SPL-1 Project Report, 2022

CodeToImage

Embedding Statement-Level Information of Code into Image

SE 305: Software Project Lab

Submitted by

Swadhin Pal

BSSE Roll No. : 1302

BSSE Session: 2020-2021

Supervised by

Kazi Muheymin-Us-Sakib

Signature

Designation: Professor

Institute of Information Technology

Institute of Information Technology

University of Dhaka

[25-05-2023]

Table of Contents

1. Introduction

Data embedding is a widely recognized process of concealing data within a host medium, such as images, videos, audio, or text. It has found applications in various fields, including information security, steganography, and digital watermarking. The project CodeToImage focuses on the novel concept of embedding statement-level information of C programs into image vectors. By combining concepts from code centralities and statement embedding, CodeToImage aims to represent code structures and relationships visually.The project CodeToImage encompasses two major fields: Code Centralities and Statement Embedding. Code centralities involve the use of centrality measures, commonly employed in network analysis, to evaluate the importance and centrality of nodes within a code network. By considering the code as a network of interconnected components, such as functions, modules, and classes, CodeToImage leverages these centrality measures to extract meaningful information about the code's structure.

In addition to code centralities, the project employs statement embedding techniques. Statement embedding focuses on capturing the structural information of individual statements within the code. By representing statements as vectors, CodeToImage aims to encode the relationships and dependencies between statements.The primary objective of CodeToImage is to combine code centralities and statement embedding to embed both types of information into an image vector. This innovative approach seeks to create visual representations that not only convey the statement-level information but also capture the overall structure and flow of the code.

By embedding statement-level information into image vectors, CodeToImage offers several potential benefits. Firstly, it enhances code comprehension by providing a visual representation that captures the intricacies of code structures. Developers can gain a holistic understanding of the code's logic and dependencies, facilitating debugging and maintenance tasks. Secondly, visual representations enable efficient collaboration and communication among developers, as they provide an intuitive medium for discussing and analyzing code. Lastly, the image vectors generated by CodeToImage can serve as a compact and portable representation of code snippets, facilitating code sharing and distribution.

1.1. Background Study

In this project implementation some background knowledge is required such as:-

1.1.1. String Manipulation

String manipulation refers to the process of modifying, extracting, or transforming strings of characters in programming languages. Strings are sequences of characters, such as letters, numbers, and symbols, that are used to represent text-based data. Manipulating strings allows developers to perform various operations, including concatenation, splitting, substitution, and formatting, to achieve specific tasks and manipulate the content and structure of strings. String manipulation is vastly used in standardization and segmentation of the code.

