UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Variable Dependency Graph Summarization

Marie Therese Mikalsen

INF-3981 Master’s Thesis in Computer Science - June 2024

Supervisors

Main supervisor:	Elisavet Kozyri	UiT The Arctic University of Norway, Faculty of Science and Technology, Department of Computer Science

“We are surrounded by data, but starved for insights.”
–Jay Baer

“The goal is to turn data into information, and information into insight.”
–Carly Fiorina

Abstract

In personalized software, collected user data is used to give a tailored user experience. A user might be interested in understanding how their data (inputs) resulted in their personalized output. The Variable Dependency Graph (VDG) can explain how inputs of a program flow to the output. However, with increasing program size, there is a need for summarizing the VDGs and understanding these summarizations.

The first contribution of this thesis is the exploration of a fitting technique to summarize the VDG for a user. The result of the search was the technique TG-SUM, which supports all the VDGs graph characteristics that this thesis identified, as well as facilitates the visualization for a user. The second contribution of this thesis was to help a user interpret the VDG summarization by experimenting with how different code patterns tend to be summarized.

Acknowledgements

First and foremost, I want to thank my supervisor Elisavet Kozyri for her support, guidance, and expertise throughout my thesis. Her insights and suggestions have been invaluable for this research and working with her has been a wonderful experience. I also want to thank my partner, friends, and family for their love and support.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

CFG	Control Flow Graph
SSA	Static Single Assignment
VDG	Variable Dependency Graph

1 Introduction

Our daily routines are increasingly influenced by software, with smartwatches analyzing our sleep, streaming services suggesting music and movies, applications help manage our finances, etc. These types of software leverages data collected about us to personalize our experience. While software personalization enhances user experience, the users are left unaware of how their data leads to their personalized output. With an increase in this software, users of the software might be interested in understanding how their collected data (inputs) results in their personalized output.

An explanation of how the inputs flow to the output(s) can be illustrated in a Variable Dependency Graph (VDG), where the nodes correspond to the program variables and the edges correspond to the dependencies between them. An analysis of a VDG makes it possible to understand the data flow from a program's inputs to its output(s). A VDG can be generated for a program with the use of dependency analysis. However, as the program's size increases, the VDG size correspondingly increases, causing the graphs to be less understandable for humans.

Listing 1.1: C program to determine daily calorie need

```c
int main() {
    char gender;
    int age, weight, height, activityLevel;
    float bmr, dailyCalories;
    (receive input)
    if (gender == 'M'  gender == 'm') {
        bmr = 88.362 + (13.397 * weight)
        + (4.799 * height) - (5.677 * age);
    } else (gender == 'F'  gender == 'f') {
        bmr = 447.593 + (9.247 * weight)
        + (3.098 * height) - (4.330 * age);
    }

    // Activity level can be from 1-5
    if (activityLevel == 1) {
        dailyCalories = bmr * 1.2;
    } else if (activityLevel == 2) {
        dailyCalories = bmr * 1.375;
    } else if (activityLevel == 3) {
        dailyCalories = bmr * 1.55;
    } else if (activityLevel == 4) {
        dailyCalories = bmr * 1.725;
    } else if (activityLevel == 5) {
        dailyCalories = bmr * 1.9;
    }
    (print result)
    return 0;
}
```

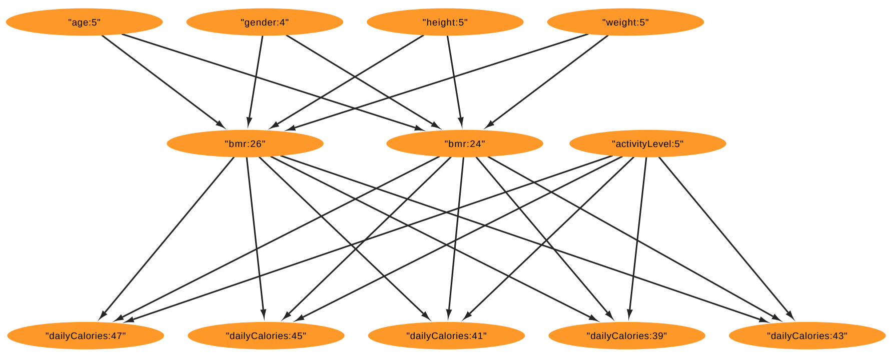


Figure 1.1: VDG from the code in Listing 1.1

To illustrate the potential complexity of a VDG, we consider a program collecting data about a user´s gender, age, weight, height, and activity level to calculate the user's daily calorie needs. The program can be viewed in listing 1.1. This program has been generated using ChatUit [2], with a prompt to create a program calculating a user's daily calorie need based on personal inputs. The VDG generated from the program can be viewed in figure 1.1. Looking at the graph, it can be seen that a quite small program can result in a graph that could benefit from simplification.

With large graphs, there is a need for summarization without losing valuable information for the user. In a preceding project [3], a search was performed for graph summarization techniques applied to dependency graphs similar to VDGs. The search discovered that graph summarization techniques have not been explored for the case of VDGs. Therefore, that project created two building blocks to find a fitting technique for VDGs. The first was a modified dependency tool to generate VDGs based on the input of program code, and the second was an overview of the graph summarization field.

1.1 Thesis statement

With the result from the preceding project [3], this thesis investigates how a VDG can be summarized without losing valuable information for a user. After finding a fitting technique for the VDG, this thesis also wants to help users interpret the summarization. With these intentions, this thesis aims to address the following research questions:

1. How can a variable dependency graph of a program be summarized to decrease its size without losing valuable information for a user?

2. How can a user interpret the summarization of a variable dependency graph?

1.2 Contribution

This thesis has primarily two contributions, first, a fitting summarization technique to fulfil research question 1. Second, experimentation of how different code patterns are summarized to help the user interpret the summarization to fulfil research question 2.

1.3 Context

The thesis is written in the context of the Cyber Security Group (CSG) at the University of Tromsø. The CSG group has three closely related research areas: fundamental systems, system support for healthy human beings and system support for sustainability [1]. Both for fundamental and sustainable systems there is a need for trust. Included under trust is transparency, where a system's explainability can help achieve transparency, and explainability can be attained through data flow control. This thesis falls under the CSG group work in trust, with the main purpose of explainability.

1.4 Thesis Outline

The remainder of the thesis outline:

Chapter 2: Background introduces foundational concepts pertinent to the thesis.

Chapter 3: Investigating variable dependency graph summarization presents the preceding project of this thesis.

Chapter 4: Choosing a Graph Summarization Technique explains the process of researching and identifying graph summarization techniques fitting for VDGs and the choice of summarization technique.

Chapter 5: Design presents the design of the chosen technique and the dependency tool modifications.

Chapter 6: Interpreting the VDG summarization experiments with the summarization technique to help users interpret the VDG summarization.

Chapter 7: Discussion discusses the choices and results of the thesis and how they satisfy the research questions, and makes suggestions for future work.

Chapter 8: Conclusion summarizes the thesis.

2 Background

2.1 Dependency Analysis

With the help of dependency analysis, program parts that are dependent on each other can be identified. Dependency analysis of programs has been used for compiler optimization [4, 5, 6] and to enhance program understanding [7] for many years. In compiler optimization, dependency analysis can help support different optimizations e.g. uncover possible parallelism opportunities [5]. On the other hand, dependency analysis for a better understanding of a program can improve maintainability, security and trust [7].

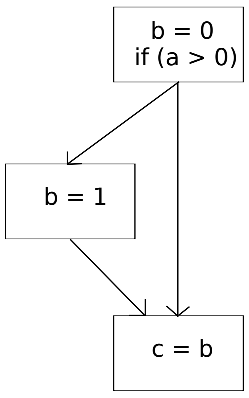
Notice how this thesis uses dependency analysis to enhance the program understanding at a variable level. This analysis results in a VDG, where nodes are variables and edges are dependencies between them.

2.1.1 Dependencies

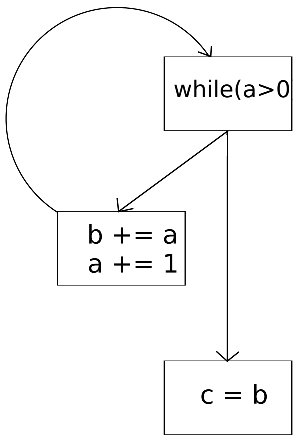
Dependencies between variables in a program can either be explicit or implicit. An explicit dependency between variables is when one variable uses another variable in its assignment. In the assignment $a = b$, the variable a explicitly depends on variable b, resulting in a VDG where $b \longrightarrow a$. Implicit dependencies occur within conditional commands, e.g. **if** statements. In the **if** statement $if\ a > 0\ then\ b = 2$, the variable b implicitly depends on variable a, resulting in a VDG where $a \longrightarrow b$.

2.1.2 Control Flow Graph (CFG)

The control flow of a program is the order the program is executed, and a control flow analysis can be used for compiler optimization [8, 9]. The control flow of a program can be represented in a Control Flow Graph (CFG). Each node in the CFG consists of consecutively executed code called basic blocks, while the edges are the transfer of control between basic blocks. Examples of control transfer in a program are conditional commands (e.g. **if** and **while** statements), function calls, **goto** statements, and **return** statements.



(a) CFG for **if** statements



(b) CFG for **while** statements

Figure 2.1: CFG examples

As an example, when encountering an **if** statement a program will either enter it (when the condition is true) or not (when the condition is false). A CFG when encountering an **if** statement can be viewed in figure 2.1a. For a **while** statement a program will also either enter it or not, however, if it was entered it will at the end link back to the **while** condition to check whether the condition has changed. An example of a CFG for a **while** statement can be viewed in figure 2.1b.

2.1.3 Static Single Assignment (SSA)

In Static Single Assignment (SSA) form variables are unique and can therefore only be assigned once. When translating into SSA form, each reassignment of a variable results in a new variable being created. SSA has been applied for compiler optimization [9, 10], and in the paper *What's In a Name?* [11] examples are shown of how SSA can remove false dependencies in program code.

Listing 2.1: Code example that can lead to false dependecies

```
int main() {
    int a = 5;
    int b = a * 2;
    ...
    b = 2;
    int c = b + 3;
}
```

To illustrate how SSA removes false dependencies view listing 2.1. In this code, variable b uses variable a in its first assignment. Later in the code, variable b is reassigned with no dependencies, before variable c uses variable b in its assignment. Without SSA, the VDG from this code would be $a \longrightarrow b \longrightarrow c$. However, since b has been reassigned with no dependencies before c uses it this leads to a false dependency between the variables a and c. Applying SSA would remove this false dependency and the resulting VDGs could be $a_1 \longrightarrow b_1$ and $b_2 \longrightarrow c_1$.

