COMP36111: Advanced Algorithms I
Lecture 3: String Matching

Ian Pratt-Hartmann

Room KB2.38: email: ipratt@cs.man.ac.uk

2016–17
Outline

The string matching problem

The Rabin-Karp algorithm

The Boyer-Moore Algorithm

The Knuth-Morris-Pratt algorithm
• Suppose we are given some English text

A blazing sun upon a fierce August day was no greater rarity in southern France then, than at any other time, before or since. Everything in Marseilles, and about Marseilles, had stared at the fervid sky, and been stared at in return, until a staring habit had become universal there.

and a search string, say “Marseilles”.

• We would like to find all instances (or just the first instance) of the search string in the text.

• What’s the best way?
• Suppose we are given some English text
  
  A blazing sun upon a fierce August day was no greater rarity in southern France then, than at any other time, before or since. Everything in Marseilles, and about Marseilles, had stared at the fervid sky, and been stared at in return, until a staring habit had become universal there.

  and a search string, say “Marseilles”.

• We would like to find all instances (or just the first instance) of the search string in the text.

• What’s the best way?
• As usual, we start by modelling the data:
  • let $\Sigma$ be a finite non-empty set (the alphabet);
  • let $T = T[0], \ldots, T[n-1]$ be a string length $n$ over a fixed alphabet $\Sigma$;
  • let $P = P[0], \ldots, P[m-1]$ be a string length $m$ over $\Sigma$;
• We formalize the notion of an occurrence of one string in another:
  • string $P$ occurs with shift $i$ in string $T$ if $P[j] = T[i+j]$ for all $j$ ($0 \leq i < |P|$).
• The we have the following problem

MATCHING
Given: strings $T$ and $P$ over some fixed alphabet $\Sigma$.
Return: the set of integers $i$ such that $P$ occurs in $T$ with shift $i$. 
The string matching problem

The Rabin-Karp algorithm

The Boyer-Moore Algorithm

The Knuth-Morris-Pratt algorithm

• Here is a really stupid algorithm

```
begin naiveMatch(T, P)
    I ← ∅
    for i = 0 to |T| − |P|
        j ← 0
        until j = |P| or T[i + j] ≠ P[j]
        j++
        if j = |P|
            I = I ← {i}
    return I
end
```

• Graphically

```
0 1 2 3 4 5 6 7
| a b d |  |  |  |  |  |
0 1 2 3 4 5 6 7
| a b c d |  |  |  |
0 1 2 3 4 5 6 7
```

• Running time is $O(|T| \cdot |P|)$. 
Here is a really stupid algorithm

begin naiveMatch(\(T, P\))
    \(I \leftarrow \emptyset\)
    for \(i = 0\) to \(|T| - |P|\)
        \(j \leftarrow 0\)
        until \(j = |P|\) or \(T[i + j] \neq P[j]\)
        \(j++\)
        if \(j = |P|\)
            \(I = I \leftarrow \{i\}\)
    return \(I\)
end

Graphically

```
\begin{array}{ccccccc}
  & & & & & & \\
  & & & a & b & d & \\
\end{array}
```

Running time is \(O(|T| \cdot |P|)\).
• Here is a really stupid algorithm

\[
\text{begin naiveMatch}(T,P) \\
\quad I \leftarrow \emptyset \\
\quad \text{for } i = 0 \text{ to } |T| - |P| \\
\quad \quad j \leftarrow 0 \\
\quad \quad \text{until } j = |P| \text{ or } T[i+j] \neq P[j] \\
\quad \quad \quad j++ \\
\quad \quad \text{if } j = |P| \\
\quad \quad \quad I = I \leftarrow \{i\} \\
\quad \quad \text{return } I \\
\end{naiveMatch}
\]

• Graphically

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>a</th>
<th>b</th>
<th>d</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Running time is \(O(|T| \cdot |P|)\).
Outline

The string matching problem

The Rabin-Karp algorithm

The Boyer-Moore Algorithm

The Knuth-Morris-Pratt algorithm
• Let \( n = |T| \) and \( m = |P| \).