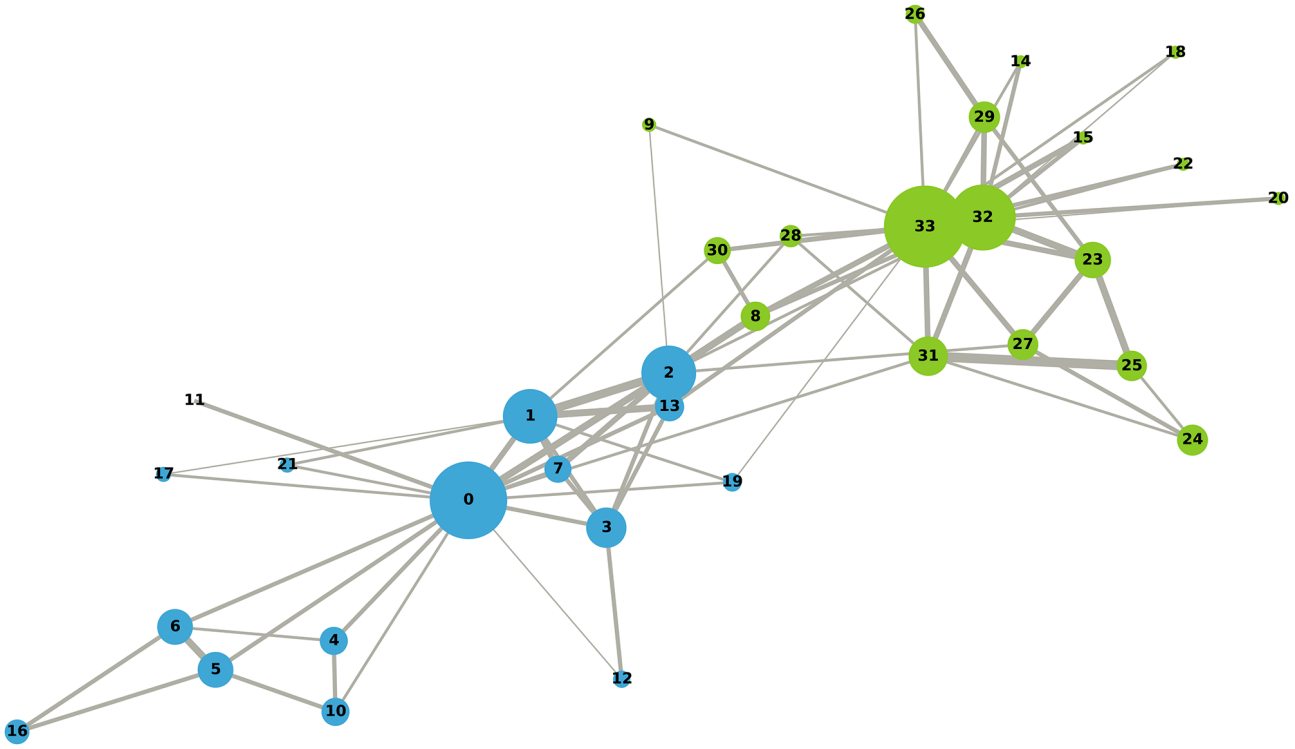
1.1.2. Dependency Graph

Control Dependency Graph

A Control Dependency Graph (CDG) is a graphical representation that depicts the control flow dependencies between statements or blocks of code in a program. It provides a visual representation of how the program's control flow branches and merges based on conditional statements, loops, and function calls.The CDG captures the order and relationship between different program statements by representing them as nodes in the graph and the control dependencies between them as edges. The control dependencies indicate that the execution of one statement or block of code depends on the outcome of another statement or block.In a CDG, the basic blocks or statements of a program are represented as nodes, and the control dependencies between them are represented as directed edges. A control dependency from statement A to statement B indicates that statement B can only be executed if statement A evaluates to a specific condition or is executed first. The control dependency graph helps in understanding the flow of execution within a program and identifying potential control flow issues, such as loops, conditional branches, or unreachable code.
Control dependencies can be represented using different types of edges in a CDG:

1. Conditional Edges: Represent the control flow based on conditional statements, such as if-else or switch-case constructs. These edges indicate the alternative paths the program can take based on the outcome of the condition.
2. Loop Edges: Represent the control flow within loops, such as for, while, or do-while loops. These edges indicate the iteration and repetition of statements until a specific condition is met.
3. Function Call Edges: Represent the control flow between different functions or procedures. These edges indicate that control transfers from the calling function to the called function and returns back once the called function completes its execution.

By analyzing the CDG, programmers can gain insights into the control flow behavior of their code. It helps in understanding the order of execution, identifying potential code paths, detecting dead code (unreachable statements), and analyzing the impact of changes to the control flow structure. CDGs are commonly used in software analysis, debugging, optimization, and program comprehension. They provide a visual representation of control dependencies, aiding developers in understanding complex program structures and identifying potential issues related to control flow.



Data Dependency Graph

A Data Dependency Graph (DDG) is a graphical representation that illustrates the dependencies between data elements or variables in a program. It provides a visual representation of how data flows and is used or modified by different statements or blocks of code.The DDG captures the dependencies between data elements by representing them as nodes in the graph and the dependencies between them as edges. These dependencies indicate that the value of one data element is used or influenced by the value of another data element.In a DDG, variables or data elements of a program are represented as nodes, and the data dependencies between them are represented as directed edges. A data dependency from variable A to variable B indicates that the value of variable B depends on the value of variable A.
There are different types of data dependencies that can be represented in a DDG:

1. Read-after-Write (RAW) Dependency: Represents a data dependency where a variable is read after being written in a previous statement. This dependency ensures that the value being read is the most up-to-date value produced by the preceding statement.
2. Write-after-Read (WAR) Dependency: Represents a data dependency where a variable is written after being read in a previous statement. This dependency ensures that the value being written does not overwrite the value being read before it has been used.
3. Write-after-Write (WAW) Dependency: Represents a data dependency where a variable is written multiple times in consecutive statements. This dependency ensures that the values written to the variable are ordered correctly and do not overlap or conflict with each other.

