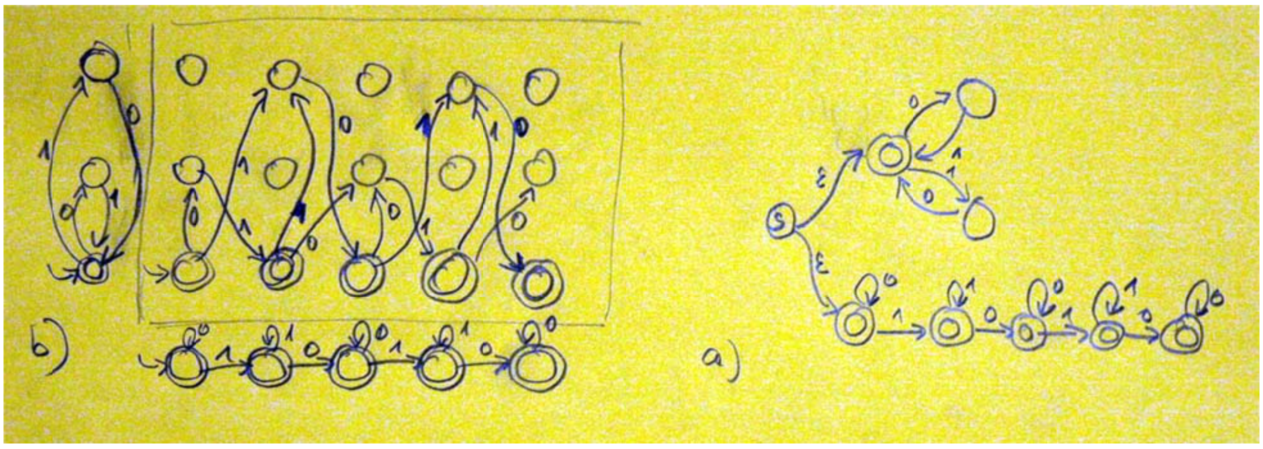
* 1. There are two languages L_1 and L_2 over the alphabet {0, 1}. Words of L_1 are described by the regular expression 0*1*0*1*0*, words of L_2 are described by the regular expression (01+10)*. Construct a finite automaton which
a) accepts language $L_1 \cup L_2$, b) accepts language $L_1 \cap L_2$.

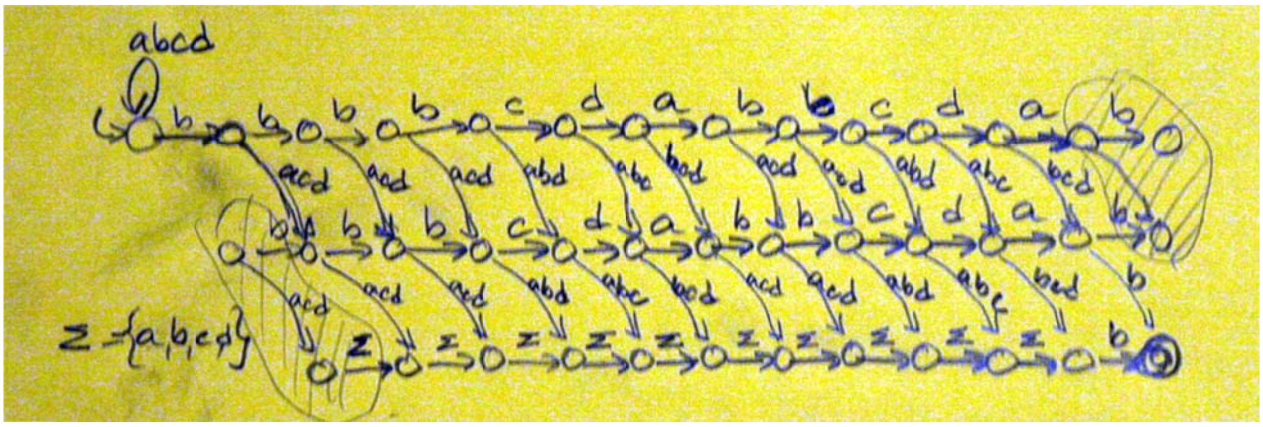


* 2. The automaton A_1 resp. A_2 accepts the language L_1 resp. L_2 over alphabet A, |A| = *k*. Each of A_1 and A_2 has *n* internal states. We have to determine whether $L_1 \cap L_2$ is finite. What is the asymptotic complexity of the task?

