Profiling transitive closure algorithms

Vreda Pieterse
Computer Science
University of Pretoria
vreda.pieterse@up.ac.za

June 25, 2016

1 Introduction

This document describes the process and artefacts that was used to gather data regarding the performance of algorithms. The profiling procedure described here was performed to measure the performance of a number of transitive closure algorithms using time and memory as metrics. This can be generalised to measure the performance of other algorithms with other data by substituting the relevant data and code. The document also covers a description of generating synthetic data used in these profiling jobs.

2 Computer configuration

2.1 Hardware

Table 1: Hardware configuration

Hardware	Description
CPU	Intel(R) Core(TM) i5-3470 Duo CPU E6750 @ 2.66GHz
Motherboard	Intel Corporation
Memory	2x 1024MB PC3200 DDR2
Hard Disk Drive	Seagate Barracuda Serial SATA Rev 3.0, SATA 7200 rpm
Size Of Disk	500GB

2.2 Operating System

All profiling jobs were performed on a computer running the Debian GNU/Linux 7 (wheezy)operating system version x86_64. The operating system was configured to run using only the terminal.

2.3 Programming language and Compiler

The programs that are observed are compiled using the standard GCC compiler for compiling C++ programs. The compiler version of c++ used is gcc 4.7.2.

To be able to measure elapsed time, the source code was instrumented with instructions to output time stamps. To enable the clock functions found in the real time (rt) library, the code had to be compiled with a flag to link the rt library.

To be able to measure memory usage, the compiled code was executed using Valgrind. To enable the execution to be observed by Valgrind it was required that the source code is compiled using the `dl` and `pthread` libraries.

2.4 Experimental design

We are interested in comparing a number of TC algorithms to one another in terms of their relative performance in terms of execution time in relation with characteristics of input data. We designed an experiment to achieve this goal. The design of the experiment is described in terms of the aspects specified by the Evaluate Collaboratory [5] in order to support the repeatability of the measurements reported in this research.

To obtain valid measurements the experimenter has to gather adequate data while minimising possible effects caused by extraneous factors [16]. There are many factors contributing to the performance of specific implementations of algorithms. We are aware of the factors in Table 2 that may influence measurements in our experiment.

Table 2: Factors that may influence measurements

Memory allocation strategies and caching effects
Heating of processors and cooling effects
Background processes
Network activities
Compiler optimising effects

The memory allocation strategies applied by different operating systems may differ from one another and may also be adapted to best suit the current hardware configuration; they may have marked impact on the performance of processes. Sometimes the strategy is dynamically adapted to the memory access patterns. Very often background processes of which the user may be unaware may be active; these include processes such as anti-virus software, automatic software updates, backup procedures, and indexing processes. Controlling such activities is further complicated by the fact that many of these activities may be under the control of a network server.

In accordance with guidelines by Pieterse and Flater [12] it is important to describe how the influence of uncontrollable factors was minimised.

To eliminate some of the factors that are often hard to control, we executed our experiment on a computer that was specifically prepared for this purpose to ensure that the conditions in which the measurements were taken were as consistent as possible over all experimental runs.

To minimise possible effects that can be attributed to background processes and network activities, we decided to run our benchmarks on an isolated machine with minimal software installed. Software that is not crucial for the operation of the computer and not needed to perform our benchmarks was not installed. The machine was configured to boot up on the command line without network connection. Our experiments were executed only after a cold boot of the machine.

To protect against further inadvertent introduction of systematic, uncontrollable effects that may be caused by the factors such as caching and cooling procedures performed by the operating system, or other factors that we are not aware of, we decided to randomise the order in which we applied our treatments. To achieve this, we created five experimental runs. Each experimental run is a list of all the treatments in our experiment where each treatment appears once and in which the order of the treatments is randomised. By gathering the data for all treatments in each experimental run, we compiled five repetitions of each treatment for which each has a randomised chance of the effects of the uncontrolled factors.

