ECE ÇINAR, 150150138 27 MAY, 2020

BLG336E - Analysis of Algorithms II, Spring 2020
Homework 3

Test Selection and Prioritization with Dynamic Programming

Table of Contents

1. Code Explanation

1.1.TestSuite Class

Each test suite is represented by a TestSuite instance.

1.1.1. Getters & Setters

ID, number of bugs, running time and test case frequencies are attributes of each test suite. There are getter and setter methods written for each attribute of a suite.

1.1.2. printTest

This method prints ID, number of bugs and running time information of a test suite.

1.2.Optimization Class

Optimization class consists of many optimization methods such as selectSubset, calculateDistance, prioritize and also other necessary methods.

1.2.1. setTests

Pushes given test suites into the list of suites.

1.2.2. setTimeLimit

Sets the time limit read from the input text file.

1.2.3. selectSubset

This method selects a subset of tests that maximizes the number of bugs subject to the time limit. It only works on discrete time data. After calculating each cell of the table, the algorithm traces back to find the selected test suites. It doesn't return anything but it fills the list of selected test suites. This method will be explained in detail in Part 1 section.

1.2.4. calculateDistance

This method calculates the dissimilarity between two integer vectors and returns it. Dissimilarity rate is calculated based on the minimum number of operations that should be performed in order to convert one vector into the other. This algorithm and its operations will be explained in detail in Part 2 section.

1.2.5. orderFrequencies

This method converts an integer vector representing the frequency profiles into another integer vector representing the same frequencies in an ordered manner. Multimaps, which are ordered maps that allow having elements with duplicate keys, are used. Integers of each frequency profile and their indexes are inserted into a multimap as key-value pairs. Since multimaps keep keys in an ordered manner, method iterates over the map and copies each value into an integer vector and obtains ordered frequencies.

Ex.: 1-3-2-0-2-1 → 4-1-6-3-5-2

1.2.6. prioritize

This method arranges order of test cases for each selected test suite and stores the orderings in a vector of integer vectors. Firstly, it calculates number of zeros for each test case of a test suite. Then, the test case(s) with lowest number of zeros is/are inserted into a vector representing the final ordering of test cases. After that, *orderFrequencies* method is called in order to order the test case frequencies. Finally, *calculateDistance* is called in order to calculate dissimilarity between the remaining test cases and the base test case which has the highest coverage. Test cases are sorted in descending dissimilarity order and added to the final ordering vector.

1.2.7. printOutputs

Prints necessary information of calculations.

1.3. File Class

1.3.1. Constructor

Opens the text file.

1.3.2. read

Uses stringstream in order to read data. Firstly, it reads the time limit and calls *setTimeLimit* for the Optimization instance. It skips the next two lines since they are empty. After that, it reads lines into a stringstream and tokenizes the stream into strings. Attributes of TestSuite instances are set using the tokenized information and those instances are added to the test suite list of Optimization instance. Finally, the method returns the Optimization instance.

1.3.3. Destructor

Closes the text file.

2. Part 1

2.1. Mathematical Representation

Test suite selection algorithm is implemented in *selectSubset* method. Mathematical representations of the problem and optimization algorithm are given below. The number of bugs in test suite i is represented by b_i and running time of the same test suite is represented by t_i. T represents the time limit and c_i represents existence of the test suite.

$$\textbf{Problem}: \text{ maximize } \sum_{i=1}^{n} b_i c_i \text{ subject to } \sum_{i=1}^{n} t_i c_i \leq T \,, c_i \in \{0,1\}$$

$$\textbf{Algorithm}: OPT(i,t) = \begin{cases} 0 & , \quad if\ i = 0, \\ OPT(i-1,t\,) & , \quad if\ t_i > t, \\ \max\{\, OPT(i-1,t), b_i + OPT(i-1,t-t_i)\}, & \text{otherwise.} \end{cases}$$

2.2. Complexity

According to C++ documentations, *max*, *push_back* and *size* methods have constant complexity so they won't affect the overall complexity. Also, variable initialization, declaration and mathematical operations have constant time complexity. The number of columns in our table, $\boldsymbol{T}$, is equal to the time limit plus one. The number of rows in our table, $\boldsymbol{n}$, is equal to the number of test suites plus one. Based on the facts that size of the table is $n\ to\ T$ and each cell will be updated once, complexity of table updates is **O($\boldsymbol{nT}$)**.

