UNIVERSIDADE DO VALE DO ITAJAÍ
CENTRO DE CIÊNCIAS TECNOLÓGICAS DA TERRA E DO MAR
CURSO DE CIÊNCIA DA COMPUTAÇÃO

DOMAIN-SPECIFIC LANGUAGE PARA GRAFOS

por

Jailson Nicoletti

Itajaí (SC), julho de 2014

UNIVERSIDADE DO VALE DO ITAJAÍ
CENTRO DE CIÊNCIAS TECNOLÓGICAS DA TERRA E DO MAR
CURSO DE CIÊNCIA DA COMPUTAÇÃO

DOMAIN-SPECIFIC LANGUAGE PARA GRAFOS

Área de Grafos

por

Jailson Nicoletti

Relatório apresentado à Banca Examinadora do Trabalho Técnico-científico de Conclusão do Curso de Ciência da Computação para análise e aprovação.
Orientador: Rafael Santiago, M. Sc.

Itajaí (SC), julho de 2014

AGRADECIMENTOS

Agradeço aos meus pais Jose Nicoletti e Nilza Maria Nicoletti e irmã Janara Nicoletti, por todo o apoio oferecido que me permitiu permanecer focado somente no trabalho durante o ultimo semestre. Também agradeço ao meu cunhado Ricardo Ghisi por todas as dicas e lições que me ajudaram não só na execução do trabalho, mas também no amadurecimento como profissional.

Ao professor Rafael de Santigo, meu orientador, por acreditar no meu potencial e por fornecer conhecimento, cobrança e suporte no decorrer da realização do projeto.

RESUMO

NICOLETTI, Jailson. **Domain-Specific Language para grafos.** Itajaí, 2014. 99 f. Trabalho Técnico-científico de Conclusão de Curso (Graduação em Ciência da Computação) – Centro de Ciências Tecnológicas da Terra e do Mar, Universidade do Vale do Itajaí, Itajaí, 2014.

O presente trabalho foi idealizado com o objetivo de diminuir a distância entre as bases teórica e pratica de grafos. Foi criada uma linguagem restrita ao contexto de grafos, capaz de descrever algoritmos que resolvem problemas com o uso dessa estrutura de dados. Foi feita uma pesquisa referente a problemas que envolvem grafos e suas características principais, seguida de uma pesquisa com foco em artigos científicos contendo utilização de algoritmos de grafo. Estas pesquisas foram feitas com o objetivo de obter o conhecimento necessário para definir como seria a linguagem. Uma segunda etapa na pesquisa foi conduzida na área de compiladores e de linguagens especificas de domínio, visando identificar as ferramentas, métodos e melhores praticas no desenvolvimento de uma linguagem desse tipo. A implementação contou com o auxilio da ferramenta ANTLR, responsável pela definição da gramática e integração com Java. Os testes das instruções foram feitos com a utilização da ferramenta JUnit, que nesse trabalho teve a responsabilidade de comparar os resultados obtidos com os esperados em cada um dos testes programados para as instruções. Além dos testes das instruções foram criados exemplos com algoritmos clássicos de grafos para confirmar a capacidade da linguagem de resolver os problemas da classe de grafos. Com o resultado positivo na execução das instruções e a execução correta dos algoritmos de exemplo, a ferramenta pode ser considerada uma alternativa positiva para o desenvolvimento de algoritmos voltados a grafos e também possui potencial didático por apresentar nativamente estruturas e instruções utilizados em livros e artigos desse tema.

Palavras-chave: Grafo. Domain Specific Languages. Algoritmos.

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.*

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE EQUAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

ANTLR	Another Tool for Language Recognition
DSL	Domain-Specific Language
GPL	General Purpose Language
EUA	Estados Unidos da América
IDE	Integrated Development Enviroment
SGBD	Sistema gerenciador de banco de dados.
SQL	Structured Query Language
TTC	Trabalho Técnico-científico de Conclusão de Curso
UNIVALI	Universidade do Vale do Itajaí

SUMÁRIO

1 INTRODUÇÃO

A computação fornece uma série de ferramentas para resolução de problemas. Para isto é necessário ter formas de representar estruturas existentes no mundo real. Como exemplo, podemos citar as ligações entre locais de um mapa ou um círculo de amizades. Estas são entre outras, estruturas passiveis de representação através de grafos (BOAVENTURA NETTO, 2006).

A teoria dos grafos surgiu em 1736, quando Leonhard Paul Euler, físico e matemático do século XVIII, estudava o problema das pontes de Königsberg. O nome do problema vem do nome da cidade que era território da Prússia até 1945. Königsberg é dividida por um rio, com duas ilhas que se interligavam com as margens através de sete pontes. A partir dai surgiu a seguinte questão "é possível atravessar todas as pontes sem que seja necessário passar pela mesma mais de uma vez?". Utilizando uma representação de grafo Euler conseguiu provar que seria impossível executar o feito. Após isso houve pouco desenvolvimento sobre a teoria dos grafos. Somente em 1936 que Denes Konig escreveu o primeiro livro sobre o tema. (HARJU, 2011, p.2)

A Ciência da Computação aborda grafos para temas na área de Teoria da Computação e como modelos para resolução de problemas. Porém as linguagens de programação genéricas não são orientadas à construção de programas que resolvam problemas através de grafos. Kreher *et al* (1999 p. 106,107,108,112), descrevem alguns algoritmos para problema de grafos de forma sintética se comparado a soluções utilizando linguagens de programação de propósito geral. Os autores utilizam operações de união e intersecção entre vértices e arestas para resolver os problemas.

Para facilitar a criação de algoritmos que trabalham com grafos, é necessária uma linguagem que torne mais simples a descrição dos seus problemas. Como ocorreu com a linguagem FIDO, uma DSL (Domain-specific Language) que foi projetada para conseguir expressar de forma concisa conjuntos regulares de strings e árvores (KLARLUND, NILS, 1997) .

DSL são linguagens projetadas para lidar com um domínio específico de problema. Alguns aspectos são importantes para o projeto de uma DSL (FOWLER, 2010):

- Linguagem de programação de computador: precisa ser de fácil entendimento e executável pelo computador.

- Linguagem natural: precisa ter um senso de fluência não só nas expressões individuais mas também no todo quando juntado.
- Expressividade limitada: a linguagem deve atender somente as necessidades de suporte de tipos de dados, controle e estruturas necessárias para seu domínio.
- Foco no domínio: a linguagem precisa ter um foco somente em seu domínio.

Como linguagem focada em um domínio, uma DSL fornece abstração necessária para resolução de problemas no contexto da aplicação. Na literatura, algumas vantagens são citadas:

- Produção aprimorada: são mais concisas e facilitam a interpretação do código pelo programador ou especialista da área, devido a sua sintaxe definida de forma similar aos termos utilizados pelo domínio do problema;
- Verificação facilitada: a semântica de uma DSL pode controlar algumas propriedades vitais para o funcionamento do sistema, desta forma conseguindo controlar itens como proteção contra deadlocks. (GULWANI *et al*., 2001);
- Aumentam a produtividade, confiança, manutenção e portabilidade. (DEURSEN e KLINT, 1998);

No entanto o uso de DSL também traz algumas desvantagens como o custo de projetar implantar, manter a linguagem, realizar o treinamento dos usuários, a dificuldade em achar um bom escopo e a potencial perda de eficiência. (KRUEGER, 1992).

Apesar das desvantagens citadas, o desenvolvimento e aplicação deste conceito, viabilizaram a criação de várias DSLs. Entre elas estão: SQL, HTML e CSS.

Um exemplo de DSL relacionada ao contexto da informática é a linguagem WebCal, que visa facilitar a resolução de alguns problemas de cache gerados por sites, oferecendo uma flexibilidade maior na configuração das politicas do cache, podendo gerar facilmente novas politicas locais de cache ou protocolos de inter-cache (GULWANI *et al*., 2001).

No contexto de esportes, a linguagem Easytime visa melhorar a configuração dos sistemas de medição do tempo através de um sistema de cronometragem flexível e a redução do número de equipamentos necessários para medição. Nos testes realizados pela equipe de desenvolvimento Easytime se mostrou útil para organizadores de competições e clubes esportivos auxiliando nos resultados de competições menores, e simplificando a configuração do sistema de cronometragem em grandes competições. (FISTER JUNIOR *et al*, 2011)

Devido à forma que são projetadas, DSLs são ferramentas que conseguem simplificar o trabalho dos especialistas e programadores dos sistemas que atuam na sua área de domínio,

além de conseguir assegurar que controles importantes para o correto funcionamento do sistema sejam atendidos (FOWLER, 2010). Desta forma auxiliando o desenvolvimento de seu campo de domínio.

A partir do estudo de grafos, do conhecimento que será adquirido sobre criação de linguagens e boas maneiras para a criação de DSL, será projetada uma nova linguagem que permita pessoas com conhecimento em grafos executar algoritmos que trabalhem aspectos da Teoria de Grafos.

1.1 PROBLEMATIZAÇÃO

1.1.1 Formulação do Problema

Há diversas áreas que resolvem seus problemas através de grafos. O campo da Biologia Molecular, Informática, Pesquisa Operacional e a Química, com o estudo das grandes redes, são exemplos de estudos direcionados com o auxílio da teoria dos grafos (BOAVENTURA NETTO, 2006, p.279). Logo, o uso de linguagens de programação de propósito geral podem dificultar a resolução destes problemas, pois são comumente utilizadas por profissionais de áreas da computação.

Este trabalho propõe o desenvolvimento de uma DSL, que visa simplificar a codificação de algoritmos de grafos utilizando uma sintaxe restrita ao contexto. Dessa forma a linguagem permitiria que qualquer pessoa com conhecimento na teoria de grafos possa interpretar ou produzir códigos da DSL, independente da área do profissional.

Como gerar uma DSL para simplificar o desenvolvimento de algoritmos envolvendo grafos?

1.1.2 Solução Proposta

Este trabalho propõe o desenvolvimento de uma DSL direcionada a produção de algoritmos que resolvam problemas de grafos. Para teste, serão desenvolvidos algoritmos com a sintaxe definida na linguagem criada. Os scripts criados serão executados em um analisador que será desenvolvido para executar os algoritmos da DSL criada.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo deste trabalho é desenvolver uma DSL para codificação de algoritmos que utilizem grafos.

1.2.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

- Definir a forma padrão de se trabalhar com grafos;
- Executar estudo sobre DSL já existentes no contexto de grafos;
- Identificar como será estruturada a linguagem;
- Desenvolver a DSL para trabalho com grafos;
- Analisar ao menos três algoritmos no contexto de grafos através de um interpretador gerado para a DSL;
- Elaborar documento contendo descrição de teste comparativo entre DSL e linguagens de propósito geral, realizado com pessoas que conhecem problemas resolvidos com grafos.

1.3 Metodologia

Este trabalho foi dividido em três etapas: fundamentação teórica, desenvolvimento e documentação.

Para a fundamentação teórica foram realizados estudos sobre a teoria de grafos para conhecer o tema da linguagem proposta, a estrutura de uma linguagem de programação através de materiais de compiladores e em paralelo a pesquisa de linguagem foram avaliados materiais específicos de DSLs para compreender melhor os diferenciais de uma linguagem planejada pra uso geral e a linguagem com domínio especifico, que será aplicada no trabalho. A parte final da fundamentação teórica buscou conteúdo referente a desenvolvimento de algoritmos que envolvam a Teoria de Grafos com o objetivo de encontrar padrões no desenvolvimento.

A etapa de desenvolvimento, contou com o desenvolvimento da IDE que implementa o interpretador da linguagem. Foram aplicados os conceitos aprendidos na fundamentação

teórica, o conteúdo de compiladores e DSL foram utilizados como guia para implementação da linguagem, enquanto a parte de grafo serviu como base para as definições da sintaxe.

Na etapa de documentação foi feita a redação do trabalho de conclusão de curso que apresentou o conteúdo obtido na fase fundamentação teórica e descreveu a linguagem e seu processo de implementação.

1.4 Estrutura do trabalho

Este documento está estruturado em quatro capítulos. O Capítulo 1, Introdução, apresenta uma visão geral do trabalho. No Capítulo 2, Fundamentação Teórica, é apresentada uma revisão bibliográfica sobre: grafos, assim como uma análise a respeito das DSLs. O capítulo 3, apresenta a linguagem e como foi implementada, além de descrever os testes executados. No capitulo 4 são apresentadas as conclusões do trabalho, onde é citado os métodos utilizados no desenvolvimento, as dificuldades e soluções encontradas durante o trabalho. Por fim os apêndices apresentam itens criados no decorrer do trabalho que serviram como apoio no desenvolvimento e testes da linguagem.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção visa fundamentar as decisões que serão tomadas na fase de projeto e apresentar os temas abordados. Está dividida em: Grafos, Linguagens de programação, Domain-Specific Language, Algoritmos para problemas em grafos.

A seção de grafos irá apresentar a definição e características de um grafo além de explicar alguns problemas que serão posteriormente utilizados na analise de algoritmos, seção que visa identificar padrões de sintaxe na descrição de algoritmos para problemas de grafos.

A seção de Linguagens de programação irá descrever a estrutura de desenvolvimento de uma linguagem de programação e a seção Domain-Specific Language vai explicar o significado do termo DSL e citar características importantes de sua estrutura e desenvolvimento.

2.1 Grafos

A partir da criação da Teoria dos Grafos no século XVIII muito foi produzido utilizando os conceitos abordados por ela. Várias áreas de conhecimento como química orgânica, biologia e eletrônica tiveram estudos que utilizaram suas técnicas. No entanto, inicialmente houve pouco interesse neste estudo. Após a criação de Euler no século XVIII, o primeiro livro dedicado a esta teoria foi lançado em 1936 e a maior parte da produção cientifica a respeito apareceu a partir de 1970 e nesta mesma época suas aplicações foram impulsionadas pela utilização do computador. (NETTO, 2006).

Um grafo é um conjunto finito não vazio de vértices e suas possíveis ligações. Pode ser representado formalmente como uma dupla G = (V,A). Onde V é um conjunto não vazio de vértices e A é o conjunto de arestas (ligações entre os vértices). Cada aresta é representada por um conjunto de pares não ordenados de elementos de V.(SZWARCFITER, 1988).

Dentro do grafo, os vértices podem ser associados a objetos do mundo real, como por exemplo: uma cidade, pessoa, parte de um circuito eletrônico, entre outros. Quando representado graficamente os vértices normalmente são pontos ou círculos. Eles possuem um grau que é definido pelo seu número de vizinhos ou seja o número de vértices que estão conectados a ele.(DIESTEL, 2005).

As ligações entre os vértices do grafo são as arestas, representadas por linhas, curvas ou retas, conectadas aos vértices, desta forma representando sua ligação. As ligações podem representar várias coisas como o caminho entre duas cidades, a amizade entre duas pessoas, ligação entre dois componentes de um circuito entre várias outras possibilidades. Dependendo o que as arestas representam podem possuir valores para representar dados como a distância entre dois locais. (HARTSFIELD, 1990). A Figura 1 representa um grafo com seis vértices representados por círculos e sete arestas representados por linhas.

