CS840 Project 4:

Cyclomatic Complexity

In C++ Programs

Samuel Gluss

Jozo Dujmovic

4-1-2016

Abstract – Cyclomatic complexity for a program is defined as the number of logic conditions in a program plus one. Cyclomatic complexity is directly correlated to cognitive complexity, which means it is an effective barometer for how challenging a piece of code is to understand. Cognitive complexity is not always obvious to determine, as it depends on the experience and ability of the developer, as well as how abstruse the code is. However, because cyclomatic complexity is relatively easy to determine objectively, it is possible to use it as a tool to compare programs which are similar in terms of their conceptual difficulty level. The resulting value returned can give some insight into how difficult the code may be to follow through its various control statements. A program with no loops or controls is a simple vertical path from top to bottom, whereas a program with many loops and controls may take a convoluted path which is confusing to the observer, having as they do only five or so registers in their brains.

# I. Introduction:

This experiment will be conducted by examining seven C++ programs, five of which are from a programming languages class assignment, and two of which are benchmarks from a previous project. These programs should give a good representation of the average C++ module, although none of them implement object oriented principles beyond utilizing a random generator.

The parser is a Python script which loads a programs source code into memory, then strips out all non-normal code such as comments, preprocessor directives, strings and characters. It does this with a state machine that only adds code to the output if it is deemed to be in a ‘normal’ code segment. This ensures that none of these are accidentally matched by the regexes which are then used to match any for and while loops, if statements, and case switches. Inside each match for an if, for, or loop condition, the match.count() function is used to find the number of logical and (&&) and logical or (||) present in each of these bodies. Once the entire source file has been parsed, these values are added up to obtain the cyclomatic complexity, defined as the number of condition bodies, plus the number of Boolean operators within these bodies.

Contents

[I. Introduction: 1](#_Toc447395786)

[II. Methodology 3](#_Toc447395787)

[III. Cyclomatic Complexity of Human Produced C++ Programs 5](#_Toc447395788)

[Appendix – Data 7](#_Toc447395789)

[Appendix – Code 8](#_Toc447395790)

[Python Scripts 8](#_Toc447395791)

[MatLab Scripts 13](#_Toc447395792)

[Analyzed C++ Files 16](#_Toc447395793)

[Citations 19](#_Toc447395794)

# II. Methodology

Once cppParse.py is finished executing, it will have populated a comma-separated outfile in matrix format. Each row in the matrix represents a single set of counts for a sourcefile. Each column is one of the counters, in the following output format:

File#, commas in variable init, equals in variable init, ifCount, forCount, whileCount, semiCount, switchCount, caseCount, ANDCount, ORCount, PLOC, LLOC, cyclomaticComplexity

These values are required to compute LLOC and V (cyclomatic complexity) for each sourcefile. LLOC is defined as the sum of the number of for/while loops, if statements, semicolons, switches, and function definitions. Furthermore, LLOC is incremented for each assignment and addition variables intitialized in any variable instantiation statement.

Example: int x; // LLOC = 1

int x,y,z; // LLOC = 3

int x = 2; // LLOC = 2

These values are read into a MatLab dataset object array, which allows indexed access to any counter value for each of the source files.

An example of cyclomatic complexity calculation is below, for source code which calculates Parabolic Approximation.

// ConsoleApplication1.cpp : Defines the entry point for the console application.

//

#include <iostream>

void print(int a, int b, int c, int x1, int x3)

{

}

int determinant(int a, int b, int c, int d, int e, int f, int g, int h, int i)

{

int retVal = a \* (e \* i - f \* h) - b \* (d \* i - f \* g) + c \* (d \* h - e \* g);

return retVal;

}

void calculateCoefficientsCramers(int x1, int y1, int x2, int y2, int x3, int y3)

{

//use cramers rule to calculate coefficients for quadratic function

//calculate determinant of coefficient matrix (D) first

int det = determinant(x1\*x1, x1, 1, x2\*x2, x2, 1, x3\*x3, x3, 1);

Cyclomatic complexity is incremented once for the if condition

if (det == 0)

{

std::cout << "determinant of coefficient matrix is zero, Cramers rule is not applicable";

return;

}

//calculate coefficients for y = ax^2 + bx + c

// a = Da / D

int a = determinant(y1, x1, 1, y2, x2, 1, y3, x3, 1) / det;

// b = Db / D

int b = determinant(x1\*x1, y1, 1, x2\*x2, y2, 1, x3\*x3, y3, 1) / det;

// c = Dc / D

int c = determinant(x1\*x1, x1, y1, x2\*x2, x2, y2, x3\*x3, x3, y3) / det;

}

void calculateCoefficientsLagrange(int x1, int y1, int x2, int y2, int x3, int y3)

{

//calculate

double denom1 = (x1 - x2)\*(x1 - x3);

double denom2 = (x2 - x1)\*(x2 - x3);

double denom3 = (x3 - x1)\*(x3 - x2);

double a = (y1 / denom1) + (y2 / denom2) + (y3 / denom3);

double b = -(((y1 \* (x2 + x3)) / denom1) + ((y2 \* (x1 + x3)) / denom2) + ((y3 \* (x1 + x2)) / denom3));

double c = ((y1 \* x2 \* x3) / denom1) + ((y2 \* x1 \* x3) / denom2) + ((y3 \* x1 \* x2) / denom3);

}

int main()

{

//test points

int x1 = 1;

int y1 = 9;

int x2 = 2;

int y2 = 23;

int x3 = 3;

int y3 = 45;

Cyclomatic complexity is incremented once for the if condition, and twice for the logical AND operators in the statement below

if (x1 != x2 && x2 != x3 && x1 != x3)

{

calculateCoefficientsCramers(x1, y1, x2, y2, x3, y3);

calculateCoefficientsLagrange(x1, y1, x2, y2, x3, y3);

}

else

{

std::cout << "All x values must be unique" << std::endl;