When applying SSA to capture dependencies with conditional commands present, there is a need for a CFG. Looking back at the CFG example in figure 2.1a, when the variable c uses the variable b in its assignment it can either be the variable b assigned before the **if** statement or inside **if** statement. Viewing the CFG for the **while** statement in figure 2.1b, when variable b is assigned, it is dependent on variable a assigned before the **while** statement, and on a assigned inside the loop (e.g. on the second+ iteration). Therefore, as the assignment in the loop can occur several times (several iterations of the loop) it is more complex to support.

2.2 Graph Summarization

Graphs help visualize data and are particularly useful with interconnected datasets[1]. With the increasing amount of interconnected data, the necessity for simplification became apparent [15, 16] and a method to simplify these datasets are with graph summarization.

1. The term interconnected datasets [15, 16] is used in the graph summarization field to refer to datasets where the data items are linked to each other through various relationships (e.g., social networks)

2.2.1 Graphs

A graph consists of a tuple (V, E) where V is the set of vertices or nodes, and E is the set of edges connecting the nodes. Graphs have various characteristics [15, 17], and several are presented in table 2.1.

Table 2.1: Graph Characteristics

Characteristic	**Counterpart**	**Description**
Static	Dynamic	A static graph is unchanging, whereas a dynamic graph can change over time.
Directed	Undirected	The edges have a direction ($\longrightarrow$) in directed graphs, whereas the edges are bidirectional ($-$) in an undirected graph.
Weighted	Unweighted	The edges in a weighted graph have weights, indicating edges have different amounts of significance. Whereas the edges in an unweighted graph have no weights.
Labelled	Unlabelled	In a labelled graph nodes and/or edges have labels, whereas there are no labels in an unlabelled graph.
Cyclic	Acyclic	In a cyclic graph, a node is reachable from itself, whereas there are no cycles in an acyclic graph.
Simple	Non-simple/Multi-graphs	Simple graphs have no loops or multiple edges between two nodes, whereas non-simple graphs/multi-graphs can have them.

3 Investigating variable dependency graph summarization

This chapter presents the report *Investigating variable dependency graph summarization* [3], which was the result of my capstone project in 2023. This capstone project resulted in a modified dependency tool to capture the variable dependency graph from a program code and an overview of the graph summarization field.

3.1 Dependency tool

Before choosing a dependency tool the desired properties of the dependency analysis had to be decided. With dependency analysis, we wanted to capture both the explicit and implicit dependencies. Another desired property was SSA, to remove false dependencies. With SSA, the need for a CFG followed, resulting in the desired properties: explicit dependencies, implicit dependencies, SSA and CFG.

The modified dependency tool in the capstone project was srcSlice [12, 13]. srcSlice is a lightweight tool for static program analysis. It was chosen because it is open-source, allows modification, and captures explicit dependencies between variables. srcSlice has also been used by other research papers, including one using it to generate VDGs [14]. The expected input file for srcSlice is an XML format provided by a tool from the same authors named srcML, where srcML supports conversion of C, C++, C# and JAVA.

srcSlice had to be modified to capture implicit dependencies, CFG and SSA. Modified srcSlice worked at a function level and supported variable assignments, **if** statements (including **else if**) and simple **while** statements at delivery. It did not support conditional commands inside **while** statements. Type rules were used to explain how dependencies were captured by the tool. However, the type rules did not include the implementation of SSA.

Listing 3.1: While example

```
1 int main() {
2     int a = 5;
3     while(a < 10){
4         a += 2;
5     }
6 }
```

In the capstone project, SSA was implemented by including the line number in the variable name when assigned. As mentioned, supporting SSA inside **while** statements are complex as there can be several iterations. With the capstone project´s implementation of SSA in mind, look at listing 3.1. Here the dependency analysis will at the first analysis of the loop, capture the dependency $a_2 \longrightarrow a_4$ both explicitly and implicitly. However, on a potential second iteration of the loop, the dependency would be $a_4 \longrightarrow a_4$. Therefore, with SSA implemented there is a need for a second execution of the dependency analysis on the loop to capture both of these dependencies. Two executions of the dependency analysis are sufficient as the variable's name won't change after the second analysis.

3.2 Graph Summarization field

Obtaining an overview of the graph summarization field in the capstone project was based on three recent surveys from 2018 [15], 2020 [16], and 2023 [17]. Division of the summarization methods into categories was based on the two latter. The methods are divided into graph clustering, statistical inference, and goal-driven summarization. The newest techniques under Graph Neural Network (GNN) were not included in this overview.

3.2.1 Graph Clustering

In clustering techniques clusters are identified within the graph. Nodes in the cluster should be closely connected to each other, and be sparsely connected to the other nodes. Clustering is often divided into structural and attribute clustering. Structural clustering bases itself on a graph´s structure such as the topology and connectivity of nodes. In attributed clustering, nodes' attributes are used when finding clusters and this is particularly useful when the nodes have rich attribute information.

3.2.2 Statistical Inference

Statistical inference techniques are divided into pattern mining and sampling. In pattern mining, patterns such as frequent subgraphs are identified in a graph or a graph database. In sampling techniques, the result is a sample of the original graph, and such techniques are often used to estimate graph properties when processing the entire graph is expensive [17].

3.2.3 Goal-Driven Summarization

Goal-driven techniques create a summarization to meet a specific goal, and under this are query-driven and utility-driven techniques. In query-driven techniques, the summary graph consists of the query-relevant parts of the original graph. Utility-driven techniques summarize a graph where the main goal is to preserve a desired utility. Influence-based techniques, often applied on social networks, have the main goal of preserving the most influential nodes and/or edges within the graph, and were placed under the utility-driven techniques in the capstone project.

4 Choosing a Graph Summarization Technique

This chapter explains the process of finding a summarization technique fitting for the VDGs outputted by our modified dependency tool.

4.1 VDG Characteristics

The VDG produced by our modified dependency tool is static, as it does not change over time, and unweighted, meaning no edges have more impact than others. The graph is directed, as a statement $x = y$ would mean a flow of information from y to x, but not a flow from x to y. The nodes in the graph are labelled with their name, while the edges have no labels. In a program, a variable can give information to itself (e.g. $a = a + 1$), and therefore self-loops may occur, resulting in a cyclic and non-simple graph.

4.2 Identify Fitting Techniques

The VDGs produced by the dependency tool are static, directed, unweighted, labelled, and can have self-loops. All of these characteristics should be accounted for when choosing a fitting summarization technique. Another aspect is that the purpose of the summarization is to summarize to ease visualization without losing too much information for the user. Therefore, a fitting technique should fulfil or be modified to fulfil the criteria listed below:

1. VDG characteristics

2. Visualization without losing too much information for a user

4.2.1 Clustering

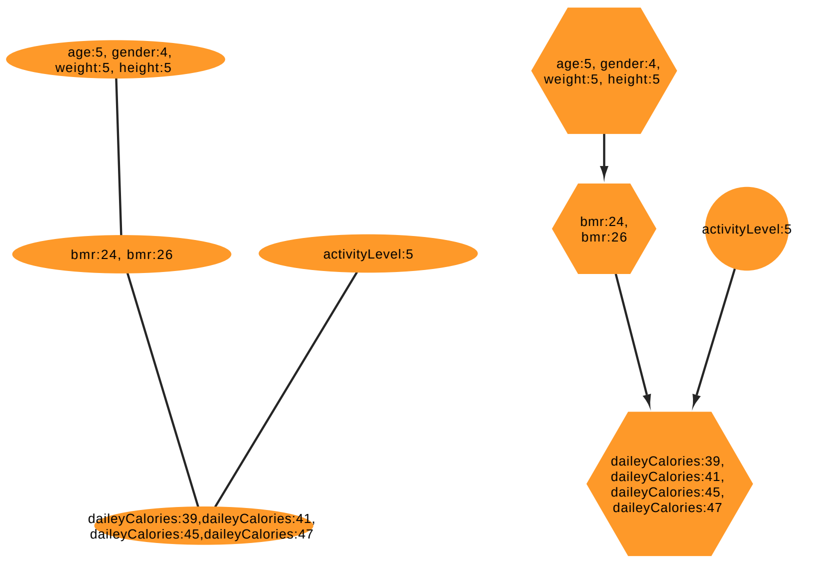
Within the clustering techniques, the structural clustering techniques are more fitting as the nodes in our VDGs do not have rich attribute information. Clustering is among the most popular summarization techniques [15], and there are many techniques to choose between. However, the search narrows since most techniques are designed for simple undirected graphs [15, 20]. When exploring clustering techniques complying with one or both of the criteria listed above, papers such as *Graph summarization with quality guarantees*[1] [19] and *Summarizing Labeled Multi-graphs*[2] [20] were found.

In the paper by Riondato et al. [19], the graph summary aims to enhance query efficiency. This is achieved by storing a lossy or lossless graph summary in the main memory that can accurately answer queries. The technique applies to simple, static, undirected graphs, as well as accommodates unweighted graphs, weighted graphs, and self-loops. This technique would need modifications to support directed and labelled graphs, where the paper mentions measures for supporting directed graphs.

TG-SUM [20] generates a lossless summary to enhance the visualization and is designed to support several different graph characteristics. The technique applies to static graphs and accommodates undirected, directed, unweighted, weighted, unlabeled, and labelled graphs, as well as edge multiplicities and self-loops. With all these graph characteristics supported, there is no need to alter this technique to support the outputted VDGs.

1. Source code available at: `https://github.com/rionda/graphsumm`
2. Source code available at: `https://github.com/DimBer/TGsum`

Both of these papers had their code for the summarization techniques available. Therefore, it was possible to easily experiment with how they summarized VDGs produced by our tool. One VDG used in experiments is the VDG from the introduction (figure 1.1). The technique by Riondato et al. would not summarize the graph without any input parameters (e.g., supernodes in the summary), while TG-SUM summarized this graph into four clusters.


age:5, gender:4, weight:5, height:5
bmr:24, bmr:26
activityLevel:5
daileyCalories:39,daileyCalories:41, daileyCalories:45,daileyCalories:47
age:5, gender:4, weight:5, height:5
bmr:24, bmr:26
activityLevel:5
daileyCalories:39, daileyCalories:41, daileyCalories:45, daileyCalories:47

(a) Summary with technique by Riondato et al. **(b)** Summary with TG-SUM

Figure 4.1: Cluster summarization of the graph in figure 1.1

Since TG-SUM resulted in four clusters, the input of four supernodes was tested in the technique by Riondato et al. for several runs. The result was the same four clusters as found by TG-SUM at each run. Several runs were also tested for different inputs of supernodes, and in these cases, the nodes in the clusters were more likely to vary from each run. This could indicate that the clustering with four supernodes is the optimal summarization.