By analyzing the DDG, programmers can understand the flow and dependencies of data within their code. It helps in identifying potential data hazards, such as data races or incorrect ordering of data operations, and assists in optimizing code by rearranging statements to minimize dependencies.
DDGs are widely used in compiler optimization techniques, such as loop optimizations, register allocation, and instruction scheduling. They provide insights into the data dependencies within a program, allowing compilers to analyze and transform code to improve performance and reduce execution time.
Additionally, DDGs are valuable in program comprehension, debugging, and analyzing the impact of code changes. They aid developers in understanding how data is propagated and used throughout the program, identifying potential issues related to data flow, and facilitating the maintenance and optimization of code.

Program Dependency Graph

A Program Dependency Graph (PDG) is a comprehensive graphical representation that combines both control dependencies and data dependencies in a program. It merges the Control Dependency Graph (CDG) and the Data Dependency Graph (DDG) to provide a unified view of the dependencies between statements, blocks of code, and data elements.
In a PDG, the nodes represent individual statements or blocks of code, as well as variables or data elements used within the program. The edges in the graph capture both control dependencies and data dependencies between these nodes.
Control dependencies in a PDG represent the control flow of the program, indicating how the execution of one statement or block depends on the outcome of other statements or blocks. This includes conditional branches, loops, and function calls.
Data dependencies in a PDG represent the flow of data between different statements or blocks, indicating how variables or data elements are used, modified, and shared. This includes read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependencies.
By combining control and data dependencies, the PDG provides a more comprehensive understanding of how the program executes and how data flows within the program. It allows developers and researchers to analyze the interaction between control flow and data flow, enabling them to identify potential issues related to both aspects.

PDGs are particularly useful in program analysis, optimization, and understanding the behavior of complex programs. They facilitate tasks such as detecting performance bottlenecks, identifying opportunities for parallelization, optimizing data accesses, and detecting potential errors or vulnerabilities.
Overall, the Program Dependence Graph offers a holistic representation of the dependencies within a program, merging both control and data aspects, providing a powerful tool for understanding, analyzing, and optimizing software systems.

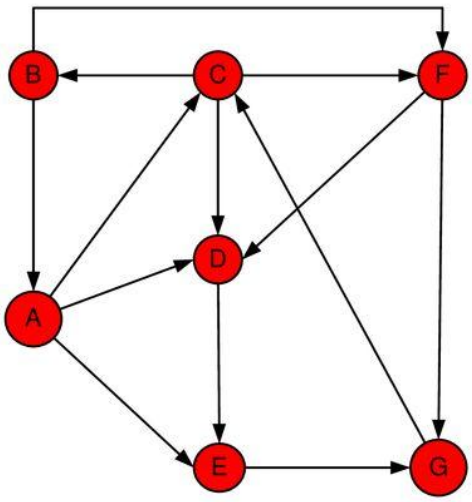
1.1.3. Centralities

Centrality measures quantify the importance or influence of nodes in a network.Degree centrality is based on the number of direct connections a node has.Closeness centrality measures how quickly a node can reach other nodes in the network.

Degree Centrality:

1. Degree centrality is a fundamental measure in network analysis that quantifies the importance or centrality of a node based on its degree, which is simply the number of direct connections it has. In the context of a network, the nodes represent entities, and the connections represent relationships or interactions between them. Degree centrality assesses the prominence of a node by considering the number of direct connections it has, indicating how well-connected or influential it is within the network. Nodes with high degree centrality are often considered as hubs or key players within the network.

Degree Centrality (Directed Graph)Example



Node	In-Degree	Out-Degree	Centrality	Rank
A	1	3	1/2	**1**
B	1	2	1/3	**3**
C	2	3	1/2	**1**
D	3	1	1/6	**5**
E	2	1	1/6	**5**
F	2	2	1/3	**3**
G	2	1	1/6	**5**

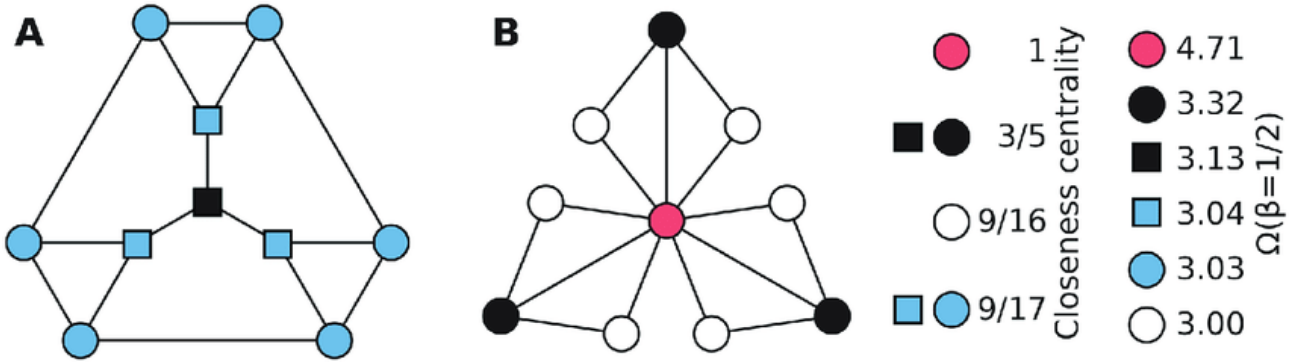
Normalized by the maximum possible degree