Construct an automaton A3 accepting $L_1 \cap L_2$. This automaton has *n*^2 states and, as it is an NFA, it can have up to (*n*^2)^2 = *n*^4 transitions. The language accepted by the automaton is finite iff there is no loop on any path from the start state of A3 to any of the final states of A3.
(*) In other words, when we have to reduce the transition diagram of A3 to the states which are accessible from the start state of A3 and simultaneously from which some final state of A3 is reachable. These states can be found in two BFS searches -- all states accessible form the strat state can be found in one BFS and the all states from which some end state is accessible can be also found in one BFS which runs from all final states in the reverse direction of the transitions.
The resulting transition diagram has to be a directed graph without any loops -- Directed acyclic graph (DAG).
How to check if a directed graph is a DAG?
A) Run DFS and assign each node its open time. If the search investigates an edge (x, y) where both x and y are open and openTime(x) < openTiume(y), then the graph contains a cycle. If no such situation happens, the graph is acyclic.
B) Tarjan algorithm of detecting strongly connected components uses the idea of A). If the number od strongly connected components is equal to the number of nodes in the graph then the graph is acyclic. Otherwise it contains a strongly connected component of nontrivial (> 1) size and therefore it contains a cycle.
Both A) and B) can be verified in time O(|V| + |E|).
Reduction of the automaton in paragraph (*) and also the acyclicity check can be done in time can be done in time O(|V| + |E|), whwrw |E| is the number of edges in automaton A3. As A3 has n^2 nodes, the number of edges in A3 is O((n^2)^2). Thus, the whole verification if A3 accepts a finite language can be acomplished in time O(n^4).

The analysis holds for a general NFA. If, on the other hand, both A1 and A2, and therefore also A3, are DFA, than there can be at most |A| = *k* outgoing edges from each state of A3. The complexity of reach of BFS, DFS, Tarjan algorithm run on A3 would be then
O(n^2 + *k*×n^2) = O(*k* × n^2)
which would be also the complexity of the whole verification process.

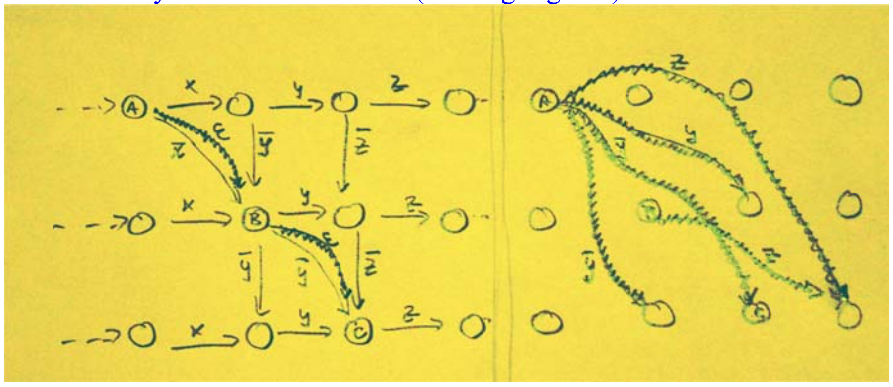
* 3. The alphabet is {*a*, *b*, *c*, *d*}. A critical string over this alphabet issuch a string which begins and ends with the symbol *b* and moreover its Hamming distance from the pattern *abbbcdabbcdab* is greater than 2. Construct a finite automaton which will detect in a text the occurences of all critical strings.



