Lecture 11: String Matching II

COMP526: Efficient Algorithms

Will Rosenbaum
University of Liverpool

Updated: November 7, 2024

Announcements

1. **NO QUIZ THIS WEEK!**
2. Programming Assignment Posted
   - **TESTING CODE UPDATED**
     - small bug in tritonic array generation
     - download new version
   - Due Wednesday, 13 November
3. Attendance Code:

Meeting Goals

Discuss String Matching procedures:

- Brute Force
- DFA procedure
- Knuth-Morris-Pratt

String Matching

The String Matching Problem

Input:

- A **text** $T \in \Sigma^*$ of length n
- A **pattern** $P \in \Sigma^*$ of length m (typically $m \ll n$)

Output:

- The index of the **first occurrence** of P in T, or -1 if T does not contain P as a substring:
  - $\min\{i \mid T[i, i+m) = P\}$

Example.

- $T = 10110011011101$
- $P_1 = 1101$
  - Output: $i \leftarrow 6$
- $P_2 = 000$
  - Output: $i \leftarrow -1$

Brute Force Matching

Brute Force Matching

Guess an index i where a match might occur

- Possible guesses $i = 0, 1, \ldots, n - m - 1$

Check if match at i:

- is $T[i, i + m) = P$?
- verify each character individually

Cost = number of comparisons made

Brute Force Matching

Guess an index i where a match might occur

- Possible guesses $i = 0, 1, \ldots, n - m - 1$

Check if match at i:

- is $T[i, i + m) = P$?
- verify each character individually

```
procedure VerifyMatch(T, P, i)
    j ← 0
    while j < m do
        if T[i + j] ≠ P[j] then
            return False
        end if
        j ← j + 1
    end while
    return True
end procedure
```

Cost = number of comparisons made

Brute Force Matching

Guess an index i where a match might occur

- Possible guesses $i = 0, 1, \ldots, n - m - 1$

Check if match at i:

- is $T[i, i + m) = P$?
- verify each character individually

```
procedure VerifyMatch(T, P, i)
    j ← 0
    while j < m do
        if T[i + j] ≠ P[j] then
            return False
        end if
        j ← j + 1
    end while
    return True
end procedure
```

Cost = number of comparisons made

PollEverywhere Question

What are the worst case and best case running times of VerifyMatch?

pollev.com/comp526

Brute Force Matching

Guess an index i where a match might occur

- Possible guesses $i = 0, 1, \ldots, n - m - 1$

Check if match at i:

- is $T[i, i + m) = P$?
- verify each character individually

```
procedure VerifyMatch(T, P, i)
    j ← 0
    while j < m do
        if T[i + j] ≠ P[j] then
            return False
        end if
        j ← j + 1
    end while
    return True
end procedure
```

Best and Worst Cases:

Cost = number of comparisons made

Brute Force Matching

Guess an index i where a match might occur

- Possible guesses $i = 0, 1, \ldots, n - m - 1$

Check if match at i:

- is $T[i, i + m) = P$?
- verify each character individually

Cost = number of comparisons made

Brute force. Guess and check every value $i = 0, 1, \ldots, n - m - 1$

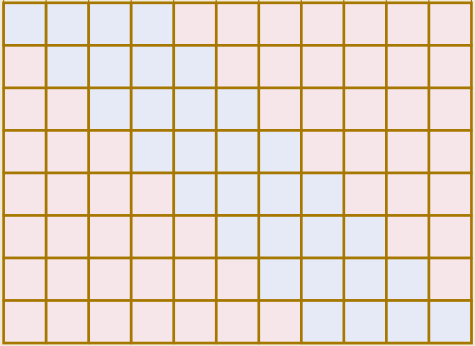
- Worst case running time is $\Theta(nm)$
  - What is example has cost $\Omega(nm)$?
- Best case cost is $\Theta(m)$

Brute Force Example

Example

- $T = abbbababbab$
- $P = abba$

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



procedure
BRUTEFORCEMATCH(T, P)
    for $i = 0, 1, \ldots, n - m - 1$ **do**
        if VERIFYMATCH(T, P, i) **then**
            return i
        end if
    end for
    return -1
end procedure

Brute Force Efficiency

The **worst case** complexity of brute force search is $\Theta(nm)$...