$$C_d^{\text{norm}}(v_i) = \frac{d_i}{n-1}$$

9

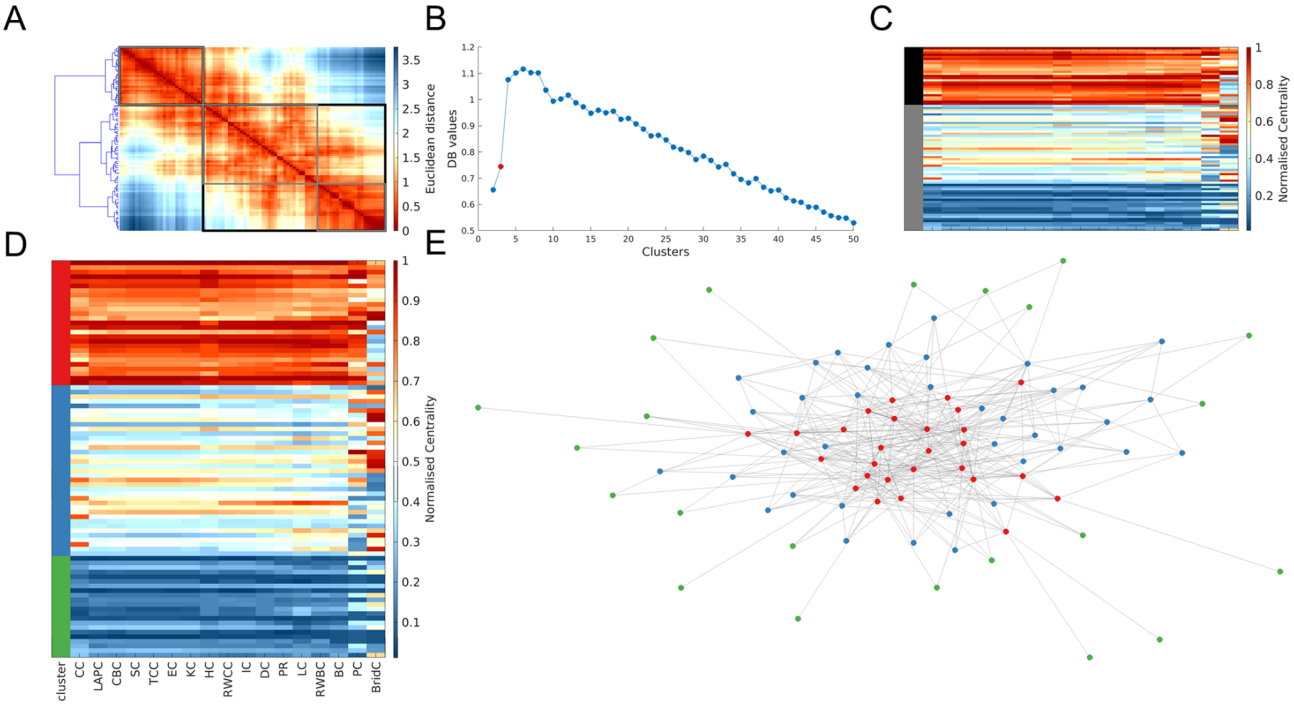
Closeness Centrality:

2. Closeness centrality measures how close a node is to other nodes in terms of the shortest paths between them. It quantifies the efficiency of communication or information flow from a node to all other nodes in the network. Nodes with high closeness centrality are those that can reach other nodes quickly and directly, acting as central points of information exchange. Closeness centrality is calculated by summing up the lengths of the shortest paths from a node to all other nodes and then calculating the reciprocal of this sum. Nodes with high closeness centrality are often crucial for disseminating information efficiently throughout the network.



