Yohannes M Himawan
David T Kraft

CS 337 String Matching Report

Note: all the times are in ms.

I. Comparing collisions rate for RK with different hash functions.

First experiment:
using base 256 number kind of hash with 17 as our prime number.
The text file size = 6MB
the pattern size = 4 characters
the pattern number = 10
8 is common words, 1 rare word, 1 non-word

```
                    <pattern>
Collisions for RK:  just: 145
Collisions for RK:  form: 33
Collisions for RK:  much: 390
Collisions for RK:  turn: 239
Collisions for RK:  does: 1182
Collisions for RK:  talk: 1569
Collisions for RK:  song: 11814
Collisions for RK:  body: 3517
Collisions for RK:  mimp: 362762
Collisions for RK:  dzrn: 396908
```

average collisions without the 2 special case = 987.63
average collisions overall = 76757.1

Second experiment:
using base 256 number kind of hash with 100,000,007 as our prime number.
The text file size = 6MB
the pattern size = 4
the pattern number = 10
8 is common words, 1 rare word, 1 non-word
average collisions = 0.
There are no collisions at all for this big number and small pattern size.

Third experiment:
using base 256 number kind of hash with 1,117 as our prime number.
The text file size = 6MB
the pattern size = 4
the pattern number = 10
8 is common words, 1 rare word, 1 non-word

```
Collisions for RK:  just: 2
Collisions for RK:  form: 0
Collisions for RK:  much: 1
Collisions for RK:  turn: 4
Collisions for RK:  does: 17
```

Collisions for RK: talk: 27
Collisions for RK: song: 95
Collisions for RK: body: 18
Collisions for RK: mimp: 4343
Collisions for RK: dzrn: 4498

average overall collisions rate = 900.5
average common words collisions = 20.5

Conclusions:

The better our hash function is and the bigger prime number we use and the more sophisticated our hash function is then we will have less collisions which is good because that will make our RK algorithm runs much faster then using a trivial hash function and a small primer number.

One more thing, we may need to improvise on how we do the mod operations because the way it was built in java the mod operation take a really long time to do.

This cause any hash function without any mod operation will run much faster then the one with mod even a simple one with just summing the ascii value will run faster then our sophisticated hash function when we have to do mod several times.

II. How RK algorithm affected by different parameters?

I will try to run RK algorithm with different sizes in pattern and different sizes in text file.

1st Experiment: small text small pattern
using base 256 number kind of hash with 7,919 as our prime number.
The text file size = 25kb
the pattern size = 4 characters

Time for RK: this: 56
Comparisons for RK: this: 166
Collisions for RK: this: 0
Time for RK: qwej: 101
Comparisons for RK: qwej: 26010
Collisions for RK: qwej: 0

2nd Experiment: medium text medium pattern
using base 256 number kind of hash with 7,919 as our prime number.
The text file size = 874kb
the pattern size = 140 characters

Time for RK: I did not sleep well, though my bed was comfortable enough, for I had all sorts of queer dreams. There was a dog howling all night under my: 1164
Comparisons for RK:4443
Collisions for RK: 3

3rd Experiment large text large pattern

same hash function as above
The text file size = 6mb
the pattern size = 1050 characters
Time for RK: 1209
Comparisons for RK: : 28993
Collisions for RK: : 5

4th experiment : small text big pattern
same hash function as above
The text file size = 874kb
the pattern size = 765 characters

Time for RK: 91
Comparisons for RK: : 11827
Collisions for RK: : 5

5th experiment: big text small pattern
same hash function as above
The text file size = 6mb
the pattern size = 38 characters

Time for RK: ADVENTURE II. THE RED-HEADED LEAGUE: 1207
Comparisons for RK: : 48375
Collisions for RK: : 8

Conclusions:

RK performs well on small text file with big patterns as long as we have a good hash functions. On Big text file with small patterns, if the pattern is common and we have a good hash function it will performs well overall. On the other case if the pattern is short and not common and we don't use a really good hash function while we are having a big text file then RK will performs badly.

Comparisons of RK and KMP with Java built-in string search function(String.contains())

RK with java built in :
RK runs slower in general with the java built-in string search routine.