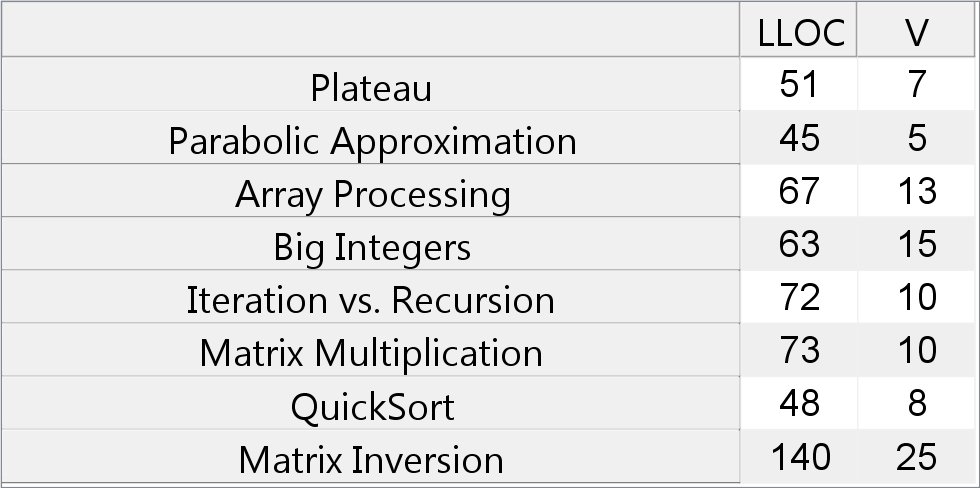
}

return 0;

}

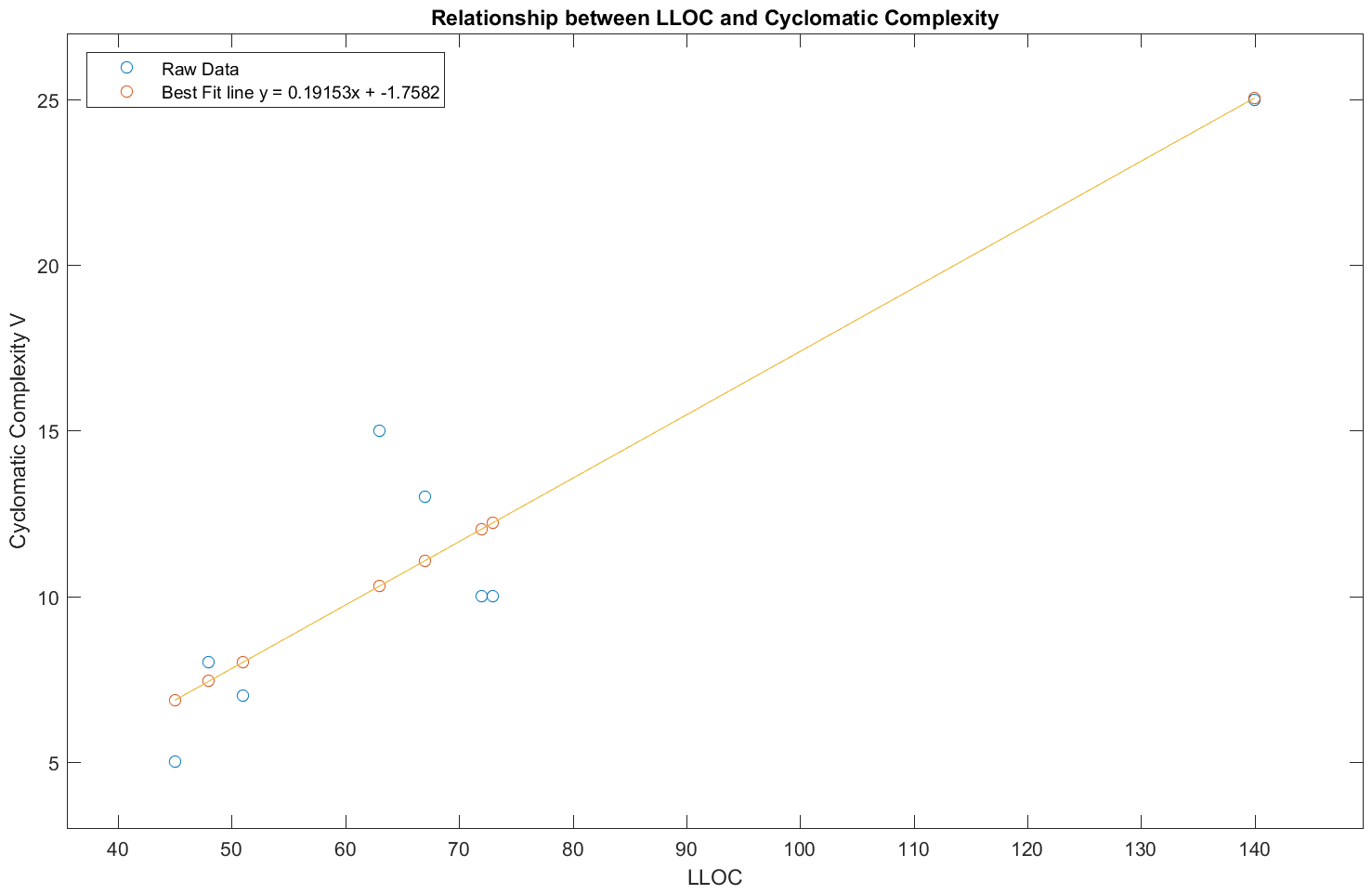
Cyclomatic complexity is incremented for two if conditions, and for two logical AND operators that are within conditions. Plus one for turning the program into a closed graph equals 5.

The data for all of the source files is as follows:



These programs are all approximately the same level of difficulty conceptually. This indicates that the Cyclomatic complexity should be related in some way to the number of logical lines of code. The next step in the analysis will be to plot these points and determine if a linear relationship could exist between LLOC and V.

# III. Cyclomatic Complexity of Human Produced C++ Programs



In this data plot the data points for V/LLOC are visualized, as well as a best fit line for this data. The best fit line seems to match the points reasonably well, with a reasonable distribution on each side. At least some variation is to be expected, as some programs will have a dramatically larger number of conditional branches than others (e.g. a state machine vs. a matrix multiplication program).

On average though, the cyclomatic complexity increases by about 1 for every 5 LLOC. That is to say, out of any five lines of executable code, one will have a conditional branch on average.

