CS3230R Presentation (13 Mar 2012)

Text Searching Algorithms

Presented by: Aldrian Obaja

What is Text Searching?

- "... to find within a text *t* a match for a pattern *p*, where *text*, *pattern*, and *match* can be interpreted in different ways"
- The different ways:
    - Simple text searching
    - Rabin-Karp algorithm
    - Knuth-Morris-Pratt algorithm
    - Boyer-Moore(-Horspool) algorithm
    - Approximate matching
    - Regular expression

Why do we want Text Searching?

- Search for a student's name in a students list
- Search for a word or phrase in a document
- Search for approximate file name in a directory
- Search for a phone number in a form

Assumptions and Notations

- We are searching on English documents, containing English alphabets and punctuations
- Notations we will use in this presentation (and also in the book):
    - Big-O notation for time complexity
    - p for pattern to be searched, t for source text
    - $m = |p|$ and $n = |t|$

Simple Text Searching

- The simplest to implement
- Compare *t* against *p* letter by letter until match or end of text is found
- Example we are going to search for "001" in "010001"

Example

```
j = 0
|
001
010001
|
i = 0
```

```
j = 1
 |
001
010001
|
i = 0
```

```
j = 0
 |
 001
010001
 |
i = 1
```

```
j = 0
  |
  001
010001
  |
i = 2
```

```
j = 1
   |
  001
010001
   |
i = 2
```

```
j = 2
    |
  001
010001
  |
i = 2
```

```
j = 0
   |
   001
010001
   |
i = 3
```

```
j = 1
    |
   001
010001
    |
i = 3
```

```
j = 2
     |
   001
010001
   |
i = 3
```

Match found at i=3

Too slow!

- This algorithm runs in O(m*n) since for each character in t we need to start matching with p until mismatch or until end of text
- For large documents and length of the pattern, this can be really slow
- Say look up for term "text searching" in the book "Algorithms"
- Can we do better than this?

Rabin-Karp Algorithm

- Idea: create a fingerprint for every substring in t that has the same length as p
- Compare the letters only when the fingerprint match
- We need to find fingerprint function that fulfills:
  - Maps strings with length m to q values
  - Distribute the string evenly among the q values
  - Easy to compute

Example

p = "0011", t = "0001001000"

f[i] = parity of the string t[i..i+3]

i	0	1	2	3	4	5	6	7	8	9
t[i]	0	0	0	1	0	0	1	0	0	0
f[i]	1	1	1	0	1	1	1			

Table 1: calculation of f[i]

Since the parity of p is 0, we check for matching only at those positions where $f[i]$=0
In this case, it is only at i=3, which after checking it doesn't match p. So return -1

Complexity Analysis

- Assuming the fingerprint for p and t can be calculated in $O(m)$ and $O(n)$ respectively and the fingerprint distributes the substring into q values evenly, this algorithm is expected to compare p only with $1/q$ of the total substring in t, resulting in $O(m + m*n/q) = O(m+n)$ on average if $q > m$
- Note that this is expectation only, since the substring might not be evenly distributed

Knuth-Morris-Pratt Algorithm

- During searching, often we have found partial match, which means we already have some information about the text and the pattern, can we use this information to speed up the search?
- Idea: Use the information of the pattern and the text to shift the position of comparison hopefully more than 1 position

Example

Searching for "Tweedledum" in "Tweedledee and Tweedledum"

*Tweedled*um

*Tweedlede*e and Tweedledum

Shift 8 positions since the partial text "Tweedled" does not contain any "T" after the first "T" to be matched with the "T" in the pattern

        Tweedledum

Tweedledee and Tweedledum

Example

Searching for "pappar" in "pappappapparrassan"

pappar

pappappapparrassan

↓ Shift 3 positions

***pa**ppar*

pappappapparrassan

↓ Shift 3 positions

***pa**ppar*

pappappapparrassan

Shift Table

- The idea is to use partial match so that after shifting either we shift the pattern entirely or there are still partial match
- If $p[0..k]$ has matched $t[i..i+k]$, then shifting s position means $p[0..k\text{-}s]$ should match $t[i+s..i+k]$
- Since $p[0..k\text{-}s] = t[i+s..i+k] = p[s..k]$, then the shift table at index k contains the number minimum $s > 0$ such that $p[0..k\text{-}s] = p[s..k]$