...but when is this **actually** achieved?

Brute Force Efficiency

The **worst case** complexity of brute force search is $\Theta(nm)$...

...but when is this **actually** achieved?

Example. Consider the case where P contains *no repeated characters*.

Brute Force Efficiency

The **worst case** complexity of brute force search is $\Theta(nm)$...

...but when is this **actually** achieved?

Example. Consider the case where P contains *no repeated characters.*

- Claim: brute force search running time is now $O(n)$
  - In fact, at most $2n$ comparisons made!
  - *Why?*

Brute Force Efficiency

The **worst case** complexity of brute force search is $\Theta(nm)$...

...but when is this **actually** achieved?

Example. Consider the case where P contains *no repeated characters*.

- Claim: brute force search running time is now $O(n)$
  - In fact, at most $2n$ comparisons made!
  - *Why?*
- Which of these comparisons were unnecessary?
  - How can you search with fewer comparisons?

Brute Force Efficiency

The **worst case** complexity of brute force search is $\Theta(nm)$...

...but when is this **actually** achieved?

Example. Consider the case where P contains *no repeated characters*.

- Claim: brute force search running time is now $O(n)$
  - In fact, at most $2n$ comparisons made!
  - *Why?*
- Which of these comparisons were unnecessary?
  - How can you search with fewer comparisons?

More generally: How can we use results of *previous comparisons* to avoid making unnecessary comparisons in the future?

- Goal: never re-read a character from T!

Matching with a DFA

Sliding Comparisons

Example

- `T = aabababbabacaa`
- `P = ababaca`

$$a \quad a \quad b \quad a \quad b \quad a \quad b \quad b \quad a \quad b \quad a \quad b \quad a \quad c \quad a \quad a$$

Idea:

- Scan through T keeping track of current matches
- Each new character T read, compare it to next character of P
- If mismatch slide P so that **longest prefix** of P matches

Sliding Comparisons

Example

- `T = aababab babacaa`
- `P = ababaca`

$a \quad a \quad b \quad a \quad b \quad a \quad b \quad b \quad a \quad b \quad a \quad b \quad a \quad c \quad a \quad a$

$a \quad b \quad a \quad b \quad a \quad c \quad a$

Idea:

- Scan through T keeping track of current matches
- Each new character T read, compare it to next character of P
- If mismatch slide P so that **longest prefix** of P matches

Sliding Comparisons

Example

- T = aababababbabacaa
- P = abababaca

$a \quad a \quad b \quad a \quad b \quad a \quad b \quad b \quad a \quad b \quad a \quad b \quad a \quad c \quad a \quad a$

$a \quad b \quad a \quad b \quad a \quad c \quad a$

$\quad\;\; a \quad b \quad a \quad b \quad a \quad c \quad a$

Idea:

- Scan through T keeping track of current matches
- Each new character T read, compare it to next character of P
- If mismatch slide P so that **longest prefix** of P matches

Sliding Comparisons

Example

- T = aababababbabacaa
- P = ababaca

a	*a*	*b*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*a*	*b*	*a*	*c*	*a*	*a*
a	*b*	*a*	*b*	*a*	*c*	*a*									
	a	*b*	*a*	*b*	*a*	*c*	*a*								
			a	*b*	*a*	*b*	*a*	*c*	*a*						

Idea:

- Scan through *T* keeping track of current matches
- Each new character *T* read, compare it to next character of *P*
- If mismatch slide *P* so that **longest prefix** of *P* matches

