Notice to the Reader

This work was originally written in Brazilian Portuguese and converted to English via artificial intelligence, as full manual translation would be very time-consuming. The language instructions will not be translated for now; therefore, any programming language instructions, even if they appear translated in the PDF, will remain in Portuguese in the language.

Universidade do vale do Itajaí
Center for Earth and Marine Technological Sciences
Bachelor’s Degree in Computer Science

Domain-Specific Language for Graph

by

Jailson Nicoletti

Itajaí (SC), julho de 2014

University of Vale do Itajaí Center for Technological Scienc
es of Earth and Sea

Domain-Specific Language For graphs

Graph area

by

Jailson Nicoletti

Report presented to the Examining Board of t
he Technical-Scientific Work of Completion
of the Computer Science Course for Analysis
and Approval. Advisor: Rafael Santiago, M.
SC.

Itajaí (SC), julho de 2014

Thanks

I thank my parents Jose Nicoletti and Nilza Maria Nicoletti and Sister Janara Nicletti, f or all the support offered that allowed me to stay focused only on work during the last semester . I also thank my brother -in -law Ricardo Ghisi for all the tips and lessons that helped me not o nly to perform the work, but also in maturing as a professional.

To Professor Rafael de Santigo, my advisor, for believing in my potential and providin g knowledge, collection and support during the project.

SUMMARY

NICOLETTI, Jailson. Domain-Specific Language for graphs. Itajaí, 2014. 99 f. Technical-Scientific Work for Course Conclusion (Undergraduate in Computer Science)-Center for Technological Sciences of Earth and Sea, University of Vale do Itajaí, Itajaí, 2014.

The present work was designed to reduce the distance between the theoretical bases and practice of graphs. A language restricted to the context of graphs has been created, capable of describing algorithms that solve problems with the use of this data structure. Research on problems involving graphs and their main characteristics were done, followed by a research focused on scientific articles containing use of graph algorithms. These research was done to obtain the necessary knowledge to define what language would look like. A second stage in the research was conducted in the area of compilers and specific domain languages, aiming to identify the tools, methods and better practices in the development of such a language. The implementation was assisted by the Antlr tool, responsible for defining grammar and integration with Java. The instructions tests were done using the Junit tool, which in this work was responsible for comparing the results obtained with those expected in each of the tests scheduled for the instructions. In addition to testing instructions, examples with classic graph algorithms were created to confirm the language's ability to solve graph class problems. With the positive result in the execution of instructions and the correct execution of example algorithms, the tool can be considered a positive alternative for the development of graph -oriented algorithms and also has didactic potential by natively presenting structures and instructions used in books and articles on that theme.

Keywords: graph. Domain Specific Languages. Algorithms.

Abstract

This present work was idealized with the goal of reduce the distances between theoretical and practical bases of graphs. A language restricted to the graphs context was created, capable of describing algorithms that solve problems using this data structure. A research referring to graphs problems and their main characteristics, followed by a research focused on scientific articles containing usage of graph algorithms. These researches were done with the goal of obtaining the necessary knowledge to define how these language would be. A second phase of the research was conducted in the field of compilers and domain-specific languages, looking for identifying the tools, methods e best practices in these kind of language. The development had the help of the ANTLR tool, responsible for the definition of grammar and Java integration. The instructions test were done using JUnit tool, in these work it had the responsibility of comparing the obtained results to the expected for each one programmed to the instructions. Besides the instructions tests, it was created examples with classical algorithms of graphs to confirm the capacity of the language to solve graphs problems. With the positive result from the execution of the instructions and the correct execution of the example algorithms, this tool may be considered a positive alternative to graph algorithm development and also has a didactic potential for natively presenting instructions and structures commonly used in books and articles of these matter.

Keywords*: Graph. Domain Specific Language. Algorithms.*

List of Figures

List of staff

List of Equations

List of abbreviations and acronyms

Antlr	Another Tool for Language Recognition
DSL	Domain-Specific Language
GPL	General Purpose Language
USA	United States of America
IDE	Integrated Development Envioment
SGBD	Database Manager System.
SQL	STRUTURED QUREY LANGUAGE
TTC	Technical-Scientific Work for Course Conclusion
Univali	University of Vale do Itajaí

SUMMARY

1 Introduction

Computing provides a number of troubleshooting tools. For this it is necessary to have ways to represent existing structures in the real world. As an example, we can cite connections between locations on a map or a circle of friendships. These are among others, structures that c an be representation through graphs (Boaventura Netto, 2006).

Graph theory came in 1736, when Leonhard Paul Euler, the eighteenth -century physici st and mathematician, studied the problem of Königsberg bridges. The name of the problem co mes from the name of the city that was Prussia territory until 1945. Königsberg is divided by a river, with two islands that interconnected with the margins through seven bridges. From there came the following question, "Is it possible to cross all bridges without having to go through it more than once?" Using a representation of Grafo Euler was able to prove that it would be im possible to perform the feat. After that there was little development about graph theory. Only i n 1936 that Denes Konig wrote the first book on the subject. (HARJU, 2011, p.2)

Computer science addresses graphics for computing theory topics and as models for pr oblem solving. However, generic programming languages are not oriented to the construction of programs that solve problems through graphs. Kreher *et al* (1999 p. 106,107,108,112), desc ribe some synthetic graph problem algorithms compared to solutions using general purpose pro gramming languages. The authors use union and intersection operations between vertices and e dges to solve problems.

To facilitate the creation of graphs that work with graphs, language is needed that make s the description of your problems simpler. As with Fido language, a DSL (Domain-Specific L anguage) that was designed to concise regular sets of strings and trees (Klarlund, Nils, 1997).

DSL are languages designed to deal with a specific problem of problem. Some aspects are important for the design of a DSL (Fowler, 2010):

- Computer Programming Language: It needs to be easy to understand and executable by the computer.

•NATURAL LANGUAGE: It must have a sense of fluency not only in individual e xpressions but also in whole when attached. • Limited expressiveness: Language m ust only meet the data type support needs, control and structures needed for its dom ain. • Focus on the domain: language needs to focus only on its domain.

As language focused on a domain, a DSL provides necessary abstraction for problem s olving in the application context. In literature, some advantages are cited:

•Enhanced Production: They are more concise and facilitate the interpretation of th e code by the programmer or expert in the area, due to their syntax defined similarly to the terms used by the domain of the problem; • Facilitated Verification: A DSL semantics can control some vital properties for system operation, thus managing to control items such as Deadlock protection. (Gulwani *et al.*, 2001); • increase produc tivity, trust, maintenance and portability. (Deursen and Klint, 1998);

However, the use of DSL also brings some disadvantages such as the cost of projecting to deploy, maintaining language, performing users, difficulty finding a good scope and potenti al loss of efficiency. (Krueger, 1992).

Despite the disadvantages cited, the development and application of this concept made it possible to create several DSLs. Among them are: SQL, HTML and CSS.

An example of DSL related to the context of informatics is webcal language, which aim s to facilitate the resolution of some websites generated by websites, offering greater flexibility in the configuration of cache policies, and can easily generate new local cache policies or inter coche protocols (Gulwani *et al.*, 2001).

In the context of sports, the Easytime language aims to improve the configuration of ti me measurement systems through a flexible timing system and the reduction of the number of equipment required for measurement. In the tests carried out by the Easytime development tea m it was useful for competition organizers and sports clubs helping the results of smaller comp etitions, and simplifying the setting of the timing system in major competitions. (Fister Junior *et al,* 2011)

Due to the form that are designed, DSLs are tools that can simplify the work of experts and programmers of the systems that work in their domain area, In addition to being able to ensure that important controls for the correct operation of the syste m are served (Fowler, 2010). Thus assisting the development of its field of dominance.

From the study of graphs, the knowledge that will be acquired about creating languages and good manners for the creation of DSL, a new language will be designed that allows people with knowledge in graphs to perform algorithms that work aspects of graph theory.

1.1 Problematization

1.1.1 Problem Formulation

There are several areas that solve their problems through graphs. The field of molecula r biology, informatics, operational research and chemistry, with the study of large networks, ar e examples of studies directed with the aid of graph theory (Boaventura Netto, 2006, p.279). T herefore, the use of general purpose programming languages can make it difficult to resolve th ese problems as they are commonly used by professionals from computing areas.

This work proposes the development of a DSL, which aims to simplify the coding of gr aph algorithms using a context restricted syntax. In this way language would allow anyone wit h knowledge in graph theory to interpret or produce DSL codes, regardless of the professional area.

How to generate a DSL to simplify the development of algorithms involving graphs?

1.1.2 Proposed solution

This work proposes the development of a DSL directed to the production of algorithms that solve graph problems. For testing, algorithms will be developed with the syntax defined in the created language. The scripts created will be executed in an analyzer that will be developed to perform the Created DSL algorithms.

1.2 Objectives

1.2.1 General Objective

The purpose of this work is to develop a DSL for coding algorithms that use graphs.

1.2.2 Specific Objectives

The specific objectives of this work are:

•Define the standard way of working with graphs; • Run DSL Study already existing i n the context of graphs; • Identify how language will be structured; • Develop DSL to work with graphs; • Analyze at least three algorithms in the context of graphs through an interpreter generated for DSL; • Prepare document containing comparative test desc ription between DSL and general purpose languages, performed with people who know problems solved with graphs.

1.3 Methodology

This work was divided into three steps: theoretical foundation, development and docum entation.

For the theoretical foundation studies on graph theory to know the theme of the proposed langu age, the structure of a programming language through compiler materials and in parallel langua ge research were evaluated specific materials of DSLs to better understand the differentials of a planned language for general use and language with specific domain, which will be applied t o work. The final part of the theoretical foundation sought content regarding the development of algorithms that involve graph theory to find patterns in development.

The development stage had the development of the IDE that implements the language interpret er. The concepts learned in the foundation were applied

Theoretical, compiler and DSL content were used as a guide for language implementation, whi le the graph part served as the basis for syntax definitions.

In the documentation stage, the course completion work that presented the content obtained in the theoretical foundation phase and described the language and its implementation process wa s written.

1.4 Work Structure

This document is structured in four chapters. Chapter 1, introduction, presents an over view of the work. In chapter 2, theoretical foundation, a bibliographic review is presented abou t: graphs, as well as an analysis of DSLs. Chapter 3 presents language and how it was implem ented, as well as describing the tests performed. In chapter 4 are presented the conclusions of t he work, where the methods used in the development, the difficulties and solutions found durin g the work are cited. Finally the appendages present items created throughout the work that ser ved as support in language development and testing.

2 Theoretical Rationale

This section aims to substantiate the decisions that will be made in the project phase an d present the topics covered. It is divided into: graphs, programming languages, Domain-Speci fic Language, graph problems algorithms.

The graph section will present the definition and characteristics of a graph besides expl aining some problems that will later be used in the analysis of algorithms, a section that aims t o identify syntax patterns in the description of algorithms for graph problems.

The programming languages will describe the developmental language structure and th e Domain-Specific Language section will explain the meaning of the DSL term and cite import ant characteristics of its structure and development.

2.1 Graphs

From the creation of the theory of graphs in the eighteenth century it was much produce d using the concepts addressed by it. Several areas of knowledge such as organic chemistry, bi ology and electronics had studies that used their techniques. However, initially there was little interest in this study. After Euler's creation in the eighteenth century, the first book dedicated t o this theory was released in 1936 and most of scientific production onwards appeared from 19 70 and at this time its applications were driven by computer use. (NETTO, 2006).

A graph is a non -empty finite set of vertices and their possible connections. It can be f ormally represented as a double g = (v, a). Where V is a non -empty set of vertices and A is the set of edges (bonds between the vertices). Each edge is represented by a set of pairs not sorted of V. elements (SZWARCFITER, 1988).

Within the graph, vertices can be associated with real -world objects, such as a city, per son, part of an electronic circuit, among others. When graphically represented the vertices are usually points or circles. They have a degree that is defined by their number of neighbors ie th e number of vertices that are connected to it. (DIESTEL, 2005).

The links between the vertices of the graph are the edges, represented by lines, curves o r straight, connected to the vertices, thus representing their connection. The calls can represent several things like the path between two cities, the friendship between two people, a connectio n between two components of a circuit between various other possibilities. Depending on what edges represent may have values to represent data such as the distance between two locations. (HARTSFIELD, 1990). Figure 1 represents a graph with six vertices represented by circles an d seven edges represented by lines.

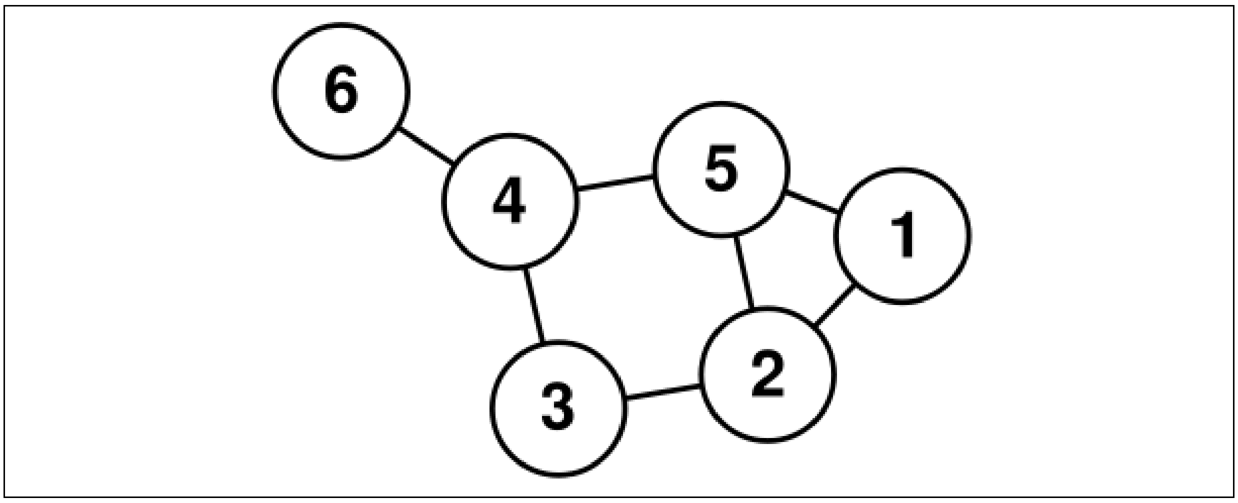


Figure 1. Example of graph. Source
: Danielamaral.wikidot (2012).

When two vertices of a graph are connected by an edge, they are defined as neighborin g or adjacent vertices. This connection between a pair of vertices can be made by more edges, i e the vertices A and B are connected by the vertices f and g at the same time, in this relations hip f and g would be called parallel edges. This variable number of edges and vertices in a gr aph define respectively the size of the graph and its order. (BALAKRISHNAN, 2005).

The following will be presented characteristics of graph classification that will be used in the description of graph problems and the algorithms that work these problems.

2.1.1 Graph Classification

Some terms were created to determine characteristics of a graph. This section will pres ent and explain the necessary terms to understand the problems that are presented.

Ties are connections between the same vertex, as shown in Figure 2, where vertex 1 has an edge that comes and goes straight to it. Arch is the denomination given to an edge within a directed graph called the digit, the differential in this case is that connections in the calls is giv en only in one direction, ie, vertex 1 may be connected to 2 and no direct connection between v ertex 2 and 1. Following the concept of a one -way, where cars can follow only in the determin ed direction. As illustrated in Figure 3 (Netto, 2006). In this work the ties and arches will be c alled in their widespread form as edges.

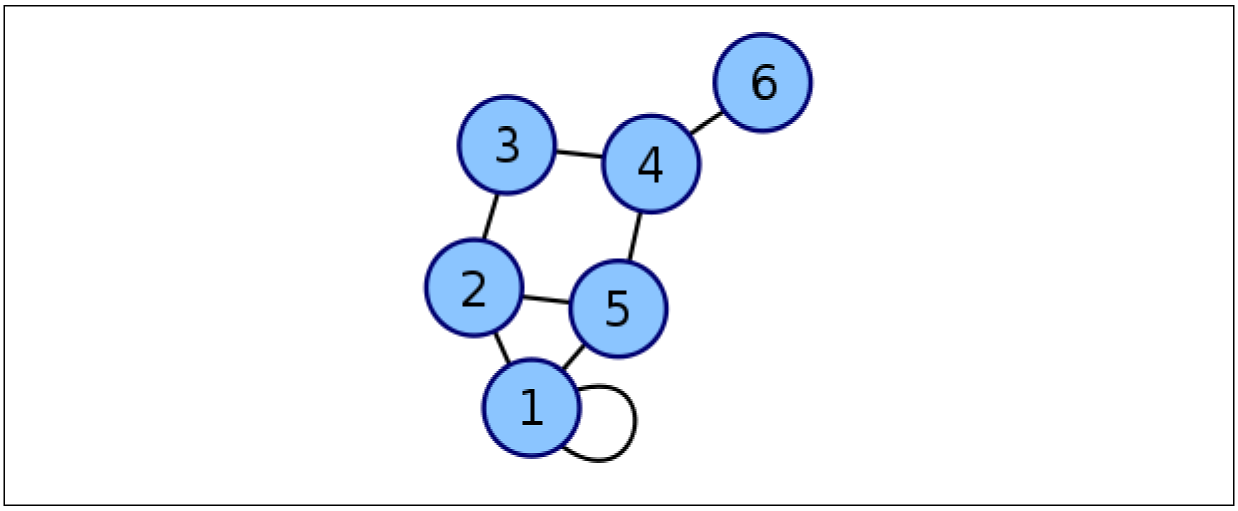


Figure 2. GRAFO WITH LAS
T SOURCE: Wikipedia, Laço (
2012)

In digraphs the degree of entry is accounted for from the number of arches that are dire cted to it while the degree of exit is the amount of arches that are directed to another vertex fro m it. (SZWARCFITER, 1988).

DIRECTED GRAPHERS Also called digraphs have connection degrees, where strongl y connected indicates a structure in which all connected edges have round trip paths. (unilater ally or semi-strongly connected) Connected when everywhere there is at least one connection b etween them, even if it is unilateral.

Weakly connected or disconnected when there is at least one pair of vertices is not connected r est of the graph. (SZWARCFITER, 1988), (NETTO, 2006).

The attainable or achievable term is used in directed graphs to demonstrate the possibili ty that vertex A has to reach vertex B, and there is no need for direct connection, ie it is attaina ble if there is any path between two vertices. If any vertex can reach all the other vertices of a given graph then it is called the graph root. (SZWARCFITER, 1988). In non -oriented graphs all related vertices are also attainable. (Netto, 2006)

One walk between two vertices of a graph is the passage through the vertices and edges that separate them and the number of edges defines the size of the ride. This process lists all v ertices and edges elapsed, except for the simple graphs, where it is not necessary to list the edg es due to the existence of a maximum of one connection between each pair of vertices. The tou r is called a trail if no edge is repeated and is called the way when no vertex appears more than once on the list. A walk with the equal start and destination vertex is called the closed path an d if it has no repeated edges is called as a circuit. A cycle is a circuit that has no repeated verti ces. (BALAKRISHNAN, 2005)

A connected graph is so called when there is a walk among all its vertices, that is, from any of the vertices it is possible to reach all others (DIESEL, 2005). Figure 3 below has a conn ected graph.

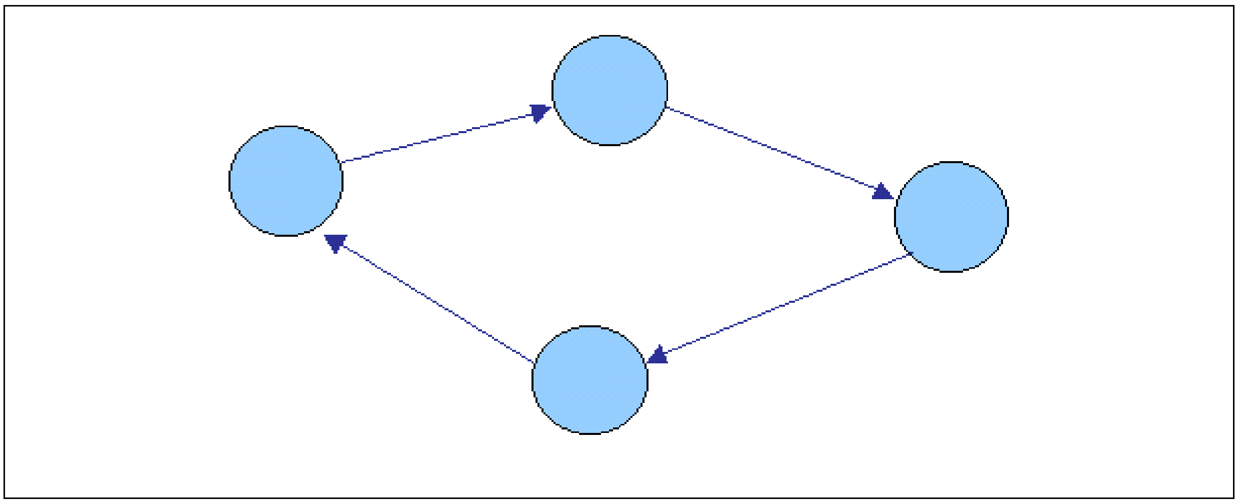


Figure 3. Connected graph
Source: EDUC.FC.UL. (2
012)

Graphs are often used to represent real -world structures. Creation of maps and circuits are are as that use a particular type of graph, the flat graphs. This type of graph has the characteristic o f being designed on a plan so that there are no cross -crossed edges. (Boaventura, 2006) Figure 4 below has a planar graph in its normal form and soon followed in its topological form.

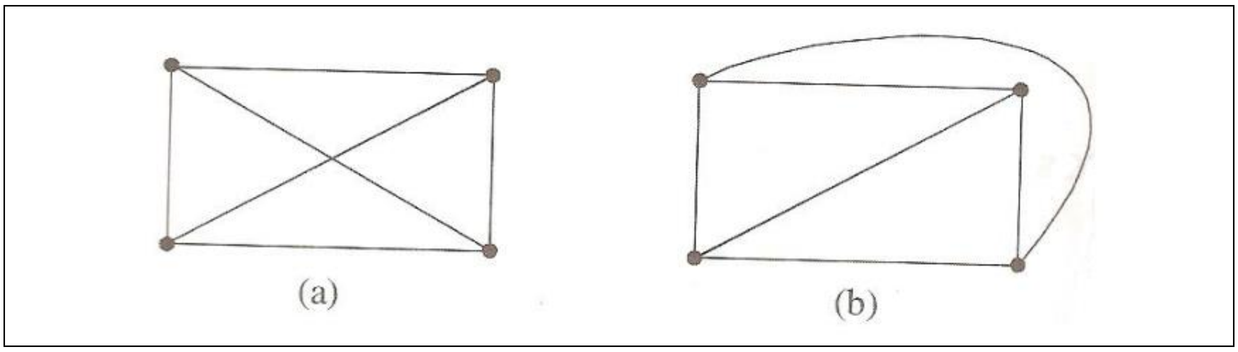


Figure 4. Planar graph and its topological form (B) S ource: Boaventura (2006)

Isomorphism is a structural equivalence in graphs, that is, isomorphism is the bijection betwee n two graph vertices so that two vertices are adjacent in a graph only if adjacent also in the sec ond graph. Thus if the name of the vertices of one graph is changed according to the names of t he other graph the two would be the same (IME.USP, 2012).

An contraction operation in a graph is a summary where two or more vertices merge a nd inherit the calls each vertex had with the rest of the graph. From this operation it is possible to find the relationship *minor* of a graph. If an X graph is summed up by the contraction operat ion and the resulting graph is a subsuction of y then it is said that graph x is *minor* of Y. (Die sel, 2005). Figure 5 below presents a series of contractions performed on a graph, for a better u nderstanding the first contraction A happens in the vertices D and K that become DK.

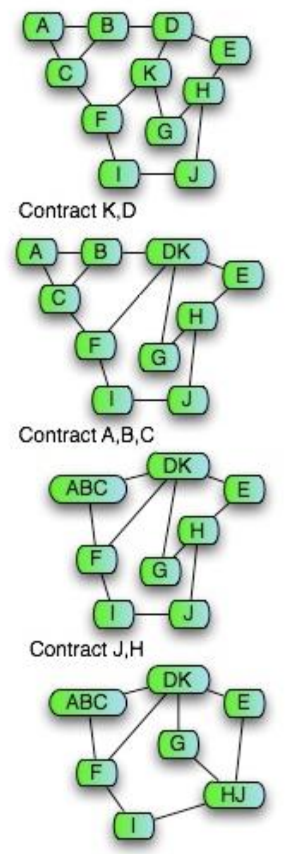


Figure 5. Contract operations in a graph Source
: ScienceBlogs (2012)

A graph G is called R-Partitis, where r is a larger number than one, if its vertices allow a partition in R classes so that each edge has its ends in different classes. In cases where the gr aph would be 2-partitis it usually assumes the name bipartite. (DIESTEL, 2005). Figure 6 belo w illustrates a 3-partitis graph.

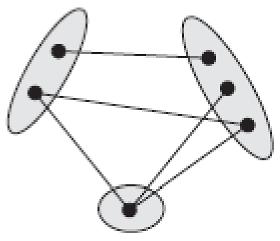


Figure 6. 3-Partitis graph
Source: (DIESEL, 2005)

2.1.2 Graph problems

Among the existing graph problems, search issues gain special importance due to the la rge number of solutions solved through their algorithms (many of them solved in exponential ti me). A major problem in the search within the structure of a graph is the lack of location orien tation within the graph, that is, it is necessary to verify the vertex visited because there is no cl ear sequential vertex structure for vertex. (SZWARCFITER, 1988).

This section will define three problems surrounding graphs: minimum path, color and maximum click. Subsequently, implementations made by different authors will be presented, a iming to identify language patterns in creating graphs.

2.1.2.1 Minimum path (resolution with dijkstra algorithm)

The problem of the minimum path seeks the shortest distance between one vertex and a nother, taking into account the sum of the value of the weights of the edges traveled. This subj ect is of great importance due to the constant need for human beings to identify the lowest cost for any purpose, either for an individual's convenience or for a large company to reduce costs. (Shezad, 2009).