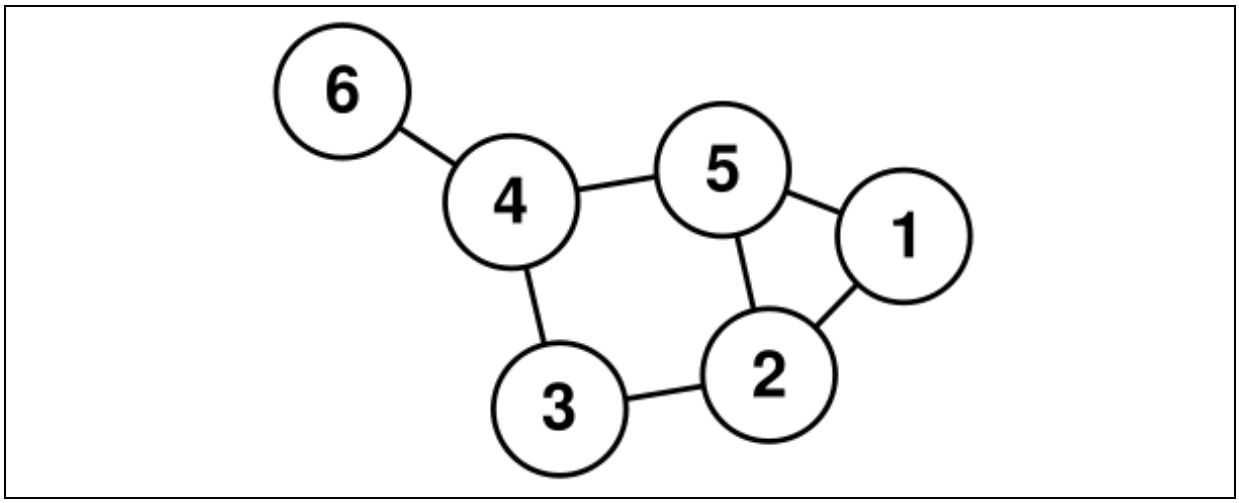


Figura 1. Exemplo de grafo.
Fonte: Danielamaral.wikidot (2012).

Quando dois vértices de um grafo estão conectados por uma aresta passam a ser definidos como vértices vizinhos ou adjacentes. Esta ligação entre um par de vértices pode ser feita por mais arestas, ou seja, os vértices *A* e *B* estarem conectados pelos vértices *f* e *g* ao mesmo tempo, nesta relação às arestas *f* e *g* seriam chamadas de arestas paralelas. Este número variável de arestas e vértices em um grafo definem respectivamente o tamanho do grafo e sua ordem. (BALAKRISHNAN, 2005).

A seguir serão apresentadas características de classificação de grafos que serão utilizadas na descrição de problemas de grafos e nos algoritmos que trabalham estes problemas.

2.1.1 Classificação de Grafos

Alguns termos foram criados para determinar características de um grafo. Esta seção irá apresentar e explicar os termos necessários para a compreensão dos problemas que são apresentados.

Laços são ligações entre o mesmo vértice, conforme demonstrado na Figura 2, onde o vértice 1 possui uma aresta que sai e volta diretamente para ele. Arco é a denominação dada para uma aresta dentro de um grafo direcionado também chamado de dígrafo, o diferencial neste caso é que a conectividade nas ligações é dada somente em uma direção, ou seja, o vértice 1 pode estar ligado ao 2 e não haver ligação direta entre o vértice 2 e 1. Seguindo o conceito de uma via de mão única, onde os carros podem seguir somente na direção determinada. Conforme ilustrado na Figura 3. (NETTO, 2006). Nesse trabalho os laços e arcos serão chamados em sua forma generalizada como arestas.

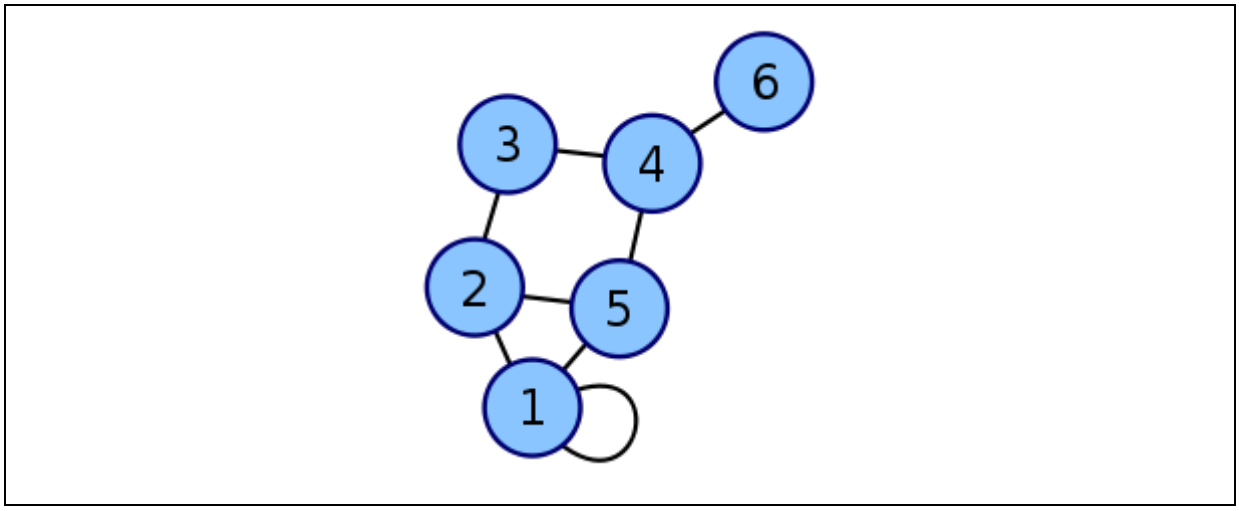


Figura 2. Grafo com laço
Fonte: Wikipedia, **Laço** (2012)

Nos dígrafos o grau de entrada é contabilizado a partir do número de arcos que estão direcionadas para ele enquanto o grau de saída é a quantidade de arcos que são direcionados para outro vértice a partir dele. (SZWARCFITER, 1988).

Grafos direcionados também chamados de dígrafos possuem graus de conexão, onde fortemente conexo indica uma estrutura em que todas as arestas conectadas possuam caminhos de ida e volta. (unilateralmente ou semi-fortemente conexo) conexo quando em todo par de vértice existe ao menos uma conexão entre eles, mesmo que esta seja unilateral.

Fracamente conexo ou desconexo quando existe pelo menos um par de vértices não estiver conectado restante do grafo.(SZWARCFITER, 1988), (NETTO,2006).

O termo atingível ou alcançável é utilizado nos grafos direcionados para demonstrar a possibilidade que o vértice A tem de atingir o vértice B, não havendo a necessidade de conexão direta, ou seja, é atingível se existe algum caminho entre dois vértices. Se algum vértice consegue alcançar todos os outros vértices de determinado grafo então este é chamado de raiz do grafo.(SZWARCFITER, 1988). Em grafos não orientados todos os vértices conexos são também atingíveis. (NETTO, 2006)

Um passeio entre dois vértices de um grafo é a passagem pelos vértices e arestas que os separam sendo que o número de arestas define o tamanho do passeio. Este processo lista todos os vértices e arestas transcorridos, com exceção dos grafos simples, onde não é necessário listar as arestas devido a existência de no máximo uma ligação entre cada par de vértices. O passeio é chamado de trilha se nenhuma aresta é repetida e é chamado de caminho quando nenhum vértice aparece mais de uma vez na lista. Um passeio com o vértice de partida e destino iguais é chamado de caminho fechado e se ele não possuir arestas repetidas é denominado como circuito. Um ciclo é um circuito que não possui vértices repetidos. (BALAKRISHNAN, 2005)

Um grafo conectado é assim chamado quando existe um passeio entre todos os seus vértices, ou seja, partindo de qualquer um dos vértices é possível atingir todos os outros (DIESTEL, 2005). A Figura 3 abaixo apresenta um grafo conexo.

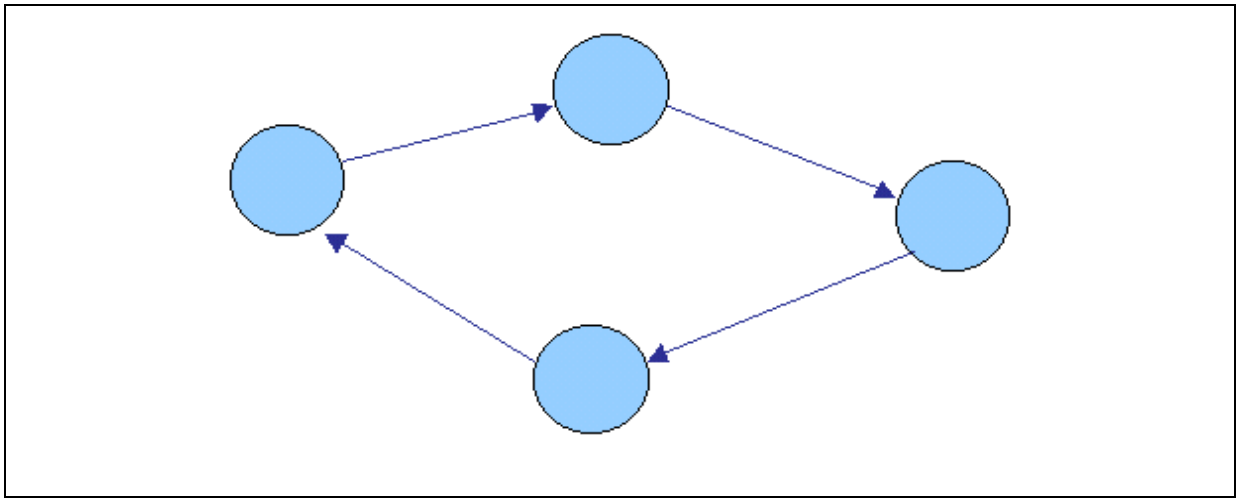


Figura 3. Grafo conectado

Fonte: Educ.fc.ul. (2012)

Grafos são frequentemente utilizados para representar estruturas do mundo real. A criação de mapas e circuitos são áreas que utilizam um tipo particular de grafo, os grafos planares. Este tipo de grafo tem a característica de poder ser desenhado em um plano de forma que não existam arestas que se cruzem. (BOAVENTURA,2006) A Figura 4 abaixo apresenta um grafo planar em sua forma normal e logo seguida em sua forma topológica.

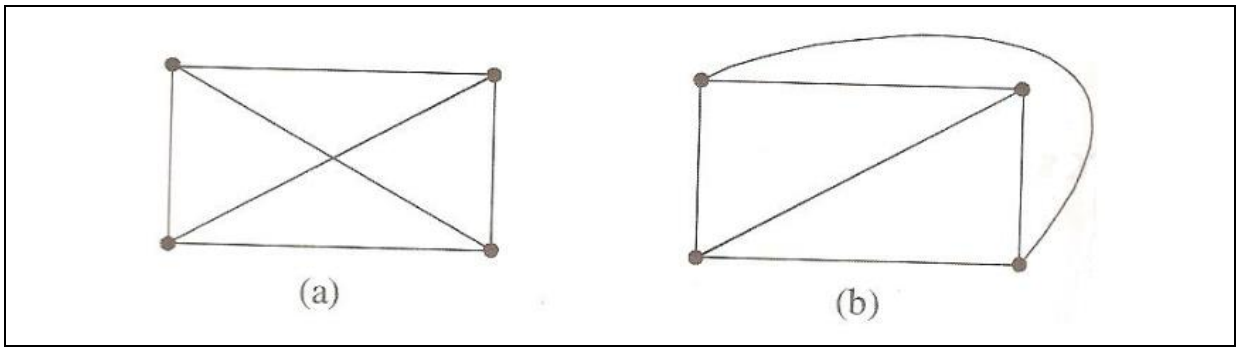


Figura 4. Grafo planar (a) e sua forma topológica (b)
Fonte: Boaventura (2006)

Isomorfismo é uma equivalência estrutural em grafos, ou seja, o isomorfismo é a bijeção entre os conjuntos de vértices de dois grafos de forma que dois vértices são adjacentes em um grafo somente se for adjacente também no segundo grafo. Desta forma se o nome dos vértices de um grafo for alterado conforme os nomes do outro grafo os dois ficariam iguais (Ime.usp, 2012).

Uma operação de contração em um grafo é um resumo onde dois ou mais vértices se fundem e herdam as ligações que cada vértice tinha com o restante do grafo. A partir desta operação é possível encontrar a relação *minor* de um grafo. Se um grafo X é resumido pela operação contração e o grafo resultante é um subgrafo de Y então se diz que o grafo X é *minor* de Y. (DIESTEL, 2005). A Figura 5 abaixo apresenta uma série de contrações executadas em um grafo, para melhor compreensão a primeira contração a acontece nos vértices D e K que se tornam DK.

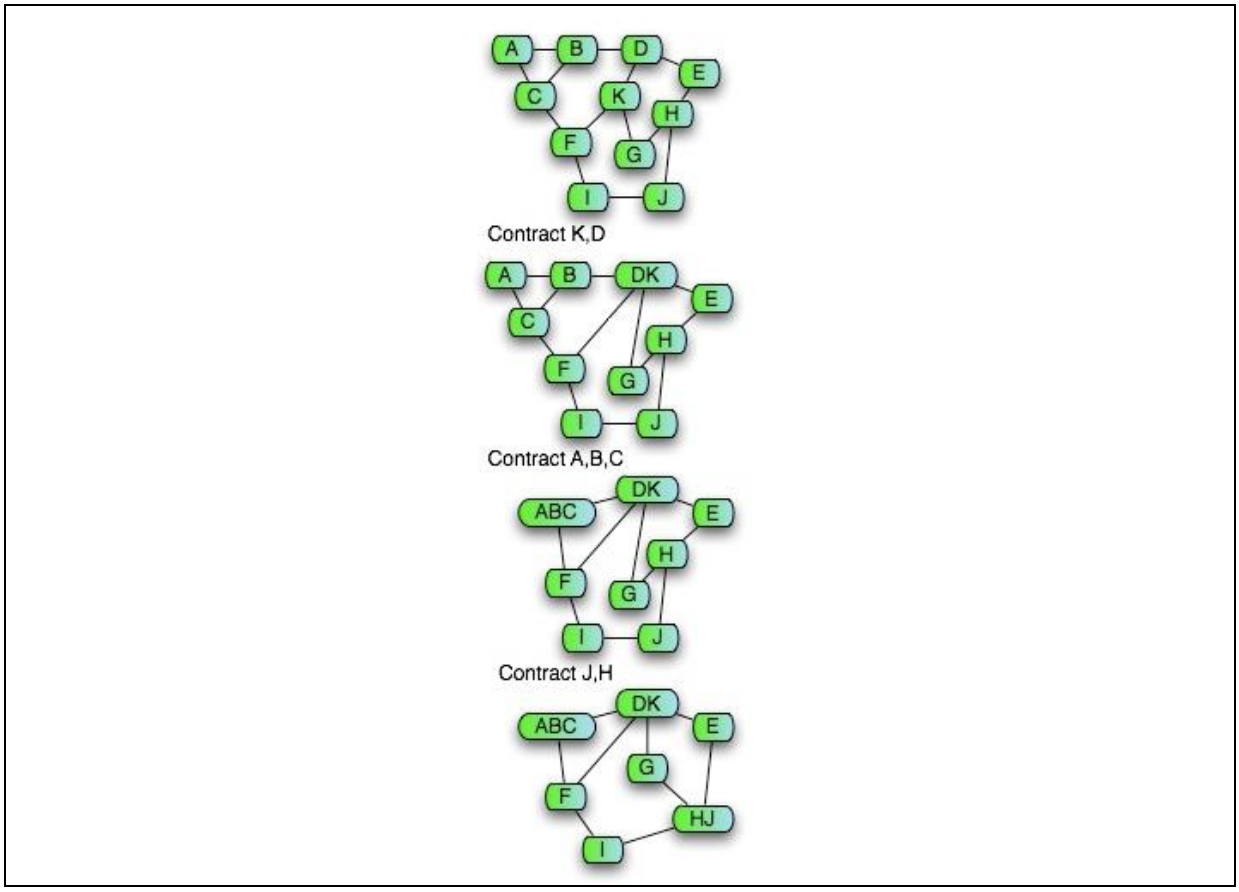


Figura 5. Operações de contração em um grafo

Fonte: Scienceblogs (2012)

Um grafo G é chamado de r-partite, onde r é um número maior que um, se seus vértices permitem uma partição em r classes de forma que cada aresta tenha suas extremidades em diferentes classes. Em casos onde o grafo seria 2-partite ele normalmente assume o nome bipartite. (DIESTEL, 2005). A Figura 6 abaixo ilustra um grafo 3-partite.