Algorithm

```
knuth_morris_pratt_search(p,t){
1  m = p.length
2  n = t.length
3  knuth_morris_pratt_shift(p,shift)
4  i=0, j=0
5  while(i+m<=n){
6     while(t[i+j]==p[j]){
7        j=j+1
8        if(j>=m) return i
9     }
10    i = i+shift[j-1]
11    j = max(j-shift[j-1],0)
12 }
13 return -1
}
```

Algorithm

```
knuth_morris_pratt_shift(p,shift){
1  m = p.length
2  shift[-1] = 1, shift[0] = 1
3  i=1, j=0
4  while(i+j<m){
5     if(p[i+j]==p[j]){
6        shift[i+j] = i
7        j=j+1
8     } else {
9        if(j==0) shift[i] = i+1
10       i = i+shift[j-1]
11       j = max(j-shift[j-1],0)
12    }
13 }
}
```

Complexity Analysis

- The algorithm runs in $O(n+m)$ time
- Proof: tracing the value of $2i+j$ at line 6
  - If the condition is true, then j, and hence $2i+j$ increases by 1
  - If the condition is false, then $2i+j$ increases by at least *shift*[j-1], which is at least 1
  - Since $2i+j < 2(n-m)+m$, then line 6 can be executed at most $2n-m$ times, which is $O(n)$, and combined with the preprocessing of pattern in $O(m)$, total the algorithm runs in $O(n+m)$

Boyer-Moore

- Idea: why don't search from right to left?
- Algorithm: Implemented with simple text searching plus two heuristics: occurence and match
- Occurence: If we know the character in $t[i+m\text{-}1]$ is not contained in the pattern, then we can shift over this position to $i=i+m$
- Match: Use partial match – similar to Knuth-Morris-Pratt shift table, but for reverse pattern

Boyer-Moore-Horspool

- Horspool suggested the use of shifting based on the last compared letter in the text, to match the last occurence of that letter to the left of $p[m\text{-}1]$
- In practice, this runs fast, although the worst case runtime is $O(mn)$

Examples

| → | | → | |
kettle → kettle → **kettle**
tea kettle tea **kettle** tea k**e**ttle
| | |

| | |
date → date → date
detective det**e**ctive detective
| | |

Approximate Pattern Matching

- Search not for exact match, but for close match
- Mainly use "Edit distance":
  - The distance between two strings defined as the number of edits required to transform one string to the other
- Best approximate match:
  - From a text to find a substring with lowest edit distance from the pattern

Examples

- The result of searching "retrieve" in "retreive,retreeve,retreev" will result in the substring "retreeve" at position 9 with distance 1 (the minimum possible for this text)

Don't Care Match

- A variant of text searching, using additional symbol '?' that can match any character.
- Assume we have algorithm that find exact match of a set of pattern, in a text that runs in O(m+n) (Aho and Corasick algorithm)
- Then we can search for the don't care match by splitting the pattern into set of pattern by the '?' symbol and then running the modified version of above algorithm

Regular Expression

- To search for strings that follow a specific pattern
- Examples: phone number (90589284), master card number (1234-5678-9012-3456), file extension (any.jpg, some.txt,home.html), e-mail address (somebody@somedomain.com)
- Uses three features: **concatenations, alternations**, and **repetitions**

Features of Regular Expression

- **concatenations:** simplest feature, allow a search for a concatenated list of characters: "html", "abcd". Simbolized as "·"
- **alternations:** allows a search from a list of options, example: (jpg|gif) would match the string "jpg" and the string "gif"
- **repetition:** allows a search for repetitive pattern, which makes the query pattern easier. Example: "(0|1)*" will match any binary text such as "000011" or "11001"

