

Project Report

SYMBOLIC EXECUTION

Course: SE305 Software Project Lab – 1

Submitted by

<i>Toukir Ahammed</i>	<i>Roll: 0806</i>	<i>2015-2016</i>
-----------------------	-------------------	------------------

Document Version: 1.0

Submitted to

SPL I Coordinators

Rezvi Shahariar, Assistant Professor, IITDU

Amit Seal Ami, Lecturer, IITDU



Institute of Information Technology

University of Dhaka

[30-04-2017]

To Whom It May Concern

This is to certify that TOUKIR AHAMMED, BSSE0806 has successfully completed the project titled "SYMBOLIC EXECUTION" at Institute of Information Technology, University of Dhaka under my supervision and guidance in the fulfillment of requirements of Software Project Lab – I.

Alim Ul Gias
Lecturer
Institute of Information Technology
University of Dhaka

Acknowledgement

At first I would like to thank Almighty Allah for helping me to accomplish my goals.

I would like to express my deepest gratitude to all of those who has supported me to complete this project. I am grateful to my supervisor Alim Ul Gias for helping me in all way to complete this project. I would also like to thank the coordinators of Software Project Lab – 1 who has instructed us throughout this project to complete this project.

Lastly I would like to thank all of my classmates for assisting me and providing valuable insights throughout the project.

Toukir Ahammed
BSSE0806
Institute of Information Technology
University of Dhaka

Executive Summary

Almost every software requires checking whether certain properties are hold by the program or not. One approach is may be test the program with random inputs. But it has the limitation exploring all possible paths. An efficient solution to this problem is symbolic execution which is capable of exploring many possible paths without providing concrete inputs. After performing a symbolic execution the program outputs a set of expression containing constraints for each path. Solving these constrains a solution is for each variable is achieved. The goal of the project is to achieve this solution which can be used for testing and verification in software development.

Table of Contents

1. Introduction	1
2. Background Study	3
2.1 Symbolic Execution	3
2.2 Basic Block	3
2.3 Control Flow Graph.....	3
2.4 Control Flow Path	4
2.5 Path Constraint (PC) and Constraint Solving	4
3. Broad Domain	4
4. Challenges	5
5. Dependencies.....	6
5.1 Software	6
5.2 Hardware	6
6. Methodology	7
7. Program Output.....	10
8. Achievements.....	12
8.1 Technological.....	12
8.2 Personal.....	13
9. User Manual	13
10. Conclusion	14

Table of Figures

Figure 1: Step by step procedure of this project	7
Figure 2: A sample input file	10
Figure 3: sample input file after numbering basic block	11
Figure 4: Output for the sample input	12
Figure 5: <i>main</i> function	16
Figure 6: <i>basicBlockNnumbering</i> function(part 1)	17
Figure 7: <i>basicBlockNnumbering</i> function(part 2)	17
Figure 8: <i>processBlock</i> function (part 1)	18
Figure 9: <i>processBlock</i> function (part 2)	19
Figure 10: <i>processBlock</i> function (part 3)	20
Figure 11: <i>processBlock</i> function (part 4)	21
Figure 12: <i>processBlock</i> function (part 5)	22
Figure 13: <i>solve</i> function (part 1)	23
Figure 14: <i>solve</i> function (part 2)	24
Figure 15: <i>solve</i> function (part 3)	25
Figure 16: <i>solve</i> function (part 4)	26
Figure 17: <i>solve</i> function (part 5)	27
Figure 18: <i>solve</i> function (part 6)	28
Figure 19: <i>solve</i> function (part 7)	29

Figure 20: <i>sol/ve</i> function (part 8)	30
Figure 21: <i>sol/ve</i> function (part 9)	31
Figure 22: <i>sol/ve</i> function (part 10)	32
Figure 23: <i>sol/ver</i> function	32

1. Introduction

Symbolic execution is a useful technique of analyzing a source code to identify which inputs cause each part of a program to execute. In recent years it has been a major part of software development and verification as a popular way to aid software testing. It is an effective technique for generating high coverage of test cases and finding deep errors in the complex software.

In concrete execution a program is run on a specific input value and thus a single control flow path is explored. On the other hand, symbolic execution does not execute a program with a concrete value of input. In symbolic execution a symbolic value (e.g., a) is used as input. This symbol can take any value in the domain. So the program can take any feasible path in the control flow graph and thus explore multiple control flow paths simultaneously. A program which is executed using concrete values as input results a series of concrete values as output. In symbolic execution the value is replaced by a symbol and as a result a set of expressions is produced as output.

Symbolic execution has a variety of many important uses. The main uses of symbolic execution are in the software development area. It can be used for test case generation, path domain checking, program proving, symbolic debugging and program reduction. It can also be used to check various kinds of errors including assertion violations, uncaught exception, security vulnerabilities and memory corruption. Besides this there are many uses of symbolic execution in the field of software development, testing and verification.