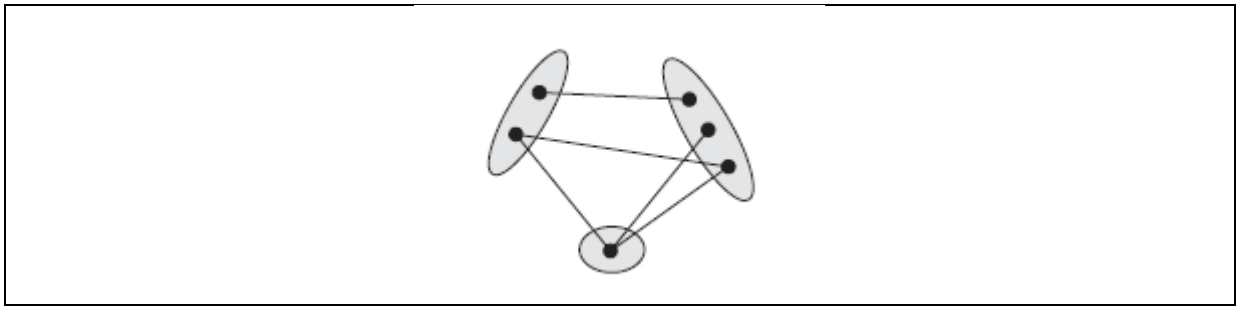


Figura 6. Grafo 3-partite

Fonte: (Diestel, 2005)

2.1.2 Problemas em Grafos

Dentre os problemas existentes na área de grafos, as questões referente a busca ganham uma importância especial devido ao grande número de problemas solucionáveis através de seus algoritmos (muitos deles resolvidos em tempo exponencial). Um grande problema na busca dentro da estrutura de um grafo é a falta de orientação de localização dentro do grafo, isto é, se torna necessário a verificação de vértice visitado por não haver uma clara estrutura sequencial de vértice para vértice. (SZWARCFITER, 1988).

Esta seção irá definir três problemas que envolvem busca em grafos: Caminho mínimo, Coloração e Clique máximo. Posteriormente serão apresentados implementações feitas por diferentes autores, tendo como objetivo identificar padrões de linguagem na criação de algoritmos que trabalham grafos.

2.1.2.1 Caminho Mínimo (resolução com algoritmo de dijkstra)

O problema do caminho mínimo procura a menor distância entre um vértice e outro, levando em conta a soma do valor dos pesos das arestas percorridas. Este assunto tem grande importância devido a constante necessidade que o ser humano tem de identificar o menor custo para algum objetivo, seja por conveniência de um individuo ou para uma grande companhia diminuir custos. (SHEZAD, 2009).

O algoritmo de Dijkstra resolve este problema com o tempo $O(n^2)$ em grafos que possuam arestas com valor maior ou igual a zero. Ele utiliza uma busca pelo vizinho mais próximo, a partir de um vértice inicial, a cada iteração ele encontra o vértice adjacente com aresta de menor custo, acumula este valor e fecha o vértice. Esta sequencia é repetida até que o vértice destino seja encontrado, quando isso acontece o algoritmo volta nos vértices fechados e procura novas adjacências para executar o mesmo processo acumulando o custo do novo caminho. Ao final o caminho com o menor custo acumulado é o escolhido pelo algoritmo (NETTO, 2006). A Figura 7 apresenta uma ilustração da sequencia de passos do algoritmo de dijkstra executada em um grafo.

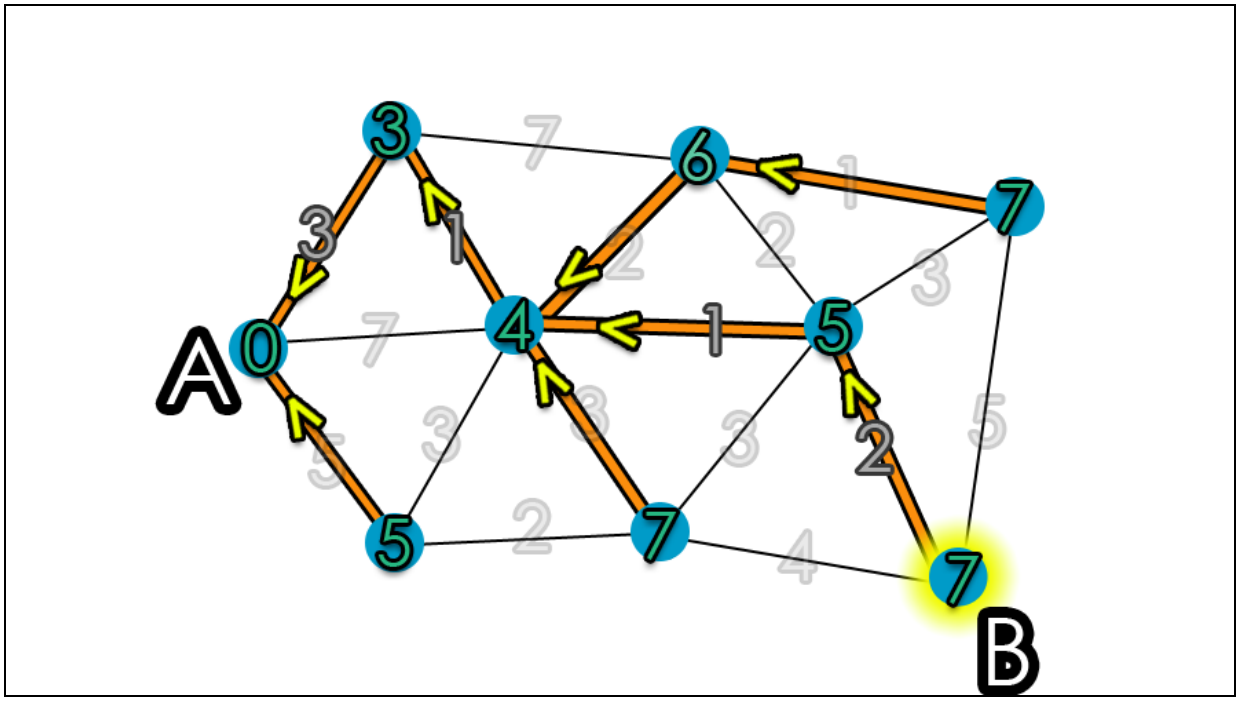


Figura 7. Execução algoritmo de Dijkstra
Fonte: Vasir (2012)

2.1.2.2 Coloração de grafos

A coloração de vértices é um processo que colore o grafo de forma que vértices adjacentes não tenham a mesma cor. O número mínimo de cores necessárias para colorir o grafo é definido pela variável k. Este valor mínimo é chamado de número cromático e o grafo correspondente será chamado de k-cromático. Desta forma um grafo que precisa de 3 cores para ser colorido será chamado de 3-cromático. (DIESTEL,2005)

A coloração de grafos é normalmente utilizada em situações que utilizem análise combinatória, onde diferentes informações geram conjuntos que precisam ser unidos para resolver algum problema, um exemplo conhecido é o problema dos exames. Ele consiste na busca do menor número de dias necessários para a realização de um exame onde os candidatos precisam fazer determinadas provas (BOAVENTURA, 2006).

2.1.2.3 Clique Máximo

Na Teoria de Grafos o Clique é um subconjunto de vértices conectados de forma que exista uma aresta entre cada dupla de vértices. Seu tamanho é definido pela cardinalidade de seu conjunto de vértices. (Ime.usp, 2012) O Clique com a maior cardinalidade (maior quantidade de elementos do conjunto) do grafo é chamado de Clique Máximo, sendo que

podem haver vários dentro de um grafo. (ROSEN, 1999). A Figura 8 apresenta um exemplo de Clique Máximo entre os vértices 3,4,5,6 que estão mutuamente conectados.

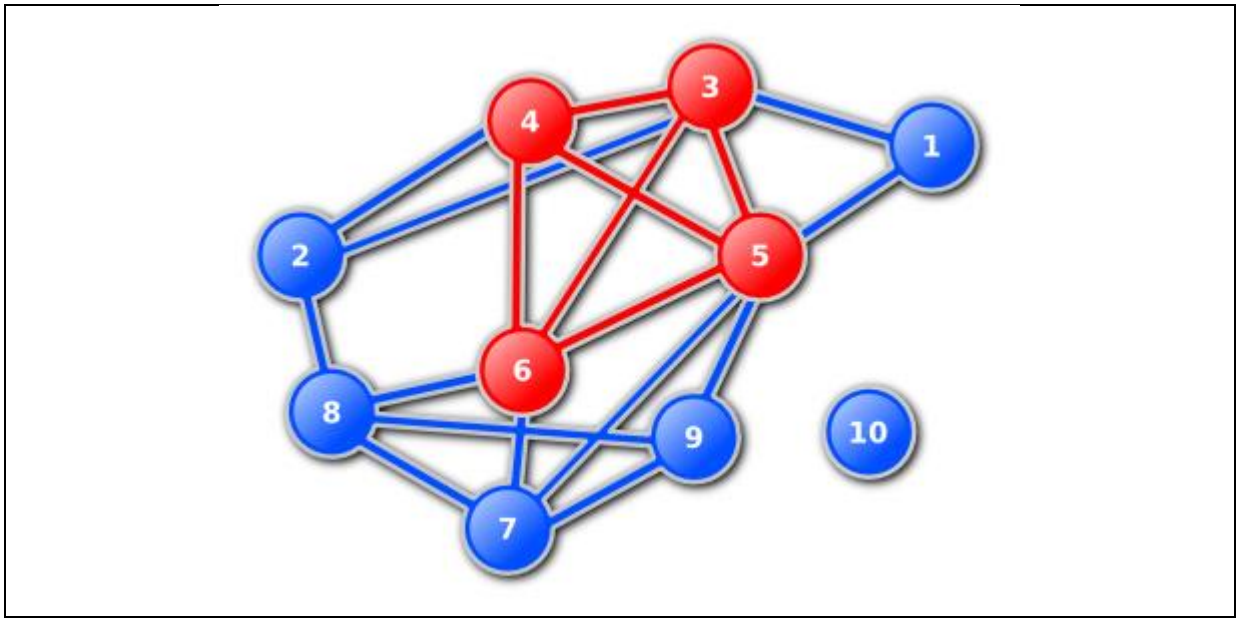


Figura 8: Clique Máximo. Vértices 3,4,5,6.
Fonte: Reactive-search (2012)

2.2 Linguagens de programação

Linguagens de programação são conjuntos de instruções definidas em uma gramática que servem para executar os comandos de um algoritmo e assim dar ordens ao computador. (SAID, 2007).

No projeto de uma nova linguagem é necessário definir sua gramática e semântica. A gramática ou sintaxe define os termos que a linguagem vai reservar e as regras que o programador deve seguir para gerar o código. Através das regras da gramática é gerada a árvore sintática também chamada de *parse tree*. As regras gramaticais são criadas de forma que cada termo tenha uma forma de utilização predefinida e um determinado comportamento. O comportamento gerado pelo termo da sintaxe da linguagem é determinado semântica. (FOWLER, 2010).

Neste conjunto de estruturas que formam uma linguagem a gramática é a parte em que o programador expressa a lógica de sua aplicação. Ela consiste em um conjunto de termos reservados para a linguagem e que são utilizados sob algumas regras definidas para a formação dos comandos. Estas cadeias de termos podem formar produções mais complexas através de associações entre elas. A união entre as cadeias, pode ser feita de várias formas,

desde que estejam seguindo as regras definidas, com isso podem ser utilizadas de formas diferentes e consequentemente produzirem programas para diversos finalidades. (Jandl Jr, 2001)

A partir do reconhecimento das cadeias e dos termos contidos em cada uma, é iniciado a interpretação do código. Esta fase é chamada de analise léxica e se responsabiliza pela primeira fase do processo que ira gerar as ações definidas no código fonte. Neste estágio o analisador léxico recebe o texto contendo o código escrito e identifica seus símbolos definidos pela gramática. (SEBESTA, 2002)

Com a analise léxica concluída os símbolos identificados precisam ser analisados para identificar se estão na estrutura definida pela gramática. Este passo é chamado de analise sintática e faz a inserção dos termos encontrados dentro de uma estrutura de dados, normalmente chamada de arvore sintática. Com os termos arranjados na arvore o objetivo é seguindo as predefinições da gramática conseguir chegar a um símbolo inicial, o que valida o código de entrada dentro da gramática.(LOUDEN,2004)

Através da análise sintática o *parser* consegue garantir que todas as regras da gramática sejam cumpridas, porém, as linguagens costumam ter necessidade de controlar outras informações como o tipo dos dados, declaração de variáveis e outros tipos de informações dependendo da linguagem. Esta tarefa é cumprida pela analise semântica que atua na verificação do relacionamento entre as partes do código, gerando uma interpretação do conteúdo de entrada e fornecendo aos componentes internos da linguagem os dados necessários para a execução do programa. (RICARTE, 2008)

Os dados obtidos na análise semântica, possuem a informação necessária para executar o comportamento descrito pela DSL, porém, alguns detalhes podem ser adicionados para garantir a qualidade da DSL. Conforme acontece na maioria dos programas projetados, a existência de camadas de software com papéis bem definidos, facilitam o processo de desenvolvimento e manutenção de software (MARTINS, 2007).

2.3 Domain-Specific Language

DSL (Domain-Specific Language) é um termo criado para definir uma linguagem que foi projetada para atender uma área especifica. Esta especificidade aplicada na sintaxe visa implementar softwares com flexibilidade e que tornem o processo de desenvolvimento mais ágil. São normalmente pequenas, declarativas e oferece uma sintaxe restrita, conforme a necessidade do domínio trabalhado. A clareza na sintaxe possibilita que o próprio usuário final possa escrever algumas tarefas, como acontece com macros em softwares de planilhas eletrônica. (DEURSEN, 1998)

Como em projetos comuns, feitos em GPLs (General Purpose Language), é necessário haver uma abstração que modele o domínio do problema e facilite a programação das regras do programa. Normalmente são utilizadas bibliotecas ou *frameworks* que abstraem parte da lógica do programa principal. (FOWLER, 2010).