An element of this plot that arouses some suspicion the initial value of -1.76, which suggests that an empty program has a cyclomatic complexity which is negative. This is impossible however, as determined by the alternative formula for cyclomatic complexity:

From this, it’s possible to determine that an empty program with no edges, nodes, or connected components should have a cyclomatic complexity V = 1. This could be caused by the presence of the matrix inversion program, which has a high proportion of conditional bodies, and has a large LLOC. This, combined with the comparatively tight clustering of the other data points, could cause the intercept to be thrown off. The inclusion of additional source code files with larger LLOC counts would help attenuate this type of error.

It does seem that a linear model could be fit to the LLOC/V data, from which it follows that V must be directly dependent on LLOC. This is by no means a rule, as the type of program also makes

# Appendix – Data

Outfile.txt

1,0,9,3,3,0,34,0,0,0,0,55,51,7

2,0,17,2,0,0,24,0,0,2,0,42,45,5

3,4,13,7,5,0,37,0,0,0,0,60,67,13

4,0,4,0,2,2,51,1,10,0,0,76,63,15

5,0,13,5,3,1,47,0,0,0,0,95,72,10

8,10,16,8,16,0,89,0,0,0,0,98,140,25

6,1,14,1,8,0,48,0,0,0,0,53,73,10

7,2,7,3,1,3,31,0,0,0,0,51,48,8

Format:

File#, commas in variable init, equals in variable init, ifCount, forCount, whileCount, semiCount, switchCount, caseCount, ANDCount, ORCount, PLOC, LLOC, cyclomaticComplexity

# Appendix – Code

## Python Scripts

**cppParse.py**

Reads all C++ files in a target directory. For each file, it uses a state machine to sanitize the code of any non-normal code such as comments, strings, characters, and preprocessor directives. It then uses regular expressions to count the occurrences of if, for, and while loop condition bodies. It also counts semicolons, switch statements, case checks, and function definitions.

The program then matches all control bodies, and matches any occurrences of logical AND and OR operators that appear within them. It does the same for commas and assignment operators within variable initializations, which counts multiple initialization and initialization-assignment.

These various counters are then aggregated appropriately to acquire the LLOC (Logical Lines of Code) as well as V, the cyclomatic complexity of the program. These values are joined with commas, then output to the end of the output file with a terminating newline. This way, each program gets its own line of values in the output file. This is done for each source file until the output file is a comma separated matrix of counter values.

**Source Code:**

import re,glob

import pdb; pdb.set\_trace()

from enum import Enum

outputFilename = 'outfile.txt'

cppTypes = '(long\slong|long\sdouble|long\sint|char|bool|short|int|long|float|double|time\_point)'

controls = '(for|while|if)'

class State(Enum):

normal = 0

startComment =1

string=2

preprocessing=3

char=4

lineComment=5

blockComment=6

blockCommentEnd=7

specialString=8

specialChar=9

# function which eliminates any comments/preprocessor/other nonlogical lines

def getNormalCode(source):

retVal = []

state = State.normal

for character in source:

if state == State.normal:

if character == '/':

state = State.startComment

elif character == '"':

state = State.string

elif character == '#':

state = State.preprocessing

elif character == '\'':

state = State.char

else:

state = State.normal

elif state == State.preprocessing:

if character == '\n':

state = State.normal

else:

state = State.preprocessing

elif state == State.startComment:

if character == '/':

state = State.lineComment

elif character == '\*':

state = State.blockComment

else:

state = State.normal

elif state == State.lineComment:

if character == '\n':

state = State.normal

else:

state = State.lineComment

elif state == State.blockComment:

if character == '\*':

state = State.blockCommentEnd

else:

state = State.blockComment

elif state == State.blockCommentEnd:

if character == '/':

state = State.normal

elif character == '\*':

state = State.blockCommentEnd

else:

state = State.blockComment

elif state == State.string:

if character == '\\':

state = State.specialString

elif character == '"':

state = State.normal

else:

state = State.string

elif state == State.specialString:

state = State.string

elif state == State.char:

if character == '\\':

state = State.specialChar

elif character == '\'':

state = State.normal

else:

state = State.char

elif state == State.specialChar:

state = State.char

if state == State.normal:

retVal.append(character)

return(''.join(retVal))

# function which cleans contents of brackets for certain counting ops

def stripBrackets(source):

retVal = []

bracketLevel = 0

for character in source:

if character == '{' or character == '[':

bracketLevel += 1

elif character == '}' or character == ']':

bracketLevel -= 1

elif bracketLevel == 0:

retVal.append(character)

return(''.join(retVal))

def stripParen(source):

retVal = []

parenLevel = 0

for character in source:

if character == '(':

parenLevel += 1

elif character == ')':

parenLevel -= 1

elif parenLevel == 0:

retVal.append(character)

return(''.join(retVal))

def getCounts(filepath):

with open(filepath,'r') as f:

code = f.read()

#semiCount = code.count(';')

# determine PLOC by finding all nonempty lines

# lines must have something other than space, or curly braces

newLineCount = len(re.findall('[a-zA-Z0-9]+[^\n]\*\n',code))

#blankLinesCount = len(re.findall('\n[\s\{\}]\*?\n',code))

PLOC = newLineCount# - blankLinesCount

# eliminate preprocessing, comments, strings, and chars

code = getNormalCode(code)

ifCount = len(re.findall('[^a-zA-Z0-9]if[\s\(]',code))

forCount = len(re.findall('[^a-zA-Z0-9]for[\s\(]',code))

whileCount = len(re.findall('[^a-zA-Z0-9]while[\s\(]',code))

semiCount = len(re.findall('[^;\n];',code))

switchCount = len(re.findall('[^a-zA-Z0-9]switch[\s\(]',code))

caseCount = len(re.findall('[^a-zA-Z0-9]case[^:]\*:',code))

funcs = re.findall('(' + cppTypes+'\s+[a-zA-Z0-9]+[\(]{1}[^\)]\*[\)]{1}[\n\s]\*[\{]{1})',code)

funcCount = len(funcs)

# only get logical AND OR inside control statements, switch cases

matches = re.findall('(' + controls+'\s\*\([^\)]+\))',code)

logANDCount = 0

logORCount = 0

for match in matches:

logANDCount += match[0].count('&&')

logORCount += match[0].count('||')

