CMPT 360: Lab Assignment #7
Efficient Substring Finder

Brady Coles

last updated: January 3, 2018

Contents

1 Course Goals

This assignment fulfills the following goals:

- a group IV language (Objective-C)
- implemented on the Mac OS X platform

2 Problem Description

The given problem was to implement an *efficient* algorithm for finding a substring within another string.

The implemented algorithm is the Knuth-Morris-Pratt algorithm for substring finding. The algorithm was published in 1977 jointly by Donald Knuth, Vaughan Pratt, and James H. Morris (*Knuth-Morris-Pratt algorithm* n.d.). The efficiency of this algorithm will be explained in the Program Documentation section.

3 Sample I/O

The goal of a substring finding algorithm is to either return the index of the first instance the substring was found in a larger string, or to return a list of every index where the substring was found in the larger string. The algorithm is the same in either case, and it is trivial to modify one version to another. Instead of returning a valid match immediately, the match would be added to a list and the algorithm left to continue, with the list being returned once the whole string was searched.

The implemented algorithm implements the first version, finding the first instance. In cases where the substring is contained in the larger string, this method is more efficient, as the algorithm can exit before searching the entire string.

The following examples show what this algorithm should output when looking for the `pattern` in `string`. The samples are using zero indexing, so a match that starts on the first character returns `0`, and so on.

```
pattern: car
string: racecar
index: 4

pattern: abcdabd
string: abc ababd adabcdabda
index: 12

pattern: Hello World
string: hi
index: No match

pattern: zippy
string: The quick brown fox.
index: No match

pattern: 123
string: 12345
index: 0
```

4 Program Documentation

4.1 Problem Solution

Let us take m and n to be the length of the pattern and string, respectively, where the pattern is the substring we wish to find in string. The naive solution is to start by checking if the pattern occurs at the

first character in string, and going character by character until either the entire pattern is matched, or a mismatch is found. If a mismatch is found, go back to the second character in string, and try matching the pattern starting there. This continues until a pattern is matched or until the remaining characters in string are shorter than the pattern.

In the case where no match is found, the fastest runtime is when a mismatch occurred immediately on each starting position, and is the slowest when a mismatch occurred on the last character of the pattern. Asymptotically, the running time is $\Theta(mn)$.

This obviously requires improvement. Many algorithms have improved on this by preprocessing either the string or pattern so that more information is available when doing the search. The algorithm by Knuth-Morris-Pratt (KMP) takes advantage of the fact that if a partial match has been made, we know something about the characters in that match, and should be able to jump directly to the next possible location where the substring could be found, rather than naively moving only one index over.

For example, if you are searching for `abcabd` in `ababcabcabd`, then if we have found the partial match from index `2`: `abcab`, then we know the next possible match must start at index `5`, since the fourth and fifth characters in the pattern are the same as the first two characters in pattern. The KMP algorithm preprocesses the pattern to find all partial matches of the pattern within itself. It creates a partial match table that indicates how many characters before the current character are a prefix of the pattern itself, so that when the search is occurring, the search can skip directly to the next possible substring location when a mismatch is found.

The partial match table is thus made up of the longest prefix that ends the character before the current one. For example:

a	b	c	a	b	d
-1	0	0	0	1	2

Note that if you are matching against the pattern, and you have a mismatch on the final character, the partial match table says that the previous `2` characters are a prefix for the pattern, so you can move the new location where the algorithm is checking to two characters behind the one currently being checked.

The first and second entries in the table must be `-1` and `0` in order for the algorithm to ignore the fact that the pattern is of course a prefix for itself.

Once the table has been made, then the search algorithm can intelligently jump to the next possible substring location when a mismatch occurs, rather than naively stepping through every starting position one at a time.

4.2 Pseudocode

The algorithm is broken into two parts, the algorithm that creates the partial match table, and the algorithm that searches the string.

