String Matching: Knuth-Morris-Pratt algorithm

Madhavan Mukund

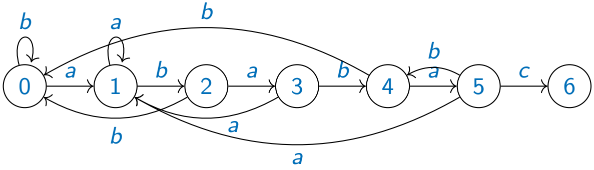
https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 10

Pattern matching using automata

- Finite state automaton for pattern `p`
  - States $\{0, 1, \dots, m\}$
  - State i denotes match upto `p[:i]`
  - Transition $i \xrightarrow{a} j$ descibes how to update the match on reading `a`
- Start scanning text in initial state 0
- In state i, read `t[j]`, take the transition labelled `t[j]`
- If we reach the final state m, we have found a full match for `p`
- Single scan of `t` suffices



Processing *abababac*

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{c} 6$$

Knuth-Morris-Pratt algorithm

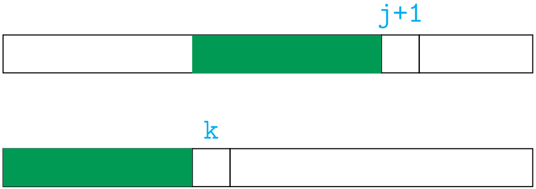
- Compute the automaton for p efficiently

Knuth-Morris-Pratt algorithm

- Compute the automaton for `p` efficiently
- Match `p` against itself
    - `match[j] = k` if suffix of `p[:j+1]` matches prefix `p[:k]`

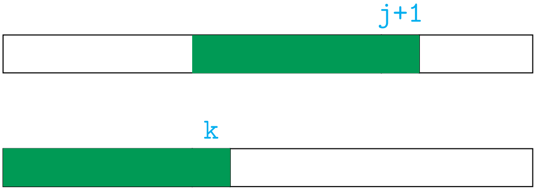
Knuth-Morris-Pratt algorithm

- Compute the automaton for p efficiently
- Match p against itself
    - match[j] = k if suffix of p[:j+1] matches prefix p[:k]
- Suppose suffix of p[:j+1] matches prefix p[:k]



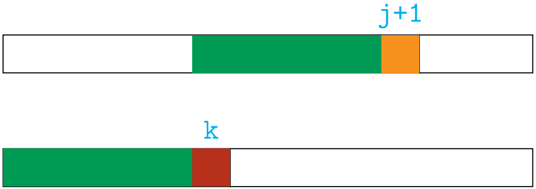
Knuth-Morris-Pratt algorithm

- Compute the automaton for p efficiently
- Match p against itself
  - match[j] = k if suffix of p[:j+1] matches prefix p[:k]
- Suppose suffix of p[:j+1] matches prefix p[:k]
  - If p[j+1] == p[k], extend the match



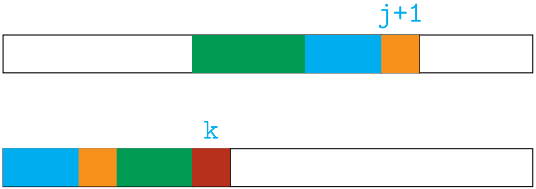
Knuth-Morris-Pratt algorithm

- Compute the automaton for `p` efficiently
- Match `p` against itself
  - `match[j] = k` if suffix of `p[:j+1]` matches prefix `p[:k]`
- Suppose suffix of `p[:j+1]` matches prefix `p[:k]`
  - If `p[j+1] == p[k]`, extend the match
  - Otherwise try to find a shorter prefix that can be extended by `p[j+1]`



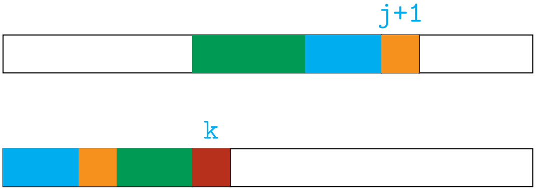
Knuth-Morris-Pratt algorithm

- Compute the automaton for p efficiently
- Match p against itself
  - match[j] = k if suffix of p[:j+1] matches prefix p[:k]
- Suppose suffix of p[:j+1] matches prefix p[:k]
  - If p[j+1] == p[k], extend the match
  - Otherwise try to find a shorter prefix that can be extended by p[j+1]