The goal of this project is to build a tool which can perform symbolic execution of a source code. There will be a source code written in C programming language as input which has to be executed symbolically.

After reading this source code, the tool should have to find all possible execution paths. The tool should also have to show the path constraints for each path that should be followed to cover that specific path. At last a possible solution for each path should have to be provided as final output.

Although, the idea about symbolic execution was introduced about more than three decades ago [1, 2] it has recently become more popular as an effective technique for generating test cases for software testing. There are some popular tools for symbolic execution for different programming languages such as KLEE, KITE for LLVM, Java Path Finder (JPF), jCUTE, JBSE for Java, Otter for C etc. In this project a simple symbolic execution tool has been developed for the source code written in C programming language.

The scope of this project is defined as follows: the tool produces only all possible paths for the given source code, shows path constraints for each path and finally gives a possible solution for each path. It does not provide the diagrammatic representation of the control flow graph. It does not produce expressions for any output and intermediate variable. Hopefully the limitations will be overcome and other features will be provided in future work.

The simple symbolic execution technique has been used in this project. At first a control flow graph has been generated from the given source code. Every basic block has been considered as a node. A directed edge has been provided between two nodes if the second node can be executed immediate after the first node. The edges of the graph are labeled with the associated constraints to go from one node to another node. All possible execution paths have been produced by traversing the control flow graph from the entry point to exit point of the program. A list of constraints for each path has also been produced while traversing the control. These constraints have been referred as path constraints (PC). A

possible solution for each path has been found by solving these constraints.

2. Background Study

There are some terminologies associated with this project. These terminologies are described in this section.

2.1 Symbolic Execution

Symbolic execution or sometimes referred as symbolic evaluation means executing a program with symbolic value rather than a concrete value. In normal execution the input of the program is a concrete value. But in symbolic execution the input of the program is a symbol that can take any value within its domain. In symbolic execution a program can adopt multiple paths simultaneously when there is a fixed path for a fixed value in normal execution. The output of a symbolically executed program is a set of expressions for each path.

2.2 Basic Block

A basic block is a code sequence where no branches in except to the entry point and no branches out except to the exit point. A basic block must have the following property:

- One entry point, meaning no code within this block is the destination of any jump instruction.
- One exit point, meaning only the last statement of this block can cause the program to begin executing code in a different basic block.

2.3 Control Flow Graph

Control flow graph is a directed graph where each node represents a basic block and there is an edge from node A to B if node B can be executed

immediately after node A. There are two special nodes called entry node from where the graph is started and exit node to which the graph is ended.

2.4 Control Flow Path

A control flow path is a simple path starting from entry node to exit node in a control flow graph. All possible control flow paths can be achieved by traversing the control flow graph from entry node to exit node. This traverse can be done easily with Depth First Search (DFS).

2.5 Path Constraint (PC) and Constraint Solving

Path constraint is an expression of some conditions that should be maintained to cover that associated path. A solution of a constraint is the assignment of one value for each variable that satisfies the constraint.

3. Broad Domain

Symbolic execution is a technique which is used for validation of software. As software engineering is becoming more concerned about the tools to facilitate software development and validation. Symbolic execution has got the attention in this field because it can be used to generate high coverage of test cases. In symbolic execution all possible execution paths can be checked which is very important in testing. Symbolic execution can provide facilities for automated testing in software development.

There are various uses of symbolic execution. Some of these are mentioned below:

Symbolic execution can be used for test case generation. The symbolic input values in the expression for each output variable can be substituted by a concrete values to generate different test cases. Another use of symbolic execution is Program reduction. It means producing a program

with fewer statements than the original one. King describes how symbolic execution is used in program reduction [3]. Symbolic execution is also used in symbolic debugging. The tracing of the execution of a program is a more powerful debugging technique. Symbolic execution enhances the tracing facilities by displaying the expression for each variable. Thus it is used in symbolic debugging efficiently.

Path domain checking is another useful application of symbolic execution. There can be three results [4] when a path is executed with a single case:

- incorrect output owing to one or more faults (universally incorrect)
- correct output although a fault exists (coincidentally correct)
- correct output and no fault exist (universally correct)

To distinguish between universally correct and coincidentally correct output symbolic execution can be used.

4. Challenges

There were so many challenging situations I have faced throughout this project. The most important challenge was going through the timeline of this project and completing it within the deadline properly. There were some challenges in handling the whole project, dividing the main problem in sub-problem, finding appropriate data structures, algorithms for them, implementing them step by step, documenting properly and report writing.