A DSL é implementada acima deste modelo e fornece um novo nível de abstração para o programador. Ela pode tornar o modelo mais simples para pessoas treinados a resolverem problemas específicos sua compreensão e possibilitam a configuração do ambiente em tempo de execução. O modelo fornece o comportamento da aplicação que possui os dados oriundos do código da DSL previamente interpretados pelo parser. A DSL é um adjunto do modelo que fornece algumas utilidades para o programador. (FOWLER, 2010).

Embora em projetos de GPL seja natural a criação do modelo antes do programa, na produção de DSLs é comum o projeto da linguagem ser feito antes do modelo. Isto acontece, pois atuam em camadas diferentes interligadas em tempo de execução pelo parser. (FOWLER, 2010).

É importante conhecer os benefícios na utilização de DSLs para conseguir identificar quando é viável sua utilização. Estes benefícios são:

- Produtividade no desenvolvimento: A clareza na sintaxe torna mais fácil a identificação de erros e facilita a manutenção, desta forma evitando erros no produto final e diminuindo tempo gasto com manutenção de código. Além disso também proporciona uma abstração melhor para que o programador não perca o foco do problema a ser resolvido;

- Comunicação com especialistas da área: No projeto da DSL pode ser definida uma sintaxe que cumpra com este quesito. Desta forma os usuários finais do sistema conseguem ler e interpretar o código, facilitando a detecção de erros na lógica em alguns casos permitindo a codificação por eles mesmos;

- Mudança em contexto de execução: Serve para arquivos de configuração de sistemas com configuração necessária, alterando o arquivo o sistema passaria a ter um comportamento diferente. como por exemplo alterar entre base de produção e teste, para troca de consultas sql em tempo de execução, a possibilidade do mesmo código ser traduzido em diferentes locais, como por exemplo uma validação de dados que roda tanto na visão quanto no server mesmo que estes estejam em linguagens diferentes, também oferece a possibilidade de usuários fornecerem novas regras em tempo de execução através de uma linguagem mais compreensível para usuários do sistema;

- Modelo computacional alternativo: Muitas vezes o modelo de programação imperativo não se aplica tão bem a todos os problemas. Em casos onde um modelo de programação declarativa se torna mais interessante as DSLs conseguem ajudar através da adaptação para estes modelos de computação. (FOWLER, 2010).

Para definir melhor as características de uma DSL são definidos três tipos:

- DSL Interna - É uma forma de utilizar partes de uma linguagem de uso genérico para gerar uma especialização capaz de abstrair um problema Este tipo de DSL utiliza o interpretador da linguagem de uso geral para ser processada;

- DSL Externa – Utiliza uma sintaxe própria, que é interpretada por um parser implementado em uma linguagem genérica;

- Language workbench - DSL executada em uma ferramenta própria criada para ela, permite uma melhora no uso contra a DSL externa pois na DSL externa é possível somente alterar o texto enquanto neste estilo podem ser adicionados ferramentas especificas para aquele tipo de linguagem.

O desenvolvimento de um modelo semântico oferece várias vantagens no projeto de uma DSL.Alguns exemplos são:

- permite testar separadamente o comportamento e a interpretação da DSL;
- facilitar a verificação da interpretação do código através da observação dos dados inseridos no modelo semântico a partir da DSL;
- facilita a evolução da linguagem sem o risco de alterar os comportamentos já definidos;
- em casos onde existem vários parsers o teste de integridade entre eles é facilitado pela verificação dos dados no modelo semântico;
- possibilita o teste do comportamento da linguagem separadamente através da inserção manual de dados;
- separa o pensamento das camadas do parser das camadas de comportamento (FOWLER, 2010).

O modelo semântico oferece um molde para as informações da DSL, tomando a forma necessária para abstrair o domínio do problema na linguagem de uso geral. A partir dele podem ser definidas as ações e respostas da DSL. Embora este modelo seja muito similar ao modelo de objetos, o termo é diferente, pois em alguns casos o modelo pode não ser uma derivação do domínio do problema central em um sistema (FOWLER, 2010).

2.3.1 Exemplo de DSL

Para identificar estruturas básicas necessárias em uma linguagem que aborde grafos foi pesquisado linguagens de programação que já possuíam esse objetivo. Foi feita uma busca no site *http://scholar.google.com.br/* por artigos e livros que descrevessem uma DSL com escopo na execução de algoritmos de grafos, porém até a data de conclusão desse trabalho não foi encontrada nenhuma linguagem para esse fim.

Para compreender como especificar uma DSL, é descrito um exemplo selecionado devido às seguintes características: uso de conjuntos, escopo especifico. Nesta busca foi encontrado um artigo de Klarlund apresentando a linguagem FIDO. Os dados da linguagem levantados por ele são apresentados a seguir. (KLARLUND, 1997).

Nesta pesquisa foi encontrada a Monadic Second-Order Logic (M2L), que através das regularidades identificadas no domínio do problema conseguem reduzir problemas desafiadores a questões de strings regulares ou linguagem de arvores. A desvantagem desta lógica é que ela possui um limite inferior não elementar em seus procedimentos de decisão. (KLARLUND, 1997).

A implementação desta lógica é chamada de MONA e pode lidar com fórmulas não triviais, tendo registro de até quinhentos mil caracteres. O problema é que a descrição destes conjuntos se torna muito complicada e exige que os programadores M2L gastem grande parte do seu tempo revisando códigos complexos. (KLARLUND, 1997).

FIDO foi implementada para fornecer um compilador otimizado de fórmulas M2L e já foi utilizado em varias aplicações reais. Na Equação 1 é apresentada uma fórmula no padrão M2L que é apresentada na linguagem na Equação 2 dentro da sintaxe FIDO (KLARLUND, 1997).

Equação 1. Fórmula no padrão M2L
Fonte: Klarlund (1997)

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

Equação 2. Fórmula M2L no padrão FIDO
Fonte: Klarlund (1997)

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

2.4 Algoritmos para Problemas em Grafos

Esta seção irá apresentar nove algoritmos encontrados em artigos de diferentes autores para resolver respectivamente os três problemas da Teoria de Grafos: Caminho Mínimo (utilizando dijkstra), Coloração e Clique máximo. Ao final serão comentadas as características identificadas nos algoritmos, quanto a suas sintaxes.

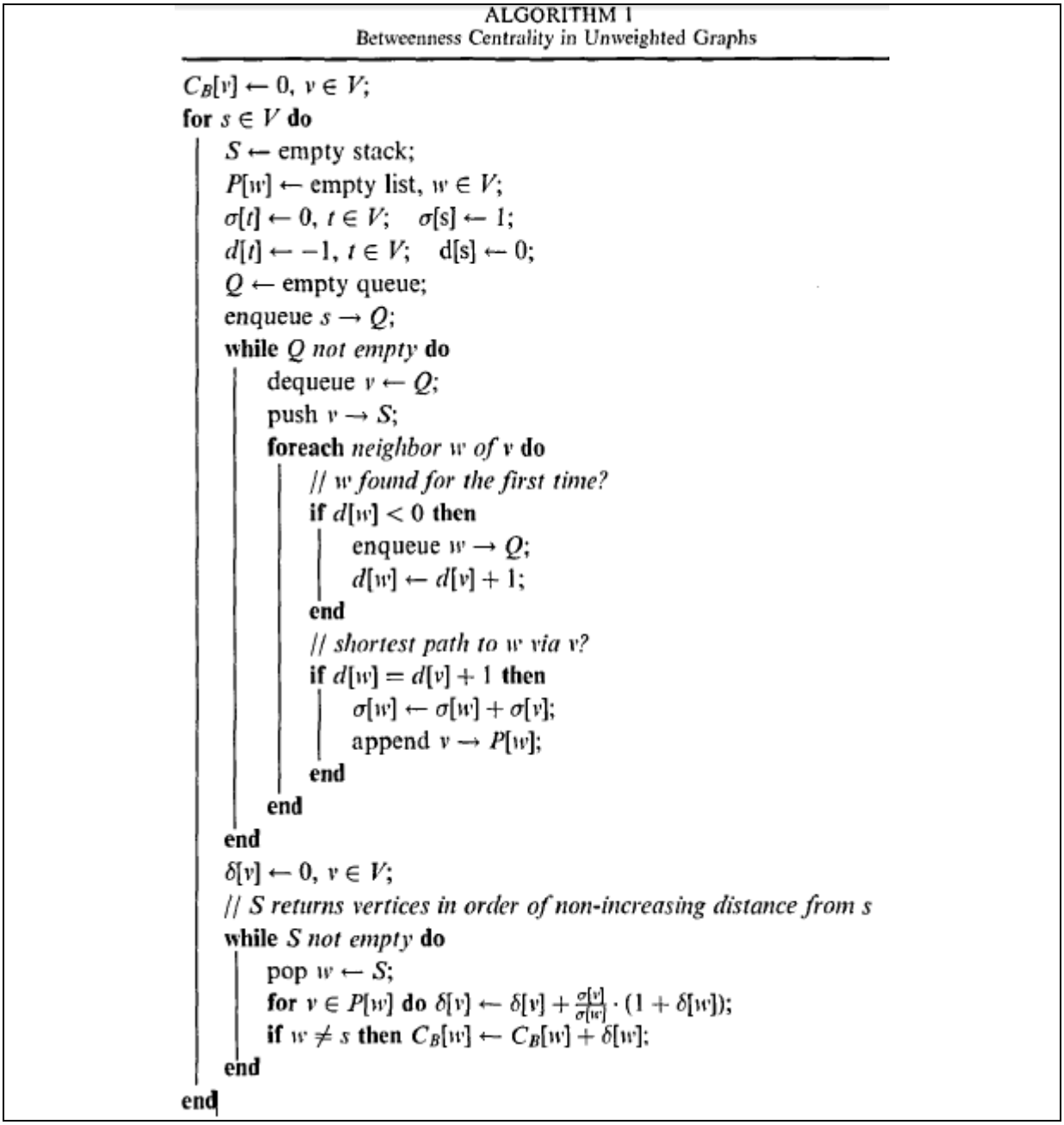
2.4.1 Dijkstra

Essa seção apresenta artigos publicados que relatam aplicações do algoritmo de dijkstra, ao todo serão apresentados três artigos dos seguintes autores: Brandes (2000), Shehzad (2009) e Shi (1998).

2.4.1.1 A faster algorithm for Betweenness centrality in unweighted graphs

Brandes escreveu um artigo que aborda o índice de centralidade no vértice de um grafo. O objetivo é classificar um vértice de acordo com a posição no grafo, assunto que vem ganhando mais importância com o aumento de redes complexas principalmente dentro de redes sociais. No trabalho de Ulrik, o algoritmo de Dijkstra é utilizado para verificar o número de menores caminhos entre cada par de vértices do grafo com peso (BRANDES, 2000). O algoritmo é apresentado na Figura 9.

O algoritmo apresenta algumas características como, o laço especifico que percorre todo o conjunto de vértices. Além disso pode se notar o uso de três estruturas de dados para cumprir seus objetivos: pilha, fila e lista.



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, $w \in V$;
  $\sigma[t] \leftarrow 0,\ t \in V;\quad \sigma[s] \leftarrow 1;$
  $d[t] \leftarrow -1,\ t \in V;\quad d[s] \leftarrow 0;$
  $Q \leftarrow$ 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

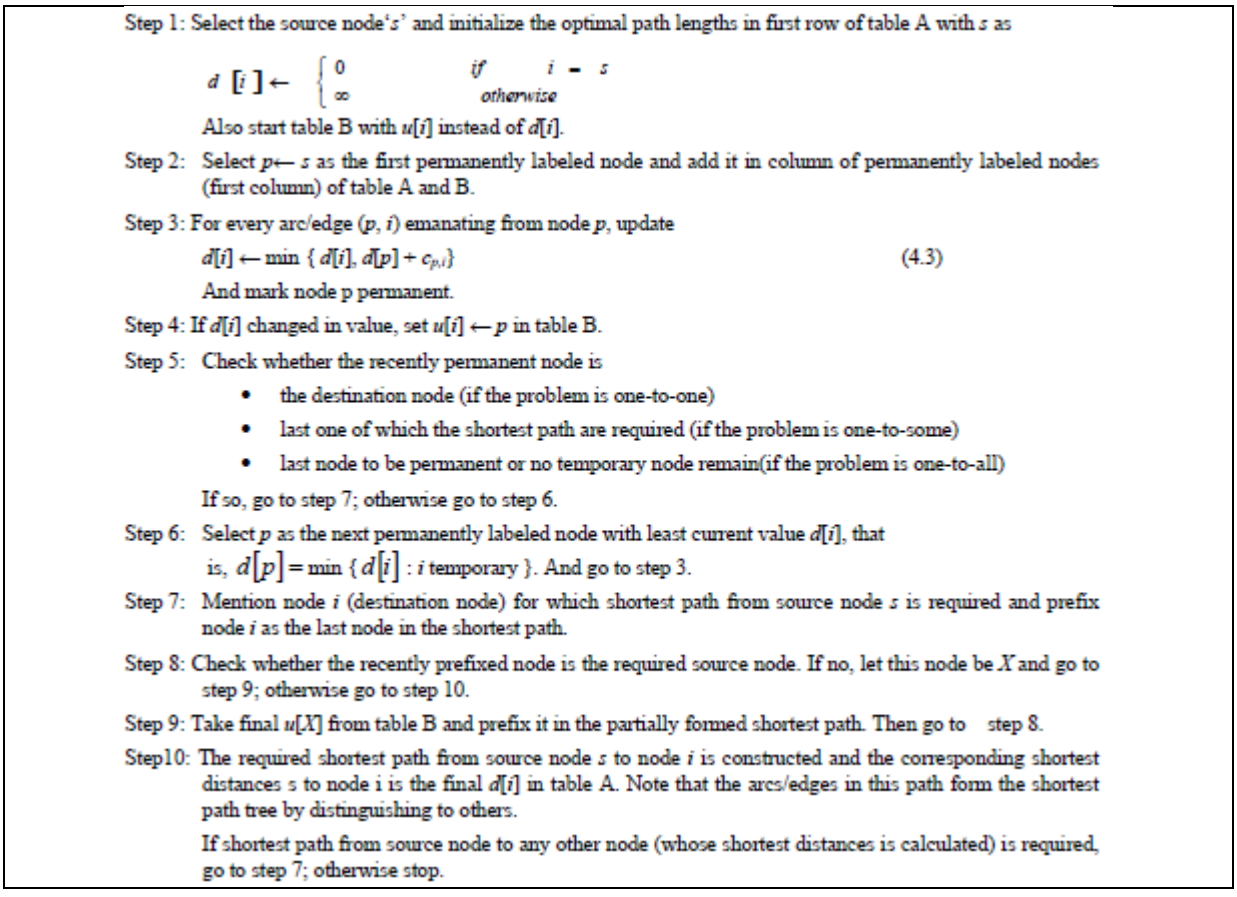
Figura 9. Algoritmo Dijkstra 1.

Fonte: Brandes (2000).