Knuth-Morris-Pratt algorithm

- Compute the automaton for `p` efficiently
- Match `p` against itself
  - `match[j] = k` if suffix of `p[:j+1]` matches prefix `p[:k]`
- Suppose suffix of `p[:j+1]` matches prefix `p[:k]`
  - If `p[j+1] == p[k]`, extend the match
  - Otherwise try to find a shorter prefix that can be extended by `p[j+1]`
- Usually refer to `match` as failure function `fail`
  - Where to fall back if match fails



Computing the fail function

- Initialize fail[j] = 0 for all j

```
def kmp_fail(p):

  # Initialize
  m = len(p)
  fail = [0 for i in range(m)]

  # Update
  j,k = 1,0
  while j < m:
    if p[j] == p[k]: #k+1 chars match
      fail[j] = k+1
      j,k = j+1,k+1
    elif k > 0:      #find shorter prefix
      k = fail[k-1]
    else:            #no match found at j
      j = j+1
  return(fail)
```

Computing the fail function

- Initialize fail[j] = 0 for all j
- k keeps track of length of current match
- j is next position to update fail

```
def kmp_fail(p):

  # Initialize
  m = len(p)
  fail = [0 for i in range(m)]

  # Update
  j,k = 1,0
  while j < m:
    if p[j] == p[k]: #k+1 chars match
      fail[j] = k+1
      j,k = j+1,k+1
    elif k > 0:      #find shorter prefix
      k = fail[k-1]
    else:            #no match found at j
      j = j+1
  return(fail)
```

Computing the fail function

- Initialize fail[j] = 0 for all j
- k keeps track of length of current match
- j is next position to update fail
- If p[j] == p[k] extend the match, set fail[j] = k+1

```
def kmp_fail(p):

  # Initialize
  m = len(p)
  fail = [0 for i in range(m)]

  # Update
  j,k = 1,0
  while j < m:
    if p[j] == p[k]: #k+1 chars match
      fail[j] = k+1
      j,k = j+1,k+1
    elif k > 0:      #find shorter prefix
      k = fail[k-1]
    else:            #no match found at j
      j = j+1
  return(fail)
```

Computing the fail function

- Initialize fail[j] = 0 for all j
- k keeps track of length of current match
- j is next position to update fail
- If p[j] == p[k] extend the match, set fail[j] = k+1
- If p[j] != p[k] find a shorter prefix that matches suffix of p[:j]
  - Step back to fail[k-1]

```
def kmp_fail(p):

  # Initialize
  m = len(p)
  fail = [0 for i in range(m)]

  # Update
  j,k = 1,0
  while j < m:
    if p[j] == p[k]: #k+1 chars match
      fail[j] = k+1
      j,k = j+1,k+1
    elif k > 0:      #find shorter prefix
      k = fail[k-1]
    else:            #no match found at j
      j = j+1
  return(fail)
```

Computing the fail function

- Initialize fail[j] = 0 for all j
- k keeps track of length of current match
- j is next position to update fail
- If p[j] == p[k] extend the match, set fail[j] = k+1
- If p[j] != p[k] find a shorter prefix that matches suffix of p[:j]
    - Step back to fail[k-1]
- If we don't find a nontrivial prefix to extend, retain fail[j] = 0, move to next position

```
def kmp_fail(p):

  # Initialize
  m = len(p)
  fail = [0 for i in range(m)]

  # Update
  j,k = 1,0
  while j < m:
    if p[j] == p[k]: #k+1 chars match
      fail[j] = k+1
      j,k = j+1,k+1
    elif k > 0:      #find shorter prefix
      k = fail[k-1]
    else:            #no match found at j
      j = j+1
  return(fail)
```

Analysis of `fail` computation

- Want to show this takes time $O(m)$

```
def kmp_fail(p):

  # Initialize
  m = len(p)
  fail = [0 for i in range(m)]

  # Update
  j,k = 1,0
  while j < m:
    if p[j] == p[k]: #k+1 chars match
      fail[j] = k+1
      j,k = j+1,k+1
    elif k > 0:      #find shorter prefix
      k = fail[k-1]
    else:            #no match found at j
      j = j+1
  return(fail)
```