The first challenge I have faced in implementation was to construct a control flow graph from a given source code. It can be easily understood that once the control graph has been constructed, it is easy to find all control flow paths by traversing the graph. It was not so smooth to build a control flow graph from a source code where there could be nested blocks in the source code. To build a control flow graph the basic blocks of the source code must be identified and numbered properly. It was

another challenge to identify basic blocks in the source code and numbering them. After this the next challenge was to determine whether there was an edge or not between two nodes where each basic block was represented by a node. There was also another difficulty to put an edge label based on the constraint to go from one node to another. After constructing the control flow graph another new challenge has been faced. It was traversing the graph from the entry node to exit node and exploring all possible execution paths. Finding the path constraint for each path and storing them mapping with the associated path was not so easy. The last challenge but not the least was solving all the path constraints to seek a possible solution for each individual path. There were also some other difficulties regarding reading input file, handling data structure.

5. Dependencies

Dependencies can be divided into two parts. One is software dependencies and another one is hardware dependencies.

5.1 Software

GNU Compiler Collection (GCC) 4.9.2 or later version must have been installed to run this tool. This tool can be run in Windows and Linux based operating system.

5.2 Hardware

This tool can be run in any computers with modern hardware configuration available at present.

6. Methodology

A simple symbolic execution technique has been used to perform symbolic execution of the given source code in this project. The idea was to divide the whole problem into sub-problems and solve them step by step. The step by step procedure is shown in Figure 1.

The first task was to read the source code file on which symbolic execution would be performed. Before this the input file name has been taken from user with a prompt message. Then the given source file has been read with *ifstream* and *istreamstream*. The functions used to do this task are as follows: *openInputFile*, *openOutputFile*, *closeInputFile* and *closeOutputFile*.

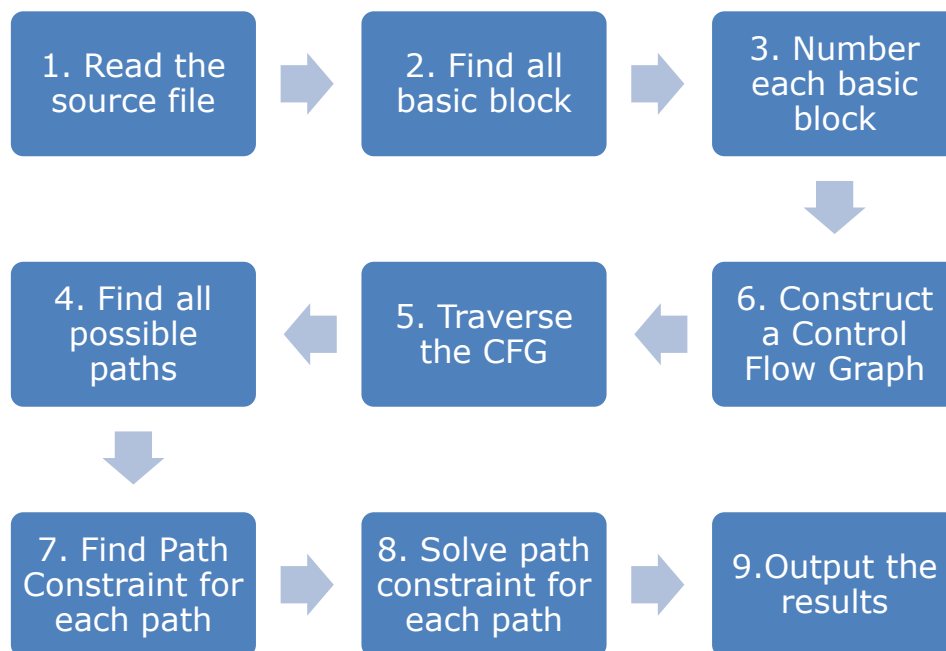


Figure 1: Step by step procedure of this project

The next task was finding all basic blocks in the source code and numbering them. This task has been done with the function named *basicBlockNumbering* (Appendix Figure 6-7). The source code file has

been rewritten once again after numbering in the second version of source code file. The algorithm of this function is given bellow:

1. read the source file line by line an rewrite in a new file
2. repeat the step-1 until the first curly brace occurs
3. continue reading the source file line by line and rewrite the line adding block number at the first
4. if the line contains *if*, *for*, *while* or *}* go to step-5 else skip step-5
5. increment block number
6. repeat step-3 to step-4 until end of the file

After numbering the basic block a control flow graph has been constructed using the function named *processBlock* (Appendix Figure 8-12). The algorithm of this functions is given below:

