Project Report

SYMBOLIC EXECUTION

Course: SE305 Software Project Lab – 1

Submitted by

Toukir Ahammed	*Roll: 0806*	*2015-2016*

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

Table of Figures

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., ɑ) 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 constrains 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 has 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 (coincidently correct)
- correct output and no fault exist (universally correct)

To distinguish between universally correct and coincidently 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 constrains 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 *istringstream*. The functions used to do this task are as follows: *openInputFile*, *openOutputFile*, *closeInputFile* and *closeOutputFile*.

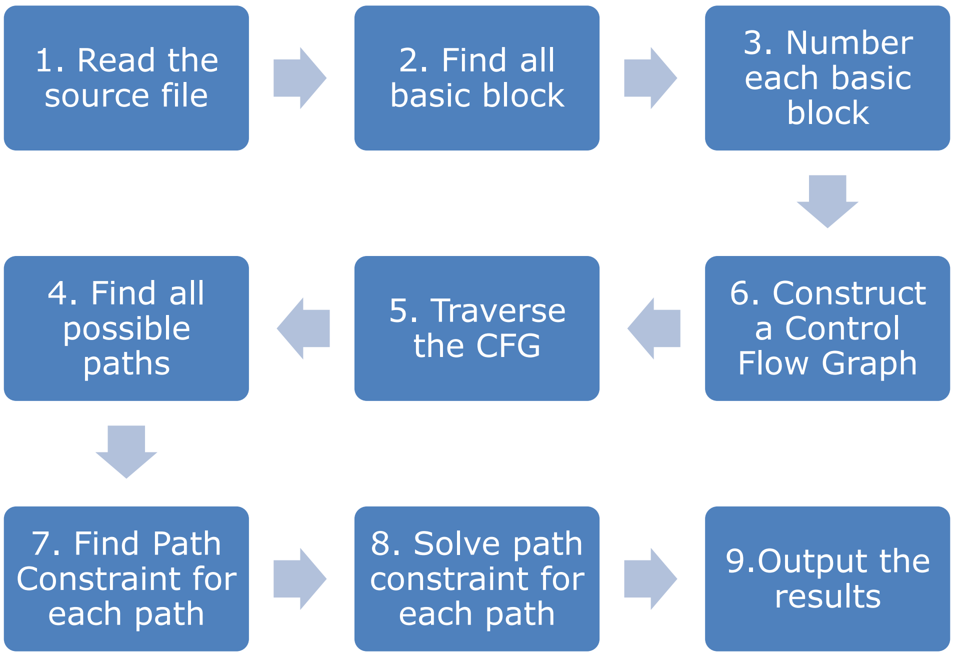

Figure 1: Step by step procedure of this project

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.

```c
#include <stdio.h>
int main()
{
    int x;
    scanf("%d", &x);

    if ( x > 0 )
    {
        printf("%d is postive\n", x);
    }
    else
    {
        if ( x == 0 )
        {
            printf("It is zero\n");
        }
        else
        {
            printf("%d is negative\n", x);
        }
    }
    return 0;
}
```

Figure 2: A sample input file

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

```
#include <stdio.h>
int main()
{
1   int x;
1   scanf("%d", &x);
1
1   if ( x > 0 )
2   {
2       printf("%d is postive\n", x);
2   }
3   else
3   {
3       if ( x == 0 )
4       {
4           printf("It is zero\n");
4       }
5       else
5       {
5           printf("%d is negative\n", x);
5       }
6   }
7   return 0;
7}
```

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 glow 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 has 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.

```
int main()
{
    printString("Enter the file name only if in the same directory (e.g. input.c)\nOr, specify the full path
    cout << "\nEnter the filename:" << endl;
    cin >> inputFileName;

    basicBlockNumbering();

    myQueue.push(1);

    int lastBlock = 0;

    while(!myQueue.empty())
    {
        int top = myQueue.front();
        myQueue.pop();

        processBlock(top);

        lastBlock = max(lastBlock, top);

    }

    showCFG();

    findAllPaths(1, lastBlock);
    printAllPaths();
    printAllConditions();
    solve();

    return 0;
}
```

Figure 5: *main* function

```
void basicBlockNumbering()
{
    ifstream inFile;
    ofstream outFile;

    openinputFile(inputFileName, inFile);
    outputFileName = inputFileName.substr(0, inputFileName.size()-2) + "_v_1.c" ;
    openOutputFile(outputFileName, outFile);

    string line;

    while (1)
    {
        getline(inFile,line);
        outFile << line << endl;

        if (line == "int main()")   // start numbering after "int main()"
        {
            getline(inFile,line);   // consume "{" immediate after "int main()"
            outFile << line << endl;
            break;
        }
    }

    int blockNumber = 1;

    string word;