Katz Centrality:

3. Katz centrality is a measure that considers both the direct connections and the indirect connections (higher-order paths) of a node within a network. It assigns centrality scores to nodes based on the sum of the contributions from their immediate neighbors and all other nodes in the network. Katz centrality allows for capturing the influence of nodes that may not have many direct connections but are connected to other highly central nodes. It assigns higher centrality scores to nodes that have direct connections to other central nodes and to nodes that are connected to many other nodes in the network. The centrality score of a node in Katz centrality is determined by a combination of its direct connections and the influence it receives from other nodes in the network.



- Katz centrality: $x_i = \frac{1}{\lambda_1} \sum_j A_{ij} x_j + \beta$

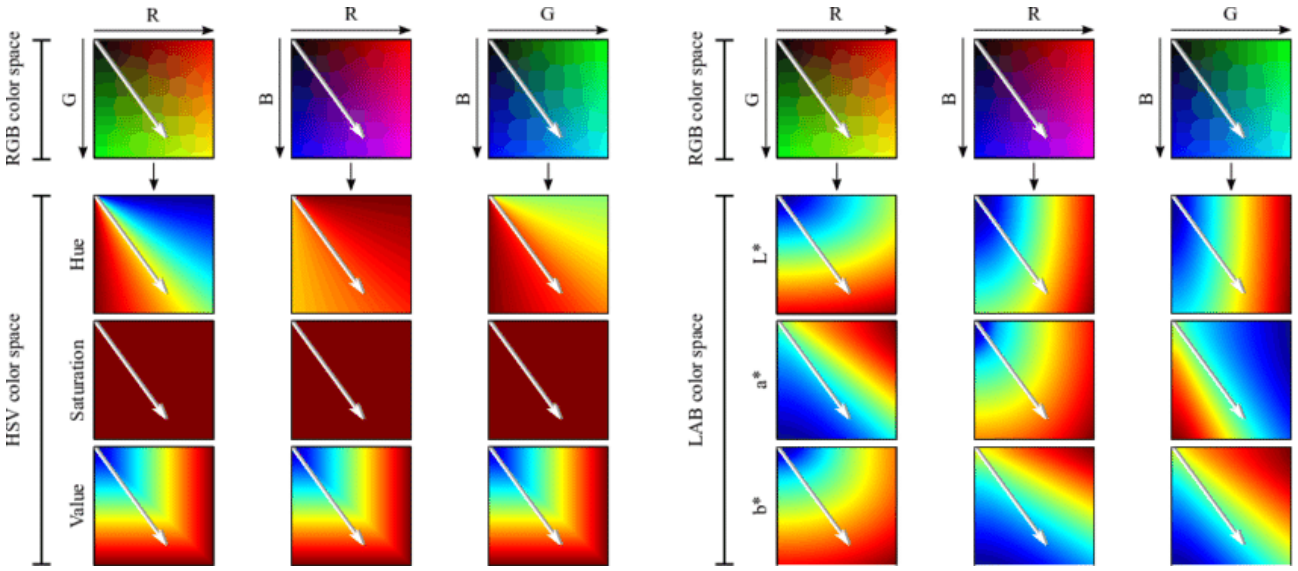
Degree centrality, closeness centrality, and Katz centrality are all widely used centrality measures in network analysis, each providing unique insights into the importance and influence of nodes within a network. These centrality measures are applied in various domains, including social network analysis, information flow analysis, recommendation systems, and analyzing the structure and dynamics of complex networks.

1.1.4. Statement Embedding

Statement embedding is a process similar to sentence embedding, but tailored specifically for code statements. It involves manually assigning numerical representations, typically in a five-dimensional space, to different components of code statements such as keywords, variables, built-in functions, user-defined functions, and special symbols. This embedding captures the structural and contextual information of code statements, enabling analysis and comparison based on their embeddings. By representing code statements in a compact numerical form, statement embedding facilitates tasks such as code search, code recommendation, and similarity analysis in the context of programming languages.

1.1.5. Image Vector

An RGB image vector refers to the representation of an image as a numerical vector in the RGB color space. RGB stands for Red, Green, and Blue, which are the primary colors used to create a wide range of colors in digital images.
In an RGB image, each pixel is composed of three color channels: red, green, and blue. The intensity values of these channels determine the color and brightness of the pixel. In a typical 24-bit RGB image, each channel is represented by 8 bits, allowing for 256 intensity levels (0-255) for each color channel. Therefore, the combination of these three channels can represent approximately 16.7 million different colors (256^3).