1. *push* the first block in a *queue*
2. take the *front* from the *queue*
3. find true block, false block (if there any) and exit block for current
4. if false block exists:
 - a. give a directed edge from current block to its true block and label it with true condition
 - b. give a directed edge from current block to its false block and label it with false condition
 - c. give a directed edge from true block to exit block and label it with nothing
 - d. give a directed edge from false block to exit block and label it with nothing
5. if false block does not exist:
 - a. give a directed edge from current block to its true block and label it with true condition
 - b. give a directed edge from true block to exit block and label it with nothing

- c. give a directed edge from current block to exit block and label it with nothing
6. *pop* the current block from the *queue*
7. repeat step-2 to step-6 until the *queue* is empty

After constructing the control flow graph it has been shown in a tabular form with the function named *showCFG*. Then all the paths from entry node to exit node has been explored with the help of the functions named *findAllPaths* and *findAllPathsUtil*. DFS algorithm has been used to implement these functions. Then all possible paths from entry to exit point of the source code have been displayed with the *printAllPaths* function. *printAllConditions* function has been used to show path constraints associated with every path.

At last *solve* (Appendix Figure 13-22) function has been used to find and show a possible solution for each path. This function can solve single variable simple equations and inequalities with the help of a function named *solver* (Appendix Figure 23). The variable can at most occurs two times in a path constraint.

7. Program Output

A sample input file is shown in Figure 2.

```
1  #include <stdio.h>
2  int main()
3  {
4      int x;
5      scanf("%d", &x);
6
7      if ( x > 0 )
8      {
9          printf("%d is positive\n", x);
10     }
11     else
12     {
13         if ( x == 0 )
14         {
15             printf("It is zero\n");
16         }
17         else
18         {
19             printf("%d is negative\n", x);
20         }
21     }
22     return 0;
23 }
```

Figure 2: A sample input file

The input file after numbering the basic block is shown in Figure 3.

```
1  #include <stdio.h>
2  int main()
3  {
4  1  int x;
5  1  scanf("%d", &x);
6  1
7  1  if ( x > 0 )
8  2  {
9  2      printf("%d is positive\n", x);
10 2  }
11 3  else
12 3  {
13 3      if ( x == 0 )
14 4      {
15 4          printf("It is zero\n");
16 4      }
17 5      else
18 5      {
19 5          printf("%d is negative\n", x);
20 5      }
21 6  }
22 7  return 0;
23 7}
24
```

Figure 3: sample input file after numbering basic block

The final output of the program is shown in Figure 4.

```
=====
Control Flow Graph
=====
Edges          Label
=====
(1, 2)         x > 0
(1, 3)         x <= 0
(2, 7)
(3, 4)         x == 0
(3, 5)         x != 0
(4, 6)
(5, 6)
(6, 7)

=====
All Paths
=====
Path 1: 1 ==> 2 ==> 7
Path 2: 1 ==> 3 ==> 4 ==> 6 ==> 7
Path 3: 1 ==> 3 ==> 5 ==> 6 ==> 7

Total Paths: 3

=====
All Paths Constraints
=====
Path 1: x > 0,
Path 2: x <= 0, x == 0,
Path 3: x <= 0, x != 0,

=====
Solution for all paths:
=====
Path 1: x: 1,
Path 2: x: 0,
Path 3: x: -1,

Process returned 0 (0x0)   execution time : 5.172 s
Press any key to continue.
```

Figure 4: Output for the sample input

8. Achievements

The achievements of this project can be divided into two parts.

8.1 Technological

The target of this project was making a tool to execute a program symbolically and it has been achieved successfully. The developed tool in this project can perform symbolic execution of a source code written in C programming language. It can also show all possible execution paths

along with their path constraints. A possible solution of each path has also been found with this tool.

8.2 Personal

There are a lot of personal achievements from this project for me. As this is my first project I have learned so many things and gained some new experiences throughout this project. I have learned how to handle a project, how to write proposal for a project, how to maintain time scale of a project and how to write a project report. I have known about symbolic execution by reading various articles and papers. I have also known about control flow graph, control flow paths, path constraints and constraint solving. I have learned how to construct a control flow graph, how to traverse it and how to find all possible paths and how to find a solution by solving constraints. My programming skill has been increased and capability of handling large code has been achieved throughout this project.

9. User Manual

The user manual to run this tool is described as follows:

1. Extract the Symbolic_Execution.zip file
2. Open command line (for Windows) or terminal (Linux)
3. Type the following command:
 - i. `g++ -o main main.cpp`
 - ii. `main` (for Windows) or, `./main` (for Linux)
4. Then enter the file name if the input file is in the same folder, otherwise specify the full path address (Example: `"C:\Users\iit\Desktop\input.c"`)

10. Conclusion

Symbolic execution is an efficient technique to aid software testing. The goal of this project has been achieved by developing a tool that can perform symbolic execution. It can execute a source code written in C programming language. There were some challenges in developing this tool. All the obstacles have been overcome successfully. A control flow graph has been generated as the first step of solution. Then traversing the graph all possible execution paths have been explored. Path constraints for each path have been gathered while traversing the graph. Finally a possible solution has been given for each path in the control flow graph. This tool has been checked with various type of source code. The future scope of this project may be overcoming the limitations of this project. Throughout this project I have learned handling a project and maintaining large code. The experiences I have gained from this project will help me in future project.