```

Figure 6: *basicBlockNmumbering* function(part 1)

```
    while (getline(inFile,line))
    {

        outFile << blockNumber << line << endl;      //read and print line by line
        //cout << line << endl;
        istringstream iss;
        iss.str(line);

        while (iss>>word)
        {
            if (word == "if" || word == "}" || word == "for" || word == "while")
            {
                blockNumber++; // increment blocknumber if "if" or "}" or "for" is found in a line
            }
        }

    }

    cout << "\nNumbering basic block has been finished and stored in \"" << outputFileName << "\" successfully."<< endl;

    closeinputFile(inFile);
    closeOutputFile(outFile);

}

```

Figure 7: *basicBlockNmumbering* function(part 2)

```cpp
void processBlock(int blockNumber)
{
    //cout << "\nProcessing " << blockNumber << endl;
    ifstream inFile;
    openinputFile(outputFileName, inFile);

    int currentBlockNumber = 0;
    int exitBlockNumber = findExit(blockNumber);

    string line = "";
    string condition = "";

    int rightBlockNumber = 0;
    int leftBlockNumber = 0;

    while(getline(inFile, line))
    {
        string word = "";
        istringstream iss;
        iss.str(line + "$");

        condition = "";
        iss >> currentBlockNumber;
        iss >> word;
        //cout << line << endl;

        if(currentBlockNumber == blockNumber && word == "if")
        {
            condition = parseIfCondition(iss);

```

Figure 8: *processBlock* function (part 1)

```
            rightBlockNumber = currentBlockNumber + 1;
            adjMatrix[blockNumber][rightBlockNumber] = 1;
            edges[blockNumber][rightBlockNumber] = condition;
            myQueue.push(rightBlockNumber);

            //inFile.get();
            skipABlock(inFile);

            getline(inFile,line);
            istringstream iss1;
            iss1.str(line+" $");

            iss1 >> currentBlockNumber;
            iss1 >> word;

            if(word == "else")
            {
                leftBlockNumber = currentBlockNumber;
                adjMatrix[blockNumber][leftBlockNumber] = 1;
                edges[blockNumber][leftBlockNumber] = reverseCondition(condition);
                myQueue.push(leftBlockNumber);

                skipABlock(inFile);

                inFile >> currentBlockNumber;

                adjMatrix[rightBlockNumber][currentBlockNumber] = 1;
                adjMatrix[leftBlockNumber][currentBlockNumber] = 1;
                myQueue.push(currentBlockNumber);

            }
            else
```

Figure 9: *processBlock* function (part 2)

```
            else
            {
                adjMatrix[rightBlockNumber][currentBlockNumber] = 1;
                adjMatrix[blockNumber][currentBlockNumber] = 1;
                edges[blockNumber][currentBlockNumber] = reverseCondition(condition);
                myQueue.push(currentBlockNumber);


            }

            if(exitBlockNumber)
            {
                adjMatrix[blockNumber][exitBlockNumber] = 0;
                adjMatrix[currentBlockNumber][exitBlockNumber] = 1;
            }

            closeinputFile(inFile);

            break;


        }
        else if(currentBlockNumber == blockNumber && word == "for")
        {
            condition = parseForCondition(iss);

            rightBlockNumber = currentBlockNumber + 1;
            adjMatrix[blockNumber][rightBlockNumber] = 1;
            edges[blockNumber][rightBlockNumber] = condition;
            myQueue.push(rightBlockNumber);

```

Figure 10: *processBlock* function (part 3)

```
            skipABlock(inFile);

            getline(inFile,line);
            istringstream iss1;
            iss1.str(line+" $");

            iss1 >> currentBlockNumber;

            adjMatrix[rightBlockNumber][currentBlockNumber] = 1;
            adjMatrix[blockNumber][currentBlockNumber] = 1;
            edges[blockNumber][currentBlockNumber] = reverseCondition(condition);
            myQueue.push(currentBlockNumber);

            if(exitBlockNumber)
            {
                adjMatrix[blockNumber][exitBlockNumber] = 0;
                adjMatrix[currentBlockNumber][exitBlockNumber] = 1;
            }

            closeinputFile(inFile);

            break;

        }
        else if(currentBlockNumber == blockNumber && word == "while")
        {
            condition = parseWhileCondition(iss);

            rightBlockNumber = currentBlockNumber + 1;
            adjMatrix[blockNumber][rightBlockNumber] = 1;
            edges[blockNumber][rightBlockNumber] = condition;
```

