Project 4: String Matching with Dynamic Programming – Part 2

CPSC 335 - Algorithm Engineering
Fall 2017
Instructors: Kevin Wortman (kwortman@fullerton.edu), Laurence Bernstein (lebernstein@fullerton.edu)

Abstract

This project is a follow on to Project 3. You will implement the Local String Alignment algorithm. that solve the same problem. This is a dynamic programming algorithm that is an extension of the one you wrote in project 3, but uses penalties and performs a local alignment instead of a global alignment.

The Problem

The local string matching problem is another version of the string matching problem that we investigated in Project 3. However, the local alignment seeks to find the best match for a string within another string, allowing deletions at the beginning and the end to not affect the score.

Example:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
         JKLXNO
```

This is a very useful matching when trying to locate a small piece of DNA within the larger context of an entire genome, or when attempting to find a piece of a protein (a peptide) within the string representing the whole protein. Often these alignments use penalties based on the probabilities that insertions, deletions and particular substitutions occur. We will use a "penalty matrix" to describe these penalties.

Local String Alignment
input: two strings: S1 and S2 and a scoring matrix P. *output:* the score of the substring of S2 which has the maximum score of all possible substrings of S2. Also the strings representing the alignment of the two strings.

The Algorithm

You must implement the following algorithm for the local string alignment problem.

```
local_alignment(S1, S2, P, M1, M2):
      n = size(S1)
      m = size(S2)
      D[n+1][m+1]                   // The main DP array
      B[n+1][m+1]                   // For the back pointers
      for i = 0 to n
            for j = 0 to m
                  D[i][j] = 0
                  B[i][j] = '?'
      for i = 1 to n
            for j = 1 to m
                  up = D[i-1][j] + P(S1[i-1], "*");
                  left = D[i][j-1] + P("*", S2[j-1]);
                  diag = D[i-1][j-1] + P(S1[i-1], S2[j-1]);
                  if left > up
                     if left > diag
                        B[i][j] = 'l'
                     else
                        B[i][j] = 'd'
                  else
                     if up > diag
                        B[i][j] = 'u'
                     else
                        B[i][j] = 'd'
                  D[i][j] = max(up, left, diag, 0)


      // The best score is located somewhere on the bottom row
      best_score = 0
      best_i = size(S1)
      for j = 1 to m
            if D[best_i][j] > best_score
                  best_score = D[best_i][j]
                  best_j = j
```

```
// Now we follow the back pointers to get the exact alignment
done = false
i = best_i
j = best_j
M1 = ""
M2 = ""
while !done
      if B[i][j] = 'u'
            M1 = M1 + string1[i-1]
            M2 = M2 + "*"
            i--
      else if B[i][j] = 'l'
            M1 = M1 + "*"
            M2 = M2 + string2[j-1]
            j--
      else if B[i][j] = 'd'
            M1 = M1 + string1[i-1]
            M2 = M2 + string2[j-1]
            i--
            j--
      else if B[i][j] = '?'
            done = true;

// Reverse the strings because we created them backwards
reverse(M1)
reverse(M2)

return best_score
```

The Search Methods

You must also implement the following method that find the best match amongst all the loaded proteins:

```
local_alignment_best_match(L, S, P, M1, M2):
      best_protein = protein[0]
      best_score = 0;
      m1 = ""
      m2 = ""
      for i = 0 to size(L) - 1
            score = local_alignmentlocal_alignment(L[i], S, P, M1, M2)
            if score > best_match
                  best_score = score;
                  best_protein = L[i]
                  M1 = m1
                  M2 = m2
    return best_protein
```

Implementation

You are provided with the following files.

1. `proteins_large.txt` is a reduced version of the protein database for the species Saccharomyces cerevisiae which is a species of yeast. The file contains the protein descriptions along with the full sequences of each of the proteins. The file is in the standard "FASTA" format used by most researchers working with protein lists.
2. `project.hh` is a C++ header that defines skeleton functions for the two algorithms described above. You are responsible for implementing these two functions:
    - `local_alignment()`
    - `local_alignment_best_match()`

   There are also a provided load functions to load the proteins_large.txt file into the ProteinVector and the penalty data into the penalty matrix object..
3. `project4_test.cc` is a C++ program with a `main()` function that performs unit tests on the functions defined in `project4.hh` to see whether they work, prints out the outcome, and calculates a score for the code. You can run this program to see whether your algorithm implementations are working correctly.

4. `project4_main.cc` is an additional C++ program with a `main()` function that runs the exhaustive_best_match() and dynamicprogramming_best_match() methods on 5 test strings (peptides) of various lengths. You will run this program to generate the results data for your report.
5. `rubrictest.hh` is the unit test library used for the test program; you can ignore this file.
6. `timer.hh` contains a small `Timer` class that implements a precise timer using the `std::chrono` library in C++11.
7. `README.md` contains a brief description of the project, and a place to write the names and CSUF email addresses of the group members. You need to modify this file to identify your group members.

Obtaining and Submitting Code

This document explains how to obtain and submit your work:

GitHub Education Instructions

Here is the invitation link for this project:

https://classroom.github.com/g/Tzga-GJL

What to Do

First, add your group member names to `README.md`. Implement all the skeleton functions in the provided header file. Use the test program to check whether your code works.

Once you are confident that your algorithm implementations are correct, you only need do the following:

1. Run the provided main program and report the matches that your algorithm found. You should report:
    a. The name of the protein matched in each case
    b. The score of the match
    c. The two strings representing the alignment of the match

Grading Rubric

Your grade will be comprised of three parts: *Form, Function,* and *Analysis.*

Function refers to whether your code works properly as defined by the test program. We will use the score reported by the test program as your Function grade.

Form refers to the design, organization, and presentation of your code. A grader will read your code and evaluate these aspects of your submission.

Analysis refers to the correctness of your results.

The grading rubric is below.

1. Function - 12 points:
    a. `local_alignment()` passes all tests: 8 points
    b. `local_alignment_best_match()` passes all tests: 4 points
2. Form - 18 points
    a. README.md complete: 3 points
    b. Style (whitespace, variable names, comments, etc.): 3 points
    c. Design (where appropriate, uses encapsulation, helper functions, data structures, etc.): 3 points
    d. Craftsmanship (no memory leaks, gross inefficiency, taboo coding practices, etc.): 3 points
3. Analysis - 20 points
    a. Results: 20 points (4 points each match)

Deadline

The project deadline is Monday, December 11, 11 am.

You will be graded based on what you have pushed to GitHub as of the deadline. Commits made after the deadline will not be considered. Late submissions will not be accepted.