In figure 4.1a, the summarization with the technique by Riondato et al. can be viewed with the input parameter of four supernodes. While in figure 4.1b, the summarization by TG-SUM is presented. Comparing the summarizations, it can be seen that the technique by Riondato et al. does not support directed graphs and that TG-SUM has added features to enhance the visualization e.g. size of supernodes and glyphs.

4.2.2 Statistical Inference

With the statistical inference techniques, pattern mining techniques are of more interest than sampling techniques. Sampling techniques sample part of the graph and would therefore not comply with the purpose of visualization without too much information loss. Pattern mining techniques on the other hand can simplify the visualization of a graph by identifying and summarizing frequent patterns/subgraphs into glyphs. With a balance between simplification and information preservation, the purpose of visualization without too much information loss could be achieved.

When exploring pattern mining techniques complying with one or both of the criteria listed above, the paper *Motif Simplification: Improving Network Visualization Readability with Fan, Connector, and Clique Glyphs* [21] by Dunne and Shneiderman was found. In the paper, a pattern mining technique identifies patterns and replaces them with meaningful glyphs. This technique supports static undirected graphs and has been made available as part of the NodeXL network analysis tool [22]. NodeXL is a plugin for Microsoft Excel that works on the Windows operating system and has a free and paid version.

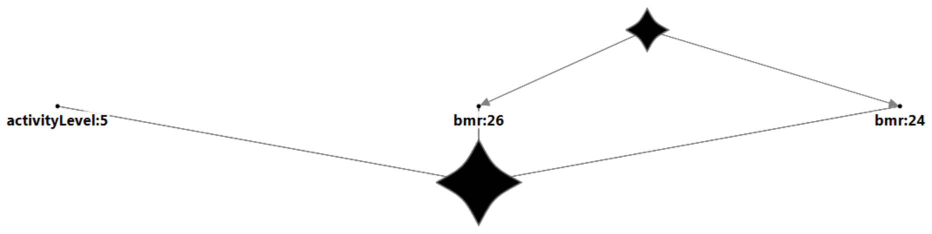


Figure 4.2: Summarization of the graph in figure 1.1 with the technique by Dunne and Shneiderman

With the free version of NodeXL, it was possible to experiment with how the technique summarized VDGs produced by our tool. In figure 4.2, the graph from the introduction (figure 1.1) has been summarized with the technique. Viewing the figure, two glyphs have been found, one glyph for all the inputs to the "*bmr*" variables and the other glyph for all the output variables. The edges in the graph do have a direction as the NodeXL tool has the option of selecting whether a graph is directed or undirected. Still, the technique expects an undirected input graph [15, 21].

4.2.3 Goal-Driven

Within the goal-driven techniques, utility-driven techniques are more fitting than query-driven techniques. Query-driven techniques retrieve the query-relevant parts of the graph and would therefore not comply with the purpose of visualization without too much information loss. As mentioned, several techniques can fall under utility-driven techniques, where techniques such as clustering can be used to preserve the utility. Whether these techniques are defined as the technique used to preserve the utility, or as a utility-driven technique can differ.

When exploring utility-driven techniques complying with one or both of the criteria listed above, papers such as *DepComm: Graph Summarization on System Audit Logs for Attack Investigation* [18] and *RDF graph summarization for first-sight structure discovery*[3] [26] were found.

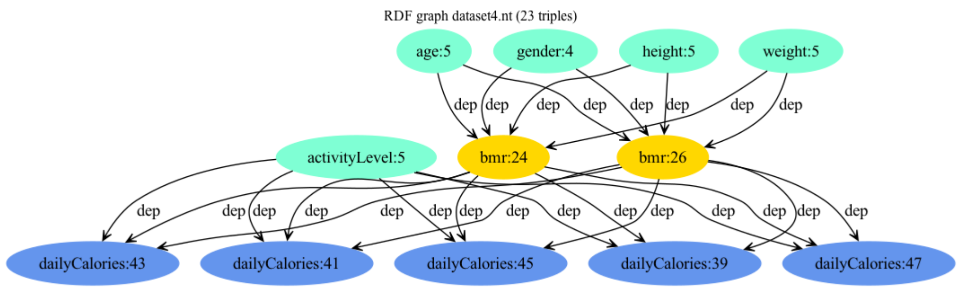
DepComm [18] applies summarization on dependency graphs to accelerate the analysis of the graph. At first glance, DepComm seemed to fit both criteria well. However, since their dependency graph is generated from system audit logs and the summarization is tailored to their purpose of attack investigations it cannot be used to fulfil our criteria.

The paper by Goasdoué et al. [26] aims to summarize Resource Description Frameworks (RDF) graphs to enhance the first-sight analysis of the graph. An RDF graph has data triples, where e.g. *Jane studies English* would result in the triplets *(Jane, studies, English)*. For the VDG, this could be converted from $x \longrightarrow y$ to be represented as *(x, dep, y)*. As the paper had the code for the summarization techniques available, it was possible to experiment with how they summarized VDGs produced by our tool. One VDG used in experiments is the VDG from the introduction (figure 1.1).

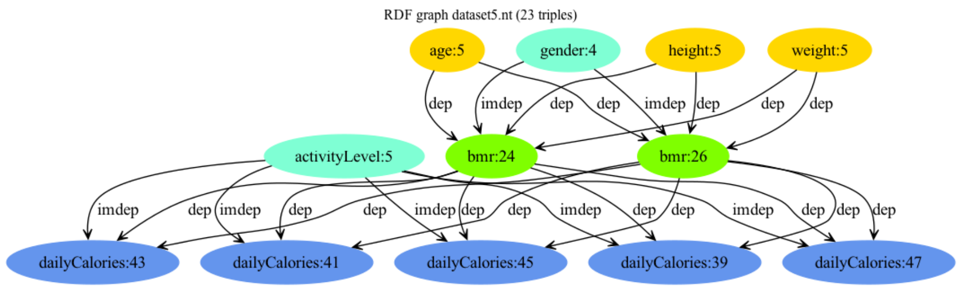
In figure 4.3a, the graph from the introduction can be viewed as an RDF graph with one type of dependency between variables. While in figure 4.3b, the graph from the introduction can be viewed as an RDF graph with a distinction between explicit and implicit dependencies. The technique´s summarizations of these RDF graphs can be viewed in figure 6.13.

The summarization with one dependency type viewed in figure 6.13a resulted in three supernodes. The first supernode consists of all input nodes (nodes with outgoing edges), the second supernode of the middle nodes (nodes with incoming and outgoing edges), and the last supernode of the output nodes (nodes with incoming edges). The summarization with two dependency types

3. Source code available at: `https://gitlab.inria.fr/cedar/RDFQuotient`

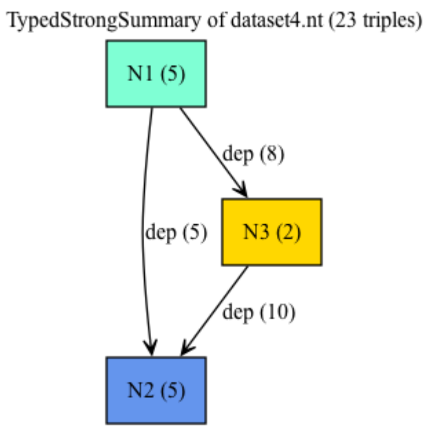


(a) One type of dependencies between variables

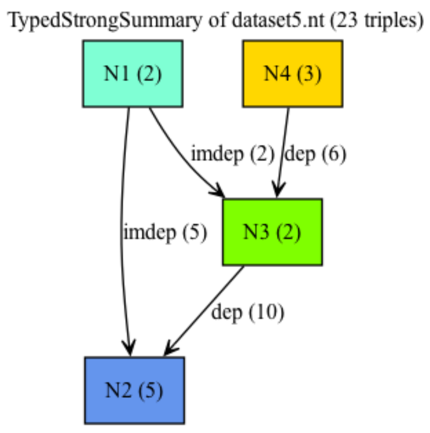


(b) Distinction between explicit and implicit dependencies between variables

Figure 4.3: Graph from figure 1.1 as RDF graph



(a) Summary of one type of dependencies between variables



(b) Summary with distinction between explicit and implicit dependencies between variables

Figure 4.4: Summarizations with the technique presented by Goasdoué et al.

viewed in figure 6.13b resulted in four supernodes. The difference from the summary in figure 6.13a is that the first supernodes containing the input nodes were divided into two supernodes, where the input nodes either have explicit or implicit dependency types.

Other techniques explored under utility-driven techniques are influence-based techniques. This technique has often been applied to social networks. In social networks, people influence people, while in a VDG, it can be viewed as variables that influence other variables. Some influence-based techniques for static directed graphs are CSI [23], SPINE [24], and VEGAS [25].

CSI and SPINE use past information propagation from the social network in their algorithms. This gives them a temporal aspect, as they use temporal activities in the network when summarizing [15]. This log of past propagation cannot be generated with our static analysis of the program code, and the temporal aspect is not wanted. Therefore, both CSI and SPINE were not appropriate.

VEGAS does not use information propagation and is fit for labelled graphs. The VEGAS algorithm summarizes citation networks to enhance visualization based on a user's interest. This algorithm starts with selecting a source node, which can be one of the papers in the citation network, and then summarizes based on the selected source node. As this technique summarizes based on a source node it does not comply with the purpose of visualization without too much information loss.

4.2.4 The Chosen Summarization Technique

Looking at the experimentation of the clustering techniques it can be seen that TG-SUM is a better fit than the technique by Riondato et al. The reason is that TG-SUM supports all the VDG graph characteristics (criterion 1) and has a supergraph output with visualization enhancements (supports criterion 2 better).

In the pattern mining technique by Dunne and Shneiderman, the glyphs found are the same as two of the supernodes by the clustering techniques. Compared to TG-SUM, this summarization technique also has visualization enhancements, with the use of glyphs and sizes corresponding to the contained nodes (criterion 2). One of the differences in the techniques is that TG-SUM supports more of the graph characteristics of the outputted VDG e.g. directed edges, and self-loops (criterion 1). As TG-SUM has the benefits of the visualization properties and has more support for the outputted VDGs characteristics it is the better choice of the two.

The experimentations with the RDF summarization technique show that the algorithms take the incoming edges, the outgoing edges, and the edge labels into account when summarizing. The summarization results show false edges e.g. the supernode includes the node "*acitivityLevel:5*" and points to the supernode containing "*bmr:24*". A conclusion from experimenting with such a summarization technique could be that the labels of edges in a VDG have less importance than the graph´s structure.