• Think of the elements of \( \Sigma \) as digits in a base-\( b \) numeral, where \( b = |\Sigma| \).

• Then \( P \) is the number \( P[0] \cdot b^{m-1} + \cdots + P[m-1] \cdot b^0 \).

• Similarly, \( T[i, \cdots, i+m-1] \) is
  \[ T[i] \cdot b^{m-1} + \cdots + T[i+m-1] \cdot b^0. \]

• To calculate \( T[i+1, \cdots, i+m] \) from \( T[i, \cdots, i+m-1] \), write:

\[
T[i+1, \cdots, i+m] = (T[i, \cdots, i+m-1] - T[i] \cdot b^{m-1}) \cdot b + T[i+m].
\]
• These numbers can get a bit large.
• However, we can work modulo \( q \), for some constant \( q \) (usually a prime) such that \( bq \) is about the size of a computer word.
• Of course, we have

\[
T[i+1, \ldots, i+m] = (T[i, \ldots, i+m-1] - T[i] \cdot b^{m-1}) \cdot b + T[i+m] \pmod{q}.
\]
• If \( T[i, \ldots, i+m-1] \neq P \pmod{q} \), then we know we do not have a match at shift \( i \).
• If \( T[i, \ldots, i+m-1] = P \pmod{q} \), then we simply check explicitly that \( T[i, \ldots, i+m-1] = P \).
• The worst-case running time of this algorithm is also $O(|T| \cdot |P|)$.
• On average, however, it works much better:
  • A rough estimate of the probability of a spurious match is $1/q$, since this is the probability that a random number will take a given value modulo $q$. (Well, that’s actually nonsense, but never mind.)
  • A reasonable estimate of the number of matches is $O(1)$, since patterns are basically rare.
• This leads to an expected performance of about $O(n + m + m(n/q))$
• Thus, expected running time will be about $O(n + m)$, since presumably $q > m$. 
Here is the algorithm

begin Rabin-Karp($T, S, q, b$)
  $I \leftarrow \emptyset$
  $m \leftarrow |P|$
  $t \leftarrow T[0] \cdot b^{m-1} + \ldots + T[m-1] \cdot b^0 \mod q$
  $p \leftarrow P[0] \cdot b^{m-1} + \ldots + P[m-1] \cdot b^0 \mod q$
  $i \rightarrow 0$
  while $i \leq |T| - m$
    if $p = t$
      $j \leftarrow 0$
      while $P[j] = T[i+j]$ and $j < |P|$
         $j++$
      if $j = |P|$
        $I \leftarrow I \cup \{i\}$
        $t \leftarrow (t - T[i] \cdot b^{m-1}) \cdot b + T[i+m] \mod q$
      $i++$
    return $I$
end
Outline

The string matching problem

The Rabin-Karp algorithm

The Boyer-Moore Algorithm

The Knuth-Morris-Pratt algorithm
• The Boyer-Moore algorithm is a refinement of the naïve string matching algorithm.

• It checks the pattern against the text starting from the end (rather than the beginning).

- It also incorporates and two refinements:
  - the bad character heuristic;
  - the good suffix heuristic.
• The bad character heuristic: suppose $T[i+j] \neq P[j]$.
  • Let $k$ be largest such that $T[i+j] = P[k]$ (-1 if no such $k$).
  • Then we can safely increase $i$ by $j - k$.
  • Denote this value of $k$ by $\lambda(T[i+j])$. Notice $\lambda$ depends only on $P$ and the character $T[i+j]$.

• Graphically:

```
0 1 2 3
a b c d
```

In this example, $j = 2$ and $k = 0$; so we can increase $i$ by 2.
• The good character heuristic is more complicated.

• Suppose we have a mismatch at some pattern position $j$.

\[
\begin{array}{cccc}
0 & | & \text{mismatch} & | & m - 1 \\
\end{array}
\]

\[
i
\]

• Call $P[j + 1, \ldots, m - 1]$ the good suffix.

• Let us say that two strings are end-equivalent if one is a suffix of the other.

• Note: it takes a bit of thinking to show that this really is an equivalence relation!

