SPL-1 Project Report,2022

Control Flow Graph

SE 305 : Software Project Lab

Submitted by:

Mst. Fareya Azam

BSSE Roll: 1331

Session: 2020-2021

Supervised by:

Dr. Sumon Ahmed

Designation: Associate Professor

Institute of Information Technology

Supervisor's Signature…………………………………..

Institute of Information Technology

University of Dhaka

Date: 21-05-2023

Contents

1. Introduction:

Control Flow Graph (CFG) analysis is a fundamental technique used in software engineering to understand the control flow of a program. By constructing a CFG, developers can visualize the flow of execution through different statements, loops, and conditions, gaining valuable insights into program behavior, identifying potential issues, and optimizing code.In this project, we have developed a program that performs control flow analysis on source code files. The control flow analysis involves several steps. Firstly, the input source code file is read and divided into lines. Each line is processed to remove comments, handle special characters, and tokenize the code into individual words or symbols. This preprocessing step ensures that the control flow graph accurately represents the code's structure, excluding any irrelevant information.Next, the program constructs the control flow graph by traversing the processed code line by line. Control flow graph nodes are created for each statement or block of code, and directed edges represent the control flow between these nodes. By analyzing the control flow graph, we can identify loops, conditionals, decision points, and the overall structure of the program.

Additionally, our program incorporates various functions to identify keywords, operators, integers, and other elements within the code. These functions assist in labeling the control flow graph nodes, providing additional information about the program's structure and aiding in program comprehension.This project also analyze the control flow graph through BFS and DFS algorithm and gives us valuable information such as parent nodes, discovery time, finishing time, and distance from a source node. Using control flow analysis techniques, we can enhance our understanding of complex programs, identify potential issues of our program.

1.1 Background study:

I had to study the following topics for the project:

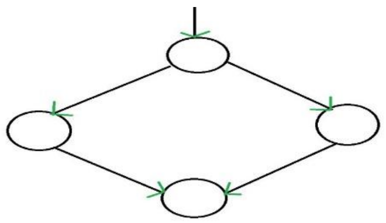
Types of statement in C:

Statements are the executable parts of the program that will do some actions. Types of statements that we need are:

a. Selection statement:

1.if

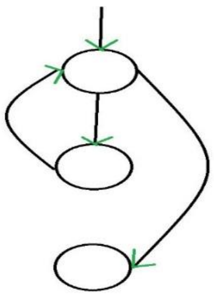
2.else if
3.else
4.Switch

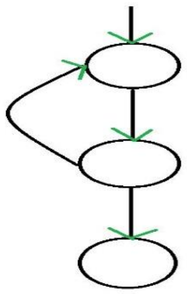

If-then-else

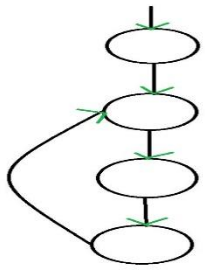
b. Iterative statement:

Loops:

- Do loop
- Do-While loop
- For loop


while


do-while


for

C. Jump Statements :

break, goto, continue.

D. Expressive statement:

Y=X+10;

For the implementation of the control flow graph project, there are several key areas that you will need to study and understand. Here are some important aspects to focus on:

Lexical Analysis:

Study the concepts and techniques related to lexical analysis, which involves breaking down the source code into tokens or meaningful units. Learn about tokenization, handling whitespace, special characters, and comments in the code. Understand how to identify and classify different types of tokens such as keywords, identifiers, operators, literals, and special symbols.

Parsing:

Gain knowledge of parsing technic to analyze the structure of source code. This helped me extracting meaningful information from the source code and identify control flow constructs like conditionals, loops etc.

Control Flow Analysis:

Study the principles and techniques of control flow analysis. Learn about the different types of control flow graphs, including structured control flow graphs, hierarchical control flow graphs, and extended control flow graphs. Understand how to model control dependencies, decision points, and loops in the control flow graph.

Graph Theory:

Develop a solid understanding of basic graph theory concepts. Study graph representations, traversal algorithms (depth-first search, breadth-first search), graph properties (connectivity, cycles). This knowledge will be crucial for analyzing the control flow graph.

By studying and gaining proficiency in these areas, I acquire the necessary knowledge and skills to implement the control flow graph project effectively.

1.2 Challenges

The challenges I had face while working on this project:

- **Identifying Control Flow Constructs:**
  Constructing a control flow graph based on stored nodes and edges make it more challenging to accurately identify control flow constructs. Mapping of nodes and edges to

their corresponding control flow constructs, such as conditionals, loops, becomes more error prone. Ensuring the correctness and accuracy of the generated control flow graph becomes a priority, requiring careful examination of the nodes and edges to their corresponding control flow structures. It was the toughest part of my project.

