CS337 Project 4
String Matching Algorithms

TA in charge: Tanvi Motwani (tanvi@cs.utexas.edu)

Assigned: 20.April.Wednesday

1. Overview

The purpose of this project is to learn about string matching by comparing multiple string searching algorithms to see under what circumstances one algorithm may outperform another.

Your job is to implement the *Rabin-Karp* algorithm, *Knuth-Morris-Pratt* algorithm, *Boyer-Moore* algorithm and Brute force string matching algorithm , as described in class, to search an input text file for each requested string from a different input patterns file. The program should return an indication if the patterns are found, as well as timing information about the search algorithm. You will be provided with a large text document and a set of pattern strings.

The writeup of the project should explain your results and give a summary of the performance for the different algorithms you have implemented.

Do NOT use any string matching or search functions that are built into Java when writing your algorithm implementations. Also, you may use whatever you wish when reading in the search strings file, but do NOT use search or find routines when reading in the source file.

2. Performance

This project is about comparing different algorithms. As such, the program you write must terminate in a reasonable amount of time for all strings provided. If your program does not perform adequately, then there is something wrong with your program and it must be examined. Common inefficiencies may include: failure to properly buffer blocks of characters from the disk drive; using an inefficient data structure; repeating steps unnecessarily; or any number of simple mistakes.

3. User Interface

Your program needs to take in three command-line parameters: the first argument is the name of the pattern file, `pattern.txt` containing the strings to find, second, `source.txt` is the name of the text file to search, third, `results.txt` is the name of the file to output the match results. The main Java class needs to be named `strMatch.java`. An example command-line might look like:

```
java strMatch pattern.txt source.txt results.txt
```

```
&Into the fray good men&
&To be, or not to be&
&There's a blank space at the end of this search string &
&This string has a line-feed
that must be matched &
```

Figure 1: Sample pattern string file

4. Input

The pattern strings file will be a text file with strings separated by blank lines and/or newlines and/or spaces. Each search string will be surrounded by an ampersand (`&') symbol. See Figure 1 for a sample file. For simplicity you may assume that ampersands (`&') will not appear in the pattern strings themselves.

Any input pattern file may be used with your project. Please use the provided pattern file (*samplePattern.txt*) and input text file (*sampleSource.txt*) for the algorithm comparison in your report.

The input text files used for testing your code will be created on Unix systems, so it should run properly on a Unix machine.

5. Output

See Figure 2 for a sample pattern file and the EXACT format expected in the output file. You may use Standard-Out or create another file to store the statistics needed to study your algorithm implementations. Each search algorithm needs to find the first match and can then terminate the search for that pattern. When running time trials for your report it is important to remember that there are different types of time: CPU time, real time, user time, and so on. It is quite possible that the real time taken to run your algorithm may be much longer than the CPU time taken because of process scheduling on a shared computer, disk I/O and screen output. But then, I know you'll explain all the different things you encounter in your report. The time details of the algorithms reported by you will be verified , **please do NOT give wrong results by making up your own numbers as you will be penalized heavily, please report the exact time details you encounter.**

Input File:

```
&To be, or not to be&
&That there's a blue cucumber!&
&Hippocampelephantocamelos&
```

results.txt:

```
RK FAILED: To be, or not to be
KMP FAILED: To be, or not to be
BM MATCHED: To be, or not to be
Brute Force FAILED: To be, or not to be
RK MATCHED: That there's a blue cucumber!
KMP MATCHED: That there's a blue cucumber!
BM MATCHED: That there's a blue cucumber!
Brute Force MATCHED: That there's a blue cucumber!
RK MATCHED: Hippocampelephantocamelos
KMP MATCHED: Hippocampelephantocamelos
BM MATCHED: Hippocampelephantocamelos
Brute Force MATCHED: Hippocampelephantocamelos
```

Figure 2: Sample pattern file and the corresponding output file `results.txt`. *These are real quotes from real plays. Can you guess the source file?*

6. Report in `Report.pdf`

- You need to give a report comparing the efficiency (comparing runtime) of *Rabin-Karp, Knuth-Morris-Pratt, Boyer-Moore and Brute Force* algorithms when using the pattern strings provided with the sample text file, as well as your own test. You need to compare these three algorithms with the naïve Brute Force Algorithm that you have implemented and show how the runtime is lesser than the Brute force method. If your RP, KMP and BM algorithms are poorly implemented then they might give more runtime than the Brute Force algorithm, this condition should be avoided and will be penalized if occurs. You should code these algorithms **very efficiently** so that it reflects in the results.

