DM548 – Computer Architecture

Assignment 1, Fall 2015

Mads Petersen

Algorithm

```
file1 = Open(queries file)
file2 = Open(text file)
lastNewline = 0
While(More to read in file1)
  inputBuffer = Read(file1)
  position I = Search('\n',inputBuffer)
  searchWord = Substring(lastNewline, position I,inputBuffer)
  lastNewline = position I
  file1.currentPosition = position I
  searchWordLength = Length(searchWord)
  i = 0
  searchWordOccurences = 0
  While(More to read in file2)
    inputBuffer = Read(file2)
    While(End of input buffer not reached)
      targetText = Substring(i,searchWordLength+i,inputBuffer)
      If targetText == searchWord
        searchWordOccurences = searchWordOccurences + 1
        Print(i)
      i = i + 1
  If I == 0
    Print(-1)
```

The algorithm implemented iterates first through the queries file, fetching the search words 1 at a time by going through the file 1 character at a time looking for a newline character. When a newline character is found, everything from the position of the last newline character + 1, to the position of the current newline character - 1, is copied to a variable. The target file to be searched is then iterated through by looking at the first k characters, where k is the length of the search word. If these k characters are a match to the k characters of the search word, 0 is printed. Then the text is shifted 1 position. I.e. the first run looks at characters $c_0, c_1, ..., c_k$, the next run looks at $c_1, c_2, c_3, ..., c_{k+1}$. If this next sequence is a match to the search word, then 1 is printed to standard out. In this way the entire file is run through in the the last k characters. When there is $k-1$ or less characters left it skips to the next search word, since there is no longer any possible matches. This take $O(m)$ repetitions of *scasb*, and then 1 file read to get the m characters, then 1 file update to move the file to current position just after the current search word. Then there is $n-m$ file reads from the text file to fetch and search all possible sequences. These each use $m \times cmpsb$ operations.This gives a run time of $O(m + (n-m) \times m) \times q$ where q is the number of queries. Each *cmpsb* operation might vary some in execution time since if the first letter doesn't match then only 1 letter needs to be compared before skipping to the next word.

Testing

Testing is done on a virtual Linux machine running on 1 3,4 GHz i7-2600K CPU core and 1 GB ram. The text files of length n = 10.000, 100.000 and 1.000.000 characters in length are multiples of The Oxford Circus by Alfred Budd obtained from `http://www.gutenberg.org/ebooks/50358`. The tests

are timed with the *time* command and real time is what is reported below, the queries consists of random numbers and letters of length m.

10 queries	$m = 10$	m = 100	m = 1000
n = 10.000	0,05s	0,07s	0,025s
n = 100.000	0,083s	0,114s	0,191s
n = 1.000.000	0,666s	0,841s	1,115s

100 queries	$m = 10$	m = 100	m = 1000
n = 10.000	0,097s	0,138s	0,208s
n = 100.000	0,878s	1,242s	1,831s
n = 1.000.000	6,279s	8,171s	11,852s

1000 queries	$m = 10$	m = 100	m = 1000
n = 10.000	0,837s	1,306s	2,405s
n = 100.000	8,018s	11,954s	18,473s
n = 1.000.000	1m5,868s	1m25,605s	1m56,136s

$n =$ length of search text in characters including whitespace.
$m =$ length of query.

As can be seen from the above, the length of the query has little impact on the execution time. The greatest impact comes from the length of the text to search and from the number of queries. It can also be seen that going up by a factor of 10 for either text length or number of queries also takes the execution time up by around a factor of 10. Looking at the results the algorithm seems to fit in $\Theta(qn)$ running time, where q is number of queries and n is length of text to be searched. Keeping either of these parameters down, also keeps the running time manageable, the length of the queries is almost irrelevant.