3 Profiling

3.1 Data and tools

The data, scripts and programs used to gather the measurements, are stored in the tarball called `TC_Benchmarking.tgz` that accompanies this document. These are organized as shown in the following diagram.

```
|__ Data
|   |__ InputFiles
|   |__ BoostPlain
|   |__ StdPlain
|   |__ BoostShort
|   |__ StdShort
|__ Measurements
|   |__ Time
|   |__ Memory
|__ Tools
```

3.1.1 Data directory

The given scripts use the data in the `Data` directory. For these scripts to execute successfully the `Data` directory should contain the algorithms of which the performance needs to be measured as well as test data to be used by these algorithms when performing their tasks. In the given tarball the `InputFiles` directory is empty. The scripts to perform the profiling jobs expect appropriately named data files in this directory. The attributes of the data files that we used are described in more detail in Section 3.3. These files can be created manually or generated using the program discussed in Section 3.3. The remaining four directories each contains source code of a number of algorithms. All the TC Algorithms that were profiled are included in the given tarball. These are discussed in more detail in Section 3.2.

3.1.2 Tools directory

The `Tools` directory contains scripts and programs that are used to generate data, to perform the measurements and to analyse memory statistics. A program to generate input data is discussed in Section 3.3. The scripts that perform the profiling jobs to measure time and memory are respectively discussed in Section 4.2 and 5.2. The tools to extract the required data from the files that is produced when running Valgrind are discussed in Section 5.4.

3.1.3 Measurements directory

The `Measurements` directory as two subdirectories — one called `Time` for time measurements and one called `Memory` for memory measurements. Both these directories are empty. The given scripts produce appropriately named data files in these directories.

3.2 Algorithms

We have implemented ten different algorithms that calculate the transitive closure (TC) of a given binary relation. They are described in Tables 3 and 4.

We have assigned a four-letter code to identify each algorithm. These codes are used to refer to the algorithms when we discuss them and also in naming the files containing their implementations. The algorithms are classified in two classes according to the fundamental method they use to construct the transitive closure of a given relation $R \subseteq U \times U$.

The algorithms classified as *coat algorithms* start with R and systematically add edges to this relation until a stage is reached when the resulting relation is the transitive closure of the original relation R. We call this approach *coat* because the given relation is coated with the necessary additional edges to turn it into the smallest possible transitive relation containing R. The edges that have to be added in each iteration are determined using multiplications of the adjacency matrix.

The algorithms classified as *grow algorithms* use a function $f : (\mathcal{P}(U)) \mapsto (\mathcal{P}(U \times U))$ that is defined such that $f.\varnothing = R$ and $f.U = R^+$. This function should also be defined such that the value of $f.(A \cup \{u\})$ can be calculated using the value of $f.A$. The algorithm starts with $A = \varnothing$ and systematically adds elements of U to A. After adding an element u to A, the value of $f.(A \cup \{u\})$ is calculated using the value of $f.A$. The algorithm continues to add elements to A and to recalculate $f.A$ until the value of $f.U$, and consequently R^+, is found. We call this approach *grow* because the argument of the function is grown by adding additional elements until this argument contains all of U. The inner loop of these algorithms typically determines an updated version of the adjacency matrix through boolean addition of two rows in the matrix.

Table 3: Coat algorithms

Code	**Name**	**Description**
Fuse	Fused Coat	An optimisation of Prosser's algorithm: A loop that multiplies two matrices and a loop that adds two matrices are fused.
Moni	Monitored Coat	Apply repeated matrix multiplication and use a change monitor to know when to stop.
Neat	Neat Coat	An optimisation of the monitored coat algorithm: A loop that multiplies two matrices and a loop that adds two matrices are fused.
Pros	Prosser [14]	Apply repeated matrix multiplication and stop after $n-1$ iterations.

The algorithms operate on two-dimensional Boolean matrices. A practical way to implement an $n \times n$ Boolean matrix in C++ is by defining a vector of n bit-vectors. There are a variety of bit-vector implementations that are suitable for this purpose such as `std::vector<bool>` from the C++ Standard Template Library (STL), from the Boost library [15]; `boost::dynamic_bitset`, `Qt::QBitArray` [6], BitMagic's `bm::bvector<>` [8], `Bit::Vector` by Beyer [4], Dippenstein's [7] `Bitarray`, and `CBitArray` by Mostafa [10].

Pieterse *et al.* [13] found that `boost::dynamic_bitset`'s memory performance ranks high in comparison with other implementations. Furthermore, because it capitalises on the operations that are defined in `std::bitset` to implement its bit-wise operations, its time performance ranks the highest of these implementations. It was also observed that the performance of `std::vector<bool>` is unacceptably slow. Here we experiment with two options, namely a standard character array and `boost::dynamic_bitset`. The latter because the implementations in the `boost` libraries are widely accepted as being efficient and the former because it allows for accurate implementations without the need for additional language libraries. When using a character array to store bit-vector, the data is simply stored as a character string of 0's and 1's. The `boost::dynamic_bitset` stores the data in raw format as Booleans.