# TODO: something is broken with this regex for human code

# get all lines where variables are defined or initialized

matches = re.findall('[^a-zA-Z0-9]+('+cppTypes+'[\\*]\*\s+[^;\{\}]+;)',code)

commas = 0

equals = 0

for match in matches:

# strip brackets and parens so contents aren't included

match = stripBrackets(match[0])

match = stripParen(match)

commas += match.count(',')

equals += match.count('=')

# LLOC calculation

LLOC = equals + commas + semiCount + ifCount + forCount + whileCount + switchCount + funcCount

cyclomaticComplexity = ifCount + whileCount + forCount + caseCount + logANDCount + logORCount + 1

# debug print

#print('program code after cleaning:\n')

#print(code)

#print

# aggregate output string

outStr = ''

outStr += str(commas) + ','

outStr += str(equals) + ','

outStr += str(ifCount)+ ','

outStr += str(forCount)+ ','

outStr += str(whileCount)+ ','

outStr += str(semiCount)+ ','

outStr += str(switchCount)+ ','

outStr += str(caseCount)+ ','

outStr += str(logANDCount)+ ','

outStr += str(logORCount)+ ','

outStr += str(PLOC)+ ','

outStr += str(LLOC)+ ','

outStr += str(cyclomaticComplexity) + '\n'

return outStr

# get list of cpp files

cppFiles = []

for file in glob.glob("cppFiles/\*.cpp"):

cppFiles.append(file)

# clear outfile

open(outputFilename, 'w').close()

for file in cppFiles:

with open(outputFilename, "a") as outfile:

# The below line of code seeks to the end of the outfile

outfile.seek( 0, 2 )

# get benchmark number

fileNo = re.findall('([0-9]+)\.cpp',file)

outfile.write(fileNo[0] + ',')

outfile.write(getCounts(file))

**cropDataPlot.py** – now with edge-scanning, user definable padding, and out-of-bounds protection.

Processes each PNG image file in the directory. Scans each image to find the first non-white pixel coming from each of the cardinal directions. It then crops the image, removing enough whitespace to leave a user-defined padding area. If the user-defined padding would fall out of bounds of the image, the tool uses the image boundary for that side instead. This tool dramatically speeds up processing of MatLab generated figures for inclusion into technical documentation.

**Source Code:**

# Image Cropping Utility

# Samuel Gluss

# latest update 4-2-2016

# added edge detection and auto cropping with out-of-bounds protection

# Set padding around figure below:

padding = 10

import os,sys,glob,Image

#import pdb; pdb.set\_trace()

# threshold is the minimum average pixel value considered to be non-white

# findX is true if scanning for an x value, false for finding y values

def getEdge(image, thresh, oBeg, oEnd, iBeg, iEnd, findX):

inc = 1 if oBeg < oEnd else -1

# find edge

for i in range(oBeg,oEnd, inc):

for j in range(iBeg,iEnd):

# set pixel to correct coordinate based on scanning direction

pixel = image.getpixel((i,j)) if findX else image.getpixel((j,i))

# if a non-white pixel is found, return value

if (sum(pixel)/float(len(pixel))) < thresh:

return i

# find edges of image to crop

def findEdges(image, edges, imageWidth, imageHeight):

imageWidth -= 1

imageHeight -= 1

# maximum value considered to be non-white

threshold = 250

# scan right to get left edge

edges[0] = getEdge(image, threshold,0,imageWidth,0,imageHeight,True)

# scan left to get right edge

edges[1] = getEdge(image, threshold,imageWidth,0,0,imageHeight,True)

# scan down to get top edge

edges[2] = getEdge(image, threshold,0,imageHeight,0,imageWidth,False)

# scan up to get bottom edge

edges[3] = getEdge(image, threshold,imageHeight,0,0,imageWidth,False)

return edges

# load all files in current directory with .png extension

imageFiles = []

for file in glob.glob("\*.png"):

imageFiles.append(file)

# debug print

print imageFiles

print

# trim margins on each image

for image in imageFiles:

imageToResize = Image.open(image)

w, h = imageToResize.size

# edge left, right, top, bottom

edges = [0,0,0,0]

findEdges(imageToResize, edges, w, h)

lCrop = edges[0] - padding

rCrop = edges[1] + padding

tCrop = edges[2] - padding

bCrop = edges[3] + padding

# protect against out-of-bounds

lCrop = lCrop if lCrop > 0 else 0

rCrop = rCrop if rCrop < w else w

tCrop = tCrop if tCrop > 0 else 0

bCrop = bCrop if bCrop < h else h

imageToResize.crop((lCrop, tCrop, rCrop, bCrop)).save(image)

## MatLab Scripts

**p4analysis.m** – data processing driver file

Loads data set into object, which is then processed by subroutines to produce analysis and figures

**Source Code:**

% Samuel Gluss

% CS840 project 3 PLOC/LLOC and Cyclomatic Complexity

% 3-26-2016

% get working directory for project

wd = fileparts(mfilename('fullpath'));

wd = [wd '\..\'];

% get data from cppParse and BM outfiles

data = dlmread([wd 'outfile.txt']);

% sort rows of outfile by file number

data = sortrows(data);

% create dataClass array

dataClasses = BMDataSet.empty();

for i = 1:size(data,1)

dataClasses(i) = BMDataSet(data(i,:));

end

% project4 LLOC vs V

LLOCV([dataClasses.cyclomaticComplexity], [dataClasses.LLOC])

VLLOCTable([dataClasses.cyclomaticComplexity], [dataClasses.LLOC])

DataSet.m – a class with constructor to store python output

**Source Code:**

classdef DataSet

properties

runNum

commas

equals

ifCount

forCount

whileCount

semiCount

switchCount

caseCount

ANDCount

ORCount

PLOC

LLOC

cyclomaticComplexity

end

methods

% methods, including the constructor are defined in this block

function obj = DataSet(data)

obj.runNum = data(1);

obj.commas = data(2);

obj.equals = data(3);

obj.ifCount = data(4);

obj.forCount = data(5);

obj.whileCount = data(6);

obj.semiCount = data(7);

obj.switchCount = data(8);

obj.caseCount = data(9);

obj.ANDCount = data(10);

obj.ORCount = data(11);

obj.PLOC = data(12);

obj.LLOC = data(13);

obj.cyclomaticComplexity = data(14);

end

end

end

**VLLOCTable.m** – produces a table of the LLOC and V values for each analyzed program

This function has been improved to now output the table to disk in PNG format programmatically.