Dijkstra's algorithm solves this problem over time (n^2) in graphs that have edges greater than or equal to zero. He uses a search for the nearest neighbor, from an initial vertex, with e ach iteration he finds the adjacent vertex with lower cost, accumulates this value and closes the vertex. This sequence is repeated until the fate vertex is found, when this happens the algorith m returns to closed vertices and seeks new surroundings to perform the same process by accum ulating the cost of the new path. In the end the path with the lowest accumulated cost is chose n by the algorithm (Netto, 2006). Figure 7 presents an illustration of the sequence of steps of t he dijkstra algorithm performed in a graph.

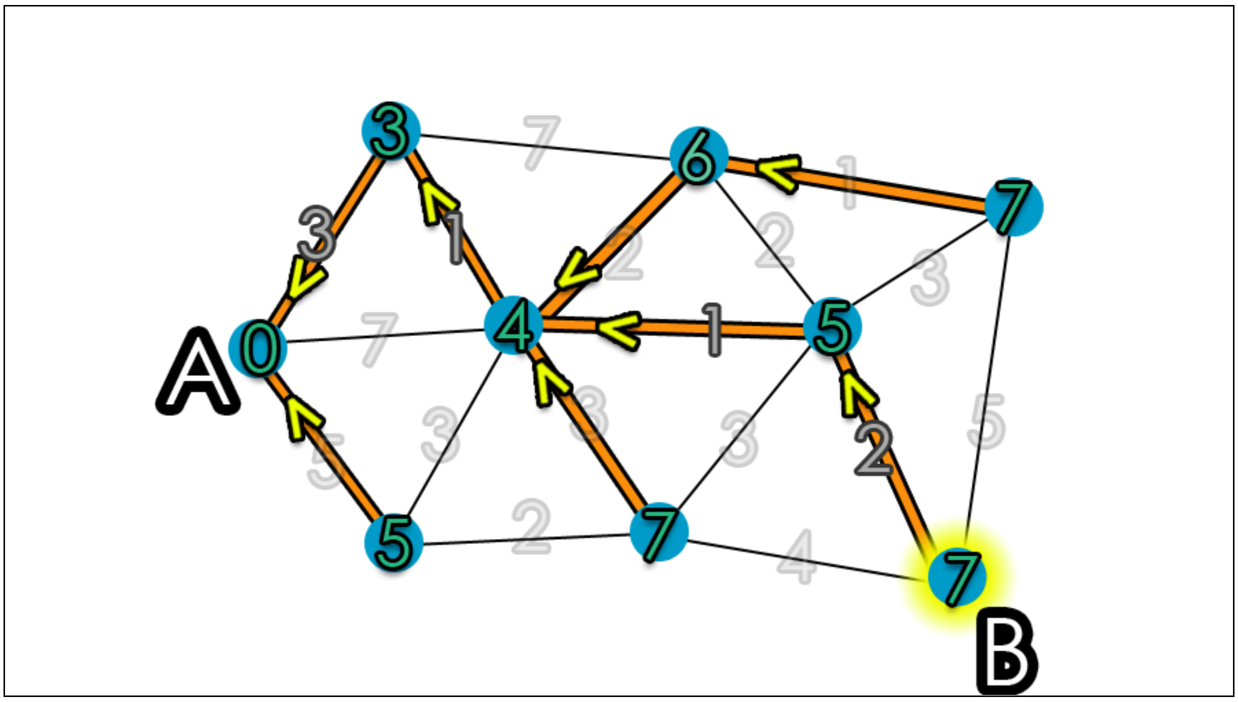


Figure 7. Execution Dijkstra Algorithm S
ource: VASIR (2012)

2.1.2.2 Graph color

The coloring of vertices is a process that colors the graph so that adjacent vertices do n
ot have the same color. The minimum number of colors required to color the graph is defined by variable k. This minimum value is called chromatic number and the corresponding graph wi
ll be called k-chromatic. This way a graph that needs 3 colors to be colorful will be called 3-c
hromatic. (DIESTEL, 2005)

Graph color is usually used in situations that use combinatorial analysis, where differen
t information generates sets that need to be united to solve a problem, a known example is the problem of exams. It consists of seeking the fewer days needed to perform an exam where ca
ndidates need to take certain tests (Boaventura, 2006).

2.1.2.3 Click Maximum

In graph theory the click is a subset of vertices connected so that there is an edge betwe
en each pair of vertices. Its size is defined by the cardinality of its vertices set. (Imme.usp, 201
2) The click with the highest cardinality (the largest amount of elements of the set) of the grap
h is called the maximum click, and

There may be several in a graph. (ROSEN, 1999). Figure 8 presents an example of maximum c lick between the vertices 3,4,5,6 that are mutually connected.

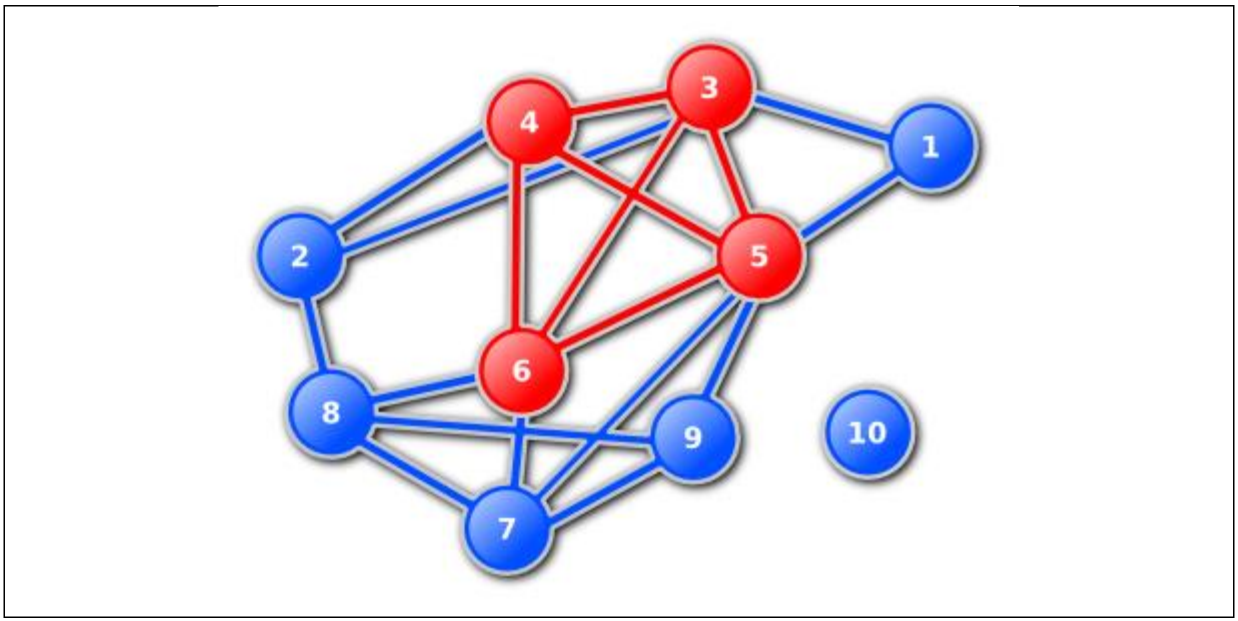


Figure 8: Maximum click. Vertices 3,4,5,6.

Source: Reactive-Search (2012)

2.2 Programming languages

Programming languages are sets of instruction defined in a grammar that serve to execu te the commands of an algorithm and thus give orders to the computer. (Said, 2007).

In the design of a new language it is necessary to define its grammar and semantic. Gra mmar or syntax defines the terms that language will book and the rules the programmer must f ollow to generate the code. Through the rules of grammar is generated the syntactic tree also c alled *parse tree*. Grammatical rules are created so that each term has a predefined form of use and a certain behavior. The behavior generated by the term of language syntax is determined s emantic. (Fowler, 2010).

In this set of structures that form a language grammar is the part where the programmer expresses the logic of its application. It consists of a set of terms reserved for language and wh ich are used under some rules defined for the formation of commands. These terms of terms ca n form more complex productions through associations between them. The union between cha ins can be done in many ways, As long as they are following the rules defined, they can be used in different ways and consequently produce programs for various purposes. (Jandl Jr, 2001)

From the recognition of the chains and the terms contained in each one, the interpretation of the Code begins. This phase is called lexical analysis and is responsible for the first phase of the process that will generate the actions defined in the source code. At this stage the lexicon analyzer receives the text containing the written code and identifies its symbols defined by grammar. (SEBESTA, 2002)

With lexical analysis completed the identified symbols need to be analyzed to identify if they are in the structure defined by grammar. This step is called syntactic analysis and inserts the terms found within a data structure, usually called the syntactic tree. With the terms arranged in the tree the goal is to follow the grammar's predicts can reach an initial symbol, which validates the input code within the grammar. (LOUDEN, 2004)

Through syntactic analysis *parser* can ensure that all grammar rules are met, but languages often need to control other information such as data type, variable declaration and other types of information depending on language. This task is fulfilled by the semantic analysis that acts to verify the relationship between the parts of the code, generating an interpretation of the input content and providing internal language components the data necessary for the execution of the program. (RICARTE, 2008)

The data obtained in the semantic analysis have the information needed to perform the behavior described by DSL, but some details can be added to ensure the quality of DSL. As in most projected programs, the existence of software layers with well -defined roles facilitate the process of software development and maintenance (MARTINS, 2007).

2.3 Domain-Specific Language

DSL (Domain-Specific Language) is a term created to define a language that is designe d to meet a specific area. This specificity applied to the syntax aims to implement flexibility s oftware that make the development process more agile. They are usually small, declarative an d offers restricted syntax, as needed by the domain worked. Clarity in syntax enables the end u ser himself to write some tasks, as with macros in spreadsheet software. (DEURSEN, 1998)

As in common projects, made in GPLs (General Purpose Language), there must be an a bstraction that shapes the mastery of the problem and facilitates the programming of program r ules. Libraries or *frameworks* are usually used that abstract part of the logic of the main progra m. (Fowler, 2010).

DSL is implemented above this model and provides a new level of abstraction for the p rogrammer. It can make the model simpler for trained people to solve specific problems their u nderstanding and enable the setting of the environment at run time. The model provides the be havior of the application that has the data from the DSL code previously interpreted by Parser. DSL is a model adjunct that provides some uses for the programmer. (Fowler, 2010).

Although in GPL projects it is natural to create the model before the program, in DSLs production it is common for language design to be done before the model. This happens as the y act in different layers interconnected at execution time by Parser. (Fowler, 2010).

It is important to know the benefits of using DSLs to be able to identify when their use is viable. These benefits are:

- Development Productivity: Syntax clarity makes it easier to identify errors and fac ilitate maintenance, thus avoiding errors in the final product and reducing time spe nt on code maintenance. Moreover, it also provides better abstraction for the prog rammer not to lose the focus of the problem to be solved;

•Communication with experts in the area: In the DSL project a syntax can be define
d that complies with this. In this way the end users of the system can read and inter
pret the code, facilitating the detection of logic errors in some cases allowing the co
dification by themselves; • Change in Execution Context: Fits Systems Configurati
on Files with Setup, Changing the File The system would have a different behavior.
As for example, change between production base and testing, to exchange SQL que
ries at execution time, the possibility of the same code being translated into differen
t locations, such as a validation of data that runs both vision and server even if it is i
n different languages, also offers the possibility for users to provide new rules at ti
me to more comprehensible to system users; • Alternative Computational Model:
Often the imperative programming model does not apply so well to all problems. I
n cases where a declarative programming model becomes more interesting that DS
Ls can help by adapting to these computing models. (Fowler, 2010).

To better define the characteristics of a DSL three types are defined:

•Internal DSL - It is a way to use parts of a generic language to generate a specializ
ation capable of abstracting a problem this type of DSL uses the general use langu
age interpreter to be processed;

•External DSL - uses its own syntax, which is interpreted by a parser implemented i
n a generic language;

•LANGUAGE WORKBENCH - DSL performed on its own tool created for it, allo
ws for an improvement in the use against external DSL because external DSL can
only change the text while in this style can be added specific tools for that type of l
anguage.

The development of a semantic model offers several advantages in the design of a DSL. Some examples are:

- allows you to test the behavior and interpretation of DSL separately;
- Facilitate verification of interpretation of the code by observing the data inserted i n the semantic model from DSL;
- Facilitates the evolution of language without the risk of altering the behaviors alre ady defined;
- In cases where there are several parsers the integrity test between them is facilitate d by checking the data in the semantic model;
- enables the test behavior of language separated through manual data insertion;
- Separates the thinking from the layers from the behavior layers (Fowler, 2010).

The semantic model offers a mold for DSL information, taking the necessary way to ab stract the mastery of the problem in the language of general use. From it can be defined the act ions and responses of DSL. Although this model is very similar to the object model, the term is different, as in some cases the model may not be a domain of the central problem domain in a s ystem (Fowler, 2010).

2.3.1 DSL Example

To identify necessary basic structures in a language that addresses graphs, programming languages was researched that already had this goal. A search was done on the *http://scholar.google.com.br/* website by articles and books that described a DSL with a scope in the execution of graph algorithms, but until the date of completion of this work was found no language for this purpose.

To understand how to specify a DSL, a selected example is described due to the followi ng characteristics: use of sets, specific scope. In this search was found an article by Klarlund p resenting the Fido language. The language data raised by him are presented below. (Klarlund, 1997).

In this research was found the Monadic Second-Norder Logic (M2L), which through th e regularities identified in the domain of the problem can reduce challenging problems to regul ar string or trees language issues. The disadvantage of this logic is that it has a non -elementar y lower limit in its decision procedures. (Klarlund, 1997).

The implementation of this logic is called Mona and can deal with non -trivial formulas , having up to five hundred thousand characters. The problem is that the description of these s ets becomes very complicated and requires M2L programmers to spend most of their time by r eviewing complex codes. (Klarlund, 1997).

Fido has been implemented to provide an optimized M2L formula compiler and has alr eady been used in several real applications. In equation 1 is presented a formula in the M2L sta ndard that is presented in language in equation 2 within the Fido syntax (Klarlund, 1997).

Equation 1. M2L Standard Formula
Source: Klarlund (1997)

$$\psi \equiv \forall 1\, p : (p \notin X0 \wedge p \notin X1) \Longrightarrow (\exists 1 q : p < q \wedge (q \epsilon X0 \wedge q \notin X1))$$

Equation 2. Formula M2L in the Fido Standard

Source: Klarlund (1997)

$$\begin{aligned}&type\ T = a, b(next{:}\,T) \mid c;\\ &string\ x; T;\\ &\forall\, pos\ p{:}\,x.(\,p = a \Longrightarrow \exists pos\ q{:}\,x.(p < q \wedge q = c))\end{aligned}$$

2.4 Algorithms for graph problems

This section will present nine algorithms found in articles by different authors to solve t he three problems of graph theory: minimal path (using dijkstra), coloring and maximum click. In the end will be commented on the characteristics identified in the algorithms, as to their synt axes.

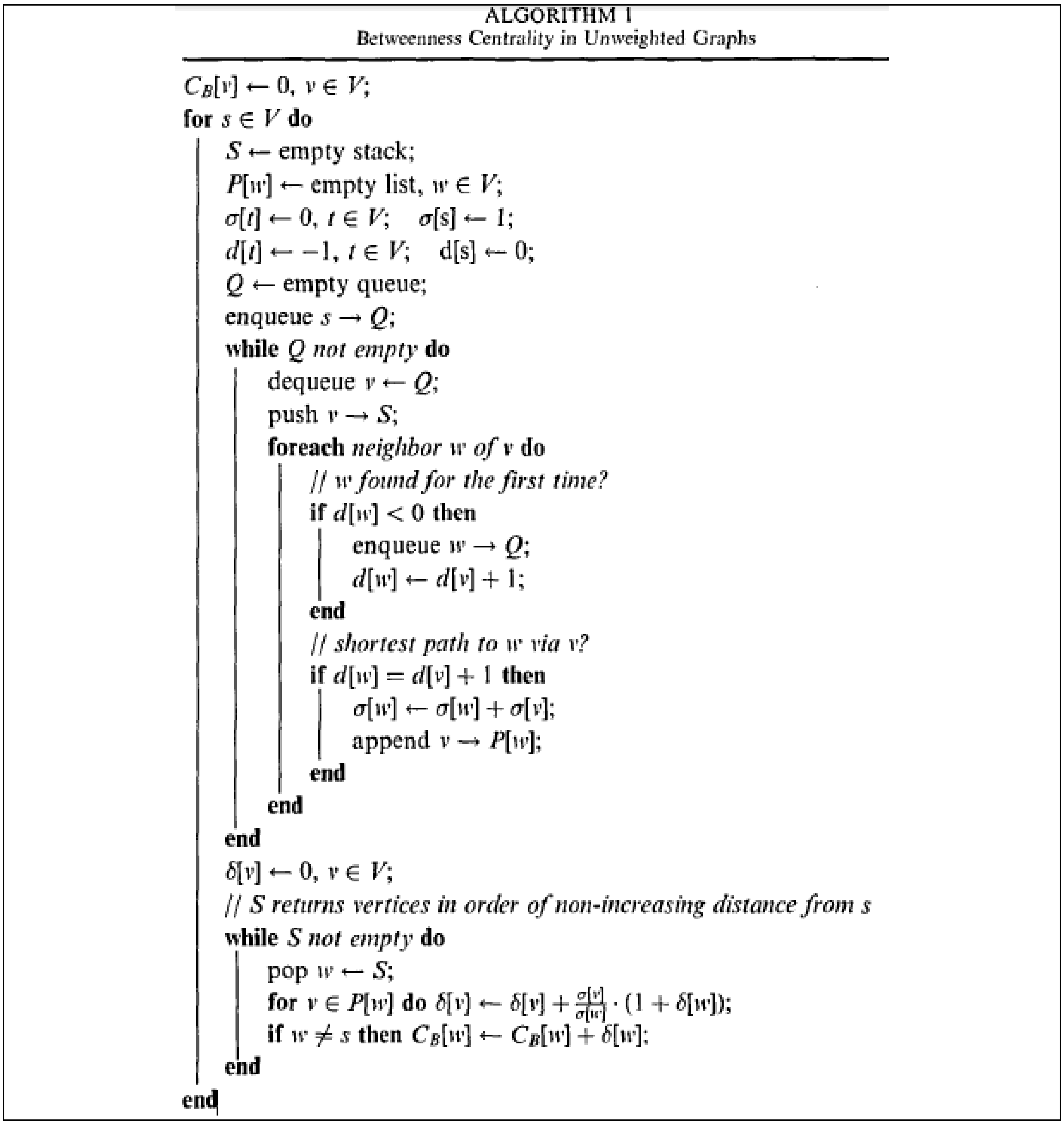
2.4.1 Dijkstra

This section presents published articles that report applications of the Dijkstra algorith m, in all will be presented three articles from the following authors: Brandes (2000), Shehzad (2009) and SHI (1998).

2.4.1.1 A FASter Algorithm for Betweenness Centrality In Unweight Graphs

Brandes wrote an article that addresses the centrality index in the vertex of a graph. The goal is to classify a vertex according to the position on the graph, a subject that has been gainin g more importance with the increase of complex networks mainly within social networks. In U lrik's work, the dijkstra algorithm is used to verify the number of smaller paths between each p air of weight of the graph with weight (Brandes, 2000). The algorithm is presented in Figure 9 .

The algorithm has some characteristics such as the specific bond that runs through the e ntire set of vertices. In addition, you can see the use of three data structures to meet your goals: pile, queue and list.



ALGORITHM 1
Betweenness Centrality in Unweighted Graphs

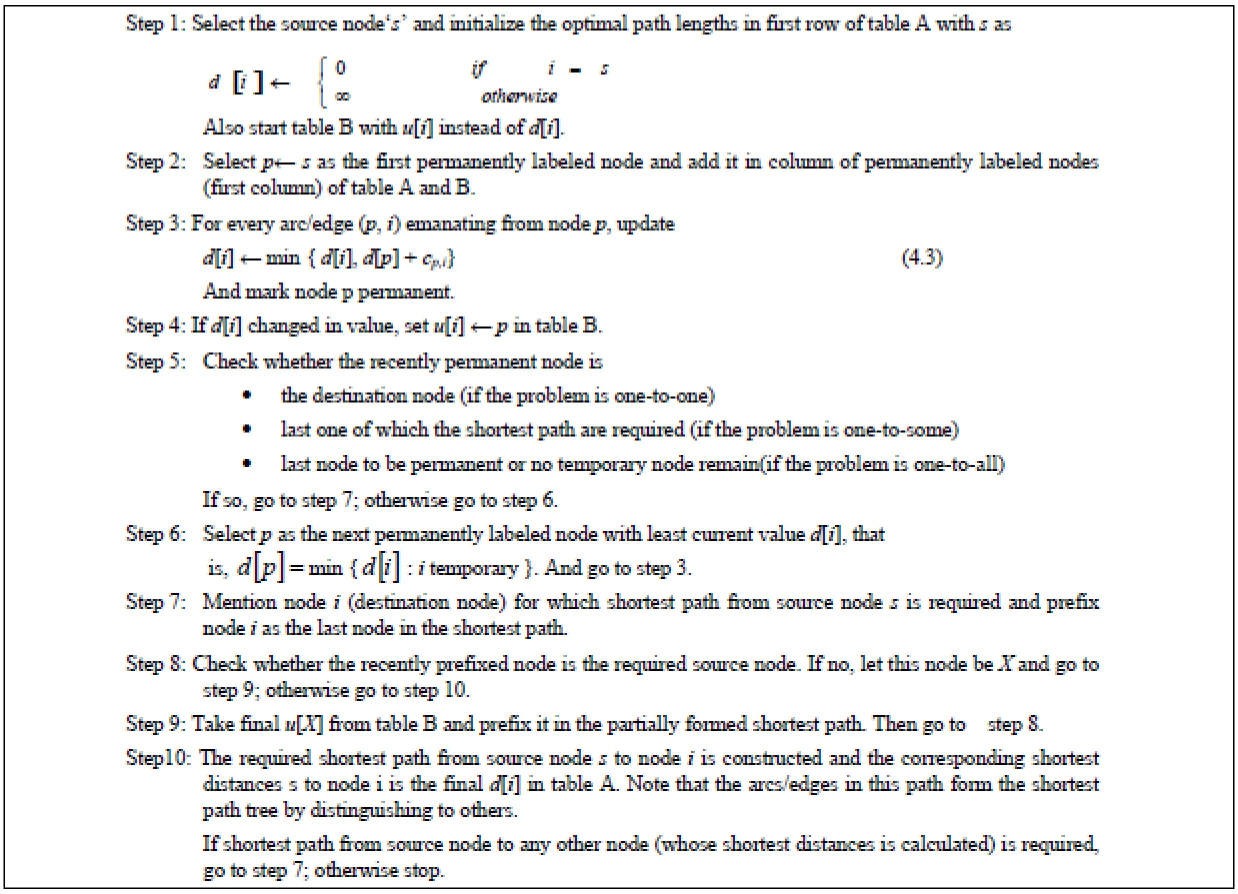
```
$C_B[v] \leftarrow 0,\ v \in V$;
for $s \in V$ do
    $S \leftarrow$ empty stack;
    $P[w] \leftarrow$ empty list, $w \in V$;
    $\sigma[t] \leftarrow 0,\ t \in V$;    $\sigma[s] \leftarrow 1$;
    $d[t] \leftarrow -1,\ t \in V$;    $d[s] \leftarrow 0$;
    $Q \leftarrow$ empty queue;
    enqueue $s \rightarrow Q$;
    while $Q$ not empty do
        dequeue $v \leftarrow Q$;
        push $v \rightarrow S$;
        foreach neighbor $w$ of $v$ do
            // $w$ found for the first time?
            if $d[w] < 0$ then
                enqueue $w \rightarrow Q$;
                $d[w] \leftarrow d[v] + 1$;
            end
            // shortest path to $w$ via $v$?
            if $d[w] = d[v] + 1$ then
                $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$;
                append $v \rightarrow P[w]$;
            end
        end
    end
    $\delta[v] \leftarrow 0,\ v \in V$;
    // $S$ returns vertices in order of non-increasing distance from $s$
    while $S$ not empty do
        pop $w \leftarrow S$;
        for $v \in P[w]$ do $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$;
        if $w \neq s$ then $C_B[w] \leftarrow C_B[w] + \delta[w]$;
    end
end
```

Figure 9. Dijkstra Algorithm 1.
Source: Brandes (2000).

2.4.1.2 Evaluation of Shortest Paths In Road

In 2009 Farrukh Shehzad wrote the article Evaluation of Shortest Paths in Road Networ k. The study is based on road distance optimization by choosing the best route and for that the Dijkstra and Floyd-Warshall algorithms are applied. A modified version of the dijkstra algorit hm was applied in this study by generating as products the table containing the lowest distance from the root vertex to all others and table B helps in recovering the shortest corresponding pat hs (SHEHZAD, 2009). The algorithm is shown in Figure 10 below.