For each of the algorithms, four different versions were implemented. For each of the above mentioned two data representation models, the algorithm was implemented in one of two ways. One with the application of short circuiting and the other without. The implementations are labeled 'Plain' or 'Short' to indicate if the algorithm is the original implementations (plain) or an implementation that applies short-circuiting (short). They are also labeled 'Boost' or 'Std' to indicate if the algorithm uses a character array for data representation (std)

Table 4: Grow algorithms

Code	**Name**	**Description**
Bakr	Baker [3]	Iterates through the entries in the matrix in row order and uses a change monitor to know when to stop.
BlkC	Blocked Column [2, 1]	A variation of Warshall's algorithm: Columns are processed in blocks while applying loop tiling to minimise cache thrashing. The elements within the block and within the square section around the diagonal are processed in column order. After processing the elements in the square section, the rest of the elements within a block are processed in row order.
BlkR	Blocked Row [2, 1]	A variation of Warren's algorithm: Rows are processed in blocks while applying loop tiling to minimise cache thrashing. The elements within a block are processed in column order.
Mart	Martynyuk [9]	Iterates through the entries in the matrix in row order and stop after $log_2 n$ iterations.
Wrrn	Warren [17]	Iterates through the entries in the matrix in row order. Process entries below the main diagonal in the first pass and those above in the second pass. It calculates the transitive closure in two passes.
Wrsh	Warshall [18]	Iterates through the entries in the matrix in column order. It calculates the transitive closure in one pass.

or uses a dynamic bitset as implemented in the Boost library (Boost) for data representation. The source code of each of the types of implementation is stored in its own appropriately named directory.

The directory for each of the classes of algorithms has four subdirectories. These directories are shown in the following diagram for `ShortBoost`. Each of the other three directories has the same detail for that particular class of implementations of the algorithm.

```
Data
 └── ShortBoost
      ├── Memory
      │    ├── Code
      │    └── execute
      └── Time
           ├── Code
           └── execute
```

The source code is treated differently for the different profiling jobs. For this reason separate implementations of the algorithms are stored for each type of profiling job.

- The `Code` directories holds the source code for the implementations of the algorithms that are used for the specific profiling job. For time profiling, these files contains the instrumentation to output time stamps at specified points in the execution of the program. For memory profiling these files do not contain instrumentation.
- Each `Code` directory, contains a `makefile`. The implementations need to be compiled in a specified way for each of the profiling jobs. The given makefile creates an executable for each of the algorithms stored in this directory with the appropriate compiler flags for the specific profiling job and move them to the `execute` directory.
- The `execute` folders are empty at first. The required executables are generated using the `makefile` in the `Code` directory.

3.3 Data

The algorithms operate on binary relations. A relation can be represented as a directed graph where the nodes are the elements of the domain and an arc from node i to node j signifies that element i relates to element j. A two-dimensional Boolean matrix in which an on-bit in cell (i, j) represents an arc from node i to node j can also be used to represent such relation. This matrix is called the adjacency matrix of the relation. Here the relations are specified in terms of an adjacency matrices. We vary both the number of nodes—i.e. the number of elements in the relation—and the number of arches—i.e. the number of related element pairs in the relation.

We use the controlled variable called *size* to refer to the chosen dimension of the adjacency matrix. In the data we used, size has five states which vary from 0 to 1000 in steps of 200.

We use the controlled variable called *density* to refer to a measure of the number of arcs in the relation. A matrix with 100% density is the adjacency matrix of a relation where every node is connected to every other node in the graph representing the relation, whereas a matrix with 0% density represents a relation that has no arcs — i.e the empty relation. There are nine density values in the data we generated. These values range from 0% to 40% in steps of five.