2.4.1.2 Evaluation of Shortest paths in road

Em 2009 Farrukh Shehzad escreveu o artigo Evaluation of Shortest Paths in Road Network. O estudo é baseado na otimização da distância em estradas através da escolha da melhor rota e para isso são aplicados os algoritmos de Dijkstra e de Floyd-Warshall. Uma versão modificada do algoritmo de Dijkstra foi aplicada neste estudo gerando como produtos a tabela A que contém a menor distancia do vértice raiz para todos os outros e a tabela B ajuda na recuperação dos caminhos mais curtos correspondentes (SHEHZAD, 2009). O algoritmo é apresentado na Figura 10 abaixo.

O autor faz o uso de uma descrição textual, e utiliza laços de repetição através da função goto. Além disso foram utilizadas tabelas para guardar valores referente a execução do algoritmo.



Step 1: Select the source node 's' and initialize the optimal path lengths in first row of table A with s as

$$d[i] \leftarrow \begin{cases} 0 & \text{if } i = s \\ \infty & \text{otherwise} \end{cases}$$

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

$$d[i] \leftarrow \min \{ d[i], d[p] + c_{p,i} \} \qquad (4.3)$$

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.

Figura 10. Algoritmo Dijkstra 2.

Fonte: Shehzad (2009)

2.4.1.3 Parallel Mesh Algorithms for grid graph shortest paths with application to separation of touching chromosomes

A separação de cromossomos que se encostam dos que se sobrepõe em imagens é abordado por Hongchi Shi (1998) em seu artigo. O principal processo desta aplicação é a busca do menor caminho em grafos de malha, devido a este importante papel dos algoritmos de caminho mínimo foi feito um estudo em alguns destes algoritmos, entre eles o algoritmo de Dijkstra que é apresentado na Figura 11.

SHI (1998) faz o uso de uma descrição bem resumida e utiliza a teoria dos conjuntos para trabalhar com vértices e a operação de união para completar o caminho mínimo.

Initialize $V_T = \{\mathbf{u}_0\}$ and $l_{\mathbf{u}_0}(\mathbf{u}) = 0$ if $\mathbf{u} = \mathbf{u}_0$ and ∞ 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 $\mathbf{v}$.

Figura 11. Algoritmo Dijkstra 3
Fonte: Shi (1998)

2.4.2 Coloração

Esta seção irá apresentar algoritmos encontrados em artigos referente a coloração, procurando identificar a forma que os algoritmos funcionam e particularidades de cada algoritmo criado pelos autores: Halldórsson (1993), Klotz(2002) e Wigderson (1983).

2.4.2.1 A still better performance guarantee for aproximate graph coloring

Halldórsson apresentou um algoritmo de aproximação para conjuntos independentes de um grafo em combinação com trabalhos anteriores que tinham alcançado tempo O(n(log log n/ log n)^3) para coloração mínima de grafo. O novo algoritmo alcançou tempo O(n(log log n)^2 / log^{3}n). O algoritmo criado é apresentado na Figura 12.

```
SampleIS (G, k)
{G is k-colorable, |G| = n}
begin
    if |G| ≤ 1 then return G
    forever do
        randomly pick a set I of log_k n nodes
        if I is independent then
            if |N̄(I)| ≥ n/k · log n/2 log log n then
                return (I ∪ SampleIS(N̄(I), k))
            else
                I_2 = CliqueRemoval(N̄(I)) ∪ I
                if |I_2| ≥ log^3 n/6 log log n then return (I ∪ I_2)
                {else I ⊈ A}
            endif
        endif
    od
end
```

Figura 12: Algoritmo de coloração 1
Fonte: Halldórsson (1993)

O funcionamento do algoritmo é descrito a seguir: A variável I é um conjunto de tamanho log k n de vértices independentes do grafo. A variável ¬(N(I)) contem os vértices do grafo original removendo os itens do conjunto I. A cada iteração é verificado se ¬(N(I)) é maior que (n/k. log n/2 log log n). Em caso positivo a mesma função é chamada novamente informando o grafo contido na variável ¬(N(I)) unida com I e o número de cores. Em caso negativo a função cliqueremoval vai encontrar um conjunto independente no grafo e unir com I na variável I2. Caso I2 seja maior que $\log^3$ n/6 log log n será retornado a união de I com I2.

Neste algoritmo é possível notar o uso de recursividade na função para encontrar os vértices independentes e o uso das variáveis definindo conjuntos específicos uteis para as funções

2.4.2.2 Graph Coloring Algorithms

(Klotz, 2002) avaliou algoritmos determinísticos de contração e sequenciais para encontrar o número cromático de um grafo. O algoritmo de contração, apresentado na Figura 13.

O funcionamento do algoritmo é especificado a seguir: Para cada vértice do grafo é escolhido um vértice de maior grau x e colorido. Então é definido um conjunto NN com os vértices que não são adjacentes de x. Então este conjunto é percorrido identificando todos os itens de que podem ser contraídos em x, faz a contração do vértice encontrado em x e atualiza o conjunto NN. No fim da busca em NN o x é removido de G e a operação repete. Diferente dos algoritmos anteriores O exemplo acima possui uma busca que contrai dois vértices em um e embora utilize conjuntos não faz utilização das funções padrão de conjuntos.

Uma característica interessante deste algoritmo é a utilização da contração de vários vértices em um e a descrição do código que varia entre uma linguagem de programação padrão e explicações em linguagem natural.

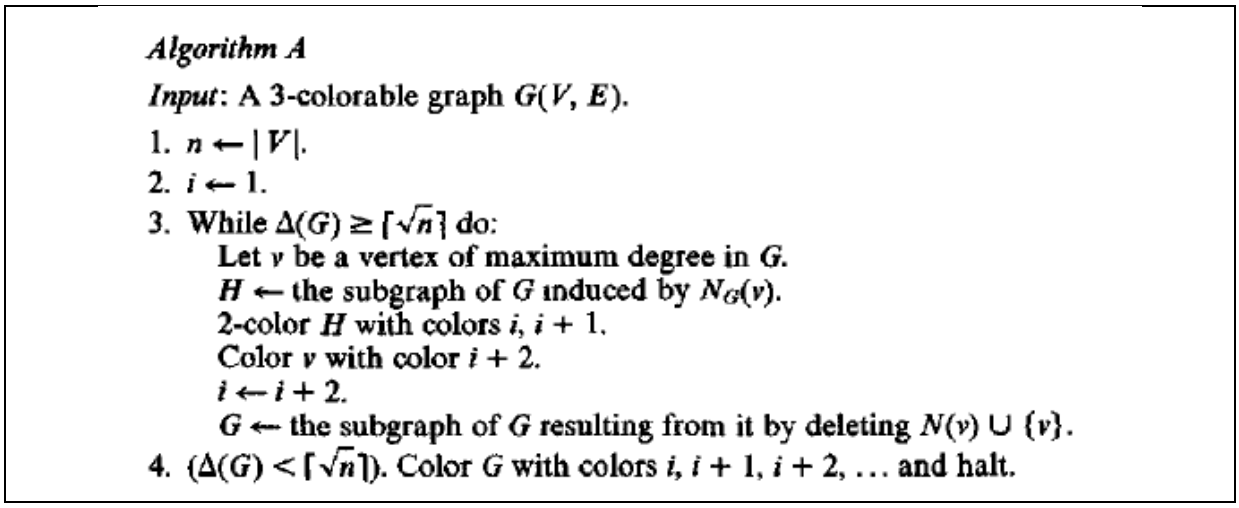
```
Given a graph G = (V, E) with vertex set V = V(G) and edge set E.
A coloring F : V ⟶ ℕ is established and will be returned.
colornumber = 0;                                   //number of used colors
while (|V(G)| > 0){
   determine a vertex x of maximal degree in G;
   colornumber = colornumber + 1;
   F(x) = colornumber;
   NN= set of non-neighbors of x;
   while (|NN| > 0){                  //find y ∈ NN to be contracted into x
      maxcn = −1;   //becomes the maximal number of common neighbors
      ydegree = −1;                                  //becomes degree(y)
      for every vertex z ∈ 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.
```

Figura 13. Algoritmo de coloração 2

Fonte: Klotz (2002)

2.4.2.3 Improving the Performance Guarantee for Approximate Graph Coloring

O trabalho de Wigderson foca na busca de um algoritmo mais eficiente que O(n/log n). Para chegar ao objetivo o autor apresenta alguns algoritmos, entre eles o algoritmo apresentado na Figura 14, para colorir grafos 3-cromatico.



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, $i + 1$, $i + 2$, ... and halt.

Figura 14: Algoritmo de coloração 3

Fonte: Wigderson (1983).

O funcionamento do algoritmo ocorre da seguinte forma: é criado um subgrafo H com a vizinhança do vértice v. Os vértices do subgrafo H são coloridos alternadamente com 2 cores, em seguida o vértice v é colorido com uma terceira cor. O grafo G recebe esta alteração, marca os vértices como coloridos e as operações repetem até o maior grau do grafo seja menor que a raiz quadrada dos itens não coloridos.

Algumas características importantes neste algoritmo são: utilização de funções pré definidas no artigo como Ng(v) que contem a vizinhança de um vértice v. A utilização da teoria de conjuntos para remover o conjunto de vértices não coloridos de G e adicionar os coloridos.

2.4.3 Clique Máximo

Alguns artigos relatando aplicações do problema do Clique Máximo foram publicados, a seguir serão apresentados alguns algoritmos criados pelos autores: Augustson (1970), Kreher (1999) e Carmo (2012).

2.4.3.1 An analysis of some graph theoretical cluster techniques

Augustson apresenta dois algoritmos para identificar o tipo de cluster. O foco dele são em clusters desenvolvidos para sistemas que retornam informação conhecidos como thesauri. Para isso são utilizados algoritmos que buscam o maior subgrafo completo. A Figura 15 apresenta o algoritmo criado.

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.

Figura 15. Algorimo Clique Máximo 1
Fonte: Augustson (1970)

O algoritmo funciona através de um laço que percorre os vértices do grafo e a cada vértice verifica suas adjacências guardando no conjunto C. Enquanto o algoritmo percorre as adjacências de M, algumas verificações são aplicadas para unir os itens em C ao final do laço entre os vértices, o algoritmo obtém o Clique Máximo dentro de C.

Entre as principais características do código esta o frequente uso da função goto line que é incomum na descrição deste tipo de algoritmo. Além disso, o uso da teoria de conjuntos de forma que a cada iteração de M faz testes e une novos itens a C que ao final do algoritmo se torna o Clique Máximo do grafo.

2.4.3.2 Discrete mathematics and its applications

(Kreher, 1999) apresenta no 4º capitulo de seu livro um algoritmo recursivo para obter o Clique Máximo do grafo. O algoritmo é apresentado na Figura 16.

Algorithm 4.14: MAXCLIQUE1 (ℓ)

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

Figura 16. Algoritmo Clique Máximo 2
Fonte: Kreher (1999)

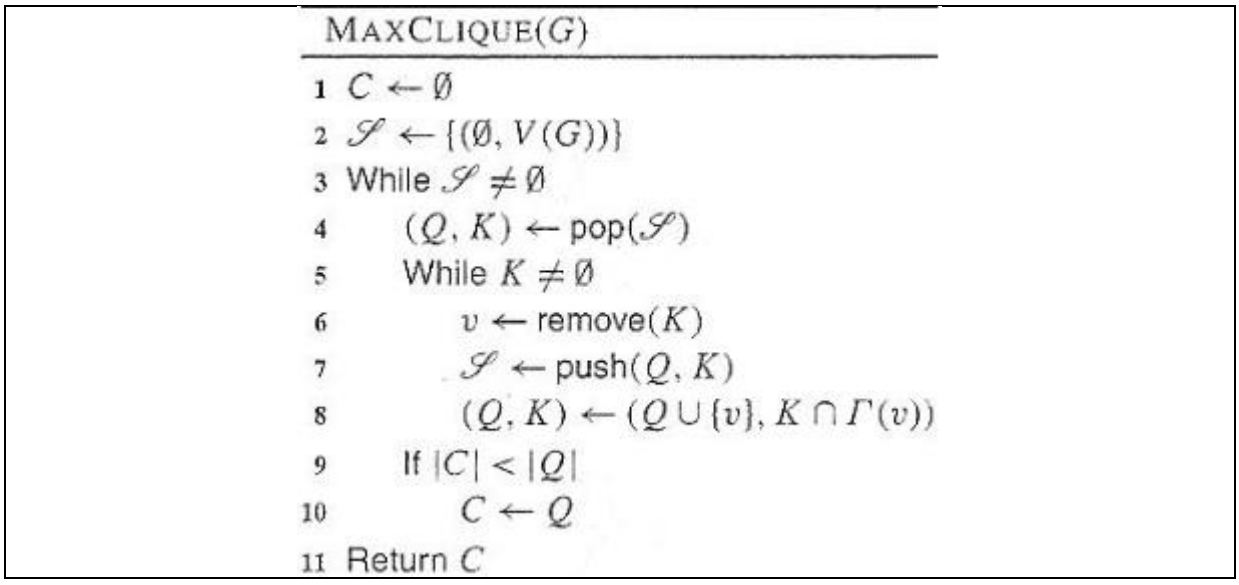
Este algoritmo funciona recursivamente, na primeira iteração a o conjunto C recebe todos os vértices do grafo, e o vetor xl recebe o primeiro vértice verificado VI e a função é novamente chamada. Nas próximas verificações a variável de controle optSize que guarda o tamanho do maior Clique e a variável l, que contem um contador do tamanho do Clique atual, são comparadas. Se l for maior que optSize, o valor de optSize aumenta e a variável optClique recebe o novo maior Clique. Após este passo são verificados os próximos vértices adjacentes de VI sendo que em cada iteração é verificado se o vértice atual possui adjacências com todos

os vértices adjacentes de VI e com o próprio VI. Estes passos são repetidos em todos os vértices do conjunto C, ao final do processo a variável optClique terá o Maximum Clique.

O algoritmo possui algumas características interessantes como a forma que os conjuntos A e B são utilizados, a cada iteração o A recebe as adjacências do ultimo vértice avaliado e B os próximos vértices a serem avaliados do ultimo vértice. Além disso o uso de função recursiva garantiu um código bem limpo e definido em poucas linhas

2.4.3.3 Branch and bound algorithms for the maximum clique problem

Carmo verificou alguns algoritmos para a solução exata do problema do Clique Máximo e descreveu resultados comparativos de oito algoritmos. A Figura 17 apresenta um destes algoritmos.



MAXCLIQUE(G)

```
1  C ← ∅
2  𝒮 ← {(∅, V(G))}
3  While 𝒮 ≠ ∅
4      (Q, K) ← pop(𝒮)
5      While K ≠ ∅
6          v ← remove(K)
7          𝒮 ← push(Q, K)
8          (Q, K) ← (Q ∪ {v}, K ∩ Γ(v))
9      If |C| < |Q|
10         C ← Q
11 Return C
```

Figura 17. Algoritmo Clique Máximo 3
Fonte: Carmo (2012).