Analysis of `fail` computation

- Want to show this takes time $O(m)$
- `j` incremented $m-1$ times in `while`

```
def kmp_fail(p):

  # Initialize
  m = len(p)
  fail = [0 for i in range(m)]

  # Update
  j,k = 1,0
  while j < m:
    if p[j] == p[k]: #k+1 chars match
      fail[j] = k+1
      j,k = j+1,k+1
    elif k > 0:      #find shorter prefix
      k = fail[k-1]
    else:            #no match found at j
      j = j+1
  return(fail)
```

Analysis of `fail` computation

- Want to show this takes time $O(m)$
- `j` incremented $m-1$ times in `while`
- But we also have iterations where `k` decreases and `j` is unchanged

```
def kmp_fail(p):

  # Initialize
  m = len(p)
  fail = [0 for i in range(m)]

  # Update
  j,k = 1,0
  while j < m:
    if p[j] == p[k]: #k+1 chars match
      fail[j] = k+1
      j,k = j+1,k+1
    elif k > 0:      #find shorter prefix
      k = fail[k-1]
    else:            #no match found at j
      j = j+1
  return(fail)
```

Analysis of fail computation

- Want to show this takes time $O(m)$
- j incremented $m-1$ times in while
- But we also have iterations where k decreases and j is unchanged
- Total number of decreases to k cannot exceed total number of increments to k

```
def kmp_fail(p):

  # Initialize
  m = len(p)
  fail = [0 for i in range(m)]

  # Update
  j,k = 1,0
  while j < m:
    if p[j] == p[k]: #k+1 chars match
      fail[j] = k+1
      j,k = j+1,k+1
    elif k > 0:      #find shorter prefix
      k = fail[k-1]
    else:            #no match found at j
      j = j+1
  return(fail)
```

Analysis of `fail` computation

- Want to show this takes time $O(m)$
- `j` incremented $m-1$ times in `while`
- But we also have iterations where `k` decreases and `j` is unchanged
- Total number of decreases to `k` cannot exceed total number of increments to `k`
- Overall `k` is incremented at most $m-1$ times

```
def kmp_fail(p):

  # Initialize
  m = len(p)
  fail = [0 for i in range(m)]

  # Update
  j,k = 1,0
  while j < m:
    if p[j] == p[k]: #k+1 chars match
      fail[j] = k+1
      j,k = j+1,k+1
    elif k > 0:      #find shorter prefix
      k = fail[k-1]
    else:            #no match found at j
      j = j+1
  return(fail)
```

Analysis of `fail` computation

- Want to show this takes time $O(m)$
- `j` incremented $m-1$ times in `while`
- But we also have iterations where `k` decreases and `j` is unchanged
- Total number of decreases to `k` cannot exceed total number of increments to `k`
- Overall `k` is incremented at most $m-1$ times
- Hence overall complexity is $O(m)$

```
def kmp_fail(p):

  # Initialize
  m = len(p)
  fail = [0 for i in range(m)]

  # Update
  j,k = 1,0
  while j < m:
    if p[j] == p[k]: #k+1 chars match
      fail[j] = k+1
      j,k = j+1,k+1
    elif k > 0:      #find shorter prefix
      k = fail[k-1]
    else:            #no match found at j
      j = j+1
  return(fail)
```

Implementing string search using fail function

- Scan t from beginning

```
def find_kmp(t, p):
  n,m = len(t),len(p)
  if m == 0:
    return 0 # pattern is empty
  fail = kmp_fail(p) # preprocessing
  j = 0 # index into text
  k = 0 # index into pattern
  while j < n:
    if t[j] == p[k]: # matched p[0:k+1]
      if k == m - 1: # match is complete
        return(j - m + 1)
      j,k = j+1,k+1 # extend match
    elif k > 0:
      k = fail[k-1] # use smaller prefix
    else:
      j = j+1
  return(-1) # reached end without match
```

Implementing string search using `fail` function

- Scan `t` from beginning
- `j` is next position in `t`
- `k` is currently matched position in `p`

```
def find_kmp(t, p):
  n,m = len(t),len(p)
  if m == 0:
    return 0 # pattern is empty
  fail = kmp_fail(p) # preprocessing
  j = 0 # index into text
  k = 0 # index into pattern
  while j < n:
    if t[j] == p[k]: # matched p[0:k+1]
      if k == m - 1: # match is complete
        return(j - m + 1)
      j,k = j+1,k+1 # extend match
    elif k > 0:
      k = fail[k-1] # use smaller prefix
    else:
      j = j+1
  return(-1) # reached end without match
```