After exploring several techniques of different categories, the TG-SUM technique was chosen. TG-SUM is lossless, supports all the graph characteristics of the outputted VDG, and has multiple features for enhanced visualization. With all these traits, the technique outcompeted the other graph summarization techniques found.

5 Design

In this chapter, the design of the chosen technique TG-SUM is presented, as well as the modifications made to the dependency tool in this thesis.

5.1 TG-SUM

TG-SUM [20] is a graph summarization technique that generates a lossless supergraph to enhance the visualization, where it is designed to support several different graph characteristics. The output from TG-SUM is a summary graph file, where the supernode contains a label, the number of nodes it consists of, a glyph, and the number of edges it summarizes. The glyph representing the super node can be a clique, in-star, out-star, or disconnected set.

This technique utilizes the two-part Minimum Description Length (MDL) paradigm to find the summarization that minimizes the total description cost. The total description includes the bits to encode the summary graph and the bits to encode the corrections needed to restore the original graph.

When searching for a summary a two-step process is applied where they first find possible groups of nodes, they named candidate sets, before deciding which candidate sets to merge into super nodes. MDL is used to determine whether a candidate is good. A good candidate should be large (reduce the bits to encode the summary graph) and have high quality (not increasing the bits required to encode corrections too much).

Candidate sets are ordered in a list based on their size and quality before the list is processed and the candidates are merged into supernodes. In this process, the total cost is monitored, and a candidate set is only merged if it does not decrease the quality of the summarization too much. A benefit of this merging process is the ability to summarize at multiple resolutions. For measuring the quality of candidates, the technique has extended the Jaccard similarity metric to support directed graphs. The grouping of nodes is based on their similarity by utilizing Locality Sensitive Hashing (LSH).

5.2 Dependecy tool Modification

5.2.1 Modified Type Rules

Type rules were created in the capstone project [3] to explain how the dependency tool captured the dependencies between variables. These rules have been expanded to be more accurate. In this expansion, a new term named *active locations* is introduced. Active locations refer to the current versions a variable can be at the given time of a program. Looking back at listing 2.1, the active versions of b on lines 4 to 26 would be b_1, while the active version of b from line 26 would be b_2.

In the type system, G is the set of dependencies so far in the program ($a_\ell \longrightarrow b_\ell$, etc.), V is the set of active locations of a variable at that time in the program ($a{:}\ell_1, \ell_2$, etc.), ctx is the set of variables implicitly influencing the program at that time of the program (e.g. inside an **if** statement with the condition $a > 0$ the ctx would consist of a_ℓ), **e** is for the variables in an expression, and **c** for commands. The function $LA(x, V)$ in the type system has the input of a variable x and the active variables in V. The output is the active variable versions in V (e.g. $L(a, V)$ returns $a{:}\ell_1$ and $a{:}\ell_2$). The notation $ctx, G, V \vdash \mathbf{c}, G', V'$ signifies that according to the context ctx, set of dependencies G, and set of active variables V, the command **c** is type correct and results in the new set of dependencies G' and the new set of active variables V'.

$$\frac{G' = G \cup \{ctx \longrightarrow \mathbf{x}_\ell \quad LA(\mathbf{e}, V) \longrightarrow \mathbf{x}_\ell\} \quad V' = V[x \mapsto \ell]}{ctx, G, V \vdash \ell:\ \mathbf{x{:}{=}e},\ G', V'} \tag{5.1}$$

The assignment rule for the type system can be viewed in equation 5.1. The explicit dependencies are captured with $LA(e, V) \longrightarrow x_l$, here the function LA

is used to find the currently active locations of the variable(s) in **e** and the active variable(s) is added as a dependency to x_l. The implicit dependencies are captured with $ctx \longrightarrow x_l$, and the active variable version of x is added or with the statement $V' = V[x \mapsto \ell]$, which indicates that the active location of variable x is set to location ℓ.

Listing 5.1: Assignment rule example

```
1 int main() {
2       int a = 5;
3       int b = a;
4 }
```

In listing 5.1, the assignment rule would first be applied to variable a, where there would be no dependencies, so G would remain empty. However, the set of active variable versions V would be updated with a:2. On the assignment of b, b:3 would be added to V, and the function $LA(a, V)$ would output a_2, resulting in the dependency $a_2 \longrightarrow b_3$.

$$\frac{ctx' = ctx \cup LA(\mathbf{e}, V) \quad ctx', G, V \vdash \mathbf{c1},\ G_1, V_1 \quad ctx', G, V \vdash \mathbf{c2},\ G_2, V_2}{ctx, G, V, \vdash \ell:\ \textbf{if e then c1 else c2 end},\ G_1 \sqcup G_2, V_1 \sqcup V_2} \tag{5.2}$$

The type rule of **if** statements can be viewed in equation 5.2. Commands inside the **if** statements are correct if the variable(s) in the condition is captured in the context. Therefore, when an **if** statement is encountered the function LA is used to find the currently active locations of the condition variable(s) (**e** in the equation) and augment the context with these when the commands in the **if** statement are analysed. After the analysis of the branches, the set of dependencies from the different branches are merged and the active locations of variables from the different branches are merged.

Listing 5.2: If rule example

```
 1 int main() {
 2       int a = 5;
 3       int b = 0;
 4       if(a > 0){
 5             b = 1;
 6       }else{
 7             b = 2;
 8       }
 9       int c = b;
10 }
```

In listing 5.2, the assignment rule would first be applied to variables a and b before the **if** statement is encountered. The **if** rule would then ensure that the condition variable a:2 is added to the context during the **if** statement with the use of function $LA(a, V)$. When the commands inside the **if** statements have been analysed V_1 will have replaced b:3 with b:5, and G_1 will consist of $a{:}2 \longrightarrow b{:}5$. In the **else** however, V_2 will have replaced b:3 with b:7, and G_2 will consist of $a{:}2 \longrightarrow b{:}7$. After the merging, the resulting V consist of $(b{:}5, 7)$, while the resulting G consists of $(a{:}2 \longrightarrow b{:}5, a{:}2 \longrightarrow b{:}7)$. When the assignment rule is executed on variable c, the function $LA(b, V)$ will return b:5 and b:7 leading to the dependencies $b{:}5 \longrightarrow c{:}9$ and $b{:}7 \longrightarrow c{:}9$.

$$\frac{\begin{array}{c}\text{ctx1} = \text{ctx} \cup LA(e, V) \quad ctx1, G, V \vdash \mathbf{c},\ G_1, V_1 \\ \text{ctx2} = \text{ctx} \cup LA(e, V_1) \quad ctx2, G_1, V_1 \vdash \mathbf{c},\ G_2, V_2\end{array}}{ctx, G, V \vdash \ell :\ \textbf{while e do c end},\ G \sqcup G_2, V \sqcup V_2} \tag{5.3}$$

Equation 5.3 shows the type rule for a **while** statement. The body of a **while** loop needs two analyses to be correct, as explained with listing 3.1 in a previous chapter. Similarly as in the **if** statements, commands inside the **while** statements are correct if the variable(s) in the condition is captured in the context. Therefore, when a **while** statement is encountered the function LA is used to find the currently active locations of the condition variable(s) (**e** in the equation) and augment the context with these when the commands in the **while** statement are analyzed.

After the first analysis, a second analysis is started with the set of dependencies and active variable locations from the first analysis. After the second analysis, the set of dependencies from the second analysis and before encountering the **while** statement are merged ($G \sqcup G_2$). The active locations of variables from the second analysis and before encountering the **while** statement are also merged ($V \sqcup V_2$). The reason is that the program might or might not have entered the **while**.

Looking back at listing 3.1, the assignment rule would first ensure V added a:2. When encountering the **while** statement, the **while** rule would add a:2 to its ctx, and during the first analysis capturing the dependency $a_2 \longrightarrow a_4$, both explicitly and implicitly, as well as replacing the active version of a in V to a:4. In the second analysis, the function $LA(a, V)$ would return a_4 and capture the dependency $a_4 \longrightarrow a_4$, both explicitly and implicitly.

Notice how this thesis uses the implementation of SSA and CFG to implement the type system rules, where SSA is implemented with the use of the line number the variable is assigned, and the CFG is used to know when and what to merge with conditional commands.

5.2.2 While statements

The **while** implementation from the capstone project was not sufficient and needed modification to support **if** and **while** statements inside it. **While** statements need a second analysis, and one solution could have been to do a second parse through the **while** loops, straight after it finished. However, as srcSlice uses a SAX parser to parse through the XML document, and this parser retains no memory, this was not an option. Since changing the parser would be too much work the solution was to save the events occurring inside the **while** loops and do a second run-through of these events. This implementation modification ensures the program supports the type rule in equation (5.3).

5.2.3 File format for summarization tool

The summarization tool expected two files, one label file and one edge list (graph file). The label file consists of node IDs and their labels, while the edge list consists of two node IDs in the format "*source target*". As an example, the dependency $a_2 \longrightarrow b_3$, where node a_2 had ID 1 and node b_3 had ID 2, would result in "1 2". For the tool to be able to produce these files, each variable first needed to have an identity number. This identity number is given as they are encountered in the program.

While experimenting with the summarization technique it was also noticed that the order of the edge list could in some cases alter the result of the summarization. For this reason, the edge list is sorted to ensure a consistent summarization result. An option to reverse the ordering has also been added to support more experimentation.

6 Interpreting the VDG summarization

In this chapter, we experiment with the summarization technique TG-SUM to help users interpret VDG summarization. When analyzing code, we identified VDG structures that correspond to distinctive code patterns. These code patterns occurred within the source code analyzed but could also occur in real-world scenarios. This chapter will start by presenting the identified graph structures, how they tend to be summarized, and code patterns that result in the graph structures. After we know how the graph structures tend to be summarized, we will present graph structure combinations in programs to show examples of how the found summarization tendencies transfer to larger graphs with several structures.

TG-SUM supports four different node shapes in the summarizations, a circle for nodes not in a cluster, a square for cliques, a triangle for in- and out-stars, and a hexagon for disconnected sets. The different glyphs summarize different underlying structures, and it is useful to understand these structures to interpret the summarization better. Therefore, an explanation is provided for the supported glyphs in table 6.1.

Table 6.1: Glyph Interpretation

Glyph	Description
Clique	All the nodes in the cluster are connected.
In-stars	One central node has outgoing edges to all the other nodes in the cluster.
Out-star	One central node with incoming edges from all the other nodes in the cluster.
Disconnected Set	No edges between the nodes in the cluster.

