CSMTSP Text Searching and Processing - Solutions

1. (a) See last pages.

(b) The Aho-Corasick is a deterministic finite automaton together with a failure function (link), it is defined by a six-tuple $(Q, \Sigma, g, f, q_0, F)$ where:

1. Q is a set of states.
2. Σ is a finite input alphabet.
3. $g : Q \times \Sigma \to Q \cup \{fail\}$ is the forward transition.
4. $f : Q \to Q$ is the failure transition (link).
5. q_0 is the initial state.
6. F is a set of final states and is a subset of Q.

The failure link has the following property:

Suppose that in the transition graph of the g function, state q_i represents the string s_i and state q_j represents the string s_j. Then, $f(q_i) = q_j$ if and only if s_j is the longest proper suffix of s_i that is also a prefix of some keyword.

(c) The search procedure of the AC automaton is defined by the following procedure for moving from one state to another according to the current symbol being read from the text. Let $p \in Q$ be a state of the automaton, f be the failure link and $\delta \in \Sigma$ be a symbol of the alphabet.

Next_State(p, δ)
if $g(p, \delta)$ is defined **then** return $g(p, \delta)$
else if $f(p)$ is defined **then** return Next_State$(f(p), \delta)$
else return q_0 (initial state)

(d) Preprocessing the AC automaton for a set of strings X has a time and space complexity of $O(|X| \log |\Sigma|)$. The time complexity for the search procedure on a text of length n is $O(n \log |\Sigma|)$. The maximal time spent on a single symbol of the text during the search procedure is $O(\max\{|x| : x \in X\})$.

2. (a) Assume that a mismatch occurs between the character $x[i] = \delta$ of the pattern and the character $t[pos + i - 1] = \tau$ of the text during an iteration of the Boyer-Moore algorithm at position pos. Then, $t[j+1 \ldots j-i+m] = x[i+1 \ldots m] = u$ and $t[j] \neq x[i]$. The good-suffix rule consists in aligning the suffix u of x with its rightmost occurrence in x that is preceeded by a character different from $x[i]$. For example,

$$\begin{aligned} t &= \ldots a\ a\ b\ a\ b\ a\ \ldots \\ x &= \ldots b\ a\ b\ a\ b\ a \\ x &= \ldots c\ a\ b\ a\ b\ a\ \ldots \end{aligned}$$

If no such rightmost occurrence of u exists then the shift consists in aligning the longest suffix v of u with a matching prefix of x. For example,

$$\begin{aligned} t &= \ldots a\ a\ b\ a\ b\ a\ \ldots \\ x &= \ldots b\ a\ b\ a\ b\ a \\ x &= \ldots \qquad\quad b\ a\ \ldots \end{aligned}$$

The table D is defined as follows:
$D[i] = d(x[i+1 \ldots m])$, for $i = 0, \ldots, m$, where $d(u) = \min\{|z| > 0 | (x \text{ suffix of } uz) \text{ or } (\tau uz \text{ suffix of } x \text{ and } \tau u \text{ not suffix of } x, \text{ for } \tau \in \Sigma)\}$.

(b) The bad-character rule consists in aligning the text character $t[j]$ with it's rightmost occurrence in the pattern x. For example,

$$\begin{aligned} t &= \ldots a\ c\ d\ e\ f\ g\ \ldots \\ x &= \ldots b\ c\ d\ e\ f\ g \\ x &= \ldots a\ \ldots\text{contains no } a \end{aligned}$$

If $t[j]$ does not appear in the pattern x then the left end of the pattern is alighned with the next character to the right of $t[j]$, i.e. $t[j+1]$ For example,

$$\begin{aligned} t &= \ldots a\ c\ d\ e\ f\ g\ \ldots \\ x &= \ldots b\ c\ d\ e\ f\ g \\ x &= \quad\ \ldots\text{contains no } a \end{aligned}$$

procedure Compute_DA(x: string, m integer);
begin
  for all $\delta \in \Sigma$ **do** $DA[\delta] = m$;

bf for $i := 1$ **to** $m - 1$ **do** $DA[x[i]] = m - i$;
end

(c) **procedure** BM(x, t: strings, m, n: integers);
begin
$pos := 1$;
while $pos \leq n - m + 1$ **do begin**
$i := m$;
while $i > 0$ **and** $x[i] = t[pos + i - 1]$ **do** $i := i - 1$;
bf if $i = 0$ **then** writeln('x occurs in t at position', pos) ;
$pos := pos + \max(D[i], DA[t[pos + i - 1]] - m + i)$;
end
end

The worst case running time of the BM algorithm is $O(nm)$, for example: pattern $x = aaaaaaa$ and text $t = a^n$. The best case running time of the BM algorithm is $O(n/m)$, for example: pattern $x = aaaaaab$ and text $t = a^n$.