Here is the partial match table algorithm. Both this pseudocode and the listed program are based on the psuedocode found on the wikipedia article for this algorithm (*Knuth-Morris-Pratt algorithm* n.d.). Both algorithms are assuming zero indexing for arrays.

```
function partialTable(pattern)
begin
    table = array(pattern.length)
    table[0] := -1
    table[1] := 0
    i := 2
    m := 0
    while (i < pattern.length)
    begin
        if (pattern[m] = pattern[i-1])    # Prefix continues
        then
            table[i] := m
            i++
            m++
```

```
        else if (m > 0)    # Prefix does not continue, set to next longest possible
        then
            m = table[m]
        else    # No prefix, set to 0 and continue
            table[i] := 0
            i++
        end
    end
    return table
end
```

Once the table has been created, the search algorithm can take place.

```
function findSubstring(pattern, string)
begin
    table := partialTable(pattern)
    i := 0
    m := 0
    while (i < string.length and m <= (string.length - pattern.length))
    begin
        if (string[i] = pattern[i - m])    # The partial match continues
        then
            if ((i - m + 1) = pattern.length)    # All of pattern has been matched
            then
                return m
            end
            i++
        else if (i = m)    # No match, was checking first character, move on
        then
            i++
            m++
        else    # No match, may be partial match, jump to next partial match
            m = i - table[i - m]    # This is where partial match table is used
        end
    end
    return nil    # No match found
end
```

4.3 Running Time

Unlike the naive algorithm, the KMP algorithm never backtracks while searching. Although a character may be checked multiple times if multiple partial matches exist before it, once a character has been passed over, the algorithm never backtracks. This significantly improves efficiency, and in the average case, the number of times the algorithm will check a character is not very many compared to the length of the pattern or string. In the worst case, the algorithm could still check every character many times. Using m for the length of the pattern and n for the length of the string, the running time of the search itself is $\Theta(n)$, but there is still the preprocessing (*String searching algorithm* n.d.).

The preprocessing step also never backtracks, and so on average will spend very few iterations on each character in the pattern, yielding a run time of $\Theta(m)$. The combined run time $\Theta(m + n)$ is much more efficient than the naive approach which is $\Theta(mn)$ (*String searching algorithm* n.d.), especially when the pattern is non-repetitive, as that minimizes the iterations per character in both the partial match and search algorithm.

4.4 Documentation

The included program includes the `KMPSubstring` class which contains the substring finding algorithm, as well as a main program which allows a user to find substrings by inputting strings and patterns to the console. The console program is simply a way to test the algorithm, and uses unsafe methods for getting console input (buffer overflow is not protected).

4.4.1 KMPSubstring Class

This class provides allows a substring to be found within a string using the Knuth-Morris-Pratt method.

`class KMPSubstring`	
Public Methods	
Name	Description
`firstMatchOf(NSString, inString:NSString)`	Returns the first index where the pattern (first string) was found in the (second) string. Returns `-1` if no match found. Match is case sensitive.
`contains(NSString, inString:NSString)`	Returns true if the pattern (first string) was found in the (second) string. Returns false if no match found. Match is case sensitive.

