Lab 7: Pattern Matching

Michael Morikawa

April 28, 2020

Lab Questions

Question 1 Are there any situations where BM better than KMP? Explain.
Yes, if the pattern is short and doesn't have any prefix/suffix in common within its substrings.

Question 2 Is Brute Force Pattern Matching with BM's looking-glass heuristic always better than original Brute Force Pattern Matching? Explain.
It will not always be better than the original brute force algorithm. This is because it is possible for a mismatch to only occur at the beginning of a pattern and thus the looking glass heuristic will take more comparisons.

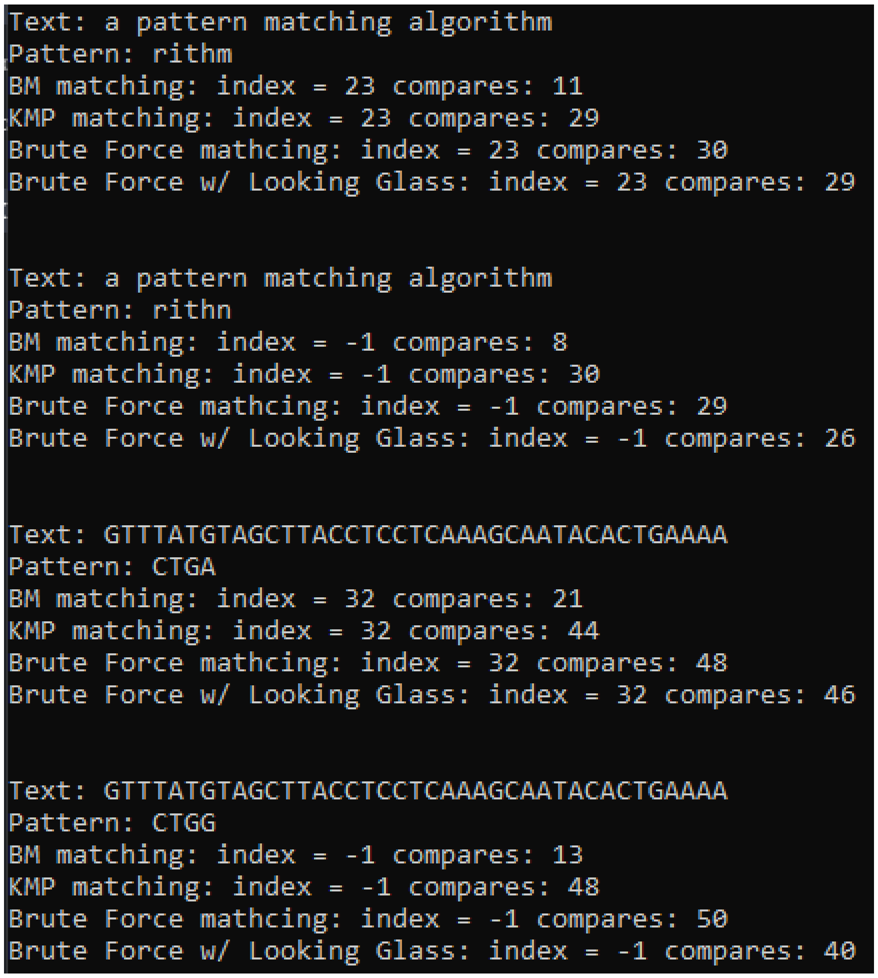
Source Code

main.cpp

```cpp
#include <iostream>
#include <string>
#include <vector>

//Code for BM and KMP pattern matching provided by the book modified to count comparisons

int bruteForce(const std::string &text, const std::string &pattern, int &compares);
int BMmatch(const std::string &text, const std::string &pattern, int &compares);
std::vector<int> buildLastFunction(const std::string &pattern);
int KMPmatch(const std::string &text, const std::string &pattern, int &compares);
std::vector<int> computeFailFunction(const std::string &pattern);
void output(const std::string &text, const std::string &pattern);
//Extra Credit
int bruteForceLookingGlass(const std::string &text, const std::string &pattern, int &compares);

int main()
{
    std::string text_1 = "a pattern matching algorithm";
    std::string pattern_1a = "rithm";
    std::string pattern_1b = "rithn";
    std::string text_2 = "GTTTATGTAGCTTACCTCCTCAAAGCAATACACTGAAAA";
    std::string pattern_2a = "CTGA";
    std::string pattern_2b = "CTGG";
    output(text_1, pattern_1a);
    output(text_1, pattern_1b);
    output(text_2, pattern_2a);
    output(text_2, pattern_2b);
}

/*
Simplified version of the Boyer-Moore algorithm. Returns the index of
the leftmost substring of the text matching the pattern, or -1 if none.
*/
int BMmatch(const std::string &text, const std::string &pattern, int &compares)
{
    compares = 0;
    std::vector<int> last = buildLastFunction(pattern);
    int n = text.size();
    int m = pattern.size();
    int i = m - 1;
    if (i > n - 1) // pattern longer than text?
        return -1; // ...then no match
    int j = m - 1;
    do
    {
        compares++;
        if (pattern[j] == text[i])
            if (j == 0)
            {
                return i; // found a match
            }
            else
            { // looking-glass heuristic
                i--;
                j--; // proceed right-to-left
            }
        else
        { // character-jump heuristic
            i = i + m - std::min(j, 1 + last[text[i]]);
            j = m - 1;
        }
    } while (i <= n - 1);
    return -1; // no match
}
// construct function last
std::vector<int> buildLastFunction(const std::string &pattern)
{
    const int N_ASCII = 128; // number of ASCII characters
    int i;
    std::vector<int> last(N_ASCII); // assume ASCII character set
    for (i = 0; i < N_ASCII; i++)   // initialize array
    {
        last[i] = -1;
    }
    for (i = 0; i < pattern.size(); i++)
    {
        last[pattern[i]] = i; // (implicit cast to ASCII code)
    }
    return last;
}
```

```
// KMP algorithm
int KMPmatch(const std::string &text, const std::string &pattern, int &compares)
{
    compares = 0;
    int n = text.size();
    int m = pattern.size();
    std::vector<int> fail = computeFailFunction(pattern);
    int i = 0; // text index
    int j = 0; // pattern index
    while (i < n)
    {
        compares++;
        if (pattern[j] == text[i])
        {
