String searching

Samy Aittahar

ULiege - INFO0049

Tuesday 6th March, 2018

String searching problem

Given a text T and a pattern P, determine whether the pattern appears in the text and if so, at which position(s).

E.g., *co* appears in *cocacola* at position 0 and 4.

Application : Spam detection

Application : Spam detection (cont'd)

- Actually, you benefit of it everyday without being aware of it,
- Take a look at your spam folder in your mailbox, you'll see recurrent words/symbols.
- Your mail provider, also aware of this fact (or should be), run algorithms based on string matching to detect them and to keep them away,
- In some ways, it prevent people from seeing their live destroyed (think of sCams...),
- In practice, these algorithms are more clever than those you will implement, but it is still important as it is the genesis of string matching algorithms.
- There are of course many others applications, such as intrusion detection, plagiarism detection, bioinformatics...

Naive search

```
 1: procedure strmatch_bf(T, P)          ▷ Positions where P occurs in T
 2:     I ← {}
 3:     for i from 0 to |T| − |P| do
 4:         while j ⩽ |P| and i + j ⩽ |T| and T[i + j] = P[j] do
 5:             j ← j + 1
 6:         end while
 7:         if j = |P| then
 8:             I ← I ∪ {i}
 9:         end if
10:     end for
11:     return I
12: end procedure
```

Complexity ?

Naive search

```
 1: procedure strmatch_bf(T, P)          ▷ Positions where P occurs in T
 2:     I ← {}
 3:     for i from 0 to |T| − |P| do
 4:         while j ⩽ |P| and i + j ⩽ |T| and T[i + j] = P[j] do
 5:             j ← j + 1
 6:         end while
 7:         if j = |P| then
 8:             I ← I ∪ {i}
 9:         end if
10:     end for
11:     return I
12: end procedure
```

Complexity : $O(|T||P|)$

Naive search - Example

T = "bananananobano", P = "nano"

	b	a	n	a	n	a	n	o	b	a	n	o
i=1	X											
i=2		X										
i=3			n	a	n	X						
i=4				X								
i=5					n	a	n	o				
i=6						X						
i=7							n	X				
i=8								X				
i=9									X			

"nano" occurs at position 5.

Redundancy

- Some comparisons are simply wasted work !
- For example, after iteration $i = 3$, we know that $T[4] = n$ and $T[5] = a$.
- Is there some way to use that knowledge to skip comparisons ?

Skipping characters in text

- After a partial match of j characters of P in T starting at position i, we already know the content of $T[i] \dots T[i+j]$.
- Thus, we can know if there is an overlapping. It is especially the case between iterations $i = 3$ and $i = 4$
- It means that we could *skip* the comparison at $i = 3$.
- How do we modify the naive search algorithm to take overlapping into account ?

(Less) Naive search

```
1:  procedure strmatch_bf(T, P)                    ▷ Positions where P occurs in T
2:      I ← {}
3:      i ← 0
4:      while i ⩽ |T| − |P| + 1 do
5:          j ← 0
6:          while j ⩽ |P| and i + j ⩽ |T| and T[i + j] = P[j] do
7:              j ← j + 1
8:          end while
9:          if j = |P| then
10:             I ← I ∪ i
11:         end if
12:         i ← i + max(1, overlap(P[0...j], P[0...|P|]))
13:     end while
14:     return I
15: end procedure
```

Still $O(|T||P|)$ but with less comparisons.

$overlap(x, y)$ returns the longest words that is a suffix of x and a prefix of y, except x and/or y.

More comparison skipping

- Let's have a closer look to iterations $i = 3$ and $i = 5$.
- The n that overlaps has already been considered in the $i = 2$.
- It means that we do not have to test this n character at iteration 5 (i.e., j would start from 2 instead of 1).
- Let's modify again the algorithm to have a first version of *Knuth-Morris-Pratt* algorithm.

KMP search

```
1:  procedure strmatch_bf(T, P)                    ▷ Positions where P occurs in T
2:      I ← {}
3:      i ← 0
4:      o ← 0
5:      while i ⩽ |T| − |P| + 1 do
6:          j ← o
7:          while j ⩽ |P| and i + j ⩽ |T| and T[i + j] = P[j] do
8:              j ← j + 1
9:          end while
10:         if j = |P| then
11:             I ← I ∪ i
12:         end if
13:         o ← overlap(P[0...j], P[0...|P|])
14:         i ← i + max(1, j − o)
15:     end while
16:     return I
17: end procedure
```

Still $O(|T||P|)$ but with less less comparisons.