Sliding Comparisons

Example

- `T = aababaabbabacaa`
- `P = ababaca`

a	*a*	*b*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*a*	*b*	*a*	*c*	*a*	*a*
a	*b*	*a*	*b*	*a*	*c*	*a*									
	a	*b*	*a*	*b*	*a*	*c*	*a*								
			a	*b*	*a*	*b*	*a*	*c*	*a*						
								a	*b*	*a*	*b*	*a*	*c*	*a*	

Idea:

- Scan through T keeping track of current matches
- Each new character T read, compare it to next character of P
- If mismatch slide P so that **longest prefix** of P matches

Representing States and Matches

Question. What information do we need to compute and store to determine next comparison?

Representing States and Matches

Question. What information do we need to compute and store to determine next comparison?

- How many matches in P have we made so far?
- What what is the longest matching prefix for each possible next character in T
  - if we read character x, how far do we need to "slide" P to match a prefix?

Representing States and Matches

Question. What information do we need to compute and store to determine next comparison?

- How many matches in P have we made so far?
- What what is the longest matching prefix for each possible next character in T
  - if we read character x, how far do we need to "slide" P to match a prefix?

Information to store

- **states** that represent number of matches with current prefix of P
- **transitions** from current state to next states, depending on next character read from T

Note. This information depends *only* on the pattern P, not the text T.

DFAs

A **Deterministic Finite Automaton** (**DFA**) consists of:

- A finite set Q of **states**
- A finite **alphabet** Σ
- A **transition function** $\delta : Q \times \Sigma \to Q$
- An **initial state** $q_0 \in Q$
- A set $F \subseteq Q$ of **accepting states**

DFAs

A **Deterministic Finite Automaton** (**DFA**) consists of:

- A finite set Q of **states**
- A finite **alphabet** Σ
- A **transition function** $\delta : Q \times \Sigma \to Q$
- An **initial state** $q_0 \in Q$
- A set $F \subseteq Q$ of **accepting states**

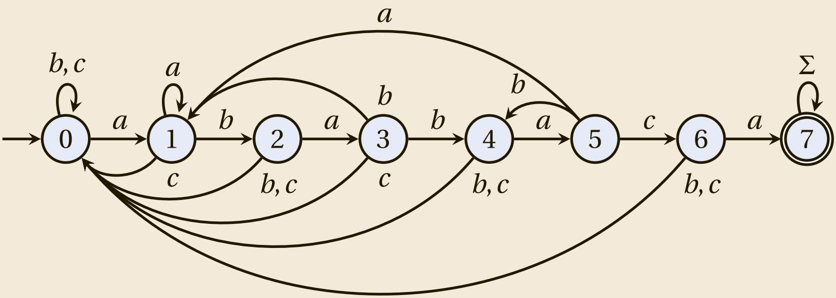
Interpretation. A DFA is used to determine if a string (text) T has some property (e.g., containing a pattern P):

- Start at the state q_0
- Read characters from T sequentially
  - if in state q and read character c, move to state $\delta(q, \sigma)$
- Return TRUE if end in "accepting" state

DFA Example

Example

- T = aabacaababacaa
- P = ababaca



text	a	a	b	a	c	a	a	b	a	b	a	c	a	a
state														

DFA Efficiency

PollEverywhere Question

Given a DFA for matching $P[0, m)$ in $T[0, n)$, what is the running time of applying the DFA? Assume following links is $O(1)$ time.

1. $\Theta(nm)$
2. $\Theta(n \log m)$
3. $\Theta(n + m)$
4. $\Theta(n)$

pollev.com/comp526

DFA Efficiency

Observe: If we are *given* a DFA, executing it

- reads each character of T once
- updates state once per character

$\Longrightarrow$ running time $O(n)$

So the overall running time for pattern matching with a DFA is $O(n)+$ time to build DFA

- assuming computation of δ is $O(1)$.

PollEverywhere Question

Given a DFA for matching $P[0,m)$ in $T[0,n)$, what is the running time of applying the DFA? Assume following links is $O(1)$ time.