**Source Code:**

function [ ] = VLLOCTable( VData, LLOCData )

% This function will use a dataset and opslabels

% to output a table with useful data

labelFontSizeIncrease = 2;

% Generate the HTML injection string

htmlFSInc = strcat(['<html><font size=+' num2str(labelFontSizeIncrease) '>']);

V = cell2mat({VData})';

LLOC = cell2mat({LLOCData})';

data = horzcat(LLOC, V)';

% row names

rNames = {'Plateau';'Parabolic Approximation';'Array Processing';...

'Big Integers';'Iteration vs. Recursion';'Matrix Multiplication';...

'QuickSort';'Matrix Inversion'}';

% column labels

cNames = {'LLOC';'V'};

% This is a silly trick I learned on the internet

% HTML injection works in MatLab

cNames = strcat(htmlFSInc,cNames);

rNames = strcat(htmlFSInc,rNames);

f = figure('Position', [0, 0, 1280, 720]);

t = uitable(f,'Data',data',...

'ColumnName',cNames,...

'RowName',rNames,...

'FontSize',18);

% Set width and height

t.Position(3) = t.Extent(3);

t.Position(4) = t.Extent(4);

% Change figure dimensions so table can be saved correctly

f.Position(3) = t.Extent(3)+20;

f.Position(4) = t.Extent(4)+20;

% The below code requires the findjobj function

% download it here: http://www.mathworks.com/matlabcentral/fileexchange/14317-findjobj-find-java-handles-of-matlab-graphic-objects

jscrollpane = findjobj(t);

jTable = jscrollpane.getViewport.getView;

cellStyle = jTable.getCellStyleAt(0,0);

cellStyle.setHorizontalAlignment(cellStyle.CENTER);

% Table must be redrawn for the change to take affect

jTable.repaint;

% outputing data to disk

% get pixels per inch, for scaling conversion

pixperinch = get(0,'ScreenPixelsPerInch');

% get output directory

wd = fileparts(mfilename('fullpath'));

figDir = [wd '\..\figures\tables\'];

% dimensions must account for figure position

set(gcf,'PaperUnits','inches','PaperPosition',...

[0 0 (t.Extent(3)+t.Position(1))/pixperinch (t.Extent(4)+t.Position(2))/pixperinch])

% output will still have margins on left and bottom, these must be

% trimmed

print([figDir 'LLOCVTable'],'-dpng','');

end

**LLOCV.m** – calculates linear regression fit coefficients for LLOC, V and plots the information

**Source Code:**

function [] = LLOCV(VData, LLOCData)

xdata = LLOCData;

ydata = VData;

% linear fit data

coeffs = polyfit(xdata, ydata, 1);

% Get fitted values

fittedY = polyval(coeffs, xdata);

figure('Position', [0, 0, 1280, 720]);

h = plot(xdata, [ydata; fittedY], 'O');

% change y axis tick exponent

ax = gca;

ax.YAxis.Exponent = 0;

% set title, labels and legend

title('Relationship between LLOC and Cyclomatic Complexity');

ylabel('Cyclomatic Complexity V');

xlabel('LLOC');

legend(h,'Raw Data',['Best Fit line y = ' num2str(coeffs(1)) 'x + ' num2str(coeffs(2))],...

'Location','northwest');

%add regression line

hold on

h = plot(xdata, fittedY, '-');

%rescale figure

domainSize = max(xdata) - min(xdata);

rangeSize = max(ydata) - min(ydata);

axis([min(xdata) - domainSize \* .1,max(xdata)+ domainSize \* .1,...

min(ydata)- rangeSize \* .1,max(ydata)+ rangeSize \* .1])

% outputing data to disk

wd = fileparts(mfilename('fullpath'));

figDir = [wd '\..\figures\'];

set(gcf,'PaperUnits','inches','PaperPosition',[0 0 12 7])

print([figDir 'LLOCV'],'-dpng','');

end

## Analyzed C++ Files

**Cs600part1.cpp** – longest string of consecutive values

// hmw2part1.cpp : Defines the entry point for the console application.

// Samuel Gluss

// CS 600 Homework 2 Part 1

// 10/4/2015

#include <iostream>

#include <stdlib.h>

#include <algorithm>

using namespace std;

void printElements(int \*&array, int length)

{

//useful function we can use to print the elements in a dynamic array

cout << "array elements:" << endl;

for (int i = 0; i < length; i++)

{

cout << array[i] << " ";

}

cout << endl << endl;

}

int maxlen(int \*&testArray, int& length)

{

//find the longest string of consecutive numbers in the sorted array

int currentMax = testArray[0];

int currentMaxCounter = 1;

int runnerUp = 0;

int runnerUpCounter = 0;

for (int i = 1; i < length; i++)

{

//if next value is the same as running current max, increment maxcounter

if (currentMax == testArray[i])

{

currentMaxCounter++;

}

//if next element is a runnerup element, increment runnerupCounter

else if (runnerUp == testArray[i])

{

runnerUpCounter++;

//if runnerup has the same count or greater than current max, change runnerup to be new longest string

if (runnerUpCounter >= currentMax)

{

currentMaxCounter = runnerUpCounter;

runnerUpCounter = 0;

currentMax = runnerUp;

runnerUp = 0;

}

}

//if next element is neither a max or a runnerup, start runnerup over with new value and counter

else

{

runnerUp = testArray[i];

runnerUpCounter = 1;

}

}

return currentMaxCounter;

}

int main()

{

//create our initial array

int arraySize = 100;

int \*testArray = new int[arraySize];

//populate the array with some random values from 1 to 25

for (int i = 0; i < arraySize; i++)

{

testArray[i] = rand() % 25 + 1;

}

cout << "unsorted array:" << endl;

printElements(testArray, 100);

//sort the array using algorithm library sort

sort(testArray, testArray + 100);

cout << "sorted array:" << endl;

printElements(testArray, 100);

