**November 2014**

CSC 342 – Exploration 5

James Francis/Sam Jentsch

Account Numbers: CSC342107/CSC342111

08

**Fall**

­­

Table of Contents

Problem Statement 3

Algorithm 3

Running our programs: 4

Implementation Similarities 5

Warshall’s Algorithm Implementation 5

DFS Brute Force Algorithm Implementation 5

Experiment 6

Generating Data sets 6

Issues 6

Machines 6

Timing Methods 7

Data Files and Sample Output 7

Charts and Timing Results 8

gprof 12

hprof 12

Conclusions 12

Future Work 13

Problems Encountered 14

Time Chart 14

Appendix A – C++ Implementation Listing: Warshall’s 15

Appendix B – Java Implementation Listing: Warshall’s 16

Appendix C – C++ Implementation Listing: Brute Force 18

Appendix D – Java Implementation Listing: Brute Force 20

Appendix E – References 21

# Problem Statement

Use Warshall’s algorithm and a Brute Force technique to find the transitive closure of a given graph (represented as an adjacency matrix).

The transitive closure of a graph is a boolean matrix in which each element in the row and column (i, j) is 1 if and only if there is a path from the ith vertex to the jth vertex of the graph. If a path doesn’t exist between the vertices, (i, j) is 0.

Essentially, both Warshall’s algorithm and the Brute Force algorithm find the reachability matrix for a given graph. We ran a series of tests on both algorithms and compared the times each required to find the transitive closure for the graphs supplied.

# Algorithm

Warshall’s Algorithm for finding the transitive closure of a graph:

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

*R ← A*

*for k ← 1 to n do*

*for i ← 1 to n do*

*for j ← 1 to n do*

*R[i,j] ← Rprev[i, j] | (Rprev[i, k] and Rprev[k, j])*

*return R*

Warshall’s algorithm is an example of dynamic programming. Each vertex from R0 to Rn is computed using information available in the preceding matrix. For example, computing Rk requires only R(k-1), where R(k-1) is the reachability matrix for the graph whose paths include only nodes numbered less than or equal to (k-1). Rk will be the reachability matrix for the graph whose paths include nodes numbered up to k.

Warshall’s algorithm operates using the following formula:

rij(k) = rij(k-1)  or (rik(k-1) and rkj(k-1))

DFS Brute Force algorithm for finding the transitive closure of a graph:

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

*for vertex ← 1 to n*

*DFS(vertex)*

*For each vertex visited in the DFS, mark the vertex as reached in the*

*Reachability matrix R*

*Reset the visited nodes*

*return R*

*DFS(v)*

*for each unvisited vertex adjacent to v*

*mark the node as visited*

*DFS(adjacentNode)*

The DFS Brute Force algorithm operates by performing a depth-first search on each node in the graph. The nodes that are visited using each node as a starting vertex for a depth-first search are recorded and copied into the reachability matrix R for the node the search was performed on. After all n nodes have been used as the source for a depth first search and the reachable nodes for each node have been recorded, the transitive closure of the graph has been found.Implementation

### Running our programs:

**All files are submitted in account: cs342107**

The class files used for our exploration are as follows.

for C++:

Warshall\_C.cpp

BFTransitiveClosure\_C.cpp

For JAVA:

Warshall\_Java.java

BFTransitiveClosure\_Java.java

Each of these files when compiled, separately, will produce a program that will generate a boolean matrix representation of a transitive closure for a supplied adjacency matrix.

Assuming that **Warshall\_C.cpp** was compiled as **Warshall\_C**,what follows is an example of how to execute the program:

**Warshall\_C ../DATAFILES/Node4.dat**

The above line will run Warshall’s algorithm on an adjacency matrix with 4 nodes. The Node4.dat file is formatted in the following manner:

4

0 1 0 0

0 0 0 1

0 0 0 0

1 0 1 0

Where 4 represents the number of nodes in the graph, and 1 represents a direct path to another node, and a 0 represents no direct path to a node.