1. $\Theta(nm)$
2. $\Theta(n \log m)$
3. $\Theta(n+m)$
4. $\Theta(n)$

pollev.com/comp526

DFA Efficiency

Observe: If we are *given* a DFA, executing it

- reads each character of T once
- updates state once per character

$\Longrightarrow$ running time $O(n)$

So the overall running time for pattern matching with a DFA is $O(n)+$ time to build DFA

- assuming computation of δ is $O(1)$.

But how do we build the DFA?

PollEverywhere Question

Given a DFA for matching $P[0,m)$ in $T[0,n)$, what is the running time of applying the DFA? Assume following links is $O(1)$ time.

1. $\Theta(nm)$
2. $\Theta(n\log m)$
3. $\Theta(n+m)$
4. $\Theta(n)$

pollev.com/comp526

DFA Interpretation & Construction

Semantic Question. What does it *mean* to be in state q?

DFA Interpretation & Construction

Semantic Question. What does it *mean* to be in state q?

- Current position in T matches P up to the first q characters
- Symbolically $T[j-q+1,j] = P[0,q)$

Question. What happens when we read $T[j+1]$?

DFA Interpretation & Construction

Semantic Question. What does it *mean* to be in state q?

- Current position in T matches P up to the first q characters
- Symbolically $T[j-q+1, j] = P[0, q)$

Question. What happens when we read $T[j+1]$?

- If $T[j+1] = P[q]$, transition to state $q+1$
- If $T[j+1] \neq P[q]$, find the length $q' \leq q$ of the longest prefix of P that matches $T[j-q', j+1]$ that matches $P[0, q')$

	a	a	b	a	b	a	b	b	a	b	a	b	a	c	a	a
$q = 5$		a	b	a	b	a	c	a								
$q' = 4$				a	b	a	b	a	c	a						

DFA Interpretation & Construction

Semantic Question. What does it *mean* to be in state q?

- Current position in T matches P up to the first q characters
- Symbolically $T[j-q+1, j] = P[0, q)$

Question. What happens when we read $T[j+1]$?

- If $T[j+1] = P[q]$, transition to state $q+1$
- If $T[j+1] \neq P[q]$, find the length $q' \leq q$ of the longest prefix of P that matches $T[j-q', j+1]$ that matches $P[0, q')$

	a	a	b	a	b	a	b	b	a	b	a	b	a	c	a	a
$q = 5$		a	b	a	b	a	c	a								
$q' = 4$				a	b	a	b	a	c	a						

- **Insight:** if $T[j+1] = c$ this is the same as matching $P[0..q]$ against $P[1..q)c$
  - we can use the DFA constructed so far to find this!

DFA Interpretation & Construction

- **Insight:** if $T[j+1] = c$ this is the same as matching $P[0..q]$ against $P[1..q)c$
  - we can use the DFA constructed so far to find this!

Inductive Construction.

- Start with states 0 and 1 with

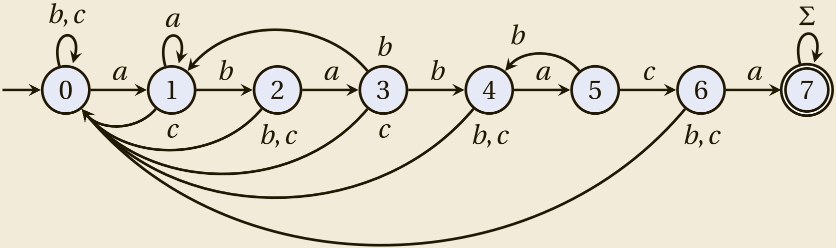
$$\delta(0, c) = \begin{cases} 1 & \text{if } P[0] = c \\ 0 & \text{otherwise.} \end{cases}$$

- Once we've constructed DFA up to state q:
  - take $\delta(q, P[q]) = q+1$
  - for $c \neq P[q]$, find $\delta(q, c)$ by applying DFA to $P[1, q)c$

DFA Interpretation & Construction

- **Insight:** if $T[j+1] = c$ this is the same as matching $P[0..q]$ against $P[1..q)c$
  - we can use the DFA constructed so far to find this!
- Once we've constructed DFA up to state q:
  - take $\delta(q, P[q]) = q+1$
  - for $c \neq P[q]$, find $\delta(q,c)$ by applying DFA to $P[1,q)c$

Example. Compute $\delta(5, a)$ for $P =$ `ababaca`.