//execute the maxlen function here, output the results

cout << "Longest consecutive digit string has length of " << maxlen(testArray, arraySize) << endl;

system("pause");

return 0;

}

**Cs600part2.cpp** – Parabolic Approximation

#include <iostream>

void print(int a, int b, int c, int x1, int x3)

{

}

int determinant(int a, int b, int c, int d, int e, int f, int g, int h, int i)

{

int retVal = a \* (e \* i - f \* h) - b \* (d \* i - f \* g) + c \* (d \* h - e \* g);

return retVal;

}

void calculateCoefficientsCramers(int x1, int y1, int x2, int y2, int x3, int y3)

{

//use cramers rule to calculate coefficients for quadratic function

//calculate determinant of coefficient matrix (D) first

int det = determinant(x1\*x1, x1, 1, x2\*x2, x2, 1, x3\*x3, x3, 1);

if (det == 0)

{

std::cout << "determinant of coefficient matrix is zero, Cramers rule is not applicable";

return;

}

//calculate coefficients for y = ax^2 + bx + c

// a = Da / D

int a = determinant(y1, x1, 1, y2, x2, 1, y3, x3, 1) / det;

// b = Db / D

int b = determinant(x1\*x1, y1, 1, x2\*x2, y2, 1, x3\*x3, y3, 1) / det;

// c = Dc / D

int c = determinant(x1\*x1, x1, y1, x2\*x2, x2, y2, x3\*x3, x3, y3) / det;

}

void calculateCoefficientsLagrange(int x1, int y1, int x2, int y2, int x3, int y3)

{

//calculate

double denom1 = (x1 - x2)\*(x1 - x3);

double denom2 = (x2 - x1)\*(x2 - x3);

double denom3 = (x3 - x1)\*(x3 - x2);

double a = (y1 / denom1) + (y2 / denom2) + (y3 / denom3);

double b = -(((y1 \* (x2 + x3)) / denom1) + ((y2 \* (x1 + x3)) / denom2) + ((y3 \* (x1 + x2)) / denom3));

double c = ((y1 \* x2 \* x3) / denom1) + ((y2 \* x1 \* x3) / denom2) + ((y3 \* x1 \* x2) / denom3);

}

int main()

{

//test points

int x1 = 1;

int y1 = 9;

int x2 = 2;

int y2 = 23;

int x3 = 3;

int y3 = 45;

if (x1 != x2 && x2 != x3 && x1 != x3)

{

calculateCoefficientsCramers(x1, y1, x2, y2, x3, y3);

calculateCoefficientsLagrange(x1, y1, x2, y2, x3, y3);

}

else

{

std::cout << "All x values must be unique" << std::endl;

}

return 0;

}

**Cs600part3.cpp** – Array Processing

// hmw2part3.cpp : Defines the entry point for the console application.

// Samuel Gluss

// CS600 Homework 2 Part 3

// Array Reduce

// 10/5/2015

#include <iostream>

#include <stdlib.h>

using namespace std;

void printElements(int \*&array, int length)

{

//useful function we can use to print the elements in a dynamic array

cout << "array elements:" << endl;

for (int i = 0; i < length; i++)

{

cout << array[i] << " ";

}

cout << endl;

}

void reduce(int \*&array, int& length)

{

int top1 = 0, top2 = 0, top3 = 0, sizeCounter = 0, reducedArrayIndex = 0;

//get the values of the highest valued items in the array

for (int i = 0; i < length; i++)

{

if (array[i] > top3)

{

if (array[i] > top2)

{

if (array[i] > top1)

{

top3 = top2;

top2 = top1;

top1 = array[i];

}

else if (array[i] != top1)

{

top3 = top2;

top2 = array[i];

}

}

else if (array[i] != top2)

{

top3 = array[i];

}

}

}

//print top 3 values

cout << "highest values: " << top1 << " " << top2 << " " << top3 << endl;

//determine size of reduced array

for (int i = 0; i < length; i++)

{

if (array[i] < top3)

{

sizeCounter++;

}

}

//allocate new array to hold the reduced array values

int \*reducedArray = new int[sizeCounter];

//initialize reduced array

for (int i = 0; i < length; i++)

{

if (array[i] < top3)

{

reducedArray[reducedArrayIndex] = array[i];

reducedArrayIndex++;

}

}

//reassign the original matrix

delete(array);

array = reducedArray;

length = sizeCounter;

}

int main()

{

//create our initial array

int arraySize = 20;

int \*myArray = new int[arraySize];

//populate the array with some random values from 1 to 100

for (int i = 0; i < arraySize; i++)

{

myArray[i] = rand() % 100 + 1;

}

//print our initial array

printElements(myArray, arraySize);

reduce(myArray, arraySize);

//print the reduced array

printElements(myArray, arraySize);

system("PAUSE");

return 0;

}

**Cs600part4.cpp** – Print Big Integers

// hmw2part4.cpp : Sam Gluss 10/5/2015

// CS600 HMW2 Part 4: printing big integers

#include <iostream>

#include <string>

using namespace std;

int getDigitCount(int number)

{

//counts the viable digits in the number

int digitCount = 0;

while (number) {

number /= 10;

digitCount++;

}

return digitCount;

}

char nthdigit(int number, int digitPos)

{

//function which returns the character of the desired digit

int retVal;

while (digitPos--)

{

number /= 10;

}

retVal = (number % 10) + '0';

return retVal;

}

void bigInt(int toPrint)