1.2. Challenges

The implementation of the project presents several key challenges that require careful consideration and expertise. These challenges are outlined below:

1.2.1. Recursion Flow in Control Dependency Graph

Handling recursion flow, the relation between function calls and return statements are very challenging parts in Control Dependency Graph. It takes a huge amount of time and labor.

1.2.2. Calculating Data Dependency:

Determining data dependencies, especially for unary and relational operators, is an intricate task that demands meticulous attention to detail. Unraveling the intricate relationships between variables for relational operators requires substantial time and concentration, making it a time-consuming and mentally demanding aspect of the project.

1.2.3. Max Eigenvalue:

The search for the maximum eigenvalue involves employing sophisticated algorithms that inherently consume a significant amount of computational resources and time. In this project, the Power Iteration method has been chosen as the approach to calculate the maximum eigenvalue. However, comprehending the underlying principles of linear algebra and understanding the intricacies of this method necessitates a comprehensive grasp of the subject matter.

1.2.4. Katz Centrality:

Katz centrality, being one of the complex centrality measures, poses considerable challenges. Implementing Katz centrality involves deciphering the intricacies of its formula, including the calculation of inter-node dependencies utilizing variables such as alpha (a value smaller than the inverse of the maximum eigenvalue) and beta (an influential factor in the dependency calculation). The iterative nature of the process and the need for careful computations make the implementation of Katz centrality a highly demanding task.

1.2.5. Statement Embedding:

The process of embedding the five essential features, namely keywords, built-in functions, user-defined functions, variables, and special symbols, into image dimensions presents a core challenge in this project. Expanding the dimensions through bitwise operations further intensifies the complexity. Moreover, the standardization of code, creation of a comprehensive vocabulary, and encoding the features in a manner that ensures their unique regeneration after decoding requires meticulous attention to detail and significant time investment.

Addressing these challenges is vital for the successful implementation of the project. Overcoming these hurdles demands a comprehensive understanding of the underlying concepts, meticulous planning, and diligent execution. By navigating these complexities, the project can unlock valuable insights and achieve the desired outcomes.

2. Project Overview

This project contains some segments and modules. These are described below:

2.1. Standardization

Code segments and statements have been standardized in this module. For example white space and extra new lines have been removed, curly braces have been organized in a standard form. For this segment file reading and writing skills are required.

2.2. Segmentation

Defining blocks for code segments has been performed here. Basically a code block is identified using its opening and closing curly braces.This is similar to stack operation. Thus basic knowledge in stack is needed there.

2.3. Built-in-flow

In this module the control flow has been defined for built in words specially keywords such as if-else if-else, switch-case, loops (while, for, do while), continue, break etc.

2.4. Function and Recursion flow

How control shifts in functions among prototypes, function body and function calls and in recursive calls, return statements is performed here. It is one of the challenging parts of the project.

2.5. Control Dependency Graph

Defining flow for keywords, loops, functions and recursion in a summed up for builds CDG.

2.6. Data Dependency Graph

In this module data flow in variables for operators has been updated.

2.7. Degree Centrality

Degree Centrality has been counted from in-degrees and out-degrees in a node.

2.8. Closeness Centrality

In this segment it is counted how close the nodes are of the graph representing the program.

2.9. Katz Centrality

Defines how the neighboring nodes in all levels up to infinity influence the adjacent nodes being counted in this module.

2.10. Normalization (Not statistical normalization)

In this module variables and functions have been given a similar name to avoid the conflicts among variables and functions.

2.11. Statement Embedding

Code Statements have been embedded for five features such as keywords, variables, functions, symbols etc.

