Course Notes for

CS 1501
Algorithm Implementation

By
John C. Ramirez
Department of Computer Science
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

- KMP (Knuth Morris Pratt) string matching
  - Improves the worst case, but not the normal case
  - Idea is to prevent index from ever going "backward" in the text string
    - This will guarantee Theta(N) runtime in the worst case
    - Think about the worst case for brute force:

A = XXXXXXXXXXXXXXXXXXXXXXXXXXXY

P = XXXXY

- – When we shift the pattern down, we "re-match" M-2 of the Xs that matched the first time

KMP

  - We then match the X that previously failed, and then mismatch on the Y
  - This is done over and over as we move down the text
  - If we could recognize in some way that these initial characters will match (since they are the same characters over and over again) without having to go back then we can save time

- How is it done?
  - Pattern is preprocessed to look for "sub" patterns
  - As a result of the preprocessing that is done, we can create a 2-d array that is used give the “state” for each possible character in each position of the pattern

KMP

- We don't want to worry too much about the details here
  - See author's code if you are really interested

```
public int search(String txt)
{    int M = pat.length(); // pat is instance var
     int N = txt.length();
     int i, j;
     for (i = 0, j = 0; i < N && j < M; i++)
          j = dfa[txt.charAt(i)][j];
     if (j == M) return i - M; // found
     return N; // not found
}
```

- Note that the code above is clearly Theta(N)
  - dfa array access is constant time, and i increments by one on each iteration

KMP

- However, preprocessing must be done to generate the dfa array (not discussed here), requiring MR space (if alphabet has R characters) and time
  - Fairly complex preprocessing
  - There is an alternative version that uses less space (M rather than MR)
  - Idea is similar and it is still complex
- This algorithm is useful if the text is not buffered (i.e. we cannot back up in text)
- However, as stated previously, in the normal case it does not improve over brute force
  - Due to its overhead and complexity it is not preferred in normal circumstances

- Let's take a different approach:
  - We just discussed hashing as a way of efficiently accessing data
  - Can we also use it for string matching?
  - Consider the hash function we discussed for strings:

    `s[0]*B`$^{n-1}$ **`+ s[1]*B`**$^{n-2}$ **`+ … + s[n-2]*B`**^{1} **`+ s[n-1]`**

    – where B is some integer (31 in JDK)

    - Recall that we said that if B == number of characters in the character set, the result would be unique for all strings (if we can store it)
    - Thus, if the integer values match, so do the strings

- Ex: if B = 32
  - h("CAT") === 67*32^2 + 65*32^1 + 84 == 70772
  - To search for "CAT" we can thus "hash" all 3-char substrings of our text and test the values for equality
- Let's modify this somewhat to make it more useful / appropriate
  1) We need to keep the integer values of some **reasonable size**
     - Ex: No larger than an int or long value
  2) We need to be able to **incrementally update** a hash value so that we can progress down a text string looking for a match

- Both of these are taken care of in the Rabin Karp algorithm
  1) The hash values are calculated "mod" a large integer, to guarantee that we won't get overflow
  2) Due to properties of modulo arithmetic, characters can be "removed" from the beginning of a string almost as easily as they can be "added" to the end
     - Idea is with each mismatch we "remove" the leftmost character from the hash value and we add the next character from the text to the hash value
     - See example on next slide

Rabin Karp

[illegible]

A =

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	H	E	T	H	E	Y	T	H	Y	T	H	E	M	E	M

P =

0	1	2	3
T	H	E	M

H(THEM)

- Note that we are still moving left to right in the text as we mismatch
- However, we are not directly comparing characters in this case
- Rather we are comparing hash values
- Look at the code in RabinKarp.java

Rabin Karp

- Note:
  - We know hashing only guarantees no collisions if h(x) is unique for any key and key space is smaller than "table size"
    - This is not practical to implement
  - So what if a collision occurs?
    - The hash values would match but in fact the strings would not
  - Should we test for this?
    - As long as the "table size" is very large the probability of a collision is very low
      - > So the algorithm could make a mistake but it is not likely
    - Alternatively, we could verify the result with a char by char test

Rabin Karp

- If we don’t check for collisions:
  - Algorithm is guaranteed to run in Theta(N) time
  - Algorithm is highly likely to be correct
    - But it could fail if a collision occurs
  - Text calls this the Monte Carlo version
- If we do check for collisions:
  - Algorithm is highly likely to run in Theta(N) time
    - What would worst case be and why?
    - Discuss
  - Algorithm is guaranteed to be correct
  - Text calls this the Las Vegas version
- However, we still haven’t improved on the "normal case" runtime of Brute Force

- What if we took yet another approach?
  - Look at the pattern from right to left instead of left to right
    - Now, if we mismatch a character early, we have the potential to skip many characters with only one comparison
    - Consider the following example:

      A = ABCWABCXABCYABCZ

      P = ABCD
    - If we first compare D and W, we learn two things:
      1) W does not match D
      2) W does not appear anywhere in the pattern

      – We will see how we can do 2) soon

- Now with one mismatch we can move down the entire length of the pattern (M positions)

A =

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	W	A	B	C	X	A	B	C	Y	A	B	C	Z

P =

0	1	2	3
A	B	C	D

- If we know W is not in the pattern, no match could include W, so we can skip past it
- We do the same for X
- …and so on down the text

Boyer Moore

- Assuming our search progresses as shown, how many comparisons are required, given a text of length N and a pattern of length M?