5 Program Listing

```
1  /*
2   * Author: Brady Coles
3   * Substring finding program using Knuth-Morris-Pratt algorithm.
4   */
5
6  #import <Foundation/Foundation.h>
7
8  /* A class for finding substrings within a string. Uses the Knuth-Morris-Pratt
9   algorithm, which preprocesses the substring to enable faster searching of the
10  string. */
11 @interface KMPSubstring : NSObject
12 /* Returns the first index (starting from zero) in string where a full
13  match of patter was found. If no match found, returns -1 */
14 - (int) firstMatchOf:(NSString *) pattern inString:(NSString *) string;
15 /* Simple wrapper of firstMatchOf that checks if a match was found */
16 - (BOOL) contains:(NSString *) pattern inString:(NSString *) string;
17
18 @end
19
20 /* Extends KMPSubstring, but without exposing the additional members */
21 @interface KMPSubstring()
22 /* Creates the partial match table of the 'pattern'required to find a substring
23  with firstMatchOf. This is what allows the KMP algorithm to be effcient. */
24 - (int *) partialTable:(NSString *) pattern ;
25 @end
26
```

```
/* Implements the methods in KMPSubstring and the extension. */
@implementation KMPSubstring

/* Finds the first match of 'pattern' in 'string'. Gets a partial match table from
 partialTable. Running time, excluding the call to partialTable is O(n) where n is
 the length of the string being searched. */
- (int) firstMatchOf:(NSString *)pattern inString:(NSString *)string {
    /* If the pattern is longer than the string or the pattern is empty, there
     can be no matches. */
    if (pattern.length > string.length || pattern.length == 0)
        return -1;

    /* The location (in string) of the start of the currently tested match */
    int m = 0;
    /* The location (in string) of the current character being tested against */
    int i = 0;
    /* Partial match table for pattern */
    int *t = [self partialTable:pattern];

    /* Begin algorithm */
    while (i < string.length && m <= string.length - pattern.length) {
        if ([string characterAtIndex:i] == [pattern characterAtIndex:(i - m)]) {
            /* Current character matches character in pattern, go to next character
             or return position of match if whole pattern checked */
            if ((i - m + 1) == pattern.length) {
                return m;
            }
            i++;
        }
        else if (m == i) {
            /* Current character not a match, and first character in pattern being
             tested, simply try starting at next character */
            m++;
            i++;
        }
        else {
            /* Had found partial match, use partial match table to find next possible
             location for a match to start. Testing can stay at current index, since
             all prior characters in pattern must be true according to the partial
             match table. */
            m = i - t[i - m];
        }
    }
    /* Match never found, whole string searched. Return -1 to indicate no match. */
    return -1;
}

/* Simply checks if firstMatchOf returns a valid index, returning true, otherwise
 no match was found, and so returns false. Note that no information is stored
 between runs, so call firstMatchOf if you want the index of a match, and check
 that the return value is a valid index (ie, -1 is invalid and indicates no match).
 Calling contains before firstMatchOf runs algorithm twice. */
- (BOOL) contains:(NSString *)pattern inString:(NSString *)string {
    return ([self firstMatchOf:pattern inString: string] >= 0);
```

```
}

/* Creates a partial match table of the pattern. Indicates how many previous
 characters are a prefix of the pattern, allowing the search algorithm to know
 exactly where the next possible match could begin, based on how much of pattern
 had already been matched. */
- (int *) partialTable:(NSString *) pattern {
    /* An empty pattern can't be matched. */
    if (pattern.length == 0) {
        return NULL;
    }
    /* A single character is a special case, since only one of two
     specially initialized entries can exists */
    int *table = malloc(pattern.length * sizeof(int));
    if (pattern.length == 1) {
        table[0] = -1;
    }
    /* in order to maintain the correctness of the algorithm when checking
     partial matches at the beginning of 'pattern', the first and second entries
     are always -1 and 0. */
    else {
        table[0] = -1;
        table[1] = 0;
    }
    /* The current index in 'pattern' being calculated for the table */
    int i = 2;
    /* The index of the next character to check to continue a partial match */
    int m = 0;

    while (i < pattern.length) {
        if ([pattern characterAtIndex:(i-1)] == [pattern characterAtIndex:m]) {
            /* previous character continues a partial match */
            /* Show number of characters in partial match ending at previous character */
            table[i] = m + 1;
            /* Set next character for partial match to next character in 'pattern' */
            m++;
            /* Move to next entry in table and next character in 'pattern'*/
            i++;
        }
        else if (m > 0) {
            /* Match not found, but partial match exists, so change to next
             possible partial match containing that character and try again.
             Next attempt may have m = 0 */
            m = table[m];
        }
        else {
            /* Match not found, and m = 0, so no more possible partial matches
             including that character, move on. */
            table[i] = 0;
            i++;
        }
    }
    return table;
}
```

```
135
136 @end
137
138
139 // Main program. Used to test KMPSubstring.
140 int main(int argc, const char * argv[]) {
141     @autoreleasepool {
142         NSLog(@"Substring finder using Knuth-Morris-Pratt Algorithm");
143         char str[100]; // Temporary container for input.
144         KMPSubstring *s = [[KMPSubstring alloc] init];
145
146         /* Repeat until stopped by user */
147         while (true) {
148             /* Get input */
149             NSLog(@"Enter a string to find a substring in:");
150             gets(str); // Unsafe
151             NSString *string = [NSString stringWithUTF8String:str];
152             NSLog(@"Enter a pattern to search for");
153             gets(str); // Unsafe
154             NSString *pattern = [NSString stringWithUTF8String:str];
155             /* Result */
156             int index = [s firstMatchOf:pattern inString:string];
157             NSLog(index >= 0 ?
158                   [NSString stringWithFormat:@"pattern '%@' found in '%@' at index %d",
159                     pattern, string, index]
160                   :[NSString stringWithFormat:@"pattern '%@' not found in '%@'",
161                     pattern, string]);
162             /* Check is user is done */
163             NSLog(@"Would you like to search another string? (y/n)");
164             gets(str); // Unsafe
165             NSString *answer = [NSString stringWithUTF8String:str];
166             if ([s firstMatchOf:@"y" inString: answer] != 0) {
167                 NSLog(@"Goodbye");
168                 break;
169             }
170         }
171     }
172     return 0;
173 }
```