The generated data sets should be saved in the `InputFiles` directory. In the profiling jobs reported in this document the data was generated once and reused in all the jobs. The following is a program that can be used to generate one data file with the specified attributes. We changed it to loop through the different required sizes and densities to create files with names that encode their attributes. The source code of this program is saved in the `Tools` directory in a file called `generateData.cpp`

```
#include <iostream>
#include <fstream>
#include <ctime> // initialise random generator
#include <cmath> // using sqrt()
#include <cstdlib> // generate random numbers
#include "boost/dynamic_bitset.hpp"

using namespace std;
using namespace boost;

/** Generate random arcs match to the specified density */
dynamic_bitset<> getMatrix(unsigned int dimension, short density)
{
        unsigned int onBits = 0;
        unsigned int numOfBits = dimension * dimension;
        unsigned int wantedBits = density * numOfBits / 100;
        srand( time( 0 ) );  // Initialize random number generator.
        const int BUCKET_SIZE = RAND_MAX / dimension;
                // determine number of buckets with equal change
                // of getting a number generated
        dynamic_bitset<> generatedBitset;
        generatedBitset.resize(numOfBits);

        int x, y, pos;

        while(onBits < wantedBits)
        {
                do
                {
                        x = rand() / BUCKET_SIZE;
                }
                while (x >= dimension);
                do
                {
                        y = rand() / BUCKET_SIZE;
                }
                while (y >= dimension);

                pos = dimension * x + y;
                if(!generatedBitset.test(pos))
                {
                        generatedBitset.set(pos);
                        ++onBits;
                }
        }
        return generatedBitset;
}

/** write data file */
void generate( dynamic_bitset<>  outputMatrix, char fileName[] )
{
        ofstream outfile;
        outfile.open (fileName);
        unsigned int dimension = sqrt(outputMatrix.size());
        outfile << dimension << ',' << outputMatrix << endl;
        outfile.close();
}
```

```
int main()
{
    int dimension; cout << "Dimension: ";
    cin >> dimension;
    short percentage; cout << "Percentage filled: ";
    cin >> percentage;
    char fileName[20]; cout << "File name: ";
    cin >> fileName;
    generate(getMatrix(dimension, percentage), fileName);
    return 0;
}
```

3.4 Treatments

We call one execution of one algorithm using one data file as input a treatment. A profiling job consists of many treatments that are formed by varying some factors. In our case the treatments were formed by varying three factors, namely algorithm, size and density.

The scripts that are used to execute a profiling job reads treatments from a treatment list. A treatment list should be stored in a text file. Treatment files are not provided. Each treatment list should contain all the combinations of treatments listed in a random order. In our case a typical treatment is the name of the algorithm followed by a data file name. The data file name contains the size and the density of the input data matrix. In our case this file is created using the following program. The source code is stored in a file called `Tools/makeTreatmentList.cpp`. The program produces the list on the standard output. When executing the program, one should pipe the output to the desired text file.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <sstream>

using namespace std;
int main(int, char*[]) {

    std::vector<std::string> cols
    = {"Bakr", "BlkC", "BlkR", "Fuse", "Mart", "Empt",
    "Moni", "Neat", "Pros", "Wrrn", "Wars"};

    std::vector<std::string> rows;
    stringstream convert;

    convert << "\t" << "A1" <<"_" << "0" << ".txt";
    rows.push_back(convert.str());
    convert.str("");

    convert << "\t" << "A1" <<"_" << "1" << ".txt";
    rows.push_back(convert.str());
    convert.str("");
    for( int density = 200; density <= 1000; density+=200 )
    {

        for( int size = 0; size <=40; size+=5 )
        {
          convert << "\t" << "A" << density <<"_" << size << ".txt";
          rows.push_back(convert.str());
          convert.str("");
        }
    }

    std::vector<std::string> vec;
    for (auto i : cols)
        for (auto j : rows)
            vec.emplace_back(i + j);
    std::random_shuffle(vec.begin(), vec.end());

    for (auto i : vec) std::cout << i << '\n';
    return 0;
}
```

4 Measuring elapsed time

4.1 Instrumentation for time measurements

To measure the time taken for an algorithm a program called getRealTime.cpp which was coded by David R. Nadeau [11]. The function was modified to include only the code to specific code which caters to the Linux configured machine. This function queries the operating system's time before and after the function that is measured is called. This is C++ code that is used in the source code to insert the instrumentation to output time stamps at the desired points during the execution. This code is saved in a file called `getRealTime.cpp`. A copy of this file is stored in each directory where the source code of the algorithms that are prepared for time profiling are stored.