The author makes use of a textual description, and uses repetition ties through the got f unction. In addition, tables were used to store values regarding the execution of the algorithm.



```
Step 1: Select the source node ‘$s$’ and initialize the optimal path lengths in first row of table A with $s$ as
        $d\,[i\,] \leftarrow \begin{cases} 0 & if \quad i = s \\ \infty & otherwise \end{cases}$
        Also start table B with $u[i]$ instead of $d[i]$.
Step 2: Select $p \leftarrow s$ as the first permanently labeled node and add it in column of permanently labeled nodes
        (first column) of table A and B.
Step 3: For every arc/edge $(p, i)$ emanating from node $p$, update
        $d[i] \leftarrow \min \{ d[i], d[p] + c_{p,i} \}$                (4.3)
        And mark node p permanent.
Step 4: If $d[i]$ changed in value, set $u[i] \leftarrow p$ in table B.
Step 5: Check whether the recently permanent node is
            • the destination node (if the problem is one-to-one)
            • last one of which the shortest path are required (if the problem is one-to-some)
            • last node to be permanent or no temporary node remain(if the problem is one-to-all)
        If so, go to step 7; otherwise go to step 6.
Step 6: Select $p$ as the next permanently labeled node with least current value $d[i]$, that
        is, $d[p] = \min \{ d[i] : i \text{ temporary} \}$. And go to step 3.
Step 7: Mention node $i$ (destination node) for which shortest path from source node $s$ is required and prefix
        node $i$ as the last node in the shortest path.
Step 8: Check whether the recently prefixed node is the required source node. If no, let this node be $X$ and go to
        step 9; otherwise go to step 10.
Step 9: Take final $u[X]$ from table B and prefix it in the partially formed shortest path. Then go to   step 8.
Step10: The required shortest path from source node $s$ to node $i$ is constructed and the corresponding shortest
        distances s to node i is the final $d[i]$ in table A. Note that the arcs/edges in this path form the shortest
        path tree by distinguishing to others.
        If shortest path from source node to any other node (whose shortest distances is calculated) is required,
        go to step 7; otherwise stop.
```

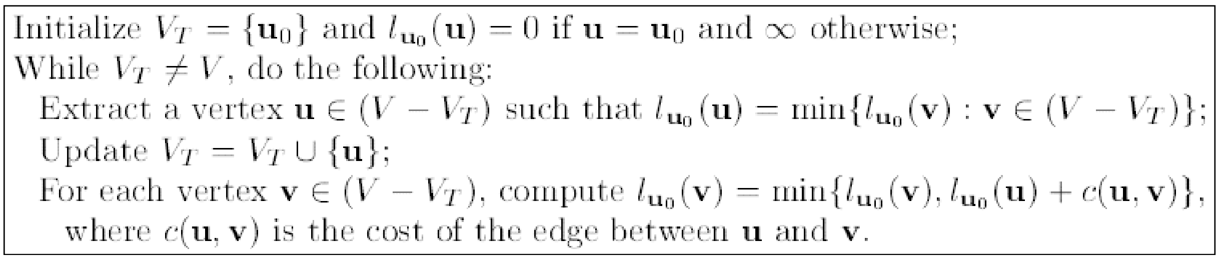
Figure 10. Dijkstra algorithm 2.

Source: Shehzad (2009)

2.4.1.3 Parallel Mesh Algorithms For Grid Graph Shortest Paths with Application to Separation of Touching Chromosomes

The separation of chromosomes that lean against those overlapping in images is approached by Hongchi Shi (1998) in his article. The main process of this application is the search for the smallest path in mesh graphs, due to this important role of the minimum path algorithms, a study was done in some of these algorithms, including the dijkstra algorithm that is presented in Figure 11.

SHI (1998) use a very summarized description and uses set theory to work with vertices and the union operation to complete the minimum way.



```
Initialize $V_T = \{\mathbf{u}_0\}$ and $l_{\mathbf{u}_0}(\mathbf{u}) = 0$ if $\mathbf{u} = \mathbf{u}_0$ and $\infty$ otherwise;
While $V_T \neq V$, do the following:
  Extract a vertex $\mathbf{u} \in (V - V_T)$ such that $l_{\mathbf{u}_0}(\mathbf{u}) = \min\{l_{\mathbf{u}_0}(\mathbf{v}) : \mathbf{v} \in (V - V_T)\}$;
  Update $V_T = V_T \cup \{\mathbf{u}\}$;
  For each vertex $\mathbf{v} \in (V - V_T)$, compute $l_{\mathbf{u}_0}(\mathbf{v}) = \min\{l_{\mathbf{u}_0}(\mathbf{v}), l_{\mathbf{u}_0}(\mathbf{u}) + c(\mathbf{u}, \mathbf{v})\}$,
    where $c(\mathbf{u}, \mathbf{v})$ is the cost of the edge between $\mathbf{u}$ and $\mathbf{v}$.
```

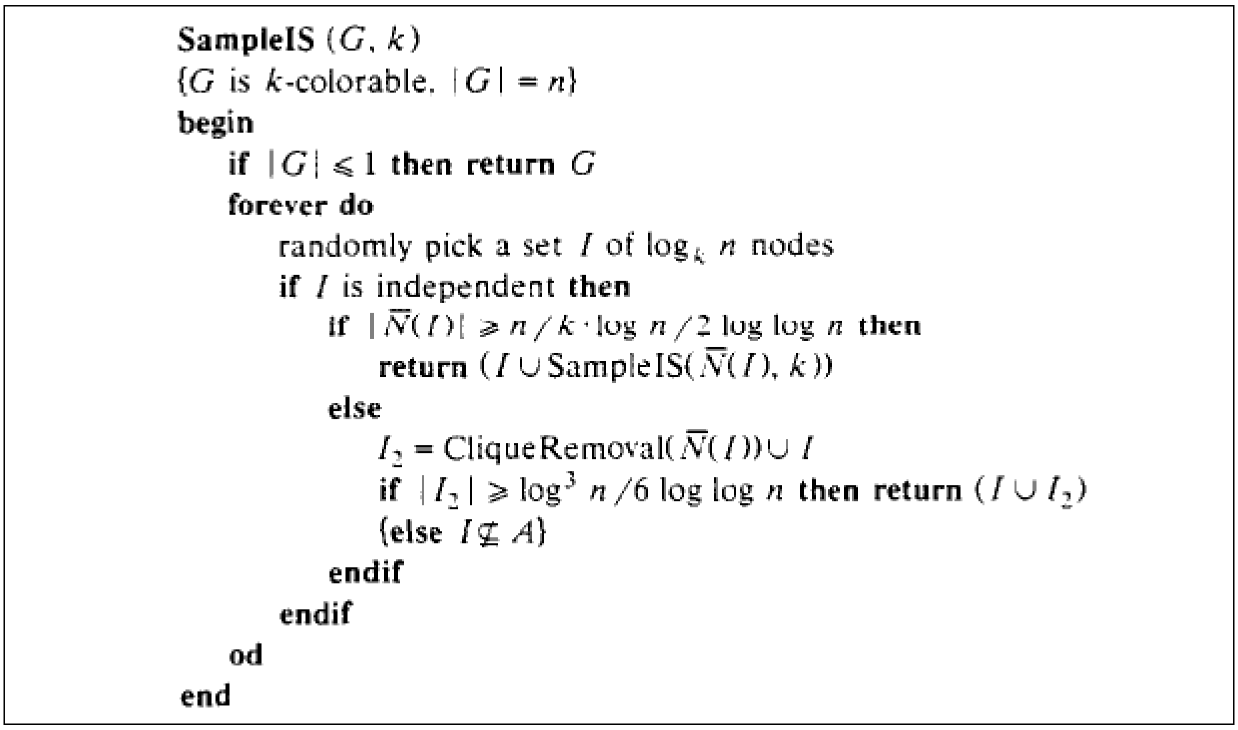
Figure 11. Dijkstra Algorithm 3
Source: SHI (1998)

2.4.2 Coloring

This section will present algorithms found in coloring articles, seeking to identify the way the algorithms function and particularities of each algorithm created by the authors: Halldórsson (1993), Klotz (2002) and Wigderson (1983).

2.4.2.1 A Still Better Performance Guarantee for approximate graph coloring

Halldórsson presented an approximation algorithm to independent sets of a graph in combination with previous work that had reached time O (n (log log n/ log n) ^{3}) for minimum graph color. The new algorithm has reached time (n (log log n) ^{2} / log^{3}n). The algorithm created is presented in Figure 12.



```
**SampleIS** $(G, k)$
$\{G$ is $k$-colorable, $|G| = n\}$
**begin**
    **if** $|G| \leqslant 1$ **then return** $G$
    **forever do**
        randomly pick a set $I$ of $\log_k n$ nodes
        **if** $I$ is independent **then**
            **if** $|\overline{N}(I)| \geqslant n/k \cdot \log n / 2 \log \log n$ **then**
                **return** $(I \cup \text{SampleIS}(\overline{N}(I), k))$
            **else**
                $I_2 = \text{CliqueRemoval}(\overline{N}(I)) \cup I$
                **if** $|I_2| \geqslant \log^3 n / 6 \log \log n$ **then return** $(I \cup I_2)$
                $\{$**else** $I \not\subseteq A\}$
            **endif**
        **endif**
    **od**
**end**
```

Figure 12: Coloring Algorithm 1 Sou
rce: Halldórsson (1993)

The operation of the algorithm is described below: variable I is a set of size log k n of i ndependent graph vertices. The variable ¬(n (i)) contains the original graph vertices removing the items from set I. Each iteration is verified if ¬(n (i)) is greater than (n/k. Log n/2 log log n). If so, the same function is called again by stating the graph contained in the variable ¬(n (i)) united with i and the number of colors. If negative the click -removal function will find an ind ependent set on the graph and unite with i in variable I2. If I2 is greater than $\log^3$ n/6 log log n will be returned to the union of i with i2.

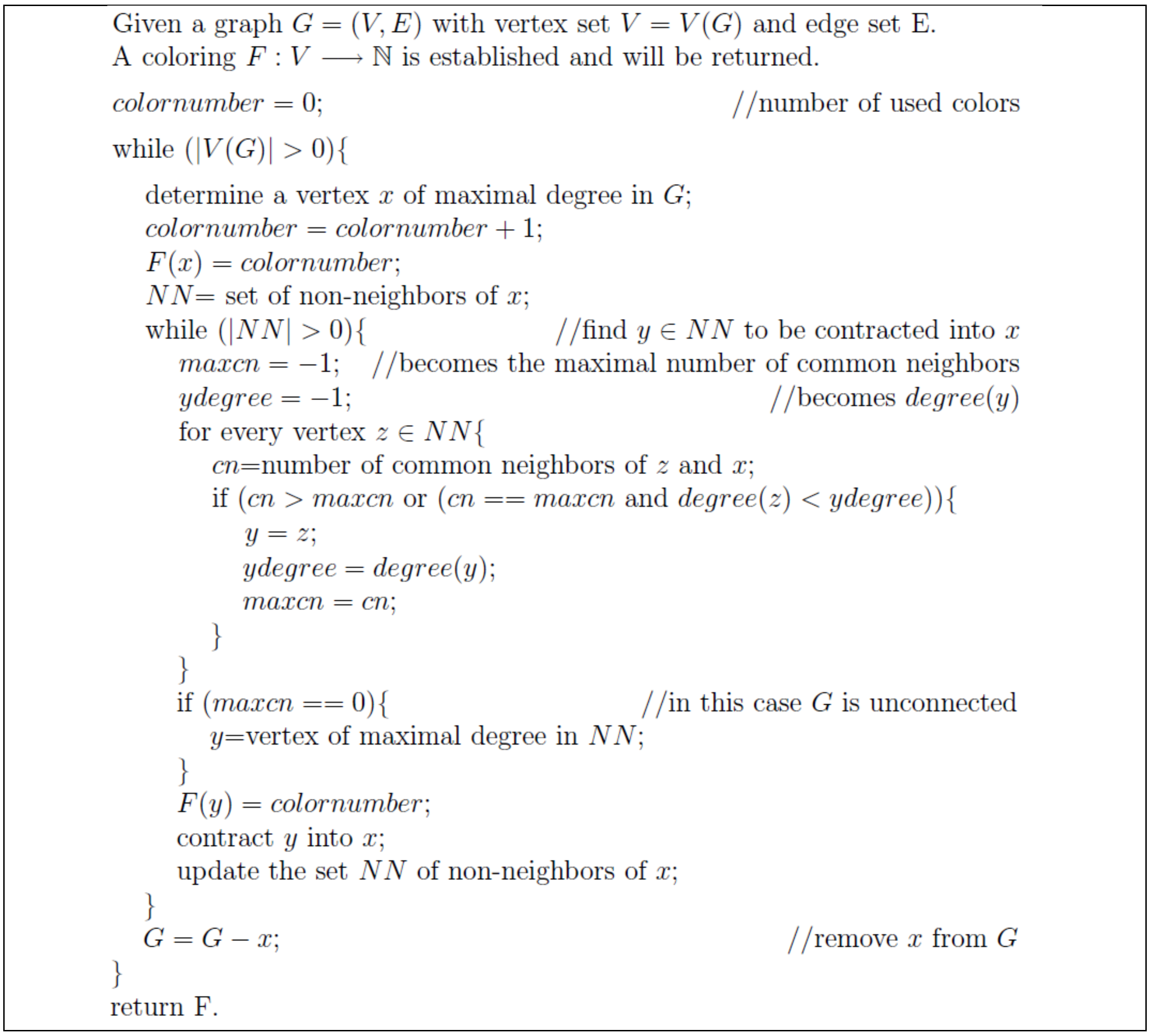
In this algorithm it is possible to notice the use of recursiveness in the function to find i ndependent vertices and the use of variables defining specific sets for the functions

2.4.2.2 Graph Coloring Algorithms

(KLOTZ, 2002) Evaluated deterministic and sequential contraction algorithms to find t he chromatic number of a graph. The contraction algorithm, presented in Figure 13.

The operation of the algorithm is specified below: For each vertex of the graph a larger
and colorful vertex is chosen. Then an NN set is defined with the vertices that are not adjacent
to x. So this set is traveled by identifying all items that can be contracted in X, contraction of t
he vertex found in X and updates the NN set. At the end of the search in nn x is removed from
g and the operation repeats. Unlike previous algorithms the example above has a search that co
ntracts two vertices in one and although it uses sets do not use the standard sets of sets.

An interesting feature of this algorithm is the use of the contraction of various vertices i
n one and the description of the code that varies between a standard programming language an
d explanations in natural language.

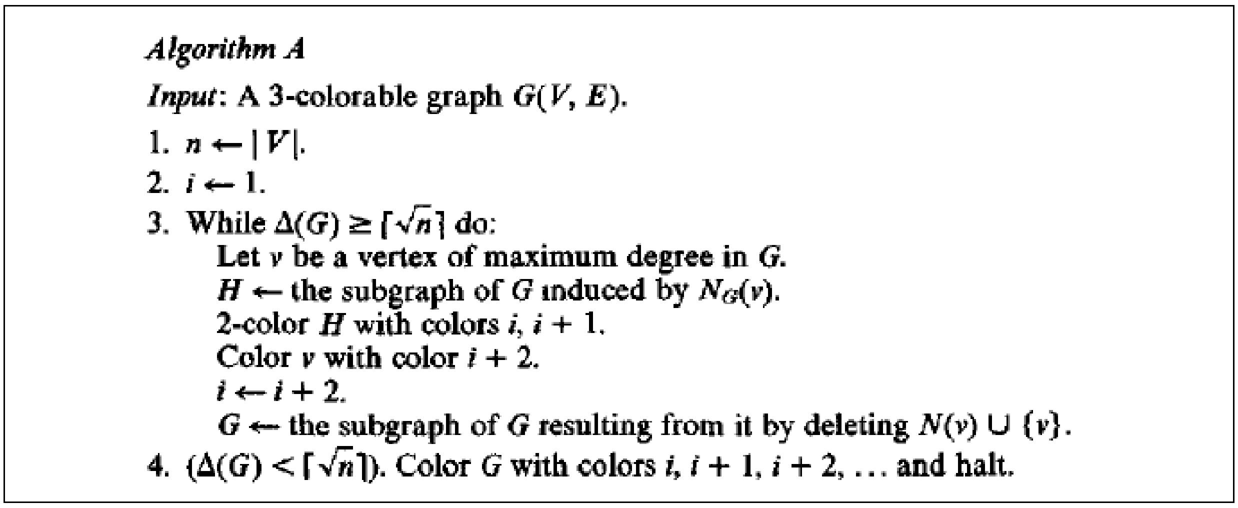


```
Given a graph $G = (V, E)$ with vertex set $V = V(G)$ and edge set E.
A coloring $F : V \longrightarrow \mathbb{N}$ is established and will be returned.
$colornumber = 0;$                                  //number of used colors
while $(|V(G)| > 0)\{$
   determine a vertex $x$ of maximal degree in $G$;
   $colornumber = colornumber + 1;$
   $F(x) = colornumber;$
   $NN$= set of non-neighbors of $x$;
   while $(|NN| > 0)\{$                   //find $y \in NN$ to be contracted into $x$
      $maxcn = -1;$   //becomes the maximal number of common neighbors
      $ydegree = -1;$                              //becomes $degree(y)$
      for every vertex $z \in NN\{$
         $cn$=number of common neighbors of $z$ and $x$;
         if $(cn > maxcn$ or $(cn == maxcn$ and $degree(z) < ydegree))\{$
            $y = z;$
            $ydegree = degree(y);$
            $maxcn = cn;$
         }
      }
      if $(maxcn == 0)\{$                     //in this case $G$ is unconnected
         $y$=vertex of maximal degree in $NN$;
      }
      $F(y) = colornumber;$
      contract $y$ into $x$;
      update the set $NN$ of non-neighbors of $x$;
   }
   $G = G - x;$                                  //remove $x$ from $G$
}
return F.
```

Figure 13. Coloring Algorithm 2 Sou
rce: Klotz (2002)

2.4.2.3 Improving The Performance Guarantee for Approximate Graph Coloring

Wigderson's work focuses on the search for a more efficient algorithm than (n/log n). To reach the goal, the author presents some algorithms, including the algorithm presented in Fi gure 14, for coloring 3-chromatic graphs.



```
Algorithm A
Input: A 3-colorable graph $G(V, E)$.
1. $n \leftarrow |V|$.
2. $i \leftarrow 1$.
3. While $\Delta(G) \geq \lceil \sqrt{n} \rceil$ do:
      Let $v$ be a vertex of maximum degree in $G$.
      $H \leftarrow$ the subgraph of $G$ induced by $N_G(v)$.
      2-color $H$ with colors $i, i + 1$.
      Color $v$ with color $i + 2$.
      $i \leftarrow i + 2$.
      $G \leftarrow$ the subgraph of $G$ resulting from it by deleting $N(v) \cup \{v\}$.
4. $(\Delta(G) < \lceil \sqrt{n} \rceil)$. Color $G$ with colors $i, i + 1, i + 2, \ldots$ and halt.
```

Figure 14: Coloring Algorithm 3

Source: Wigderson (1983).

The functioning of the algorithm occurs as follows: a subgress is created with the vortex v. Th e vertices of the undergraph h are colorful alternately with 2 colors, then the vertex V is colore d with a third color. GRAFO G receives this change, marks the vertices as colorful and operati ons repeat to the highest degree of the graph is smaller than the square root of the non -colored items.

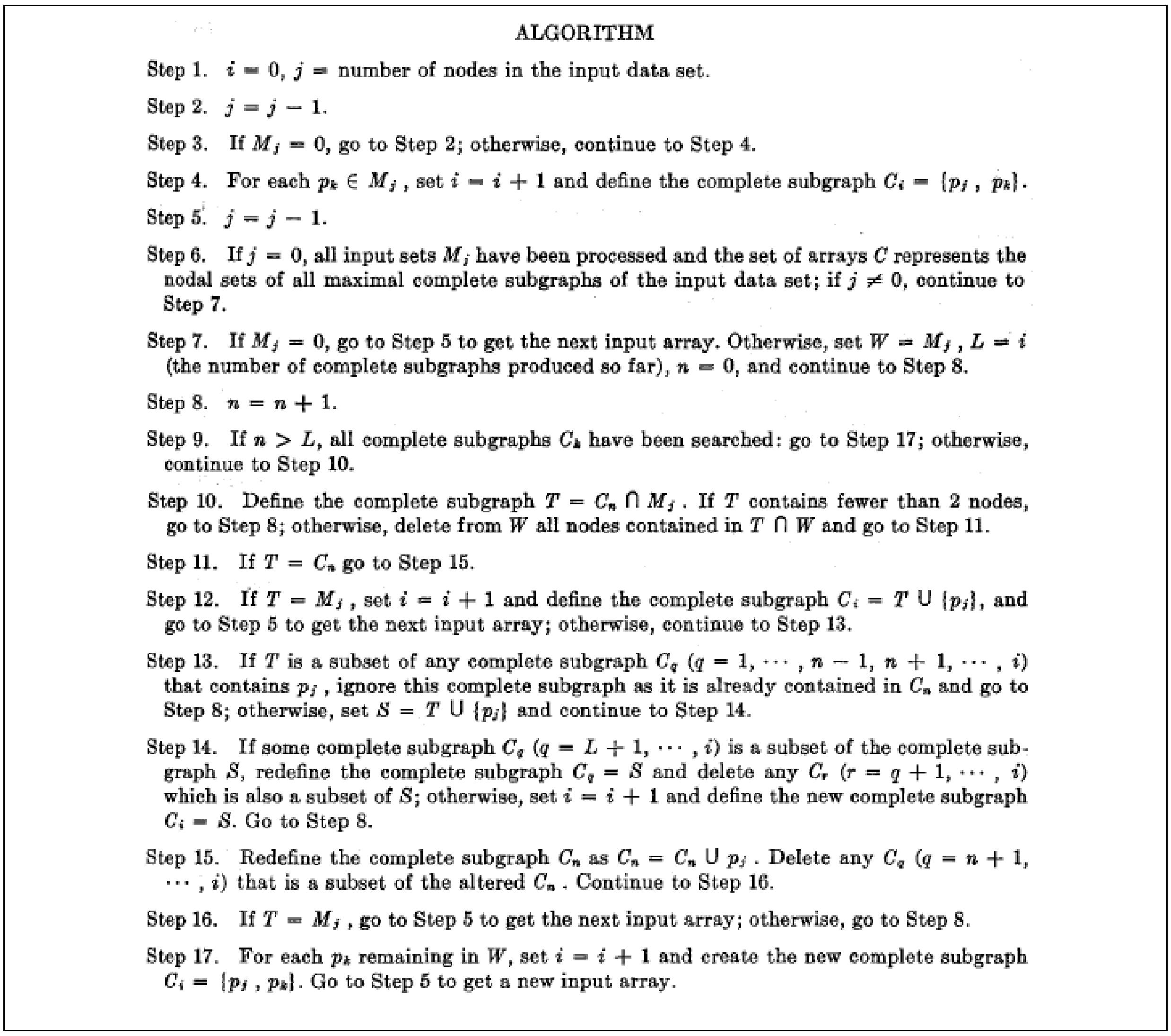
Some important features in this algorithm are: use of pre -defined functions in the articl e as NG (V) that contains the vicinity of a vertex v. The use of the theory of sets to remove the non -color vertices of G and add the color.

2.4.3 Click Maximum

Some articles reporting applications of the maximum click problem were published, the n some algorithms created by the authors will be presented: Auguston (1970), Kreher (1999) a nd Carmo (2012).

2.4.3.1 An Analysis of Some Graph Theoretical Cluster Techniques

Auguston presents two algorithms to identify the type of cluster. Its focus is on clusters developed for systems that return information known as thesauri. This is used algorithms that seek the largest complete subgraph. Figure 15 presents the created algorithm.



ALGORITHM

Step 1. $i = 0$, $j =$ number of nodes in the input data set.

Step 2. $j = j - 1$.

Step 3. If $M_j = 0$, go to Step 2; otherwise, continue to Step 4.

Step 4. For each $p_k \in M_j$, set $i = i + 1$ and define the complete subgraph $C_i = \{p_j, p_k\}$.

Step 5. $j = j - 1$.

Step 6. If $j = 0$, all input sets M_j have been processed and the set of arrays C represents the nodal sets of all maximal complete subgraphs of the input data set; if $j \neq 0$, continue to Step 7.

Step 7. If $M_j = 0$, go to Step 5 to get the next input array. Otherwise, set $W = M_j$, $L = i$ (the number of complete subgraphs produced so far), $n = 0$, and continue to Step 8.

Step 8. $n = n + 1$.

Step 9. If $n > L$, all complete subgraphs C_k have been searched: go to Step 17; otherwise, continue to Step 10.

Step 10. Define the complete subgraph $T = C_n \cap M_j$. If T contains fewer than 2 nodes, go to Step 8; otherwise, delete from W all nodes contained in $T \cap W$ and go to Step 11.

Step 11. If $T = C_n$ go to Step 15.

Step 12. If $T = M_j$, set $i = i + 1$ and define the complete subgraph $C_i = T \cup \{p_j\}$, and go to Step 5 to get the next input array; otherwise, continue to Step 13.

Step 13. If T is a subset of any complete subgraph C_q $(q = 1, \cdots, n - 1, n + 1, \cdots, i)$ that contains p_j, ignore this complete subgraph as it is already contained in C_n and go to Step 8; otherwise, set $S = T \cup \{p_j\}$ and continue to Step 14.

Step 14. If some complete subgraph C_q $(q = L + 1, \cdots, i)$ is a subset of the complete subgraph S, redefine the complete subgraph $C_q = S$ and delete any C_r $(r = q + 1, \cdots, i)$ which is also a subset of S; otherwise, set $i = i + 1$ and define the new complete subgraph $C_i = S$. Go to Step 8.

Step 15. Redefine the complete subgraph C_n as $C_n = C_n \cup p_j$. Delete any C_q $(q = n + 1, \cdots, i)$ that is a subset of the altered C_n. Continue to Step 16.

Step 16. If $T = M_j$, go to Step 5 to get the next input array; otherwise, go to Step 8.

Step 17. For each p_k remaining in W, set $i = i + 1$ and create the new complete subgraph $C_i = \{p_j, p_k\}$. Go to Step 5 to get a new input array.