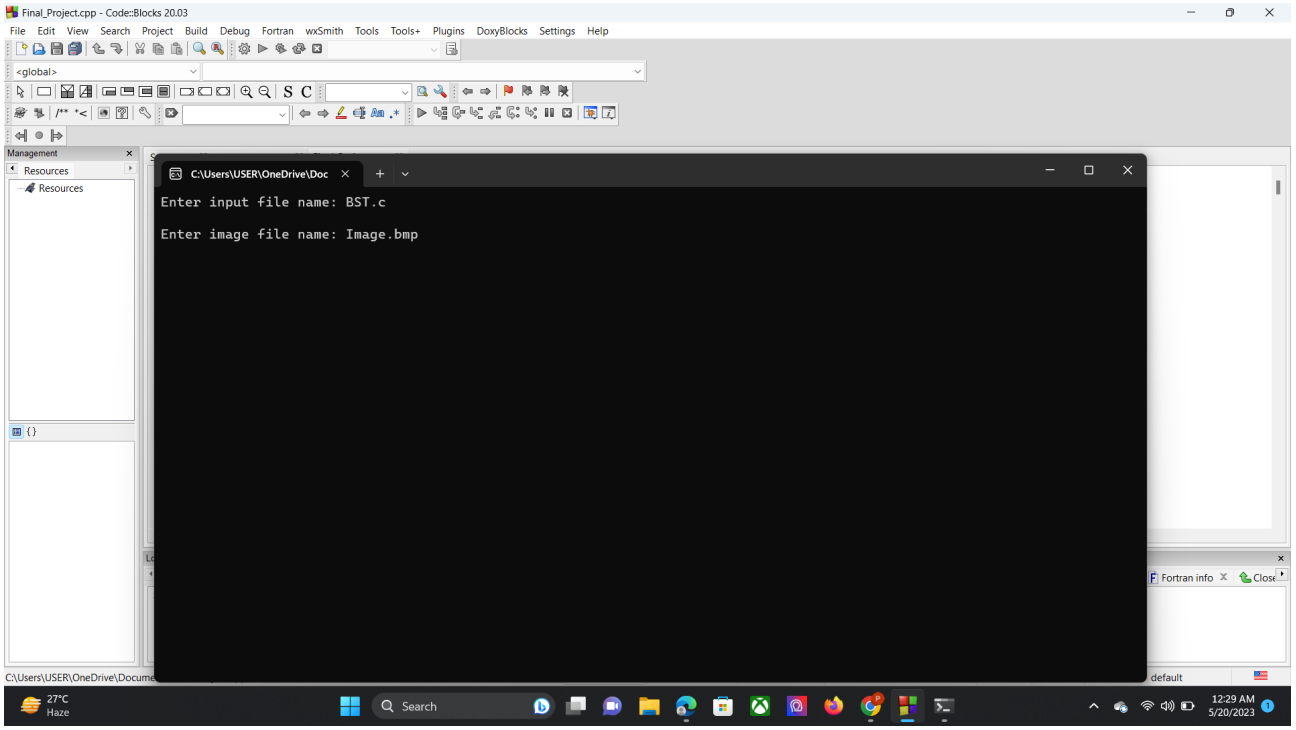
2.12. Image Vector

By scalar multiplication of centralities and embedded vectors, RGB vectors have been generated. And finally from these RGB vectors, the final image has been generated.

3. User Manual

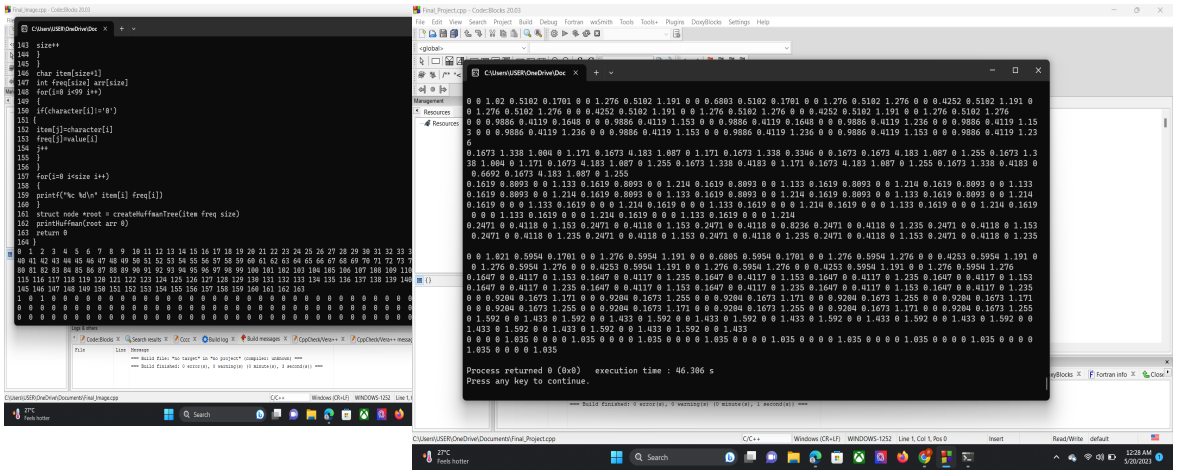
3.1. Selecting Files

First of all the user has to import an C code file from his current repository where the application is running and then give a name for the image file of corresponding code to be converted into (N.B. image.bmp format).