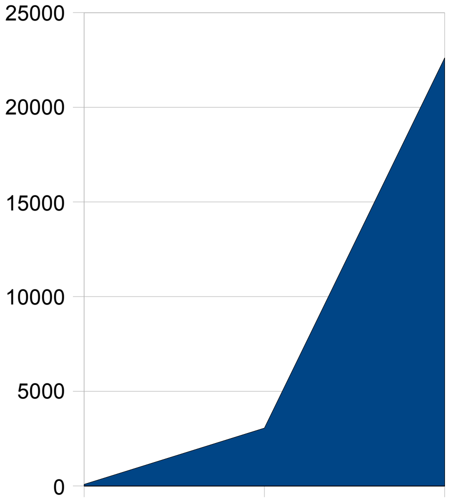
KMP with java built in:
KMP runs almost in par with the java built-in string search routine. This result makes us think that the java contains method from string might have used KMP or Boyer-Moore algorithm.

Comparisons of RK and KMP with naïve algorithm.

RK and KMP in general runs faster than the naïve algorithm. KMP runs faster than RK in most cases. We could get into worst case when we have a lot of mod operations for our hash function that will make our naïve a little bit faster than our RK. Also, when we have a lot of collisions RK will performs as bad as our naïve algorithm.

GRAPHS:

RK graphs with small pattern and increased file size:
using base 256 number kind of hash with 7,919 as our prime number.
The text file size = 25kb, 874kb, 6488kb
the pattern size = 10-50 characters



RUNTIME

Comparisons
3586
722231
6488667

80
3057
22614

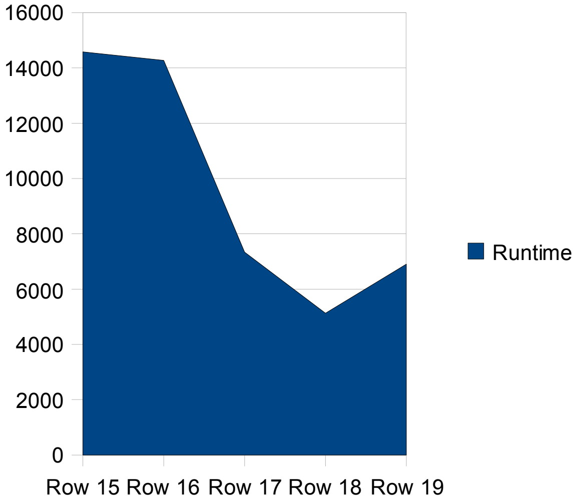
as shown here the runtime and comparisons increases as the text file increases.

Now, we will do this again but with longer patterns.

Runtime
71
2663
16671

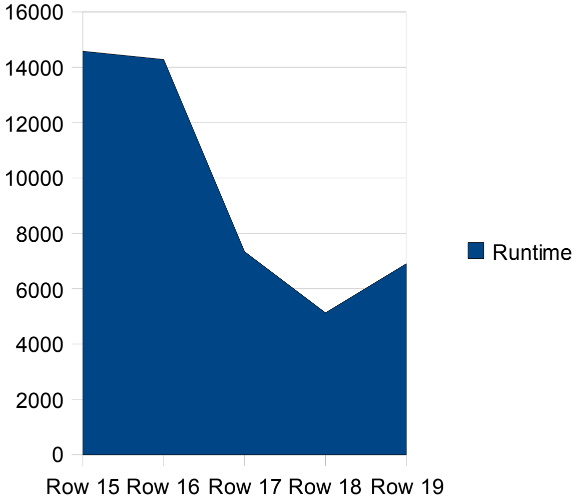
It shows that with longer patterns we decrease the average runtime slightly and we have a decrease in collisions and comparisons rate too.