Figure 15. Algorimo Click Maximum 1

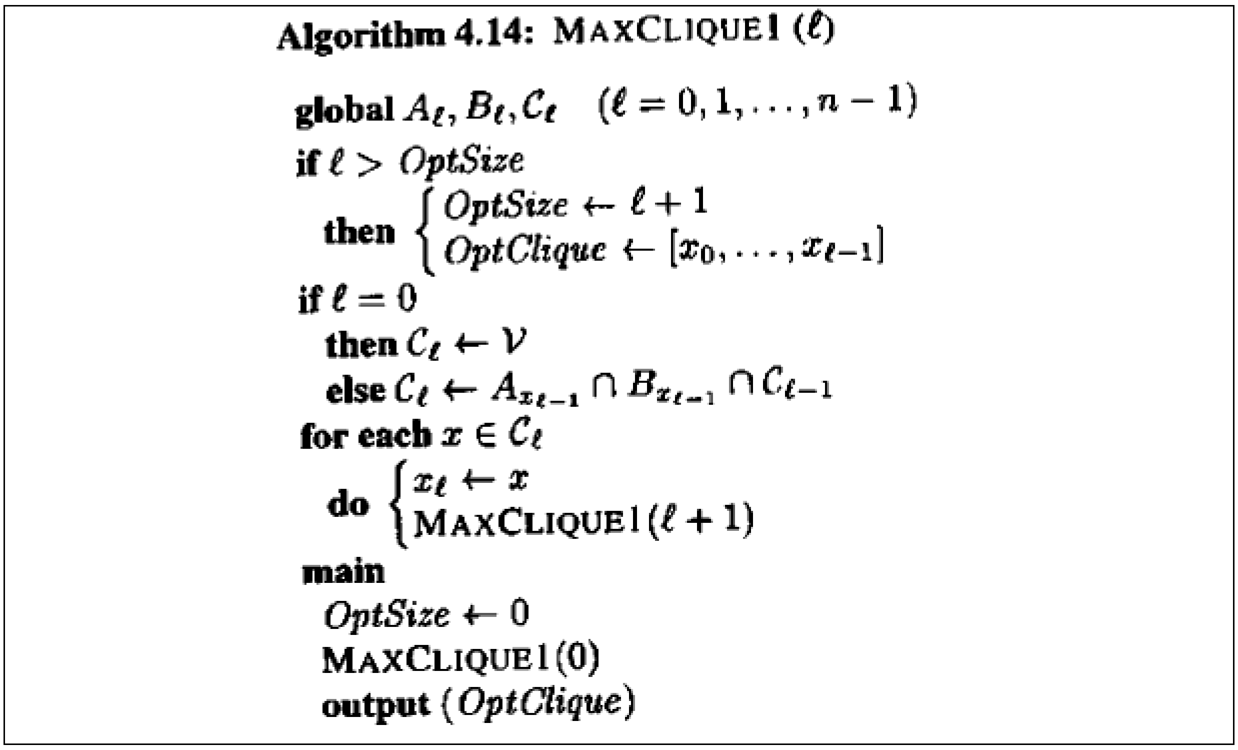
Source: Auguston (1970)

The algorithm works through a loop that runs through the vertices of the graph and eac h vertex verifies its surroundings by keeping in set C. While the algorithm travels the surround ing M, some checks are applied to unite the items in C at the end of the loop between the vertic es, the algorithm gets the maximum click within C.

Among the main characteristics of the code is the frequent use of the goto line function which is uncommon in the description of this type of algorithm. In addition, the use of set theo ry so that with each M iteration tests and unites new items A C that at the end of the algorithm becomes the maximum click of the graph.

2.4.3.2 Discreet Mathematics and ITS Application NS

(Kreher, 1999) presents in the 4th chapter of his book a recursive algorithm to obtain th e maximum click of the graph. The algorithm is presented in Figure 16.



```
Algorithm 4.14: MAXCLIQUE1 ($\ell$)

global $A_\ell, B_\ell, C_\ell$   ($\ell = 0, 1, \ldots, n-1$)
if $\ell > OptSize$
  then { $OptSize \leftarrow \ell + 1$
       { $OptClique \leftarrow [x_0, \ldots, x_{\ell-1}]$
if $\ell = 0$
  then $C_\ell \leftarrow V$
  else $C_\ell \leftarrow A_{x_{\ell-1}} \cap B_{x_{\ell-1}} \cap C_{\ell-1}$
for each $x \in C_\ell$
  do { $x_\ell \leftarrow x$
     { MAXCLIQUE1($\ell + 1$)
main
  $OptSize \leftarrow 0$
  MAXCLIQUE1(0)
  output ($OptClique$)
```

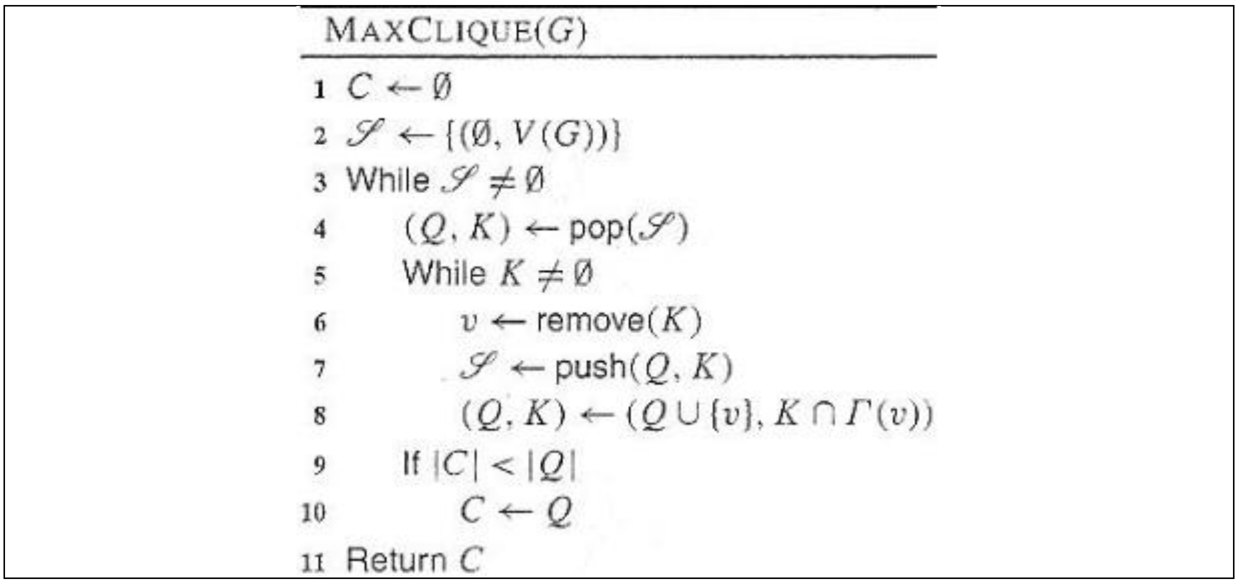
Figure 16. Algorithm Click Maximum 2
Source: Kreher (1999)

This algorithm operates recursively, in the first iteration the set C receives all the vertic es of the graph, and the XL vector receives the first VI VERTE and the function is again called . In the next checks the Optsize control variable that holds the size of the largest click and the L variable, which contains a current click size counter, are compared. If l is greater than optsiz e, the optsize value increases and the optclique variable gets the new largest click. After this st ep, the next adjacent vertices of VI are verified and in each iteration is verified if the current ve rtex has surroundings with all the adjacent vertices of VI and with VI itself. These steps are repeated in all vertices of set C, at the end of the process the optclique variable will have maximum click.

The algorithm has some interesting characteristics such as the form that sets A and B are used, with each iteration A A receive the adjacencies of the last vertex evaluated and B the next vertices to be evaluated from the last vertex. In addition, the use of recursive function guaranteed a very clean and defined code in a few lines

2.4.3.3 branch and bound algorithms for the maximum click problem

Carmo verified some algorithms for the exact solution of the maximum click problem and described comparative results of eight algorithms. Figure 17 presents one of these algorithms.



```
MaxClique($G$)
$C \leftarrow \emptyset$
$\mathscr{S} \leftarrow \{(\emptyset, V(G))\}$
While $\mathscr{S} \neq \emptyset$
    $(Q, K) \leftarrow$ pop($\mathscr{S}$)
    While $K \neq \emptyset$
        $v \leftarrow$ remove($K$)
        $\mathscr{S} \leftarrow$ push($Q, K$)
        $(Q, K) \leftarrow (Q \cup \{v\}, K \cap \Gamma(v))$
    If $|C| < |Q|$
        $C \leftarrow Q$
Return $C$
```

Figure 17. Algorithm Click Maximum 3
Source: Carmo (2012).

The algorithm works through two repetition bonds, the external loop controls the vertices that will be evaluated. In the inner loop that travels all the vertices is defined an initial vertex and verified its click during iterations, each iteration is done a test to see if the current click exceeded the last click maximum. At the end of the internal loop process the click of the vertex chosen in the outer loop is defined, and the rest of the vertices is evaluated until the entire graph has been researched.

An interesting point of this algorithm is the control made using the stack and the execution based on set theory.

2.4.4 Considerations on algorithms

Although the nine algorithms presented have been made on different dates by various a uthors, some characteristics are repeated in several of them and will be presented along with th e number of times they were found in different algorithms.

Table 1 below shows in the left column the identified characteristics and in the right col umn the number of times each feature appeared in the algorithms.

Table 1. Characteristics of the algorithms

Característica utilizada	**Número de algoritmos**
Laço de repetição	9
Condicional	9
Conjuntos	9
Função	5
Variáveis especiais	3
Recursividade	2
Tabela	2
Pilha	2
Fila	1

Some of the characteristics appear in all algorithms which shows that they are a standard form of graph work. The characteristics that appeared less often are usually variable used to make so me control in the code or in the case of the last characteristic cited, to summarize some informa tion previously described in the article, for example, the surroundings of a vertex.

Although there are characteristics that are repeated in different algorithms, the syntax defined by each author to describe the algorithms are different. In addition, the algorithms were written without the concern of being executed on a computer, so there are no variable statements, setti ng set values and other important features in a programming language.

The algorithms were sought in articles that addressed specific topics that use graph theory and developed grammatics were created to support only the problems covered, so each author creat ed a different syntax with the specific scope to meet the problem studied in the article. All arti cles studied had definitions during the article that were later used in the algorithms. This mode of approach that uses the text of the article to specify functions

From the algorithms abstract some specifications that in a programming language would need t o have been specified.

Each algorithm has different syntaxes appropriate to the needs of the algorithm, which simplifi es the memorization of the terms. However, as they are very different generate the need to und erstand the way each author found to express the algorithm.

The data observed in the algorithms will be evaluated during the DSL project, aiming to create a language that can solve different graph problems with a format that is suitable for the types o f operations that are normally used in graphs.

3 Development

The implementation of language followed an iterative method, that is, from an initial pr ototype, which idealized what language should look like, versions were created by implementi ng pieces of language and these pieces were changed case during use that was identified that so mething best suited to the context of language could replace it.

The concept of implementing language is iteratively quoted in the book *Domain-Specific Language* by Martin Fowler. Based on this concept, language has evolved from what was initially idealized, available in Appendix A.1, to the final version, as shown in the exampl e of the in -depth search algorithm, available in Appendix A.2.

This chapter is documented the language development process, presenting the tools and design patterns used. It is divided into functional requirements, language, implementation and examples.

3.1 Functional Requirements

Functional Requirements (RF) are responsible for presenting the features that language and IDE should contemplate and will be displayed below:

3.1.1 Language Requirements

The functional requirements of the language will be presented below.

•RF01: Language should allow the user to solve graph problems through the def ined syntax; • RF02: Language must implement set handling operations; • RF0 3: Language should implement graph search strategies: Width search and depth search. • RF04: Language should allow the implementation of new graph searc h strategies. • RF05: Language should allow creation and use of functions.

•RF06: Language must allow recursion in its functions. • RF07: Language sho uld allow the creation of variables of the type: boolean, number, text, set and gra ph. • RF08: Language should have a function to write text on the screen as a pr ogram information output to the user.

3.1.2 IDE Requirements

IDE's functional requirements will be presented below.

•RF01: IDE should be able to perform algorithms in the language being defined in t his project. • RF02: IDE should produce a text output when the language writing fu nction is called. • RF03: IDE should be able to open, edit and save the files produc ed by it. • RF04: IDE should have a syntax.

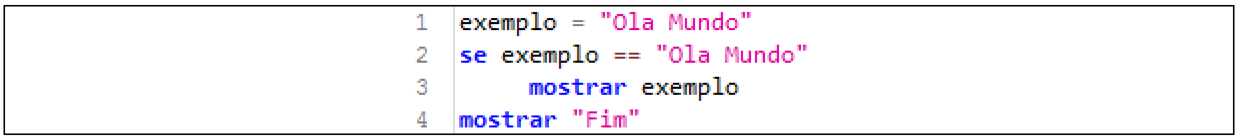
3.2 Language

This section presents the structures, operation and the types of language.

3.2.1 Language Structure

Language commands are designed to meet the aspects cited in the introduction of this d ocument as important in producing a DSL. The aspects are: being easy to understand and exec utable by the computer, have a sense of fluency, expressiveness limited to the context.

Language is oriented by identification, that is, the beginning and end of a context is based on the tab of the code. Figure 18 presents this feature.



```
exemplo = "Ola Mundo"
se exemplo == "Ola Mundo"
    mostrar exemplo
mostrar "Fim"
```

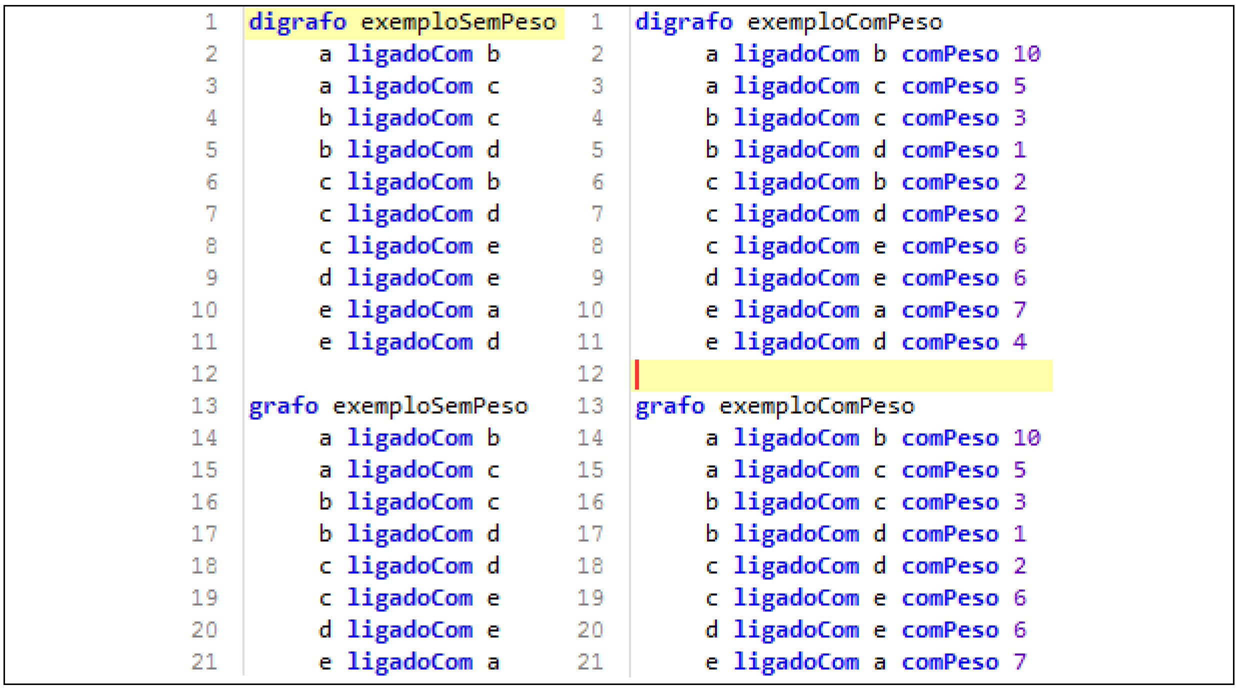
Figure 18. Hello World in DSLFG

In addition to the orientation by identity in Figure 18, it may be noted to the absence of the need for a main function. The commands are compiled and executed in the direction from top to bottom, without the need to belong to any special function or structure.   Another important aspect is that there is no visible character at the end of the instructions, language recognizes the end of an instruction with the line break.

The next section sub -items will present the main commands of the language and explain the operation.

3.2.1.1 Graph builder

The graph builder was planned to create the graph connections and generate the vertices in the same step. The command is presented in Figure 19.

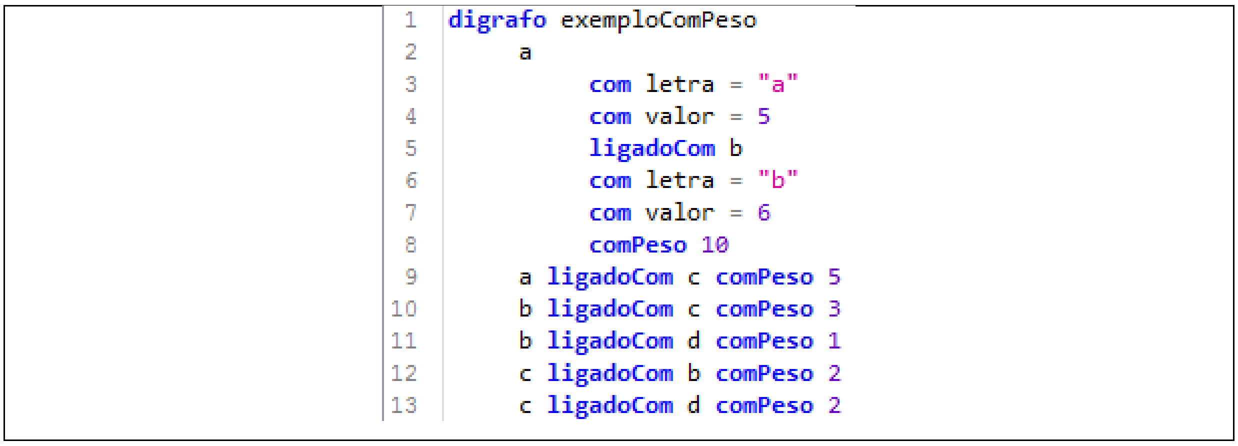


```
digrafo exemploSemPeso
    a ligadoCom b
    a ligadoCom c
    b ligadoCom c
    b ligadoCom d
    c ligadoCom b
    c ligadoCom d
    c ligadoCom e
    d ligadoCom e
    e ligadoCom a
    e ligadoCom d

grafo exemploSemPeso
    a ligadoCom b
    a ligadoCom c
    b ligadoCom c
    b ligadoCom d
    c ligadoCom d
    c ligadoCom e
    d ligadoCom e
    e ligadoCom a
```

```
digrafo exemploComPeso
    a ligadoCom b comPeso 10
    a ligadoCom c comPeso 5
    b ligadoCom c comPeso 3
    b ligadoCom d comPeso 1
    c ligadoCom b comPeso 2
    c ligadoCom d comPeso 2
    c ligadoCom e comPeso 6
    d ligadoCom e comPeso 6
    e ligadoCom a comPeso 7
    e ligadoCom d comPeso 4

grafo exemploComPeso
    a ligadoCom b comPeso 10
    a ligadoCom c comPeso 5
    b ligadoCom c comPeso 3
    b ligadoCom d comPeso 1
    c ligadoCom d comPeso 2
    c ligadoCom e comPeso 6
    d ligadoCom e comPeso 6
    e ligadoCom a comPeso 7
```

Figure 19. GRAFO CREATION

The command recognizes the identifiers of the vertices and generates them in the same process that connects it with the second vertex. Two other characteristics can be noted in Figure 19, th e distinction from graph to digit and the possibility of adding the weight of a vertex. In additio n to the format presented, it is also possible to add properties to the vertices. Figure 20 exempl ifies the syntax.

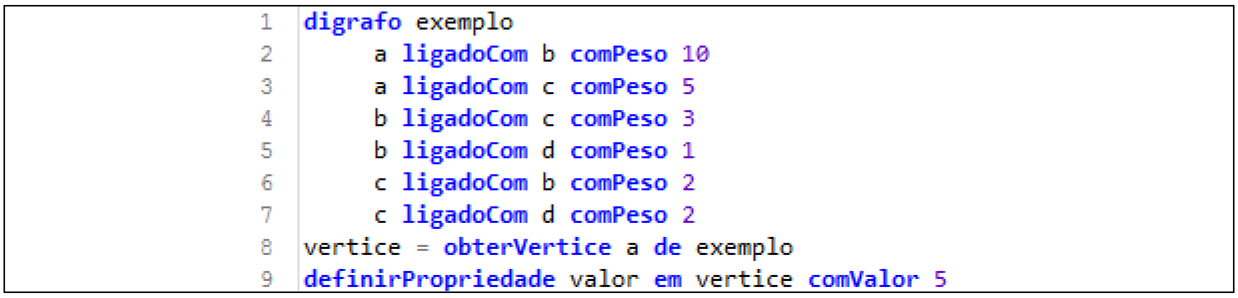


```
digrafo exemploComPeso
    a
        com letra = "a"
        com valor = 5
        ligadoCom b
        com letra = "b"
        com valor = 6
        comPeso 10
    a ligadoCom c comPeso 5
    b ligadoCom c comPeso 3
    b ligadoCom d comPeso 1
    c ligadoCom b comPeso 2
    c ligadoCom d comPeso 2
```

Figure 20. GRAFO WITH PROPERTIES.

The "com" clause is called right after the vertex identifier and adds all the properties mentione d within the vertex.

The second way to add properties is with a command after the graph is already created. The de fine property command requests the name of the property, the vertex in which it will be added and the value, with this information adds the new property in the vertex. The instruction is pre sented at line nine of Figure 21.

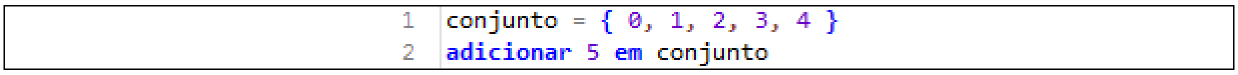


```
1 digrafo exemplo
2     a ligadoCom b comPeso 10
3     a ligadoCom c comPeso 5
4     b ligadoCom c comPeso 3
5     b ligadoCom d comPeso 1
6     c ligadoCom b comPeso 2
7     c ligadoCom d comPeso 2
8 vertice = obterVertice a de exemplo
9 definirPropriedade valor em vertice comValor 5
```

Figure 21. Define vertex property

3.2.1.2 Set Creation

As identified in Chapter 2, graph algorithms usually use set operations to solve their problems.
To meet this need language has instructions for creating and operating sets. Figure 22 shows t
he creation of a set followed by the addition of an element.



```
conjunto = { 0, 1, 2, 3, 4 }
adicionar 5 em conjunto
```

Figure 22. Set creation

The creation of the whole is done by the characters of opening and closing key and the insertio
n of elements is made between the keys, separated by comma. After creating a set you can add
new elements using the instruction presented in the second line of Figure 22.

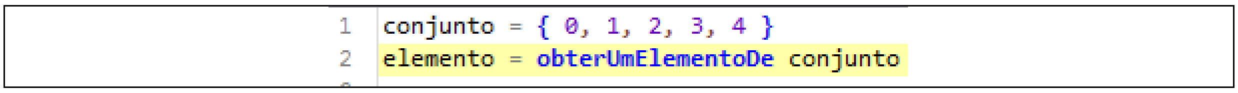
In addition to these commands, others were implemented to be possible to manipulate the sets.
The other commands needed for the handling of the set will be presented in the next subsection
s.

3.2.1.3 Language Functions

To enable the manipulation of graph and set structures, special commands have been created th
at allow and manipulate the data of these structures. The twelve functions created for this purp
ose will be presented below.

3.2.1.3.1 Obtain a set element

Two ways of obtaining elements of one set, one random and another ordered elements were cr
eated. The random form seeks any element within a specified set and returns the result to the i
nstruction in which it was called. Figure 23 presents the syntax of the instruction.

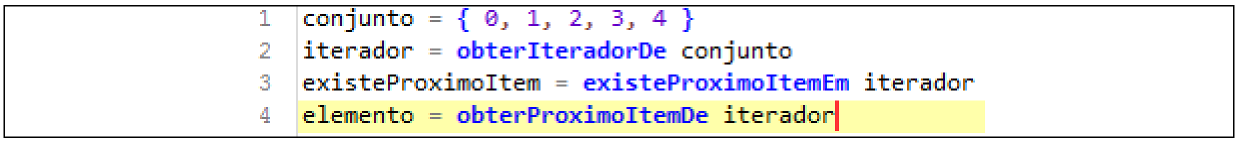


```
conjunto = { 0, 1, 2, 3, 4 }
elemento = obterUmElementoDe conjunto
```

Figure 23. Obtain a set of randomly.

The second form is divided into three steps, first the iterator of the set is obtained, then questio
ned if there are more unlisted elements, in the last step the next

element is obtained from the iterator. Figure 24 illustrates the creation of a set, obtaining the it erator, checking next element and acquisition of the next element.



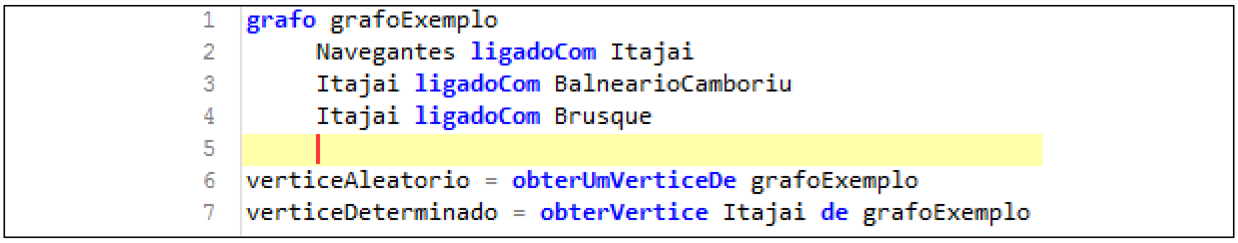
```
1  conjunto = { 0, 1, 2, 3, 4 }
2  iterador = obterIteradorDe conjunto
3  existeProximoItem = existeProximoItemEm iterador
4  elemento = obterProximoItemDe iterador
```

Figure 24. Obtain set element with iterated   r.

The third line command will get a boolean value that can be tested on a conditional deviation o r repetition loop. In addition to these functions, four sets of set theory that return new sets wer e implemented. These operations will be cited in the operators' subsection.

3.2.1.3.2 Get a vertex from a graph

It is possible to get a vertex of a graph in two ways, specifying the desired vertex or tak ing anyone. The two commands are presented in Figure 25. On the sixth line the command had returned a random vertex of the graph and in the seventh the vertex "*Itajai*".