(d) If the table R_next is given then the table D can be implemented in the following way:

for $j = 1$ **to** $m - 1$ **do begin**
$i := R\_next[j]$;
$D[i] := j$;
end
end

3. (a) **procedure** Compute_KMP_next(x: string, m: integer);
begin
$KMP\_next[1] := 0; j := 0$;
for $i := 1$ **to** m **do begin**
while $j > 0$ **and** $x[i] \neq x[j]$ **do** $j := KMP\_next[j]$;
$j := j + 1$;
if $i = m$ **or** $x[i + 1] \neq x[j]$ **then** $KMP\_next[i + 1] := j$
else $KMP\_next[i + 1] := KMP\_next[j]$;
end
end

(b) KMP_next table for the pattern $x =$ `ababbabaab`.

	1	2	3	4	5	6	7	8	9	10	11
x	a	b	a	b	b	a	b	a	a	b	
KMP_next	0	1	0	1	3	0	1	0	4	1	3

(c) The search procedure of the KMP algorithm is the same as the one for the MP algorithm except that we use the table KMP_next for the shift instead of MP_next.

```
procedure KMP(x, t: strings, m, n: integers);
begin
    i := 1; j := 1 ;
    while j ≤ n do begin
        while (i = m + 1) or (i > 0 and x[i] ≠ t[j]) do
            i := KMP_next[i]
        end
        i := i + 1; j := j + 1 ;
        bf if i = m + 1 then writeln('x occurs in t at position', j − i + 1) ;
    end
end
```

(d) See last pages.

(e) The delay on a two-letter alphabet is constant. The delay on a three-letter alphabet is $O(\log m)$.

4. (a) The Hamming distance of a two strings x and y is the number of mismatches allowed between the two. For example, if $x = abcdefgh$ and $y = bbcaefdh$ then the Hamming distance of x and y is 3 since we have 3 mismatches occurring at position 1, 4 and 7.

The Levenshtein distance is an edit distance that allows insertion, deletion and substitutions of one character at a time each having a unit cost. For example, if $x = abacadba$ and $y = bcdadaba$ then x and y have a Levenshtein distance of 4. Since we have a mismatch (substitution) at position 1, an insertion at position 2, another mismatch at position 5, and a deletion at position 7 as illustrated below.

1 2 3 4 5 6 7 8 9
a b a c a d b a
b c b a d a b a

(b) Build a $(n+1) \times (m+1)$ table C where we place the string y at the top and the string x on the left. Initialize all entries $C[i,0] = C[0,j] = 0$. Now proceed to compute $C[i,j]$ by taking the minimum value of:
$C[i,1,j-1] + substitute(x_i, y_j)$
$C[i-1,j] + delete(x_i)$
$C[i,j-1] + insert(y_j)$
according to the costs of the operations: substitute, delete and insert.

(c) By using dynamic programming the above procedure can be extended to that of computing a shortest source-to-sink path in an edge-weighted grid of a directed acyclic graph once the table C has been computed. This will result in an optimum edit script for transforming x into y with a minimum total cost.

(d) Let $cost(del) = cost(ins) = 1$, and $cost(subst) \leq cost(del) + cost(ins)$. Then no edit script will use the substitution operation. The pairs of matching symbols preserved in an optimal edit script constitute a longest common subsequence of x and y. If s is the length of a longest common subsequence between x and y, then on a minimum edit distance path in the grid C, because no substitution is made then we either go down of up. This results in the minimum edit distance e being the sum of the lengths of x and y minus twice the length of the longest common subsequence between x and y.

5. (a) See last pages.

(b) Let $y \in \Sigma^*$ be a string of length n. The suffix trie of y is a digital search tree representing the suffixes of y. Each node of the trie is identified with a substring of y, each terminal node (leaf) is identified with a suffix of y and finally each edge is labelled with a symbol of y.
The suffix tree T_y of y is a tree obtained from the suffix trie of y by deleting all nodes having outdegree 1 which are not terminal and by labelling the arcs with substrings of y instead of symbols. The construction of a suffix tree relies on the computation of the suffix link function S_y and is defined as follows: if a node p is identified with a substring $av, a \in \Sigma, v \in \Sigma^*$ then $S_y(p) = q$ which is the node identified with v.

(c) Recall that in the construction of a suffix tree we insert suffixes of decreasing length. By definition, if p is a fork then p has outdegree two at least or is a terminal node with outdegree one and represents the head of a suffix. Denote av as the head of at least two suffixes ending at node p, and let $S_y(p) = q$, where q is the node identified with v. Then we know that in some previous iteration the suffix starting with v was inserted in the tree and since at least two tails split at p then similarly they must be a split at q or else it is a terminal state with outdegree one.

Let T_y be the suffix tree associated with the string y of length $n+1$. Suppose that T_y is a complete binary tree, thus branching by two at each level. This will maximize the number of internal nodes in T_y. By proceeding by induction we can see that the two subtrees of T_y of size $(n+1)/2$ also have $n/2$ internal nodes. i.e. the sequence: $1, 1+2 = 3, 3+4 = 7, 7+8 = 15, 15+16 = 31, \ldots$ Hence for a string of length n, there are at least n and at most $2n$ nodes in the tree.

