COMP36111: Advanced Algorithms I
Lecture 5: Searching strings

Ian Pratt-Hartmann

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

2019–20
In this lecture, we consider the problem of searching for strings in text. We present a simple but effective improvement to the naïve algorithm, as well as a more sophisticated, and very elegant, algorithm. The lecture three parts:

- the string matching problem;
- the Rabin-Karp algorithm;
- the Knuth-Morris-Pratt algorithm.
Outline

The string matching problem

The Rabin-Karp algorithm

The Knuth-Morris-Pratt algorithm

Summary
• 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 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 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 at position $i$ in string $T$ if $P[j] = T[i+j]$ for all $j$ ($0 \leq i < |P|$).
• We thus have the following task

MATCHING
Given: strings $T$ and $P$ over some fixed alphabet $\Sigma$.
Return: the index $i$ such that $P$ first occurs in $T$ at position $i$, or ”No match” if $P$ does not occur in $T$. 
Here is a naïve algorithm

begin naiveMatch(T, P)
    for i = 0 to |T| − |P|
        j ← 0
        until j = |P| or T[i + j] ≠ P[j]
            j++
        if j = |P|
            return i
    return ”No match”
end

Graphically

```
  0 1 2 3 4 5 6 7 8
 0   a b d     
```

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

\[
\begin{align*}
\text{begin naiveMatch}(T,P) \\
\text{for } i = 0 \text{ to } |T| - |P| \\\n\quad j \leftarrow 0 \\
\quad \text{until } j = |P| \text{ or } T[i+j] \neq P[j] \\
\quad j++ \\
\quad \text{if } j = |P| \\
\quad \quad \text{return } i \\
\quad \text{return } "\text{No match}" \\
\end{align*}
\]

end

• Graphically

\[
\begin{array}{cccc}
\text{a} & \text{b} & \text{d} \\
\text{a} & \text{b} & \text{c} & \text{d} \\
0 & i & j \\
\end{array}
\]

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

\[
\begin{align*}
\text{begin naiveMatch}(T,P) \\
\text{for } i = 0 \text{ to } |T| - |P| \\
\quad j &\leftarrow 0 \\
\quad \text{until } j = |P| \text{ or } T[i+j] \neq P[j] \\
\quad j &++ \\
\quad \text{if } j = |P| \\
\quad \quad \text{return } i \\
\quad \text{return } \text{"No match"}
\end{align*}
\]

end

Graphically

\[
\begin{array}{cccc}
\text{a} & \text{b} & \text{d} \\
\hline
0 & i \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{a} & \text{b} & \text{c} & \text{d} \\
\hline
0 & j
\end{array}
\]

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

The string matching problem

The Rabin-Karp algorithm

The Knuth-Morris-Pratt algorithm

Summary
• 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.)
  • We get at most one match, since we stop when we find one.