### Implementation Similarities

The implementations run by passing a command line argument containing a path to a data file representing an adjacency matrix. Data files are expected to contain the number of nodes in the graph followed by the graph itself.

For example:

4

0 0 1 0

0 1 0 1

1 0 0 0

0 0 0 0

### 

Both implementations and both algorithms use two-dimensional integer arrays to represent the adjacency matrix of the graph.

This adjacency matrix is read from a file (with a path supplied as a command line argument) and passed to the algorithm for processing.

The dimension of the adjacency matrix is also recorded and used throughout all programs when processing the adjacency matrix.

### Warshall’s Algorithm Implementation

Warshall’s algorithm was implemented using a single two-dimensional integer array as the adjacency matrix throughout the algorithm. All changes are made to this single array, so the algorithm doesn’t require any additional allocated space for the intermediate reachability matrices.

### DFS Brute Force Algorithm Implementation

The DFS Brute Force algorithm uses global variables for the adjacency matrix of the original graph, the reachability matrix, and an array to keep track of nodes visited during a depth-first search originating at a source vertex. Additionally the dimension of the graph is stored as a global variable. We decided to store these data members globally to simplify the implementation of the depth-first search.

# Experiment

The experiment was run on the Unix server and a Mac OS X machine for each of the implementations for analysis. Data sets were generated using the method described in the Generating Data Sets section for graph sizes beginning at n = 4 and increasing by powers of 2 up to 4096. The results of the timing for each graph were recorded and graphed for analysis using a log2 scale.

### Generating Data sets

|  |
| --- |
| generateAdjacencyMatrix(int numberOfNodes)     for ( i ← 0; i < numberOfNodes; i++)         for ( j ← 0; j < numberOfNodes; j++)         if (i == j)         adjMatrix[i][j] ← 0     else     int randomNumber = random()     adjMatrix[i][j] ← randomNumber     adjMatrix[j][i] ← randomNumber |

The above was used to generate adjacency matrices of different sizes. The random number was generated between 0 and 1 and as sample data for finding the transitive closure of the graphs created.

### Issues

Initially, we attempted to run the experiment using data files ranging from n = 4 to n = 10, but we found that our times were not varying or large enough to find any sort of trend in the data. We remedied this by increasing the size of our data files and changing the scale of the graph appropriately.

### Machines

The machines used for testing include:

* **15” Macbook Pro**
  + OS: OSX 10.10
  + MEMORY: 8 GB 1600 MHz DDR3
  + PROCESSOR: 2GHz quad-core Intel core i7
* **13” Macbook Pro**
  + OS: OSX 10.10
  + MEMORY: 8 GB 1600 MHz DDR3
  + PROCESSOR: 2.4 GHz Intel Dual Core i5
* **SFA Unix Server**
  + OS: Linux 2.6.18-371.12.1.el5PAE i686
  + MEMORY: Undetermined (required sudo access)
  + PROCESSOR: eight-core Intel(R) Xeon(TM) MP CPU 3.66GHz

### Timing Methods

Due to The differences with time granularity between Java and C++ we tried to be as precise as possible while maintaining some level of uniformity between the data produced. Given that C++ can only report time in milliseconds using the tools provided for us in the handouts, and Java can use a nanosecond timer, giving us 1E06 more precision, we needed to convert the reported Java times and round to the nearest millisecond, to have results similar to what we were getting with C++. All the results reported herein are the result of either output from a C++ timer method provided or conversion to milliseconds from nanoseconds and rounding to two decimal places.

Timing was achieved through first preparing the algorithm for execution (reading in appropriate test data from the file and initializing variables), starting the timer, executing the methods implementing the algorithm, and stopping the timer.

