CS1332 Fall 2018 - Week 11

10/29/18

Topics

- I. Pattern Matching Intro
  - A. Link to examples we used in class: https://docs.google.com/spreadsheets/d/1yPo81_QUnY5YePIsB8eQeBq-BjBRR99P_-pyDcohtXo/edit?usp=sharing
  - B. Pattern Matching/String Searching - given a pattern of characters and a body of text, search through the text to find a match to the pattern
  - C. Algorithms
    - 1. Brute Force - like bubble sort, intuitive, but not efficient → if you have a mismatch, you shift where you started looking in the text by one
    - 2. Boyer-Moore and KMP - smartly shifts when a mismatch occurs
    - 3. Rabin-Karp - like radix sort, decomposes the problem → the pattern and substrings of the text are hashed, when the hashcodes match, you compare the characters
- II. Brute Force
  - A. Terminology
    - 1. n - length of the text (body of text we are searching in)
    - 2. m - length of the pattern (string we are searching for)
  - B. Algorithm
    - 1. t: loop from index 0 to n-m (starting index in text)
      - a) i: loop from index 0 to m-1 (index in pattern)
        - (1) compare pattern[i] and text[t + i]
        - (2) if they match:
          - (a) if i < m-1 → keep searching
          - (b) if i == m-1 → you found a match!
        - (3) if they do not match → exit inner loop
  - C. Efficiency
    - 1. Worst case - O(mn)
      - a) Eg. text “aaaaaaaaaaaaaaaa”, pattern “aaaaaab”
    - 2. Best case (searching for all occurrences) - O(mn)
    - 3. Best case (searching for just the first occurrence) - O(m)
      - a) Eg. text “aaabbbbaaabbabab”, pattern “aaab”
- III. Boyer-Moore
  - A. See Canvas for some papers about Boyer-Moore
  - B. Concept
    - 1. Preprocess the pattern
    - 2. Start with the pattern aligned at the front of the text and shift it right, but start comparisons from the end of the pattern
  - C. Preprocessing

1. *Last occurrence table* - records the index at which each letter in the pattern appears last (anything not in the pattern's alphabet is represented by * and will return a -1 upon querying the table during the algorithm)
    - a) Eg. pattern "abacab"

Character	a	b	c	*
Last Occurrence	4	5	3	-1

- D. Algorithm
    1. Align index 0 of the pattern with index 0 of the text, start comparing characters at the back of the pattern
        - a) If they match → decrement and compare again
        - b) If they do not match → query the last occurrence table for the character *in the text* that mismatched
            - (1) If the query returns a non-negative value that has not yet been passed, realign the pattern so the index of the last occurrence aligns with the mismatch in the text
            - (2) If the query returns a non-negative value that has already been passed (eg. the last occurrence index is greater than the one we're currently at), shift the pattern to the right by 1
            - (3) If the query returns -1, shift the pattern over this character
- E. Optimal Scenarios
    1. Large alphabets - there is a greater chance for characters that do not exist in the pattern to exist in the text

<u>Activities</u>

- I. Sorting Reality Check

10/31/18

<u>Topics</u>

- I. Boyer-Moore Continued
    - A. Time Complexity
        1. Worst case - O(mn)
            - a) Eg. text "aaaaaaaaaaaaaaaa", pattern "baaaaaa"
        2. Best case (searching for all occurrences) - O(m + n)
            - a) O(m) to generate last occurrence table
            - b) O(n) to look at all the characters in the text roughly once
        3. Best case (searching for just the first occurrence) - O(m)
            - a) Eg. text "aaabbbbaaabbabab", pattern "aaab"
- II. Knuth-Morris-Pratt (KMP)
    - A. Concept
        1. Preprocess the pattern → locate the lengths of the prefixes in the pattern that are also suffixes of different substrings in the pattern

B. Preprocessing

   1. *Failure Table* - for each index, records the length of the prefix that is also a suffix in the substring from 0 to the current index
      a) Prefix - letters at the beginning of a string
      b) Suffix - letters at the end of a string
   2. Eg. pattern “revararev”

index	0	1	2	3	4	5	6	7	8
char	r	e	v	a	r	a	r	e	v
failure	0	0	0	0	1	0	1	2	3

      a) At index 4, the substring we are looking at is “revar”, there is only one letter at the beginning that is repeated at the end.
      b) At index 8, the substring we are looking at is “revararev”, the length of the prefix that is also a suffix is 3.