• 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, P, q, b)
    m ← |P|
    t ← T[0] \cdot b^{m-1} + \cdots + T[m-1] \cdot b^0 \mod q
    p ← P[0] \cdot b^{m-1} + \cdots + P[m-1] \cdot b^0 \mod q
    i → 0
    while i ≤ |T| − m
        if p = t
            j ← 0
            while P[j] = T[i + j] and j < |P|
                j++
            if j = |P|
                return i
        t ← (t − T[i] \cdot b^{m-1}) \cdot b + T[i + m] \mod q
        i++
    return "No match"
end
```
Outline

The string matching problem

The Rabin-Karp algorithm

The Knuth-Morris-Pratt algorithm

Summary
• We begin with a simple observation. Suppose a string $Q$ matches against a properly shifted version of itself.
• Let the ‘tail’ be $A$, with $0 < h = |A|$. Then, $P$ consists of repeated $A$’s and some prefix $A''$ with $0 \leq h'' = |A''| < h$.
• Write $A = A'A''$, and observe that $A''$ are the first $h''$ characters of $A$. 

$$Q = \begin{array}{c} \\
\end{array}$$

$$Q = \begin{array}{c} \\
\end{array}$$
We begin with a simple observation. Suppose a string $Q$ matches against a properly shifted version of itself.

Let the ‘tail’ be $A$, with $0 < h = |A|$. Then, $P$ consists of repeated $A$’s and some prefix $A''$ with $0 \leq h'' = |A''| < h$.

Write $A = A'A''$, and observe that $A''$ are the first $h''$ characters of $A$.

$$Q = \boxed{\phantom{A}}$$

$$Q = \boxed{\phantom{A}} \quad A$$
• We begin with a simple observation. Suppose a string \( Q \) matches against a properly shifted version of itself.

• Let the ‘tail’ be \( A \), with \( 0 < h = |A| \). Then, \( P \) consists of repeated \( A \)'s and some prefix \( A'' \) with \( 0 \leq h'' = |A''| < h \).

• Write \( A = A'A'' \), and observe that \( A'' \) are the first \( h'' \) characters of \( A \).

\[
Q = \begin{array}{c}
\vspace{1cm}
\end{array} & A
\]

\[
Q = \begin{array}{c}
\vspace{1cm}
\end{array} & A
\]
We begin with a simple observation. Suppose a string $Q$ matches against a properly shifted version of itself.

Let the ‘tail’ be $A$, with $0 < h = |A|$. Then, $P$ consists of repeated $A$’s and some prefix $A''$ with $0 \leq h'' = |A''| < h$.

Write $A = A'A''$, and observe that $A''$ are the first $h''$ characters of $A$.

\[
Q = \begin{array}{c}
\vspace{1cm}
\end{array} \quad A
\]

\[
Q = \begin{array}{c}
\vspace{1cm}
\end{array} \quad A \quad A
\]
• We begin with a simple observation. Suppose a string $Q$ matches against a properly shifted version of itself.

• Let the ‘tail’ be $A$, with $0 < h = |A|$. Then, $P$ consists of repeated $A$’s and some prefix $A''$ with $0 \leq h'' = |A''| < h$.

• Write $A = A'A''$, and observe that $A''$ are the first $h''$ characters of $A$.

$$Q = \begin{array}{|c|c|}
\hline
\multicolumn{2}{|c|}{A} \\
\hline
\multicolumn{2}{|c|}{A} \\
\hline
\end{array}$$

$$Q = \begin{array}{|c|c|}
\hline
\multicolumn{2}{|c|}{A} \\
\hline
\multicolumn{2}{|c|}{A} \\
\hline
\end{array}$$
• We begin with a simple observation. Suppose a string $Q$ matches against a properly shifted version of itself.

• Let the ‘tail’ be $A$, with $0 < h = |A|$. Then, $P$ consists of repeated $A$’s and some prefix $A''$ with $0 \leq h'' = |A''| < h$.

• Write $A = A'A''$, and observe that $A''$ are the first $h''$ characters of $A$.

\[
Q = \begin{array}{c|c|c}
A & A & A \\
\end{array}
\]

\[
Q = \begin{array}{c|c|c}
A'' & A & A \\
\end{array}
\]
• We begin with a simple observation. Suppose a string $Q$ matches against a properly shifted version of itself.
• Let the ‘tail’ be $A$, with $0 < h = |A|$. Then, $P$ consists of repeated $A$’s and some prefix $A''$ with $0 \leq h'' = |A''| < h$.
• Write $A = A'A''$, and observe that $A''$ are the first $h''$ characters of $A$.
\[
Q = \begin{array}{c|c|c|c}
A' & A'' & A & A
\end{array}
\]
\[
Q = \begin{array}{c|c|c|c}
A'' & A & A & A
\end{array}
\]
• We begin with a simple observation. Suppose a string $Q$ matches against a properly shifted version of itself.
• Let the ‘tail’ be $A$, with $0 < h = |A|$. Then, $P$ consists of repeated $A$’s and some prefix $A''$ with $0 \leq h'' = |A''| < h$.
• Write $A = A'A''$, and observe that $A''$ are the first $h''$ characters of $A$.

\[
Q = \begin{array}{c}
A'' \mid A' \mid A'' \\
A \mid A
\end{array}
\]

\[
Q = \begin{array}{c}
A'' \mid A \\
A \mid A
\end{array}
\]

• Thus, $Q = A''A^s$ for some $s > 0$, with $A = A'A''$. 
• The basic idea of the KMP algorithm is as follows. Suppose we have a mismatch at some pattern position $j$.

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

• Now let us look at the longest prefix of the good prefix which is also a proper suffix of the good prefix.

• Call this string $Q$, and let $k = |Q|$. 
• Suppose there were a match involving a shift of \( < j - k \).

\[
\begin{array}{c|c|c}
0 & \ldots & j \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\langle \ldots \rangle & \langle \ldots \rangle & \rangle \\
< j - k & > k \\
\end{array}
\]

• Then \( k \) is not be maximal. Contradiction.
• Suppose there were a match involving a shift of \( \lt j - k \).

\[
\begin{array}{cccc}
0 & \text{green} & j & \text{red} \\
\text{......} & \text{......} & \text{......} & \text{......} \\
\lt j - k & \gt k \\
\end{array}
\]

• Then \( k \) is not be maximal. Contradiction.
• 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 $\pi(0) = 0$.

• So we could shift the pattern up by $j - \pi(j)$.

• **Think what would happen to $j$ if you did.** It would be replaced by $\pi(j)$:

```
\[\begin{array}{c}
\text{Pattern} \\
\hline
\text{\quad \quad \quad \quad} \\
\text{\quad \quad \quad \quad} \\
\text{\quad \quad \quad \quad} \\
\text{\quad \quad \quad \quad} \\
\text{\quad \quad \quad \quad} \\
\hline
\text{\quad \quad \quad \quad} \\
\text{\quad \quad \quad \quad} \\
\text{\quad \quad \quad \quad} \\
\text{\quad \quad \quad \quad} \\
\text{\quad \quad \quad \quad} \\
\end{array}\]
```

• Set $j \leftarrow \pi(j)$ and continue: 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 (assuming we can compute $\pi$):

```plaintext
begin KMP($T$, $P$)
    compute-$\pi$(P)
    $i \leftarrow 0$, $j \leftarrow 0$
    while $i < |T|$
        if $P[j] = T[i]$
            if $j = |P| - 1$
                return $i - |P| + 1$
            else
                $i++$, $j++$
        else if $j > 0$
            $j \leftarrow \pi[j]$
        else
            $i++$
    return “Not found”
end
```
• We need to think about computing $\pi$.
• Here is the general idea behind $\text{compute-}\pi(P)$

\[ j \quad |P| - 1 \]

• $j$ points to the square after the longest confirmed good-prefix for $P[0] \ldots P[i - 1]$.
• if we get a match, we can simply increase both $i$ and $j$, and update $\pi(i)$. 
• We need to think about computing $\pi$.
• Here is the general idea behind $\text{compute-}\pi(P)$

\[
\begin{array}{cccccc}
\pi(j - 1) & \uparrow & j & \uparrow & i & |P| - 1
\end{array}
\]

• $j$ points to the square after the longest confirmed good-prefix for $P[0] \ldots P[i - 1]$.
• if we get a match, we can simply increase both $i$ and $j$, and update $\pi(i)$.
• if we get a mis-match, we do not have to start at the beginning of the pattern; we can use $\pi(j - 1)$ to jump over characters we know will match.
- The following algorithm computes $\pi$.

```
begin compute-\pi(P)
  i ← 1
  j ← 0
  f(0) ← 0
  while i < |P|
    if P[i] = P[j]
      f(i) ← j + 1
      i++, j++
    else if j > 0
      j ← f(j − 1)
    else
      f(i) ← 0
      i++
  return f
end
```
The running time of \textit{Knuth-Morris-Pratt}(T,P) (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, using a similar argument, the running time of \textit{compute-}\(\pi\)(\(P\)) is \(O(|P|)\).

- Hence, overall running time is \(O(|P| + |T|)\).
Outline

The string matching problem

The Rabin-Karp algorithm

The Knuth-Morris-Pratt algorithm

Summary
Summary

• In this lecture, we have considered:
  • the naïve string matching algorithm;
  • the Rabin-Karp optimization;
  • the Knuth-Morris-Pratt algorithm.

• Reading:
  • Goodrich and Tamassia 23.3.