6.1 Graph Structures

6.1.1 Linear

A linear graph structure can occur in the VDG when a variable uses one variable in its assignment, and that variable already used one other variable in its assignment, and so on. An example code resulting in a linear graph structure can be viewed in listing 6.1, and the linear graph structure is provided in figure 6.1. The experimentation with the linear graph structure revealed that the structure tends not to be summarized.

Listing 6.1: Code resulting in the structure in figure 6.1

```
int main() {
 int a = 5;
 int b = a * 2;
 int c = b * 3;
 ...
 return 0;
}
```

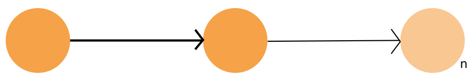


Figure 6.1: Linear structure

6.1.2 Many-to-One

Figure 6.2 shows a graph structure with many input nodes pointing to one output node. This is present in a VDG when one variable uses several other variables in its assignment. Listing 6.2 provides a code example resulting in the graph structure. Experimentation with this graph structure revealed that it tends to be summarized as one supernode with an in-star glyph (triangle), where the central node of the cluster is the output node. The summarization can be viewed in figure 6.3.

Listing 6.2: Code resulting in the structure in figure 6.2

```
int main() {
 int a = 2;
 int b = 3;
 ...
 int sum = a + b + ...;
 return 0;
}
```

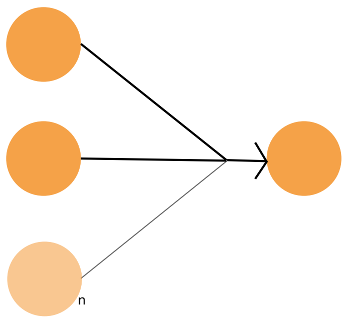


Figure 6.2: Many-to-one structure

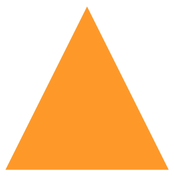


Figure 6.3: Summarization of the graph structure in figure 6.2

6.1.3 One-to-Many

A graph structure with one input node pointing to several output nodes (presented in figure 6.4) can be found in a VDG with explicit and implicit dependencies. In the explicit dependency case, this structure occurs when several variables use the same variable in their assignments. A code example with explicit dependencies resulting in the structure can be viewed in listing 6.3.

In the implicit dependency case, this structure occurs when one variable is used in the conditional command for the entire **if** statement. Then variables are assigned within the statement. An example code is provided in listing 6.4. The experiments uncovered that the structure tends to be summarized as one supernode with an out-star glyph (triangle), where the central node of the cluster is the input node. The summarization of this structure can be viewed in figure 6.5.

Listing 6.3: Explicit dependecies resulting in the structure in figure 6.4

```
int main() {
  int a = 5;
  int b = a + 1;
  int c = a + 2;
  ...
  return 0;
}
```

Listing 6.4: Implicit dependecies resulting in the structure in figure 6.4

```
int main() {
  int a = [0,n];
  int res = 0;
  if(a==0){
    res = 1;
  }else if(a==1){
    res = 2;
  }...
  return 0;
}
```

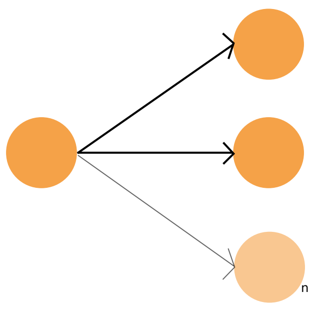


Figure 6.4: One-to-many structure

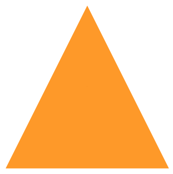


Figure 6.5: Summarization of the graph structure in figure 6.4

6.1.4 All-to-All

A graph structure with each input node pointing to all output nodes can be viewed in figure 6.6. This graph structure can occur in a VDG with explicit and implicit dependencies. However, this structure was discovered when analyzing **if** statements (implicit dependencies). With implicit dependencies, the structure occurs when several variables are used in the conditional command for the entire **if** statement, and then variables are assigned within the statement. An example can be found in listing 6.6.

After discovering this structure from implicit dependencies, consideration was given to the possibility of the structure deriving from explicit dependencies. The conclusion was that this structure could occur with explicit dependencies if several variables used the same variables in its assignment. Listing 6.5 provides a code example with explicit dependencies. Looking at this example compared to the implicit code example, this example seems more contrived (e.g. why would $b1$ not just be assigned $a1$ ($b1 = a1$)). However, the example is included to demonstrate the possibility of it occurring.

The tendencies discovered with experimentation of this structure are that it tends to summarize input nodes with all the same outgoing edges as one disconnected set (hexagon) and that it tends to summarize output nodes with all the same incoming edges as one disconnected set. This leads to the summarization viewed in figure 6.7.

Listing 6.5: Explicit dependecies resulting in the structure in figure 6.6

```
int main() {
 int a = 1;
 int b = 2;
 ...
 int a1 =
   a + b + ...;
 int b1 =
   a + b + ...;
 ...
 return 0;
}
```

Listing 6.6: Implicit dependecies resulting in the structure in figure 6.6

```
int main() {
 int a = 1;
 int b = 2;
 ...
 int res = 0;
 if(a>0 && b>1
         && ...){
    res = 1;
 }else if(a>1 &&
    b>2 && ...){
    res = 2;
 }...
 return 0;
}
```

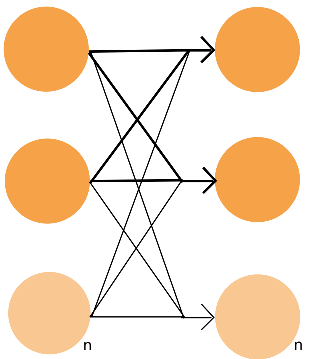


Figure 6.6: Many-to-many structure

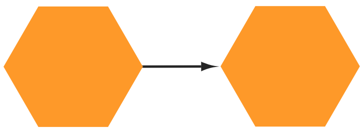


Figure 6.7: Summarization of the graph structure in figure 6.6

6.1.5 One-to-All-to-One

Figure 6.8 shows a graph structure with one input node pointing to all middle nodes, then each middle node pointing to one output node. This structure can be derived from both explicit and implicit dependencies. In the explicit scenario, this structure occurs when several variables use the same variable in their assignments, and then these variables are used by one variable in its assignment. An example is provided in listing 6.7.

In the implicit scenario, this structure occurs when one variable is used in the conditional command for the entire **if** statement. Then the same variable is reassigned in each branch before it is used by another variable, in its assignment, outside the **if** statement. View listing 6.8 for an example code with implicit dependencies. The experimentation revealed that middle nodes with the same incoming and outgoing edges tend to be summarized as one disconnected set (hexagon). Figure 6.9 shows the summarization.

Listing 6.7: Explicit dependecies resulting in the structure in figure 6.8

```
int main() {
 int a = 5;
 int b = a*2;
 int c = a*3;
 ...
 int res =
  b + c + ...;
 return 0;
}
```

Listing 6.8: Implicit dependecies resulting in the structure in figure 6.8

```
int main() {
 int a = [0,n];
 int b = 0;
 if(a==0){
   b = 1;
 }...
 else if(a==1)
 ...
 else{
   b = n;
 }...
 int res = b;
 return 0;
}
```

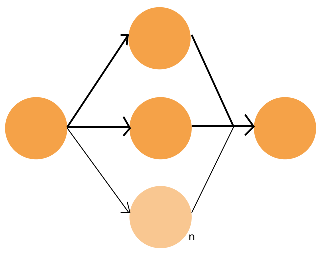


Figure 6.8: One-to-all-to-one structure

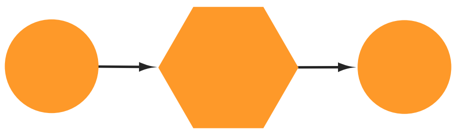


Figure 6.9: Summarization of the graph structure in figure 6.8

6.1.6 Some-to-Some

Figure 6.10 presents a graph structure with several input nodes with different outgoing edges to several output nodes. Both explicit and implicit dependencies can lead to this structure. This structure was discovered when analyzing **if** statements (implicit dependencies). The structures occur with implicit dependencies when different conditional command variables are used for each branch of the **if** statement. Then a variable is assigned in each branch. Listing 6.10 provides a code example with implicit dependencies.

Consideration was then given to the possibility of this structure deriving from explicit dependencies. For explicit dependencies, the structure could occur if one variable used n variables in its assignment, the next variable used $n - 1$, and so on. An example can be viewed in listing 6.9. This example is included to demonstrate the possibility of it occurring, but it seems more contrived than the implicit dependency example.

In this structure, input nodes with the most outgoing edges tend to be summarized as one disconnected set (hexagon), and output nodes with the most incoming edges tend to be summarized as one disconnected set. The resulting summarization can be viewed in figure 6.11.

Listing 6.9: Explicit dependecies resulting in the structure in figure 6.10

```
int main() {
 int a = 1;
 int b = 2;
 ...
 int a1 =
   a + b + ...;
 int b1 =
   b + ...;
 ...
 return 0;
}
```

Listing 6.10: Implicit dependecies resulting in the structure in figure 6.10

```
int main() {
 int a = 1;
 int b = 2;
 ...
 int res = 0;
 if(a>0){
     res = 1;
 }else if(b>2){
     res = 2;
 }...
 else{
     res = 20;
 }
 return 0;
}
```

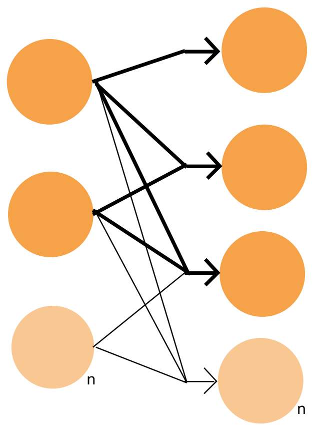


Figure 6.10: Some-to-some structure

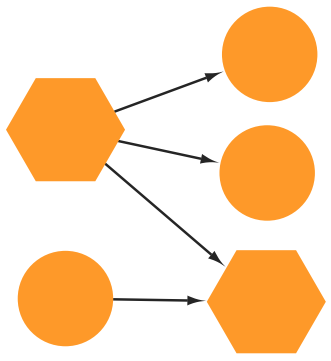


Figure 6.11: Summarization of the graph structure in figure 6.10

6.1.7 Some-to-Some-to-One

A graph structure with several input nodes with different outgoing edges to several middle nodes each pointing to one output node can be viewed in figure 6.12. This structure was discovered when analyzing implicit dependencies. The structure occurs when different conditional command variables are used for each branch of the **if** statement. Then a variable is assigned in each branch before they are used by another variable, in its assignment, outside the **if** statement. One code example with implicit dependencies resulting in the structure is provided in listing 6.12.