- **Handling comments:**
  The source code may have single line comments (//) or block comments (/*….*/) and unnecessary white spaces . Handling different comment types and ensuring their proper removal was challenging. Failure to remove comments can lead to incorrect control flow representation .

- **Handling Complex Code Patterns:**
  Dealing with complex code patterns, such as deeply nested loops or conditionals was challenging. Extracting meaningful control dependencies and decision points from such code patterns required additional effort and careful analysis of the code logic.

- **Debugging codes:**
  Debugging programs while observing multiple variables and their interactions with multiple methods was challenging.

2. Project Overview

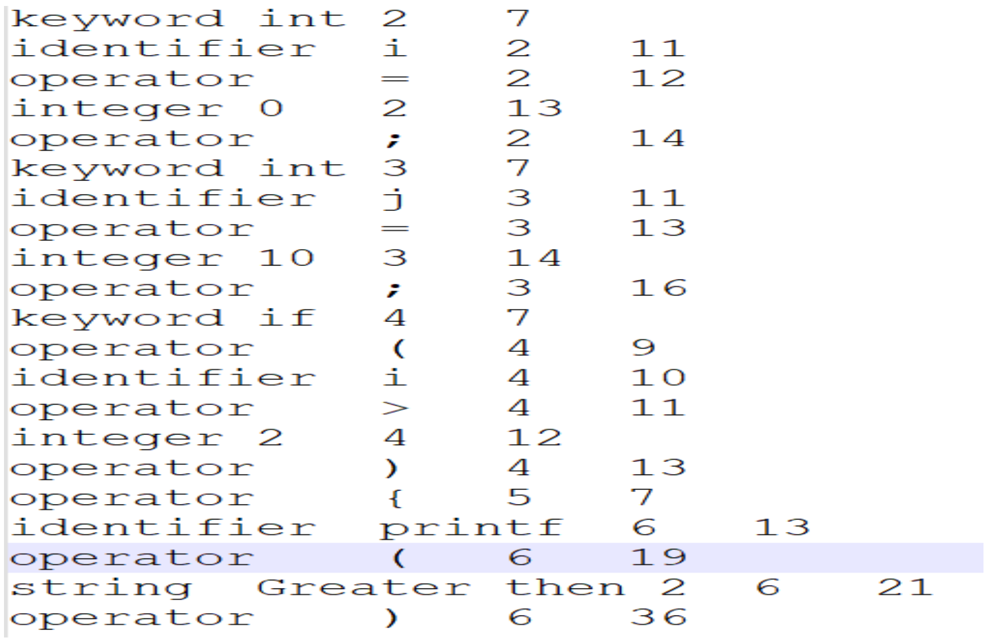
The project focuses on constructing a control flow graph (CFG) for a given codebase using a brute force approach. The CFG represents the flow of control within a program by modeling its basic blocks and the relationships between them. The primary objective of this project is to construct a CFG by exhaustively examining the code and explicitly creating edges between nodes to represent control flow. The main steps involved in this approach are:

- code analysis
- basic block identification
- edge creation.

Code analysis:

This phase involves parsing the code and extracting relevant information. This includes identifying keywords, such as conditionals (if, else if, else), loops (for, while), and other control flow constructs. Additionally, the analysis phase involves tracking braces, parentheses, and other delimiters to accurately identify code blocks. In the **tokenization.cpp** file I have done this work. Let's describe the functionalities of this file:

A struct '**eachlinestruct**' is defined, which represents each line of code. It contains two members: **'line_no**' (line number) and 'word' (the actual line of code). An array of **'eachlinestruct**' called **'each_line**' is declared, which can hold up to 2000 lines of code.The code defines several helper functions to identify different types of tokens: '**isOperator()'** checks if a given character is an operator.The '**checkArithmaticOperator()'** function checks if a given character is an arithmetic operator. **'isDigit()'** checks if a given character is a digit. **'isInteger()'** checks if a string represents an integer. '**isDouble()'** checks if a string represents a double (floating-point number). '**PrintString()'** creates a token string for identifiers, integers, and doubles. '**keyword_Identifier()'** identifies keywords or identifiers and creates token strings accordingly. The '**codeLine()'** function takes the input source code and splits it into lines, storing each line along with its line number in the `each_line` array. It returns the total number of lines.The '**tokenization()'** function performs the actual tokenization process. It opens a file named "TokenFile.txt" for writing the tokens. It iterates over each line of code, tokenizes it, and writes the tokens to the file.The '**commentsRemoval()'** function removes single-line and multi-line comments from the source code using simple string manipulation. It reads the input source code from a file and returns the modified code without comments. The '**sourceCodeMakeToken()'** function is the entry point of the program. It calls '**commentsRemoval()'** to remove comments from the source code and then calls '**codeLine()'** and '**tokenization()**' to perform tokenization on the modified code. It also displays the modified code without comments. Finally, the '**tokenizationCall()'** function is provided as an example of how to call the tokenization process. It calls '**sourceCodeMakeToken()'** and prints a success message.



```
keyword int  2    7
identifier   i    2    11
operator     =    2    12
integer 0    2    13
operator     ;    2    14
keyword int  3    7
identifier   j    3    11
operator     =    3    13
integer 10   3    14
operator     ;    3    16
keyword if   4    7
operator     (    4    9
identifier   i    4    10
operator     >    4    11
integer 2    4    12
operator     )    4    13
operator     {    5    7
identifier   printf   6    13
operator     (    6    19
string  Greater then 2   6    21
operator     )    6    36
```

Identifying basic blocks:

A basic block represents a sequence of instructions with a single entry point and a single exit point. This step involves partitioning the code into distinct blocks based on control flow constructs, such as conditionals and loops etc.In the makecfg.cpp file , the function '**analyze()**' analyzes the code to identify control flow structures and strores information in the 'struct_info'vector. Here are several functions that help in finding basic blocks:

The function **'ifline**' checks if a given line number belongs to a control flow structure.The function '**findStarting**' returns the index of the structure with the given line as its starting line. The function '**findEndingLine**' finds the ending line number of a control flow structure.

Edge creating:

The '**edgeCreating()**' function create edges between nodes based on the control flow structure information. The edges represent the flow of control from one block to another. For example, an edge is created from a conditional block to the subsequent block based on the evaluation of the condition. Similarly, edges are created between blocks inside loops to represent the loop's iterative behavior.

```
if ( struct_info[ id ].keyword == "for" || struct_info[ id ].keyword == "while" )
{
   edge[ node ].push_back( node + 1 );
   edge[ node ].push_back( struct_info[ id ].end_line + 1 );

   edgeCreating( struct_info[ id ].line_begin + 1, struct_info[ id ].end_line - 1 );

   edge[ struct_info[ id ].end_line ].push_back( node );
   node = struct_info[ id ].end_line + 1;
}

else if ( struct_info[ id ].keyword == "if" || struct_info[ id ].keyword == "else if" || struct_info[ id ].keyword == "else" )
{
   int last = findEndingLine( struct_info[ id ].line_begin ) + 1;
   edge[ node ].push_back( node + 1 );
   edge[ node ].push_back( struct_info[ id ].end_line + 1 );
```

Throughout the process, the project leverages iterative algorithms and techniques to handle nested control flow structures. Recursive function calls are used to analyze and create edges within nested conditionals or loops. There are some helper functions that helped in creating adjacency list , as the representation of the control flow graph. The '**createNode'** function creates a new node with the given data.The '**addEdge**' function adds an edge between two nodes in the adjacency list.

Finally I implemented **BFS** and **DFS** algorithm for analyzing the control flow graph and print the traversal results including color ,parent and finishing time.

3. User manual

Here are the steps that will guide you to effectively run the project:

1. Firstly ensure that you have the necessary dependencies installed, such as a C++ compiler and the required libraries.

2.Before you can analyze a control flow graph, you need to generate the control flow graph from your code. So, before running the project we need to create a input C file that will contain the source code, for which it will create a CFG. Open the tokenization.cpp file and make sure to store the tokens into TokenFile.txt file.

3.Generate the control flow graph using makeCFG.cpp file.

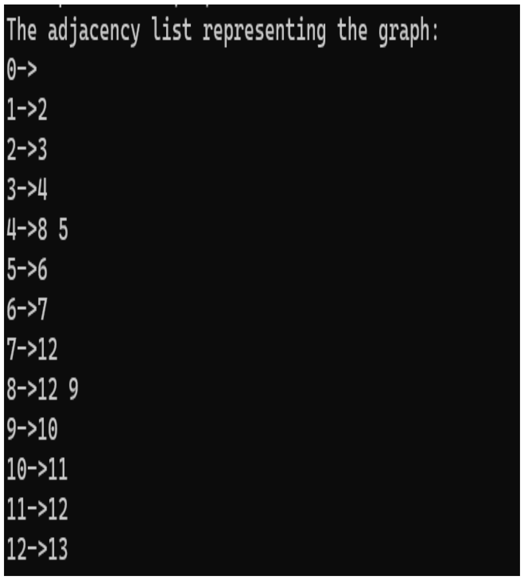
4.The Control Flow Graph Analyzer provides two analysis algorithms:

Depth First Search (DFS) and Breadth First Search (BFS). These algorithms can help you gather information about the control flow in your code, such as discovery time, finishing time, parent nodes, and distance from a source node.