Reference

- [1] "SELECT – a formal system for testing and debugging programs by symbolic execution" by R.S. Boyer, B. Elspas, and K.N.Levitt. , SIGPLAN Not., 1975.
- [2] "Symbolic execution and program testing" by J.C., King, Commun. ACM, 1976.
- [3] "Program reduction using symbolic execution" by King, J.C., SIGSOFT Software Engineering Notes, 1981, page – (9-14).
- [4] "Symbolic execution systems – a review" by P.David Coward, Software Engineering Journal, 1998, page - 230.

Appendix

The important functions from the source code are added here.

```
62 |int main()
63 |{
64 |    printString("Enter the file name only if in the same directory (e.g. input.c)\nOr, specify the full path
65 |    cout << "\nEnter the filename:" << endl;
66 |    cin >> inputFileName;
67 |
68 |    basicBlockNumbering();
69 |
70 |    myQueue.push(1);
71 |
72 |    int lastBlock = 0;
73 |
74 |    while(!myQueue.empty())
75 |    {
76 |        int top = myQueue.front();
77 |        myQueue.pop();
78 |
79 |        processBlock(top);
80 |
81 |        lastBlock = max(lastBlock, top);
82 |
83 |    }
84 |
85 |    showCFG();
86 |
87 |    findAllPaths(1, lastBlock);
88 |    printAllPaths();
89 |    printAllConditions();
90 |    solve();
91 |
92 |    return 0;
93 |}
94 |
```

Figure 5: *main* function

```

132 void basicBlockNumbering()
133 {
134     ifstream inFile;
135     ofstream outFile;
136
137     openinputFile(inputFileName, inFile);
138     outputFileName = inputFileName.substr(0, inputFileName.size()-2) + "_v_1.c" ;
139     openOutputFile(outputFileName, outFile);
140
141     string line;
142
143     while (1)
144     {
145         getline(inFile,line);
146         outFile << line << endl;
147
148         if (line == "int main()") // start numbering after "int main()"
149         {
150             getline(inFile,line); // consume "{" immediate after "int main()"
151             outFile << line << endl;
152             break;
153         }
154     }
155
156     int blockNumber = 1;
157
158     string word;
159

```

Figure 6: *basicBlockNmumbering* function(part 1)

```

160     while (getline(inFile,line))
161     {
162
163         outFile << blockNumber << line << endl; //read and print line by line
164         //cout << line << endl;
165         istringstream iss;
166         iss.str(line);
167
168         while (iss>>word)
169         {
170             if (word == "if" || word == "}" || word == "for" || word == "while")
171             {
172                 blockNumber++; // increment blocknumber if "if" or "}" or "for" is found in a line
173             }
174         }
175     }
176
177
178     cout << "\nNumbering basic block has been finished and stored in \"" << outputFileName << "\" successfully."<< endl;
179
180     closeinputFile(inFile);
181     closeOutputFile(outFile);
182 }
183
184

```

Figure 7: *basicBlockNmumbering* function(part 2)


```

185 void processBlock(int blockNumber)
186 {
187     //cout << "\nProcessing " << blockNumber << endl;
188     ifstream inFile;
189     openInputFile(outputFileName, inFile);
190
191     int currentBlockNumber = 0;
192     int exitBlockNumber = findExit(blockNumber);
193
194     string line = "";
195     string condition = "";
196
197     int rightBlockNumber = 0;
198     int leftBlockNumber = 0;
199
200     while(getline(inFile, line))
201     {
202         string word = "";
203         istringstream iss;
204         iss.str(line + "$");
205
206         condition = "";
207         iss >> currentBlockNumber;
208         iss >> word;
209         //cout << line << endl;
210
211         if(currentBlockNumber == blockNumber && word == "if")
212         {
213             condition = parseIfCondition(iss);
214

```

Figure 8: *processBlock* function (part 1)

```

215 rightBlockNumber = currentBlockNumber + 1;
216 adjMatrix[blockNumber][rightBlockNumber] = 1;
217 edges[blockNumber][rightBlockNumber] = condition;
218 myQueue.push(rightBlockNumber);
219
220 //inFile.get();
221 skipABlock(inFile);
222
223 getline(inFile,line);
224 istream iss1;
225 iss1.str(line+" $");
226
227 iss1 >> currentBlockNumber;
228 iss1 >> word;
229
230 if(word == "else")
231 {
232     leftBlockNumber = currentBlockNumber;
233     adjMatrix[blockNumber][leftBlockNumber] = 1;
234     edges[blockNumber][leftBlockNumber] = reverseCondition(condition);
235     myQueue.push(leftBlockNumber);
236
237     skipABlock(inFile);
238
239     inFile >> currentBlockNumber;
240
241     adjMatrix[rightBlockNumber][currentBlockNumber] = 1;
242     adjMatrix[leftBlockNumber][currentBlockNumber] = 1;
243     myQueue.push(currentBlockNumber);
244
245 }
246 else