The highlighted areas are unecessary, may be removed from the automaton.

* 4. There is a finite automaton A which can detect in a text all its substrings which Levenshtein distance from a given pattern *p* is less than a given value *k*. Automaton A contains some epsilon-transitions. Construct an example of this automaton for $|p| = 6$ and $k = 3$. Remove all epsilon-transitions and write down the resulting automaton.

The shape of the automaton will be analogous to the shape on slide 21 of the lecture slides. There will be two more vertical columns of states and only three rows of states, corresponding to maximum distance 0, 1, 2. The removal of epsilon transitions is in the picture, each pair of consecutive epsilon-transitions (highlighted) is removed and substituted by five new transitions (also highlighted).



* 5. We define reduced Levenshtein distance of the words *v*, *w* (over alphabet A) to be the minimum number of edit operations which will transform word *v* into word *w*. In this case, we consider as edit operations only the operations Insert and Delete. Construct a NFA without epsilon-transitions which will be able to detect in a text any string which reduced Levenshtein distance form the pattern *abaabacc* is exactly 2.

Hint only:
We have to be careful because the string distance from the pattern has to be *exactly* 2.
There are some cases. To avoid confusion, construct the final automaton as a union of five automata.

Delete followed by insert at the same place can be counted as a single rewrite operation (composed of 2 operations).
The same applies to insert operation followed by delete immediately before or after the
inserted character. So, it is easier to consider rewrite operations separately.
The first automaton will accept only the pattern modigfied by one rewrite operation.
The second and the third automaton will correspond to application of two consecutive operations of the same type -- two times insert (second automaton) and two times delete (third automaton).
The fourth automaton will correspond a delete operation followed by the insert operation. In this automaton, some delete operations in substring *aa* and *cc* *cannot* be followed by the insert operation. We want to avoid the situation when e.g. first *a* in *aa* is deleted and then another *a* is inserted past the remaining *a*. This sequence of operations would lead to the string identical with the original pattern, while we need its distance to be 2.
The fifth automaton would be analogous to the fourth one only the order of operations would be reversed, first insert and then delete.

6. Let us denote by symbol d(x, y) the Levenshtein distance between words x and y. There are three words u, v, w and it holds that d(u, v) = d_1, d(v, w) = d_2. What are possible values of d(u, w) in relation to d_1 and d_2? The alphabet is the same for all words involved.

The Levenshtein distance is a distance in the technical sense of the word (a metric), that is, triangle inequality holds for the distances between any three words u, v, w: d(u, v) + d(v, w) ≤ d(u, w).

Obviously if sequence of operations S1 turns u into v and sequence of operations S2 turns v into w then the concatenated sequence S1.S2 turns u into w. However, there also might be shorter sequences of operations which turn u into w. For example:
u = abcdefgh
v = defghabc
w = bcdefgha
it holds, d(u, v) = 3, d(v, w) = 2, d(u, w) = 1.

7. Construct a NFA which will detect in a text any element of the set of all continguous substrings of the pattern P = *abcdefghijklmnopqrstuvwxzy*.

List all continguous substrings, there are 26+25+ ... 2 + 1 = 351 of them. For each substring construct an automaton which accepts exactly this substring and no other word. Create an automaton which accepts the union of all 351 substrings. The union automaton will have an additional start state S and an epsilon transition from S to each start state of 351 automatons. Add the self-loop transition from S to S labeled by the whole alphabet *abcd...xzy*.
The number of states of this automaton is 1×26 + 2×25 + 3×24 + ... + 25×2 + 26×1 + 1 = 1639.
The following method is more convenient (smaller number of states). Create NFA which accepts each unempty prefix of P. Its structure is identical to the structure of NFA which accepts just P and nothing else (slide 10 in lecture 7), the only difference is that all states, except for the start state, are final. Now, for each transition (state α) --> (state β) labeled by character γ add a transition from start state to state β labeled by character γ. Do not forget the self loop at the start state. This automaton has only 27 states and its diagram has 54 transition arrows.

