Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

# Analysis of the Scope of Variables Using the Graph Theory

**Hoang Nam Tran**

Supervisor: RNDr. Ingrid Nagyová, Ph.D.
May 2024

# BACHELOR‘S THESIS ASSIGNMENT

## I. Personal and study details

| | | | |
|---|---|---|---|
| Student's name: | **Tran Hoang Nam** | Personal ID number: | **499160** |
| Faculty / Institute: | **Faculty of Electrical Engineering** | | |
| Department / Institute: | **Department of Cybernetics** | | |
| Study program: | **Open Informatics** | | |
| Specialisation: | **Artificial Intelligence and Computer Science** | | |

## II. Bachelor’s thesis details

Bachelor’s thesis title in English:

**Analysis of the Scope of Variables Using the Graph Theory**

Bachelor’s thesis title in Czech:

**Analýza rozsahu proměnných s využitím teorie grafů**

Guidelines:

The aim of this work is to design and implement a system for detection of errors in variable scopes in C programs. The work assumes the use of a graph coloring method that can detect the necessity of coexistence of variables in memory.
1. Give a definition the scope of variable in the program. Describe the errors in programming related to variable scope.
2. Familiarize yourself with the method of determining the variable scope using graph theory.
3. Understand register allocation algorithms that use graph coloring methods. Apply these procedures to the variable scope problem and describe the adjusted algorithms.
4. Design and implement a system for analyzing the scope of variables in a C program. Display the analysis output clearly for the user.
5. Perform a system testing on the tasks submitted to BRUTE. Analyze the frequency of errors in the scope of variables in the student codes.

Bibliography / sources:

[1] Palúch, S. (2008). Algoritmická teória grafov. ŽILINSKÁ UNIVERZITA V ŽILINE.
[2] Priya, Bala C. (2021) Variable Scope in C – Local and Global Scope Explained. freeCodeCamp. Dostupné z: https://www.freecodecamp.org/news/scope-of-variables-in-c-local-and-global-scope-explained/
[3] Cormen, H. T., Leiserson, C. E., Rivest, R. L. & Stein, C. (2005). Introduction to algorithms (3. vyd.). Massachusetts Institute of Technology. https://edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf
[4] Briggs, P., Cooper, Keith D., Torczon, L. (1994) Improvements to Graph Coloring Register Allocation. ACM Trans. Program. Lang. Syst. 1994, roc. 16, c. 3, s. 428–455. ISSN 0164-0925. Dostupné z DOI: 10.1145/177492.177575.
[5] Harrold, M. J., Rothermel, G. & Tech, A. O. G. (2002) Notes on Representation and Analysis of Software. https://ics.uci.edu/~lopes/teaching/inf212W12/readings/rep-analysis-soft.pdf

Name and workplace of bachelor’s thesis supervisor:

**RNDr. Ingrid Nagyová, Ph.D. Center for Software Training FEE**

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **06.02.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

| | | |
|---|---|---|
| RNDr. Ingrid Nagyová, Ph.D.<br>Supervisor’s signature | prof. Dr. Ing. Jan Kybic<br>Head of department’s signature | prof. Mgr. Petr Páta, Ph.D.<br>Dean’s signature |

## III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

| | |
|---|---|
| . | |
| Date of assignment receipt | Student’s signature |

## Acknowledgements

I would like to thank my supervisor RNDr. Ingrid Nagyová, Ph.D., for allowing me to create this app and lead my thesis.

## Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 24. 05. 2024

Hoang Nam Tran

## Abstract

This thesis deals with analyzing the scope of variables using the graph theory. More concretely, it is a part of the theory for graph coloring and the construction of graphs for coloring. Then, it describes the theory behind building a control flow graph and how to build an interference graph from the control flow graph to use as visualization for variable scope. Finally, it describes the development and use of the resulting app, which shows, as an output, these two graphs to visualize the variable scope.

**Keywords:** graphs, graph coloring, variable analysis

**Supervisor:** RNDr. Ingrid Nagyová, Ph.D.

## Abstrakt

Tato práce se zabývá analýzou rozsahu proměnných s využitím teorie grafů. Konkrétně je popsána teorie k barvení grafů a konstrukce grafu na barvení. Poté popisuje teorii za konstrukcí grafu toku řízení a grafu interference proměnných. Závěrem je popis vývoje a použití aplikace, která má jako výstup tyto dva grafy, za účelem popisu rozsahu proměnných. . .

**Klíčová slova:** grafy, barvení grafu, analýza proměnných

**Překlad názvu:** Analýza rozsahu proměnných s využitím teorie grafů

# Contents

# Figures

# Tables

# Chapter 1

# Introduction

The scope of a variable refers to the area of the source code where the variable may be used, modified, or declared. Managing this variable scope is important for code maintainability, as an excessive variable scope may lead to issues such as limited reusability of variable names, variable shadowing (where an inner scope variable hides an outer scope variable with the same name), and the potential for overlooked declared variables, which can introduce difficult-to-detect bugs.

Graph coloring is now used in register allocation to determine necessary variable storage and enable the reuse of a register for storage. For this purpose an interference graph is colored, where each node in the graph represents a variable and an edge represents a shared lifetime of the two connected variables. The aim of this thesis is to develop an application to visualize the variable scope through the before-mentioned interference graph and its coloring. Providing the users with this graph may help with easier identification of variables, which may need scope optimization.

The thesis begins with a theory for graph and graph coloring, as well as some greedy algorithms for graph coloring. Following this, we will describe variable scope in more detail, highlighting some issues caused by bad scope management. We will then explain the theory behind constructing an interference graph to determine the minimal needed scope of a variable. After which we will provide examples of other possible solutions for optimizing the variable scope. Finally, we will describe the implementation of the application that will provide the user with the information for the variable scope optimization, and we will test our application on a dataset of homework from the subject PRP to determine the information our application can provide and the limitations of our application.

# Chapter 2

# Graphs and graph coloring

This chapter serves as a reminder for definitions related to graphs and graph coloring, which will be used in the following chapters.

## 2.1 Graph

**Definition 2.1.** *A graph is an ordered triple (V(G),E(G),$\varepsilon$(G)), where V(G) is an non-empty finite set of vertices, E(G) is a finite set of edges and $\varepsilon$(G) is an incidence mapping which assigns to an edge e $\epsilon$ E(G) a pair of vertices (x,y). A graph is considered complete when every pair of distinct vertices has an incidence mapping to an edge. [1, 2]*

**Definition 2.2.** *A graph G' is a subgraph of graph G if it can be created by leaving out some (or none) vertices or edges of graph G while still maintaining the properties of a graph. [1]*

**Definition 2.3.** *Two vertices are adjacent or neighbors if an edge connects them. [2]*

**Definition 2.4.** *The degree of a vertex v is the number of edges incident with the vertex v. [1]*

**Definition 2.5.** *A plane or planar graph is a type of graph that can be drawn in the plane without its edges intersecting. Such a drawing is called planar embedding. [2]*

**Definition 2.6.** *A bipartite graph is a graph whose vertices can be partitioned into two distinct sets (X, Y) so that each edge has one end in X and the other in Y.*

**Definition 2.7.** *A complete bipartite graph is one where every pair x $\epsilon$ X and y $\epsilon$ Y has an edge.*

## 2.2 Graph coloring

**Definition 2.8.** *A coloring of a graph $G$ is an assignment of values to vertices from a set $C$ (so-called colors) so that no two neighboring vertices share a color. Graph $G$ is considered $k$-colored if a coloring using $k$ colors exists.*

**Definition 2.9.** *The chromatic number of a graph $G$ is the smallest number of elements needed to color a graph and is denoted $\chi(G)$.*

### 2.2.1 Graph coloring algorithms

This section will introduce a group of greedy algorithms that achieve graph coloring in a relatively short runtime.

#### Greedy algorithm

This algorithm selects an uncolored vertex and assigns it the smallest possible color, depending on its neighbors. The number of colors used depends on the order of vertices to color. [9]

**Algorithm 1** Greedy algorithm

```
$color \leftarrow array(V.size(), 0)$
for $v \in \mathcal{V}$ do
    $NeighborColors \leftarrow []$
    for $neighbor \in$ v.neighbors do
        if $Color[neighbor] \neq 0$ then
            NeighborColors.add(color[neighbor])
        end if
    end for
    $colorV \leftarrow 0$
    while $color[v] = 0$ do
        $colorV \leftarrow colorV + 1$
        if colorV not in NeighborColors then
            $color[v] \leftarrow colorV$
        end if
    end while
end for
```

#### Saturation Largest First / Dsatur algorithm

The saturation largest first algorithm selects the vertex with highest degree of saturation to color first. This degree is defined as the number of neighbors that were already colored in previous iterations. In case of a tie between vertices, it will choose the one with the most neighbors.[8]

**Algorithm 2** Dsatur algorithm

```
$color \leftarrow array(V.size(), 0)$
$satur \leftarrow array(V.size(), )$
$Q \leftarrow \{\}$
for $v \in \mathcal{V}$ do
    Q.add(v, satur[v], v.degree)
end for
while $Q$ not empty do
    $v \leftarrow Q$.pop()
    if $Color[v] \neq 0$ then
        $NeighborColors \leftarrow []$
        for $neighbor \in$ v.neighbors do
            if $Color[neighbor] \neq 0$ then
                NeighborColors.add(color[neighbor])
            end if
        end for
    end if
    $colorV \leftarrow 0$
    while $color[v] = 0$ do
        $colorV \leftarrow colorV + 1$
        if colorV not in NeighborColors then
            $color[v] \leftarrow colorV$
        end if
    end while
    for $neighbor \in$ v.neighbors do
        if neighbor not in Color and colorV not in satur[neighbor] then
            satur[neighbor].add(colorV)
        end if
    end for
end while
```

### Recursive largest first algorithm

The recursive largest first algorithm selects a vertex with the highest degree, then it selects another vertex that shares the largest number of neighbors with the first one and is not its neighbor. Then, it selects another with the largest number of neighbors with both and is not a neighbor of either. This repeats itself until there are no vertices that are not neighbors of the selected vertices. This creates a list of vertices that share the same color and then repeats the above step on a subgraph without the vertices from the list.[9]

**Algorithm 3** Recursive largest first algorithm

```
$color \leftarrow array(V.size(), 0)$
$V' \leftarrow V$
$currentColor = 0$
**while** $V'$ not empty **do**
    $currentColor = currentColor + 1$
    $v \leftarrow$ vertex with largest degree in V'
    $Neighbors \leftarrow$ v.neighbors
    $color[v] \leftarrow currentColor$
    $V'$.remove(v)
    **while** exists a vertex that is not in Neighbors and is in V' **do**
        $v \leftarrow$ vertex with most neighbors in Neighbors
        $Neighbors$.add(v.neighbors)
        $color[v] \leftarrow currentColor$
        $V'$.remove(v)
    **end while**
**end while**
```

The following graph describes the performance of these algorithms on random graphs with vertex count N, where the probability of an edge between vertices is 0.5.