DFA Interpretation & Construction

- **Insight:** if $T[j+1] = c$ this is the same as matching $P[0..q]$ against $P[1..q)c$
  - we can use the DFA constructed so far to find this!

Inductive Construction.

- Start with states 0 and 1 with

$$\delta(0,c) = \begin{cases} 1 & \text{if } P[0] = c \\ 0 & \text{otherwise.} \end{cases}$$

- Once we've constructed DFA up to state q:
  - take $\delta(q, P[q]) = q+1$
  - for $c \neq P[q]$, find $\delta(q,c)$ by applying DFA to $P[1,q)c$

Analysis (idea).

- Argue by induction on q that the DFA enters state q on reading $T[j]$ if and only if q is the largest number such that $T[j-q+1, j] = P[0,q)$.

DFA Lookup Table Construction

DFA *diagrams* are great for humans, but not so great for computers…

DFA Lookup Table Construction

DFA *diagrams* are great for humans, but not so great for computers…

Problems.

1. How do we represent the DFA in a computer friendly format?
2. How do construct the DFA in that format efficiently?

DFA Lookup Table Construction

Problems.

1. How do we represent the DFA in a computer friendly format?
2. How do construct the DFA in that format efficiently?

Solutions.

1. Store a **lookup table** $\delta[][]$
    - columns = states, rows = characters
    - $\delta[q][c] \leftarrow \delta(q, c)$

DFA Lookup Table Construction

Problems.

1. How do we represent the DFA in a computer friendly format?
2. How do construct the DFA in that format efficiently?

Solutions.

1. Store a **lookup table** $\delta[][]$
    - columns = states, rows = characters
    - $\delta[q][c] \leftarrow \delta(q, c)$
2. Compute column by column
    - trick: keep track of state for $P[1, q)$ because we'll reuse this for each $P[1, q)c$

DFA Lookup Table Construction

Solutions.

1. Store a **lookup table** $\delta[\,][\,]$
   - columns = states, rows = characters
   - $\delta[q][c] \leftarrow \delta(q, c)$
2. Compute column by column
   - trick: keep track of state for $P[1, q)$ because we'll reuse this for each $P[1, q)c$
   - x is largest value with $P[0, x) = P[q - x, q]$

```
procedure ConstructDFA(P[0..m))
    for c ∈ Σ do
        δ[0][c] ← 0
    end for
    δ[0][P[0]] ← 1
    x ← 0
    for q = 1, 2, ..., m − 1 do
        for c ∈ Σ do
            δ[q][c] ← δ[x][c]
        end for
        δ[q][P[q]] ← q + 1
        x ← δ[x][P[q]]
    end for
end procedure
```

DFA Lookup Table Construction

Example. $P[0..6) = $ `ababaca`

$\delta(c,q)$	0	1	2	3	4	5	6
$P[q]$	a	b	a	b	a	c	a
`a`							
`b`							
`c`							

procedure CONSTRUCTDFA($P[0..m)$)
    for $c \in \Sigma$ **do**
        $\delta[0][c] \leftarrow 0$
    end for
    $\delta[0][P[0]] \leftarrow 1$
    $x \leftarrow 0$
    for $q = 1, 2, \ldots, m-1$ **do**
        for $c \in \Sigma$ **do**
            $\delta[q][c] \leftarrow \delta[x][c]$
        end for
        $\delta[q][P[q]] \leftarrow q+1$
        $x \leftarrow \delta[x][P[q]]$
    end for
end procedure