```

Figure 9: *processBlock* function (part 2)

```

246     else
247     {
248         adjMatrix[rightBlockNumber][currentBlockNumber] = 1;
249         adjMatrix[blockNumber][currentBlockNumber] = 1;
250         edges[blockNumber][currentBlockNumber] = reverseCondition(condition);
251         myQueue.push(currentBlockNumber);
252
253
254     }
255
256     if(exitBlockNumber)
257     {
258         adjMatrix[blockNumber][exitBlockNumber] = 0;
259         adjMatrix[currentBlockNumber][exitBlockNumber] = 1;
260     }
261
262     closeinputFile(inFile);
263
264     break;
265
266 }
267
268 else if(currentBlockNumber == blockNumber && word == "for")
269 {
270     condition = parseForCondition(iss);
271
272     rightBlockNumber = currentBlockNumber + 1;
273     adjMatrix[blockNumber][rightBlockNumber] = 1;
274     edges[blockNumber][rightBlockNumber] = condition;
275     myQueue.push(rightBlockNumber);
276

```

Figure 10: *processBlock* function (part 3)

```

277     skipABlock(inFile);
278
279     getline(inFile, line);
280     istream iss1;
281     iss1.str(line + " $");
282
283     iss1 >> currentBlockNumber;
284
285     adjMatrix[rightBlockNumber][currentBlockNumber] = 1;
286     adjMatrix[blockNumber][currentBlockNumber] = 1;
287     edges[blockNumber][currentBlockNumber] = reverseCondition(condition);
288     myQueue.push(currentBlockNumber);
289
290     if(exitBlockNumber)
291     {
292         adjMatrix[blockNumber][exitBlockNumber] = 0;
293         adjMatrix[currentBlockNumber][exitBlockNumber] = 1;
294     }
295
296     closeinputFile(inFile);
297
298     break;
299
300 }
301 else if(currentBlockNumber == blockNumber && word == "while")
302 {
303     condition = parseWhileCondition(iss);
304
305     rightBlockNumber = currentBlockNumber + 1;
306     adjMatrix[blockNumber][rightBlockNumber] = 1;
307     edges[blockNumber][rightBlockNumber] = condition;

```

Figure 11: *processBlock* function (part 4)

```

305     rightBlockNumber = currentBlockNumber + 1;
306     adjMatrix[blockNumber][rightBlockNumber] = 1;
307     edges[blockNumber][rightBlockNumber] = condition;
308     myQueue.push(rightBlockNumber);
309
310     skipABlock(inFile);
311
312     getline(inFile, line);
313     istringstream iss1;
314     iss1.str(line+ " $");
315
316     iss1 >> currentBlockNumber;
317
318     adjMatrix[rightBlockNumber][currentBlockNumber] = 1;
319     adjMatrix[blockNumber][currentBlockNumber] = 1;
320     edges[blockNumber][currentBlockNumber] = reverseCondition(condition);
321     myQueue.push(currentBlockNumber);
322
323     if(exitBlockNumber)
324     {
325         adjMatrix[blockNumber][exitBlockNumber] = 0;
326         adjMatrix[currentBlockNumber][exitBlockNumber] = 1;
327     }
328
329     closeinputFile(inFile);
330
331     break;
332
333 }
334
335 }
336
337 }

```

Figure 12: *processBlock* function (part 5)

```

694 void solve()
695 {
696     printString("Solution for all paths:");
697     map <string, constraint*> myMap;
698     map <string, constraint*> :: iterator it;
699
700     for(int i=0; i<allPaths.size(); i++)
701     {
702         for(int j=0; j<constraints[i].size(); j++)
703         {
704
705             constraint* c = constraints[i][j];
706
707             it = myMap.find(c->LeftOperand);
708             if(it == myMap.end())
709             {
710                 myMap[c->LeftOperand] = c;
711             }
712             else
713             {
714                 string previous = it->second->operatorr;
715                 string current = c->operatorr;
716
717                 if(previous == ">")
718                 {
719                     if(current == ">")
720                     {
721                         it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);
722                     }
723                     else if (current == ">=")
724                     {
725                         it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);
726                     }
727                 }
728                 else if (current == "<")
729                 {
730                     if((c->rightOperand - it->second->rightOperand) >= 2)
731                     {
732                         it->second->rightOperand = (it->second->rightOperand + c->rightOperand) / 2;
733                     }
734                 }
735             }
736         }
737     }
738 }