Remarks

- The overlap function has to be computed before the string searching process. This is a preprocessing part.
- At this point, KMP seems to be easier to implement efficiently in any imperative programming language.
- Can we implement an alternative of KMP that is Prolog friendly ?

Finite state-machine (Quick reminder)

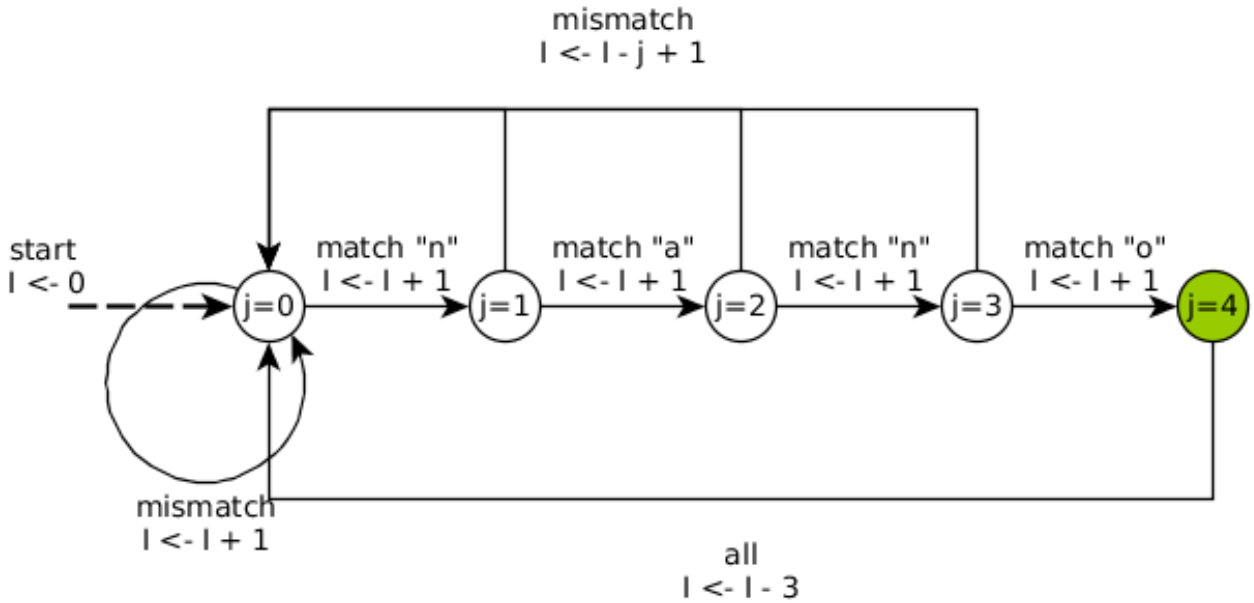
The components of a deterministic finite state-machine M are :

- A set of states S,
- A finite set of symbols Σ,
- A transition function $S \times \Sigma \rightarrow S$,
- An initial state $s_0 \in S$.
- A subset $F \subseteq S$.

Usually, F is the subset of states which is used to accept/reject inputs. In our case, we will use them to perform special actions.

FSM for naive string searching

Let us consider again $T = bananan obano$ and $P = nano$.