O algoritmo funciona através de dois laços de repetição, o laço externo controla os vértices que serão avaliados. No laço interno que percorre todos os vértices é definido um vértice inicial e verificado seu Clique durante as iterações, a cada iteração é feito um teste para ver se o Clique atual superou o ultimo Clique Máximo. No fim do processo do laço interno o Clique do vértice escolhido no laço externo é definido, e o restante dos vértices é avaliado, até que todo o grafo tenha sido pesquisado.

Um ponto interessante deste algoritmo é o controle feito utilizando a pilha e a execução baseada em teoria de conjuntos.

2.4.4 Considerações sobre Algoritmos

Embora os nove algoritmos apresentados tenham sido feitos em diferentes datas por variados autores, algumas características se repetem em vários deles e serão apresentadas junto com o número de vezes em que foram encontradas nos diferentes algoritmos.

A Tabela 1 abaixo mostra na coluna da esquerda as características identificadas e na coluna da direita o número de vezes que cada característica apareceu nos algoritmos.

Tabela 1. Características dos algoritmos

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

Algumas das características aparecem em todos os algoritmos o que mostra que são uma forma padrão de trabalho em grafos. As características que apareceram menos vezes são normalmente variáveis utilizadas para fazer algum controle no código ou no caso da ultima característica citada, para resumir algumas informações previamente descritas no artigo, por exemplo, as adjacências de um vértice.

Embora existam características que se repetem nos diferentes algoritmos, a sintaxe definida por cada autor para descrever os algoritmos são diferentes. Além disso, os algoritmos foram escritos sem a preocupação de serem executados em um computador, por isso não são feitas declarações de variáveis, definição dos valores de conjuntos entre outras características importantes em uma linguagem de programação.

Os algoritmos foram buscados em artigos que abordavam temas específicos que utilizam a teoria de grafos e as gramaticas desenvolvidas foram criadas para dar suporte somente aos problemas abordados, com isso cada autor criou uma sintaxe diferente com o escopo especifico para atender o problema estudado no artigo. Todos os artigos estudados possuíam definições no decorrer do artigo que foram posteriormente utilizadas nos algoritmos. Esse modo de abordagem que utiliza o texto do artigo para especificar funções

dos algoritmos abstrai algumas especificações que em uma linguagem de programação precisariam ter sido especificadas.

Cada algoritmo possui sintaxes diferentes adequadas as necessidades do algoritmo, o que simplifica a memorização dos termos. Porém, como são muito diferentes geram a necessidade de compreender a forma que cada autor encontrou para expressar o algoritmo.

Os dados observados nos algoritmos serão avaliados durante o projeto da DSL, visando criar uma linguagem que possa resolver diferentes problemas de grafos com um formato que seja adequado para os tipos de operações que são normalmente utilizados em grafos.

3 DESENVOLVIMENTO

A implementação da linguagem seguiu um método iterativo, ou seja, partindo de um protótipo inicial, que idealizava como deveria ser a linguagem, foram criadas versões implementando pedaços da linguagem e esses pedaços eram alterados caso durante o uso fosse identificado que algo mais adequado ao contexto da linguagem poderia substitui-lo.

O conceito de implementar a linguagem iterativamente é citado no livro *Domain-Specific Language* de Martin Fowler. Com base nesse conceito a linguagem evoluiu do que foi idealizado inicialmente, disponível no apêndice A.1, para a versão final, conforme apresentado no exemplo do algoritmo de busca em profundidade, disponível no apêndice A.2.

Neste capitulo é documentado o processo de desenvolvimento da linguagem, apresentando as ferramentas e padrões de projeto utilizados. Está dividida em Requisitos Funcionais, linguagem, implementação e exemplos.

3.1 Requisitos Funcionais

Os requisitos funcionais (RF) são responsáveis por apresentar as funcionalidades que a linguagem e a IDE deverão contemplar e serão exibidos abaixo:

3.1.1 Requisitos da linguagem

Os requisitos funcionais da linguagem serão apresentados abaixo.

- RF01: A linguagem deverá permitir ao usuário resolver problemas de grafos através da sintaxe definida;
- RF02: A linguagem deverá implementar operações de manipulação de conjuntos;
- RF03: A linguagem deverá implementar as estratégias de busca em grafo: busca em largura e busca em profundidade.
- RF04: A linguagem deverá permitir a implementação de novas estratégias de busca em grafo.
- RF05: A linguagem deverá permitir criação e utilização de funções.

- RF06: A linguagem deverá permitir recursividade em suas funções.
- RF07: A linguagem deverá permitir a criação de variáveis do tipo: booleano, número, texto, conjunto e grafo.
- RF08: A linguagem deve ter uma função para escrever texto na tela como saída de informações do programa para o usuário.

3.1.2 Requisitos da IDE

Os requisitos funcionais da IDE serão apresentados abaixo.

- RF01: A IDE deverá ser capaz de executar algoritmos na linguagem que esta sendo definida neste projeto.
- RF02: A IDE deverá produzir uma saída em modo texto quando a função de escrita da linguagem for chamada.
- RF03: A IDE deverá ser capaz de abrir, editar e salvar os arquivos produzidos por ela.
- RF04: A IDE deverá possuir um salientador de sintaxe.

3.2 Linguagem

Essa seção apresenta as estruturas, funcionamento e os tipos da linguagem.

3.2.1 Estrutura da linguagem

Os comandos da linguagem foram projetados para atender os aspectos citados na introdução deste documento como importantes na produção de uma DSL. Os aspectos são: Ser de fácil entendimento e executável pelo computador, ter um senso de fluência, expressividade limitada ao contexto.

A linguagem é orientada por identação, ou seja, o inicio e fim de um contexto é baseado na tabulação do código. A Figura 18 apresenta essa característica.

```
1  exemplo = "Ola Mundo"
2  se exemplo == "Ola Mundo"
3       mostrar exemplo
4  mostrar "Fim"
```

Figura 18. Olá mundo em dslfg

Além da orientação por identação na Figura 18 pode ser notado à ausência da necessidade de uma função principal. Os comandos são compilados e executados no sentido de cima para baixo, sem a necessidade de pertencer a nenhuma função ou estrutura especial. Outro aspecto importante é que não existe um caractere visível no final das instruções, a linguagem reconhece o fim de uma instrução com a quebra de linha.

Os próximos subitens da seção irão apresentar os principais comandos da linguagem e explicar o funcionamento.

3.2.1.1 Construtor de grafos

O construtor de grafos foi planejado com o objetivo de criar as conexões do grafo e gerar os vértices no mesmo passo. O comando é apresentado na Figura 19.

```
1  digrafo exemploSemPeso
2       a ligadoCom b
3       a ligadoCom c
4       b ligadoCom c
5       b ligadoCom d
6       c ligadoCom b
7       c ligadoCom d
8       c ligadoCom e
9       d ligadoCom e
10      e ligadoCom a
11      e ligadoCom d
12
13 grafo exemploSemPeso
14      a ligadoCom b
15      a ligadoCom c
16      b ligadoCom c
17      b ligadoCom d
18      c ligadoCom d
19      c ligadoCom e
20      d ligadoCom e
21      e ligadoCom a
```

```
1  digrafo exemploComPeso
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       c ligadoCom e comPeso 6
9       d ligadoCom e comPeso 6
10      e ligadoCom a comPeso 7
11      e ligadoCom d comPeso 4
12
13 grafo exemploComPeso
14      a ligadoCom b comPeso 10
15      a ligadoCom c comPeso 5
16      b ligadoCom c comPeso 3
17      b ligadoCom d comPeso 1
18      c ligadoCom d comPeso 2
19      c ligadoCom e comPeso 6
20      d ligadoCom e comPeso 6
21      e ligadoCom a comPeso 7
```

Figura 19. Criação de Grafo

O comando reconhece os identificadores dos vértices e os gera no mesmo processo que o liga com o segundo vértice. Duas outras características podem ser notadas na Figura 19, a distinção de grafo para dígrafo e a possibilidade de adicionar o peso de um vértice. Além do formato apresentado, também é possível adicionar propriedades nos vértices. A Figura 20 exemplifica a sintaxe.

```
1  digrafo exemploComPeso
2      a
3          com letra = "a"
4          com valor = 5
5          ligadoCom b
6          com letra = "b"
7          com valor = 6
8          comPeso 10
9      a ligadoCom c comPeso 5
10     b ligadoCom c comPeso 3
11     b ligadoCom d comPeso 1
12     c ligadoCom b comPeso 2
13     c ligadoCom d comPeso 2
```

Figura 20. Grafo com propriedades.

A cláusula “com” é chamada logo após o identificador do vértice e adiciona todas as propriedades citadas, dentro do vértice.

A segunda forma para adicionar propriedades é com um comando após o grafo já estar criado. O comando definir propriedade solicita o nome da propriedade, o vértice em que será adicionado e o valor, com essas informações adiciona a nova propriedade no vértice. A instrução é apresentada na linha nove da Figura 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
```

Figura 21. Definir propriedade em vértice

3.2.1.2 Criação de conjuntos

Conforme identificado no capitulo 2, algoritmos de grafo normalmente utilizam operações de conjunto para resolver seus problemas. Para atender essa necessidade a linguagem possui instruções para criar e operar conjuntos. A Figura 22 mostra a criação de um conjunto seguido pela adição de um elemento.

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

Figura 22. Criação de conjunto

A criação de conjunto é feito pelos caracteres de abrir e fechar chave e a inserção de elementos é feito entre as chaves, separados por vírgula. Após a criação de um conjunto é possível adicionar novos elementos utilizando a instrução apresentado na segunda linha da Figura 22.

Além desses comandos foram implementados outros para ser possível a devida manipulação dos conjuntos. Os demais comandos necessários para a manipulação de conjunto serão apresentados nas próximas subseções.

3.2.1.3 Funções da linguagem

Para possibilitar a manipulação das estruturas de grafo e conjunto, foram criados comandos especiais que permitem obter e manipular os dados dessas estruturas. As doze funções criadas para esse fim serão apresentadas a seguir.

3.2.1.3.1 Obter um elemento de conjunto

Foram criadas duas formas de obter elementos de um conjunto, uma aleatória e outra ordenada. A forma aleatória busca qualquer elemento dentro de um conjunto especificado e retorna o resultado para a instrução em que foi chamada. A Figura 23 apresenta a sintaxe da instrução.

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

Figura 23. Obter elemento de conjunto aleatoriamente.

A segunda forma é divida em três passos, primeiramente é obtido o iterador do conjunto, em seguida é questionado se existem mais elementos não visitados, no último passo o próximo

elemento é obtido do iterador. A Figura 24 ilustra a criação de um conjunto, obtenção do iterador, verificação de próximo elemento e aquisição do elemento seguinte.

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

Figura 24. Obter elemento de conjunto com iterador.

O comando da terceira linha irá obter um valor do tipo booleano que pode ser testado num desvio condicional ou laço de repetição. Além dessas funções foram implementadas quatro operações da teoria de conjuntos que retornam novos conjuntos. Essas operações serão citadas na subseção de operadores.

3.2.1.3.2 Obter um vértice de um grafo

É possível obter um vértice de um grafo de duas formas, especificando o vértice desejado ou pegando qualquer um. Os dois comandos são apresentados na Figura 25. Na sexta linha o comando retornara um vértice aleatório do grafo e na sétima o vértice "*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
```

Figura 25. Obter um vértice.

3.2.1.3.3 Obter vértices adjacentes

A função para obter os vértices adjacentes necessita da seleção de um vértice raiz, ou seja, após extrair um vértice de um grafo o comando para obter vértices adjacentes recebe esse parâmetro e com essa informação retorna um conjunto contendo os vértices adjacentes do vértice selecionado. A Figura 26 apresenta a utilização do comando.

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

Figura 26. Obter vértices Adjacentes

3.2.1.3.4 Obter peso da aresta

A função para obter o peso de uma aresta necessita que seja informado o vértice de origem e destino, com essas informações o comando retorna o peso da aresta entre os vértices. A Figura 27 apresenta a utilização do comando, nesse caso o comando irá retornar o valor cinco.

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

Figura 27. Obter peso da aresta.

3.2.1.3.5 Obter Propriedade de vértice

Pra obter a propriedade desejada de um vértice foi criada uma função que precisa receber o vértice desejado e o nome da propriedade requerida, para então retornar o valor da propriedade. A Figura 28 apresenta o uso da função na linha sete.

```
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
```

Figura 28. Obter Propriedade de vértice.

3.2.1.3.6 Obter todos os vértices e arestas de um grafo

A função para obter todos os vértices de um grafo retorna um conjunto contendo todos os vértices do grafo solicitado. A sintaxe do comando é apresentada na linha quatro da Figura 29. Na quinta linha da mesma figura é apresentada a função para obter todas as arestas do grafo, retornando um conjunto com todas as conexões grafo.

```
1  grafo grafoExemplo
2       Navegantes ligadoCom Itajai
3       Itajai ligadoCom BalnearioCamboriu
4  vertices = obterTodosOsVerticesDe grafoExemplo
5  arestas = obterTodasAsArestasDe grafoExemplo
```

Figura 29. Obter todos os vértices e arestas de um grafo.

3.2.1.3.7 Copiar grafo

A função de cópia retorna uma cópia do grafo passado, diferente do que ocorre numa atribuição normal onde é passada a referência. A Figura 30 apresenta o uso do comando

```
1  grafo grafoExemplo
2       Navegantes ligadoCom Itajai
3       Itajai ligadoCom BalnearioCamboriu
4
5  grafoIdentico = copiarGrafo grafoExemplo
```

Figura 30. Copiar grafo

3.2.1.4 Operadores

Os operadores são funções responsáveis por operações aritméticas e lógicas entre objetos da linguagem. Os quadros a seguir irão apresentar todos os operadores existentes na linguagem.

O primeiro será o Quadro 1 que irá apresentar os operadores aritméticos. Na coluna operador será apresentado o token do operador e na coluna descrição será explicado a funcionalidade dele dentro da linguagem.

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

Quadro 1. Quadro de operadores aritméticos

O segundo será o Quadro 2 que irá apresentar os operadores de conjunto. Na coluna operador será apresentado o token utilizado na linguagem, na coluna símbolo padrão estará o símbolo utilizado na literatura para a operação e na coluna descrição será explicado a funcionalidade do operador.

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.

Quadro 2. Operadores de conjunto

O Quadro 3 irá apresentar os operadores lógicos da linguagem. Na coluna operador será apresentado o token do operador e na coluna descrição será explicado a funcionalidade dele dentro da linguagem.

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

Quadro 3. Operadores lógicos

3.2.1.5 Comando Mostrar

O comando mostrar foi criado para que o usuário da linguagem pudesse apresentar o resultado de seus algoritmos. A instrução precisa de um parâmetro que contém o valor que será apresentado, esse valor pode ser um texto, número, booleano, constante da linguagem ou uma variável. A Figura 31 apresenta o uso da instrução.

```
1 variavel = 5
2 mostrar variavel
3 mostrar "Fim"
```

Figura 31. Comando mostrar

3.2.1.6 Desvio condicional

O comando “se” foi criado para possibilitar a execução de comandos somente após verificar o cumprimento de uma condição pré-determinada. A condição é formada por um objeto de qualquer tipo da linguagem à esquerda, um operador lógico e a direita outro objeto da linguagem. Caso a condição retorne valor verdadeiro os comandos adicionados no contexto do “se”, ou seja, estão identados corretamente, serão executados. A Figura 32 apresenta o uso do comando.

```
1  variavel = 5
2  se variavel == 5
3       mostrar "Dentro do desvio condicional"
4  mostrar "Fora do desvio condicional"
5
6  conjunto = { 1, 2, 5}
7  se variavel eUmElementoDe conjunto
8       mostrar "Dentro do desvio condicional"
9  mostrar "Fora do desvio condicional"
```

Figura 32. Desvio condicional

3.2.1.7 Laço de repetição

O comando “enquanto” foi criado para possibilitar a execução de comandos enquanto condição pré-determinada for verdadeira. A condição tem a mesma formação que a do comando “se”. Caso a condição retorne valor verdadeiro os comandos adicionados no contexto do “enquanto” serão executados e repetidos até que a condição retorne falso. A Figura 33 apresenta o comando.