3.2. Background Output

Then after clicking the enter key, the program will run for a while. The user can observe what is happening now as the terminal/ window shows processing outputs.



In the above picture, a program dependency graph has been generated. The user can see it on the app window.

3.3. The Final Output

The final image is given which the user can check from his image folder.

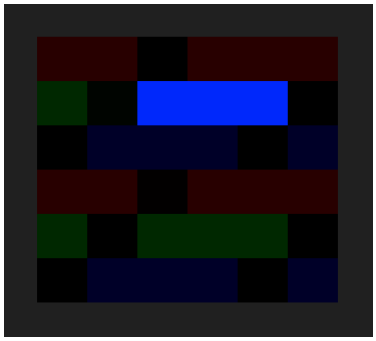


Image for “Hello World” program in C.

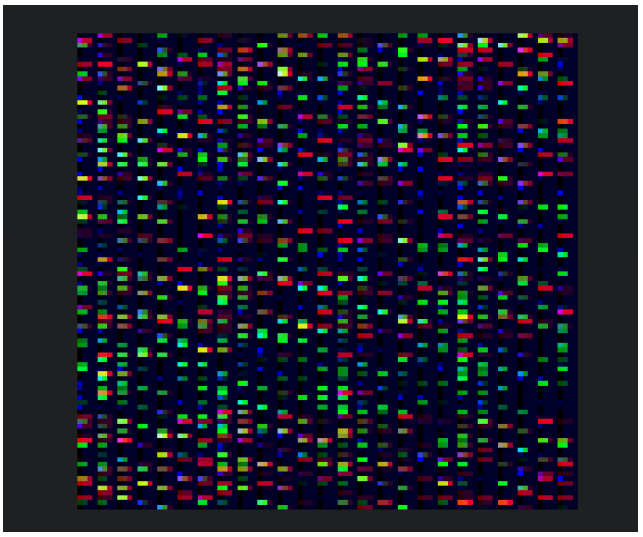


Image for “BST.c” (Binary Search Tree)

4. Conclusion

The CodeToImage project presents a unique approach to embedding statement-level information of a C program into image vectors. Throughout the implementation process, we encountered several key challenges that required careful consideration and expertise.Calculating data dependency proved to be a complex task, particularly for unary and relational operators. The identification of variable dependencies for relational operators demanded significant time and concentration. Additionally, finding the maximum eigenvalue using the Power Iteration method posed computational and conceptual challenges, as it required a solid understanding of linear algebra principles.Implementing Katz centrality, a complex centrality measure, required a deep understanding of its formula and intricate calculations involving inter-node dependencies. This process involved working with variables such as alpha and beta and performing multiple iterations, further intensifying the implementation challenge.

Furthermore, the embedding of statement-level information into image vectors posed a significant hurdle. This process involved assigning numerical representations to keywords, built-in functions, user-defined functions, variables, and special symbols. Expanding the dimensions through bitwise operations and ensuring the unique regeneration of up to five features after decoding required meticulous attention to detail and considerable time investment.
Despite these challenges, the CodeToImage project represents a novel approach to representing code statements visually. By successfully overcoming these hurdles, we have paved the way for potential applications in code search, recommendation systems, and similarity analysis within programming languages.Moving forward, further research and development are warranted to optimize the efficiency and accuracy of data dependency calculations, eigenvalue computations, and Katz centrality implementations. Additionally, exploring advanced techniques for statement embedding can enhance the quality and robustness of the image vectors generated.
In conclusion, the CodeToImage project signifies an important step towards bridging the gap between code representation and visual understanding. By embedding statement-level information into image vectors, we open up new possibilities for analyzing and manipulating code in a visually intuitive manner.

5. Appendix

References

[1] https://www.geeksforgeeks.org/katz-centrality-centrality-measure/, GeeksforGeeks, last accessed on 06 Mar 2023

[2] https://www.sciencedirect.com/topics/computer-science/degree-centrality/, ScienceDirect, last accessed on 12 Feb 2023

[3] https://cseweb.ucsd.edu/classes/fa12/cse231-a/lecture-14.pdf, University of Colorado Boulder, last accessed on 03 Jan 2023

[4] https://stackoverflow.com/questions/7904651/how-to-generate-program-dependence-graph-for-c-program, Stack Overflow, last accessed on 05 Jan 2023

[5] https://en.wikipedia.org/wiki/Power_iteration, Wikipedia, last accessed 07 Feb 2023