The structure could also occur with explicit dependencies if one variable used n variables in its assignment, the next variable used $n - 1$, and so on. Then all these variables would have to be used by a variable in its assignment. An example with explicit dependencies is provided in listing 6.11. This example is included to demonstrate the possibility of it occurring. However, it seems more contrived than the implicit dependency example.

The experimentation of this graph structure had more varying summarization results, both by adding more nodes to the structure and by changing the order of the edge list inputted to the technique. In some cases, it would be summarized similarly to the some-to-some structure above, where the input nodes with the most outgoing edges and middle nodes with the most incoming edges were clustered together. However, two other summarization tendencies did occur several times, both can be viewed in figure 6.13.

In figure 6.13a, one of the clusters consists of the input node with the most outgoing edges, the middle nodes with the least incoming edges, and the output node. This cluster is represented as an in-star, where the central node is the input node. The other cluster in this summarization consists of the middle nodes with the most incoming edges and is represented by a disconnected set.

In figure 6.13b however, the in-star cluster with an outgoing edge (represented as a clique if there were only two nodes in the cluster) consists of almost the same nodes as the in-star in figure 6.13a, except the output node is not included. The other cluster in this figure consists of the middle nodes with the most incoming edges and the output node. This cluster was represented by an out-star glyph, where the central node is the output node. Comparing the two summarizations, the difference is in which cluster the output node has been placed.

Another thing worth mentioning is how adding more output nodes with the same incoming edges altered the summarization. The technique would then prioritize summarizing the output nodes as a disconnected set, the middle nodes with the most incoming edges as a disconnected set, and the input node with the most outgoing edges together with the middle nodes with the least incoming edges as an in-star (more than two nodes in the cluster) or a clique (when only two nodes in the cluster).

Listing 6.11: Explicit dependecies resulting in the structure in figure 6.12

```
int main() {
 int a = 1;
 int b = 2;
 ...
 int a1 =
    a + b + ...;
 int b1 =
    b + ...;
 ...
 int res =
    a1 + b1 + ...;
 return 0;
}
```

Listing 6.12: Implicit dependecies resulting in the structure in figure 6.12

```
int main() {
 int a = 1;
 int b = 2;
 ...
 int res = 0;
 if(a>0){
     res = 1;
 }else if(b>2){
     res = 2;
 }...
 else{
     res = 20;
 }
 int output = res;
 return 0;
}
```

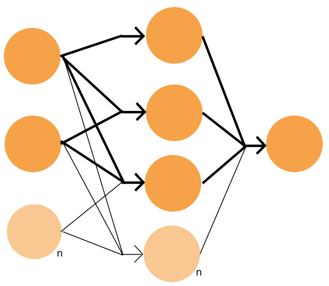
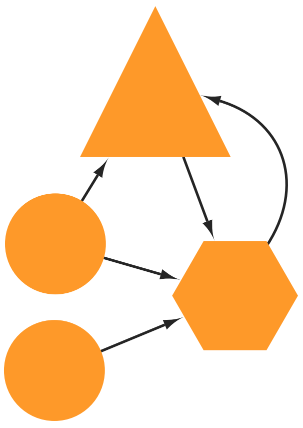
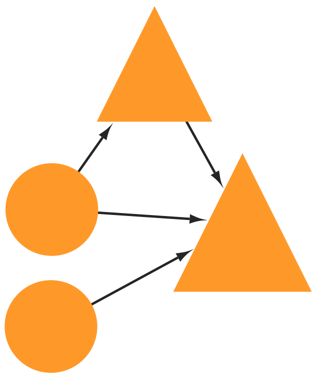


Figure 6.12: Some-to-some-to-one structure



(a) Summarization 1



(b) Summarization 2

Figure 6.13: Summarizations of the graph structure in figure 6.12

6.1.8 While

The graph structure from a simple **while** statement can be viewed in figure 6.14, and the code resulting in the graph is provided in listing 6.13. Notice how by removing the **while** loop, the alteration to the VDG would be that the self-loop of variable *i*:4 would be removed. The experimentation with the **while** graph structure revealed that this type of structure tends not to be summarized.

Listing 6.13: While loop

```
int main() {
 int i = 0;
 while(i<10){
     i+=1;
 }
 return 0;
}
```

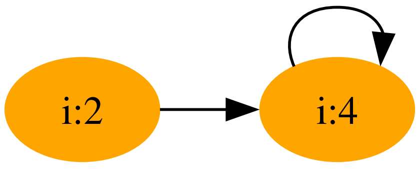


Figure 6.14: While graph structure

6.2 Testing Graph Structure Combinations in Programs

6.2.1 Introduction Code

In the code from the introduction viewed in listing 1.1, a program was created to determine daily calorie intake. The VDG from the program code can be viewed in figure 1.1, and the summarization of the graph in figure 4.1b.

Viewing this summarization, the input nodes with the same outgoing edges (*age*, *gender*, *weight*, *height*) are summarized as a disconnected set, the middle nodes with the same incoming and outgoing edges are summarized (*bmr*) as a disconnected set, and the output nodes with the same incoming edges are summarized (*dailyCalories*) as a disconnected set. The node with different incoming and outgoing edges (*activityLevel*) was not included in a cluster. This summarization conforms with the tendencies observed in the all-to-all and one-to-all-to-one structures.

6.2.2 Program Code 1

The program in listing 6.14 results in the VDG viewed in figure 6.15. This graph structure is a combination of the graph structures viewed in many-to-one and some-to-some structures. The program determines the gym membership cost for a month, the many-to-one structure occurs when the cost per month is calculated based on attended classes and the normal monthly price. The some-to-some structure occurs when the final cost is calculated, where a discount is applied to the normal monthly price based on the customer's membership.

The summarization of the graph can be viewed in figure 6.16. Viewing the summarization, the many-to-one structure was summarized as one out-star glyph. In the some-to-some structure, the input nodes with the most outgoing edges are summarized as one disconnected set and the output nodes with the most incoming edges are summarized as one disconnected set. This summarization conforms with the tendencies observed in the many-to-one and some-to-some structures.

Listing 6.14: Program to determine Gym Membership Cost this month

```
int main() {
 int gymMemnershipPerMonth = 250;
 int attendedClasses;
 char basicMember[3];
 char premiumMember[3];
 char VIPMember[3];
 (receive inputs)
 int costMonth = gymMembPerMonth+(attendedClasses*30);
 float finalMonthCost;
 if(strcmp(VIPMember, "yes") == 0){
   finalMonthCost = costMonth*0.8;
 }else if(strcmp(premiumMember, "yes") == 0){
   finalMonthCost = costMonth*0.9;
 }else if(strcmp(basicMember, "yes") == 0){
   finalMonthCost = costMonth*1;
 }
 (print result)
 return 0;
}
```

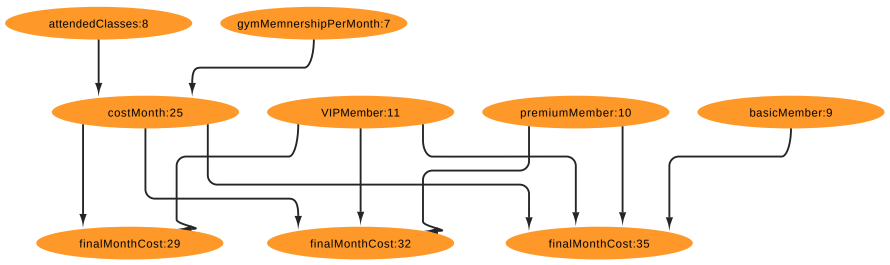


Figure 6.15: VDG from code in listing 6.14

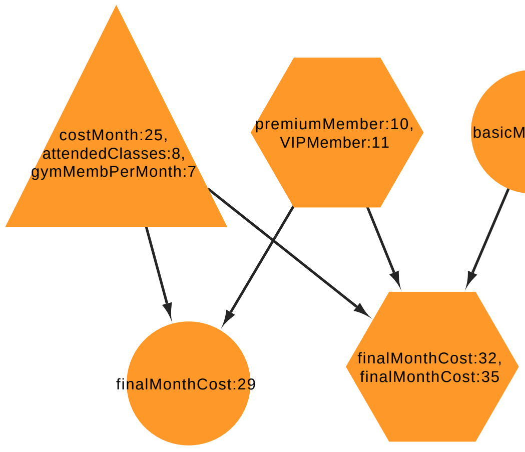

costMonth:25, attendedClasses:8, gymMembPerMonth:7
premiumMember:10, VIPMember:11
basicMember:9
finalMonthCost:29
finalMonthCost:32, finalMonthCost:35

Figure 6.16: Summarization of the VDG in figure 6.15

6.2.3 Program Code 2

The program presented in listing 6.15 results in the VDG viewed in figure 6.17. This graph structure is a combination of the graph structures viewed in linear, some-to-some-to-one, and one-to-many. The program determines the ticket price based on collected points and ticket type. The linear structure occurs when calculating the discount based on the user's points. The some-to-some-to-one structure occurs when determining the ticket discount based on the ticket type and then using the ticket discount to calculate the ticket price. While the many-to-one structure occurs when the new points are determined based on the final ticket price.

Viewing the summarization in figure 6.18, the linear structure was not summarized, the some-to-some-to-one structure was summarized similarly to one of the summarizations experienced when experimenting with some-to-some-to-one structure, viewed in figure 6.13b. Lastly, the many-to-one structure was summarized as one in-star glyph. This summarization conforms with the tendencies observed in the linear, some-to-some-to-one and many-to-one structures.