```
/*
 * Author:  David Robert Nadeau
 * Site:    http://NadeauSoftware.com/
 * License: Creative Commons Attribution 3.0 Unported License
 *          http://creativecommons.org/licenses/by/3.0/deed.en_US
 */

#include <time.h> /* clock_gettime(), time() */

/**
 * Returns the real time, in seconds, or -1.0 if an error occurred.
 *
 * Time is measured since an arbitrary and OS-dependent start time.
 * The returned real time is only useful for computing an elapsed
 * time between two calls to this function.
 */
```

```
double getRealTime( )
{
  /* POSIX. ---------------------------------------------- */
        struct timespec ts;
        const clockid_t id = CLOCK_MONOTONIC_RAW;

        if ( id != (clockid_t)-1 && clock_gettime( id, &ts ) != -1 )
            return (double)ts.tv_sec +
                                  (double)ts.tv_nsec / 1000000000.0;
}
```

4.2 Gathering measurements

We execute a script called `gather.sh` to execute a profiling job and produce a text file containing the measurements. The script is executed with a treatment list as input. The treatment list is explained in Section 3.4. The script runs each treatment written in this file. It uses the executables and data files that are respectively stored beforehand in the appropriate `execute` and `InputFile` directories.

The script takes four parameters: `$1` is the name of the treatment file. `$2` is the algorithm class, `$3` is the name of the file where the measurements should be saved and `$4` and is the number of treatments. In our case the algorithm class name is used to name the directory in which the algorithms are stored and also to name the executable files. The start time and end time of each treatment is dumped in a file specified in `$3`.

```
#!/bin/bash
#   Run this script with FOUR parameters
#   1 = Treatment list
#   2 = Algorithm class
#   3 = output file name
#   4 = number of treatments

t1=$(date +%s.%N)
 i=0
 while read  alg[i] dat[i]
 do
    ((i++))
 done < $1

  a=0
  while((a < $4));do
        echo  "${alg[a]}_${dat[a]}"
        ./../../Data/$2/Time/executables/"${alg[a]}"$2.out
            < ../../Data/InputFiles/"${dat[a]}".txt
           >> Measurements/Time/$3
        echo -e "${alg[a]}_${dat[a]}" >> Measurements/Time/$3
        a=`expr $a+1`
  done

 t2=$(date +%s.%N)
   echo -e "${t1}_\t${t2}_\t$2" >> Measurements/Time/$3
```

4.3 Running the profiling job to measure time

Here are the steps to follow to perform a profiling job to measure time

Step 1: Place data files for the required treatments in the `InputFiles` directory. Either create them manually or generate them using a program such as the programs discussed in Section 3.3.

Step 2: Compile and run `makeTreatmentList.cpp` and pipe the output to a file and save it in the`Tools/Time` directory.

Step 3: Compile the algorithms for the job using their makefile.

Step 4: Execute the `gather.sh` script with the appropriate parameters.

5 Memory profiling

5.1 Memory profiling tool

To get the memory used by all the algorithms a memory profiling tool called Valgrind[1] was used. Valgrind is an instrumentation framework for building dynamic analysis tools. In this project it was used as a memory profiling tool. The version used is 3.9.0 for AMD64/Linux systems. Valgrind was used with options:

- Stack was disabled and valgrind run in heap profiling mode.
- Max frequency was set to 100
- Massif:a heap profiler It measures how much heap memory your program uses. This includes both the useful space, and the extra bytes allocated for book-keeping and alignment purposes. It can also measure the size of your program's stack(s) although this option is not included in this profiling job.
- massif-out-file Specifies that the output should be piped to an output file. with the name of the file being named as the treatment.

5.2 Gathering measurements

`profile.sh` is an example script that executes the treatments specified in `ChunkMem.txt`. This script invokes valgrind with the options defined to profile the memory used by each treatment. For every treatment the script creates a file which is named according to the treatment. The memory statistics of the treatment is written on the file. These files are stored in a sub-directory called `memStats`. This script is stored in `Tools/profile.sh`.

[1]http://valgrind.org/