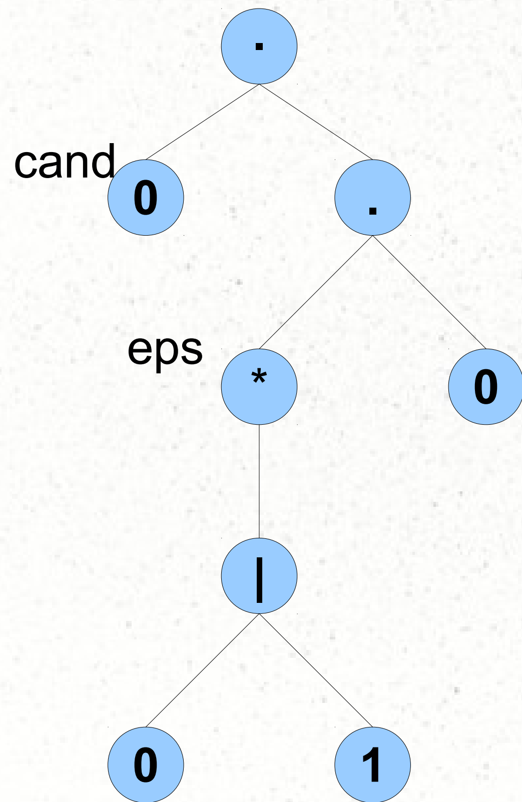
Notations

- Empty string: a string that has no length
- Candidates: the set of nodes that can be matched with next letter

Regex Tree

- To do matching using regex, we use the tree representation of the query pattern
- A node may be either one of these:
  - Concatenation node (2 children)
  - Alternation node (2 children)
  - Repetition node (1 child)
  - Leaf node (no children)

Regex Tree



The regex tree for query pattern: 0(0|1)*0

Regex Matching

Require four methods:

- **eps**: to mark trees that can match empty string
- **start**: to mark the initial candidates
- **match_letter**: to match a letter with the candidates
- **next**: to find the next candidates

next Algorithm

```
next(t,mark){
1  if(t.value=="."){
2     next(t.left,mark)
3     if(t.matched || (t.eps && mark)){
4        next(t.right,true)
5     } else {
6        next(t.right,false)
7     }
8  } else if (t.value=="|"){
9     next(t.left,mark), next(t.right,mark)
10 } else if (t.value=="*"){
11    if(t.matched)next(t.left,true)
12    else next(t.left,mark)
13 } else {
14    t.cand = mark
}
```

match_letter Algorithm

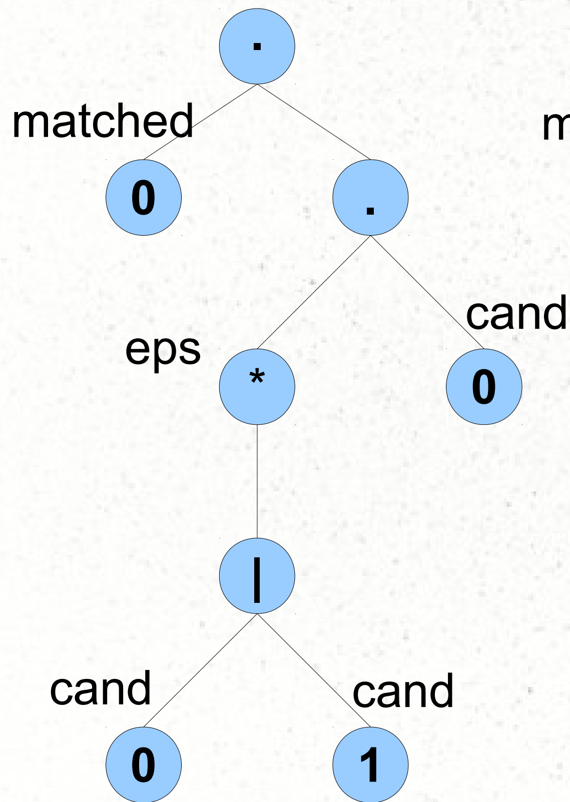
```
match_letter(t,a){
1  if(t.value==".")){
2     match_letter(t.left,a)
3     t.matched = match_letter(t.right,a)
4  } else if (t.value=="|"){
5     t.matched = match_letter(t.left,a) ||
                  match_letter(t.right,a)
6  } else if (t.value=="*"){
7     t.matched = match_letter(t.left,a)
8  } else {
9     t.matched = t.cand && (a==t.value)
10    t.cand = false
11 }
12 return t.matched
}
```