### Data Files and Sample Output

The data files used were randomly generated using the method described in the Generating Data Sets section. The following files shown and their corresponding outputs are selected from some of the smaller files to conserve space (and trees):

Example Node4.dat:

4

0 1 0 0

0 0 0 1

0 0 0 0

1 0 1 0

Input: Warshall\_C Node4.dat

Output:

1 1 1 1

1 1 1 1

0 0 0 0

1 1 1 1

### Charts and Timing Results

UNIX COMPARISONS

Charts

MAC COMPARISONS

### gprof

gprof warshall

Flat profile:

Each sample counts as 0.01 seconds.

% cumulative self self total

time seconds seconds calls s/call s/call name

99.60 1.61 1.61 1 1.61 1.61 warshall(int\*\*, int)

0.62 1.62 0.01 main

0.00 1.62 0.00 2 0.00 0.00 Time1(int)

0.00 1.62 0.00 1 0.00 0.00 global constructors keyed to main

C++ profiling, using gprof leads to results similar to JAVA profiling. All significant time spent in our implementation was within the warship function, with minimum amounts of time being spent in/on other functions.

### hprof

CPU SAMPLES BEGIN (total = 59) Mon Nov 10 22:02:49 2014

rank self accum count trace method

1 47.46% 47.46% 28 300184 Warshall\_Java.warshall

2 8.47% 55.93% 5 300182 java.util.regex.Pattern$Ques.match

3 1.69% 57.63% 1 300146 java.text.NumberFormat.getInstance

4 1.69% 59.32% 1 300134 java.util.ResourceBundle.getBundleImpl

5 1.69% 61.02% 1 300026 java.util.zip.ZipFile.getInflater

6 1.69% 62.71% 1 300110 sun.util.LocaleServiceProviderPool.<init>

7 1.69% 64.41% 1 300164 java.util.regex.Matcher.reset

8 1.69% 66.10% 1 300165 java.nio.Buffer.checkIndex

9 1.69% 67.80% 1 300166 java.nio.HeapCharBuffer.get

10 1.69% 69.49% 1 300167 java.nio.CharBuffer.toString

11 1.69% 71.19% 1 300168 java.util.regex.Matcher.end

12 1.69% 72.88% 1 300169 java.util.regex.Pattern$GroupHead.match

13 1.69% 74.58% 1 300170 java.util.regex.Matcher.lookingAt

14 1.69% 76.27% 1 300171 java.nio.HeapCharBuffer.subSequence

15 1.69% 77.97% 1 300172 java.util.Scanner.getCompleteTokenInBuffer

16 1.69% 79.66% 1 300173 java.util.Scanner.getCompleteTokenInBuffer

17 1.69% 81.36% 1 300174 java.util.Scanner.getCompleteTokenInBuffer

18 1.69% 83.05% 1 300175 java.util.Scanner.getCompleteTokenInBuffer

19 1.69% 84.75% 1 300176 java.lang.Thread.currentThread

20 1.69% 86.44% 1 300177 java.util.Scanner.nextInt

21 1.69% 88.14% 1 300178 java.util.regex.Pattern$Curly.match0

22 1.69% 89.83% 1 300179 java.util.regex.Pattern$Start.match

23 1.69% 91.53% 1 300180 java.util.regex.Pattern$Curly.match

24 1.69% 93.22% 1 300181 java.util.Scanner.readInput

25 1.69% 94.92% 1 300183 java.util.regex.Pattern$Curly.match

26 1.69% 96.61% 1 300095 sun.nio.ch.FileChannelImpl.<init>

27 1.69% 98.31% 1 300058 java.util.regex.Pattern$CharPropertyNames.defCtype

28 1.69% 100.00% 1 300089 java.io.FileInputStream.getChannel

CPU SAMPLES END

After running Hprof with Node512.dat we can see that 47% of the program’s operation was spent in the warship method of our implementation. Minimal amounts of time were spent in methods associated with JAVA Library objects/methods that support our implementation.This is an expected result.

# Conclusions

Warshall’s algorithm is O(n3).