Implementing string search using fail function

- Scan t from beginning
- j is next position in t
- k is currently matched position in p
- If t[j] == p[k] extend the match

```
def find_kmp(t, p):
  n,m = len(t),len(p)
  if m == 0:
    return 0 # pattern is empty
  fail = kmp_fail(p) # preprocessing
  j = 0 # index into text
  k = 0 # index into pattern
  while j < n:
    if t[j] == p[k]: # matched p[0:k+1]
      if k == m - 1: # match is complete
        return(j - m + 1)
      j,k = j+1,k+1 # extend match
    elif k > 0:
      k = fail[k-1] # use smaller prefix
    else:
      j = j+1
  return(-1) # reached end without match
```

Implementing string search using fail function

- Scan t from beginning
- j is next position in t
- k is currently matched position in p
- If t[j] == p[k] extend the match
- If t[j] != p[k], update match prefix

```
def find_kmp(t, p):
  n,m = len(t),len(p)
  if m == 0:
    return 0 # pattern is empty
  fail = kmp_fail(p) # preprocessing
  j = 0 # index into text
  k = 0 # index into pattern
  while j < n:
    if t[j] == p[k]: # matched p[0:k+1]
      if k == m - 1: # match is complete
        return(j - m + 1)
      j,k = j+1,k+1 # extend match
    elif k > 0:
      k = fail[k-1] # use smaller prefix
    else:
      j = j+1
  return(-1) # reached end without match
```

Implementing string search using fail function

- Scan t from beginning
- j is next position in t
- k is currently matched position in p
- If t[j] == p[k] extend the match
- If t[j] != p[k], update match prefix
- If we reach the end of the while loop, no match

```
def find_kmp(t, p):
  n,m = len(t),len(p)
  if m == 0:
    return 0 # pattern is empty
  fail = kmp_fail(p) # preprocessing
  j = 0 # index into text
  k = 0 # index into pattern
  while j < n:
    if t[j] == p[k]: # matched p[0:k+1]
      if k == m - 1: # match is complete
        return(j - m + 1)
      j,k = j+1,k+1 # extend match
    elif k > 0:
      k = fail[k-1] # use smaller prefix
    else:
      j = j+1
  return(-1) # reached end without match
```

Implementing string search using fail function

- Scan t from beginning
- j is next position in t
- k is currently matched position in p
- If t[j] == p[k] extend the match
- If t[j] != p[k], update match prefix
- If we reach the end of the while loop, no match
- Complexity is $O(n)$

```python
def find_kmp(t, p):
  n,m = len(t),len(p)
  if m == 0:
    return 0 # pattern is empty
  fail = kmp_fail(p) # preprocessing
  j = 0 # index into text
  k = 0 # index into pattern
  while j < n:
    if t[j] == p[k]: # matched p[0:k+1]
      if k == m - 1: # match is complete
        return(j - m + 1)
      j,k = j+1,k+1 # extend match
    elif k > 0:
      k = fail[k-1] # use smaller prefix
    else:
      j = j+1
  return(-1) # reached end without match
```

Implementing string search using fail function

- Scan t from beginning
- j is next position in t
- k is currently matched position in p
- If t[j] == p[k] extend the match
- If t[j] != p[k], update match prefix
- If we reach the end of the while loop, no match
- Complexity is $O(n)$
- This finds first match, modify to find all matches

```
def find_kmp(t, p):
  n,m = len(t),len(p)
  if m == 0:
    return 0 # pattern is empty
  fail = kmp_fail(p) # preprocessing
  j = 0 # index into text
  k = 0 # index into pattern
  while j < n:
    if t[j] == p[k]: # matched p[0:k+1]
      if k == m - 1: # match is complete
        return(j - m + 1)
      j,k = j+1,k+1 # extend match
    elif k > 0:
      k = fail[k-1] # use smaller prefix
    else:
      j = j+1
  return(-1) # reached end without match
```

Summary

- The Knuth, Morris, Pratt algorithm efficiently computes the automaton describing prefix matches in the pattern `p`
- Complexity of preprocessing the `fail` function is $O(m)$
- After preprocessing, can check matches in the text `t` in $O(n)$
- Overall, KMP algorithm works in time $O(m + n)$
- However, the Boyer-Moore algorithm can be faster in practice, skipping many positions