```
#!/bin/bash
i=0
x=2
exp="memStats"
freq=10
max=100
stacks="no"

while read dat1[i] dat2[i]
do
   valgrind --tool=massif
  --massif-out-file=$exp_${dat1[i]}_${dat2[i]}_$i
  --detailed-freq=$freq
  --max-snapshots=$max --"time"-unit=B
  --stacks=$stacks ./${dat1[i]} < ${dat2[i]}
    mv $exp_${dat1[i]}_${dat2[i]}_$i $exp
    i=`expr $i + 1`
done < ChunkMem.txt
```

5.3 Running the profiling job to measure memory

Here are the steps to follow to perform a profiling job to measure memory

Step 1: Place data files for the required treatments in the `InputFiles` directory. Either create them manually or generate them using a program such as the programs discussed in Section 3.3.

Step 2: Compile and run `makeTreatmentList.cpp` and pipe the output to a file and save it in the `Tools` directory.

Step 3: Compile the algorithms for the job using their makefile. Compilation is done with the -g flag for debug mode. This is required for Valgrind to be able to observe the program. Code is compiled with -O1 optimisation, as the use of -O2 and above is not recommended as Memcheck occasionally reports uninitialised-value errors which don't really exist.

Step 4: Execute a script like `profile.sh` in which the source files and the target files are correctly specified.

5.4 Analyzing the memory statistics files

The script described in Section 5.2 stores the memory statistics of all the treatments in a directory. The `memoryAnalysis.cpp` program has to be used in conjunction with the `CalculatePeak.sh` script to produce a file containing only the treatment detail and the measurement for the specific treatment.

5.4.1 CalculatePeak.sh

The script prepares a memory statistics file. It removes redundant text to simplify the processing by the next program. Only the lines of statistics beginning with memory heap is retained for all the snapshots. This script is saved in `Tools/CalculatePeak.sh`

```
#!/bin/bash

while read  dat[i] dat2[i]
do
  grep -n mem_heap_B ${dat[i]}_${dat2[i]} | cut -d : -f 1 > line.txt
  while read  line
   do
     position1=$line
     position2=`expr $position1 + 1 `
         sed -n $position1'p' ${dat[i]}_${dat2[i]} >> temp.txt
     sed -n $position2'p' ${dat[i]}_${dat2[i]} >> temp.txt
  done < line.txt
  cp temp.txt ${dat[i]}_${dat2[i]}
  rm temp.txt line.txt
  ((i++))
done < outputFiles.txt
```

5.4.2 memoryAnalysis.cpp

The code reads and store the file name and calculate the highest sum of the memory heap and write the output into the file called memoryProfile.txt. This source code is saved in `Tools/memoryAnalysis.cpp`

```
#include <iostream>
#include <fstream>
#include <string>
#include <string.h>
#include <stdlib.h>
#include <iomanip>
#include <sstream>
#include <algorithm>
//@author  Jan Fenyane
using namespace std;
int number = 0;

string parse_memory(string);
string parse_data(string);

int main()
{
   ifstream filelist;
   ofstream outputFile;
   string string1,string2;

   filelist.open("outputFiles.txt");
   outputFile.open("memoryProfile.txt");

   outputFile << "ALGORITMS"
      << setw(12) << "DATA_SET"
      << setw(12) << "SIZE"
      << setw(12) << "DENSITY"
      << setw(12) << "MEMORY" <<endl;
   if(  filelist.is_open() )
   {
      char openfile[256];
      string filename;
        while(filelist.getline(openfile,256) )
      {
         if(  filelist.eof() ) break;
         filename.append(openfile);
         string1 = parse_data(filename);
         replace(filename.begin(),filename.end(),' ','_');
         string2 = parse_memory(filename);
         outputFile << string1 << string2 << endl;
         filename.clear();
        }
   }
   else
   {
      cout << "Couldn't open file / File doesnt exist" << endl;
   }
   filelist.close();
   outputFile.close();
   return 0;
}

string parse_data(string parse)
{
        stringstream returnValue;

        size_t position1 = parse.find(' ');
        size_t position2 = parse.find('_');
        size_t position3 = parse.find('.');

        string algoritm = parse.substr(0,position1);
        string dataset = parse.substr(position1+1,1);
        string size = parse.substr(position1+2,
                (position2-(position1+2)));
        string density = parse.substr(position2+1,
                (position3-(position2+1)));

        returnValue << algoritm
                    << setw(12) << dataset
                    << setw(12) << size
                    << setw(12) << density;

        return returnValue.str();
}
```

```
string parse_memory(string parse)
{
    stringstream returnValue;
    ifstream inputFile;
    string string1, string2, string3;
    double max=0;
    char readData[256];
    int i=1;
    inputFile.open(parse);