The depth-first search algorithm by itself is O(n2). Because the Brute-Force algorithm requires performing a DFS on each of the vertices in the graph, the Brute-Force algorithm should be O(n3) as a whole, the same efficiency class as Warshall’s algorithm.

In our tests, we found that the Brute Force algorithm was consistently faster than Warshall’s algorithm (up to 4.49 times faster on the OS X machine). We believe this is a result of the Brute Force algorithm not necessarily running n3 times for each data set. Although Warshall’s algorithm consists of a triply nested loop ensuring that the efficiency will be O(n3), the Brute Force algorithm likely falls in between O(n2) and O(n3) for many graphs. This is because the graph only calls DFS on a vertex within the second loop if it is adjacent to the vertex passed, therefore the O(n3) efficiency would likely occur in a worst case scenario consisting of a fully connected graph.

The text mentions that for sparse graphs represented by their adjacency lists, the Brute Force DFS approach has a better asymptotic efficiency than Warshall’s algorithm. We believe a somewhat (but less drastic) occurrence might be taking place for our implementations of the algorithm.

Another observation we made was the Java implementation of both algorithms was actually faster than the C++ implementation:

For the Brute Force algorithm, the Java implementation was 1.62 times faster than the C++ implementation on OS X for a data size of 512. We believe this might be a result of the JVM’s management of the call stack, possibly leading to more efficient handling of the recursive calls required for the Depth-First search portion of the algorithm.

For Warshall’s algorithm Java was 2.51 times faster than C++ for a data size of 512. We were surprised to see this result for Warshall’s algorithm as C++ usually seems to be faster at processing two-dimensional primitive type arrays than Java. It is possible that the Java compiler performed some additional optimization on the triply nested loop than the C++ compiler, contributing to the performance gap observed.

### Future Work

Warshall’s algorithm doesn’t appear to be consistently faster than the Brute Force algorithm. It seems that using Warshall’s algorithm might be extremely useful in scenarios where the intermediate solutions generated by Warshall’s algorithm are required.

### Problems Encountered

Problems encountered include the issue with the small data sets, remedied by increasing the size and range of the data tested. Additionally, a stack overflow occurred for the Java implementation of the Brute Force algorithm on the Unix server for a data set of size 4096. We believe this is a result of the stack size allocated for the JVM on that machine and the number of recursive calls required by the DFS algorithm.

### Time Chart

|  |  |  |  |
| --- | --- | --- | --- |
|  | Sunday | Monday | Tuesday |
| James | 3.5 hrs | 6 hrs | 3 hrs |
| Sam | 3.5 hrs | 6 hrs | 3 hrs |

# Appendix A – C++ Implementation Listing: Warshall’s

//

// main.cpp

// Warshall\_C

//

// Created by Sam Jentsch on 11/9/14.

// Copyright (c) 2014 Sam Jentsch. All rights reserved.

//

#include <iostream>

#include <fstream>

#include <stdio.h>

#include <sys/timeb.h>

#include <time.h>

#include <stdlib.h>

#include <string>

#define START 0

#define END 1

using namespace std;

void warshall(int \*\*adjMatrix, int dimension);

void printArray(int \*\*numberArray, int dimension);

int Time1(int);

int main(int argc, const char \* argv[]) {

//string pathToDataFile = "../instr/AdjacencyMatrix.dat";

string pathToDataFile(argv[1]);

ifstream inputFile;

inputFile.open(pathToDataFile.c\_str());

int n, i, j;

inputFile >> n;

int \*\*adjMatrix;

adjMatrix = new int \*[n];

for (i = 0; i < n; i++) {

adjMatrix[i] = new int[n];

}

for (i = 0; i < n; i++) {

for (j = 0; j < n; j++) {

inputFile >> adjMatrix[i][j];

}

}

printArray(adjMatrix, n);

cout << endl;