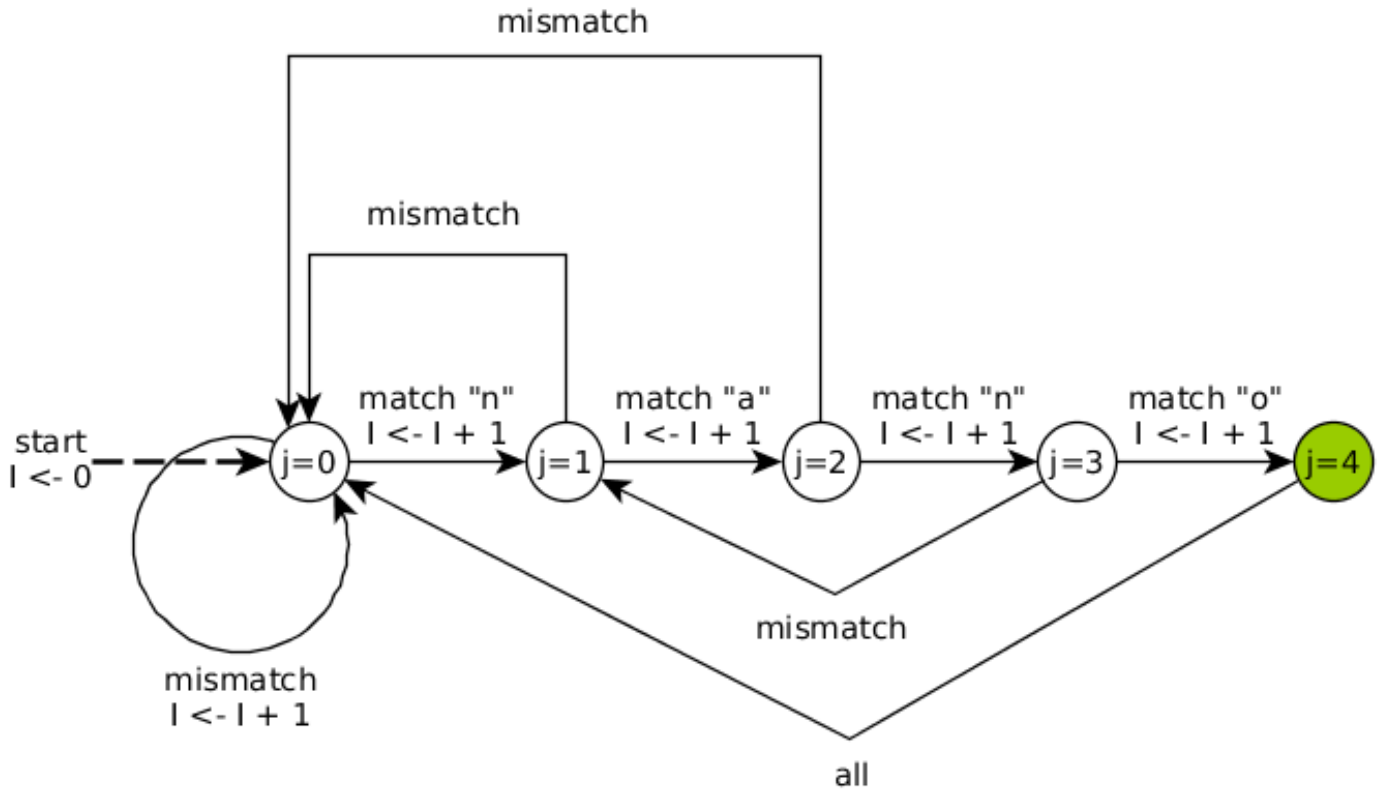
Each time $j = 4$ is reached, we store of $I - |P|$. This results in a set of positions where P occurs in T.

FSM for KMP searching

Automaton ?

Hint : First pre-compute the overlap function

FSM for KMP searching



Homework - StringS matching problem

- What if we want to look for *several* patterns in T ?
- Of course, we could build as many finite state machines as number of patterns.
- But we are more clever than that, and exploit the fact that patterns may share prefixes/suffixes.
- In this project, we expect you to either :
  - extend the current approach by using a single automata for all the patterns or
  - come up with a different approach from literature that you will carefully and clearly explain while providing the relevant reference.
- In both cases, your implementation needs to also handle several texts T.

Homework - Side problem

- A problem may occurs : the pattern may be "almost" present in the text.
- For example, in our spam detector, we may have "best deel" in instead of "best deal". We know such errors can occurs in spams.
- In exact matching, we wouldn't find that pattern, but we would if a single typo was tolerated. Also, what about 2,3,4... typos ?
- The implementation is not mandatory, but you may explain in a few some possible strategies to implement that feature.

Homework - Input/Output

- The inputs are :
  - A list of string $T = \{T_0 \dots T_n\}$,
  - A list of string $P = \{P_0 \dots P_m\}$.
- The output is a list of list of list of non-null natural number L, such that $L[i][j]$ contains the number of time P_j appears in T_i.
- If you address the side problem, then $L[i][j]$ is a pair in which the first member is the position and the second member is the number of typos found.

Homework : Deliverable and Evaluation

- Deliverable : Prolog source code file (documented and with execution examples).
- Structure and clarity of your code/documentation.
- Performance (computation time and quality of the solution)
  - Your implementation will be tested on big patterns/texts.
- Oral presentation.
  - Scheduled for 03/04/2018.

That's all folks

If you want to get more into details about string algorithms :

Dan Gusfield. 1997. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York, NY, USA.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd ed.). The MIT Press.

https://en.wikipedia.org/wiki/String_searching_algorithm