Figure 11: *processBlock* function (part 4)

```
                rightBlockNumber = currentBlockNumber + 1;
                adjMatrix[blockNumber][rightBlockNumber] = 1;
                edges[blockNumber][rightBlockNumber] = condition;
                myQueue.push(rightBlockNumber);

                skipABlock(inFile);

                getline(inFile,line);
                istringstream iss1;
                iss1.str(line+" $");

                iss1 >> currentBlockNumber;

                adjMatrix[rightBlockNumber][currentBlockNumber] = 1;
                adjMatrix[blockNumber][currentBlockNumber] = 1;
                edges[blockNumber][currentBlockNumber] = reverseCondition(condition);
                myQueue.push(currentBlockNumber);

                if(exitBlockNumber)
                {
                    adjMatrix[blockNumber][exitBlockNumber] = 0;
                    adjMatrix[currentBlockNumber][exitBlockNumber] = 1;
                }

                closeinputFile(inFile);

                break;


            }

        }
    }
```

Figure 12: *processBlock* function (part 5)

```
void solve()
{
    printString("Solution for all paths:");
    map <string, constraint*> myMap;
    map <string, constraint*> :: iterator it;

    for(int i=0; i<allPaths.size(); i++)
    {
        for(int j=0; j<constraints[i].size(); j++)
        {

            constraint* c = constraints[i][j];

            it = myMap.find(c->leftOperand);
            if(it == myMap.end())
            {
                myMap[c->leftOperand] = c;
            }
            else
            {
                string previous = it->second->operatorr;
                string current = c->operatorr;

                if(previous == ">")
                {
                    if(current == ">")
                    {
                        it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);

                    }
                    else if (current == ">=")
                    {
                        it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);

                    }
                    else if (current == "<")
                    {
                        if((c->rightOperand - it->second->rightOperand) >= 2)
                        {
                            it->second->rightOperand = (it->second->rightOperand + c->rightOperand) / 2;
```

Figure 13: *solve* function (part 1)

```
                it->second->rightOperand = (it->second->rightOperand + c->rightOperand) / 2;
                it->second->operatorr = "==";
            }
            else
            {
                it->second->rightOperand = INF;
                it->second->operatorr = "==";
            }

        }
        else if (current == "<=")
        {
            if((c->rightOperand - it->second->rightOperand) >=1 )
            {
                it->second->rightOperand = c->rightOperand;
                it->second->operatorr = "==";
            }
            else
            {
                it->second->rightOperand = INF;
                it->second->operatorr = "==";
            }


        }
        else if (current == "==")
        {
            int x = c->rightOperand;
            if(x > it->second->rightOperand)
            {
                it->second->operatorr = c->operatorr;
                it->second->rightOperand = x;
            }
            else
            {
                it->second->operatorr = c->operatorr;
                it->second->rightOperand = INF;
            }

        }
```

Figure 14: *solve* function (part 2)

```
        else if (current == "!=")
        {
            it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);
        }

    }
    else if (previous == ">=")
    {
        if(current == ">")
        {
            it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);

        }
        else if (current == ">=")
        {
            it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);

        }
        else if (current == "<")
        {
            if((c->rightOperand - it->second->rightOperand) >=1 )
            {
                it->second->rightOperand = it->second->rightOperand;
                it->second->operatorr = "==";
            }
            else
            {
                it->second->rightOperand = INF;
                it->second->operatorr = "==";
            }

        }
        else if (current == "<=")
        {
            if((c->rightOperand - it->second->rightOperand) >=0 )
            {
                it->second->rightOperand = it->second->rightOperand;
                it->second->operatorr = "==";
            }
```

Figure 15: *solve* function (part 3)

```
        else
        {
            it->second->rightOperand = INF;
            it->second->operatorr = "==";
        }


    }
    else if (current == "==")
    {
        int x = c->rightOperand;
        if(x >= it->second->rightOperand)
        {
            it->second->operatorr = c->operatorr;
            it->second->rightOperand = x;
        }
        else
        {
            it->second->operatorr = c->operatorr;
            it->second->rightOperand = INF;
        }

    }
    else if (current == "!=")
    {
        it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);
    }

}
else if (previous == "<")
{
    if(current == ">")
    {
        if((it->second->rightOperand - c->rightOperand) >= 2)
        {
            it->second->rightOperand = (it->second->rightOperand + c->rightOperand) / 2;
            it->second->operatorr = "==";
        }
```