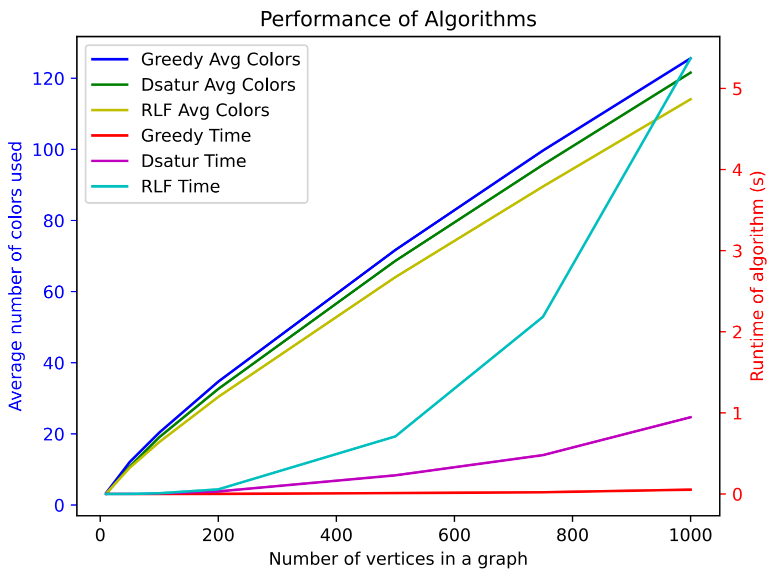


**Figure 2.1:** Performance of coloring algorithms on random graphsA.1

# Chapter 3

# Variable Scope

Variable scope is the block of code of the program where the variable is accessible, starting from the point when the variable is declared. It defines the visibility of a variable, determining where the variable can be modified. The visibility is bounded by the before-mentioned block of code and is important for the interaction between variables and reusing identifiers, meaning the same identifiers can denote a different variable at different points of code. There are four types of scope: function, block, function prototype, and file. [13]

## 3.1 Scope types

### 3.1.1 Function scope

Function scope is the only scope where the declaration of a variable is not important, and this variable can be accessed anywhere in the function and is only applied for labels and used in conjunction with goto statements. Labels are syntactically an identifier ended with ":" followed by a statement.[13]

For all other scopes, the placement of the declaration of the variable determines its scope.[13]

### 3.1.2 Block scope

A variable declared inside a block or as part of the list of parameters of a function definition has a block scope, and its scope ends with the end of the associated block.[13]

### 3.1.3 Function prototype scope

Function prototype scope applies to the declaration of functions, which are not definitions, applies to the list of parameters of the function, and ends with the end of the function declarator.[13]

### 3.1.4 File scope

File scope applies to variables declared outside of blocks or function parameters, and its scope with the end of the file. These variables are also known as global variables.
[13]

### 3.1.5 Shadowing of variables

If two variables share the same identifier there can be overlap of scopes, then one scope (the inner scope) will be a strict subset of the other (the outer scope). If so, the variable with the inner scope will hide the outer scope variable from its declaration to the end of the block. That means inside the inner scope, only the variable with the inner scope is accessible.[13]

### 3.1.6 Keywords affecting scope

Static and extern are two keywords that can affect the scope of a global variable or a function. These are used for sharing variables or functions between files, where the keyword extern marks a variable that is accessed from another file, and the static keyword limits the global variable or a function to its file only. Static keyword also denotes variables to be stored even past its scope, so, for example, the state of the variable is stored between function calls. [13]

```c
#include <stdio.h>

int x = 5; // File scope variable

int main() {
    printf("%d\n", x); // prints 5 from file scope variable

    int x = 2; // Block scope variable shadows file scope variable
    printf("%d\n", x); // prints 2 from block scope variable

    if (x > 1) {
        int x = 10; // Inner block scope variable
        printf("%d\n", x); // prints 10 from inner block scope variable
    }
    printf("%d\n", x); // prints 2 from block scope variable
    return 0;
}
```

**Figure 3.1:** Example of shadowing

## 3.2 Scope and storage of variables

The scope also affects the storage of objects when the object is guaranteed to exist and remembers its last stored value; there are three types of storage: static, automatic, and allocated. [13]

### 3.2.1 Static storage

Static memory storage is an object stored for the entire lifetime of a program and is either denoted with the static keyword, or the object is accessible globally.[13]

### 3.2.2 Automatic storage

Automatic storage is associated with the execution of a block and starts from the entry of a block until its end. This storage is used when the object is not static or global. A new object instance is initialized if the block is accessed recursively and not replaced.[13]

### 3.2.3 Allocated storage

Allocated storage is a manual memory allocation by calling calloc, malloc or realloc functions. If the allocation succeeds, the pointer returned from these functions is then used to access this memory and is assignable to any data type pointer. This allocation is stored in memory until it is manually deallocated. This means that even if the pointers to the allocation are lost or inaccessible, the memory is still occupied. That means the scope of a variable pointer can be smaller than its memory lifetime as it requires manual deallocation by the code.[13]

# Chapter 4

# Determining variable scope using graph theory

To analyze the variable scope, we need to know the order of execution of instructions in the program and possible branches or jumps inside the program. We will use control flow graphs to describe this order and use it to determine the scope of variables in a program, after which we will use these scopes to construct an interference graph to determine the efficiency of the placement of a variable.

**Definition 4.1.** *A control flow graph is a directed graph in which nodes represent basic blocks, and edges represent control flow paths between these blocks. [11]*

**Definition 4.2.** *A basic block is a linear sequence of program statements without a possibility of branching or stopping, with one leading statement as an entry point and its singular exit point as its last statement, either the exit of a program or the statement before the following leading statement. A block within a program can have multiple blocks that come before it. Another block or various blocks can follow it through directed edges unless it is a program-terminating block, which never has a successor. Additionally, a program entry block may have a predecessor outside its program.[10, 11]*

**Definition 4.3.** *An interference graph of variables is a graph where each node is a variable, and an edge represents a situation where both nodes need to be stored simultaneously.[11, 12]*

## Constructing a control flow graph

We must split the code into the basic blocks defined above to construct a control flow graph. First, we determine the set of leading statements. A leading statement can be defined as the entry point of a program, a statement that is an unconditional or conditional jump in a program, or a statement that immediately follows a statement that is an unconditional or conditional jump. Using this set, we construct the basic blocks, where each leader has a block consisting of the leader and all following statements up to but not including the next leader or end of the program. However, we might find it more convenient to use each statement as its own leader instead, meaning each statement would be its own block for future use during the construction of scope because it will provide a more precise scope of usage.[10, 11]

Now, we build edges between these blocks. An edge connects two blocks if the last statement of the first block unconditionally or conditionally jumps to the first statement of the second block or the second block immediately follows the first block, and the first block does not end with an unconditional jump statement. With this graph, we can now build an interference graph of variables. Finally, we add an entry node at the start and an exit node at the end. [10, 11]

## Building an interference graph

To build an interference graph, according to the source code, we must determine an area where the variable is necessary in memory. For this purpose, we will use the built control flow graph. We define two sets, IN and OUT, for each block. IN represents any variables coming from the preceding blocks, and OUT represents any variables going to the successors. We define two additional sets, USE and DEF, for each block except the exit node, where the program just terminates and does nothing else. USE is a set of variables that reference a variable initialized elsewhere and came from the preceding block. DEF is a set of variables initialized or defined in the block and before their use in the block. We calculate the USE and DEF sets for each block. As a consequence of our definitions, the USE set of a block must be part of its IN set, and any variables of the DEF set are not part of the IN set. Therefore, the equations for calculating the IN and OUT sets for each block are as follows:

$\text{IN[b]} = \text{USE[b]} \cup (OUT[b] - DEF[b])$

$\text{OUT[b]} = \bigcup s \in Successors\ of\ b\ IN[s]$ To prevent an undefined result from reading the IN set of the exit node, We define it as an empty set. Now that we have these sets, it is very simple, if a variable is in an OUT set of a block Bi and there is a path that leads to a USE set of another block Bj, which includes this variable, and it is not reassigned in a DEF set on that path, then it is alive in the block Bi[11, 12] The figures 4.1 show the source with the resulting CFG in 4.2 and the resulting interference graph in the 4.3.

From the interference graph in figure 4.3, we can see that the variables p and isPrime intersect with one another, however, if we look at the control flow graph or the source code, the early declaration of isPrime causes the intersection because it is actually needed at the earliest before the initialization of the for loop, as the previous if-else branching shows, that it doesn't need the variable isPrime. Still, it must deliver it to the for loop.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n;
    int isPrime = 0;
    int p = 4;
    printf("Enter␣a␣positive␣integer:␣");
    scanf("%d", &n);
    if(n < 0){
        printf("Error␣nonpositive␣number␣inputted.␣
           Example␣of␣a␣positive␣number:␣%d", p);
        return 100;
    }else if (n <= 1){
        printf("%d␣is␣a␣prime␣number.", n);
    }else{
         for ( int i = 2; i <= n / 2; i++) {
            if (n % i == 0) {
                isPrime = 1;
                break;
            }
        }
        if(isPrime == 1){
            printf("%d␣is␣a␣prime␣number.", n);
        } else{
            printf("%d␣is␣not␣a␣prime␣number.", n);
        }
    }