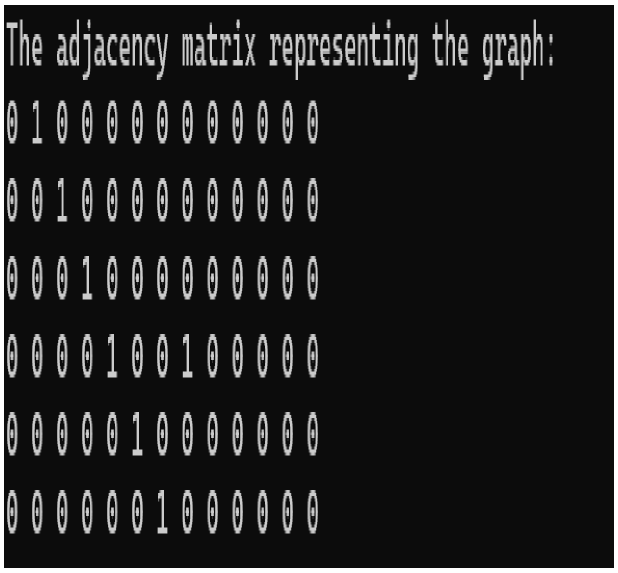
-The analyzer will display the generated control flow graph as an adjacency list and an adjacency matrix.

- It will then perform a Depth First Search (DFS) analysis on the control flow graph and display the results, including the parent nodes, discovery time, and finishing time for each node.

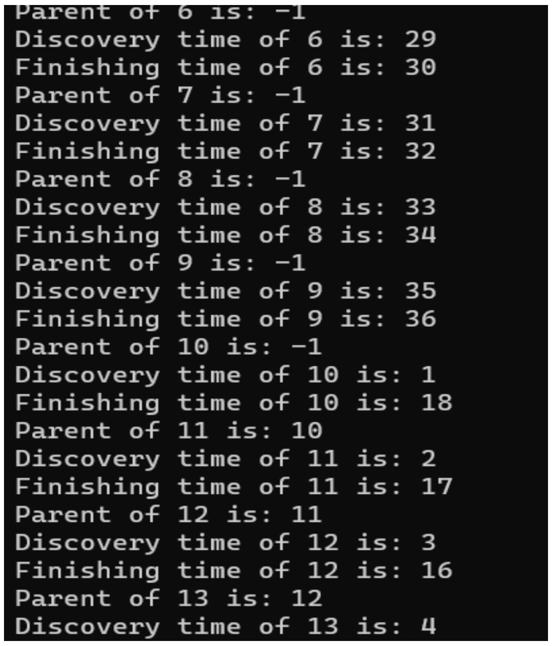
- After the DFS analysis, the analyzer will perform a Breadth First Search (BFS) analysis on the control flow graph and display the results, including the parent nodes and distance from a source node for each node.