Matching Algorithm

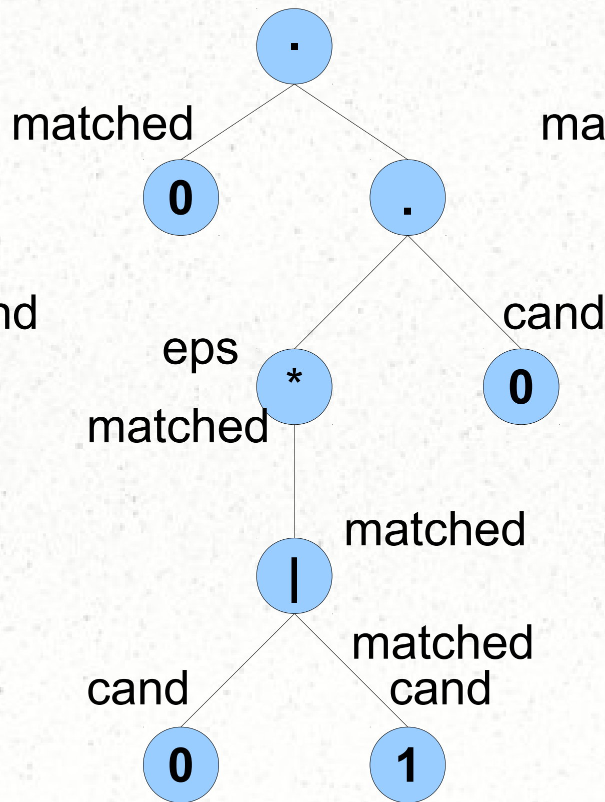
```
match(w,t){
1  n = w.length
2  epsilon(t)
3  start(t)
4  i=0
5  while(i<n){
6     match_letter(t,w[i])
7     if(t.matched)
8        return true
9     next(t,false)
10    i=i+1
11 }
12 return -1
}
```

Example

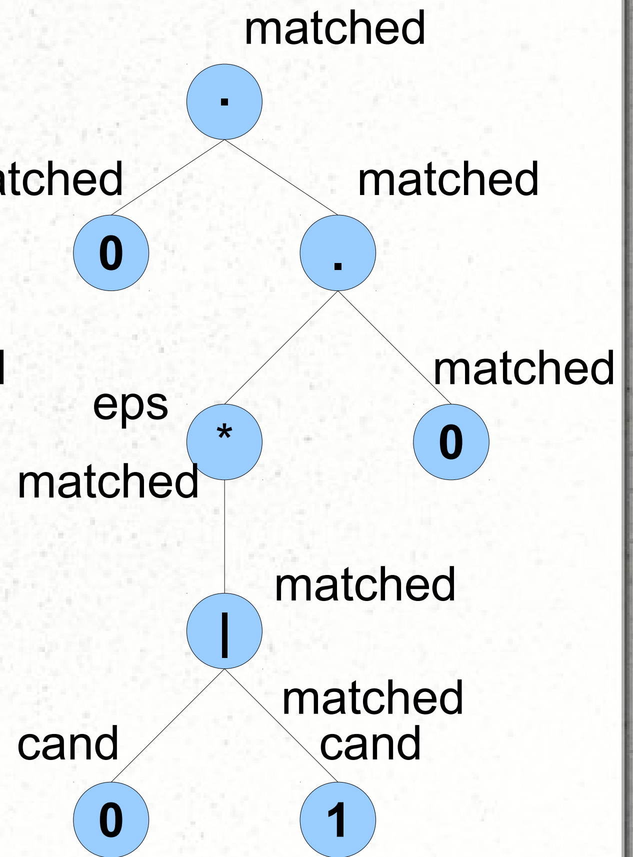
Try to match the text "010"



After matching 0 and call to `next`



After matching 1 and call to `next`



After matching 0 and call to `next`

Regex Search

- Similar to regex match, the difference is just in determining the next candidate after matching (refer to the `match` algorithm)
- In searching, the match might not be from the initial, so after matching we want to still include the start candidates as candidates
- Minor change at line 9 to **`next(t,true)`**

References

- This presentation is adapted from the Chapter 9 of the book **Algorithms** by *Richard Johnsonbaugh* and *Marcus Schaefer*

Thank You