6 I/O Listing

Here is an actual console interaction with the program. Several cases were tested.

```
2017-04-11 21:13:09.823 substring[85604:6175428] Substring finder using Knuth-Morris-Pratt Algorithm
2017-04-11 21:13:09.825 substring[85604:6175428] Enter a string to find a substring in:
warning: this program uses gets(), which is unsafe.
Hello World!
2017-04-11 21:13:13.867 substring[85604:6175428] Enter a pattern to search for
lo
2017-04-11 21:13:14.986 substring[85604:6175428] pattern 'lo' found in 'Hello World!' at index 3
2017-04-11 21:13:14.987 substring[85604:6175428] Would you like to search another string? (y/n)
y
2017-04-11 21:13:16.611 substring[85604:6175428] Enter a string to find a substring in:
abc ababd ababcdabdac
```

```
2017-04-11 21:13:43.064 substring[85604:6175428] Enter a pattern to search for
abcdabd
2017-04-11 21:13:47.934 substring[85604:6175428] pattern 'abcdabd' found in 'abc ababd ababcdabdac' at index 12
2017-04-11 21:13:47.934 substring[85604:6175428] Would you like to search another string? (y/n)
y
2017-04-11 21:13:51.651 substring[85604:6175428] Enter a string to find a substring in:
Can it find an empty string?
2017-04-11 21:14:05.163 substring[85604:6175428] Enter a pattern to search for

2017-04-11 21:14:06.130 substring[85604:6175428] pattern '' not found in 'Can it find an empty string?'
2017-04-11 21:14:06.131 substring[85604:6175428] Would you like to search another string? (y/n)
y
2017-04-11 21:14:10.338 substring[85604:6175428] Enter a string to find a substring in:

2017-04-11 21:14:12.338 substring[85604:6175428] Enter a pattern to search for
What if it searches and empty string?
2017-04-11 21:14:22.378 substring[85604:6175428] pattern 'What if it searches and empty string?' not found in ''
2017-04-11 21:14:22.378 substring[85604:6175428] Would you like to search another string? (y/n)
y
2017-04-11 21:14:29.345 substring[85604:6175428] Enter a string to find a substring in:
aaaaaaaabbba
2017-04-11 21:14:41.012 substring[85604:6175428] Enter a pattern to search for
aab
2017-04-11 21:14:43.205 substring[85604:6175428] pattern 'aab' found in 'aaaaaaaabbba' at index 6
2017-04-11 21:14:43.205 substring[85604:6175428] Would you like to search another string? (y/n)
y
2017-04-11 21:14:52.221 substring[85604:6175428] Enter a string to find a substring in:
Banana
2017-04-11 21:14:58.082 substring[85604:6175428] Enter a pattern to search for
ana
2017-04-11 21:15:00.323 substring[85604:6175428] pattern 'ana' found in 'Banana' at index 1
2017-04-11 21:15:00.323 substring[85604:6175428] Would you like to search another string? (y/n)
n
2017-04-11 21:15:02.115 substring[85604:6175428] Goodbye
Program ended with exit code: 0
```

References

[1] *Knuth-Morris-Pratt algorithm*. URL: https://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm.

[2] *String searching algorithm*. URL: https://en.wikipedia.org/wiki/String_searching_algorithm.