• Let $k$ be the largest value ($0 \leq k < m$) such that the prefix of $P[0, \ldots, k - 1]$ is end-equivalent to the good suffix.

• The good suffix heuristic says: increase $i$ by $m - k$. 
• Suppose first that $k \geq m - j - 1$. This means that the good suffix is a suffix of $P[0, \ldots, k - 1]$.

\begin{center}
\begin{tikzpicture}
\draw[|-|] (0,0) -- (10,0);
\draw[red,|-|] (1,0) -- (3,0);
\draw[green,|-|] (4,0) -- (9,0);
\node at (5,0) {$i$};
\end{tikzpicture}
\end{center}

\begin{center}
\begin{tikzpicture}
\draw[|-|] (0,0) -- (5,0);
\draw[red,|-|] (1,0) -- (2,0);
\draw[green,|-|] (3,0) -- (4,0);
\node at (2.5,0) {$j$};
\end{tikzpicture}
\end{center}

• There cannot be any match corresponding to a move of less than $m - k$, because then, the good suffix would be a suffix of a still longer prefix, contradicting the maximality of $k$. 
Suppose first that $k \geq m - j - 1$. This means that the good suffix is a suffix of $P[0, \ldots, k - 1]$.

There cannot be any match corresponding to a move of less than $m - k$, because then, the good suffix would be a suffix of a still longer prefix, contradicting the maximality of $k$. 
Suppose first that $k \geq m - j - 1$. This means that the good suffix is a suffix of $P[0, \ldots, k - 1]$.

There cannot be any match corresponding to a move of less than $m - k$, because then, the good suffix would be a suffix of a still longer prefix, contradicting the maximality of $k$. 

![Diagram showing the relationship between $i$, $j$, and $m-1$.]
• Suppose on the other hand that $k < m - j - 1$. This means that $P[0, \ldots, k - 1]$ is a proper suffix of the good suffix.

• There certainly cannot be any match corresponding to a move of $j$ or less (for then the good suffix would be a proper suffix of some prefix, and $k \geq m - j - 1$).

• But there also cannot be any match corresponding to a move of more than $j$ but less than $m - k$, because then, a longer prefix than $P[0, \ldots, k - 1]$ would be a proper suffix of the good suffix, again contradicting the maximality of $k$. 
• Suppose on the other hand that $k < m - j - 1$. This means that $P[0, \ldots, k - 1]$ is a proper suffix of the good suffix.

\[
\begin{array}{c}
\text{i} \\
\hline
0 & j & m - 1
\end{array}
\]

• There certainly cannot be any match corresponding to a move of $j$ or less (for then the good suffix would be a proper suffix of some prefix, and $k \geq m - j - 1$).

• But there also cannot be any match corresponding to a move of more than $j$ but less than $m - k$, because then, a longer prefix than $P[0, \ldots, k - 1]$ would be a proper suffix of the good suffix, again contradicting the maximality of $k$. 

Suppose on the other hand that $k < m - j - 1$. This means that $P[0, \ldots, k - 1]$ is a proper suffix of the good suffix.

There certainly cannot be any match corresponding to a move of $j$ or less (for then the good suffix would be a proper suffix of some prefix, and $k \geq m - j - 1$).

But there also cannot be any match corresponding to a move of more than $j$ but less than $m - k$, because then, a longer prefix than $P[0, \ldots, k - 1]$ would be a proper suffix of the good suffix, again contradicting the maximality of $k$. 
• Thus, we want to compute the largest $k$ ($0 \leq k < m$) such that either the prefix $P[0, \ldots, k - 1]$, is a (proper) suffix of the good suffix or the good suffix is a suffix of the prefix $P[0, \ldots, k - 1]$.

• Denote the value $m - k$ by $\gamma[j]$. Notice $\gamma$ is always positive, and depends only and depends only on $P$ and $j$. 
• The functions $\lambda$ and $\gamma$ can be pre-computed once $P$ is known (independently of $T$).