```

Figure 13: solve function (part 1)

```

733         it->second->rightOperand = (it->second->rightOperand + c->rightOperand) / 2;
734         it->second->operatorr = "=";
735     }
736     else
737     {
738         it->second->rightOperand = INF;
739         it->second->operatorr = "=";
740     }
741
742 }
743 else if (current == "<=")
744 {
745     if((c->rightOperand - it->second->rightOperand) >=1 )
746     {
747         it->second->rightOperand = c->rightOperand;
748         it->second->operatorr = "=";
749     }
750     else
751     {
752         it->second->rightOperand = INF;
753         it->second->operatorr = "=";
754     }
755
756 }
757
758 else if (current == "==")
759 {
760     int x = c->rightOperand;
761     if(x > it->second->rightOperand)
762     {
763         it->second->operatorr = c->operatorr;
764         it->second->rightOperand = x;
765     }
766     else
767     {
768         it->second->operatorr = c->operatorr;
769         it->second->rightOperand = INF;
770     }
771 }
772 }

```

Figure 14: solve function (part 2)

```

773     else if (current == "!=")
774     {
775         it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);
776     }
777
778 }
779 else if (previous == ">=")
780 {
781     if(current == ">")
782     {
783         it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);
784     }
785     else if (current == ">=")
786     {
787         it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);
788     }
789 }
790
791 else if (current == "<")
792 {
793     if((c->rightOperand - it->second->rightOperand) >=1 )
794     {
795         it->second->rightOperand = it->second->rightOperand;
796         it->second->operatorr = "==";
797     }
798     else
799     {
800         it->second->rightOperand = INF;
801         it->second->operatorr = "==";
802     }
803 }
804
805 else if (current == "<=")
806 {
807     if((c->rightOperand - it->second->rightOperand) >=0 )
808     {
809         it->second->rightOperand = it->second->rightOperand;
810         it->second->operatorr = "==";
811     }

```

Figure 15: solve function (part 3)


```

812     else
813     {
814         it->second->rightOperand = INF;
815         it->second->operatorr = "==";
816     }
817
818
819
820     }
821     else if (current == "==")
822     {
823         int x = c->rightOperand;
824         if(x >= it->second->rightOperand)
825         {
826             it->second->operatorr = c->operatorr;
827             it->second->rightOperand = x;
828         }
829         else
830         {
831             it->second->operatorr = c->operatorr;
832             it->second->rightOperand = INF;
833         }
834     }
835     else if (current == "!=")
836     {
837         it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);
838     }
839
840 }
841 else if (previous == "<")
842 {
843     if(current == ">")
844     {
845         if((it->second->rightOperand - c->rightOperand) >= 2)
846         {
847             it->second->rightOperand = (it->second->rightOperand + c->rightOperand) / 2;
848             it->second->operatorr = "==";
849         }
850     }

```

Figure 16: solve function (part 4)

```

850     else
851     {
852         it->second->rightOperand = INF;
853         it->second->operatorr = "==";
854     }
855
856 }
857 else if (current == ">=")
858 {
859     if((it->second->rightOperand - c->rightOperand) >= 1)
860     {
861         it->second->rightOperand = c->rightOperand;
862         it->second->operatorr = "==";
863     }
864     else
865     {
866         it->second->rightOperand = INF;
867         it->second->operatorr = "==";
868     }
869
870 }
871
872 else if (current == "<")
873 {
874     it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
875 }
876 else if (current == "<=")
877 {
878     it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
879 }
880 else if (current == "==")
881 {
882     int x = c->rightOperand;
883     if(x < it->second->rightOperand)
884     {
885         it->second->operatorr = c->operatorr;
886         it->second->rightOperand = x;
887     }
888 }

```

Figure 17: solve function (part 5)

```

887     }
888     else
889     {
890         it->second->operatorr = c->operatorr;
891         it->second->rightOperand = INF;
892     }
893 }
894 else if (current == "!=")
895 {
896     it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
897 }
898
899 }
900 else if (previous == "<=")
901 {
902     if(current == ">")
903     {
904         if((it->second->rightOperand - c->rightOperand) >= 1)
905         {
906             it->second->rightOperand = it->second->rightOperand;
907             it->second->operatorr = "==" ;
908         }
909         else
910         {
911             it->second->rightOperand = INF;
912             it->second->operatorr = "==" ;
913         }
914     }
915     else if (current == ">=")
916     {
917         if((it->second->rightOperand - c->rightOperand) >= 0)
918         {
919             it->second->rightOperand = it->second->rightOperand;
920             it->second->operatorr = "==" ;
921         }
922         else
923         {
924             it->second->rightOperand = INF;
925             it->second->operatorr = "==" ;
926         }
927     }