Listing 6.15: Program to calculate ticket price

```
int main() {
 int points;
 char child[3];
 char teen[3];
 char senior[3];
 char student[3];
 (receive inputs)
 int ticket_discount;
 if(strcmp(child, "yes") == 0){
   ticket_discount = 4;
 }else if(strcmp(teen, "yes") == 0){
   ticket_discount = 3;
 }else if(strcmp(senior, "yes") == 0){
   ticket_discount = 3;
 }else if(strcmp(student, "yes") == 0){
   ticket_discount = 2;
 }else{
   ticket_discount = 0;
 }
 int ticket_price = 5 - ticket_discount;
 int discount_with_points = points*0.01;
 float final_ticket_price =
       ticket_price-discount_with_points;
 int new_points;
 if(final_ticket_price > 2){
   new_points = 3;
 }else{
   new_points = 1;
 }
 (print result)
 return 0;
}
```

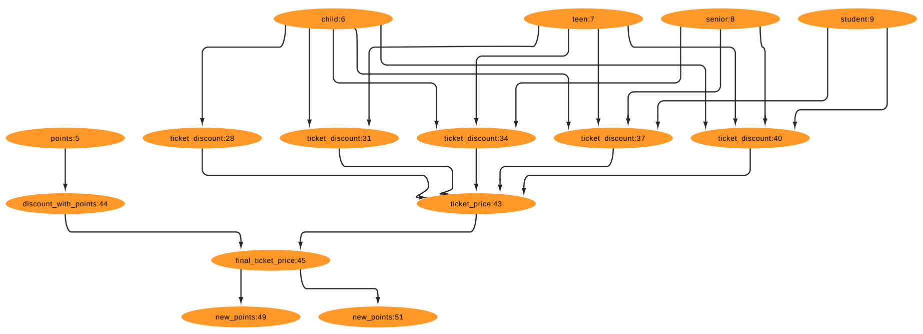


Figure 6.17: VDG from the code in listing 6.15

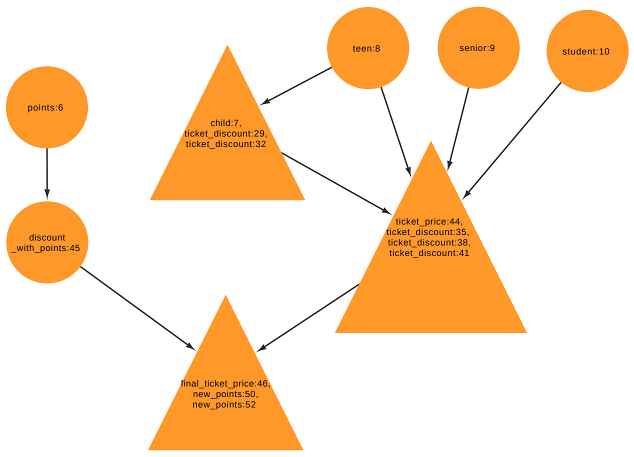

teen:8
senior:9
student:10
points:6
child:7, ticket_discount:29, ticket_discount:32
discount _with_points:45
ticket_price:44, ticket_discount:35, ticket_discount:38, ticket_discount:41
final_ticket_price:46, new_points:50, new_points:52

Figure 6.18: Summarization of the VDG in figure 6.17

6.2.4 Program Code 3

The program viewed in listing 6.16 results in the VDG in figure 6.19. In this program, a structure that tends to be summarized is placed inside a **while** structure that tends not to be summarized. This program determines the Grade Point Average (GPA) based on inputted grades. The structure placed inside the **while** statement is similar to a one-to-all-to-one structure, however, there is more than one input, making it an all-to-all-to-one structure. Without the **while** statement the input nodes would be summarized as one disconnect glyph, and the middle nodes (variables defined in **if** statement) would be summarized as one disconnected set.

The summarization of the graph is presented in figure 6.20. In the summarization, the input nodes have been summarized as one disconnected set, and all the middle nodes assigned in the **if** statement are summarized as a clique with a self-loop. As mentioned above, without the **while** statement this clique would have been a disconnected set. However, within a **while** loop all the middle nodes have outgoing edges to itself and all the other middle nodes inside the **if** statement, making the clique glyph more fitting, and a self-loop necessary. The iterator in the **while** statement has not been summarized, but the iterator's definition outside the **while** statement has been summarized with the other input nodes. However, the structure of the **while** statement with the first iterator pointing to the second definition in the loop that again points to itself has not been summarized.

This summarization conforms with the tendencies observed in the **while** structure and with the clusters in the all-to-all-to-one clusters. However, since placing it inside a **while** statement resulted in all the middle nodes having edges to each other and itself, the glyph was replaced with a clique and a self-loop added.

Listing 6.16: Program to calculate GPA

```
int main() {
 int num_grades = 8;
 char grades[num_grades];
 (receive inputs)
 int i = 0;
 int GPA = 0;
 while(i < num_grades){
   if(grades[i] == 'A'){
     GPA += 5;
   }else if(grades[i] == 'B'){
     GPA += 4;
   }else if(grades[i] == 'C'){
     GPA += 3;
   }else if(grades[i] == 'D'){
     GPA += 2;
   }else if(grades[i] == 'E'){
     GPA += 1;
   }
   i += 1;
 }
 GPA = GPA/num_grades;
 (print result)
 return 0;
}
```

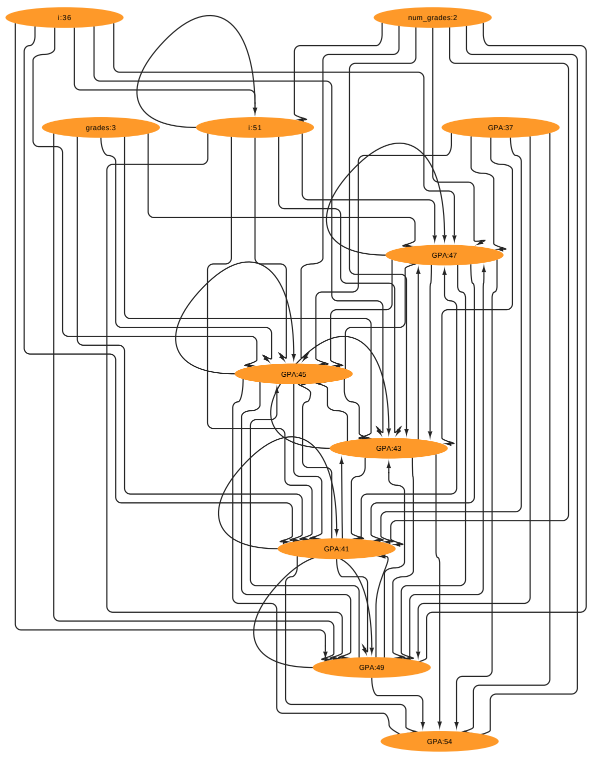


Figure 6.19: VDG from the code in listing 6.16

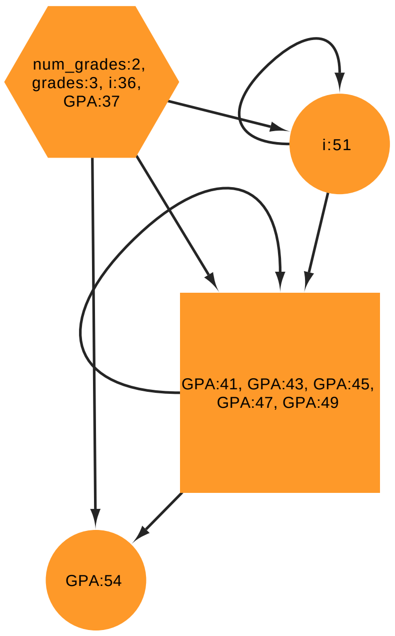

num_grades:2, grades:3, i:36, GPA:37
i:51
GPA:41, GPA:43, GPA:45, GPA:47, GPA:49
GPA:54

Figure 6.20: Summarization of the VDG in figure 6.19

7 Discussion

This chapter will discuss the choices and results of the thesis and how they satisfy the research questions from Chapter 1.1, as well as make suggestions for future work.

7.1 Summarizing the VDG

The first research question was how the VDG of a program could be summarized to decrease its size without losing valuable information for a user. Answering this research question required a tool to generate VDGs and exploration of the different techniques in the graph summarization field. My previous capstone project [3] started this process by finding and modifying a tool to generate VDGs with our desired properties from a program code, and by getting an overview of the graph summarization field.

In this thesis, the contribution to the dependency tool was expanding the type system to be more accurate, refactoring the implementation to support conditional commands inside **while** statements, and outputting the files needed by the chosen technique TG-SUM.

When choosing a graph summarization technique, the newest techniques under Graph Neural Networks (GNN) were not considered. The main reason is that the complexity of these models can make it hard to know how the summarization is derived. Therefore, as the main point of this thesis is to explain how the input results in the output, choosing a summarization technique where it can be hard to explain how the summarization is derived would be counterintuitive to the thesis purpose.

An extensive exploration was done in Chapter 4 to find a fitting summarization technique. Two criteria were created to determine whether a technique was a good fit. The first criterion was that the technique needed to support VDG characteristics, and the second criterion was to generate a summarization without losing too much information for the user. One of the arguments for not choosing several techniques was that the summarization output had too much information loss. To achieve a summarization without too much information loss we were looking for an overview of the entire graph (supergraph output) and not a sample of the graph or a customized summarization.

The result of our exploration was the technique TG-SUM. TG-SUM is lossless, outputs a supergraph, supports the VDGs graph characteristics, and has multiple features for enhanced visualization. With these traits, TG-SUM were able to fulfil the research question above.

Another interpretation of the research question, however, could be that the valuable information for a user is information the user is interested in. With this interpretation a technique such as VEGAS [25], where the summarization is based on user interest, would have been a better choice. In this case, there are two scenarios where we could know the user's interest and use this knowledge when summarizing.

In the first scenario, the summarization is personalized using collected data about the user. Then we need to remember that the VDG in this thesis is used to increase a user´s understanding of the program with explainability. A personalization of the summarization based on collected data would leave the users unaware of which of their collected data altered the summarization. Following this direction would therefore be counterintuitive to the thesis purpose.

In the second scenario, the summarization is based on user input. Then a user would be required to decide what they deem interesting or not, where they might not have enough knowledge to make such a decision. Therefore, creating an overview of the entire graph seemed the best approach. However, a user input summarization technique could be a good addition. This would allow the user to take a closer look at what they deemed interesting after reviewing the summarization overview.

7.2 Interpreting the VDG summarization

The second research question of this thesis was how to help a user interpret the VDG summarization. The contribution to answering this research question can be found in Chapter 6. For a user to interpret the summarization, they should have an underlying knowledge of what program code might be behind the summarization. Therefore, when analysing source code supported by our modified dependency tool (variable assignments, **if** statements and **while** statements), VDG structures corresponding to distinctive code patterns were identified. Afterwards, experimentation was done to show how these graph structures tend to summarize. With this process, the user has been helped to understand which graph structures are behind the summarization and which code pattern(s) can lead to this graph structure.

In addition to this knowledge, some graph structure combinations were tested to check whether this altered the previous findings. In the programs with graph structure combinations, most of the found tendencies were transferred to these combinations. One difference, however, was the summarization when placing graph structures inside a **while** loop. The **while** loop created edges between what outside the loop would have been a disconnected set. This resulted in the same cluster but with a clique glyph instead. Further exploration of how different graph structures behave inside a **while** statement could be interesting.

The graph structure combinations were presented with real-world programs instead of abstract programs. The main reason was to give a better understanding of how and where these graph structures could occur in the real world.