- You must compare different versions of *Rabin-Karp* algorithm and examine how the collision rate and runtime varies with different versions. At a minimum, you must implement a rolling linear sum and rolling base sum and keep track of collisions and runtime of both versions. Also state which versions work better under which condition.

- Also state the number of comparisons required to match the string by each algorithm. A comparison is any check to see if character/number is identical. Suppose we want to check if "abcd" and "abdf" are equal, we need 4 comparisons to conclude that they are not. So for all methods, you should count the comparisons to determine how many were made during each run. For this you can take a small test file and note down the number of comparisons made by each algorithm on the same test file.

- When creating your own test cases, be sure to look at various sizes of strings and perhaps even some degenerate cases such as files full of a single letter. You may want to compare your implementations with Java's built-in search routines to see how they compare. Your program must be able to process very small to very large source files up to 1GB in size. So you should test your algorithms on such large files. A lot of time will be needed to write this report so please start early.

- So basically you have to compare the performance for *Rabin-Karp*, *Knuth-Morris-Pratt*, *Boyer-Moore* and Brute Force algorithms. Further you have to compare different versions (different hashing methods) of *Rabin-Karp* algorithm by comparing their collision rate and runtime. **You need to create Tables and compare for each of the points mentioned above.**

- A lot of importance would be given to your report in this project. Please make a well written and detailed report. Include various experiments by varying the size of input file, varying pattern files, comparing runtime for different algorithms. Include Tables, Figures and Graphs in order to have a clear comparison. 10 extra points would be given to a well written and detailed report.

7. Turn in your project

Please follow the project protocol for files that should be turned in.
You must submit the followings:

strMatch.java
pattern.txt
source.txt
results.txt
emails.txt
readme.txt
report.pdf

Use the Linux turn-in software to submit your project. The command will look like:

turnin --submit tanvi project4 ***strMatch.java pattern.txt source.txt results.txt emails.txt readme.txt report.pdf***

After you have turned in your project, it is now possible to use turning --verify tag to check that your files have been turned in properly.

ONLY ONE submission per group!
1. Please do not turn in a tarred or compressed version of your files.
2. Please strictly follow the files names. Otherwise, your submission may not be graded.
3. No late submission is allowed. The turn-in program will be turned off after the deadline time. And no email submission is accepted.
4. You must make sure your program will work correctly on CS Linux machines.
All the text files should be generated using LINUX OS.

* readme.txt (Do not forget Section#!)

NAME of partners;
Section # of partners;
EID of partners;
CODING STATUS: (very important!);

IMPORTANT: Summarize your code status in "readme.txt" file. If your code works perfectly, explicitly say this in readme.txt. If you code does not work (can not compile, or can not run, or only work for some FSMs or can not output anything, etc.), you need to specifically explain this: what have been done so far and what have not been done, what is working and what is not working to avoid confusion! Put proper comments in all the code files.

* emails.txt

"emails.txt" content and format: (You MUST follow the exact format below! Otherwise, no comment and grade will be sent to your and your partner's emails):

Submitter_name submitter_CSaccount submitter_email
Partner_name partner_CSaccount partner_email

Example "emails.txt" (Note: There is a space between your CS account and your email):

Nam Nguyen namphuon namphuon@cs.utexas.edu
Craig Corcoran ccor ccor@cs.utexas.edu

8. Questions and Project Clarifications

Clarifications regarding this project will be posted to the blackboard and your email. Use discussion board very actively with your friends. Find your partner ASAP. It is your responsibility. Enjoy the project work !