    return 0;
}
```

**Figure 4.1:** C source code

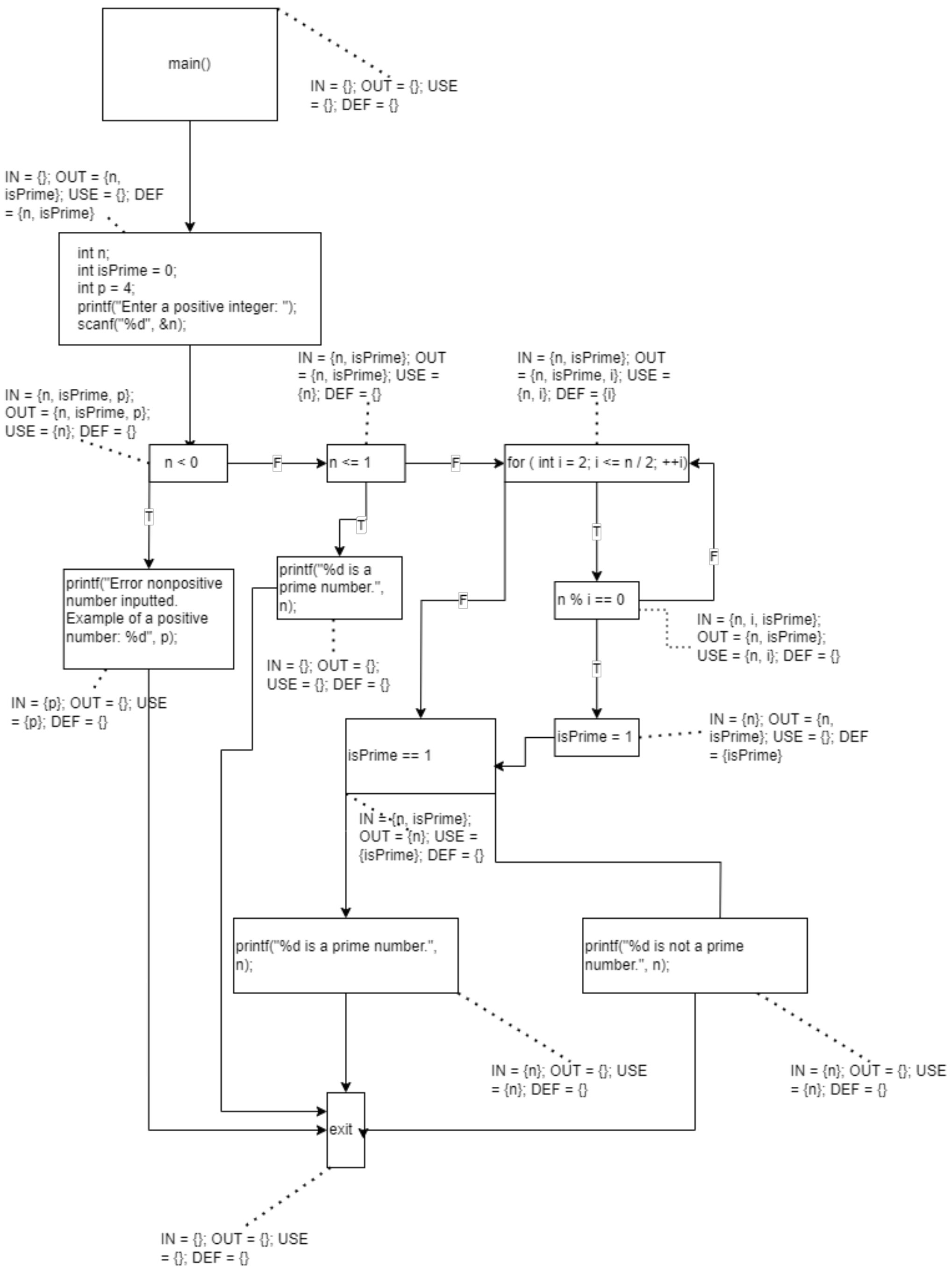

main()
IN = {}; OUT = {}; USE = {}; DEF = {}
IN = {}; OUT = {n, isPrime}; USE = {}; DEF = {n, isPrime}
int n;
int isPrime = 0;
int p = 4;
printf("Enter a positive integer: ");
scanf("%d", &n);
IN = {n, isPrime, p}; OUT = {n, isPrime, p}; USE = {n}; DEF = {}
IN = {n, isPrime}; OUT = {n, isPrime}; USE = {n}; DEF = {}
IN = {n, isPrime}; OUT = {n, isPrime, i}; USE = {n, i}; DEF = {i}
n < 0
F
n <= 1
F
for ( int i = 2; i <= n / 2; ++i)
T
T
T
F
F
printf("Error nonpositive number inputted. Example of a positive number: %d", p);
printf("%d is a prime number.", n);
n % i == 0
IN = {n, i, isPrime}; OUT = {n, isPrime}; USE = {n, i}; DEF = {}
IN = {}; OUT = {}; USE = {}; DEF = {}
IN = {p}; OUT = {}; USE = {p}; DEF = {}
T
isPrime == 1
isPrime = 1
IN = {n}; OUT = {n, isPrime}; USE = {}; DEF = {isPrime}
IN = {n, isPrime}; OUT = {n}; USE = {isPrime}; DEF = {}
printf("%d is a prime number.", n);
printf("%d is not a prime number.", n);
IN = {n}; OUT = {}; USE = {n}; DEF = {}
IN = {n}; OUT = {}; USE = {n}; DEF = {}
exit
IN = {}; OUT = {}; USE = {}; DEF = {}

**Figure 4.2:** Control flow graph of the code

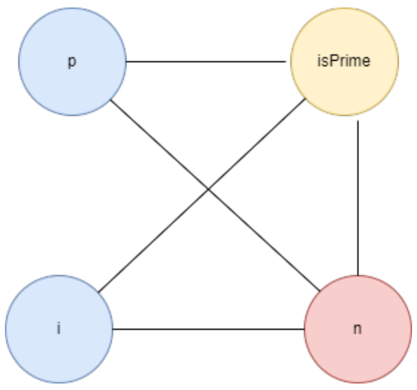


**Figure 4.3:** Interference graph of the code

# Chapter 5

# Related uses dealing with variable scope or similar problems

Our program's primary goal is to use graph coloring to determine the scope of variables. This is accomplished by analyzing the edges leading to and from a node in the interference graph. If the resulting interference graph employs a significant number of colors, it may suggest a scenario where multiple variables are in use and might hint at underlying coding style issues. Graph coloring serves diverse purposes in the context of variables.

An instance of graph coloring's application for variables is found in register allocation. This process seeks to optimize registers for variables, thereby improving the performance of source code. In this context, two variables cannot share the same register if they might be used by the code simultaneously. This determination is made using the interference graph and its coloring, where each color corresponds to a register, and variables are linked if they can be utilized simultaneously [7].

A similar problem is solved by static code analysis, which is a technique that involves examining code without executing it. This analysis provides suggestions for improvement and identifies syntax problems, including violations of coding standards, programming errors, forgotten symbols, or undefined variables.[10] It's worth noting that while static code analysis addresses a broader range of issues, it may not necessarily share a solution with the specific problem our program addresses.

# Chapter 6
# Application

## 6.1 Base Idea

The user will upload a C source file, and the application will output two interactive graphs, a control flow graph (CFG) and an interference graph, where the CFG will represent the uploaded source code.

## 6.2 Selected technologies

We selected Python to implement the application as it provides plotting and parsing libraries, which we use to construct our control flow and interference graphs. We selected a tree-sitter library for parsing to convert the file into an abstract syntax tree. An abstract syntax tree (AST) is a hierarchical structure representing the source code [12], which makes it easier to analyze the control flow graph, as we don't have to implement a direct grammar parser for C language. The selection of the tree-sitter library was primarily based on the ability to have queries for the nodes of the AST, which made parsing expressions for variable references unnecessary. The tree-sitter library also provides a playground to test the queries against a source code, which makes it easy to test if our written query captures the needed nodes for our functions. To show the results of our parsing, we selected the panel library, a web framework designed primarily to deliver data apps. One of its advantages is the ability to develop seamlessly, as it provides an option to reload the app during development if there are changes in the code. Another advantage is exporting the data as HTML without running the server, making it possible to run a console command without needing the server to run the web. Also, there is no forced library for plotting data, even though some have better integration into this library. For this reason, we selected Bokeh to plot our data inside the Panel app, as Panel is built on top of Bokeh library, has extensive documentation, and was the least problematic when working with directed graphs.

To handle the graph data structures and algorithms, including the creation and manipulation of control flow and interference graphs, we chose NetworkX. NetworkX is a popular Python library that supports complex graph operations and comes with a wide range of built-in algorithms for graph analysis, including those for graph coloring. As Bokeh offers a function that integrates NetworkX graphs into itself, provided the underlying data of the graph is serializable, it ensures we can visualize our graphs effectively within the Panel app.

## 6.3 Architecture

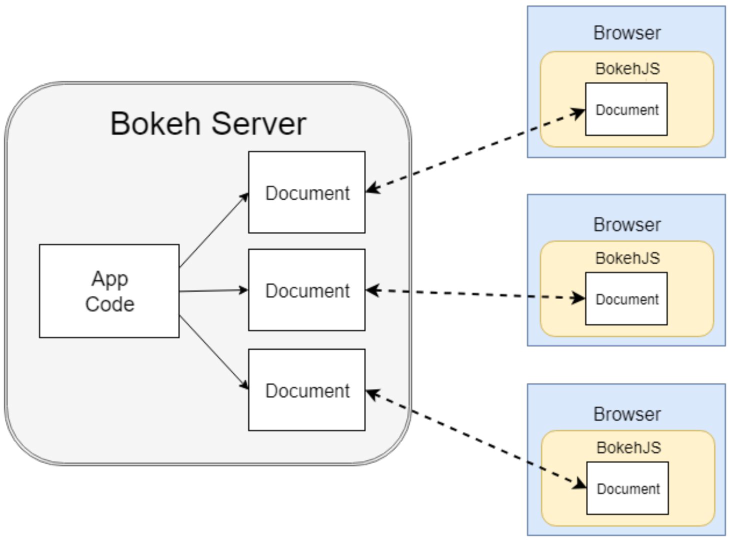


**Figure 6.1:** Panel/Bokeh server architecture[14]

As Panel is built on top of the Bokeh server, the following figure depicts the browser's interaction with the Panel server. The app code is our Python code, which builds the plots and sends the document to the browser. The decision to create a web app was made to offer our application without the need to install anything. However, another reason why Panel and Bokeh were selected was that they provide a pretty simple way to share the resulting plots, as they can be exported as HTML code, provided the callbacks for interaction between the plots are clientside, which means for our purposes, that they are written in Javascript and not Python.

.

The creation of the Bokeh document is as follows:

(i) User uploads a C source file, and it's converted by Bokeh into a base64 encoded string, which is sent as a JSON message back to our application backend.

(ii) Upon receiving the message, our application converts it back to a string, which is then worked on by our app, which includes parsing the string into a CFG and conducting a live-variable analysis on the CFG to create and color the interference graph. When this is finished, the server sends it back as a Bokeh document.

(iii) User either sees no change or Bokeh displays his source file's resulting CFG and interference graph. The application also allows users to download their graphs as HTML files.

## 6.4 Implementation

The following part of the chapter describes the steps of providing the final output for the user.

### 6.4.1 Initial steps

The application or console command accepts input from a c source file. The tree-sitter library then parses this file into an AST. This AST is then used to build a CFG.

### 6.4.2 Construction of a control flow graph from an abstract syntax tree

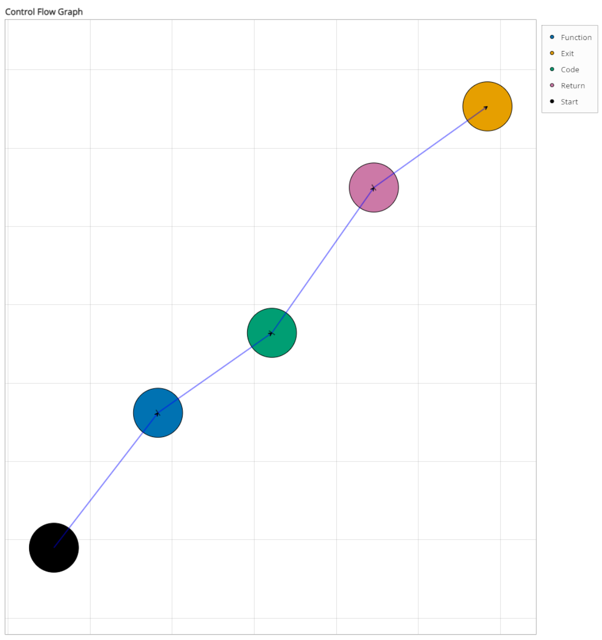
The theory behind constructing a control flow graph is written in chapter 4. It uses program statements, as mentioned in the above section, but we are not working directly with the program but with its abstract syntax tree, as was mentioned in 6.2. That way, the code is already parsed, making it easier to determine the type of statement a node in the syntax tree represents. This also makes creating the first construction of basic blocks of the control flow graph quite simple. As mentioned in the 6.4.3, a single statement can also represent a basic block, so we identify the nodes that represent a statement, and there is no need to visit the statement children for further processing. Finally, we take these basic blocks and connect them to represent the flow of the program. The straightforward part is connecting the statements in sequence one after the other. In the AST, these are represented as siblings in the tree, with the first one being the first statement and the last sibling being the previous statement. However, we must also consider the flow-altering statements, which alter a sequence by having a jump to another node, which is not necessarily a direct sibling.

#### Building basic blocks in the application

For the basic blocks, we need to parse the AST. For this, our application implements visitors[15] on the AST, allowing us to extend our operations without modifying the existing ones. So, to build our basic block, we call the function in 6.2 and let the program determine which function to call based on the statement type. When it creates a new CFG basic block from an AST node, we store the AST node inside the block to have information.