(d) Main steps:

1. Goto parent r of p
2. Use the suffix link of r
3. Go down the tree by the label of (r, p)

The running time of step 3 above is proportional to the number of arcs along the path and not to the length of the label.

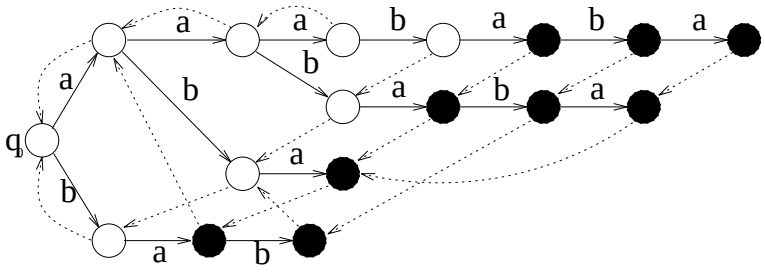

Figure 1: **Question 1. (a)** The Aho-Corasick automaton for the strings: `aaababa`, `aababa`, `aba`, `bab`. Black nodes are terminal states and failure links are dotted lines.

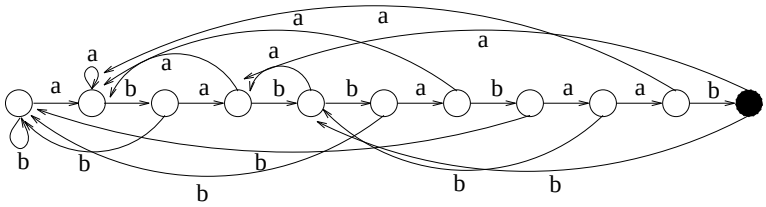

Figure 2: **Question 3. (d)** $SMA(\texttt{ababbabaab})$

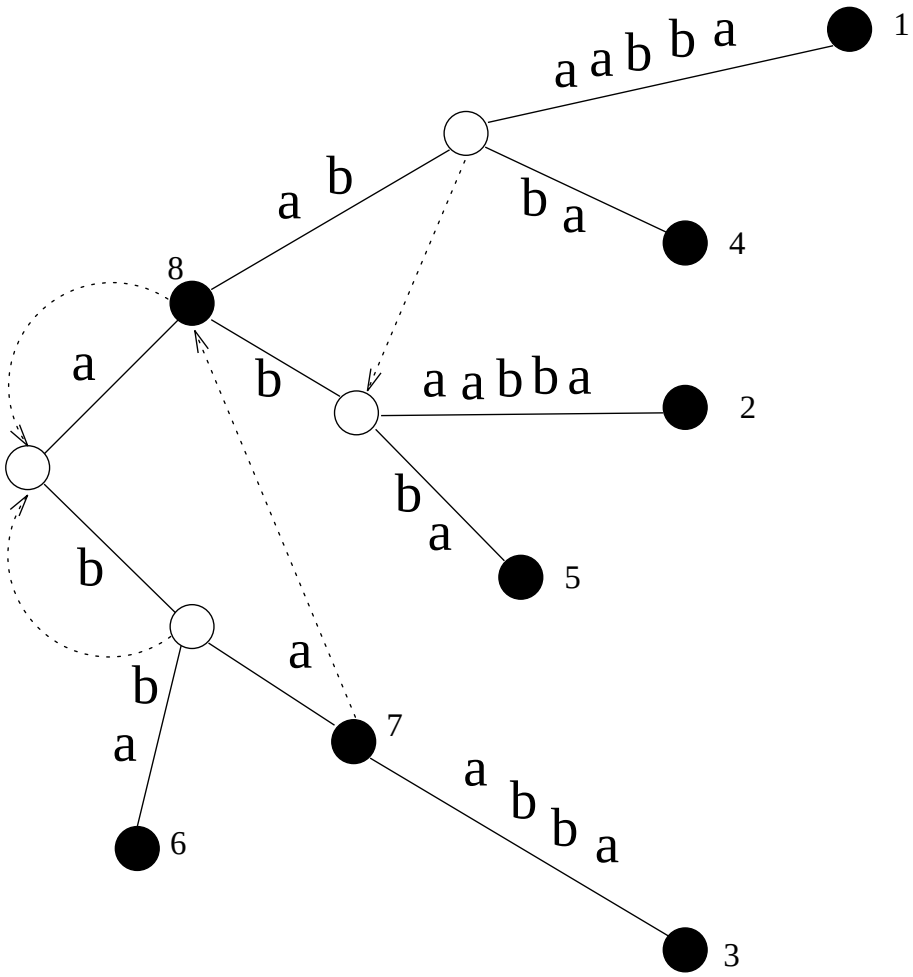

Figure 3: **Question 5.** **(a)** The suffix tree associated with the string `aabaabba`: black nodes are terminal states, suffix links are dotted lines and the numbers next to the terminal nodes represent the suffix index in the string.