{

string digit0[7] = { " $$$$$ ",

" $ $ ",

" $ $ ",

" $ $ ",

" $ $ ",

" $ $ ",

" $$$$$ ", };

string digit1[7] = { " $ ",

" $$ ",

" $$$ ",

" $ ",

" $ ",

" $ ",

" $$$$$ ", };

string digit2[7] = { " $$$ ",

" $ $ ",

" $ ",

" $ ",

" $ ",

" $ ",

" $$$$$ ", };

string digit3[7] = { " $$$ ",

" $ $ ",

" $ ",

" $$ ",

" $ ",

" $ $ ",

" $$$ ", };

string digit4[7] = { " $$ ",

" $$ $ ",

" $ $ ",

" $$$$$ ",

" $ ",

" $ ",

" $ ", };

string digit5[7] = { " $$$$$ ",

" $ ",

" $ ",

" $$$ ",

" $ ",

" $ $ ",

" $$$ ", };

string digit6[7] = { " $$ ",

" $ ",

" $ ",

" $$$ ",

" $ $ ",

" $ $ ",

" $$$ ", };

string digit7[7] = { " $$$$$ ",

" $ ",

" $ ",

" $ ",

" $ ",

" $ ",

" $ ", };

string digit8[7] = { " $$$ ",

" $ $ ",

" $ $ ",

" $$$ ",

" $ $ ",

" $ $ ",

" $$$ ", };

string digit9[7] = { " $$$ ",

" $ $ ",

" $ $ ",

" $$$ ",

" $ ",

" $ ",

" $$ ", };

int digitCount = getDigitCount(toPrint);

for (int line = 0; line < 7; line++)

{

//print each line, from top to bottom

for (int i = 0; i < digitCount; i++)

{

//determine which digit to get a line from

switch (nthdigit(toPrint, (digitCount - (i + 1))))

{

case '0':

cout << digit0[line];

break;

case '1':

cout << digit1[line];

break;

case '2':

cout << digit2[line];

break;

case '3':

cout << digit3[line];

break;

case '4':

cout << digit4[line];

break;

case '5':

cout << digit5[line];

break;

case '6':

cout << digit6[line];

break;

case '7':

cout << digit7[line];

break;

case '8':

cout << digit8[line];

break;

case '9':

cout << digit9[line];

break;

}

}

cout << endl;

}

cout << endl;

}

int main()

{

//test some big Integer prints

bigInt(2746);

bigInt(1234567890);

bigInt(5318008);

system("pause");

return 0;

}

**Cs600part5.cpp** – Iteration vs Recursion

// Samuel Gluss

// CS 600 Homework 2 Part 5

// Performance of iterative and recursive binary search

// 10/5/2015

#include <iostream>

#include <chrono>

#include <stdlib.h>

#include <algorithm>

#include <random>

using namespace std;

int iterativeBinarySearch(int \*&array, int size, int target)

{

int foundIndex = 0;

int low = 0;

int high = size - 1;

int mid = 0;

//search loop

while (low <= high)

{

mid = (low + high) / 2;

if (array[mid] > target)

{

high = mid - 1;

}

else if (array[mid] < target)

{

low = mid + 1;

}

else

{

return mid;

}

}

}

int recursiveBinarySearch(int \*&array, int target, int low, int high)

{

if (high < low)

{

return -1;

}

//set mid to midpoint, use this to section the array for search

int mid = (low + high) / 2;

if (array[mid] > target)

{

return recursiveBinarySearch(array, target, low, mid - 1);

}

else if (array[mid] < target)

{

return recursiveBinarySearch(array, target, mid + 1, high);

}

else

{

return mid;

}

}

int main()

{

//create our initial array

int arraySize = 100000000;

int \*testArray = new int[arraySize];

cout << "populating array..." << endl;

//populate the array with some random values from 1 to 25

for (int i = 0; i < arraySize; i++)

{

testArray[i] = i;

}

//setting up the random number generator

srand(time(NULL));

std::default\_random\_engine generator;

std::uniform\_int\_distribution<int> distribution(1, arraySize);

//set up timer to measure duration of tests

std::chrono::time\_point<std::chrono::system\_clock> start, end, start2, end2;

int testsToRun = 1000000;

int target = 0;

int result = 0;

cout << endl << "testing iterative search:" << endl;

//testing the iterative search

start = std::chrono::system\_clock::now();

for (int i = 0; i < testsToRun; i++)

{

target = distribution(generator);

//disable cout statement for live tests

//cout << "found : " << iterativeBinarySearch(testArray, arraySize, target) << endl;

result = iterativeBinarySearch(testArray, arraySize, target);

}

end = std::chrono::system\_clock::now();

std::chrono::duration<double> elapsed\_seconds = end - start;

cout << endl << "testing recursive search:" << endl;

//testing the recursive search

start2 = std::chrono::system\_clock::now();

for (int i = 0; i < testsToRun; i++)

{

target = distribution(generator);

//disable cout statement for live tests

//cout << "found : " << recursiveBinarySearch(testArray, target, 0, arraySize - 1) << endl;

result = recursiveBinarySearch(testArray, target, 0, arraySize - 1);

}

end2 = std::chrono::system\_clock::now();

std::chrono::duration<double> elapsed\_seconds2 = end2 - start2;

cout << "Duration of iterative searches: " << elapsed\_seconds.count() << "s" << endl;

cout << "Duration of recursive searches: " << elapsed\_seconds2.count() << "s" << endl;

system("pause");

return 0;

}

**matmult6.cpp** - Matrix Multiplication Benchmark

#include <iostream>

#include <chrono>

using namespace std;

int main()

{

// chrono::steady\_clock has previously been established to have an error of 2.689e-07 seconds

// for these experiments, we must assume a total error of 5.378e-07 seconds

// due to the worst case scenario for two time measurements (start/stop)

// initialize timers

chrono::steady\_clock::time\_point startTime, stopTime;

double multTime;

double multTimes[25];

int iteration = 0;

int rowCount = 500;

// matrix A/B initialization

double multRepCount = 10;

double\*\* a = new double\*[rowCount];

double\*\* b = new double\*[rowCount];

double\*\* c = new double\*[rowCount];

for (int i = 0; i < rowCount; i++)

{

a[i] = new double[rowCount];

b[i] = new double[rowCount];

c[i] = new double[rowCount];

}

startTime = chrono::steady\_clock::now();

for (int K = 0; K < multRepCount; K++)

{

for (int i = 0; i < rowCount; i++)

{

// initialize rows to desired values

for (int j = 0; j < rowCount; j++)

{

if (i == j)

{

a[i][j] = 2.002;

b[i][j] = 2.002;

}

else

{

a[i][j] = 1.001;

b[i][j] = 1.001;

}

c[i][j] = 0;

}

}

// Populate C with appropriate values

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

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

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

c[j][i] += a[k][i] \* b[j][k];