DFA Lookup Table Construction

PollEverywhere Question

What is the running time of ConstructDFA when P has length m and $|\Sigma| = s$?

pollev.com/comp526

procedure ConstructDFA($P[0..m)$)
  for $c \in \Sigma$ **do**
    $\delta[0][c] \leftarrow 0$
  end for
  $\delta[0][P[0]] \leftarrow 1$
  $x \leftarrow 0$
  for $q = 1, 2, \ldots, m-1$ **do**
    for $c \in \Sigma$ **do**
      $\delta[q][c] \leftarrow \delta[x][c]$
    end for
    $\delta[q][P[q]] \leftarrow q + 1$
    $x \leftarrow \delta[x][P[q]]$
  end for
end procedure

DFA Lookup Table Application

Pitting it Together

- Construct the DFA
- Apply the DFA

```
procedure ApplyDFA(T[0..n], δ, m)
    q ← 0
    for i = 0, 1, ..., n − 1 do
        q ← δ[q][T[i]]
        if q = m then
            return i
        end if
    end for
    return −1
end procedure
procedure DFAMatch(P[0..m), T[0..n))
    δ ← ConstructDFA(P, T)
    return ApplyDFA(T, δ, m)
end procedure
```

DFA Lookup Table Application

Pitting it Together

- Construct the DFA
- Apply the DFA
- Running time is $\Theta(n + m|\Sigma|)$
  - $\Theta(m|\Sigma|)$ for making DFA
  - $\Theta(n)$ for applying DFA
- Additional space overhead: $\Theta(m|\Sigma|)$
  - store the DFA

```
procedure ApplyDFA(T[0..n], δ, m)
    q ← 0
    for i = 0, 1, ..., n−1 do
        q ← δ[q][T[i]]
        if q = m then
            return i
        end if
    end for
    return −1
end procedure
procedure DFAMatch(P[0..m), T[0..n))
    δ ← ConstructDFA(P, T)
    return ApplyDFA(T, δ, m)
end procedure
```

Knuth-Morris-Pratt

Failure Link Automaton

DFA efficiency.

- Space/time to build DFA: $\Theta(m|\Sigma|)$
- Time to execute DFA: $\Theta(n)$

$\Longrightarrow$ Overall time is $\Theta(n + m|\Sigma|)$

  - additional space overhead is $\Theta(m|\Sigma|)$

Question. Can we perform string matching in time $O(n)$ with *less space overhead?*

Failure Link Automaton

DFA efficiency.

- Space/time to build DFA: $\Theta(m|\Sigma|)$
- Time to execute DFA: $\Theta(n)$

$\Longrightarrow$ Overall time is $\Theta(n+m|\Sigma|)$

  - additional space overhead is $\Theta(m|\Sigma|)$

Question. Can we perform string matching in time $O(n)$ with *less space overhead?*

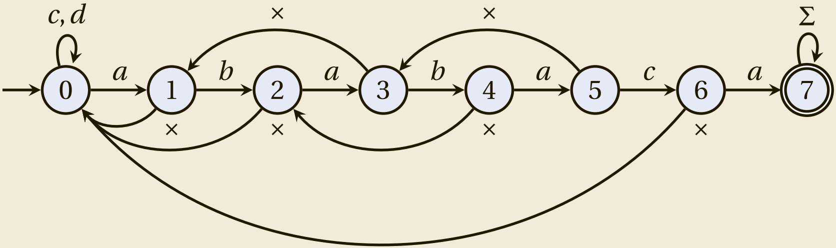
Idea. When comparison fails, don't have a separate transition for each failing character

- Just record failure and "shift" pattern as far forward as possible

Failure Link Automaton

Example

- T = aabacaababacaa
- P = ababaca



text	a	a	b	a	c	a	a	b	a	b	a	c	a	a
states														

FLA Execution

A **Failure Link Automaton** (**FLA**) consists of:

- A finite set Q of **states**
- A finite **alphabet** Σ
- A **transition function** $\varphi : Q \times (\Sigma \cup \{\times\}) \to Q$
- An **initial state** $q_0 \in Q$
- A set $F \subseteq Q$ of **accepting states**

FLA Execution

A **Failure Link Automaton** (**FLA**) consists of:

- A finite set Q of **states**
- A finite **alphabet** Σ
- A **transition function** $\varphi : Q \times (\Sigma \cup \{\times\}) \to Q$
- An **initial state** $q_0 \in Q$
- A set $F \subseteq Q$ of **accepting states**

Execution. To apply and FLA to T

- Start at the state q_0
- Read characters from T sequentially
  - if in state q and read character c:
    - if $\varphi(q, c)$ is defined, move to state $\varphi(q, c)$
    - otherwise move to state $\varphi(q, \times)$ and **re-read** c
- Return TRUE if end in "accepting" state

FLA Execution

PollEverywhere Question

Given an FLA for a pattern P of length m, how many times could we follow failure links for a single character c read from T in the worst case?

pollev.com/comp526

Execution. To apply and FLA to T

- Start at the state q_0
- Read characters from T sequentially
  - if in state q and read character c:
    - if $\varphi(q, c)$ is defined, move to state $\varphi(q, c)$
    - otherwise move to state $\varphi(q, \times)$ and **re-read** c
- Return TRUE if end in "accepting" state

FLA Execution

Execution. To apply and FLA to T

- Start at the state q_0
- Read characters from T sequentially
  - if in state q and read character c:
    - if $\varphi(q, c)$ is defined, move to state $\varphi(q, c)$
    - otherwise move to state $\varphi(q, \times)$ and **re-read** c
- Return TRUE if end in "accepting" state

FLA Running Time

More careful analysis

- If we match up to $P[j]$, then we can only follow up to j back links
- In order to witness j failures, must have witnessed j successes!

FLA Running Time

More careful analysis

- If we match up to $P[j]$, then we can only follow up to j back links
- In order to witness j failures, must have witnessed j successes!

Amortized cost of each character read from T

- If read character c is a **match**:
  - pay 1 for comparison
  - put 1 unit cost in the **bank**
- If read character c is a **mismatch**
  - *withdraw* 1 from the bank
- By analysis above account balance is always non-negative

$\implies$ amortized cost of each comparison is 2

$\implies$ hence overall running time of execution is $O(n)$

FLA Construction

Observation. Each state q has

- 1 forward link to state $q+1$
- 1 fail link

Given P, we don't need to store forward link label:

- forward link label from $P[q] = P[q+1]$

Only need to store fail link state!

- this can be stored as a single array of size m

$\Longrightarrow$ only $O(m)$ space overhead

FLA Construction

Definition. The **failure link array** *fail* of P the array of m numbers that stores the (index of) the next state for each failure

- How do we construct it?

FLA Construction

Definition. The **failure link array** *fail* of P the array of m numbers that stores the (index of) the next state for each failure

- How do we construct it?
- Again x is length of largest prefix that matches a suffix of $P[1, q)$

Example. $P[0..6) =$ `abababaca`

q	0	1	2	3	4	5	6
$P[q]$	a	b	a	b	a	c	a
$fail[q]$							

```
procedure FAILURELINK(P[0, m))
    fail[0] ← 0
    x ← 0
    for j = 1, 2, ..., m − 1 do
        fail[j] ← x
        while P[x] ≠ P[j] do
            if x = 0 then
                x ← −1
                break
            else
                x ← fail[x]
            end if
        end while
        x ← x + 1
    end for
end procedure
```

FLA Construction

Question. What is the running time of FAILURELINK on input of size m?

```
procedure FailureLink(P[0, m))
    fail[0] ← 0
    x ← 0
    for j = 1, 2, ..., m − 1 do
        fail[j] ← x
        while P[x] ≠ P[j] do
            if x = 0 then
                x ← −1
                break
            else
                x ← fail[x]
            end if
        end while
        x ← x + 1
    end for
end procedure
```

FLA Construction

Question. What is the running time of FAILURELINK on input of size m?