```
def visit(self, node: Node):
    return getattr(self, "visit_" + node.type, self.
        visit_default)(node=node)
```

**Figure 6.2:** Base visitor function



**(a) :** A control graph of hello world.

```
#include <stdio.h>
int main(void)
{
    printf("Hello␣World!");
    return 0;
}
```

**(b) :** Hello world program

## Flow-altering statements

The flow-altering statements will divert the current path, which was forming between the siblings of the syntax tree. To provide a proper control flow graph, we must determine where these diversions lead. Thanks to the abstract syntax tree, we can determine the type of the flow-altering statement; the common ones include for, while, do-while, if-else, switch-case, and goto. And knowing the type, we can model it in our control flow graph.

### Working with the flow altering statements

While constructing basic blocks in the CFG, we must consider the jumps caused by the flow-altering statements and function calls. Loops, if-else, and switches have the jump parsing implemented into themselves, as the statement where they go to is either right after it, which we can represent as an empty potential statement, or, in case of if-else or switch, the following case/condition. We also considered the code situation right after the break. Suppose the source code has a direct successor after the break statement. In that case, we mark the edge to this successor as DEAD CODE instead of removing it entirely from the CFG to provide the user with the information about the unnecessary code. We need to wait until all basic blocks are constructed for goto and function calls because we don't know if the goto label or the called function is already in the program. We can work with the goto and function calls when the whole AST is parsed. For that purpose, the application stores all the CFG nodes during parsing, representing the start of a function and a labeled statement. It also stores all the goto locations and function calls.

### Example depictions of flow altering statements

The following section shows some drawings of the flow-altering statements. Although the previous sections describe the basic blocks of the graph as a single graph, they are merged if possible to reduce visual clutter. Figure

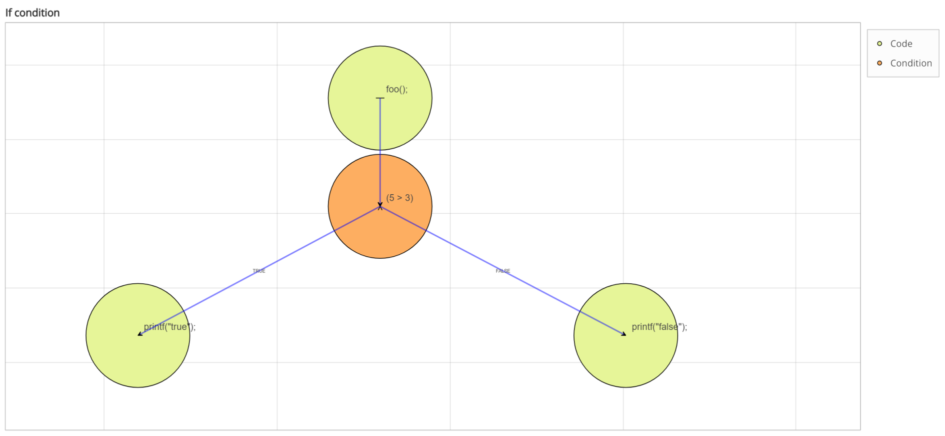
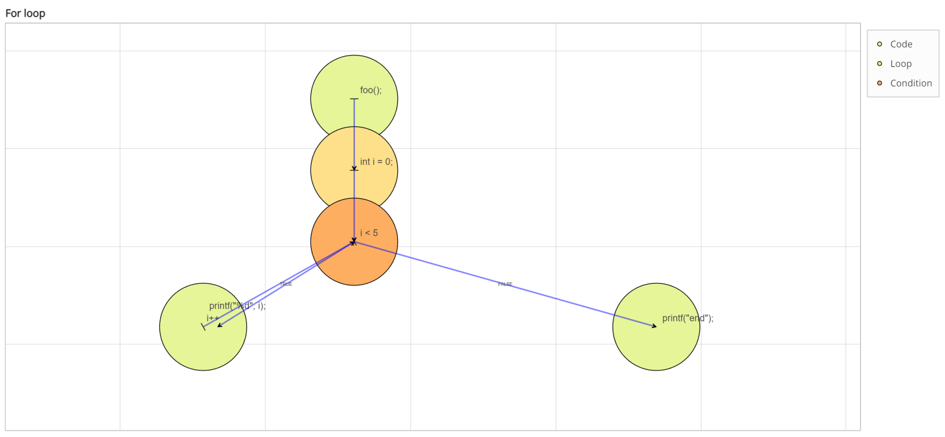
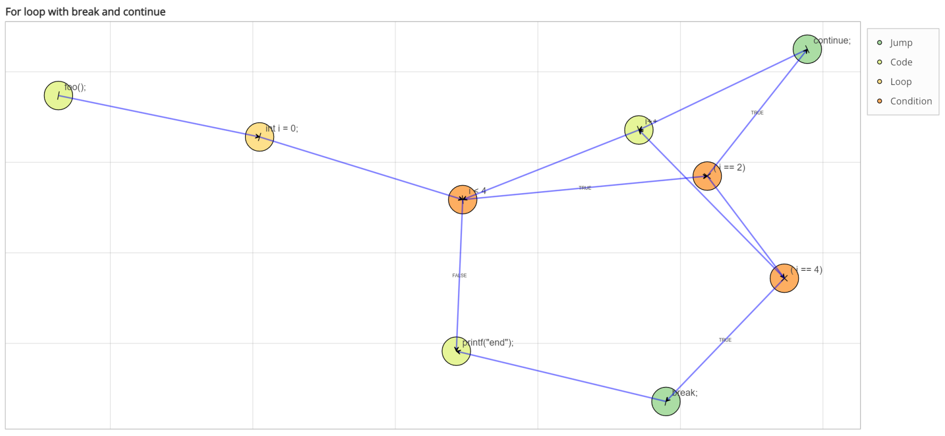


**Figure 6.4:** Drawing of if-else condition in control flow graph

6.4 shows the if-else structure in a control flow graph; a design decision was made to separate the condition from the previous code to make the condition visually distinct from the regular statements. The same decision was applied to the other flow-altering statements to visually inform the user that a branching or a jump is occurring in the program. Figure 6.5 depicts loops; it can be seen that the continue causes the loop to go to the iteration of the loop, while the break ends the loop goes to the statement, where the condition also goes to when it is evaluated as false. A similar construction



**(a) :** Drawing of a for loop



**(b) :** Drawing of a for loop with break and continue statements

**Figure 6.5:** Drawing of for loops

is applied for a while and do-while structures. The difference would be the continue statement directing the flow to the condition instead of the iterating statement.

## Final steps

As was mentioned before, during the construction of the CFG, the flow-altering statements sometimes use an empty node in the CFG to represent the target of flow alteration. Now that we have finished the construction of CFG, we can remove them and connect with the proper target. Our application collects and removes these empty nodes while keeping information about the edges, primarily describing true/false values for conditional statements.

Then it must connect the goto statements and function calls, as these can be connected after we are sure their targets are present, and more often than not, these targets are not direct successors to have an empty statement in the CFG, like in cases of loops and conditional if-else statements. For future purposes, it is helpful for our application to know which CFG nodes share the same scope. For that purpose, we use the handy ability to write a query on the AST from the tree-sitter library to get a list of scopes.