8. Let us denote by the symbol HD(v, w) resp. LD(v, w) the Hamming resp. Levenshtein distance between the words v and w over the alphabet A. Decide which of the following may be true for some words v and w which length is at least 5.
a) HD(v, w) < LD(v, w), b) HD(v, w) = LD(v, w), c) HD(v, w) > LD(v, w).

The length of both words v and w must be the same, otherwise their Hamming distance is undefined. Whenever some sequence of rewrite operations transform v to w we can consider this sequence to be also a sequence of *edit* operations which are used in Levenshtein distance definition. Therefore, Hamming distance between v and w will be always bigger or equal to Levenshtein distance between v and w. Examples:
a) Not possible
b) v = abcdefgh, w = bbcdefgh, HD(v, w) = LD(v, w) = 1
c) v =abcdefgh, w =bcdefgha, HD(v, w) = 8, LD(v, w) = 2 (operations: v.delete(0), v.insert('a', 7))

9. Write down all words over alphabet {a, b, c} which Levenshtein distance form the word *aba* is exactly a) 1, b) 2.

a) After Delete (delete first, second, third character): ba, aa, ab.
After Rewrite (rewrite first, second, third character): bba, cba, aaa, aca, abb, abc.
After Insert (insert at first, second, third, fourth position):
aaba, baba, caba, ~~aaba,~~ abba, acba, abaa, ~~abba~~, abca, abaa, abab, abac

b) The total number of cases is quite big if processed by hand -- the single word aba produced 3+6+12 = 21 cases of operations at various positions when considering distance 1. To obtain all words in distance 2 we have to aply all operations at all positions to all words obtained in problem 9 a). This leads to probably more than hundred cases, so it is better to generate the result programmatically. Unfortunately, there is no quick obvious way to do it in few lines of code. The most straightforward approach would be, in my opinion, to generate recursively all possible words up to length 5, download a function which computes Levenshtein distance (https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance) and print those words which distance from *aba* is less or equal to 2.

```
/**  Source:
```

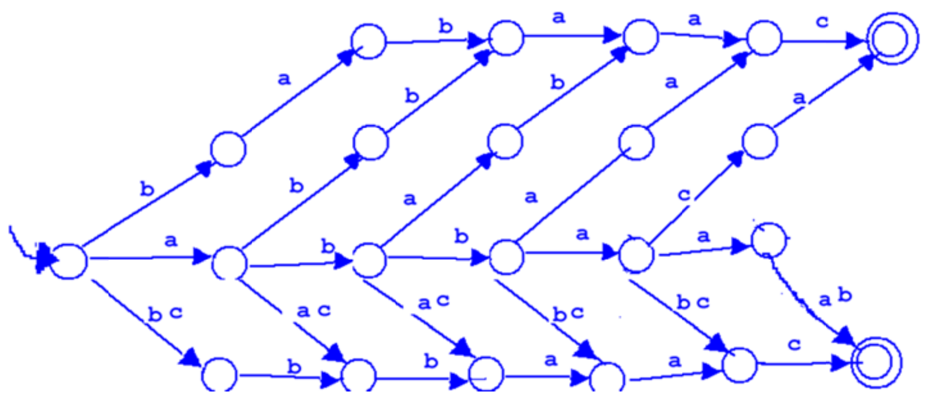
```
https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance#C++
*/
unsigned int edit_distance(const std::string& s1, const std::string& s2) {
      const std::size_t len1 = s1.size(), len2 = s2.size();
      std::vector<std::vector<unsigned int>> d(len1 + 1, std::vector<unsigned int>(len2
+ 1));
      d[0][0] = 0;
      for(unsigned int i = 1; i <= len1; ++i) d[i][0] = i;
      for(unsigned int i = 1; i <= len2; ++i) d[0][i] = i;
      for(unsigned int i = 1; i <= len1; ++i)
            for(unsigned int j = 1; j <= len2; ++j)
                       // note that std::min({arg1, arg2, arg3}) works only in C++11,
                       // for C++98 use std::min(std::min(arg1, arg2), arg3)
                       d[i][j] = std::min({ d[i - 1][j] + 1, d[i][j - 1] + 1,
                                 d[i - 1][j - 1] + (s1[i - 1] == s2[j - 1] ? 0 : 1) });
      return d[len1][len2];
}

/**
Simple recursive function generating all words within the given edit distance
from the original word.
*/
void gener( string alphabet, string origWord, string generatedWord, int distance ){
  if( generatedWord.size() > origWord.size() + distance ) return;

  generatedWord.push_back(' '); // make space for another char
  // try all characters at the current end of the generated word and recurse
  for( int i = 0; i < alphabet.size(); i++ ){
    generatedWord[generatedWord.size()-1] = alphabet[i];
    if( edit_distance(origWord, generatedWord) <= distance )
       cout << generatedWord << ", ";
    gener( alphabet, origWord, generatedWord, distance );
  }
}
```