Observations.

- x incremented once per j
- $fail[x] < x$
- Each "while" iteration decrements x

So at most $2m$ updates to x

- cf. amortized analysis
- x = bank balance

```
procedure FailureLink(P[0, m))
    fail[0] ← 0
    x ← 0
    for j = 1, 2, ..., m − 1 do
        fail[j] ← x
        while P[x] ≠ P[j] do
            if x = 0 then
                x ← −1
                break
            else
                x ← fail[x]
            end if
        end while
        x ← x + 1
    end for
end procedure
```

KMP Algorithm

Question. How do we apply the failure link array to find a match?

KMP Algorithm

Question. How do we apply the failure link array to find a match?

- Scan along $T[0, n)$
  - index i
- Maintain position in $P[0, m)$
  - index j
  - current prefix match
- When $T[i] = P[j]$, increment i and j
- Otherwise, $j \leftarrow \mathit{fail}[j]$
  - unless $j = 0$, then $i \leftarrow i + 1$

KMP Algorithm

Question. How do we apply the failure link array to find a match?

- Scan along $T[0, n)$
  - index i
- Maintain position in $P[0, m)$
  - index j
  - current prefix match
- When $T[i] = P[j]$, increment i and j
- Otherwise, $j \leftarrow fail[j]$
  - unless $j = 0$, then $i \leftarrow i + 1$

```
procedure KMP(T[0..n), P[0..m))
    fail ← FailureLink(P)
    i ← 0
    j ← 0
    while i < n do
        if T[i] = P[q] then
            i ← i + 1, j ← j + 1
            if j = m then return i − j
        else
            if j ≥ 1 then
                j ← fail[j]
            else
                i ← i + 1
            end if
        end if
    end while
end procedure
```

KMP Algorithm

Analysis:

- Running time $O(n+m)$
    - $O(m)$ to build *fail*
    - $O(n)$ to apply KMP
    - analysis uses **amortized analysis**
- Additional space $O(m)$
    - just need to store *fail* and indices

```
procedure KMP(T[0..n), P[0..m))
    fail ← FailureLink(P)
    i ← 0
    j ← 0
    while i < n do
        if T[i] = P[q] then
            i ← i + 1, j ← j + 1
            if j = m then return i − j
        else
            if j ≥ 1 then
                j ← fail[j]
            else
                i ← i + 1
            end if
        end if
    end while
end procedure
```

KMP Algorithm

Analysis:

- Running time $O(n+m)$
  - $O(m)$ to build *fail*
  - $O(n)$ to apply KMP
  - analysis uses **amortized analysis**
- Additional space $O(m)$
  - just need to store *fail* and indices

Clean Takeaway:

fail$[j]$ is the length of the longest prefix of $P[0..j]$ that is a suffix of $P[1..j]$

```
procedure KMP(T[0..n), P[0..m))
    fail ← FAILURELINK(P)
    i ← 0
    j ← 0
    while i < n do
        if T[i] = P[q] then
            i ← i + 1, j ← j + 1
            if j = m then return i − j
        else
            if j ≥ 1 then
                j ← fail[j]
            else
                i ← i + 1
            end if
        end if
    end while
end procedure
```

DFA vs FLA

Question. Which is better? DFA matching or KMP algorithm?

- KMP has overall running time $O(n+m)$
  - amortized 2 comparisons per T access
- DFA has overall running time $O(n+m|\Sigma|)$
  - 1 comparison per T access
  - $|\Sigma|$ dependence

Next Time

More String Matching!

Scratch Notes