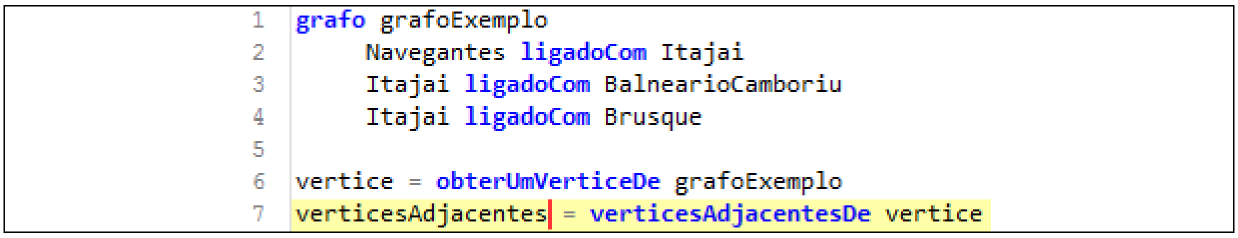


```
1  grafo grafoExemplo
2       Navegantes ligadoCom Itajai
3       Itajai ligadoCom BalnearioCamboriu
4       Itajai ligadoCom Brusque
5
6  verticeAleatorio = obterUmVerticeDe grafoExemplo
7  verticeDeterminado = obterVertice Itajai de grafoExemplo
```

Figure 25. Get a vertex.

3.2.1.3.3 Obtain adjacent vertices

The function for obtaining adjacent vertices needs to select a root vertex, ie after extract ing a vertex from a graph the command to obtain adjacent vertices receives this parameter and with this information returns a set containing the adjacent vertices of the selected vertex. Figur e 26 presents the use of the command.

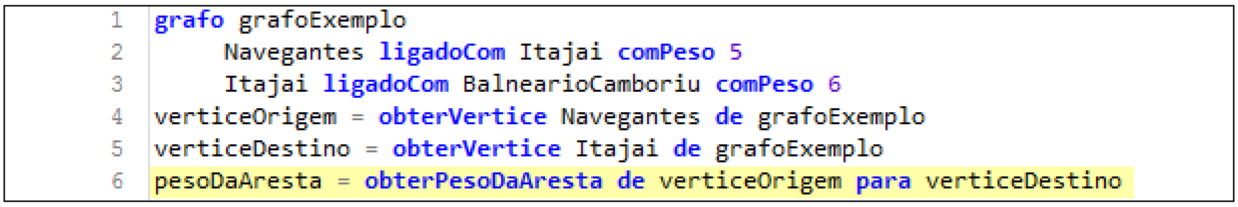


```
1  grafo grafoExemplo
2       Navegantes ligadoCom Itajai
3       Itajai ligadoCom BalnearioCamboriu
4       Itajai ligadoCom Brusque
5
6  vertice = obterUmVerticeDe grafoExemplo
7  verticesAdjacentes = verticesAdjacentesDe vertice
```

Figure 26. Obtain adjacent vertices

3.2.1.3.4 Get weight of the edge

The function for obtaining the weight of an edge needs the vertex of origin and destinat
ion to be informed, with this information the command returns the weight of the edge between
the vertices. Figure 27 presents the use of the command, in this case the command will return t
he five.

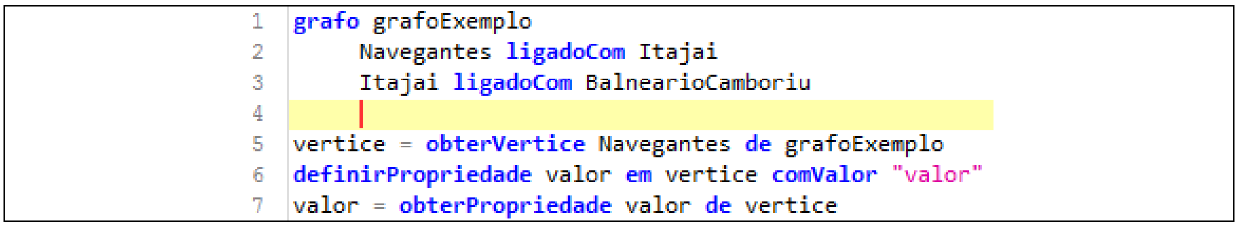


```
grafo grafoExemplo
     Navegantes ligadoCom Itajai comPeso 5
     Itajai ligadoCom BalnearioCamboriu comPeso 6
verticeOrigem = obterVertice Navegantes de grafoExemplo
verticeDestino = obterVertice Itajai de grafoExemplo
pesoDaAresta = obterPesoDaAresta de verticeOrigem para verticeDestino
```

Figure 27. Get weight of the edge.

3.2.1.3.5 Obtain vertex ownership

To obtain the desired property of a vertex, a function that needs to receive the desired v
ertex and the name of the required property must be created, and then return the value of the pr
operty. Figure 28 presents the use of the function in line seven.

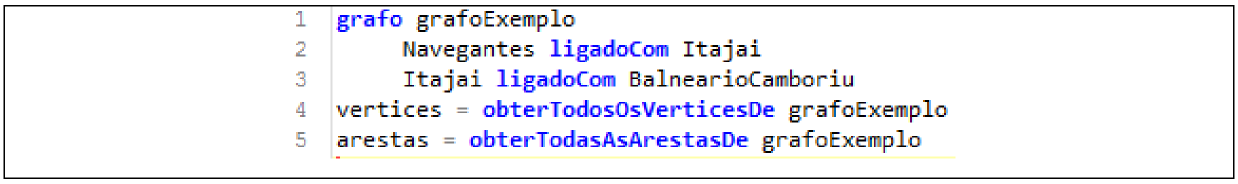


```
1 grafo grafoExemplo
2      Navegantes ligadoCom Itajai
3      Itajai ligadoCom BalnearioCamboriu
4      
5 vertice = obterVertice Navegantes de grafoExemplo
6 definirPropriedade valor em vertice comValor "valor"
7 valor = obterPropriedade valor de vertice
```

Figure 28. Obtain vertex property.

3.2.1.3.6 Get all the vertices and edges of a graph

The function to obtain all the vertices of a graph returns a set containing all vertices of t
he requested graph. The command syntax is presented on line four in Figure 29. In the fifth lin
e of the same figure is presented the function to obtain all graph edges, returning a set with all
graph connections.

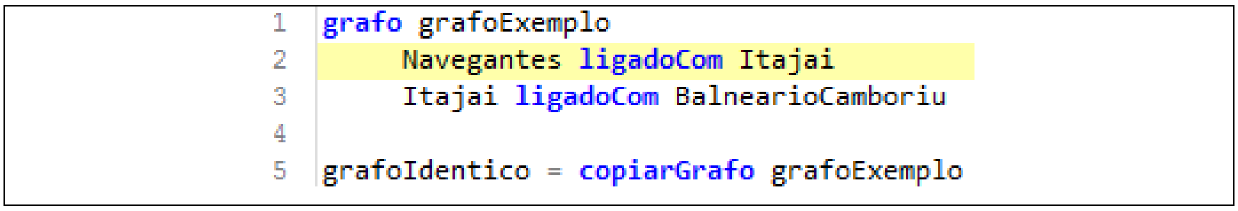


```
grafo grafoExemplo
	Navegantes ligadoCom Itajai
	Itajai ligadoCom BalnearioCamboriu
vertices = obterTodosOsVerticesDe grafoExemplo
arestas = obterTodasAsArestasDe grafoExemplo
```

Figure 29. Get all the vertices and edges of a graph.

3.2.1.3.7 Copy graph

The copy function returns a copy of the past graph, different from what happens in a no rmal attribution where the reference is passed. Figure 30 presents the use of the command



```
grafo grafoExemplo
	Navegantes ligadoCom Itajai
	Itajai ligadoCom BalnearioCamboriu

grafoIdentico = copiarGrafo grafoExemplo
```

Figure 30. Copy graph

3.2.1.4 Operators

Operators are functions responsible for arithmetic and logical operations between langu age objects. The following tables will present all existing operators in language.

The first will be Table 1 that will present the arithmetic operators. In the operator colu mn will be presented the operator token and in the description column will explain its functiona lity within the language.

Operador	**Descrição**
+	Responsável pela soma em caso de números e concatenação para Strings.
-	Responsável pela subtração em caso de números e desmembramento para Strings.
*	Responsável pela multiplicação de números.
/	Responsável pela divisão de números

mod	Após uma divisão, retorna o resto da operação
div	Após uma divisão, retorna o quociente da operação como um número inteiro, removendo as casas decimais

Table 1.

The second will be Table 2 that will present the set operators. The operator column wil l be presented the token used in language, the standard symbol column will be the symbol used in the literature for the operation and the description column will explain the operator's functio nality.

Operador	**Símbolo padrão**	**Descrição**
retirando	\	Todos os elementos que estão no conjunto da esquerda, mas não estão no conjunto da direita.
unindo	∪	A união de todos os elementos da esquerda com os elementos da direita.
interseccao	∩	Todos os elementos que estão em ambos os conjuntos da esquerda e direita.
diferenca	∆	Todos os elementos que estão no conjunto da esquerda mas não estão no conjunto da direita mais os elementos que estão no conjunto da direita mas não estão no da direita. O contrário da operação de intersecção.

Table 2. Set operators

Table 3 will present the logical operators of language. The operator column will be pre sented the operator token and in the description column will be explained to

his functionality within the language.

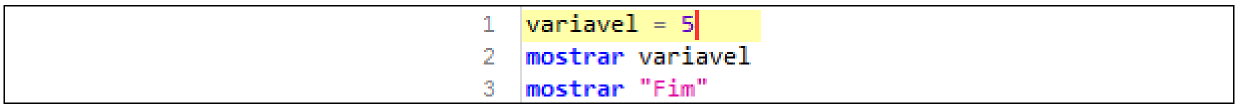
Operador	**Descrição**
==	Retorna verdadeiro se o objeto da esquerda é igual ao da direita
!=	Retorna verdadeiro se o objeto da esquerda é diferente ao da direita
<	Retorna verdadeiro se o objeto da esquerda é menor que o da direita
<=	Retorna verdadeiro se o objeto da esquerda é menor ou igual ao da direita

>	Retorna verdadeiro se o objeto da esquerda é maior que o da direita
>=	Retorna verdadeiro se o objeto da esquerda é maior ou igual ao da direita
subconjunto	Retorna verdadeiro se o conjunto da esquerda é um subconjunto do conjunto da direita
subconjuntoProprioDe	Retorna verdadeiro se o conjunto da esquerda é um subconjunto do conjunto da direita e os dois conjuntos são diferentes.
eUmElementoDe	Retorna verdadeiro se o elemento da esquerda pertence ao conjunto da direita
naoEUmElementoDe	Retorna verdadeiro se o elemento da esquerda não pertence ao conjunto da direita

Table 3. Logical operators

3.2.1.5 SHOWING COMMAND

The Show Command was created so that the language user could present the result of t heir algorithms. The instruction needs a parameter that contains the value that will be presente d, this value can be a text, number, boolean, language constant or a variable. Figure 31 present s the use of instruction.

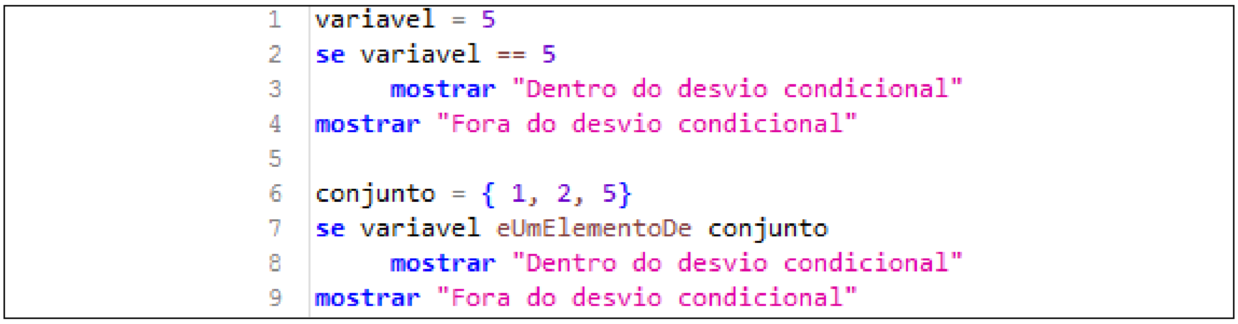


```
variavel = 5
mostrar variavel
mostrar "Fim"
```

Figure 31. SHOW COMMAND

3.2.1.6 Conditional Deviation

The “SE” command was created to enable the execution of commands only after verifyi ng the fulfillment of a predetermined condition. The condition is formed by an object of any t ype of language on the left, a logical operator and the right other object of language. If the con dition returns true value, the commands added in the context of the "if", ie they are correctly id entified, will be executed. Figure 32 presents the use of the command.

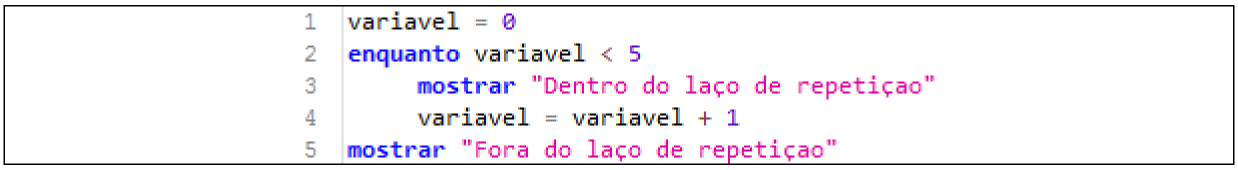


```
variavel = 5
se variavel == 5
     mostrar "Dentro do desvio condicional"
mostrar "Fora do desvio condicional"

conjunto = { 1, 2, 5}
se variavel eUmElementoDe conjunto
     mostrar "Dentro do desvio condicional"
mostrar "Fora do desvio condicional"
```

Figure 32. Conditional Deviation

3.2.1.7 Repetition lace

The “While” command was created to enable the execution of commands as a predetermined c ondition is true. The condition has the same formation as that of command “if”. If the conditio n returns true value the commands added in the context of the "while" will be executed and rep eated until the condition returns false. Figure 33 presents the command.



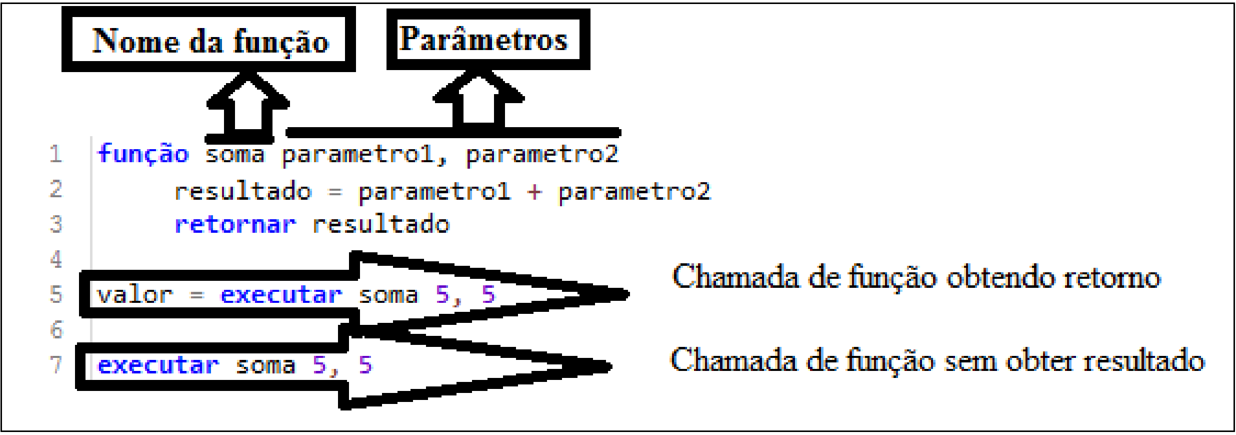
```
variavel = 0
enquanto variavel < 5
     mostrar "Dentro do laço de repetiçao"
     variavel = variavel + 1
mostrar "Fora do laço de repetiçao"
```

Figure 33. Repetition loop

3.2.1.8 User Function

User functions have been created so that it is possible to create codes excerpts outside the main context of the program and call them when necessary, avoiding code repetition and facilitating code reading (MARTIN, 2008). Language has incorporated this functionality and also enabled the creation of libraries with the functions created by the user through the Import Command, w hich will be presented in the “Import Command” section.

Figure 34 presents the declaration of function, called and obtains a result from it.


Nome da função
Parâmetros
Chamada de função obtendo retorno
Chamada de função sem obter resultado

```
1  função soma parametro1, parametro2
2       resultado = parametro1 + parametro2
3       retornar resultado
4
5  valor = executar soma 5, 5
6
7  executar soma 5, 5
```

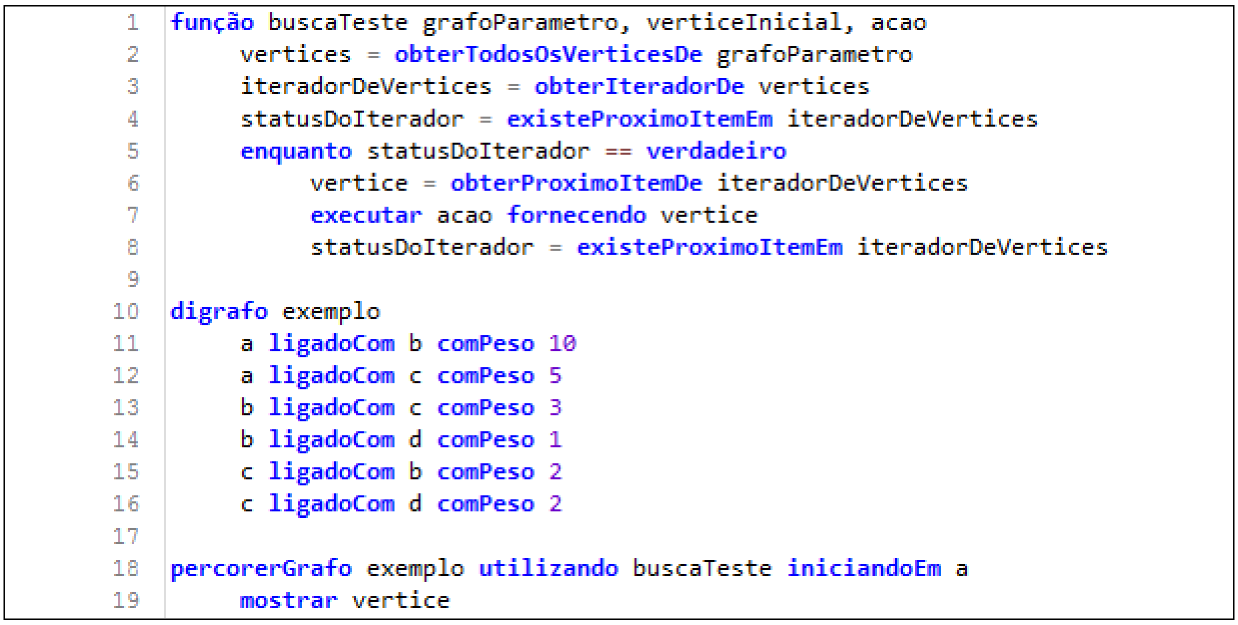
Figure 34. User Function

Figure 34 also presents the commands to return a value in a function and the function calls.

The Return Command serves that after running a function the programmer will be able to infor m the user the result of the execution of the user. The return instruction is presented at line thr ee of Figure 34. In lines five and seven of the same figure appear the calls of function. In addit ion to informing the name of the function the call passes the parameters used by separating the m by comma.

3.2.1.9 Train graph

The command was created to allow the separation between the algorithm that runs through the graph and what is done in each iteration. The command needs a function that will contain the s earch strategy to go through the graph, the graph that will be traveled and the vertex in which t he search will start. Figure 35 presents the use of functionality.



```
 1  função buscaTeste grafoParametro, verticeInicial, acao
 2       vertices = obterTodosOsVerticesDe grafoParametro
 3       iteradorDeVertices = obterIteradorDe vertices
 4       statusDoIterador = existeProximoItemEm iteradorDeVertices
 5       enquanto statusDoIterador == verdadeiro
 6            vertice = obterProximoItemDe iteradorDeVertices
 7            executar acao fornecendo vertice
 8            statusDoIterador = existeProximoItemEm iteradorDeVertices
 9
10  digrafo exemplo
11       a ligadoCom b comPeso 10
12       a ligadoCom c comPeso 5
13       b ligadoCom c comPeso 3
14       b ligadoCom d comPeso 1
15       c ligadoCom b comPeso 2
16       c ligadoCom d comPeso 2
17
18  percorerGrafo exemplo utilizando buscaTeste iniciandoEm a
19       mostrar vertice
```

Figure 35.

In the eighteen line of Figure 35 the command is called and the nineteen is added the show sho w in its context. The function called by this instruction must meet two requirements: have thre e parameters, and at some point call the instruction that performs the commands that are in the context of traveling graph. The instruction to execute these commands is in line seven, it recei ves the third parameter of the function and provides what the programmer considers necessary for the context of traveling graph, in the example of Figure 35 was the vertex variable.

The execution of the instruction travels graph and the called function occurs simultaneously. T he function receives as first parameter the graph informed by the instruction travels graph short ly after the first token, then the initial vertex that is informed right after the “starting” token, an d the third parameter receives instructions from the graphic execution context. In addition to t his flow of information the function can also inform data to the context of traveling graph, thro ugh the parameters informed after the “providing” token. Figure 36 illustrates this flow.

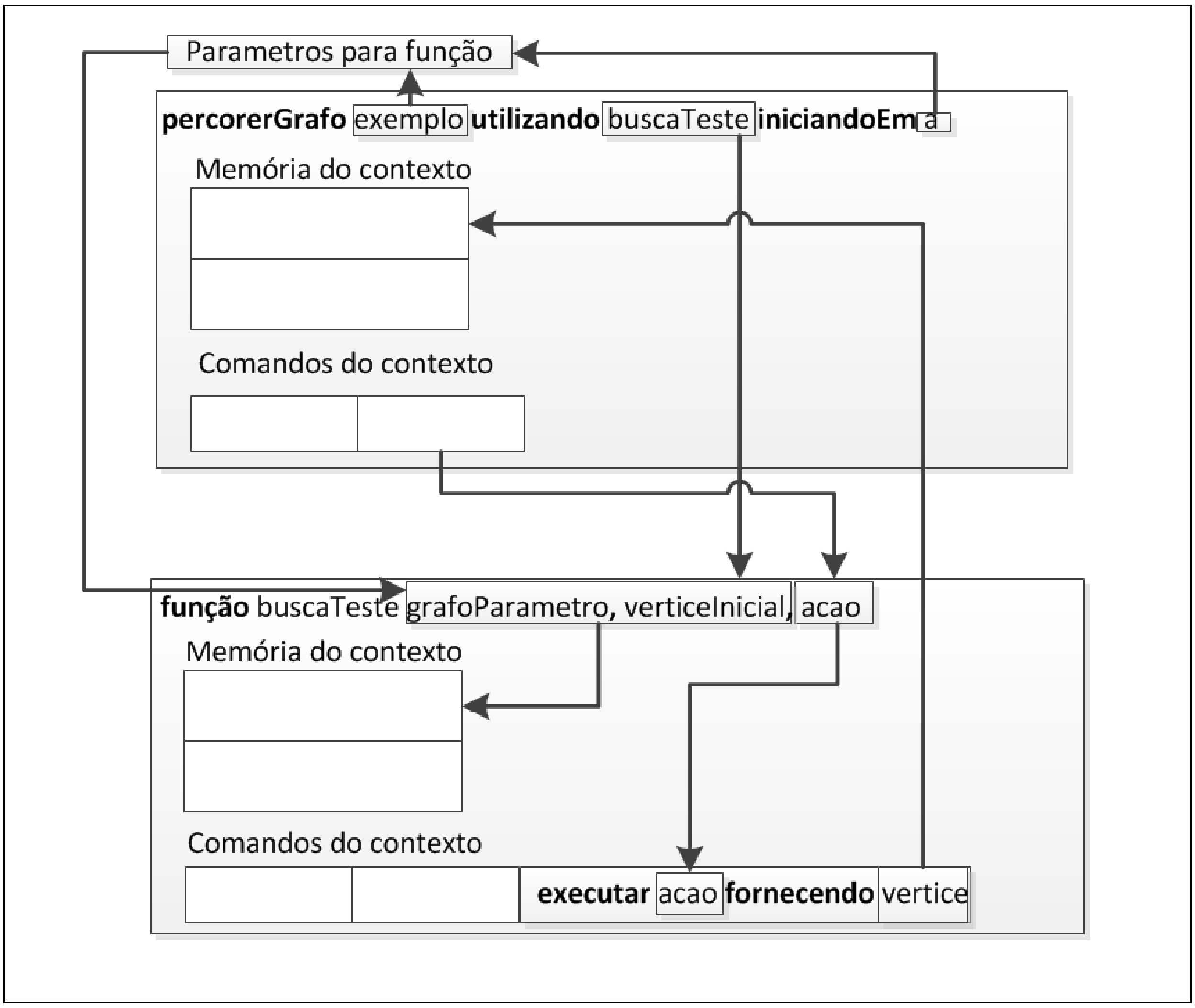
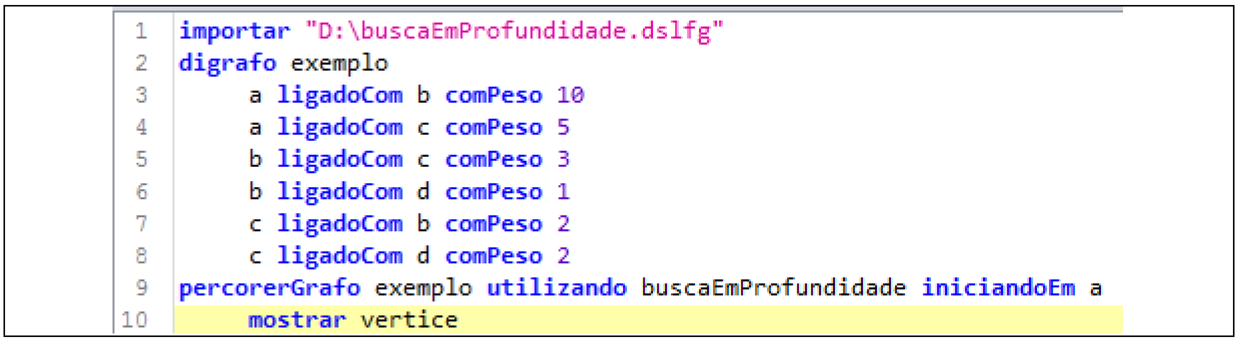

Parametros para função
percorerGrafo exemplo utilizando buscaTeste iniciandoEm a
Memória do contexto
Comandos do contexto
função buscaTeste grafoParametro, verticeInicial, acao
Memória do contexto
Comandos do contexto
executar acao fornecendo vertice

Figure 36. Instruction Flow Trapes Graph

3.2.1.10 Import command

This command allows the creation and import of libraries in language. From the path of the file that is passed to it, all functions existing in the file quoted are imported into the program under construction, if there are functions with the same name, the function of the current file and not imported will be considered. Figure 37 presents the use of the command.



```
importar "D:\buscaEmProfundidade.dslfg"
digrafo exemplo
     a ligadoCom b comPeso 10
     a ligadoCom c comPeso 5
     b ligadoCom c comPeso 3
     b ligadoCom d comPeso 1
     c ligadoCom b comPeso 2
     c ligadoCom d comPeso 2
percorerGrafo exemplo utilizando buscaEmProfundidade iniciandoEm a
     mostrar vertice
```

Figure 37. Import command

3.2.2 Types of Language

This subsection will present the types existing in language. Table 4 presents the types i n the “Type” column and describes its goal in the “Description” column.