• Thus, the Boyer-Moore algorithm is

```latex
begin naiveMatch(T,S) 
  compute the functions $\lambda$ and $\gamma$
  $l \leftarrow \emptyset$
  $i \leftarrow 0$
  while $i \leq |T| - |P|$ 
    $j \leftarrow |P| - 1$
    while $T[i + j] = P[j]$ and $j > 0$
      $j--$
      if $j = 0$
        $l \leftarrow l \cup \{i\}$
      $i \leftarrow i + \gamma[-1]$
    else
      $i \leftarrow i + \max(\gamma[j], j - \lambda(T[i + j]))$
  return $l$
end
```
Outline

The string matching problem

The Rabin-Karp algorithm

The Boyer-Moore Algorithm

The Knuth-Morris-Pratt algorithm
• The basic idea of this algorithm is as follows. Suppose we have a mismatch at some pattern position $j$.

![Diagram showing good prefix and proper suffix]

• Call $P[0, \ldots, j - 1]$ the **good prefix**.

• Now let is look at the longest prefix of the good prefix which is also a **proper** suffix of the good prefix.
• The basic idea of this algorithm is as follows. Suppose we have a mismatch at some pattern position $j$.

• Call $P[0, \ldots, j - 1]$ the good prefix.

• Now let is look at the longest prefix of the good prefix which is also a proper suffix of the good prefix.
• Suppose that the length of the longest such proper prefix is $k$.
• It should be clear that there cannot be any matches involving shifts of less than $k$ (for then, $k$ would not be maximal).

So we can shift the pattern up by $k$.
• Denote the length $k$ of the longest prefix of $P[0, \ldots, j - 1]$ that is also a proper suffix of $P[0, \ldots, j - 1]$ by $\pi(j)$, for all $j$ ($1 \leq j \leq |P|$), and set $\pi(0) = 0$. 
• Suppose that the length of the longest such proper prefix is $k$.
• It should be clear that there cannot be any matches involving shifts of less than $k$ (for then, $k$ would not be maximal).

\begin{center}
\begin{tabular}{c c c}
  & & \\
  & & \\
  & & \\
  & & \\
\end{tabular}
\end{center}

$k$ $j$

• So we can shift the pattern up by $k$.
• Denote the length $k$ of the longest prefix of $P[0, \ldots, j - 1]$ that is also a proper suffix of $P[0, \ldots, j - 1]$ by $\pi(j)$, for all $j$ ($1 \leq j \leq |P|$), and set $\pi(0) = 0$. 
Suppose that the length of the longest such proper prefix is $k$.

It should be clear that there cannot be any matches involving shifts of less than $k$ (for then, $k$ would not be maximal).

So we can shift the pattern up by $k$.

Denote the length $k$ of the longest prefix of $P[0, \ldots, j - 1]$ that is also a proper suffix of $P[0, \ldots, j - 1]$ by $\pi(j)$, for all $j$ ($1 \leq j \leq |P|$), and set $\pi(0) = 0$. 
• Don’t do it! Don’t shift the pattern up by $k$!
• Instead, think what would happen to $j$ if you did. The length of the good prefix would change:

$$ j \leftarrow \pi(j). $$

Now repeat the process: either we get a match and make progress (incrementing $j$), or we get a mismatch and execute $j \leftarrow \pi(j)$. 
Here is the algorithm.

```plaintext
def knuth_morris_pratt(T, P):
    I = {}
    compute the function π
    i = 0, j = 0
    while i < |T|
        if P[j] = T[i]
            if j = |P| - 1
                I = I ∪ {i - |P| + 1}
                i++, j = π[|P|]
                i++, j++
            else if j > 0
                j = π[j]
            else
                i++
        else
            j = π[j]
    return I
```

end
The running time (ignoring the construction of $\pi$ is $O(|T|)$.

- Letting $k = i - j$, each iteration of the loop either increments $i$ or increases $k$ by at least 1, and neither quantity reduces.
- Hence, the while loop can execute at most $2|T|$ times.

- Note that $\pi$ is one-off: it depends only $P$ and not on $T$, so its computation is not critical.
- In fact, however, $\pi$ can be computed in time $O(|P|)$, leading to an overall running time of $O(|P| + |T|)$.