CSC 342 Program Five:

Warshall’s Algorithm

Cade Archer

Jace Archer

CSC 342.001

Submission Account: cs342102

November 15, 2016

1. Problem Statement

For Program 5, I was tasked with building a transitive closure by implementing Warshall’s Algorithm. I ran the algorithm with several different sized matrices to see if the algorithm followed a cubic time efficiency.

II. Algorithm

Overview:

For warshall.java, the program read in a String representing a file name. This file holds an adjacency matrix that will be converted into a transitive closure. To build the adjacency matrix, it reads in the file line by line and split the numbers using the spaces in between them. Each line is put into a different row in a two-dimensional, integer array. Once the adjacency matrix is initialized, the transitive closure is built by using Warshall’s algorithm. Inside the *warshallAlgorithm* method, I passed in the two-dimensional array, the adjacency matrix. This passed in matrix is assigned to *R*, another two-dimensional, integer array. I also store the matrix’s length in an integer, *n*. The bulk of the algorithm is done in the next stage of the method, where three for-loops build the transitive closure. A node is looked at to see if a *1* needs to placed or not. A *1* is placed in a node if there is already a *1* in that node’s column and in that node’s row. If one of, or neither, has a *1* in that place, a *0* is stored. This is done for each node in the matrix. Once this is completed, the transitive closure stored in *R*, is returned. The last part of our program is the *print* method, that will print out the transitive closure passed in.

Stated:

*Warshall*(A[1...n,1...n])

//Implements Warshall’s Algorithm for computing the transitive //closure

//Input: The adjacency matrix A of a digraph with n vertices

//Output: *The transitive closure of the digraph*

R(0)← A

**for** k ← 1 **to** n **do**

**for** i ← 1 **to** n **do**

**for** j ← 1 **to** n **do**

R(k)[i, j] ←R(k-1)[i, j] **or** (R(k-1)[i, k] **and** R(k-1)[k, j])

**return** R(n)

Design:

When I coded the algorithm, I kept the design similar to how it was stated in the book. The main change I made was making our loops start at 0 instead of at 1 so I was using the correct array indexes.

III. Implementation

Dictionary of Variables Used:

String fileName: Holds the name of the file read in from the console

String lineOfInput: Holds a single line read in from the file

String [ ] splitInput: Holds each element of the read in line, split using a space as a delimiter

int numVertices: holds the number of vertices in the adjacency matrix

int [ ] [ ] adjacecyMatrix: holds the adjacency matrix read in from the file

int [ ] [ ] transitiveClosure: holds the transitive closure created using Warshall’s algorithm

File inputFile: holds the file the adjacency matrix is read in from

IV. Experiment

Location Program Ran: Windows Environment

Machine Used:

OS: Windows 10

RAM: 16GB

Hard Drive: 1TB Solid State

Compiler Used: Dr. Java

Files Used: warshall.java, d.txt, readme.txt

Example Test Files submitted: 6x6.txt, 8x8.txt, 10x10.txt

Read Me File:

Our program, warshall.java, requires one argument to be passed in on the command line. The argument passed in should be the name of the file that holds the adjacency matrix to build the transitive closure from.

Example of how to run the program on UNIX once its been compiled:

java warshall d.txt

Parameters:

Our program requires that a filename be inserted on the command line. This file should store an adjacency matrix that the transitive closure will be built from. The *warshallAlgorithm* itself takes in an adjacency matrix in the form of a two-dimensional integer array. The *print* method takes in the transitive closure matrix in the form of a two-dimensional integer array.

Sample Executions:

Input:

0 1 0 0

0 0 0 1

0 0 0 0

1 0 1 0

Output:

*run warshall d.txt*

The Transitive Closure for the 4x4 adjacency matrix entered:

1 1 1 1

1 1 1 1

0 0 0 0

1 1 1 1

Program Execution Time: 0.0213 milliseconds

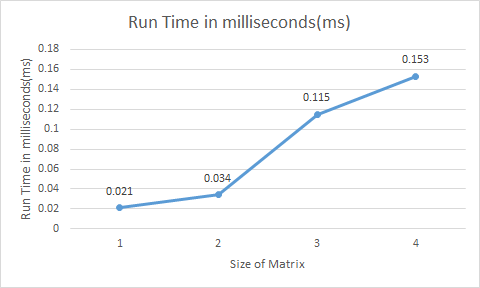
V. Analysis

Results:

To test Program 5 I ran the program with four different sizes of matrices: 4x4, 6x6, 8x8, 10x10. The only thing timed was the completion of the transitive closure. Reading in the data, creating the adjacency matrix, and printing out the transitive closure are all excluded from the timing.

Statistics Gathered:

|  |  |
| --- | --- |
| Size of Matrix | Run Time in milliseconds (ms) |
| 4x4 | .021 |
| 6x6 | .034 |
| 8x8 | .115 |
| 10x10 | .153 |



(Legend: 1 = 4x4 matrix, 2 = 6x6 matrix, 3 = 8x8 matrix, 4 = 10x10 matrix)

Comparisons:

Between the 4x4 and 6x6 matrices, there wasn’t too much of an increase in runtime. The sharp increase came when the matrix size increased to 8x8. At this point the graph gained its cubic appearance. With the last matrix, 10x10 size, the runtime increased, sharper than the first jump but smaller than second jump.

VI. Conclusions

Final Results and Observations:

Our results show that with each growth of matrix size, the completion of the algorithm took longer and longer. The graph above does show that Warshall’s algorithm grows at a cubic rate. Our data set is small so the graph isn’t perfectly cubic, but it does mirror it enough so that our conclusion can be drawn.

Detailed Time Spent:

November 8th: 2 Hours spent, researching and coding program

November 11th: 2 Hours spent, Documentation and Testing

November 12th: 3 Hours spent, Documentation and Testing

References:

warshall.java was created by using the algorithm found in our textbook “Introduction to The Design and Analysis of Algorithms” (page 307).