The words are generated by the call:

```
  gener( "abc", "aba", "", 2 );
```

There list generated by the call contains 128 words sorted in ascending lexicographical order:
(a, aa, aaa, aaaa, aaab, aaaba, aaac, aab, aaba, aabaa, aabab, aabac, aabb, aabba, aabc, aabca, aac, aaca, aacba, ab, aba, abaa, abaaa, abaab, abaac, abab, ababa, ababb, ababc, abac, abaca, abacb, abacc, abb, abba, abbaa, abbab, abbac, abbb, abbba, abbc, abbca, abc, abca, abcaa, abcab, abcac, abcb, abcba, abcc, abcca, ac, aca, acaa, acab, acaba, acac, acb, acba, acbaa, acbab, acbac, acbb, acbba, acbc, acbca, acc, acca, accba, b, ba, baa, baaa, baaba, bab, baba, babaa, babab, babac, babb, babba, babc, babca, bac, baca, bacba, bb, bba, bbaa, bbab, bbaba, bbac, bbb, bbba, bbc, bbca, bc, bca, bcaba, bcba, ca, caa, caaa, caaba, cab, caba, cabaa, cabab, cabac, cabb, cabba, cabc, cabca, caca, cacba, cb, cba, cbaa, cbab, cbaba, cbac, cbb, cbba, cbc, cbca, cca, ccaba, ccba).

10. There is a pattern p and a string q. The string q was obtained from q by applying exactly one of the two operations:
SWAP (= swapping of two immediately neigbouring symbols)
REWRITE (= substitution of a single symbol by another symbol of the alphabet)
Construct a NFA which will detect in a text any occurence of q when $p = abbaac$ and alphabet is $\{a, b, c\}$.

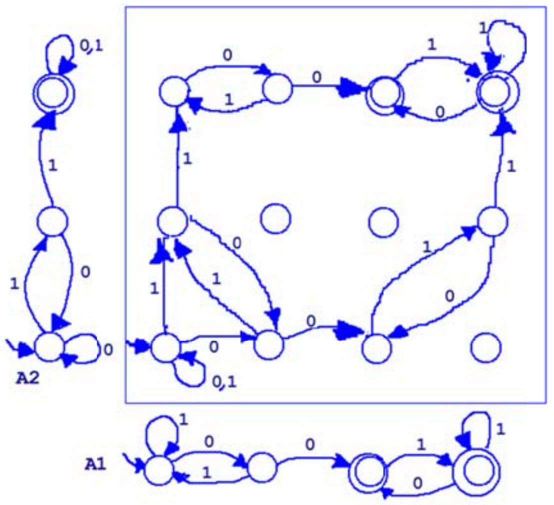


The upper diagonal transitions correspond to SWAP operations, the lower diagonal transitions correspond to REWRITE operations. There should be also a self-loop labeled by *a*,*b*,*c* at the leftmost (start) state.