Tipo	**Descrição**
Número	Trabalha com números e aceita casa decimal utilizando o separador “.”
Booleano	Aceita dois valores “verdadeiro” ou “falso”
Texto	Aceita qualquer texto digitado entre aspas duplas
Conjunto	Representa o conjunto da teoria de conjuntos, é reconhecido na linguagem através dos caracteres de abrir e fechar chaves
Grafo	Representa os grafos com ligações utilizando arestas
Dígrafo	Representa os grafos com ligações utilizando arcos

Table 4. Language Types

Language is dynamically typed, so when a variable is created, it is not necessary to inform the type, only assign the value. The types presented in Table 4 serve only to express the values ac cepted in the language.

Constant language have been implemented in language that aims to provide some user uses. Table 5 presents the constants created in the constant column and has the goal in the des cription column. Constants can normally be used as an object of the type number in arithmetic operations or the standard language functions.

Constante	Descrição
numeroMaximo	Contém o maior valor possível para o tipo Numero
numeroMinimo	Contém o menor valor possível para o tipo Numero

Table 5. Language constant

3.3 Implementation

This section describes how language has been implemented, presenting the standards a nd tools used.

The architecture was divided into three parts: user interface, interpretation and executio n. User Interface is the part responsible for obtaining user data and presenting answers from th e execution of the programs created, also has features to assist the user in the use of language. The part of the interpretation is responsible for describing the rules of grammar and transforma tion of the programs created by the user within the syntax defined in grammar into executable objects. Execution uses the objects created in the interpretation phase to finally execute the co de described by the user.

Figure 38 represents the interaction between the three levels of architecture.

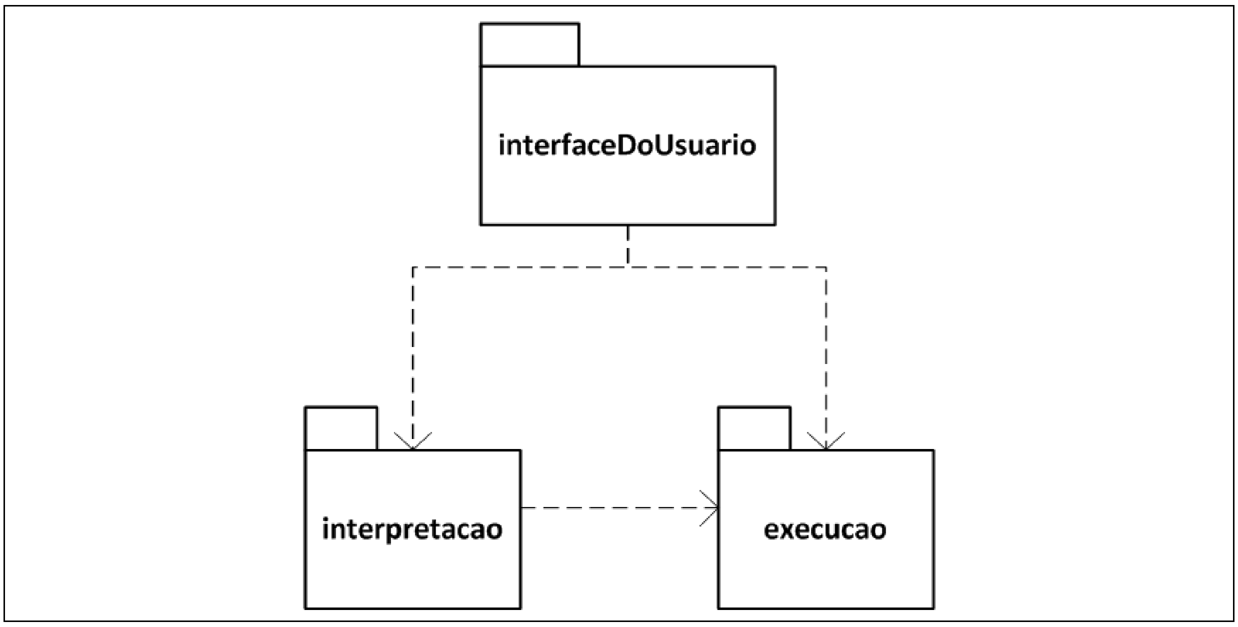


Figure 38. Interaction between architecture levels

3.3.1 Interpretation

The interpretation phase is responsible for transforming the user's code into a collection of executable objects. This is divided into three parts:

•Listener: The “listening to listening” is responsible for reading the grammar events and creating the builders. In addition, it is the interface between the “interfacedous uarium” and “interpretation” package; • Gramatic: Package that contains the rules o f grammatics produced in Antlr and the classes generated by him in Java; • Builder s: Package that contains all the (executable) primitive builders of language.

Figure 39 represents the interaction of the elements of the “interpretation” package.

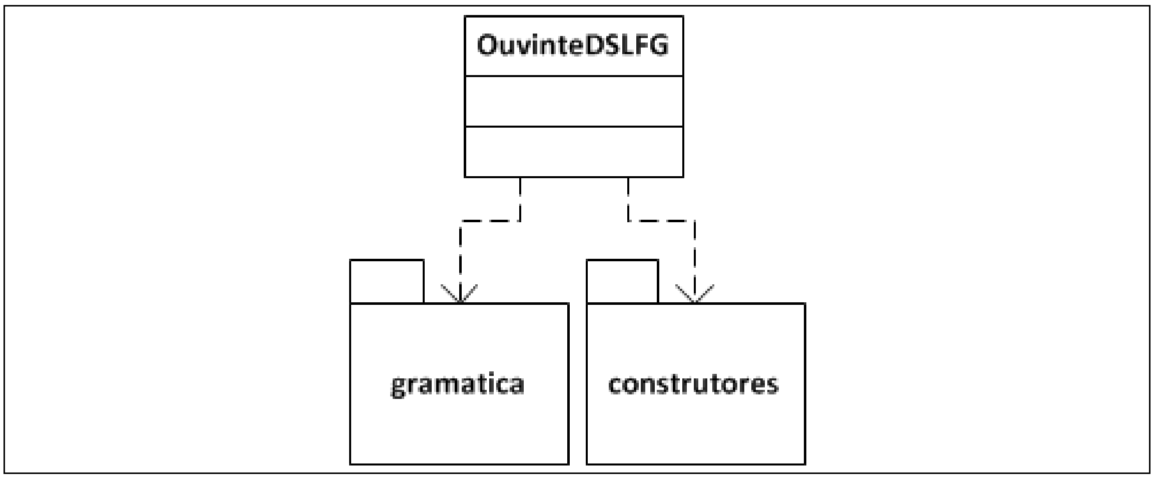


Figure 39. Interaction of the objects of the “Interpretation” package

3.3.1.1 Grammar

To implement a language, it is necessary to create an application that reads judgments a nd properly reacts to the phrases and input symbols discovered (PARR, 2012). The grammatic al package is responsible for the formation of language syntax and the transformation of rules c reated in the format of antlr into java classes.

Antlr is a tool used in the project that was developed by Terence Parr, a computer scien ce teacher at the University of San Francisco located in California, USA. The tool is a syntacti c analyzer generator and is widely used for the construction of languages. From a grammatics it generates an analyzer that can build an d walk in syntactic trees.

The grammar created within the antlr pattern is fully presented in Appendices A.3E A.4 . From these grammar files five Java classes are created among them 'DSLFGBaseListener' wh ich is responsible between the communication of AntlR classes and the classes created by the p rogrammer.

The 'listener' class extends 'dslfgbaselistener' and receives the events of the abstract syn tactic tree. Each recognized event is created a builder whose functionality will be presented in t he builders section.

3.3.1.2 Builders

The 'listener' class has methods that are called at the entrance and exit of the leaves of t he syntactic tree. And it is in these methods that builders are instantiated and then added to the command list.

The builders are enclosures of the commands of grammar. They take all the necessary i nformation from each command and instantiate an object with this information while performi ng the interpretation process. Each command has its own builder class, however, all command builders implement an interface that has the 'obtaining' method this method is responsible for c reating the primitive that will be performed at execution time.

In addition to the standard primitive builders that obtain all the necessary data from a li ne for subsequent creation of the executable there are the building builders. They implement a different interface and instead of the 'obtaining' method have the 'obtaining' method that is add ed within the primitive builder called 'AttributionPorexpression'.

Based on these two types of builders, all language instructions are charged in a list for l ater transformation into executable object. This list is in a special type of command builder tha t extends an abstract class responsible for adding the commands to its appropriate contexts.

An execution context builder has a list of commands and method to institute the executi
on context that will execute the commands on this list. Each instruction that has its own contex
t, such as the conditional deviation 'If', has its own builder. All the control that defines to whic
h context each command belongs is implemented within the abstract class' builderdcontextoxec
uto ′and is based on code identity.

The main execution context builder is in the 'listener' class and receives a new comman
d with each event launched by Gramatica. At the end of the interpretation process this context
builder instantiates the main execution context to then start the process of execution of the com
mands.

3.3.2 Execution

The execution phase begins shortly after the end of interpretation, when the listener fin
alizes its execution returning the execution context with executable commands. This section wi
ll present the main structures present in the execution phase and explain the operation.

3.3.2.1 Primitive

Primitive is an executable language command, they are primitive subcottal classes that
have the power to change the execution context or present messages to the user.

All primitives implement the 'executable' interface that has a method called 'execute' an
d receives as a parameter an execution context. Each grammar command has an equivalent pri
mitive and this implements the 'execute' method as needed by the command.

The primitive 'attributionporexpress' needs a second type of executable item as a compl
ement to its execution. This item is a class that implements the 'function' interface. This interfa
ce is implemented by all language functions and operators. It has its own method called 'obtain
ing resulted' and extending the 'executable' interface and therefore also has the 'executing' meth
od. In this way the functions when performed produce a result that are available through the m
ethod

‘Obtaining resulted’, in the case of 'attributionporexpression', the function or operation is resolv ed and then the result obtained and recorded in memory.

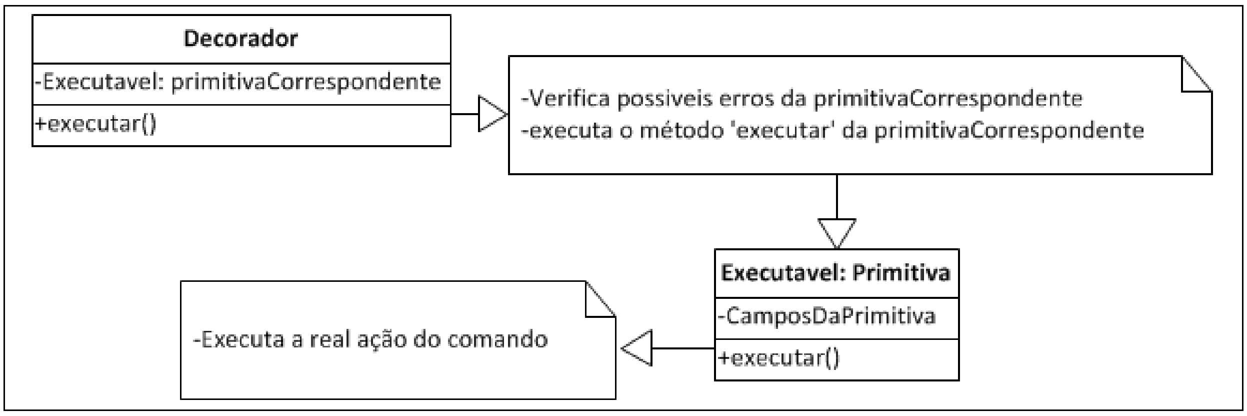
All operators cited in the Language Section Operators Subsection were implemented us ing the 'function' interface. Thus beyond the primitive 'attribution of theporexpress' the conditi ons verified in conditional deviations and repetition bonds also follow this pattern.

3.3.2.2 Error treatment

The treatment of primitive errors was based on the decorator design standard. With thi s pattern the treatment of error and the execution of the command are separated into two separa te classes, maintaining the code responsible for the concise execution, clean and easy to read. I mportant items for the maintenance phase of a system (MARTIN, 2008).

Each primitive in the system has a corresponding decorator. Every decorator follows a structure similar to primitive, implementing the executable interface, but instead of the primiti ve fields it has the corresponding primitive.

The 'executing' method of the decorator evaluates all possible errors in the primitive, as an example, the existence of the variable, the types involved and the existence of user function s are evaluated. At the end of the assessment process in case of absence of errors, the decorato r performs the execution method of his primitive with all the expected error possibilities, other wise an exception is released. Figure 40 presents the operation of the error detection process.


Decorador
-Executavel: primitivaCorrespondente
+executar()
-Verifica possiveis erros da primitivaCorrespondente
-executa o método 'executar' da primitivaCorrespondente
Executavel: Primitiva
-CamposDaPrimitiva
+executar()
-Executa a real ação do comando
Figure 40. Decorator Operation

3.3.2.3 Scope hierarchy

The context of execution is the environment responsible for scope control, where primit ives are kept and executed. It is responsible for the execution of the primitives through the imp lementation of the 'executable' interface and the recording or change of variables generated by the primitives.

For every command that requires tab, there is a context of execution. Although they ar e different all have several characteristics in common, as they are extended classes of the class 'contextodhecationaBstrate' that implements all common methods used by contexts, including methods responsible for adding variable, add variable in eviction, obtain variable and obtain fu nction.

In addition to the context methods of execution all the specializations of the abstract co ntext are also executable. The common feature of the execution of contexts is that they perfor m the list of commands they have. The particularity is in the form they perform. For example, the command is evaluated a condition before executing the commands, the command while eva luating the condition and repeats the instructions until the condition is false and the simple cont ext, containing of the initial context commands, simply performs the list of executables. Figure 41 presents the relationship between the classes mentioned.

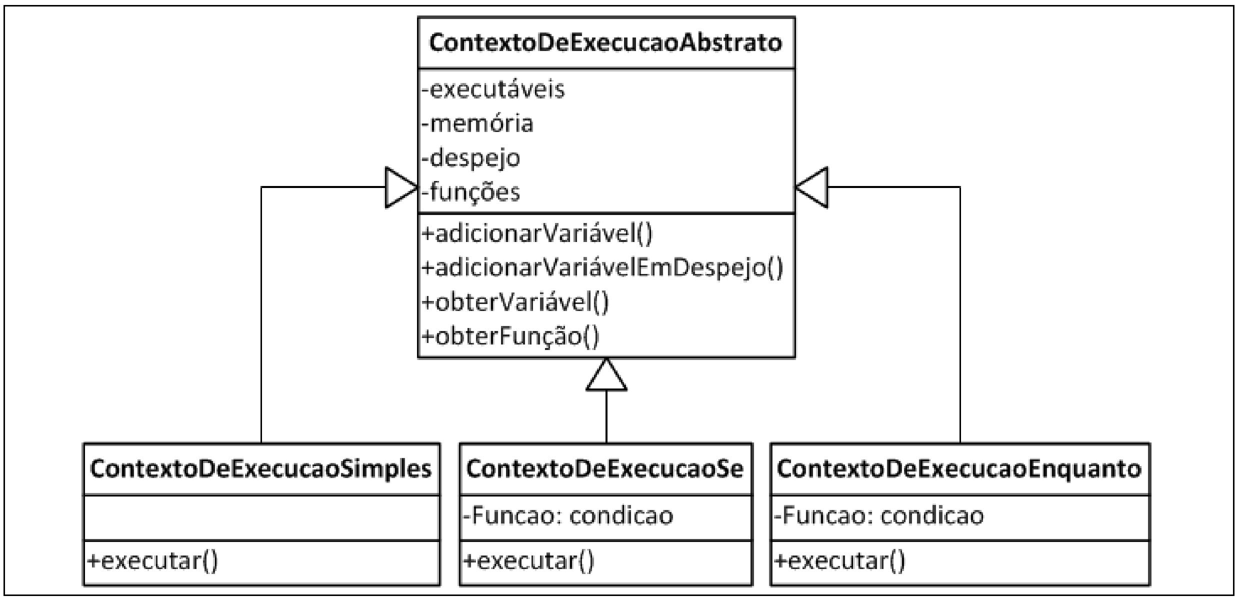
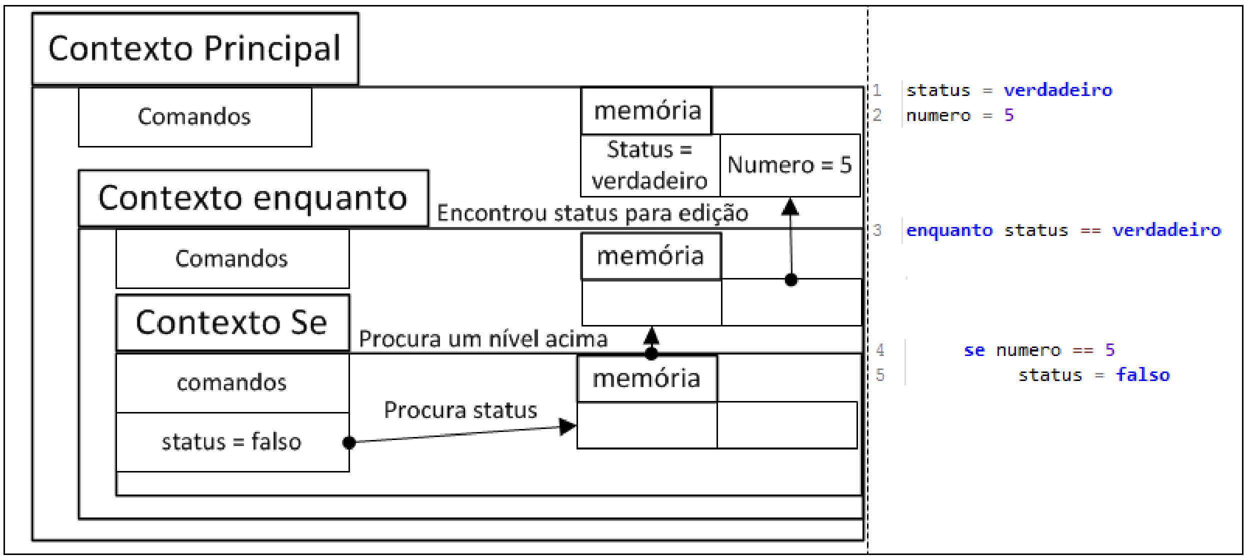


Figure 41. Relationship of execution contexts

With multiple execution contexts it is necessary to make a memory control that duly be have them. To make this control a map has been added in each execution context containing th e memory of that context. However, it was necessary

Make some controls to become possible to view and use variables in previous contexts. Figure 42 illustrates how the value of a variable is changed, presenting the right of the image the code that generates the new context of execution.



```
status = verdadeiro
numero = 5

enquanto status == verdadeiro

    se numero == 5
        status = falso
```

Figure 42. Variable Value Edition

The addition and search for variables uses the process illustrated in Figure 42. The valu e is sought on the memory map of the current context and iteratively will verify the previous c ontexts until it finds the variable sought or reaches the main context. The addition and change of a variable use the same method, when the process described above finds a variable with the given name executes the value change, otherwise it returns to the starting context and adds a ne w variable there.

User functions have a different memory precedence. When invoked by the primitive ca lled the function, they are caught from a function map and instantiated, receiving in memory th eir parameters and the main execution context as a previous context.

3.3.2.3.1 Context of execution of the instruction travels the graph

The instruction travels the graph has its own execution context, but its execution is para meterized and postponed to be performed only when the primitive 'executing percorrographer' is called.

The execution of this instruction has three key processes:

1. Context encapsulation in an object that also plays the context of executing and ex ecutable. Will be called action to simplify the explanation;

2. Preparation of instruction parameters;

3. Call of the function that contains the search strategy.

Figure 36 presents the parameters and how they are passed to the function.

Within the execution of the function, the action is stored in memory until the moment it is called by the primitive 'executing of the percorrographer'. At this point the action receives i n its memory the parameters passed by the instruction and is performed. Upon completion of t he process the function continues normally until its execution is completed.

3.3.2.4 Importing libraries

Library imports allows the use of functions defined in other files. The map of functions is obtained during the process of creating the main execution context, right after the interpretat ion phase.

During the interpretation phase a builder is created that has the path indicated for the fil e with the functions. In the creation of the main execution context the path is validated then th e interpretation of this file, obtaining the functions found and finally adding the functions in th e context in creation. From this point the functions are now available for use in primitive.

3.3.2.5 Tests

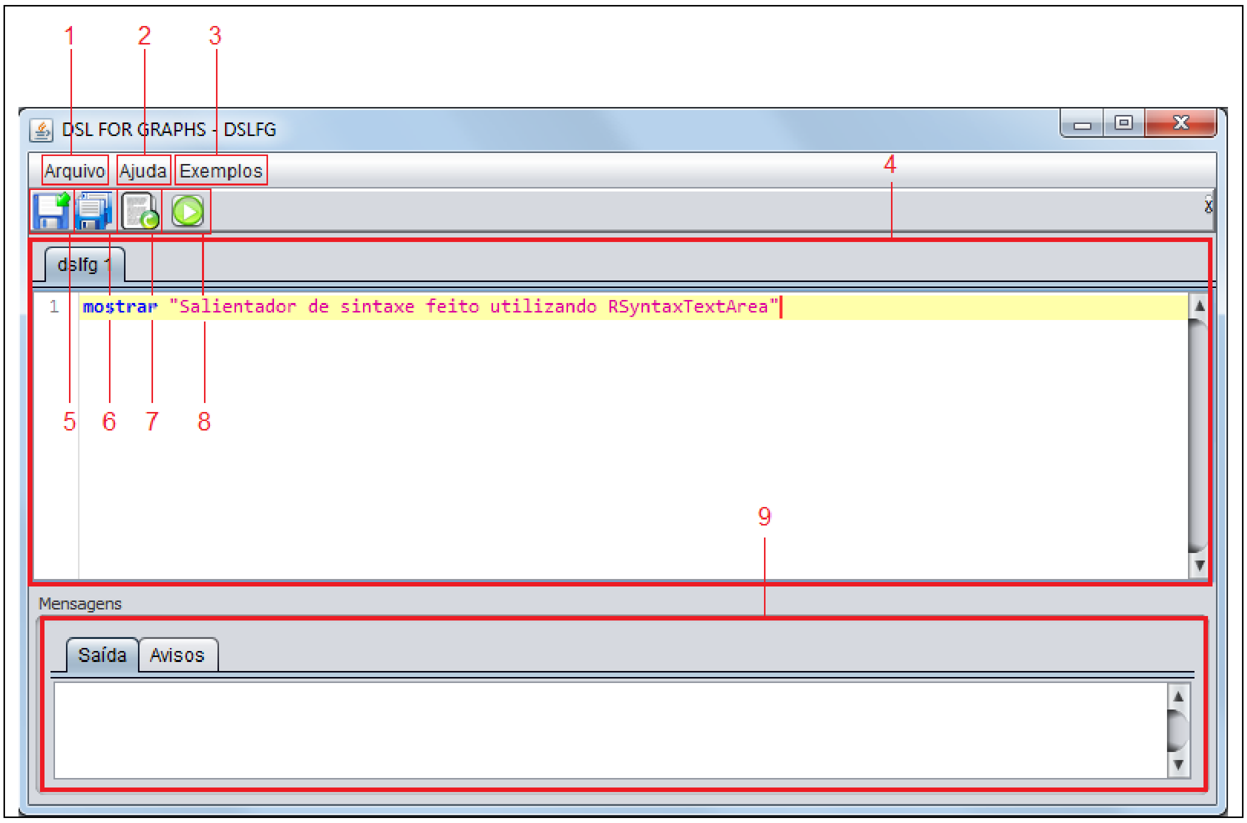
Unit tests were created for all language instructions. Their creation was made as the ne w instructions were being implemented, always followed by the execution of all previous tests, ensuring that new changes did not cause errors in the commands already implemented. The test s created are available in Appendix C.

The tests were created using Junit which is a framework used for the generation of auto mated tests. In addition to this framework, the primitive 'dump' that performs the copy of a vari able was created and adds to the eviction list. The class responsible for the execution of the test s uses Junit to compare the program's eviction list with a list that contains the expected evictio n of each test program.