KMP graphs with small pattern and increased file size



Runtime
66
1747
7338

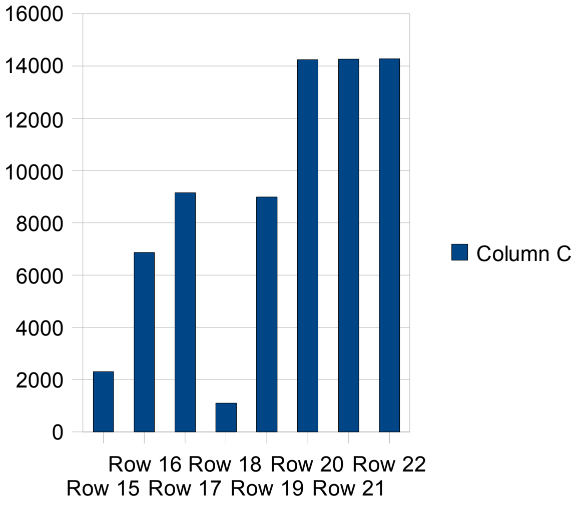
The runtime of the KMP also increases as the file size increases however it's interesting to note that even on the worse case scenario KMP won't be performing as bad as the RK.
This is another table when we run various patterns on the big text file



Runtime
14585
14278
7338
5134
6900

It runs always around 14 seconds if the pattern that we aren't searching is there and the average time is around 6 seconds whenever the pattern that we are searching is there. While in RK our worst case will be almost as bad as the naïve algorithm if we don't have the specific pattern that we are looking for in the text.

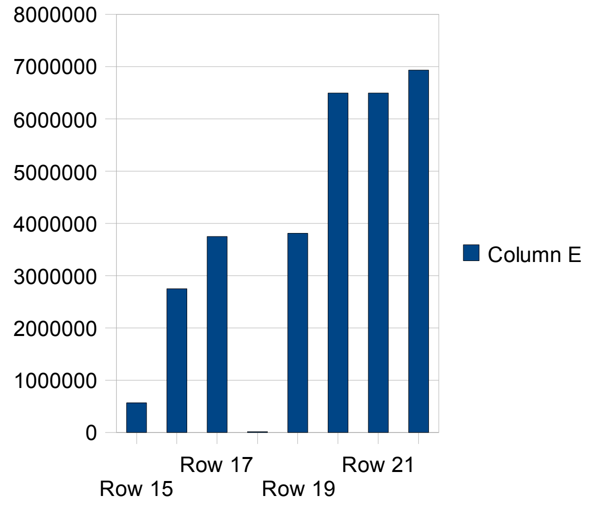
KMP with a same big text file but with various length patterns that is in the file and also not in the file.



Runtime
2305
6864
9150
1110
8993
14241
14262
14274

The last 3 cases involves string with small, medium, and large length and it is not there but they are all have the same worst case runtime which is about 14 seconds. Case no. 4 is a big string that appears early in the file that's why the runtime is really short, if we have a common string pattern that has a common char then it will might run longer because KMP isn't skipping a lot of stuff but if we have a weird string pattern that does appear in the text then KMP will runs really fast.

This is the graph of comparisons for this same case:



Comparisons
567879
2748734
3749391
18433
3811444
6493088
6493088
6933123

That 4th case is a big pattern with so it will get a lot of skipping from the KMP algorithm therefore we don't really do a lot of number of comparisons.
The first string that we do is “Arkansas” and a lot of time common words doesn't contain any substring from that word. That's also why we don't do a lot of comparisons.

Conclusions:

The runtime for both KMP and comparisons number increases as we increase the number of the patterns we are comparing. For RK whenever we increase the size of the files and we have the same patterns we will have a longer runtime and comparisons in general. However if our pattern is also long then we will have a shorter runtime than having a short pattern in an increasing text file size.

KMP is a little bit unique, it will have a almost the same worst case scenario no matter whether the string is big or small according to our experiments. The more unique our sequence of characters is for KMP is the better because KMP will skip a lot of strings according to the fail table because of that.

Answers to questions:

1. What will happen to RK algorithm if we use linear sum hashes instead of rolling hash?
   -RK will performs as bad as the naïve algorithm because we are recomputing the hash value every time we have move on the a new character.
2. Which algorithms has best performance for specific inputs?
   KMP has the best performance overall for any input file.
3. When does RK performs as bad as the naive algorithms?
   RK will perform as bad as the naïve algorithm when we have a really bad hash function that will cause a collision most of the time.

4. When does RK performs as good as KMP?
   When there's almost no collision in the hash function the RK will run almost as fast as the KMP.