C. Failure Table Building Algorithm

   1. p = pattern, i = index in prefix, j = index in pattern, f[] = failure table
   2. i = 0, j = 1, f[0] = 0
   3. while i < m (length of pattern)
      a) case 1: p[i] == p[j] → characters match
         (1) f[j] = i+1
         (2) i++, j++
      b) case 2: p[i] != p[j] && i = 0 → characters do not match, and we have not built up a prefix
         (1) f[j] = 0
         (2) j++
      c) case 3: p[i] != p[j] && i > 0 → characters do not match, but we’ve built up a prefix
         (1) reset i = f[i-1] (if the characters don’t match, we can’t increase the length of the prefix, so we try a shorter prefix)
   4. Eg. pattern “revararev”, mismatch, match, built prefix

```
      i     j
idx   0     1     2     3     4     5     6     7     8
char  r     e     v     a     r     a     r     e     v
f[]   0
```

```
      i     j
idx   0     1     2     3     4     5     6     7     8
char  r     e     v     a     r     a     r     e     v
f[]   0     0
```

	i		j						
idx	0	1	2	3	4	5	6	7	8
char	r	e	v	a	r	a	r	e	v
f[]	0	0	0						

	i			j					
idx	0	1	2	3	4	5	6	7	8
char	r	e	v	a	r	a	r	e	v
f[]	0	0	0	0					

	i				j				
idx	0	1	2	3	4	5	6	7	8
char	r	e	v	a	r	a	r	e	v
f[]	0	0	0	0	1				

		i				j			
idx	0	1	2	3	4	5	6	7	8
char	r	e	v	a	r	a	r	e	v
f[]	**0**	0	0	0	1				

	i					j			
idx	0	1	2	3	4	5	6	7	8
char	r	e	v	a	r	a	r	e	v
f[]	0	0	0	0	1	0			

	i						j		
idx	0	1	2	3	4	5	6	7	8
char	r	e	v	a	r	a	r	e	v
f[]	0	0	0	0	1	0	1		

		i						j	
idx	0	1	2	3	4	5	6	7	8
char	r	e	v	a	r	a	r	e	v
f[]	0	0	0	0	1	0	1	2	

			i						j
idx	0	1	2	3	4	5	6	7	8
char	r	e	v	a	r	a	r	e	v
f[]	0	0	0	0	1	0	1	2	3

11/2/18

Topics

I. KMP Continued

A. Terminology:

1. m - length of pattern
2. n - length of text
3. j - index in pattern
4. k - index in text
5. p[j] - char in pattern
6. t[k] - char in text
7. f[] - failure table

B. Algorithm

1. while no match and k < n
   a) case 1: if p[j] == t[k] → j++, k++
      (1) if a complete match is found (j == m) → set j = f[j], k++
   b) case 2: if p[j] != t[k] && j = 0 → k++
   c) case 3: if p[j] != t[k] && j != 0 → reset j = f[j-1]

C. Tracing an example

1. Failure table

idx	0	1	2	3	4	5	6
pattern	t	h	e	a	t	h	a
failure	0	0	0	0	1	2	0

2. Tracing

t	**h**	**e**	**-**	**t**	**h**	**e**	**a**	**t**	**h**	**-**	**t**	**h**	**e**	**a**	**t**	**h**	**e**	**a**	**t**	**h**	**a**
t	h	e	a	t	h	a	*fail at j=3, stay at place in txt*, move to f[2] in pat*														
			t	h	e	a	t	h	a	*fail at j=0, move by 1 in txt***											
				t	h	e	a	t	h	a	*fail at j=6, *, move to f[5] in pat*										
								t	h	e	a	t	h	a	*fail at j=2, move to f[1]*						
										t	h	e	a	t	h	a	**				
											t	h	e	a	t	h	a	*move to f[5]*			
															t	h	e	a	t	h	a

- D. Time Complexity
    1. Worst case - O(m + n)
    2. Best case (searching for all occurrences) - O(m + n)
        - a) O(m) to generate last occurrence table
        - b) O(n) to look at all the characters in the text roughly once
    3. Best case (searching for just the first occurrence) - O(m)

Activities

- I. Sorting reality check