  N/M

- This is a big improvement over the brute-force algorithm, since it is now a sub-linear time
- Will our search progress as shown?
  - Not always, but when searching text with a relatively large alphabet, we often encounter characters that do not appear in the pattern
  - This algorithm allows us to take advantage of this fact

- Details
  - The technique we just saw is the mismatched character (MC) heuristic
    - It is one of two heuristics that make up the Boyer Moore algorithm
    - The second heuristic is similar to that of KMP, but processing from right to left
  - Does MC always work so nicely?
    - No – it depends on the text and pattern
    - Since we are processing right to left, there are some characters in the text that we don't even look at
    - We need to make sure we don't "miss" a potential match

- Consider the following:

  A = XYXYXXYXYYXYXYZXYXYXXYXYYXYXYX

  P = XYXYZ

  - Now the mismatched character X DOES appear in the pattern
    - When "sliding" the pattern to the right, we must make sure not to go farther than where the mismatched character in A is first seen (from the right) in P
    - In the first comparison above, 'X' does not match 'Z', but it does match an 'X' two positions down (from the right) in P
      - > We must be sure not to slide the pattern any further than this amount
    - Note after the next comparison we can slide only 1 spot

- How can we figure out how far to skip?
- Preprocess the pattern to create a right array
  - Array indexed on ALL characters in alphabet
  - Each value will indicate how far we can skip if that character in the text mismatches with the pattern

```
for all i right[i] = -1;
for (int j = 0; j < M; j++)
        right[p.charAt(j)] = j;
```

  - Idea is that initially all chars in the alphabet are not in the pattern and set to -1
  - Index increases as characters are found further to the right in the pattern
    - The larger the value in the right array, the less we can skip

Boyer Moore

0	1	2	3	4

- Consider the pattern p = ABCDE

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

- So we will set
  - right['A'] = 0 { j = 0 }
  - right['B'] = 1 { j = 1 }
  - right['C'] = 2 { j = 2 }
  - right['D'] = 3 { j = 3 }
  - right['E'] = 4 { j = 4 }

- Note that -1 values will give maximum skip – these are not found in the pattern
- Characters that appear in the pattern will have right values >= 0
  - These will give smaller skips

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

```
for (int j = 0; j < M; j++)
    right[p.charAt(j)] = j;
```

Boyer Moore

0	1	2	3	4

- Consider the pattern p = XYXYZ

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

- So we will set
  - right['X'] = 0 { j = 0 }
  - right['Y'] = 1 { j = 1 }
  - right['X'] = 2 { j = 2 }
  - right['Y'] = 3 { j = 3 }
  - right['Z'] = 4 { j = 4 }

- Note that now we have -1 values for all except X, Y and Z

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2	3	4

```
for (int j = 0; j < M; j++)
    right[p.charAt(j)] = j;
```

- Now let's see how this will be used

```
public int search(String txt) {
    int M = pat.length();
    int N = txt.length();
    int skip;
    for (int i = 0; i <= N - M; i += skip) {
        skip = 0;
        for (int j = M-1; j >= 0; j--) {
            if (pat.charAt(j) != txt.charAt(i+j)) {
                skip = Math.max(1, j - right[txt.charAt(i+j)]);
                break;
            }
        }
        if (skip == 0) return i;    // found
    }
    return N;                        // not found
}
```

i updated by skip value

mismatch occurs

skip calculated

- Note that when j = M-1, if right[X] = -1, then the skip value is ((M-1) - (-1)) = M → this is the best case

- Can MC ever be poor?
  - Yes -- think about how this can occur

```
A = XXXXXXXXXXXXXXXXXXXXXXXXXXX
P = YXXXX
```

  - P must be completely compared (M char comps, right to left) before we mismatch and skip
    - So what do we skip?

    ```
    skip = Math.max(1, j - right[txt.charAt(i+j)]);
    ```

    - In this case
      - `j = 0` (we have moved all the way to the left)
      - `right['X'] = 4` (recall how right array is formed)
      - > This would give a skip of -4 (!!!)
      - > So the actual skip would be 1 (this is why we have 1 as a second option)

Boyer Moore

- We will then do the same thing again
- Thus we do M comps, move down 1, M comps, move down 1, etc
- We move down (N-M+1) times before the pattern will go past the end of the text
- This gives a total of (N-M+1)(M) comparisons or Theta(MN) when N >> M
  - Bad! Compare to Brute Force algorithm WC
- This is why the BM algorithm has two heuristics
  - The second heuristic guarantees that the run-time will never be worse than linear
  - This heuristic, as we mentioned, is similar to KMP but from right to left

- See: http://en.wikipedia.org/wiki/Boyer–Moore_string_search_algorithm

String Matching Summary

- Brute force (naïve algo):
  - Theta(N) normal case
  - Theta(MN) worst case (not likely)
- KMP
  - Theta(N) worst case
    - Preprocessing overhead
- Rabin-Karp
  - Theta(N) normal case
  - Theta(MN) worst case (Las Vegas version, very unlikely)
- Boyer-Moore
  - Mismatched char heuristic leads to poss. Theta(N/M)
    - Some preprocessing overhead
  - Second heuristic guarantees Theta(N) worst case

Approximate String Matching

- Sometymes wea wunt to fynde werds that are teckstually kloce to othr wirds
  - This allows autocorrect to "fix" our mistakes for us
    - Or to annoy us by putting in words we really don't want!
  - How does this work?
    - Uses an approximate string matching algorithm
    - For a given string we can find strings that differ by 1, 2 or more characters
    - We hope that one of these is the string we were intending to type

Approximate String Matching

- Ex: fynde
  - [fined, finder, find, fine, finned]
  - 3 2 2 2 4

    (2 if you count de → ed as 1) (3 if you count de → ed as 1)
- Some algorithms for these require dynamic programming
  - We will look at dynamic programming later in the term and may come back to this topic then