3.3.3 IDE

IDE has been developed to enable the creation and execution of user programs and pro vide some features that facilitate this task. Through it that the output and input of language dat a occurs. This section will explain the purpose of the components on the IDE screen and prese nt the process of developing some features.

Figure 43 presents the IDE screen and overlaps the image with numbers connected to th e screen components, this figure will be used in conjunction with Table 6 to present the compo nents. This picture will present in the first column the corresponding component number in the image, and in the description column an explanation of the purpose of using the component.



```
mostrar "Salientador de sintaxe feito utilizando RSyntaxTextArea"
```

Figure 43. IDE Screen

Número	**Descrição**
1	Menu responsável por operações de arquivo como abrir, salvar.
2	Menu com informações sobre o desenvolvedor e lista dos comandos.
3	Menu com exemplos de programas na linguagem.
4	Painel tabulado destinado a entrada do código do usuário.
5	Salvar arquivo da aba atual.
6	Salvar os arquivos de todas as abas.
7	Abrir nova aba para entrada de código.
8	Executar código da aba atual.
9	Painel tabulado responsável pela apresentação das mensagens da linguagem para o usuário. Aba ‘Saída’ responsável pelas saídas do comando ‘mostrar’ e aba ‘Avisos’ responsável por apresentar os erros.

Table 6. Description of screen components

Error messages that are commented on item 9 of Table 6 are released in the treatment o f errors that was cited in section 3.3.2.2 Erro treatment. The exception launched by the instruct ion decorator comes to the screen where the message is obtained and presented to the user.

In the command that is written on the code panel, color differentiation between two ter ms can be observed. This feature is called the Syntax Substitute, from English *syntax highlight.* aims to highlight the most important terms of language, improving code visu alization.

The implementation of the syntax hosting was made using the rsyntaxtextarea component, avai lable for free and open source use. This component allows the definition of color through the predefined types such as Java or the definition of new patterns through the extension of the abs tracttokenmaker class, a method that was used for the implementation of the language runner.

In addition to graphic elements IDE provides a standard library that contains deep and width se arch methods. This library can be imported through the command and normally used, as explai ned in the library import section.

3.4 Experiments

In this section will be presented the experiments created to verify the potential of the la nguage to solve the proposed problems.

3.4.1 Example algorithms

Examples were created to present the use of language in real cases, ie applying to algorithms tr aditionally applied to graphs. The examples chosen then available in Appendix D.

Appendix D.1 presents the search in depth, an algorithm that aims to verify all the attai nable vertices of a graph through a search that whenever a vertex already chooses one of its adj acents to continue the search until there are no unlisted adjacent vertices, at this time returns th e path by checking other adjacent vertices.

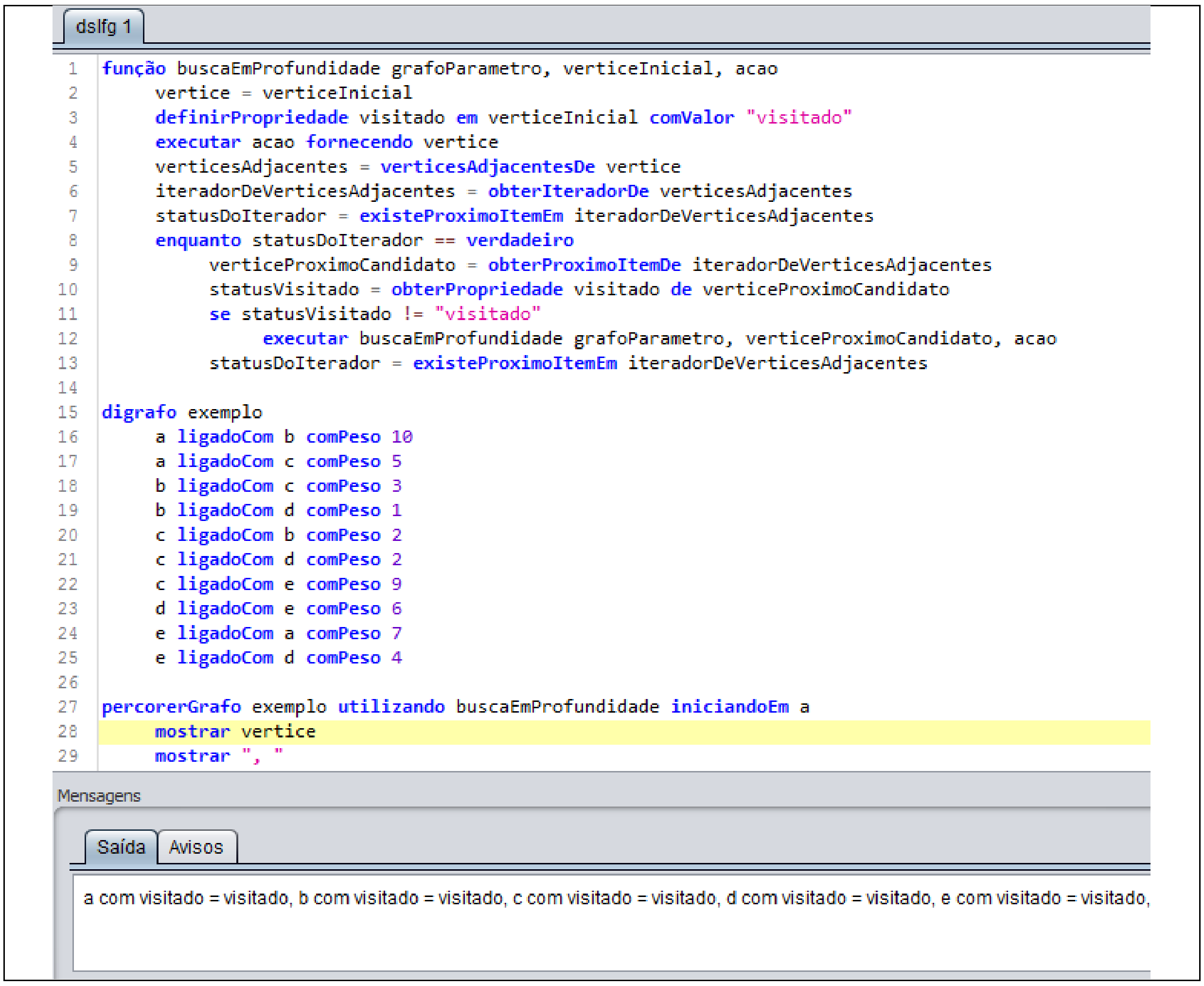
Appendix D.2 presents the width search. Unlike the search in depth, whenever this alg orithm reaches a vertex explores all the vertices of that level and sets up a set containing all adj acents of the next level to be

Then verified by repeating the process until all the attainable vertices of the graph have been v erified.

Appendix D.3 presents the implementation of the Dijkstra algorithm. The purpose of th is algorithm is to define the slightest path to all the vertices of the graph from an initial vertex. The data generated in the execution of this algorithm can provide the distance to go from the in itial vertex to a final vertex and the path traveled.

3.4.1.1 Execution of the depth search algorithm

Figure 44 presents the execution of an algorithm in the proposed language. The execut ed algorithm is the same present in Appendix D.1, responsible for the search in depth. In the “ DSLFG 1” tab is the input code typed by the user and the “output” tab the result after the progr am running. According to the logic of the algorithm specifies the code travels the graph "exam ple" and has it shows on the screen as passed in the vertices.



```
função buscaEmProfundidade grafoParametro, verticeInicial, acao
     vertice = verticeInicial
     definirPropriedade visitado em verticeInicial comValor "visitado"
     executar acao fornecendo vertice
     verticesAdjacentes = verticesAdjacentesDe vertice
     iteradorDeVerticesAdjacentes = obterIteradorDe verticesAdjacentes
     statusDoIterador = existeProximoItemEm iteradorDeVerticesAdjacentes
     enquanto statusDoIterador == verdadeiro
          verticeProximoCandidato = obterProximoItemDe iteradorDeVerticesAdjacentes
          statusVisitado = obterPropriedade visitado de verticeProximoCandidato
          se statusVisitado != "visitado"
               executar buscaEmProfundidade grafoParametro, verticeProximoCandidato, acao
          statusDoIterador = existeProximoItemEm iteradorDeVerticesAdjacentes

digrafo exemplo
     a ligadoCom b comPeso 10
     a ligadoCom c comPeso 5
     b ligadoCom c comPeso 3
     b ligadoCom d comPeso 1
     c ligadoCom b comPeso 2
     c ligadoCom d comPeso 2
     c ligadoCom e comPeso 9
     d ligadoCom e comPeso 6
     e ligadoCom a comPeso 7
     e ligadoCom d comPeso 4

percorerGrafo exemplo utilizando buscaEmProfundidade iniciandoEm a
     mostrar vertice
     mostrar ", "
```

```
a com visitado = visitado, b com visitado = visitado, c com visitado = visitado, d com visitado = visitado, e com visitado = visitado,
```

Figure 44. Algorithm executed in the IDE

3.4.2 Presentation in class

To evaluate the didactic potential, a presentation was scheduled for students of the Gra ph Discipline in the Computer Science course. The experiment was executed on June 17, 2014 with seven students in class and had the following presentation structure:

1. Language Presentation: This presentation had a theoretical introduction to DSL, demonstration of joint operations and by finishing the instructions and structure of l anguage.

2. Activity: An algorithm in the proposed language was presented with an error in t he logic and requested for the class that pointed out the error and solution. The goal was to make the language commands look carefully to be careful to answer questio ns regarding language.

3. Questionnaire: Questions were elaborated to evaluate the effectiveness of langua ge in the opinion of the evaluated. The questions and options are presented in Tabl e 7.

Pergunta	**Opções**
A lógica expressada pelos algoritmos apresentados ficou clara?	Sim, Não
Tendo como referência pseudo algoritmos já lidos referentes ao tema grafos. Você considera que seria mais fácil gerar um código executável na linguagem "dslfg", proposta na apresentação, ou em uma linguagem de propósito geral?	Dslfg, Linguagem de propósito geral (ex. java), Não tem certeza.
Que alterações você sugere para que a linguagem se torne uma boa ferramenta auxiliar na disciplina de grafos?	resposta descritiva para verificar as melhorias sugeridas por esses primeiros usuários.

Table 7. Questions of the Questionnaire

In the first question all evaluated marked the option 'Yes', indicating that all users consider that the logic of algorithms is clear in language. In the second question there were different answer s as the chart in Figure 45 presents.

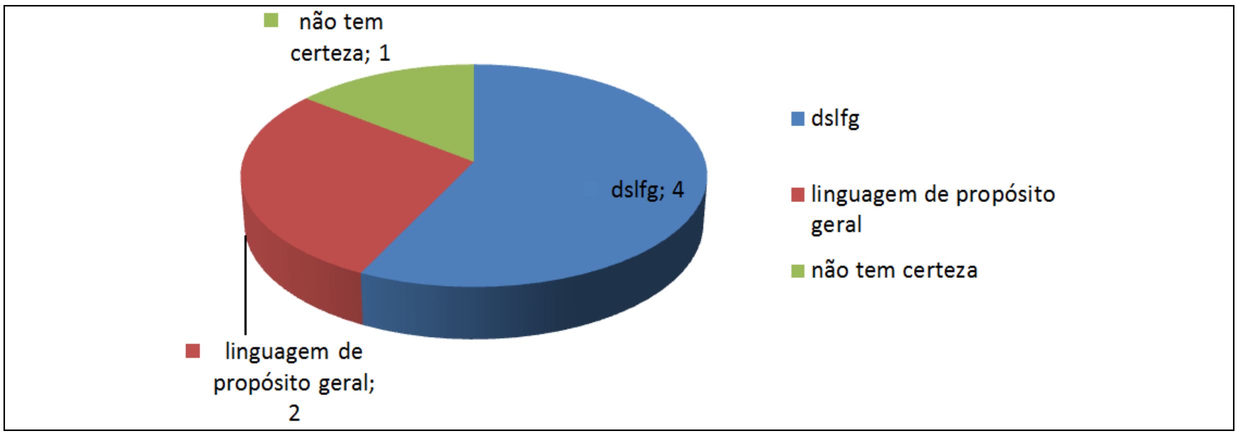


FIgura 45. Answers from the second question of the questionnaire

4 Conclusions

The objective of this work was the creation of a language that would enable the executi on of algorithms directed to graphs with a syntax that facilitates reading and understanding the logic contained in the code. For this it was necessary to study two specified areas, graphs to be able to project a language that meets the user needs and the area of compilers to enable the imp lementation of the project.

The graph study included a review regarding its structure and the assessment of algorith ms to better know the ways to solve the area's problems. The area of compilers had its study fo cused on learning the best practices in the development of a language, focusing on specific mas tery languages and knowing the methods and tools used to develop languages.

Antlr was used as a tool for defining the syntax and reception of syntactic tree events. At the beginning of the work, version four of the tool had recently been released, and although this at first made it difficult to acquire ideal material, it was chosen to continue with the update d version due to several improvements between version three and four of the tool, such as the way to describe grammar for example.

During development it was paid a lot of attention in the architecture of the software, try ing to create a code of easy reading and maintenance. For this, the concept of the agile method ology of iterative development was adopted, defining small objectives in each part of develop ment.

After completing the development of the main structures of architecture, packages were formed with the remaining instructions. About 70% spent on the project was taken for architect ure formation, the rest for the implementation of the instructions itself.

Finally, after building the tool, all instructions tests were performed to ensure that they worked correctly. With the positive result in the execution of the instructions, three algorithms were implemented within the language to ensure the operation with real graph algorithms. The results showed success to the class of algorithms in which it was tested and can be considered a positive alternative for the development of graph -oriented algorithms. In addition

Experiment conducted in the classroom demonstrated that language has potential as a didactic t ool in the graph discipline.

As future jobs were foreseen the following items:

•Flexibility of instructions: Flexibility would have changes such as allowing arithm etic expressions involving more than two elements at a time, multiple logical expres sions within conditional, among other extensions of this type; • Create instruction t o import and save ready -made graphs in 'graphml' language or other similar langua ge. The language cited is a marking language that was built to simply describe grap hs, not its algorithms (Graphml Team, 2013); • Create instruction to present the gra ph with more options, giving the possibility to show only the desired properties; • Create instructions to present graphically, possibly using ready -made frameworks s uch as ‘YED’ (YWORKS, 2014); • Develop language integration with other GPLs, enabling use in larger systems.

References

AMARAL, Daniela. Introduction to graph theory. [S.I]: Wikidot. Available at: < http://da nielamaral.wikidot.com/introduction-a-teoria-dos-braffes >. Accessed on: 10 nov. 2012.

BALAKRISHNAN, V. K. Schaum's Outline of theory and Problems of Graph Theory. New Y ork: McGraw-Hill, 2005.

Boaventura Netto, name. Graphs: theory, models, algorithms. 4.ed. Sao Paulo: Edgard B lücher, 2006.

DEURSEN, Arie Van; KLINT, Paul. Little Languages: Little Mainatenance? Amsterdam: *University of Amsterdam*, 1998. Available at: http://www.google.com.br/url?sable&rct =j&q =&esrc =s&source {v8 #v9}1&ved AA & URL = http%3A%2F%2FCITESEERX.ist.psu.edu %2fViewdoc%2FDownload%3FDOI%3D10.1. 1.13.8061%26REP%3DREP1%26TYPE%3D PDF & EI = M6Jeumckkoxe8WTLJIHAQ & USG = AFQJ CNH9AABLKPOUSQIGBBES1 OBYN-NUVW, Accessed: 23 Oct. 2012.

DIESTEL, Reinhard. Graph Theory. 3. Ed. Berlin: Springer-Verlag Heidelberg, 2005. Eu c.fc.ul. Connected. [S.I], [s.d.]. Available at: < http://www.educ.fc.ul.pt/icm/ICM2001/I CM33/Conexos.htm >. Accessed on: 03 nov. 2012. Fister Jr, Iztok; MERNIK, Marjan; B REST, Janes. Design and Implementation of Domain-Specific Language Easytime. [S.I.]: Elsevier, 2011.

FOWLER, Martin. Domain-Specific Languages. Westford: Addison-Wesley, 2010.

Graphml Team. The Graphml File Format. 2013. Available: < http:/ /graphml.graphdrawing.org/ >. Access on 06/06/2014.

GULWANI, Sumit; Tarachandani, Asha; GUPTA, Deepak; SANGHI, Dheeraj; BARRETO , Luciano Porto; MULLER, Gilles; Consel, Charles.In.: Webcal: The Domain Specific Lang uage for Web Caching. [S.I.], 2001. Available at: http://research.microsoft.com/en-us/people /sumitg/pubs/webcal_wcw00.pdf. Accessed on:
25 Oct. 2012.

HARTSFIELD, Nora; RINGEL, Gerhard. Pearls in Graph Theory. Boston: Academic Press, 1990.

Imme.usp. Graph theory. < year the information was generated. This year that goes in the quot e >. Available at: < http://www.ime.usp.br/ ~ pf/theorydesgraphs/text/theory/theorygrafs.pdf > . Accessed on: 14 nov. 2012

Klarlund, Nils; SCHWARTZBACH, Michael I. The Domain-Specific Language for Regular Sets of Strings and Trees. Article presented at Conference on Domain-Specific Languages (D SL). Saint Barbara: Usenix, 1997. Available at: http://www.google.com.br/url?sa =t&rct =j&q =&esrc =s&source =Web&cd&cd {v29.comdaqfj AA & URL = http%3A%2F%2FCITESER X.ist.psu.edu%2fViewdoc%2fdownload%3fdoi%3d10.1.

1.30.446%26REP%3DREP1%26TYPE%3DPDF & EI = 4Rdeuoummmmmm9GTT8OC4DA
& USG = AFQJ CNFXUX456B45WHJSQPTU-_RFH4IA. Accessed on 07 set. 2012. Klotz,
Walter. Graph Coloring Algorithms. Clausthal: University of Technology, 2002. Available at:
http://www.math.tu-clausthal.de/arbeitsgruppen/diskrete- optimierung/publications/2002/gca.
pdf. Accessed on: 01 nov. 2012. Kreher, Donald L.; STINSON, Douglas R. Combinarial Algo
rithms: Generation, Enumeration, and Search. Boca Raton, Fla: CRC Press, 1999. Krueger, C
harles W. Reuse software. AC Computing Surveys, v. 24, n. 2, Jun. 1992, p. 131-183. Availa
ble at: http://www.biglever.com/papers/krueger_acmreusesurvey.pdf. Accessed on: 15 Oct. 2
012. LOUDEN, Kenneth C. Principles and practices compilers. São Paulo, SP: Pioneer Thom
son Learning, 2004. Martin, Robert C. Clean Code the Handbook of Agile Software Craftsma
nship. BOSTON, MA: Pearson Education, INC, 2009. MARTINS, José Carlos Cordeiro. Soft
ware Project Management Techniques. Sao Paulo: Brasport, 2007. Parr, Terence. The definiti
ve Antlr 4 Reference. Dallas, Texas: The Pragmatic Bookshelf, 2012. Peter, Jandl Jr; ZUCHI
NI, Márcio Henrique. IC: A language interpreter. Projections Magazine. v. 19/20, jan./dez. 20
01/2002, p. 29-36. Available at: http://www.saofrancisco.edu.br/edusf/publicacoes/revistaProj
ecoes/volume_03/Uploaddre SS/Project%C3%A7Oes-6 [6374] .pdf. Accessed on: 02 Oct. 20
12. Reactivate-SEARCH. Click. Trento: Reactive Search, [s.d.]. Available at: < http://reactive
-earch.com/clique.php >. Accessed on: 13 nov. 2012. RICARTE, Ivan. Introduction to compil
ation. Rio de Janeiro: Campus, 2008. Rosen, Kenneth H. Discreet Mathematics and ITS Appli
cations. Boston: WCB/McGraw-Hil, 1999. Said, Ricardo. Programming logic course. Sao Pau
lo: Universe of Books, 2007. ScienceBlogs. Graph Contraction. < year the information was ge
nerated. This year that goes in the quote >. Available at: < http://scienceblogs.com/goodmath/
2007/07/08/graph- contraction-and-minors-1/>. Accessed on: 10 nov. 2012. Sebesta, Robert
W. Concepts of programming languages. Porto Alegre: Bookman, 2002. Shi, Hongchi; Gader
, Paul; Li, Hongzheng. Parallel Mesh Algorithms for Grid Graph Shortest Paths with Applicat
ion To Separab of Touching Chromosomes. Kluwer Academic Pubishers. Boston, 1998.

SZWARCFITER, Jayme Luiz. Graphs and computational algorithms. Rio de Janeiro: Campus, 1988.

Vasir. Dijkstra. Available at: <http://vasr.net/blog/game_development/dijkstras_algorithm_shortest_path/ l >. Accessed on: 11 Nov. 2012.

Wikipedia. Link. < year the information was generated. This year that goes in the quote >. Av ailable at: < http://en.wikipedia.org/wiki/laço_ (theory_dos_graphs) >. Accessed on: 14 nov. 2012.

YWORKS. Yed Graph Editor. 2014. Available: < http://www.yworks.com/en/pro ducts_yed_about.html/ >. Access on 06/06/2014.

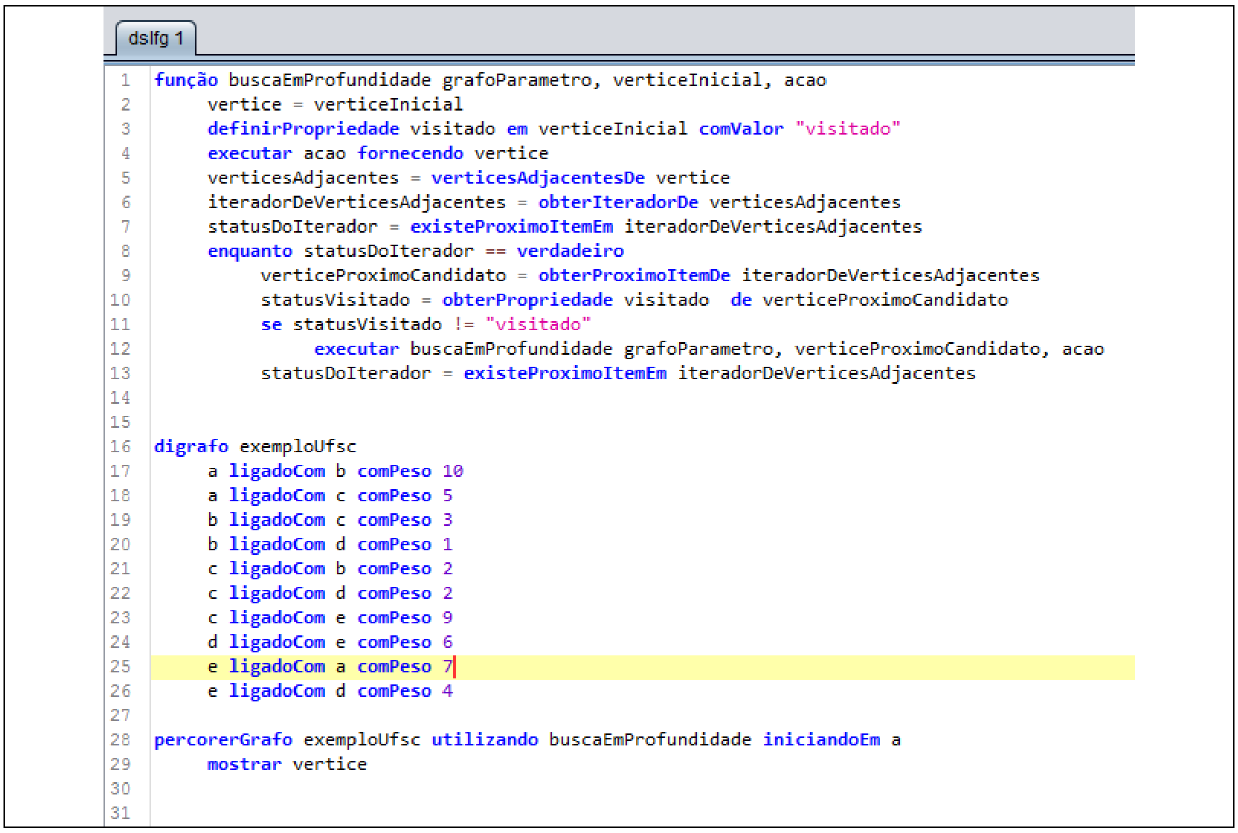
Appendix A. Examples of algorithms

A.1. Initial prototype

Import “Caminhodochivo” Show 3 Menoraresta = max Show graph graphoxemploumdol ivro navigators with population = 60000 with an effective = 300 ligadocom b Compenso 2 a ligadocom 3 b bigle d compound 2 b ligadocom 1 b Compete 3 C LigaCom and Com penso 2 DFs GRAFO, VERTICINAL, VERTICAL ACTION = Vertiiceinicial Verteices = {} While Vertices SubconjuntoPriode Add Verticeatual Vertiices Disease -Verticectual = Verticeatual VertiicesAdjacent to Vertione Vertion = Remove vertices from vertices an d verticeatual if vertices of the Different Verticee -Verticeatual {} verticeual = gets up ve rtices of thevertic

Run action by providing verticeual aresttual to obtain confrontation as parameter2 set2 = {9} as subset subset propriode parameter2 element = obtain parameter2 Add element by conjunction. the schoexampledlevro in starting navigators applying vert iceual aresting population = obtain population of navigators population = 5 + 5} 5 + 5 + 5 + 5 Show population

A.2. Depth search



```
função buscaEmProfundidade grafoParametro, verticeInicial, acao
     vertice = verticeInicial
     definirPropriedade visitado em verticeInicial comValor "visitado"
     executar acao fornecendo vertice
     verticesAdjacentes = verticesAdjacentesDe vertice
     iteradorDeVerticesAdjacentes = obterIteradorDe verticesAdjacentes
     statusDoIterador = existeProximoItemEm iteradorDeVerticesAdjacentes
     enquanto statusDoIterador == verdadeiro
          verticeProximoCandidato = obterProximoItemDe iteradorDeVerticesAdjacentes
          statusVisitado = obterPropriedade visitado  de verticeProximoCandidato
          se statusVisitado != "visitado"
               executar buscaEmProfundidade grafoParametro, verticeProximoCandidato, acao
          statusDoIterador = existeProximoItemEm iteradorDeVerticesAdjacentes


digrafo exemploUfsc
     a ligadoCom b comPeso 10
     a ligadoCom c comPeso 5
     b ligadoCom c comPeso 3
     b ligadoCom d comPeso 1
     c ligadoCom b comPeso 2
     c ligadoCom d comPeso 2
     c ligadoCom e comPeso 9
     d ligadoCom e comPeso 6
     e ligadoCom a comPeso 7
     e ligadoCom d comPeso 4

percorerGrafo exemploUfsc utilizando buscaEmProfundidade iniciandoEm a
     mostrar vertice