Trace-back operation will check each row of the table once. Operations performed while checking each row have constant time so that they won't affect the overall complexity. Since there are n rows, complexity of trace-back is **O($\boldsymbol{n}$)**. **Overall complexity of the algorithm is O($\boldsymbol{nT}$).**

2.3.Real Number Solution

This algorithm works only on discrete running time values since it assigns one table column for each integer from 1 to the time limit. Using this algorithm on real numbers can introduce both syntax and logical errors. For instance, if the time limit is a real number vector initialization will introduce a syntax error since size of a vector must be integer.

However, we can scale real numbers into integers by multiplying them by 10^n where n represents the number of decimal digits. The pseudo code below finds the running time that has maximum number of decimal digits (n) and multiplies each number by 10^n. Inserting the following code at the beginning of our algorithm enables working on real numbers. However, it should be noted that if n is too large space complexity can increase significantly. Also, using long instead of integer can be a logical decision since numbers can exceed integer limitations.

```
set max := 0
for item in runtimes
    set temp := item
    set count := 0
    while temp not equal to int temp do
        multiply temp by 10
        increment count
    if count greater than max
        set max := count

for item in runtimes
    multiply item by 10^max
```

3. Part 2

3.1.Algorithm

The edit distance algorithm I used in my implementation is “Levenshtein Distance”. There are three different operations within the scope of this algorithm which are **replacement**, **deletion** and **insertion**. Costs of deletion and insertion are set to one yet cost of replacement is state dependent. If the two characters being compared are identical, cost of replacement is set to zero since there isn’t an additional operation being performed in this case. Otherwise, it is one.

For the sake of example, let’s assume that the two characters we are comparing are ‘a’ and ‘a’. If we replace ‘a’ with ‘a’, it is clear that nothing changes so the cost of this operation is zero which is same as doing nothing. However, deleting or inserting an ‘a’ would change the output anyways so that costs of these operations are always one.

In my implementation, *calculateDistance* method calculates each cell of the table using information above and the mathematical representation below. Instead of using strings, I used integer vectors in order to represent the comparison inputs. The final calculation, which represents the overall cost of comparison, is returned by the method.

3.2.Mathematical Representation

In the representation below, a and b represent the vectors to be compared. a_i and b_i represent i^{th} element of vectors a and b respectively. $1_{(a_i!=b_i)}$ indicates that if a_i and b_i are not equal, last statement is increased by one.

$$L_{a,b}(i,j) = \begin{cases} \max(i,j) & , \quad if\ min(i,j) = 0 \\ \min \begin{cases} L_{a,b}(i-1,j)+1 \\ L_{a,b}(i,j-1)+1. \\ L_{a,b}(i-1,j-1)+1_{(a_i!=b_i)} \end{cases} & , \quad otherwise. \end{cases}$$

3.3.Complexity

In "Levenshtein Distance" algorithm, each cell of the table is updated once which means that complexity of the algorithm is equal to table size. If size of the two vectors are n_1 and n_2, overall complexity of the algorithm will be **O($n_1 n_2$)**. In our case, since size of each frequency profile of a test suite are equal, comparisons will be done between vectors of equal size. This means that overall complexity will be **O(n^2).**

4. Compilation & Outputs

In my implementation, I used C++11 features. The program should be compiled using -std=c++11 command. A sample execution is given below.

```
[[cinare15@ssh ~]$ g++ -std=c++11 150150138.cpp
[[cinare15@ssh ~]$ ./a.out data.txt
 Selected test suites:
 Test Id: TS2 Bugs Detected: 13 Running Time: 7
 Test Id: TS3 Bugs Detected: 23 Running Time: 11
 Test Id: TS4 Bugs Detected: 15 Running Time: 8
 Total running time: 26
 Test case orderings:
 TS2 1
 TS3 5 6 11 12 2 4 10 1 3 8 9 7
 TS4 3 4 1 2
 [cinare15@ssh ~]$
```