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

Figura 33. Laço de repetição

3.2.1.8 Função de usuário

Funções de usuário foram criadas para que seja possível criar trechos de códigos fora do contexto principal do programa e chama-los quando necessário, evitando a repetição de código e facilitando a leitura do código (Martin, 2008). A linguagem incorporou essa funcionalidade e também possibilitou a criação de bibliotecas com as funções criadas pelo usuário através do comando importar, que será apresentado na seção “comando importar”.

A Figura 34 apresenta a declaração de função, chamada e obtenção de resultado a partir dela.

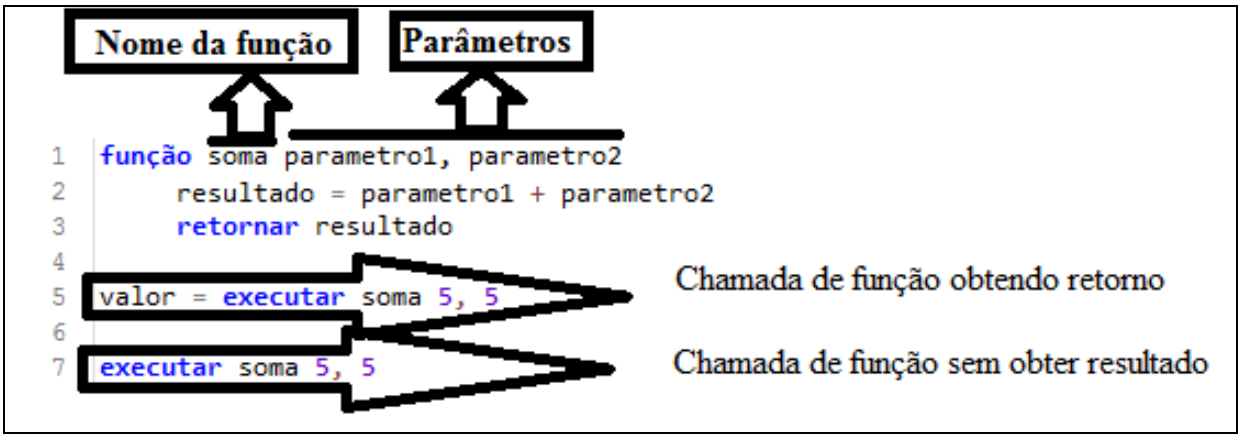


Figura 34. Função de usuário

A Figura 34 também apresenta os comandos para retornar um valor em uma função e as chamadas de função.

O comando retornar serve para que após a execução de uma função o programador consiga informar ao usuário o resultado da execução da mesma. A instrução retornar é apresentada na linha três da Figura 34. Nas linhas cinco e sete da mesma figura aparecem as chamadas de função. Além de informar o nome da função a chamada passa os parâmetros utilizados separando-os por vírgula.

3.2.1.9 Percorrer grafo

O comando foi criado para permitir a separação entre o algoritmo que percorre o grafo e o que é feito em cada iteração. O comando precisa de uma função que conterá a estratégia de busca para percorrer o grafo, o grafo que será percorrido e o vértice em que a busca irá iniciar. A Figura 35 apresenta o uso da funcionalidade.

```
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
```

Figura 35. Comando percorrer grafo

Na linha dezoito da Figura 35 o comando é chamado e na dezenove é adicionado o comando mostrar em seu contexto. A função chamada por essa instrução precisa atender a dois requisitos: possuir três parâmetros, e em algum momento chamar a instrução que executa os comandos que estão no contexto de percorrer grafo. A instrução para executar esses comandos está na linha sete, ele recebe o terceiro parâmetro da função e fornece o que o programador considerar necessário para o contexto de percorrer grafo, no exemplo da Figura 35 foi a variável vértice.

A execução da instrução percorrer grafo e da função chamada ocorrem simultaneamente. A função recebe como primeiro parâmetro o grafo informado pela instrução percorrer grafo logo após o primeiro token, em seguida o vértice inicial que é informado logo após o token “iniciadoEm”, e o terceiro parâmetro recebe as instruções do contexto de execução de percorrer grafo. Além desse fluxo de informação a função também consegue informar dados para o contexto de percorrer grafo, através dos parâmetros informados após o token “fornecendo”. A Figura 36 ilustra esse fluxo.

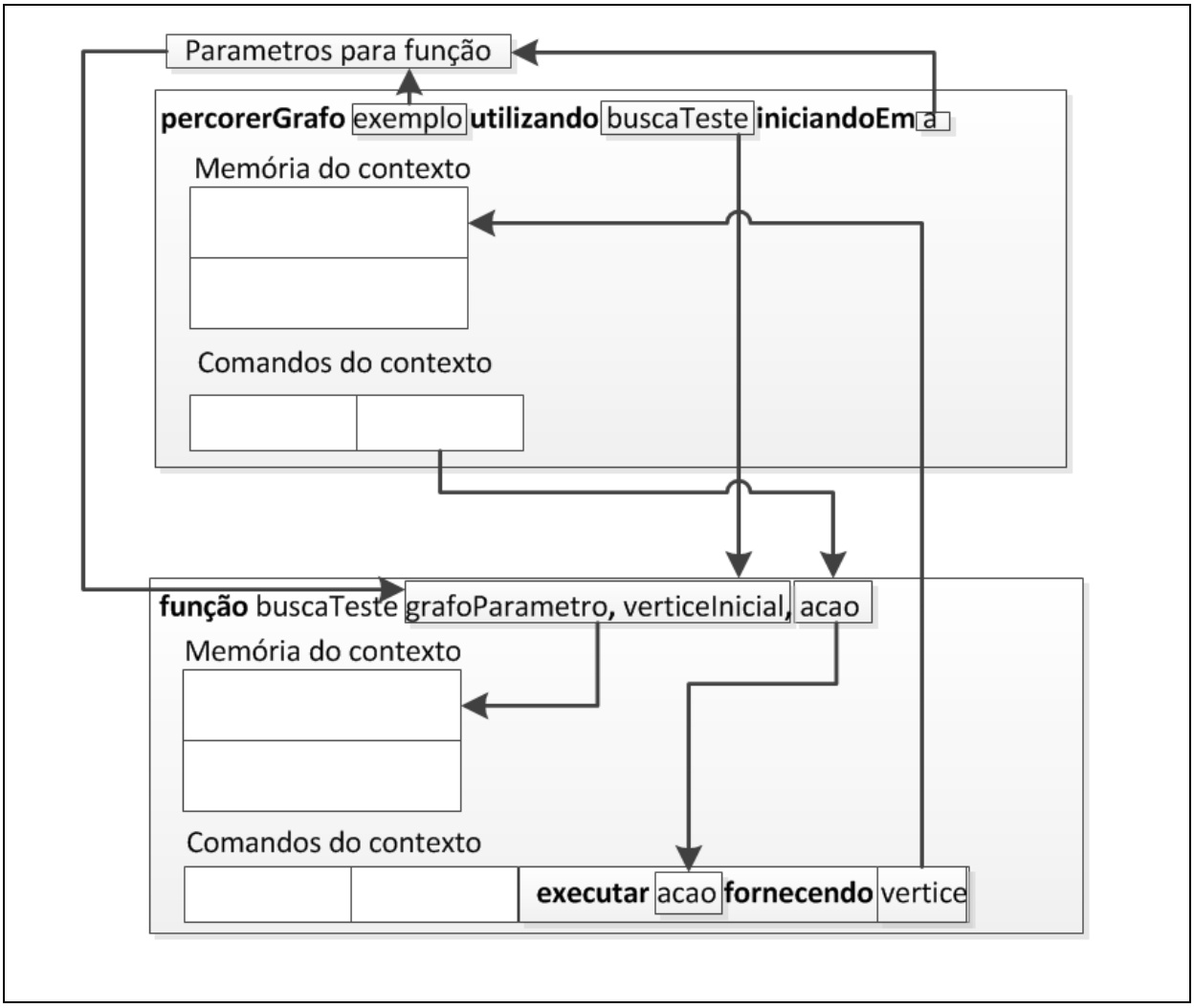


Figura 36. Fluxo da instrução percorrer grafo

3.2.1.10 Comando importar

Esse comando permite a criação e importação de bibliotecas na linguagem. A partir do caminho do arquivo que é passado para ele, todas as funções existentes no arquivo citado são importadas para o programa em construção, caso haja funções com o mesmo nome, será considerada a função do arquivo atual e não do importado. A Figura 37 apresenta o uso do comando.

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

Figura 37. Comando importar

3.2.2 Tipos Da Linguagem

Essa subseção irá apresentar os tipos existentes na linguagem. O Quadro 4 apresenta os tipos na coluna “tipo” e descreve seu objetivo na coluna “descrição”.

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

Quadro 4. Tipos da linguagem

A linguagem é dinamicamente tipada, portanto quando se cria uma variável não é necessário informar o tipo, somente atribuir o valor. Os tipos apresentados no Quadro 4 servem somente para expressar os valores aceitos na linguagem.

Foram implementadas constantes na linguagem que tem o objetivo de fornecer algumas utilidades para o usuário. O Quadro 5 apresenta as constantes criadas na coluna constante e apresenta o objetivo na coluna descrição. As constantes podem ser usadas normalmente como um objeto do tipo numero em operações aritméticas ou nas funções padrão da linguagem.

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

Quadro 5. Constantes da linguagem

3.3 Implementação

Esta seção descreve como a linguagem foi implementada, apresentando os padrões e ferramentas utilizados.

A arquitetura foi divida em três partes: interface do usuário, interpretação e execução. Interface do usuário é a parte responsável por obter dados do usuário e apresentar respostas da execução dos programas criados, também possui funcionalidades para auxiliar o usuário no uso da linguagem. A parte da interpretação é responsável pela descrição das regras da gramática e transformação dos programas criados pelo usuário dentro da sintaxe definida na gramatica em objetos executáveis. A execução utiliza os objetos criados na fase de interpretação para finalmente executar o código descrito pelo usuário.

A Figura 38 representa a interação entre os três níveis da arquitetura.

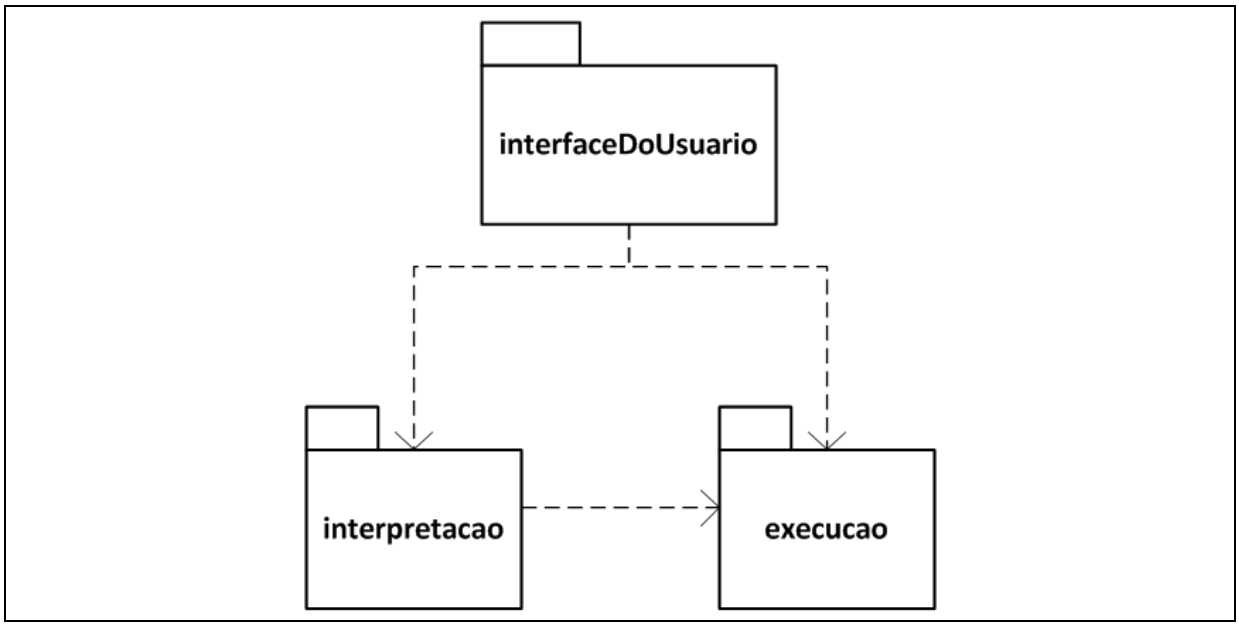


Figura 38. Interação entre os níveis da arquitetura

3.3.1 Interpretação

A fase de intepretação é responsável por transformar o código do usuário em uma coleção de objetos executáveis. Esta dividida em três partes:

- Ouvinte: A classe “OuvinteDSLFG” é responsável pela leitura dos eventos da gramatica e criação dos construtores. Além disso, é a interface entre o pacote “interfaceDoUsuario” e “interpretacao”;
- Gramatica: Pacote que contém as regras da gramatica produzidos no ANTLR e as classes geradas por ele em JAVA;
- Construtores: Pacote que contém todos os construtores de primitivas (executáveis) da linguagem.

A Figura 39 representa a interação dos elementos do pacote “interpretação”.

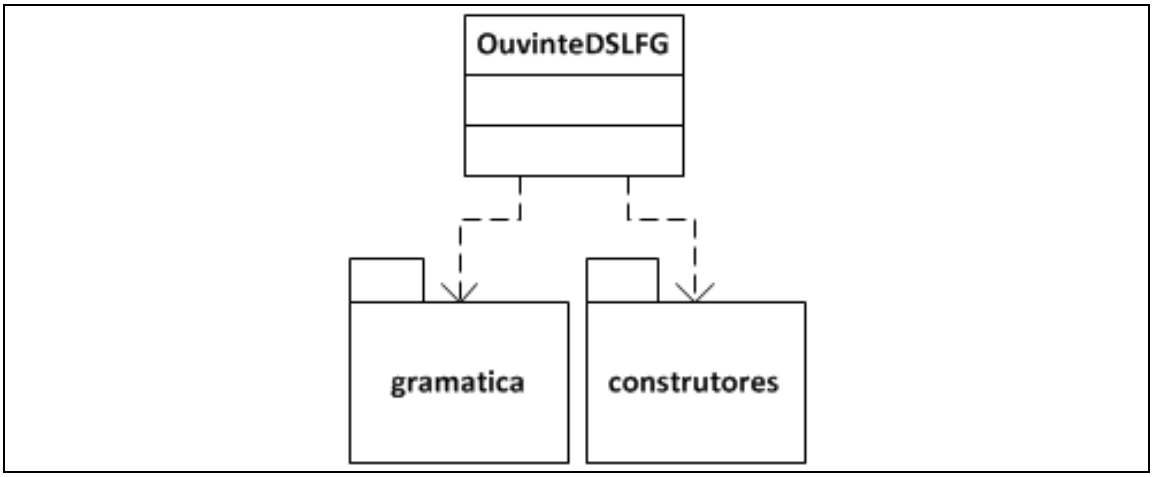


Figura 39. Interação dos objetos do pacote “interpretação”

3.3.1.1 Gramática

Para implementar uma linguagem, é necessário criar uma aplicação que lê sentenças e reage apropriadamente as frases e símbolos de entrada descobertos (Parr, 2012).O pacote gramatica é responsável pela formação da sintaxe da linguagem e da transformação das regras criadas no formato do ANTLR em classes java.