```
def scope(node):
    query = C_LANGUAGE.query("""[
  (for_statement)
  (if_statement)
  (while_statement)
  (function_definition)
  (compound_statement)
  (struct_specifier)
  (do_statement)
  (translation_unit)
] @scope""")
    captures = sorted(query.captures(node), key=lambda
        n: (n[0].end_point[0] - n[0].start_point[0]))
    return captures
```

**Figure 6.6:** Query function to build a scope list

As seen in the 6.6, the structure of this query is a list of nodes in the AST that we want to match and a label for them. The query then returns a pair of nodes and the label. These nodes then carry information about their start and end, which is also their scope in the program. At first glance, it may seem necessary to only use a query for compound statements, which are statements wrapped in  brackets and translation unit, which is the entire C source, including if, while, for, and function definitions in the query is also needed.

Firstly, a variable can be referenced as part of a condition. In the case of a function definition declared as a part of a parameter list, when we capture only the compound statement, it would either be ignored or have a scope of the parent statement of these four. Secondly, which is more important, it is unnecessary to have a compound statement as a child of a for, if, or while statement when the child would be a one-liner. This would have meant that ignoring these four would cause the child statement to have as a scope not only itself but also the scope of the parent statement of the four, which could be excessive and make our program inaccurate. These are why we must have if, while, for, and function definitions in our query list.

We can see a sorting of the list in the 6.6, which is sorted from the smallest to largest, as an inner scope will always be at most equal, never larger than the outer scope, as the outer scope always includes the inner scope. We want to have as precise a scope as possible for each CFG node. Using this list, we take each node in the CFG, which also holds the information about their start, and find the smallest scope where the node in the CFG can fit.

### 6.4.3 Control flow graph to intersection graph

Having finally finished constructing the CFG, we move on to the intersection graph, which is a graph that takes variables as nodes and edges between mean a shared lifetime, where both variables must exist at the same time. First, we need to find all declared variables. Once again, we use a query in figure 6.7 to find all types of declarations. We get the variable's name by capturing the identifiers in this query. We walk through each node in the CFG, take the stored AST node, and use this query to see if something was captured. If yes, we store the information about declaring a variable with the found identifier in the CFG node and keep a list of declared variables and the CFG node in which they were declared. For this purpose, we have created a class Variable in our application, which contains information about its textual representation, references, and definitions. We also set its scope as a scope of the CFG node, which holds the declaration. This scope we found before as the last step of the CFG construction. We now use the second query in the 6.7, which captures redefinitions, to update our variable and our CFG nodes about new definitions. Next, to get all references, we query the

```
query  = C_LANGUAGE.query("""
           (init_declarator
             declarator: (identifier) @local.var)
           (array_declarator
             declarator: (identifier) @local.var)
           (declaration
             declarator: (identifier) @local.var)
           (parameter_declaration
           declarator: (identifier) @local.definition.
              parameter)
           (pointer_declarator
           declarator: (identifier) @local.var)
           """)
assignment  = C_LANGUAGE.query("""
                 (assignment_expression
                   left: (identifier) @reassigned.var)
                   (assignment_expression
                              left: (pointer_expression)
                                 @pointer.var)
             """)
```

**Figure 6.7:** Queries to find declarations and definitions

AST this time for all identifiers, provided we filter out the identifiers we used before in the declaration and the identifiers for function calls and function definitions/declarations. We update our variables with class Variables to hold the information by storing the CFG node holding the reference. We also update the CFG node holding the reference about which variable it refers to.

Now, we can also calculate the actual needed theoretical scope, which follows a similar step as finding a scope of just one CFG node, as this time, we find a scope that fits all the reference nodes, not just one CFG node.

Following our theory of building an interference graph as described in the chapter , we have just built the DEF and USE sets, which are represented by the definition and reference lists in their CFG nodes, respectively. But we now need the IN and OUT sets in the CFG to finally build the interference graph. Figure 6.8 is the function that describes the creation of the IN and OUT sets; it goes through all nodes and updates each in and out set until convergence is reached. Excluding exit nodes is intentional, as described in the section 6.4.3, they are not supposed to be any variables during exit.

```
def get_in_out(graph):
    for node in graph.nodes:
        node.ins.update(node.ref)

    change = True
    while change:
        change = False
        for node in graph.nodes:
            if node.node_type == 'Exit':
                continue

            in_old = set(node.ins)
            out_old = set(node.out)

            node.out.clear()
            for successor in graph.successors(node):
                node.out.update(successor.ins)

            node.ins.clear()
            node.ins.update(node.out.difference(node.
               defs))
            node.ins.update(node.ref)