//-----TIMING-----//

int time = 0;

time = Time1(START);

warshall(adjMatrix, n);

time = Time1(END);

cout << "\nTIME: " << time << endl;

return 0;

}

void warshall(int \*\*R, int dimension) {

for (int k = 0; k < dimension; k++) {

for (int i = 0; i < dimension; i++) {

for (int j = 0; j < dimension; j++) {

R[i][j] = R[i][j] | (R[i][k] && R[k][j]);

}

}

}

printArray(R, dimension);

}

void printArray(int \*\*numberArray, int dimension) {

//Print the array

int i, j;

for(i = 0; i < dimension; i++) {

for(j = 0; j < dimension; j++) {

cout << numberArray[i][j] << ' ';

}

cout << "\n";

}

}//printArray

int Time1(int flag) {

//Timing method described in assignment handout.

static struct timeb time1;

struct timeb time2;

int sec;

unsigned short mil;

if (flag == START) {

ftime(&time1);

return 0;

} else {

ftime(&time2);

if (time2.millitm < time1.millitm) {

time2.time--;

mil = time2.millitm + 1000 - time1.millitm;

} else {

mil = time2.millitm - time1.millitm;

}

sec = (int)time2.time - (int)time1.time;

return sec \* 1000 + (int)mil;

}

}

# Appendix B – Java Implementation Listing: Warshall’s

import java.io.File;

import java.io.IOException;

import java.util.\*;

public class Warshall\_Java {

public static void main(String[] args) throws IOException {

//String pathToDataFile = "/Users/SamJ/Desktop/AdjacencyMatrix.dat";

String pathToDataFile = args[0];

File inputFile = new File(pathToDataFile);

Scanner fileReader = new Scanner(inputFile);

int n, i, j;

n = fileReader.nextInt();

int [][]adjMatrix = new int[n][n];

for (i = 0; i < n; i++) {

for (j = 0; j < n; j++) {

adjMatrix[i][j] = fileReader.nextInt();

}

}

printArray(adjMatrix, n);

System.out.println();

//---TIMING---//

long startTime, stopTime, overTime = 0;

startTime = System.nanoTime();

stopTime = System.nanoTime();

//overhead time for the program is calculated

overTime = stopTime - startTime;

long time;

startTime = System.nanoTime();

//CALL HORSPOOL

warshall(adjMatrix, n);

stopTime = System.nanoTime();

time = stopTime - startTime - overTime;

System.out.println("Total time: " + (time/1000000));

}

static void warshall(int R[][], int dimension) {

for (int k = 0; k < dimension; k++) {

for (int i = 0; i < dimension; i++) {

for (int j = 0; j < dimension; j++) {

//R[i][j] = R[i][j] | (R[i][k] && R[k][j]);

if(R[i][j] == 1)

R[i][j] = R[i][j];

else if(R[i][k] == 1 && R[k][j] == 1)

R[i][j] = 1;

}

}

}

printArray(R, dimension);

}

static void printArray(int numberArray[][], int dimension) {

//Print the array

int i, j;

for(i = 0; i < dimension; i++) {

for(j = 0; j < dimension; j++) {

System.out.print(numberArray[i][j] + " ");

}

System.out.println();

}

}//printArray

}

# Appendix C – C++ Implementation Listing: Brute Force

//

// main.cpp

// BFTransitiveClosure\_C

//

// Created by Sam Jentsch on 11/10/14.

// Copyright (c) 2014 Sam Jentsch. All rights reserved.