}

}

}

}

stopTime = chrono::steady\_clock::now();

multTime = chrono::duration\_cast<chrono::duration<double>>(stopTime - startTime).count() / multRepCount;

// clean memory

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

delete[] a[i];

delete[] b[i];

delete[] c[i];

}

delete[] a;

delete[] b;

delete[] c;

multTimes[iteration] = multTime;

iteration++;

cout << multTime << endl;

return 0;

}

**qsmark7.cpp** - QuickSort Benchmark

#include <iostream>

#include <chrono>

#include <random>

using namespace std;

/\*

This quicksort algorithm is taken from

http://www.algolist.net/Algorithms/Sorting/Quicksort

\*/

void quickSort(int arr[], int left, int right) {

int i = left, j = right;

int tmp;

int pivot = arr[(left + right) / 2];

/\* partition \*/

while (i <= j) {

while (arr[i] < pivot)

i++;

while (arr[j] > pivot)

j--;

if (i <= j) {

tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

i++;

j--;

}

};

/\* recursion \*/

if (left < j)

quickSort(arr, left, j);

if (i < right)

quickSort(arr, i, right);

}

int main()

{

// chrono::steady\_clock has previously been established to have an error of 2.689e-07 seconds

// for these experiments, we must assume a total error of 5.378e-07 seconds

// due to the worst case scenario for two time measurements (start/stop)

// initialize timers

chrono::steady\_clock::time\_point startTime, stopTime;

double sortTime;

// initialize random generator, set generator type/range here

random\_device rd;

mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(1, 1000000);

// array initialization

int sortCount = 10;

int arrSize = 1000000;

int\* toSort = new int[arrSize];

for (int i = 0; i < arrSize; i++)

{

toSort[i] = dis(gen);

}

startTime = chrono::steady\_clock::now();

// Call to sort function

quickSort(toSort,0,arrSize-1);

stopTime = chrono::steady\_clock::now();

sortTime = chrono::duration\_cast<chrono::duration<double>>(stopTime - startTime).count() / sortCount;

cout << sortTime << endl;

return 0;

}

**matinvert8.cpp** - Matrix Inversion Benchmark

#include <iostream>

#include <random>

#include <chrono>

using namespace std;

int main()

{

// chrono::steady\_clock has previously been established to have an error of 2.689e-07 seconds

// for these experiments, we must assume a total error of 5.378e-07 seconds

// due to the worst case scenario for two time measurements (start/stop)

// initialize random generator, set generator type/range here

random\_device rd;

mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(1, 1000);

// initialize timers

chrono::steady\_clock::time\_point startTime, stopTime;

double initTime1;

double invTime;

double initTime2;

double multTime2;

double initTimes[25];

double invTimes[25];

int iteration = 0;

for (int rowCount = 20; rowCount <= 500; rowCount += 20) {

// matrix A/B initialization

int initRepCount = 200 / sqrt(rowCount);

int multRepCount = 200 / sqrt(rowCount);

int\*\* matA = new int\*[rowCount];

startTime = chrono::steady\_clock::now();

for (int K = 0; K < initRepCount; K++)

{

for (int i = 0; i < rowCount; i++)

{

matA[i] = new int[rowCount];

// initialize rows to random values

for (int j = 0; j < rowCount; j++)

{

matA[i][j] = dis(gen);

}

}

}

stopTime = chrono::steady\_clock::now();

initTime1 = chrono::duration\_cast<chrono::duration<double>>(stopTime - startTime).count() / initRepCount;

startTime = chrono::steady\_clock::now();

for (int K = 0; K < multRepCount; K++)

{

/\*

begin inversion code from numerical recipes 2nd ed

\*/

int\* indxc = new int[rowCount];

int\* indxr = new int[rowCount];

int\* ipiv = new int[rowCount];

int i, icol = 0, irow = 0, j, k, l, ll;

double big, dum, pivinv, temp;

int\* temp2;

for (j = 0; j<rowCount; ++j)

ipiv[j] = 0;

for (i = 0; i<rowCount; ++i)

{

big = 0.0;

for (j = 0; j<rowCount; ++j)

{

if (ipiv[j] != 1)

{

for (k = 0; k<rowCount; ++k)

{

if (ipiv[k] == 0)

{

if (fabs(matA[j][k]) >= big)

{

big = fabs(matA[j][k]);

irow = j;

icol = k;

}

}

else

{

if (ipiv[k] > 1)

{

// Something bad happened?

}

}

}

}

}

++ipiv[icol];

if (irow != icol)

{

temp2 = matA[irow];

matA[irow] = matA[icol];

matA[icol] = temp2;

}

indxr[i] = irow;

indxc[i] = icol;

if (matA[icol][icol] == 0.0)

{

// Something bad happened?

}

pivinv = 1.0 / matA[icol][icol];

matA[icol][icol] = 1.0;

for (l = 0; l<rowCount; ++l)

matA[icol][l] \*= pivinv;

for (ll = 0; ll<rowCount; ++ll)

{

if (ll != icol)

{

dum = matA[ll][icol];

matA[ll][icol] = 0.0;

for (l = 0; l<rowCount; ++l)

matA[ll][l] -= matA[icol][l] \* dum;

}

}

}

for (l = rowCount - 1; l >= 0; --l)

{

if (indxr[l] != indxc[l])

{

for (k = 0; k<rowCount; ++k)

{

temp = matA[k][indxr[l]];

matA[k][indxr[l]] = matA[k][indxc[l]];

matA[k][indxc[l]] = temp;

}

}

}

/\*

end inversion code

\*/

}

stopTime = chrono::steady\_clock::now();

invTime = chrono::duration\_cast<chrono::duration<double>>(stopTime - startTime).count() / multRepCount;

initTimes[iteration] = initTime1;

invTimes[iteration] = invTime;

iteration++;

//end of execution for each rowcount test

}

cout << "init times:\n";

for (int i = 0; i < 25; i++)

{

cout << initTimes[i] << ",";

}

cout << "\n\ninversion times:\n";

for (int i = 0; i < 25; i++)

{

cout << invTimes[i] << ",";

}

return 0;

}

# Citations

1. McCabe, Thomas J. – A Complexity Measure – IEEE Transactions 1976