            if in_old != node.ins or out_old != node.
               out:
                change = True
```

**Figure 6.8:** Function building the in and out graph.

## Building the intersection graph

We can create the intersection graph as we have looped the CFG and have our IN, OUT, USE, and DEF sets. However before that, we decided to merge nodes in the CFG, which can be described as representing linear parts of code without the flow-altering statements. If these parts share the same scope and are in sequence, we will consider them one big CFG node. We take each declared variable and loop through the CFG, and if the CFG has the variable in the OUT or IN set, we create a connection of the variable with all the other variables in the OUT and IN sets. The use of USE and DEF sets is unnecessary as both are already included in the IN and OUT sets, as seen in the figure 6.8, where during construction, the IN set includes all elements of the USE set, as these are references and the OUT set is part of DEF and IN respectively. We ignore the part where a variable is just in the def set, which means it will not be used anywhere and is unimportant for all intents and purposes.

#### Coloring the intersection graph

To color our intersection, we chose the DSATUR algorithm, as it was a middle ground between best coloring and performance, as can be seen in the figure 2.1, even though during testing, there were not that many variables.

### 6.4.4 Final Result

Having finished the graphs, we will deliver the output as a bokeh document. Figure 6.9 depicts the resulting output the user will see. The nodes also have text which describes what they are. However, they are hidden as they can get quite large, and more often, they are readable only if we zoom in on the interested part. The left graph represents the CFG of the program, and the right one represents the intersection graph. The clean separation of functions and no use of global variables can be seen almost immediately, as there are, according to our legend and looking at the nodes colored by the function color, four functions in the program. There are four separate components in the intersection graph. Of course, just seeing these two graphs is not enough information. Clicking the node in the CFG graph will highlight all nodes in the intersection graph that appear in this node. When clicking on the variable node in the intersection graph, all nodes that refer to or declare the variable will be highlighted. Each variable node on hover also provides information about the theoretical needed scope and the actual scope of the variable.

## 6.5 Use cases for user

The user can use this output to find excessive scope usage. These graphs can show how the variable is used and where the variable is used through the combination of interaction between graphs and hover information. As hover information on the CFG node provides the source code lines and the lines themselves, hovering over a variable provides the calculated minimal needed scope and actual scope from declaration to the end of the CFG node scope, where the variable was declared. So, the basic use case for the user is to hover over all the variables in the graph of variables, and if there is a possible issue,

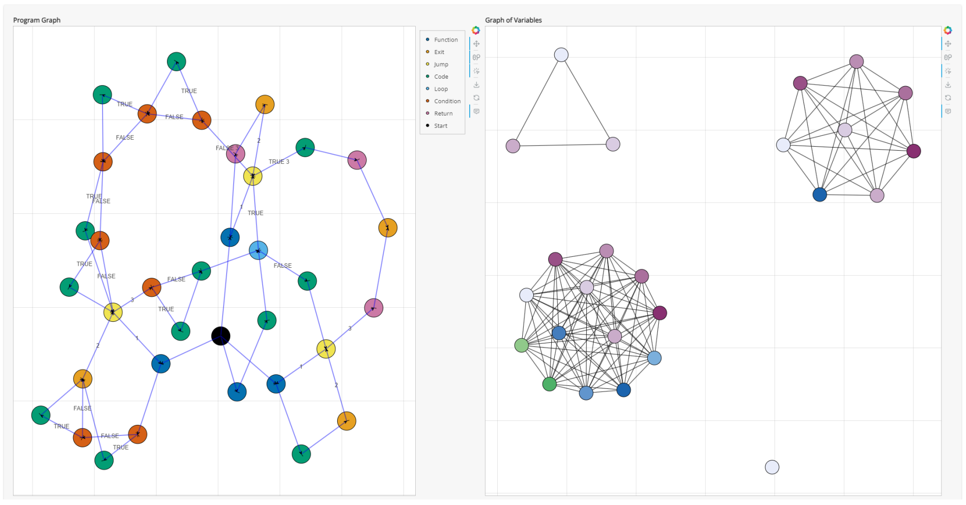


**Figure 6.9:** Resulting application

click on it and see what nodes are highlighted to see what disconnect there is between referenced scope and declared scope. Further details about errors are provided in the next chapter.

## 6.6 Testing

The testing of the application focused on creating flow-altering statements, with unit tests written on for, while, switch,if-else, and function calls. The tests were written by first manually drawing the expected output and then calculating the number of nodes and edges, with the tests comparing these results.

# Chapter 7

# Analysis of the results

## 7.1 Testing on Data

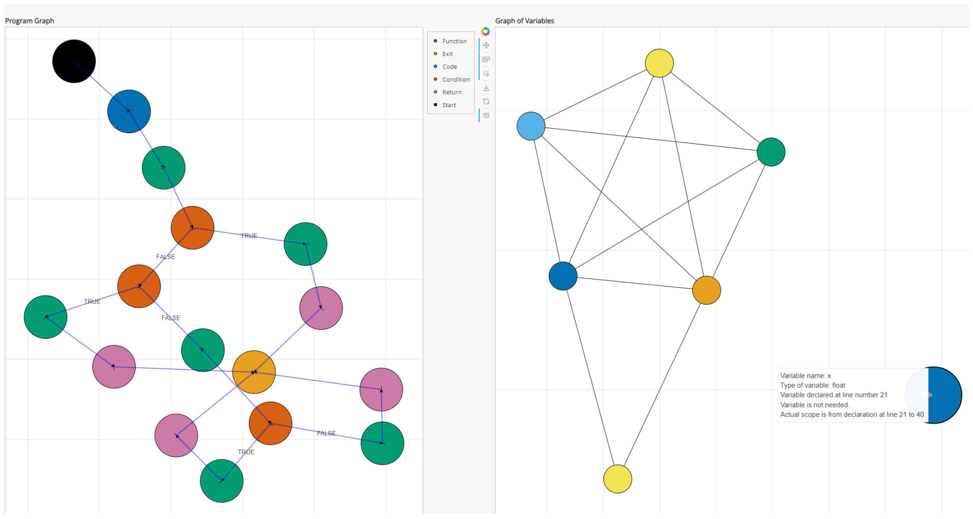
The application was run on a dataset of the first nine homework from the PRP subject, with each homework having at least 48 working submissions. There was one empty submission for the eighth homework. In total, there were 494 files upon which we tested the output of our application, with all of them combined having 14041 variables, during which we found that our algorithm for building the control flow graph and the intersection graph worked for most of them in quite a reasonable time, with the longest time being about 5 seconds. However, a bottleneck was created with a single submission, whose control flow graph was interpreted as having 927 nodes, and laying it out in our application with dot layout, which we use, as it is intended for laying out directed graphs, proved to take at least 20 minutes, when it was drawn with a layout for huge graphs it took 5 minutes to complete and display the result.

## 7.2 Found uses for our application

Thanks to this, we can detect errors related to early variable declaration, which leads to excessive scope and unused variables. These errors are depicted through the hover information on the variable and highlighted nodes in the CFG; both must be used to infer why the scope is excessive.

When dealing with unused variables, we also found that they are not separate in the code, as we initially considered when building it from references and definitions. However, what proved to be more effective was to consider only all variables exiting and entering a node. This approach makes unused declarations/definitions dead, which, during the construction of the interference graph, causes them to be ignored since they won't leave their node.

While this makes visualizing unused variables quite simple, as they are practically separate single components, it is also important to note that separate components can form a function scope. Therefore, a component consisting of a single variable does not immediately indicate that it is an unused variable. It could also be a function that uses just one variable. Thus, it is necessary to hover over the variable and click it to see where it is declared and how it is used, which would give the user the reason why it stands alone.



Variable name: x
Type of variable: float
Variable declared at line number 21
Variable is not needed.
Actual scope is from declaration at line 21 to 40.

**Figure 7.1:** Unused variable x in the code

The other error of early variable declaration is much more difficult to depict. When a variable is clicked, it highlights all the nodes where it is referred to, declared, or defined. However, seeing a scope in the control flow graph (CFG) is hard. It could be seen as starting from a node that initiates a scope (such as a condition, function, or loop). However, when there are jumps between nodes, it can be difficult to track back to the start of a scope.

Therefore, we provide the information of the theoretical scope as numbered lines, and the user can open their own source code, which might prove easier than attempting to figure out why our application considers it an early declaration by trying to trace it through the CFG. For example, Figure 7.2 shows an early declared variable and its path through the program. While the path in the provided example figure is quite clear, it might be easier for the user to just look directly at their source code. If properly formatted, a search feature can help see why the variable was considered to be declared early.

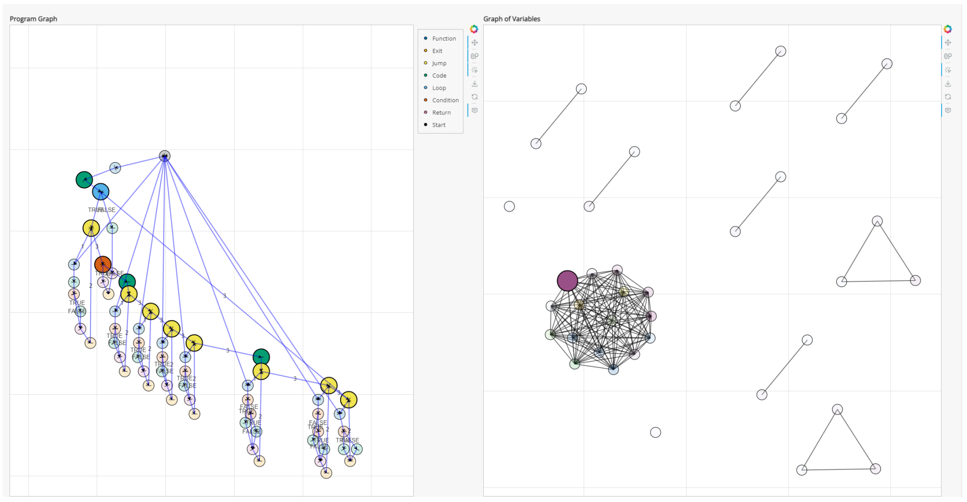


**Figure 7.2:** A variable that is early declared.

## 7.3 False positives

We also found during the analysis of early variable declarations that our application has quite severe limitations. As we consider references only during calculation, the application determines variables used only in conditional statements and in the children of these statements as having scope only in the conditional statement. However, in C99, there is no default support for declaring variables inside a condition, so they must be declared outside of it. Therefore, it is necessary to consider if the calculated theoretical scope is achievable.

There was an attempt to fix this by not considering if and while statements as their own scope. However, as mentioned in the previous chapter, this would result in excessive scope capture for one-line if and while statements, as a compound statement scope capture will not consider these. This creates an accuracy tradeoff: achieving accurate scope for one-line if and while statements leads to inaccurate scopes for variables used only in the condition and the children of the condition in the CFG. Consequently, users should consider our output more like advice rather than a definitive rule that can be applied universally, and look for themselves if the theoretical scope is applicable in their situation.