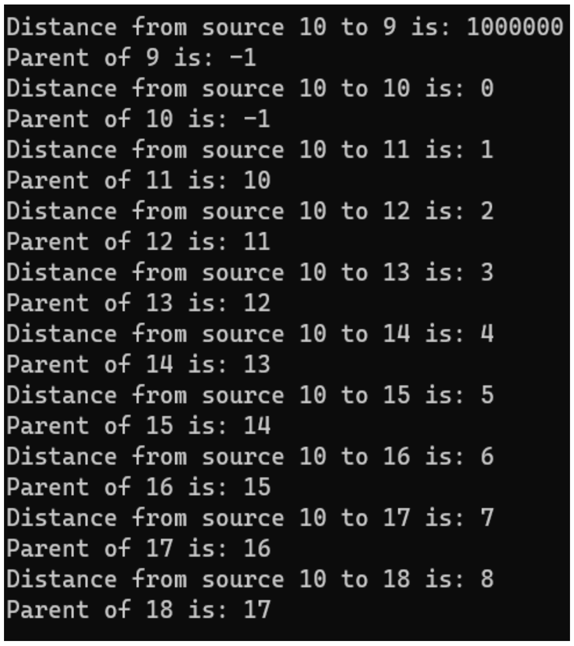
```
The adjacency list representing the graph:
0->
1->2
2->3
3->4
4->8 5
5->6
6->7
7->12
8->12 9
9->10
10->11
11->12
12->13
```



```
The adjacency matrix representing the graph:
0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0
```



```
Parent of 6 is: -1
Discovery time of 6 is: 29
Finishing time of 6 is: 30
Parent of 7 is: -1
Discovery time of 7 is: 31
Finishing time of 7 is: 32
Parent of 8 is: -1
Discovery time of 8 is: 33
Finishing time of 8 is: 34
Parent of 9 is: -1
Discovery time of 9 is: 35
Finishing time of 9 is: 36
Parent of 10 is: -1
Discovery time of 10 is: 1
Finishing time of 10 is: 18
Parent of 11 is: 10
Discovery time of 11 is: 2
Finishing time of 11 is: 17
Parent of 12 is: 11
Discovery time of 12 is: 3
Finishing time of 12 is: 16
Parent of 13 is: 12
Discovery time of 13 is: 4
```



```
Distance from source 10 to 9 is: 1000000
Parent of 9 is: -1
Distance from source 10 to 10 is: 0
Parent of 10 is: -1
Distance from source 10 to 11 is: 1
Parent of 11 is: 10
Distance from source 10 to 12 is: 2
Parent of 12 is: 11
Distance from source 10 to 13 is: 3
Parent of 13 is: 12
Distance from source 10 to 14 is: 4
Parent of 14 is: 13
Distance from source 10 to 15 is: 5
Parent of 15 is: 14
Distance from source 10 to 16 is: 6
Parent of 16 is: 15
Distance from source 10 to 17 is: 7
Parent of 17 is: 16
Distance from source 10 to 18 is: 8
Parent of 18 is: 17
```

DFS output BFS output

Here is my github link where you can find the project: https://github.com/fareya22/SPL-1

5. Interpreting the Results:

The results of the analysis algorithms provide valuable information about the control flow in your code. Here's a brief explanation of the key information:

- Parent of a node: Indicates the parent node in the control flow graph.

- Discovery time: Represents the time at which a node was discovered during the DFS traversal.

- Finishing time: Represents the time at which a node finished its exploration during the DFS traversal.

- Distance from a source node: Represents the minimum number of edges required to reach a node from the source node during the BFS traversal.

6. Making Use of the Results:

The results of the analysis can help you understand the control flow and relationships between different parts of your code. You can utilize this information for various purposes, such as:

- Identifying dependencies and relationships between functions or modules.

- Detecting loops or cycles in the control flow.

- Analyzing the complexity of the code based on the number of nodes, edges, and distance from a source node.

7. Troubleshooting:

If you encounter any issues or errors while using the Control Flow Graph Analyzer, consider the following troubleshooting steps:

- Ensure that you have the necessary dependencies installed, including a C++ compiler and the required libraries.

- Verify that your code files are located in the same directory as the `makecfg.cpp` and `main.cpp` files.

The Control Flow Graph Analyzer is a powerful tool for understanding and analyzing the control flow in your code. By following the steps outlined in this user manual, you can effectively generate the control flow graph and utilize the analysis algorithms to gain insights into the structure and behavior of your code.

4. Conclusion:

In the project, the resulting control flow graph provides a visual representation of the program's control flow and serves as a useful tool for understanding program behavior and reasoning about its execution paths. By employing a brute force approach to build the control flow graph, the project emphasizes a comprehensive analysis of the codebase. While this approach may be computationally intensive for large codebases, it ensures a thorough representation of control flow and enables a detailed understanding of the program's behavior.

Overall, this project is very interesting to me as it grows a deeper understanding of the control flow within the code. By generating and analyzing the control flow graph, I can gain insights into the structure, dependencies, and relationships between different parts of the code. With the ability to perform Depth First Search (DFS) and Breadth First Search (BFS) analysis algorithms, the analyzer provides valuable information such as parent nodes, discovery time, finishing time, and distance from a source node. This information can be used to identify code dependencies, detect loops or cycles, and analyze the complexity of the codebase. I hope ,in future ,I can add more features to this project.

5. References:

https://groups.seas.harvard.edu/courses/cs153/2018fa/lectures/Lec17-CFG-dataflow.pdf

https://www.geeksforgeeks.org/software-engineering-control-flow-graph-cfg/

https://en.wikipedia.org/wiki/Control-flow_graph

https://www.javatpoint.com/depth-first-search-algorithm

https://www.scaler.com/topics/bfs-program-in-c/