11. Alphabet A = {0, 1}. The language L_1 contains all words over A which contain the continuous substring 00 exactly once. The language L_2 contains all words over A which contain the continuous substring 11 at least once. Construct a finite automaton which will detect in a text over A all occurences of all words of the language
a) $L_1 \cap L_2$, b) $L_1 \cup L_2$.

a) The solution is the automaton inside the rectangle in the picture, automata A1 and A2 recognize languages L1, L2, respectively.

b) Take A1 and A2 in the picture, add a new start state with the self-loop labeled by 0,1 and add epsilon transitions from the new start state to the start states of A1 and A2.



12. There are two finite sets M_1 and M_2 of words over alphabet A. Language L consists of all words *w* over A for which holds that at least one prefix of w is in set M_1 and at least one suffix of w is in M_2. The whole word is considered to be its own prefix and also its own suffix. Describe an algorithm which will construct an automaton accepting L. Describe a concrete example for $|M_1| = |M_2| = 2$.

Construct an automaton A1 which accepts any word whose prefix is in M1. Automaton A1 is a union of automata. Each automaton in the union corrssponds to one word in M1. It accepts a string whose prefix is this particular word. Similarly, construct an automaton A2 as a union of automata each of which accepts a word whose suffix is one of words in M2. Remove epsilon-transitions from A1 and A2. Construct Automaton A3 as an automaton which accepts the intersection of languages accepted by A1 and A2.
Let M1 = { ab, cde }, M2 = {xyz, pqr}

13. An alphabet A, a pattern *p* over A and a fixed positive integer *k* are given. Describe an algorithm which will print out all words over A which Hamming distance from ***p*** is exactly *k*. What is the asymptotic complexity of this algorithm?

First, let us generate the sets of all positions on which the chages will take place. There are Comb(|*p*|, *k*) such sets and each can be generated in $\Theta(k)$ time. A character at a particular position can be changed (=rewritten) in |A|–1 ways. Let us fix a particular subset of *k* positions where the changes will occur. There are (|A|–1)^*k* ways to change the characters at each of those *k* positions.
As there are Comb(|*p*|, *k*) subsets, the total number of changes is Comb(|*p*|, *k*) × (|A|–1)^*k* . Take in account the $\Theta(k)$ time of generating the particular subset of *k* positions. The total time would be then
O(*k* × Comb(|*p*|, *k*) × (|A|–1)^*k*) .

14. An alphabet A, a pattern *p* over A and a fixed positive integer *k* are given. Describe an algorithm which will print out all words over A which Levenshtein distance form *p* is at most *k*. What is the asymptotic complexity of this algorithm?

The simplest way, similarly to the solution of problem 9, is to generate all words over alphabet A which length is in range [|*p*|–k, |*p*|+*k*], check the distance of each generated word from the pattern *p* and print out those words which distance from *p* is at most *k*. The number of all words with length |*p*|+*j* is

|A| ^ (|*p*|+*j*).

The complexity of computing Levenshtein distance between *p* and a word of length |*p*|+j is

|*p*| × (|*p*|+j) = O((|*p*|+j)^2).

The overall complexity is thus

O(sum(*j* = –k..k, (|*p*|+j)^2 × |A| ^ (|*p*|+*j*))).

We can expect that the biggest term in the sum corresponds to *j* = *k*, so let us substitute each term in the sum by

(|*p*|+*k*)^2 × |A| ^ (|*p*|+*k*) .

Then, we arrive to (somewhat cruder) complexity upper bound

O(k × (|*p*|+*k*)^2 × |A| ^ (|*p*|+*k*)).