Another limitation is that our application does not consider operations. For example, if we have an update expression in the form x += 2 inside a loop and use the result only within the loop, our application will consider this x as needed solely within the loop. This is only possible if the user employs a static keyword variable. Users must be aware of these limitations and interpret the application's output accordingly.

Figure 7.3 shows this by showing the path of the variable it goes through. You might notice the similarity between this figure and figure 7.2, where both are declared just outside of the loop. Still, upon further inspection of this

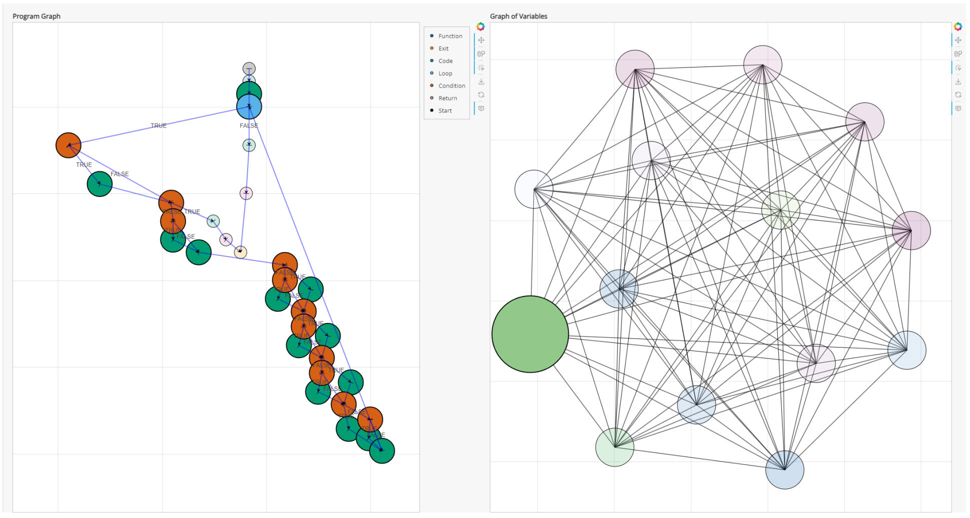


**Figure 7.3:** A variable that is part of a condition

figure, we find that this early declaration is necessary as it is the variable used in the condition. As mentioned before, we cannot declare a variable inside a condition in C99.

## 7.4 Applying the results of our analysis

In total, We found 1931 variable scope differences in 14041 variables during our attempts to find errors by analyzing a minimal scope of references, implying an error rate of approximately 13.7%. However, a lot of them were false positives due to the before-mentioned inability in c to declare variables directly in a condition, so we settled on automatic detection of the variable scope by finding a difference between the reference and the definition reaching it and the actual scope, which means we calculate the needed scope as the minimal scope which holds the references and the written definitions they use, and in the end, we found only 434 mistakes, with the errors being unused variables or redefinitions used in an inner scope with the declaration somewhere in the outer scope.

As a result, we can confidently say that about 3.1% of variables have had an erroneous scope in the source code. The information of different scopes is available as hover information on the variable node and, as such, is visible just by hovering over a variable node. The rest of the errors are left up to the interpretation of the user and can be figured out by reading the manual at C.3

# Chapter 8
# Conclusion

The aim of the application was to provide the user with information about excessive variable scope through the graph coloring and the interference graph.

During development, it showed that showing just this graph is not enough to provide the information of excessive variable scope. Therefore, we used the control flow graph, which was already built for building an interference graph, to show the information about the usage of variables and their travel through the scope. Thanks to this, we provide enough context for the user, but as looking through the control flow graph might be tedious, we also provide the theoretical needed scope and actual scope as hover information, so the user can just look directly into the source code.

So even though the goal was to show the variable scope through just variable coloring, variable coloring ended up in our application more as a showing, that variables are not needed at the same time if they share a color, and to determine if a scope is excessive, we used the interactions between interference graph and CFG, as in if a clicked node in CFG refers to declares to a variable, highlight it in the interference graph, and a clicked variable, highlights its paths from definition to its usage.

During our analysis, we found the limitations of our program, in particular when dealing with conditions and variables in them and also updating expressions inside loops. This sometimes leaves the resulting output of the application up to the correct interpretation of the user.

Despite this, we believe our application can, more often than not, show how to have a correct scope and why it is incorrect through the depiction of the path of the variable and the information about the scope.

## 8.1 Possible future development

During development, there was an attempt to implement pointer tracking, which would make it easier to find allocation and deallocation of the pointer. The attempt failed as it required further processing of functions, and working with all the options of passing pointers through the code. It also does not exactly apply to our main aim, as forgotten pointers would not be trackable in the interference graph thus the idea was to have a third graph, where by clicking on a node in the CFG, we would show a pointer graph, which would be built from a dictionary of pointers reachable in the selected CFG node and this dictionary would flow, however, this also has the limitation of not having enough information to determine if the pointer needs to be freed, does the area it points at still exist in scope particularly if it points a local variable in a function and is returned. It proved to be quite a difficult task when we must also consider all the possible operations through which the pointer can get a value and whether the value is an allocation or not.

So the current best solution in our program is to take the path of a pointer and look at its ends, which would be highlighted nodes in the CFG without a highlighted successor, and see if there are free statements there if the variable was allocated.

Another possible future development is working with projects, as right now it works with just a single source file. While this is enough for most use cases, especially as the intent is to use it for PRP homework, it might prove nice to visualize the interaction between two files and have as an option, for example, a separate file for utility functions that might be reused multiple times across different projects.

# Bibliography

[1] J. Demel. *Grafy a jejich aplikace.* https://kix.fsv.cvut.cz/demel/grafy/gr.pdf

[2] J. A. Bondy and U. S. R. Murty. *Graph theory and applications.* 5th ed. Elsevier Science Publishing Co., Inc. ISBN O 444-19451-7.

[3] K. Appel. W. Haken. *Every planar map is four colorable. Part I: Discharging."* Illinois J. Math. 21 (3) 429 - 490, September 1977. https://doi.org/10.1215/ijm/1256049011

[4] K. Appel. W. Haken. J. Koch. *Every planar map is four colorable. Part II: Reducibility.* Illinois J. Math. 21 (3) 491 - 567, September 1977. https://doi.org/10.1215/ijm/1256049012

[5] W. Klotz. *A constructive proof of Kuratowski's theorem.* Ars Combinatoria. 28. 51-54, 1989.

[6] S. G. Williamson. *Depth-First Search and Kuratowski Subgraphs.* J. ACM 31, 4 (Oct. 1984), 681–693. https://doi.org/10.1145/1634.322451

[7] G. J. Chaitin *Register allocation and spilling via graph coloring* SIGPLAN Not. 17, 6 (June 1982), 98–101. https://doi.org/10.1145/872726.806984

[8] D. Brélaz *New methods to color the vertices of a graph.* Commun. ACM 22, 4 (April 1979), 251–256. https://doi.org/10.1145/359094.359101

[9] F. T. Leighton *A Graph Coloring Algorithm for Large Scheduling Problems.* Journal of Research of the National Bureau of Standards, 84(6), 489-506. https://doi.org/10.6028/jres.084.024

[10] M. J. Harrold and G. Rothermel *Notes on Representation and Analysis of Software.* December 2002, https://faculty.cc.gatech.edu/ harrold/6340/cs6340_fall2010/Readings/rep_and_analy.pdf

[11] Frances E. Allen. 1970. *Control flow analysis.* SIGPLAN Not. 5, 7 (July 1970), 1–19. https://doi.org/10.1145/390013.808479

[12] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools. Second edition.* Addison-Wesley, Reading, MA, 2016

[13] *ISO/IEC 9899:1999 (the C99 standard)*

[14] https://panel.holoviz.org/how_to/server/index.html

[15] Bruce Eckel. 2017. *Python 3 Patterns, Recipes and Idioms*, available at https://readthedocs.org/projects/python-3-patterns-idioms-test/downloads/pdf/latest/

# Appendix A

## Tables

| Algorithm | N | Average Colors | Time |
|---|---|---|---|
| Greedy | 10 | 3.38 | 0.00001 |
| Greedy | 50 | 12.1 | 0.0001 |
| Greedy | 100 | 20.36 | 0.0006 |
| Greedy | 200 | 34.7 | 0.0014 |
| Greedy | 500 | 71.72 | 0.0114 |
| Greedy | 750 | 99.68 | 0.0114 |
| Greedy | 1000 | 125.6 | 0.0537 |
| Dsatur | 10 | 3.1 | 0.00009 |
| Dsatur | 50 | 10.82 | 0.0021 |
| Dsatur | 100 | 19.06 | 0.0078 |
| Dsatur | 200 | 32.64 | 0.0317 |
| Dsatur | 500 | 68.66 | 0.2301 |
| Dsatur | 750 | 95.68 | 0.2301 |
| Dsatur | 1000 | 121.58 | 0.9465 |
| RLF | 10 | 3.22 | 0.00004 |
| RLF | 50 | 10.48 | 0.0012 |
| RLF | 100 | 17.7 | 0.0094 |
| RLF | 200 | 30.4 | 0.0571 |