Figure 16: *solve* function (part 4)

```
    else
    {
        it->second->rightOperand = INF;
        it->second->operatorr = "==";
    }

}
else if (current == ">=")
{
    if((it->second->rightOperand - c->rightOperand) >= 1)
    {
        it->second->rightOperand =  c->rightOperand;
        it->second->operatorr = "==";
    }
    else
    {
        it->second->rightOperand = INF;
        it->second->operatorr = "==";
    }


}
else if (current == "<")
{
    it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
}
else if (current == "<=")
{
    it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
}
else if (current == "==")
{
    int x = c->rightOperand;
    if(x < it->second->rightOperand)
    {
        it->second->operatorr = c->operatorr;
        it->second->rightOperand = x;
    }
```

Figure 17: *solve* function (part 5)

```
                }
                else
                {
                    it->second->operatorr = c->operatorr;
                    it->second->rightOperand = INF;
                }
            }
            else if (current == "!=")
            {
                it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
            }

        }
        else if (previous == "<=")
        {
            if(current == ">")
            {
                if((it->second->rightOperand - c->rightOperand) >= 1)
                {
                    it->second->rightOperand = it->second->rightOperand;
                    it->second->operatorr = "==";
                }
                else
                {
                    it->second->rightOperand = INF;
                    it->second->operatorr = "==";
                }
            }
            else if (current == ">=")
            {
                if((it->second->rightOperand - c->rightOperand) >= 0)
                {
                    it->second->rightOperand = it->second->rightOperand;
                    it->second->operatorr = "==";
                }
                else
                {
                    it->second->rightOperand = INF;
                    it->second->operatorr = "==";
                }
```

Figure 18: *solve* function (part 6)

```
    else if (current == "<")
    {
        it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
    }
    else if (current == "<=")
    {
        it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
    }
    else if (current == "==")
    {
        int x = c->rightOperand;
        if(x <= it->second->rightOperand)
        {
            it->second->operatorr = c->operatorr;
            it->second->rightOperand = x;
        }
        else
        {
            it->second->operatorr = c->operatorr;
            it->second->rightOperand = INF;
        }
    }
    else if (current == "!=")
    {
        it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
    }

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

    }
```

Figure 19: *solve* function (part 7)

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

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

Figure 20: *solve* function (part 8)

```
        }
        else if (previous == "!=")
        {
            if(current == ">")
            {
                it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);
            }
            else if (current == ">=")
            {
                it->second->rightOperand = max(it->second->rightOperand, c->rightOperand);
            }
            else if (current == "<")
            {
                it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
            }
            else if (current == "<=")
            {
                it->second->rightOperand = min(it->second->rightOperand, c->rightOperand);
            }
            else if (current == "==")
            {
                int x = c->rightOperand;
                if(x == it->second->rightOperand)
                {
                    it->second->operatorr = c->operatorr;
                    it->second->rightOperand = x;
                }
                else
                {
                    it->second->operatorr = c->operatorr;
                    it->second->rightOperand = INF;
                }


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

Figure 21: *solve* function (part 9)

```
                        }
                    }

                }
            }


        }

        cout << "Path " << i+1 << ": ";
        for(it = myMap.begin(); it!=myMap.end(); it++)
        {
            constraint* c = it->second;
            int soln = solver(c);

            if(soln != INF)
                cout << it->first << ": " << soln <<", ";
            else
                cout << "No solution exist";
        }
        cout << endl;
        myMap.clear();
    }

}
```

Figure 22: *solve* function (part 10)

```
int solver(constraint* c)
{
    int solution = 0;
    string relationalOperator [] = {">", "<" , "<=", ">=", "==" , "!=" };
    int solutionMaker[] = {+1, -1 , -1, +1, 0 , +5 };

    string leftOperand = c->leftOperand;
    string operatorr = c->operatorr;
    int rightOperand = c->rightOperand;

    for (int i = 0; i < 6; ++i)
    {
        if(operatorr == relationalOperator[i])
        {
            solution = rightOperand + solutionMaker[i];
            break;
        }
    }

    return solution;

}
```

Figure 23: *solver* function