```

Figure 46. Deep search algorithm in DSLFG language

Appendix B. Grammar

A.3. Dslfglexico.g4

Lexer Grammar dslfglexico; *TIPOS_BOOLEANOS*: 'tru
e' | 'false';

CONSTANTES: 'numeromaximo' | " numeromiminity '; *OPERADOR_ARITMETICO*: '+' | '-' | '*' | '/' | '
div' | 'mod'; *OPERADOR_CONDICIONAL_DE_CONJUNTO*: 'Subconjunctoproprode' | 'Sub -employment'

|
'Eumelemental' | " noeumelemental '; *OPERADOR_CONDICIONAL*: '==' | '<=' | '}'} '{v22 <' '}'
{v25 | '! ='; *NUMERO*: *DIGITO*+ ('.' *DIGITO*+)?; *STRING*: (" '|' ') (' '|'! '| [\ u0001- \ u0021] | [
\ u0023- \ u30ff] |')* (" " " " '); *DIGITO*: [0-9];

PALAVRA: ([A-z] | [a-z] | [to -ü] | [\ u0168- \ u0169] | '|' | '|' {v47 _ ') ([A-z] | [a- z] | [at -ü] | [\ u
0168- \ u0169] | '|' | '|' | '|*DIGITO*)*;
TAB: '\ t'; *QUEBRA_DE_LINHA*: '\ r' | '\ n' | '\ r' '\ n';
COMENTARIO: '//*'.*? *QUEBRA_DE_LINHA* -> skip; *WS*: (" | '\ f'
) -> skip;

A.4. Dslfg.g4

Grammar DSLFG; Import
DSLFglexico;
start :

import* code*; Code: *QUEBRA_DE_LINHA* | Constr
ucaodegraphic | ConstrucaDedipigraph | funches |
declareconjunct | attributexpress | Add | define | w
hile | | function | call | Return | functionpergorgh
er | execute performer

; If: *TAB** 'if' (conditional of the confirm | conditional) *QUEBRA_DE_LINHA*; While: *TAB** 'whil
e' (conditional of the monitor | conditional) *QUEBRA_DE_LINHA*; Conditional Deconjunct:
PALAVRA OPERADOR_CONDICIONAL_DE_CONJUNTO PALAVRA

; Conditional: multitypes *OPERADOR_CONDICIONAL* multitypes; Import: 'Import' *STRING*; Cons
trucaDegraphy: *TAB** 'graph' *PALAVRA QUEBRA_DE_LINHA* connectovertice*; ConstrucaDeDigrap
h: *TAB** 'Diph' *PALAVRA QUEBRA_DE_LINHA* LigaTertice*; LigaDeverte: *TAB+* DeclaodaTertice '
LigaCom' Declaodevert

Ice

Pesodaaesta? *QUEBRA_DE_LINHA*;

DECLARACAOVERTICE: *PALAVRA* (*QUEBRA_DE_LINHA* Adicaveprordination* *TAB*+)? ; Ad
icodeprordance: *TAB*+ 'with' *PALAVRA* '=' multitipos *QUEBRA_DE_LINHA*; Pesodaresta: 'Compensation'
NUMERO; FUNCTION: *TAB** SHOW 'Multitipos *QUEBRA_DE_LINHA*; Declaration: *TAB** *PALAVRA* '=' set
QUEBRA_DE_LINHA; AttributionPorexpress: *TAB** *PALAVRA* '=' functions *QUEBRA_DE_LINHA*; Set: '{' ((
NUMERO ',')* *NUMERO*)? '}'; Add: *TAB** 'Add' multitipos 'in' *PALAVRA* *QUEBRA_DE_LINHA*; Deferpropritione
r: *TAB** 'Defineproprhendise' *PALAVRA* 'in' *PALAVRA* 'multitypes *QUEBRA_DE_LINHA*; Funcoes: Obtain Vert
icesAdjacent | Obtain Upon | Obtain avertice | Remove | union | Intersection | Difference | call | obtain
|} | Obtain proprietary | Obtainer | There is proximoitem | obtain proximoitem | obtertodertices | obtains
aresar | Copiarrographers | Expressaaritmetics; EXPRESSIONAARYMETICS: Multitypes (
OPERADOR_ARITMETICO multitypes)? ; Remove: *PALAVRA* 'remove' *PALAVRA*; Union:
PALAVRA 'uniting' *PALAVRA*; I~~ntersectio~~n: *PALAVRA* 'Intersection' *PALAVRA*; Difference: *PALAVRA* 'Differe
nce' *PALAVRA*; Calling: *TAB** 'Run' *PALAVRA* parameters; COPIARFRAPHE: 'Copiarrographers' *PALAVRA*;

Obtainsvertices: 'obtains or parts for' *PALAVRA*; ObtainsAsar: 'ObtertodaSaSaestasDe' *PALAVRA* ; Obtaining: 'Obtaining' *PALAVRA*; There is a proximonm: 'there is a proximoitemem' *PALAVRA*; Obtain proximonm: 'obtain proximonmde' *PALAVRA*; obtain proprietary: 'obtain proprietary' *PALAVRA* 'of' *PALAVRA*; Obtain apestaar: 'Get PetaA -Range' 'from' *PALAVRA* 'to' *PALAVRA*; obt ain vertices. Obtain Up: 'Obtain Upon Feature' *PALAVRA*; Obtain avertice: 'Obtain averticide' *PALAVRA*; Obtainvertice: 'Obtainverte' *PALAVRA* 'of' *PALAVRA*; Multitiple: *NUMERO* | *PALAVRA* | *STRING* | *CONSTANTES* | *TIPOS_BOOLEANOS*; Pour: *TAB** 'pour' *PALAVRA*; FUNCTION: 'F unction' *PALAVRA* parameters; Parameters: (*PALAVRA* ','?)*; Return: *TAB+* 'return' *PALAVRA*; F UNCAOOPERCERERGRAM: *TAB** 'Percorergher' *PALAVRA* 'Using' *PALAVRA* 'starting' *PALAVRA*; Performing Performance: *TAB** 'execute' *PALAVRA* 'providing' pa

rameters

;

Appendix C. Testing Programs

C.1. Simple set

```
xpto = {1.5,2,3,4,5} Show x
pto
```

C.2. Simple set 2

```
xpto = {} show xpto xpto =
{1.5,2,3,4,5} Show xpto Ad
d 8 in xpto show xpto
```

C.3. Command while

```
xpto = {1,2,3,4,5} xpto2 = {1} Show xpt2 while xpto2 subsj
untropropriode xpto new = get xpto add new xpto2 show xpt
o2
```

C.4. Command while with

```
xpto = {1,2,3,4,5} Show xpto xpto2 = {1} show xpto2 xpto3 = {1,2,3
} If xpto3 subsjuntopropriode show xpto3 add 13 in xpto3 xpto4 = {8
,9,10} xpto5 = xpto2 Subconjunctoproprode xpto xpto5 = {} new =}
XPto Add new xpto2 show xpto2
```

C.5. Command

xpto = {1,2,3,4,5} Show xpto xpt2 = {4,5} XP
TO2 If xpto2 Subconjuntopropriode XPTO Ad
d 8 in xpto show xpto

C.6. Command if 2

xpto = {1,2,3,4,5} Show xpto xpt2 = {4,5} sho
w xpto2 subset2 subsjuntoproprode xpto3 = {6,
7,8} Add 8 in xpto show xpto3 show xpto

C.7. Context of recursive function

Test Function Numeroparameter Number = Nu
eroparater Show number if number < 6 number
= number + 1 Run test number show number

TEST CONTRESS FUNCTION NUMBERMAMERMA
NCER NUMBER = NUMBERMAMETER SET = {} Ad
d Number Show Set If Number < 4 Number = Number + 1
Run Testconjunct Number Show Set

Number1 = 0 Perform Test Conjunct Num
ber1 Run Test Number1

C.8. Context test

Test Function2 Show y Pour y
Test Function x = {2, 3} y = 5
Run test2 return x x = {0, 1} R
un test pour x show x

C.9. Context Test 2

Test function x = {2, 3}
Run test pour x show x

C.10. Obtain a whole element

xpto = {1.5,2,3,4,5} xpto2 = obtains xpto
show xpto2

C.11. Operator removing

Set Prince = {1, 2, 3, 4, 5} Show setting set = {1, 5, 3} set.

C.12. Constant

Grafoexemploumdolivro Navigantes Ligadocom Itajai It
ajai Ligadocom Balneariocamboriu Itajai Ligadocom Br
usque

Vertice = Obtain Raventexes of GrafoexemPlouMDolivro Define Propress Valoroximo
Vertice with numeromaximo Valormones Vertex

ValorMinimo = obtain valormony for vertice valorximo = obtain valuxim
o vertice

Pouring valuximo dump val
ormony

Show Valormaximo Show
Valormones to Show "End
"

C.13. Define vertex property

Grafoexemploumdolivro Navigantes Ligadocom Itajai It
ajai Ligadocom Balneariocamboriu Itajai Ligadocom Br
usque

Vertice = Obk graphoxemploumdolivro navigators Show graphoexemploumdoli
vro show vertice define vertice value with Vertice Vertice Show Vertice Show g
rafoexemploumdolivro show "end"

C.14. Pour

Set Princepal = {1, 2, 3, 4, 5} Pouring set.

C.15. Copy graph function

Grafooriginal graph Navigators with population = 2 ligadocom Itajai wit
h population = 4 Compenso 2 Itajai LigaCom Balneariocamboi Compou
nd 3.5 Itajai Ligadocom Comprises 5 Graphocopy = graphoginal graphig
inal graphiginal

Show "grafooriginal" show grap
horiginal

verticeoriginal = obtain vertex graphoginal navigators Deforprise VertiCeiginal
Value

Show "grafooriginal" show grap
horiginal

Show "Graphopia" Show Gr
aphocopy

C.16. User function

Show "out of the function"

Function Writing Contents1, Content2 If content1 subsjuntoPriode Conteudo2 Show
"within the if that is within the function" show "within the" Set Office = {1,2,3,4}

Show "Out of Function of New" Set1 = {1, 3
} Set2 = {1, 2}

Execute Write Set1, Set2

C.17. User function with return

Set = Execute obtain conjunct Show Set Set S
et Set

Set of Parameter1 = {6.7} Set of Parameter2 = {6.7,8} Set2 = Perform confunto and set meter1, set2 Sh
ow Set2 Set 2

Set of Parameter1 = {9,10,11} Set3 = Run any set.

show "end"

Obtain Conjunct Function ConjuntoRen = {1,2
,3,4} Return sets

Function Obtain Conjuncts Parameter1, parameter2 If parameter1 subsjuntoPriode Parame
ter2 Set Office = {6,7,8} Return Set Set Show "should not have shown this in: Obtain Con
juncts"

```
f
  Anointing Announcement as Parameter2
            SetReteno = {9}
            While Setor Breeding Sub -Conjunct Proprode Parameter2
            element = Obtain Upon Parameter2
            Add element together
            Return ConjuntoRetron
            Show "I shouldn't have shown this in: obtaining anytime"
```

C.18. Function obtain weight of the edge

Grafoexemploumdolivro navigators ligadocom Itajai Compilae 1 Itaja
i Ligadocom Balneariocambori Comprese 2 Itajai Ligadocom brusque
Compensus 3

Vertice = Obtain GrafoexemPloumDolivro Vertice2 = Navigators Itajai of grafo
exemploumdolivro pesodaaesta = Vertece to vertice2 pouring pouring pouring t
o show "end"

C.19. Function Obtain Property

Grafoexemploumdolivro Navigantes Ligadocom Itajai It
ajai Ligadocom Balneariocamboriu Itajai Ligadocom Br
usque

Vertice = Obtain RaisexExPloumDolivro Propritioner Navigators Vertice Value
5 Show "Vertice:" Show Vertice Show "Value:" Value = Vertice Value Show va
lue Pour value show "End"

C.20. Function Obtain Vertex

Grafoexemploumdolivro Navigantes Ligadocom Itajai It
ajai Ligadocom Balneariocamboriu Itajai Ligadocom Br
usque

Vertice = Get vertex graphoxexamplemdolivro dump vertice show vertice show
"end"

C.21. GRAFO FUNCTION

Graf Test Function, Init, Action Set1 = {1,2, 1000} Set2 = {1} While Subco
njuntoPriode Set1 Verticerector = GAF Running Action Providing Verticere
ctor Element = Obtain Set Element

Function BusinessTest Grafoteste, Verticeinial, Accean Return = Run Grafoteste, Verti
ceinial, Acao Test

Grafoexemploumdolivro Navigantes Ligadocom Itajai It
ajai Ligadocom Balneariocamboriu Itajai Ligadocom Br
usque

Percorergher GrafoexemploumDolivro Using Searches Starting In Starting In Navigators Show VerticeRetron
Pour Verticeletan Show "End"

C.22. Iterator functions

Set = {1.2} iterator = Obteritator Status set = there i
s iterator Show Status Status Status Value = obtain
iterator Show Value

Status = There is proximoitemem iterator Show Sta
tus Value = Get ProximoitemDe Iterator Show Val
ue

Status2 = There is proximoitemem iterator Show stat
us2 Pour status2

Test Function Set ESCON, Itener Conjunctode = ObteritradeDequerd Status = There is a trader if stat
us == True element = Obtain sets combined set in conjunctioned set = set. conjunctodiree show eleme
nt

f

```
 ENTRACTO2 ENJOINE
           element = Obtain Up Set Set
           Show "element."
           show element
           Add element in conjunctodireita
           conjunction is subsidiary
           Perform joint, conjuncto -conjunction test
           Show "elementodopois"
           show element
```

SET1 = {0,1,2,3} Set2 = {} Set3 = {1,2,3,4} Set4
= {} Run 2 Set3, Set4 Show "End"

C.23. Functions obtain all vertices and Edge

GrafoexemploumDolivro Navigators LigadoCom Itajai Itajai Ligadocom Ba
lneariocamboriu Itajai LigaCom Vertices = Obtertdosvertices of Grafoexem
Dolivro Arestes = obtertodasrests from graphoxemploumdolivro show "Vert
ices" show vertices show vertices "edges" show edges dump vertices pour e
dges

C.24. Vertice

graph grafoexemploumdolivro navigators ligadocom itajai comphesus 2
Itajai ligadocom Balneariocamboriu 3.5 Itajai ligadocom brusque compu
te 5 xpto = obtain graphoexemploumdolivro show xpto

C.25. Obtain vertices adjacent

graph graphoxample navigators ligadocom Itajai Compensus 2 Itajai Lig
adocom Balneariocamboriu Compensus 3.5 Itajai Ligadocom Brusque C
ompulso 5

Show graphoexample vertice = obtain a vertexexample verticesadja
cent = vertices of vertice show vertices show vertices.

C.26. Show command

Set = {1,2,3,4,5,6,7,8,9} Show an element in the course = obtain
a set of show 20 show "end"

C.27. Nonexistent property

Example Diphec The Ligadocom B Comp
ensus 10 A Ligadocom C Compensus 5 B
Ligadocom C Compensated 3 B Ligadoco
m D Compensus 1 C Ligadocom B Compe
nso 2 C

Vertice = obtain averticede exemploufsc value = obtain visited
vertice show Value Pour value

C.28. Graph with weight

graph graphoxemploumdolivro navigators ligadocom Itajai Compeso 2 I
tajai ligadocom Balneariocamboriu 3.5 Itajai Ligadocom brusque Compe
nsus 5

dump grafoexemploumdolivro show grafo
exemploumdolivro

C.29. Full graph

GrafoexemploumDolivro Navigators with Population = 2 ligadoCom Itaj
ai with population = 4 Compensus 2 Itajai LigaCom Balneariocamboriu
3.5 Itajai LigaCom BRUSQUE Compulsion

C.30. Simple graph

Grafoexemploumdolivro Navigantes Ligadocom Itajai It
ajai Ligadocom Balneariocamboriu Itajai Ligadocom Br
usque

Diphe dijiExamploumdolivro navigators ligadoscom itaj
ai Itajai ligadoCom Balneariocamboriu Itajai brusque sh
ow "graph" show grafoexemploumdolivro show "dipo"
dipo and dick dick grafoexemploumdolivro diggaexamp
loumdolivro

C.31. Set differences

set1 = {0, 3} set2 = {0, 1} set = set1 Difference Set2 Sh
ow Set Set Set

C.32. Div

Seven = 7 three = 3 x =
Seven div to show x pou
r x show "end"

C.33. DIVISION

five = 5 four = 4 x = five /
four show x pour x show
"end"

C.34. Operator is an element of

xpto = {1,2,3,4,5} number = 5 x = false
y = fake if number eumelectode xpto n
umber = 6 x = if xpt2 eumementode xp
to y = true show x pouring y pouring y

C.35. Intersection Operator

set1 = {0} set2 = {0, 1} set = Set1 Intersection Set2 Show
Set Set Set

C.36. Larger operator

A = 0 B = 0 C = 0 If 6 > 5 a =
1 show "right"

If 5 > 5 b = 1 show "wrong"

If 4 > 5 c = 1 show "wrong" po
ur pating b pour

C.37. Operator greater or equal

A = 0 b = 0 c = 0 If 6 >= 5 a =
1 show "right"

If 5 >= 5 b = 1 show "right"

If 4 >= 5 c = 1 show "wrong"

Pour to pour B
pour

C.38. Smaller

A = 0 b = 0 c = 0 If 6 < 5 a = 1
show "wrong"

If 5 < 5 b = 1 show "wrong"

If 4 < 5 c = 1 show "right" pou
r pour b pour c

C.39. Smaller or equal operator

```
A = 0 B = 0 C = 0 If 6 <= 5 a =
1 show "wrong"



If 5 <= 5 b = 1 show "right"


If 4 <= 5 c = 1 show "right" du
mp to pour b pour
```

C.40. Mod

```
seven = 7 three = 3 x =
Seven mod three show x
pour x show "end"
```

C.41. MULTIPLICATION

```
five = 5 four = 4 x = five
* four show x pour x sho
w "end"
```

C.42. Operator is not an element of

```
xpto = {1,2,3,4,5} number = 5 x = false y =
false if number noeumelement xpto x = Tru
e number = 6 If xpto2 noeumelement xpto
y = True show x pour x pouring y
```

C.43. Different operator

Set1 = {1.2} set2 = {1.2} set1 = {3.4} set2
= {} If set1! ! = set2 Set 2 = {3.4} Show "w
orked" dumping Set Set1 Pouring Set 2 Test
= "Visited" If Test!

C.44. Equal operator

Set1 = {1.2} set2 = {1.2} set1 = {} set2 = {
3.4} If set1 == set2 set1 = {3.4} worked " If
set1 == set2 Set Set2 = {5.6} Show "went w
rong" dumping Set Rousing Set Set2 Show
"End"

C.45. Simple sum and assignment

five = 5 four = 4 x = five
+ four show x pour x sho
w "end"

C.46. Subset

```
xpto = {1,2,3,4,5} Show xpto xpto
2 = {1,2,3,4,5} show xpto2 x = fals
e y {fake if xpto2 subset xpto x = a
dd 8 in xpto2 show xpto2 if xpto2 s
ubset xpto y = pour
```

C.47. SUBTRACTION

```
five = 5 four = 4 x = five - four show x pou
r x phrase = "Remove word from here" wor
d = "word" result = phrase - word show res
ult pour out result show "end" "
```

C.48. Union Operator

```
Set1 = {0} Set2 = {0, 1} set = set1 uniting set2 Sho
w Set Set Set
```

APPENDIX D. Example Programs

D.1. Depth search

Function Search for graphoparameter, verticeinial, vertice = Verticeinicial Definitial Vertiiceinicial Comvaluatio n "Visited" Turning VertiCesadjacents = Vertices of vertexevertesadjacents = Obteritisar from verticesadjacent t o statusdoite. = There is proximoitemem iterator forverticesDJacecents while statusdoiters == True VertiCeproxi moCando = Obtain proximoitemde in iterator of the statusvisted status = Visited -to -Vertiocurate Properity If sta tusvised! = "Visited" graphoparameter, verteceproximoCando, action statusdoiter = there is a proximoitemem ite rator forverticesadjacecent

example

The Ligadocom B Compound 10
The Ligadocom C Compensus 5
B Ligadocom C Compensus 3
B Ligadocom D Compensus 1
C Ligadocom B Compensus 2
C Ligadocom D Compensus 2
C Ligadocom and Compenso 9
D Ligadocom and Compenso 6
and LigaCom the Compensus 7
and Ligadocom D Compensus 4

Percorergher Example Using Search for Proof of Starting to Show Vertex Show “,“

D.2. Search in width

Function Add VerticesAdjacentSadjacents, vertices vertices = VerticesAdjacent to vertexevertesAdjacents = Obtering VerticesAdjacentdaitiser = There is a proximoitemem in iterator forvertices. Get ProximoitemDe Ite nerDevertSeAdjacecent Add VERTICEATUAL PUT SETTUSDOITER = There is proximoitemem iterator fo rrterticesadjacecents return set

Function seeks to enter the vertices, proximous action = {} iteratoresvertices = Vertices obtainer there is a proximoitemememitator = there is transproximoitemem iteratesrtices while there is a proximoitemememitato r == true vertice = Iterators Restices StatusVisitted = Obtain Vertice Visited Properity If Vertus Vertus! =

```
                              Defirproportion visited in Vertice Comvalor "Visited"
                              proximos = perform addjacentsadjacecents proximos,
vertex
There is a proximoitememitator = there is proximoitemem iteratoresvertese in iteatorsproximos = obtainer proxi
mos status = there is proximoitemem iterates proximos if status == real execute the proximal sake of the proxima
l, action


f
  Anointing seek tolagura graphoparameter, verticeinial, action
        Vertice = Verticeinicial
        Execute action providing vertice
        Defirproportion visited in Vertice Comvalor "Visited"
        verticesadjacents = vertices of vertice
        execute searching for vertices, action

digraph   example
        Ligadocom B
        The Ligadocom C
        B Ligadocom C
        B Ligadocom D
        C Ligadocom B
        C Ligadocom D
        C Ligadocom and
        D Ligadocom and
        and ligaocom a
        and ligadocom d
```

percorergher example using seeking to lap in starting to show vertice

D.3. Dijkstra

Function DefineTesComvisedFais Freight Grafoparameter Vertices = ObtertodesVertiSde GRAFOPARETE
R Iterator = Obteritrade Vertices StatusDoitiser = There is Iterator Proper as StatusDoiters == True V12} Ob
tain Itener Deferprit. VERICE VICE VERICE "UNLESSED" STATUSDOITER = There is a plaintiff

FUNCTION DEFIRPROPRIES FORDIJKSTRA GRAFOPARETER VERTICES = Obtertodertices GRAP
HOPARETER Iterator = ObteritisDeterator Vertices StatusDoitiser = There is aetener while statusDoiters
== Verice = Get ProximoiteMde Iterator Estimates Vertice Estimates with numeromaximo with Vertice Defi
rpromise with False StatusDoters = There is aetener

Function Search for graphoparameter, verticeinial, vertice = vertical -show visited vienitial with "Visited" Vertiice Providin
g vertices. = There is proximoitemem iterator forverticesDJacentes as statusdoiters == True VerticeproximoCando = Obtain
iteratorDerterticesDJacentes.

```
                    If status visited! = "visited"
                    Perform search for graphoparameter, verteceproximoCando, action, action
                    statusdoiterator = there is proximoitemem iterator forverticesadjacent
```

Function Check ProximoVertice Grafoparameter MINORIBLE = Numeromaximo Verticerector = False Run DefineT
esComvisedFais Graphoparameter Percorer Grafoparameter Using Searches In Starting The Statusdovertice = Get clo
sed vertice if statusdovertice == False VERTICE ESTIMATE = Obtain Vertice Estimation Estimation Sertice <= Sle
eping Verticerector = Sleeping Vertece

Dijkstra Function Grafoparameter, Verticeinitial, Verticeobjective Perform Define ProperPriends For Grafoparameter D
efirPrity Estimation in Verticeinic Comvalue 0 Verticenial Vericenial as Vertcenual! VERTICEISADJACENCY DER
MATUAL IteratorDerticesAdjacecent = ObteritrarisDJanteadjacentdaitener = There is proximoitemem iterator forrtices
adjacecents while statusdoiter == True verticeemverification = obtain averator forverterators. = Obtain VertiCeatual Rai
se for VertiCem Estimated Retortectual = Vericeual Estimation Estimation Estimation = Estimated social + Estimates E
stimationAntiga = Vericemarification Estimation.

```
                If estimated < EstimatesAntiga
                Define propriety estimate in verticemamation with estimation
                DEFINED PROPORTRY IN VERTICEMEMINATION WITH VERTICAL VERTICAL
                statusdoiterator = there is proximoitemem iterator forrterticesadjacent
                VERTICEATUAL = Run check proximovertice grafoparameter
R               eordor graphoparameter
```

example

```
               The Ligadocom B Compound 10
               The Ligadocom C Compensus 5
               B Ligadocom C Compensus 2
               B Ligadocom D Compensus 1
               C Ligadocom B Compensus 3
               C Ligadocom D Comprises 9
               C Ligadocom and Compenso 2
               D Ligadocom and Compenso 4
               and LigaCom the Compensus 7
               and Ligadocom D Compensus 6
```

Verticeinitial = obtain avertice of exemploufsc verticeobjective = Obk d of exemploufsc outp
ut = Run dijkstra example, verticeini, verticeobjective show output