| RLF | 500 | 64.08 | 0.7101 |
| RLF | 750 | 89.08 | 2.1847 |
| RLF | 1000 | 114.1 | 5.3713 |

**Figure A.1:** Table of measured performances of algorithms on random graphs

# Appendix B
# Used software

Following the Methodological guideline 05/2023[1], the following was used during development.

- Microsoft Bing copilot as a search engine.[2]
- ChatGPT as text feedback and rephrasing suggestions of sentences.[3]
- Grammarly as grammar correction and text feedback.[4]

[1]https://intranet.fel.cvut.cz/cz/rozvoj/MP-pouzivani-ui.pdf
[2]https://www.bing.com/chat
[3]https://chatgpt.com/?oai-dm=1
[4]https://www.grammarly.com/

# Appendix C
# How to use the App

## C.1 Installation

- Download the app, which is part of the attached files.
- Install Python 3.12[1] or newer
- Inside the app package is a file requirements.txt containing a list of required libraries that have to be installed as part of development; install using your package manager, for example, using pip:
  pip install -r /path/to/requirements.txt
- Install the python library pygraphviz[2] separately, as it depends on graphviz;
- To run the app, if you have successfully installed pygraphviz, run in console `panel serve app.py`
- Otherwise, run in console `panel serve app_noviz.py`
  , which is the same, only the laying out algorithm is much simpler, so the result might not be as nice.
- The resulting app runs on http://localhost:5006/
- If the static output is enough, run in console `python console.py {your/source/file.c}` with optional argument `--export {path/to/file.html}`. Again, if you do not have pygraphviz, use con_noviz.py instead.

## C.2 Application

### C.2.1 Initial screen

Figure C.1 shows the application's initial screen. It is an empty main part of the screen with a sidebar that offers an option to upload a c file.

[1]https://www.python.org/downloads/release/python-3120/
[2]https://pygraphviz.github.io/

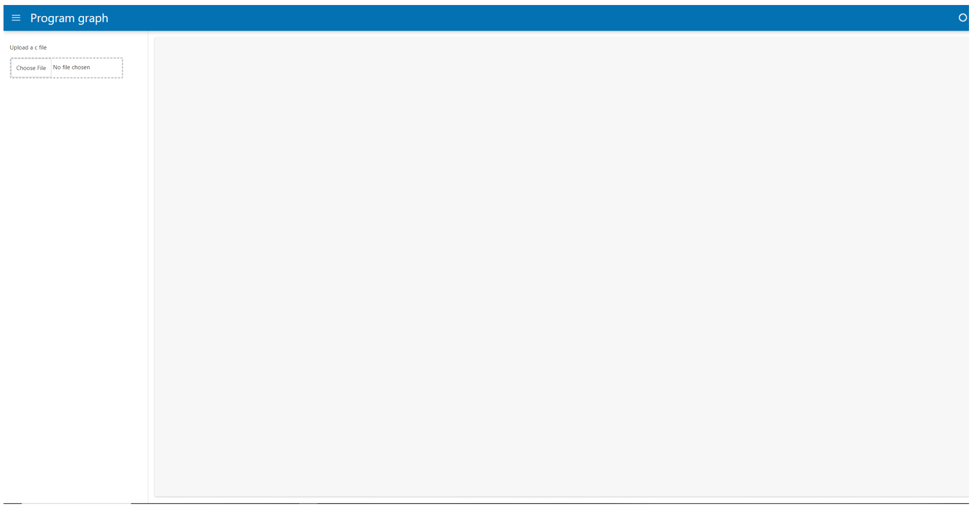


**Figure C.1:** Initial screen

### C.2.2 Uploaded File

Figure C.2 shows the result of uploading a c file. First, a new option in the sidebar downloads the main screen as a static HTML file that can be sent and maintains interactivity. On the main screen are two graphs; on the left, the graph depicts a control flow graph of the uploaded source file and a legend next to it describing the color of those nodes. On the right, we have the graph of variables, where a connection between two nodes means that there is a path in the control flow graph, which the two variables share at least a part of. Often, they form a cluster representing either a completely isolated scope or a function. It is more likely a function. Of course, global variables will connect these clusters when the global variables are used in both of them. There are also instances of singular nodes, which could mean unused variables, as they will have no scope and no interaction with the other variables, a function that takes a single variable as an argument, or a function with no arguments that has a variable it uses for itself. There are three buttons on the bottom: the first one shows or hides the code directly in the control flow graph, the second shows or hides the names of the variables in the graph of variables, and the last button toggles showing the paths for all references from the nearest definition in the control flow graph, or show its references and definitions.

### C.2.3 Toolbar

Each graph has a toolbar on the right of it. There are 6 tools:

- Pan tool allows moving the graph.
- Wheel zoom allows zooming in the graph by scrolling.
- Tap tool is for selecting and deselecting nodes.

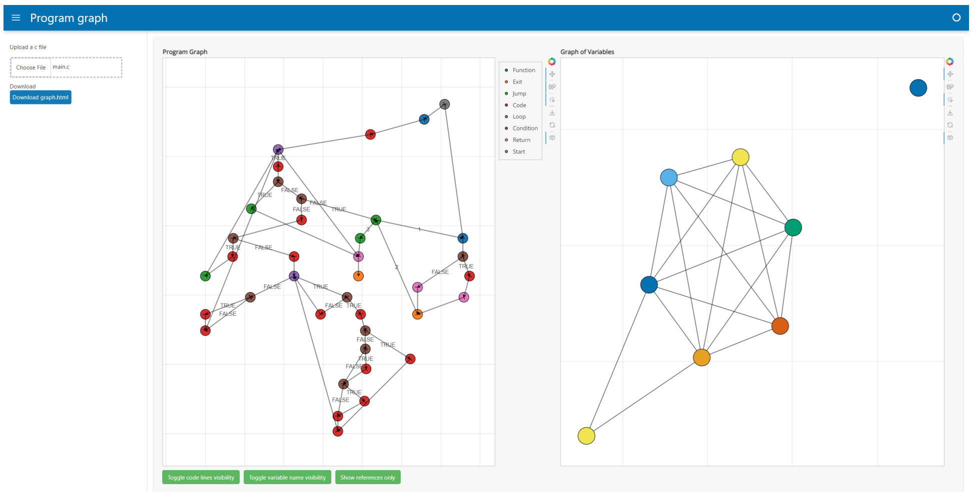


**Figure C.2:** Result of uploading a c file

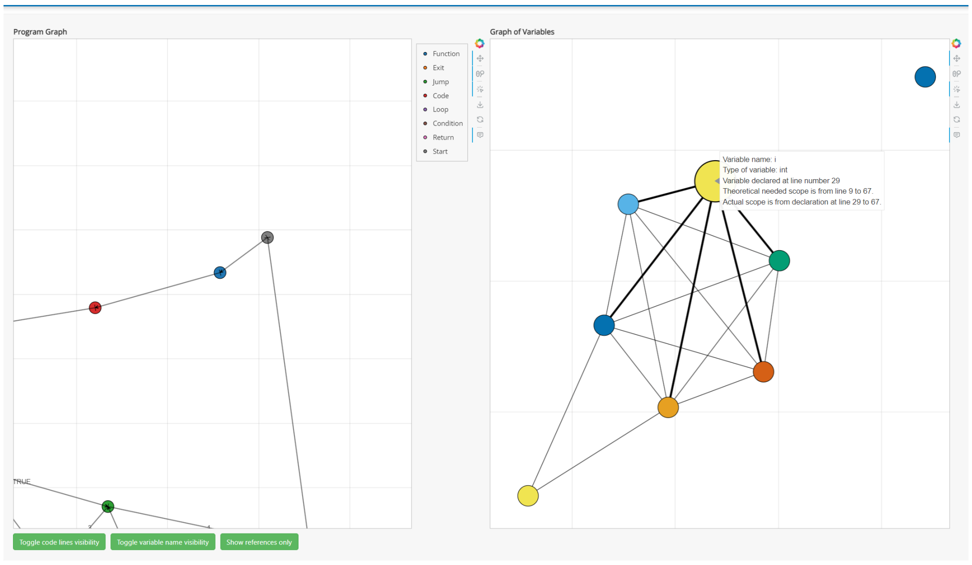
- Save allows the user to download the plot as an image.
- Reset rescales back the graph and moves the graph to the center.
- Hover makes the user see the hover information when the mouse is placed on a node.

## C.2.4 Hover information

Both graphs have hover information on their nodes. The hover information of a variable node (as shown in the figure C.3) shows the theoretical scope needed by finding the scope, which encompasses all paths the variable needs to reach its references from a definition. The actual scope shows the scope of the declared variable, from its declaration to the end of the scope, where it is declared. The rest is self-explanatory. The variable name is its name, the type, its type, and the variable declared at the line means that line is the first time the variable shows up in the source code. Figure C.4 shows the information about the source code itself. It describes what type of node it is, as can also be seen from the legend and its color, and it shows the line directly in the source code where it is placed.

## C.2.5 Interactions

Figure C.5 shows the result of a click on a node in the control flow graph; this shows the variables in the variable graph that are referenced or defined in this node. This can be useful to find which variable is used in that node, which may help if there is a problem with variable shadowing. Figure C.6 is after clicking on a variable node in the variables graph. This figure only shows the references and highlights them. While figure C.7, shows the entire path the variable has to take to reach its references from its definitions. These definitions can be recognized in the control flow graph as a node with no



Variable name: i
Type of variable: int
Variable declared at line number 29
Theoretical needed scope is from line 9 to 67.
Actual scope is from declaration at line 29 to 67.

**Figure C.3:** Variable hover information

predecessors. The use of the path can be as tracing back in the source code to see what value was probably used in the reference. It is also possible to try finding if a pointer is freed by tracing the last references and see if they are calls to a free function, with the last references being ones with no highlighted successor. The change between showing the entire path and just its references and definitions is triggered by clicking the third button, with the text describing the future result, not what is currently displayed, as can be seen in the changes of the button label from C.6 to C.7.

## C.3 How to use the output

Given this output, there are two options for what to do. You can hover/display the names of the variables to find which one interests you, and it will highlight the path this variable takes in the control flow graph if you click, which could be seen as where in the source code this variable is. You can often recognize an excessive scope by having one highlighted node far from the others, especially with no direct connection to them. Also, you can see if a variable is unused; if a single node is highlighted, look at it; if the variable is just there declared, you probably do not need it in your code, or you forgot to use it if you assigned some value to it and probably recalculated that value somewhere else and used it there. Or you click the nodes of the control flow graph to see which variables are used in that node, for which one major use case is probably variable shadowing, as you can see directly which variable is used by hovering over that variable and see where it is declared.

A good option is finding the start node in the CFG and then going from there by finding a function that interests you or just clicking through all the variables and seeing if something catches your eye. Another is to go through

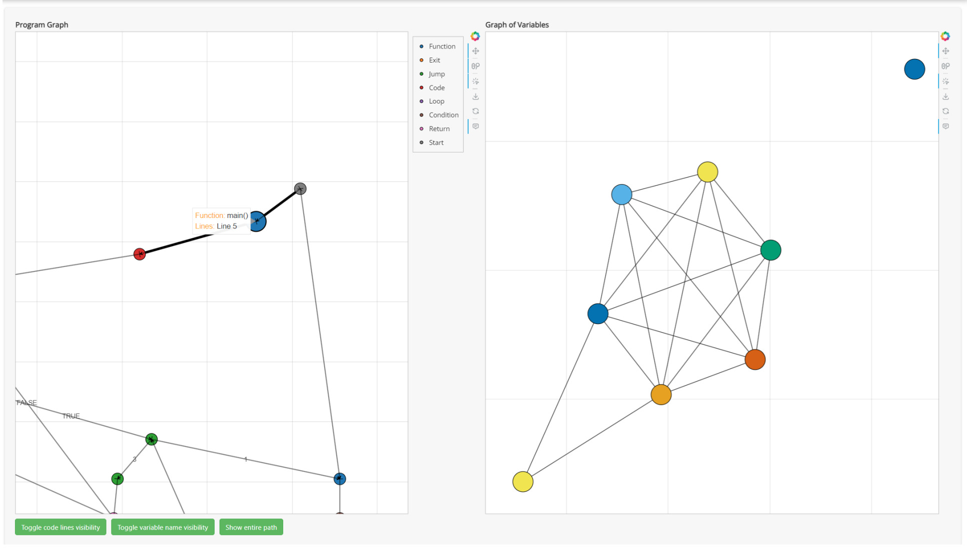


**Figure C.4:** Hover on a node of the control flow graph

the smallest variable clusters first in the Graph of variables, especially if you think that either the variable is supposed to interact with more than they are connected to or if they are singular. You are not certain of the reason. You can then select it and the CFG will display for you, where and how is the variable running through your code,

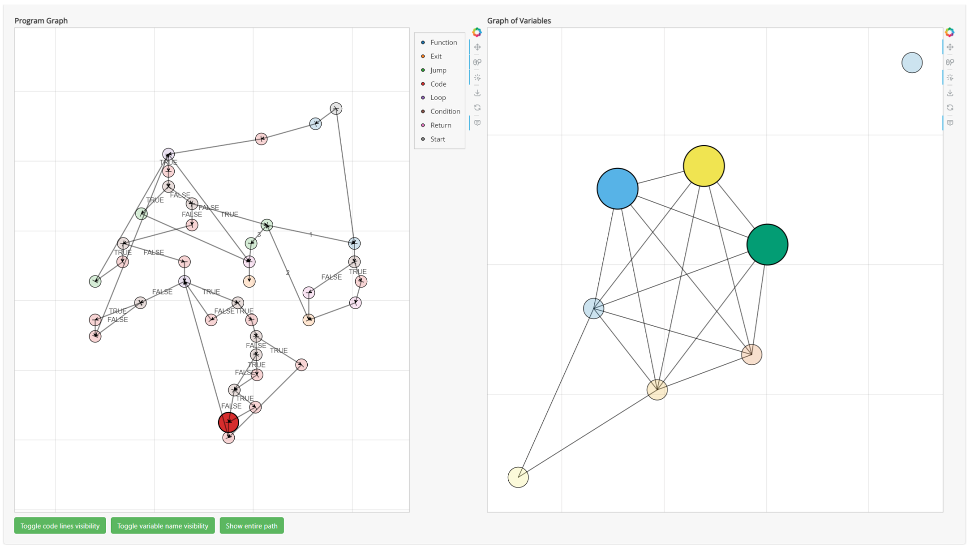


**Figure C.5:** Click on a node in the control flow graph

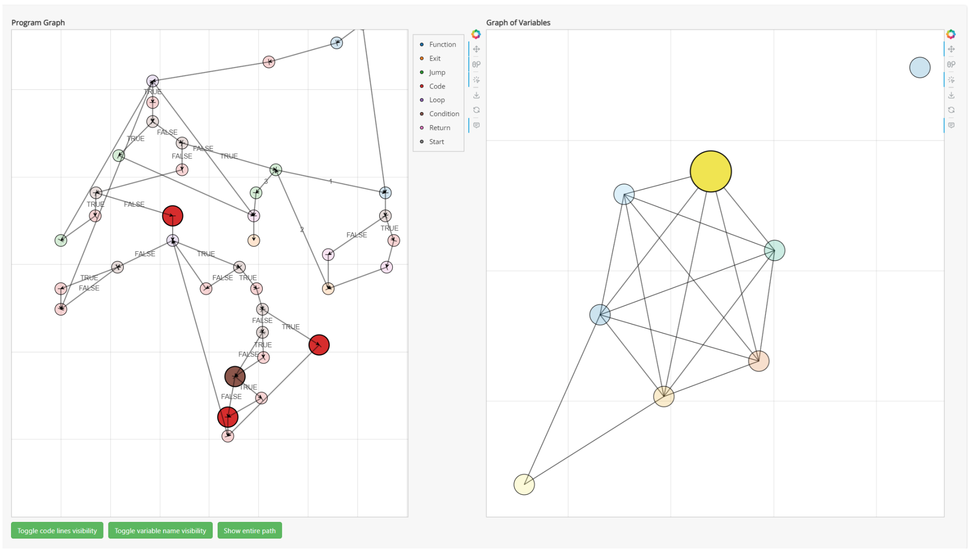


**Figure C.6:** Click on a node in the graph of variables, with references only

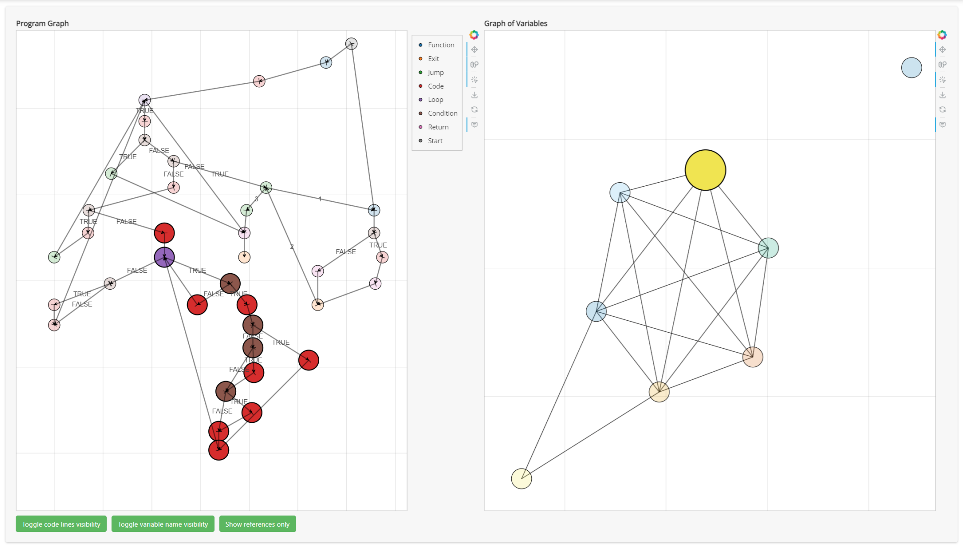


**Figure C.7:** Click on the same node in the graph of variables with the entire path

# Appendix D
# Shortcuts

|  |  |
|---|---|
| DSatur | Saturation largest first algorithm |
| RLF | Recursive largest first algorithm |
| CFG | Control flow graph |
| AST | Abstract syntax tree |
| PRP | Procedural programming for OI (subject at CTU FEE) |