```

Figure 18: solve function (part 6)

```

928     else if (current == "<")
929     {
930         it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
931     }
932     else if (current == "<=")
933     {
934         it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
935     }
936     else if (current == "==")
937     {
938         int x = c->rightOperand;
939         if(x <= it->second->rightOperand)
940         {
941             it->second->operatorr = c->operatorr;
942             it->second->rightOperand = x;
943         }
944         else
945         {
946             it->second->operatorr = c->operatorr;
947             it->second->rightOperand = INF;
948         }
949     }
950     else if (current == "!=")
951     {
952         it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
953     }
954 }
955
956 else if (previous == "==")
957 {
958     if(current == ">")
959     {
960         if(!(it->second->rightOperand > c->rightOperand))
961         {
962             it->second->operatorr = "==";
963             it->second->rightOperand = INF;
964         }
965     }
966 }

```

Figure 19: solve function (part 7)

```

967
968 ▼
969
970 ▼
971
972
973
974
975
976 ▼
977
978 ▼
979
980
981
982
983
984 ▼
985
986 ▼
987
988
989
990
991
992 ▼
993
994 ▼
995
996
997
998
999
1000
1001 ▼
1002
1003 ▼
1004
1005
1006
1007

else if (current == ">=")
{
    if(!(it->second->rightOperand >= c->rightOperand))
    {
        it->second->operatorrr = "==";
        it->second->rightOperand = INF;
    }
}
else if (current == "<")
{
    if(!(it->second->rightOperand < c->rightOperand))
    {
        it->second->operatorrr = "==";
        it->second->rightOperand = INF;
    }
}
else if (current == "<=")
{
    if(!(it->second->rightOperand <= c->rightOperand))
    {
        it->second->operatorrr = "==";
        it->second->rightOperand = INF;
    }
}
else if (current == "==")
{
    if(!(it->second->rightOperand == c->rightOperand))
    {
        it->second->operatorrr = "==";
        it->second->rightOperand = INF;
    }
}
else if (current == "!=")
{
    if(!(it->second->rightOperand != c->rightOperand))
    {
        it->second->operatorrr = "==";
        it->second->rightOperand = INF;
    }
}

```

Figure 20: solve function (part 8)

```

1009     }
1010     else if (previous == "!=")
1011     {
1012         if(current == ">")
1013         {
1014             it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);
1015         }
1016         else if (current == ">=")
1017         {
1018             it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);
1019         }
1020         else if (current == "<")
1021         {
1022             it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
1023         }
1024         else if (current == "<=")
1025         {
1026             it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
1027         }
1028         else if (current == "==")
1029         {
1030             int x = c->rightOperand;
1031             if(x == it->second->rightOperand)
1032             {
1033                 it->second->operatorr = c->operatorr;
1034                 it->second->rightOperand = x;
1035             }
1036             else
1037             {
1038                 it->second->operatorr = c->operatorr;
1039                 it->second->rightOperand = INF;
1040             }
1041         }
1042     }
1043     }
1044     else if (current == "!=")
1045     {
1046         if(!(it->second->rightOperand != c->rightOperand))
1047         {
1048             it->second->operatorr = "==";
1049             it->second->rightOperand = INF;

```

Figure 21: solve function (part 9)

```

1050         }
1051     }
1052
1053     }
1054 }
1055
1056
1057 }
1058
1059 cout << "Path " << i+1 << ": ";
1060 for(it = myMap.begin(); it!=myMap.end(); it++)
1061 {
1062     constraint* c = it->second;
1063     int soln = solver(c);
1064
1065     if(soln != INF)
1066         cout << it->first << ": " << soln << ", ";
1067     else
1068         cout << "No solution exist";
1069 }
1070 cout << endl;
1071 myMap.clear();
1072 }
1073
1074 }
1075

```

Figure 22: *solve* function (part 10)

```

1076 int solver(constraint* c)
1077 {
1078     int solution = 0;
1079     string relationalOperator [] = {">", "<", "<=", ">=", "==", "!=" };
1080     int solutionMaker[] = {+1, -1, -1, +1, 0, +5 };
1081
1082     string leftOperand = c->leftOperand;
1083     string operatorrr = c->operatorrr;
1084     int rightOperand = c->rightOperand;
1085
1086     for (int i = 0; i < 6; ++i)
1087     {
1088         if(operatorrr == relationalOperator[i])
1089         {
1090             solution = rightOperand + solutionMaker[i];
1091             break;
1092         }
1093     }
1094
1095     return solution;
1096 }
1097

```

Figure 23: *solver* function