//

#include <iostream>

#include <fstream>

#include <stdio.h>

#include <sys/timeb.h>

#include <time.h>

#include <stdlib.h>

#include <string>

#define START 0

#define END 1

using namespace std;

int \*\*adjMatrix;

int \*\*R;

int \*visited;

int dimension;

void printArray(int \*\*numberArray, int dimension);

void DFS(int node);

void DFSTransitiveClosure();

int main(int argc, const char \* argv[]) {

//string pathToDataFile = "../instr/AdjacencyMatrix.dat";

string pathToDataFile(argv[1]);

ifstream inputFile;

inputFile.open(pathToDataFile.c\_str());

int i, j;

inputFile >> dimension;

adjMatrix = new int \*[dimension];

R = new int \*[dimension];

for (i = 0; i < dimension; i++) {

adjMatrix[i] = new int[dimension];

R[i] = new int[dimension];

}

for (i = 0; i < dimension; i++) {

for (j = 0; j < dimension; j++) {

inputFile >> adjMatrix[i][j];

R[i][j] = 0;

}

}

visited = new int[dimension];

for (i = 0; i < dimension; i++) {

visited[i] = 0;

}

printArray(adjMatrix, dimension);

cout << endl;

DFSTransitiveClosure();

return 0;

}

void DFSTransitiveClosure() {

for (int vertex = 0; vertex < dimension; vertex++) {

DFS(vertex);

for (int i = 0; i < dimension; i++) {

R[vertex][i] = visited[i];

visited[i] = 0;

}

}

printArray(R, dimension);

}

void DFS(int node) {

for (int j = 0; j < dimension; j++) {

if (visited[j] == 0 && adjMatrix[node][j] == 1) {

visited[j] = 1;

DFS(j);

}

}

}

void printArray(int \*\*numberArray, int dimension) {

//Print the array

int i, j;

for(i = 0; i < dimension; i++) {

for(j = 0; j < dimension; j++) {

cout << numberArray[i][j] << ' ';

}

cout << "\n";

}

}//printArray

# Appendix D – Java Implementation Listing: Brute Force

import java.io.File;

import java.io.IOException;

import java.util.Scanner;

public class BFTransitiveClosure\_Java {

static int [][] adjMatrix;

static int [][] R;

static int [] visited;

static int dimension;

public static void main(String[] args) throws IOException {

//String pathToDataFile = "/Users/SamJ/Desktop/AdjacencyMatrix.dat";

String pathToDataFile = args[0];

File inputFile = new File(pathToDataFile);

Scanner fileReader = new Scanner(inputFile);

int i, j;

dimension = fileReader.nextInt();

adjMatrix = new int[dimension][dimension];

R = new int [dimension][dimension];

for (i = 0; i < dimension; i++) {

for (j = 0; j < dimension; j++) {

adjMatrix[i][j] = fileReader.nextInt();

}

}

printArray(adjMatrix, dimension);

System.out.println();

visited = new int[dimension];

for (i = 0; i < dimension; i++) {

visited[i] = 0;

}

//---TIMING---//

long startTime, stopTime, overTime = 0;

startTime = System.nanoTime();

stopTime = System.nanoTime();

//overhead time for the program is calculated

overTime = stopTime - startTime;

long time;

startTime = System.nanoTime();

//CALL ALGORITHM

DFSTransitiveClosure();

stopTime = System.nanoTime();

time = stopTime - startTime - overTime;

System.out.println("Total time: " + (time/1000000));

}

static void DFSTransitiveClosure() {

for (int vertex = 0; vertex < dimension; vertex++) {

DFS(vertex);

for (int i = 0; i < dimension; i++) {

R[vertex][i] = visited[i];

visited[i] = 0;

}

}

printArray(R, dimension);

}

static void DFS(int node) {

for (int j = 0; j < dimension; j++) {

if (visited[j] == 0 && adjMatrix[node][j] == 1) {

visited[j] = 1;

DFS(j);

}

}

}

static void printArray(int numberArray[][], int dimension) {

//Print the array

int i, j;

for(i = 0; i < dimension; i++) {

for(j = 0; j < dimension; j++) {

System.out.print(numberArray[i][j] + " ");

}

System.out.println();

}

}//printArray

}

# Appendix E – References

Introduction to the Design and Analysis of Algorithms by Anany Levitin

Assignment sheet – Dr. Robert Strader