            if (j == m - 1)
            {
                return i - m + 1; // found a match
            }
            i++;
            j++;
        }
        else if (j > 0)
        {
            j = fail[j - 1];
        }
        else
        {
            i++;
        }
    }
    return -1; // no match
}

std::vector<int> computeFailFunction(const std::string &pattern)
{
    std::vector<int> fail(pattern.size());
    fail[0] = 0;
    int m = pattern.size();
    int j = 0;
    int i = 1;
    while (i < m)
    {
        if (pattern[j] == pattern[i])
        { // j + 1 characters match
            fail[i] = j + 1;
            i++;
            j++;
        }
        else if (j > 0) // j follows a matching prefix
            j = fail[j - 1];
        else
        { // no match
            fail[i] = 0;
            i++;
        }
    }
    return fail;
}

int bruteForce(const std::string &text, const std::string &pattern, int &compares)
{
    compares = 0;
    int n = text.size();
    int m = pattern.size();
    int j = 0;
    for (int i = 0; i < n - m + 1; i++)
    {
        j = 0;
        compares++;
        while (j < m && text[i + j] == pattern[j])
        {
            compares++;
            j++;
        }
        if (j == m)
        {
            return i;
        }
    }
    return -1;
}

//Extra Credit, uses BM's looking glass heuristic in the brute force algorithm
int bruteForceLookingGlass(const std::string &text, const std::string &pattern, int &compares)
{
    compares = 0;
    int n = text.size();
    int m = pattern.size();
    int j = m - 1;
    int i = m - 1;
    int shift = 1;
    do
    {
        compares++;
        if (pattern[j] == text[i])
        {
            if (j == 0)
            {

                return i; // found a match
            }
            else
            { // looking-glass heuristic
                shift++;
                i--;
                j--; // proceed right-to-left
            }
        }
        else
        {
            i += shift;
            shift = 1;
            j = m - 1;
        }
    } while (i < n);
    return -1;
}
void output(const std::string &text, const std::string &pattern)
{
    int compares = 0;
    std::cout << "\nText: " << text
              << "\nPattern: " << pattern
              << "\nBM matching: index = " << BMmatch(text, pattern, compares) << " compares: " << compa
              << "\nKMP matching: index = " << KMPmatch(text, pattern, compares) << " compares: " << co
              << "\nBrute Force mathcing: index = " << bruteForce(text, pattern, compares) << " compare
              << "\nBrute Force w/ Looking Glass: index = " << bruteForceLookingGlass(text, pattern, co
}
```

Output



```
Text: a pattern matching algorithm
Pattern: rithm
BM matching: index = 23 compares: 11
KMP matching: index = 23 compares: 29
Brute Force mathcing: index = 23 compares: 30
Brute Force w/ Looking Glass: index = 23 compares: 29


Text: a pattern matching algorithm
Pattern: rithn
BM matching: index = -1 compares: 8
KMP matching: index = -1 compares: 30
Brute Force mathcing: index = -1 compares: 29
Brute Force w/ Looking Glass: index = -1 compares: 26


Text: GTTTATGTAGCTTACCTCCTCAAAGCAATACACTGAAAA
Pattern: CTGA
BM matching: index = 32 compares: 21
KMP matching: index = 32 compares: 44
Brute Force mathcing: index = 32 compares: 48
Brute Force w/ Looking Glass: index = 32 compares: 46


Text: GTTTATGTAGCTTACCTCCTCAAAGCAATACACTGAAAA
Pattern: CTGG
BM matching: index = -1 compares: 13
KMP matching: index = -1 compares: 48
Brute Force mathcing: index = -1 compares: 50
Brute Force w/ Looking Glass: index = -1 compares: 40
```