In some of the graph structures, it could be seen that two different code examples (explicit and implicit dependencies) lead to the same graph structure. As mentioned, some of these explicit examples seemed more contrived (all-to-all, some-to-some, and some-to-some-to-one). However, in the one-to-many and one-to-all-to-one structure, it cannot be distinguished between the two code examples with explicit and implicit dependencies. Therefore, these scenarios could justify making a distinction between explicit and implicit dependencies in the VDG.

In the graph structure some-to-some-to-one, the experimentation revealed different summarizations by altering the order of the edge list or adding more nodes. Before experimenting with this structure there was no ordering of the edge list. However, since this structure's summarization could be altered between analyses of the same code it was decided to order the edge list, to ensure a consistent summarization result. The edge list provided by the dependency tool can be in two directions, to enable experimentation with the order of the list in a more controlled environment.

A theory for why the order of the edge list could alter the output could be that it depends on which candidate sets are found at which time by the technique. The candidate sets are said to be ordered based on size or quality, but if this metric is quite similar it might just be about which candidate set was found at which time. This could be investigated and confirmed in the future by taking a deep dive into TG-SUM. However, that different parameters altered only the some-to-some-to-one structure could indicate that it was more difficult to find an optimal summarization in this case.

7.3 Future Work

7.3.1 Dependency Tool

The modified dependency tool supports at this time variable assignments, **if** statements (including **else if**), and **while** statements. In the future, it could be expanded to support e.g. **switch** expressions, **for** loops, **for each** loops, and **do while** loops. Another aspect is that the modified dependency tool works at a function level. It would be beneficial to support analysis across functions and files to experiment with larger VDGs.

7.3.2 VDG Characteristics

As mentioned, some graph structures in the VDG could derive from both explicit and implicit dependencies. Therefore, a potential future work could be to separate between explicit and implicit dependencies to distinguish between the scenarios. One solution could be to label the edges with "explicit" or "implicit". Edge labeling is however not supported by TG-SUM and alteration would be required.

7.3.3 TG-SUM

In this thesis, no alteration has been made to TG-SUM. Potential future work for TG-SUM could be to extend the outputted summarization file or to include an extra file with the supernode IDs and the nodes included in it. This information is today printed to the terminal. Another addition to the tool could be to make a system for labelling the supernodes when there are nodes with different labels in one supernode.

7.3.4 Framework

A potential future direction could be to create a framework for users. In this framework, a user could start by inputting a program code that results in the original VDG. The user could then choose to apply summarization resulting in the output from TG-SUM.

As briefly mentioned, it could also be a good addition to have a summarization technique that retrieves parts of the graph based on user input. Then the user could, after reviewing an overview, choose which part of the graph they wanted to take a closer look at. Such a summarization could be achieved with e.g. VEGAS, a query-driven approach or snowball sampling.

8 Conclusion

The first contribution of this thesis was an extensive search for a technique that can summarize a VDG without losing too much information for a user. A tool to generate VDGs from a program code was needed to conduct this search. In my preceding capstone project [3], a tool named srcSlice was found and modified to generate VDGs with our desired properties. The modification in the capstone project and this thesis made it possible to generate VDGs at a function level for variable assignments, **if** statements and **while** statements.

Criteria were made to help decide whether techniques were fitting. The first criterion was to support all the VDG´s graph characteristics, and the second was visualization without too much information loss for a user. The result of our exploration was the technique TG-SUM. Where TG-SUM supported all the VDGs graph characteristics, had a lossless supergraph output, and several features for enhanced visualization.

The second contribution of this thesis was to help a user interpret the VDG summarization produced by the technique. When analysing code supported by the modified dependency tool, VDG structures corresponding to distinctive code patterns were identified. We then presented the graph structures, code examples resulting in the graph structures, and their summarization. With this knowledge, the user can understand not only what graph structures are behind the supernodes in the summarization, but also which code pattern(s) are behind the graph structure.

Bibliography

[1] UiT - The Arctic University. Cyber Security Group (CSG). Retrieved April 12, 2024 from `https://uit.no/research/csg`.

[2] IT department at UiT The Arctic University of Norway. 2023. ChatUiT (Oct 10 version) [Large language model]. `https://chat.uit.no`

[3] Marie Mikalsen. 2023. Investigating variable dependency graph summarization. UiT - The Arctic University of Norway. Unpublished report.

[4] Paul B. Schneck. 1973. A survey of compiler optimization techniques. In *Proceedings of the ACM annual conference (ACM '73)*. Association for Computing Machinery, New York, NY, USA, 106–113. DOI:`https://doi.org/10.1145/800192.805690`

[5] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3, 319–349. DOI:`https://doi.org/10.1145/24039.24041`

[6] Ken Kennedy and John R. Allen. 2001. Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[7] Norman Wilde. 1990 Understanding program dependencies. Carnegie Mellon University, Software Engineering Institute.

[8] Frances E. Allen. 1970. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*. Association for Computing Machinery, New York, NY, USA, 1–19. DOI:`https://doi.org/10.1145/800028.808479`

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4, 451–490. DOI:`https://doi.org/10.1145/115372.115320`

[10] Ron Cytron, Andy Lowry, and F. Kenneth Zadeck. 1986. Code motion of control structures in high-level languages. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '86)*. Association for Computing Machinery, New York, NY, USA, 70–85. DOI:`https://doi.org/10.1145/512644.512651`

[11] Ron Cytron and Jeanne Ferrante. 1987. What's In a Name? -or- The Value of Renaming for Parallelism Detection and Storage Allocation. *International Conference on Parallel Processing*.

[12] Hakam W. Alomari, Michael L. Collard, Jonathan I. Maletic, Nouh Alhindawi, and Omar Meqdadi. 2014. srcSlice: very efficient and scalable forward static slicing. *Journal of Software: Evolution and Process 26,* 11, 931-961. DOI:`https://doi.org/10.1002/smr.1651`

[13] Christian D. Newman, Tessandra Sage, Michael L. Collard, Hakam W. Alomari, and Jonathan I. Maletic. 2016. SrcSlice: a tool for efficient static forward slicing. In *Proceedings of the 38th International Conference on Software Engineering Companion (2016)*. Association for Computing Machinery, New York, NY, USA, 621–624. DOI:`https://doi.org/10.1145/2889160.2889173`

[14] Thu-Trang Nguyen, Hue Nguyen, Quang-Cuong Bui, Pham N, Hung, Dinh-Hieu Vo and Shigeki Takeuchi. 2020. Practical approach to access the impact of global variables on program parallelism. *2020 International Conference on Advanced Computing and Applications (ACOMP)*, Quy Nhon, Vietnam, 79-86. DOI:`https://doi.org/10.1109/ACOMP50827.2020.00019`

[15] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. 2018. Graph Summarization Methods and Applications: A Survey. *ACM Comput. Surv. 51,* 3, Article 62 (May 2019), 1-34. DOI:`https://doi.org/10.1145/3186727`

[16] Angela Bonifati, Stefania Dumbrava, and Haridimos Kondylakis. 2020. Graph summarization. arXiv:2004.14794. DOI:`https://doi.org/10.48550/arXiv.2004.14794`

[17] Nasrin Shabani, Jia Wu, Amin Beheshti, Quan Z. Sheng, Jin Foo, Venus Haghighi, Ambreen Hanif, and Maryam Shahabikargar. 2023. A Comprehensive Survey on Graph Summarization with Graph Neural Networks. arXiv:2302.06114. DOI:`https://doi.org/10.48550/arXiv.2302.06114`

[18] Zhiqiang Xu, Pengcheng Fang, Changlin Liu, Xusheng Xiao, Yu Wen, and Dan Meng. 2022. DEPCOMM: Graph Summarization on System Audit Logs for Attack Investigation. *2022 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 540-557. DOI:`https://doi.org/10.1109/SP46214.2022.9833632`

[19] Matteo Riondato, David Garcia-Soriano, and Francesco Bonchi. 2014. Graph summarization with quality guarantees. *2014 IEEE International Conference on Data Mining*, Shenzhen, China, 2014, 947-952. DOI:`https://doi.org/10.1007/s10618-016-0468-8`

[20] Dimitris Berberidis, Pierre J. Liang, and Leman Akoglu. 2023. Summarizing Labeled Multi-graphs. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. 53-68. DOI:`https://doi.org/10.1007/978-3-031-26390-3_4`

[21] Cody Dunne and Ben Shneiderman. 2013. Motif simplification: improving network visualization readability with fan, connector, and clique glyphs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. Association for Computing Machinery, New York, NY, USA, 3247–3256. DOI:`https://doi.org/10.1145/2470654.2466444`

[22] Marc A. Smith, Ben Shneiderman, Natasa Milic-Frayling, Eduarda Mendes Rodrigues, Vladimir Barash, Cody Dunne, Tony Capone, Adam Perer, and Eric Gleave. 2009. Analyzing (social media) networks with NodeXL. In *Proceedings of the fourth international conference on Communities and technologies (CT '09)*. Association for Computing Machinery, New York, NY, USA, 255–264. DOI:`https://doi.org/10.1145/1556460.1556497`

[23] Yasir Mehmood, Nicola Barbieri, Francesco Bonchi, and Antti Ukkonen. 2013. CSI: Community-Level Social Influence Analysis. In *Machine Learning and Knowledge Discovery in Databases*. Springer, 48–63. DOI:`https://doi.org/10.1007/978-3-642-40991-2_4`

[24] Michael Mathioudakis, Francesco Bonchi, Carlos Castillo, Aristides Gionis, and Antti Ukkonen. 2011. Sparsification of influence networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '11)*. Association for Computing Machinery, New York, NY, USA, 529–537. DOI:`https://doi.org/10.1145/2020408.2020492`

[25] Lei Shi, Hanghang Tong, Jie Tang, and Chuang Lin. 2015. VEGAS: Visual influence graph summarization on citation networks. *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 12, 3417-3431. DOI:`https://doi.org/10.1109/TKDE.2015.2453957`

[26] François Goasdoué, Paweł Guzewicz, and Ioana Manolescu. 2020. RDF graph summarization for first-sight structure discovery. *The VLDB Journal 29*, 1191–1218. DOI:`https://doi.org/10.1007/s00778-020-00611-y`

A Declaration of the usage of AI tools

In this thesis, AI tools have been used to speed up the process of creating real-world code examples. As mentioned, in Chapter 1, AI was used to generate a program calculating a user's daily calorie need based on personal inputs.

In Chapter 6, AI was used to speed up the generation of real-world examples for program codes 1 and 2. In this process, I first created the graph structure combination desired, before creating the program with variables. This program was then inputted to AI and the AI was prompted to generate a real-world example resulting in the same variable dependencies. The resulting programs from the AI either did not generate the same VDG or made no sense. Therefore, I decided to get inspired by some of its suggestions and made alterations to get the desired graph structures with a real-world example.