    if( inputFile.is_open() )
    {
        while(inputFile.getline(readData,256) )
        {
            if( inputFile.eof() ) break;
            string1.clear();
            string1.append(readData);
            if( i % 2 == 0)
            {
                size_t position1 = string1.find('=');
                size_t position2 = string2.find('=');
                double mem1=atof(string1.substr(position1+1,
                                string1.length()).c_str());
                double mem2=atof(string2.substr(position2+1,
                         string2.length()).c_str());
                double total=mem1+mem2;
                max = ( max >= total ? max : total);
            }
            else
            {
                string2 = string1;
            }
            i++;
            memset(readData,0,sizeof(readData));
        }
    }
    else
    {
        cout << "*Couldn't open file / File doesnt exist" << endl;
        cout << parse << endl;
    }
    returnValue << setw(12) << max;
    return returnValue.str();
}
```

References

[1] Agrawal R, Dar S and Jagadish HV (1990) Direct transitive closure algorithms: design and performance evaluation, *ACM Transactions on Database Systems (TODS)*, 15(3):427–458.

[2] Agrawal R and Jagadish HV (1987) Direct Algorithms for Computing the Transitive Closure of Database Relations, in: *Proceedings of the*

13th International Conference on Very Large Data Bases, VLDB '87, 255–266, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., URL `http://0-dl.acm.org.innopac.up.ac.za/citation.cfm?id=645914.671624`.

[3] Baker JJ (1962) A note on multiplying Boolean matrices, *Communications of the ACM*, 5(2):102.

[4] Beyer S (2009) Bit::Vector, `http://guest.engelschall.com/ sb/download/Bit-Vector/`. [Online; accessed 2014-08-23].

[5] Blackburn SM, Diwan A, Hauswirth M, Sweeney PF, Amaral JN, Babka V, Binder W, Brecht T, Bulej L, Eeckhout L, Fischmeister S, Frampton D, Garner R, Georges A, Hendren LJ, Hind M, Hosking AL, Jones R, Kalibera T, Moret P, Nystrom N, Pankratius V and Tuma P (2012) Evaluate Collaboratory Technical Report #1: Can you trust your experimental results? `http://evaluate.inf.usi.ch/technical-reports/1`.

[6] Digia (n.d.) Qt - A cross-platform application and UI framework, `http://qt.digia.com/`. [Online; accessed 2014-08-23].

[7] Dipperstein M (2008) ANSI C and C++ Bit Manipulation Libraries, `http://michael.dipperstein.com/bitlibs/`. [Online; accessed 2014-08-23].

[8] Kuznetsov A, Shemanarev M, Tolstoy I, Lewis E and Khovayko O (n.d.) BitMagic, `http://bmagic.sourceforge.net/`. [Online: Accessed 2004-08-23].

[9] Martynyuk V (1963) The economical construction of a transitive closure of a binary relation, *USSR Computational Mathematics and Mathematical Physics*, 2(4):817 – 821, URL `http://www.sciencedirect.com/science/article/pii/0041555363905469`.

[10] Mostafa H (2004) Bits Array Encapsulation, `http://www.codeproject.com/Articles/9034/Bits-Array-Encapsulation`. [Online; accessed 2014-08-23].

[11] Nadeau D (2014) C/C++ tip How to measure elapsed real time for benchmarking, `http://nadeausoftware.com`.

[12] Pieterse V and Flater D (2014) *The ghost in the machine: don't let it haunt your software performance measurements*, Tech. Rep. NIST TN 1830, National Institute of Standards and technology, 100 Bureau Drive, Gaithersburg, MD 20899, URL `http://dx.doi.org/10.6028/NIST.TN.1830`.

[13] Pieterse V, Kourie DG, Cleophas L and Watson BW (2010) Performance of C++ Bit-vector Implementations, in: *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '10, 242–250, New York, NY, USA: ACM.

[14] Prosser RT (1959) Applications of Boolean matrices to the analysis of flow diagrams, in: *Proceedings of the Eastern Joint Computer Conference No. 16*, 133–138.

[15] Rivera R (n.d.) Boost C++ Libraries, `http://www.boost.org/`. [Online; accessed 2014-08-11].

[16] Schatzoff M (1981) Design of Experiments in Computer Performance Evaluation, *IBM Journal of Research and Development*, 25(6):848–859.

[17] Warren HS Jr (1975) A modification of Warshall's algorithm for the transitive closure of binary relations, *Communications of the ACM*, 18(4):218 – 220.

[18] Warshall S (1962) A Theorem on Boolean Matrices, *Journal of the ACM*, 9(1):11–12.