O ANTLR é uma ferramenta utilizada no projeto que foi desenvolvida por Terence Parr, um professor de ciência da computação da universidade de São Francisco localizada na Califórnia, EUA. A ferramenta é um gerador de analisador sintático e é amplamente utilizada

para a construção de linguagens. A partir de uma gramatica ele gera um analisador que pode construir e andar em arvores sintáticas.

A gramática criada dentro do padrão do ANTLR é apresentada integralmente nos apêndices A.3e A.4. A partir desses arquivos da gramatica são criadas cinco classes java entre elas a ‘DSLFGBaseListener’ que é a responsável entre a comunicação das classes do ANTLR e as classes criadas pelo programador.

A classe ‘Ouvinte’ estende ‘DSLFGBaseListener’ e recebe os eventos da arvore sintática abstrata. A cada evento reconhecido é criado um construtor cuja funcionalidade será apresentada na seção Construtores.

3.3.1.2 Construtores

A classe ‘Ouvinte’ possui métodos que são chamadas na entrada e saída das folhas da árvore sintática. E são nesses métodos que os construtores são instanciados e em seguida adicionados na lista comandos.

Os construtores são invólucros dos comandos da gramatica. Eles pegam toda a informação necessário de cada comando e instancia um objeto com essas informações enquanto executa o processo de interpretação. Cada comando tem sua própria classe de construtor, porém, todos os construtores de comandos implementam uma interface que possui o método ‘obterComando’ esse método é responsável pela criação da primitiva que será executada em tempo de execução.

Além dos construtores de primitiva padrão que obtém todos os dados necessários de uma linha para posterior criação do executável existem os construtores de funções. Eles implementam uma interface diferente e no lugar do método ‘obterComando’ possuem o método ‘obterFuncao’ que é adicionado dentro do construtor de primitiva chamada ‘atribuicaoPorExpressão’.

Com base nesses dois tipos de construtores todas as instruções da linguagem são carregadas em uma lista para posterior transformação em objeto executável. Essa lista fica num tipo especial de construtor de comando que estende uma classe abstrata responsável pela adição dos comandos em seus devidos contextos.

Um construtor de contexto de execução possui uma lista de comandos e método para que instancie o contexto de execução que irá executar os comandos dessa lista. Cada instrução que possui seu próprio contexto, como por exemplo o desvio condicional ‘se’, tem seu próprio construtor. Todo o controle que define a que contexto cada comando pertence está implementado dentro da classe abstrata ‘construtorDeContextoDeExecucaoAbstrato’ e é baseado na identação do código.

O construtor de contexto de execução principal fica na classe ‘ouvinte’ e recebe um novo comando a cada evento lançado pela gramatica. No final do processo de interpretação esse construtor de contexto instancia o contexto de execução principal para então iniciar o processo de execução dos comandos.

3.3.2 Execução

A fase de execução é iniciada logo após o fim da interpretação, quando o ouvinte finaliza a sua execução retornando o contexto de execução com comandos executáveis. Essa seção irá apresentar as principais estruturas presentes na fase de execução e explicar o funcionamento.

3.3.2.1 Primitivas

Primitiva é um comando executável da linguagem, são classes do sub-pacote primitivas que tem o poder de alterar o contexto de execução ou apresentar mensagens pro usuário.

Todas as primitivas implementam a interface ‘Executavel’ que possui um método chamado ‘executar’ e recebe como parâmetro um contexto de execução. Cada comando da gramatica possui uma primitiva equivalente e essa implementa o método ‘executar’ conforme necessário pelo comando.

A primitiva ‘atribuicaoPorExpressao’ necessita de um segundo tipo de item executável como complemento para sua execução. Esse item é uma classe que implementa a interface ‘Funcao’. Essa interface é implementada por todas as funções da linguagem e operadores. Ela possui um método próprio chamado ‘obterResultado’ além de estender a interface ‘Executavel’ e portanto também possui o método ‘executar’. Dessa forma as funções quando executadas produzem um resultado que ficam disponíveis através do método

‘obterResultado’, no caso de ‘atribuicaoPorExpressao’, a função ou operação é resolvida e então o resultado obtido e gravado na memória.

Todos os operadores citados na subseção operadores da seção linguagem foram implementados utilizando a interface ‘Funcao’. Dessa forma além da primitiva ‘atribuicaoPorExpressao’ as condições verificadas em desvios condicionais e laços de repetição também seguem esse padrão.

3.3.2.2 Tratamento de erros

O tratamento de erros das primitivas foi feito com base no padrão de projeto decorator. Com esse padrão o tratamento do erro e a execução do comando são separados em duas classes separadas, mantendo o código responsável pela execução conciso, limpo e de fácil leitura. Itens importantes para a fase de manutenção de um sistema (Martin, 2008).

Cada primitiva no sistema possui um decorador correspondente. Todo decorador segue uma estrutura similar a da primitiva, implementando a interface executável, porém no lugar dos campos da primitiva ele possui a primitiva correspondente.

O método ‘executar’ do decorador avalia todos os possíveis erros na primitiva, como exemplo, são avaliados a existência da variável, os tipos envolvidos e existência de funções de usuário. Ao final do processo de avaliação em caso de ausência de erros, o decorador executa o método executar de sua primitiva com todas as possibilidades de erro previstas eliminadas, caso contrário uma exceção é lançada . A Figura 40 apresenta o funcionamento do processo de detecção de erros.

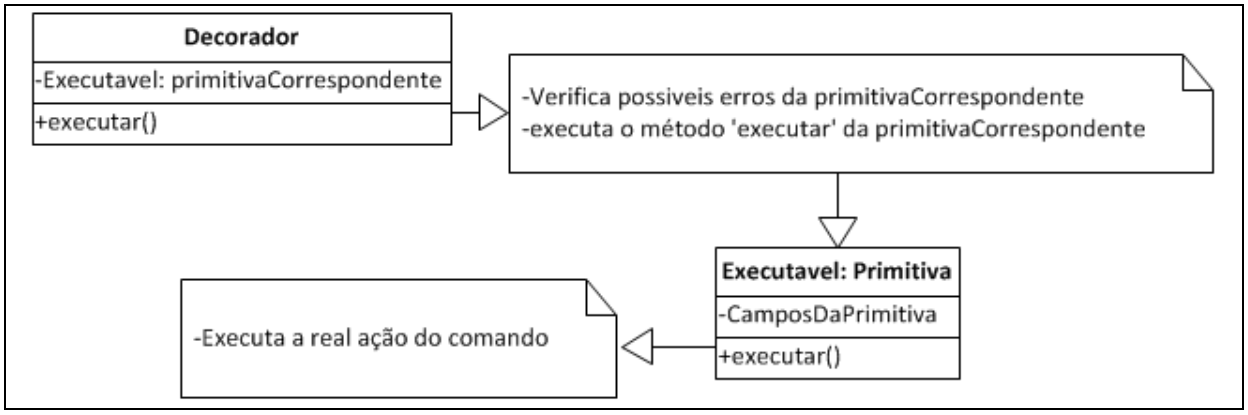


Figura 40. Funcionamento do decorador

3.3.2.3 Hierarquia de escopo

O contexto de execução é o ambiente responsável pelo controle do escopo, onde as primitivas são guardadas e executadas. É responsável pela execução das primitivas através da implementação da interface ‘Executável’ e pela gravação ou alteração das variáveis geradas pelas primitivas.

Para todo comando que exige tabulação existe um contexto de execução. Embora sejam diferentes todos possuem varias características em comum, pois são classes estendidas da classe ‘ContextoDeExecucaoAbstrato’ que implementa todos os métodos comuns utilizados pelos contextos, entre eles métodos responsáveis por adicionar variável, adicionar variável em despejo, obter variável e obter função.

Além dos métodos do contexto de execução todas as especializações do contexto abstrato são também executáveis. A característica comum da execução dos contextos é que executam a lista de comandos que possuem. A particularidade está na forma que executam. Por exemplo, o comando se avalia uma condição antes de executar os comandos, o comando enquanto avalia a condição e repete as instruções até que a condição seja falsa e o contexto simples, contentor dos comandos do contexto inicial, simplesmente executa a lista de executáveis. A Figura 41 apresenta o relacionamento entre as classes citadas.

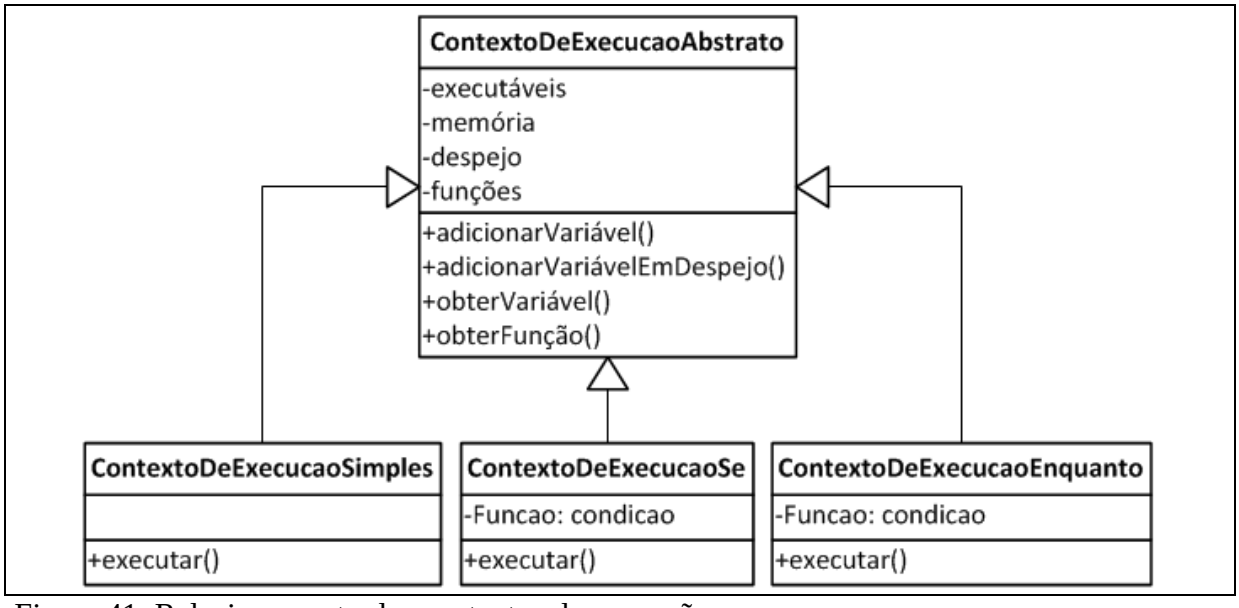


Figura 41. Relacionamento dos contextos de execução

Com múltiplos contextos de execução se torna necessário fazer um controle de memória que os comporte devidamente. Para fazer esse controle foi adicionado um mapa em cada contexto de execução contendo a memória daquele contexto. Porém, foram necessários

fazer alguns controles para se tornar possível a visualização e utilização das variáveis nos contextos anteriores. A Figura 42 ilustra como é feito a alteração do valor de uma variável, apresentando a direita da imagem o código que gera o novo contexto de execução.

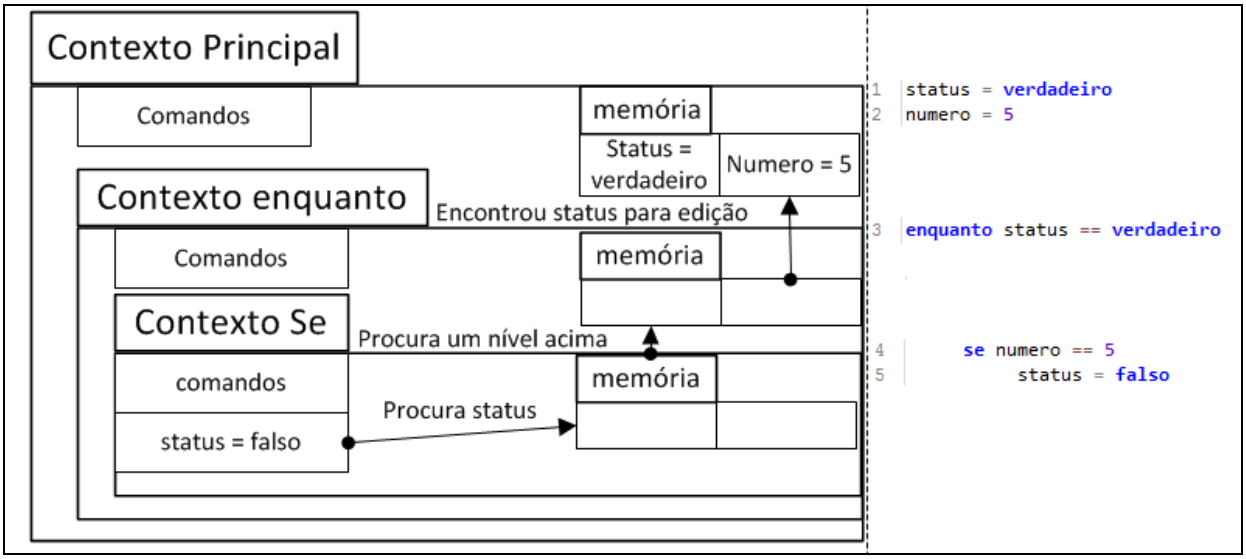


Figura 42. Edição de valor de variável

A adição e busca de variáveis utiliza o processo ilustrado na Figura 42. O valor é buscado no mapa de memória do contexto atual e iterativamente vai verificando os contextos anteriores até que encontre a variável buscada ou chegue ao contexto principal. A adição e alteração de uma variável utilizam o mesmo método, quando o processo descrito acima encontra uma variável com o nome determinado executa a alteração do valor, caso contrario retorna ao contexto de partida e adiciona lá uma nova variável.

As funções de usuário possuem uma precedência de memória diferente. Quando invocadas pela primitiva chamada de função, são pegas de um mapa de funções e instanciadas, recebendo na memória seus parâmetros e o contexto de execução principal como contexto anterior.

3.3.2.3.1 Contexto de execução da instrução Percorrer o grafo

A instrução percorrer o grafo possui seu próprio contexto de execução, porém sua execução é parametrizada e adiada para ser executada somente quando a primitiva ‘executarAcaoDePercorrerOGrafo’ é chamada.

A execução dessa instrução possui três processos chave:

1. Encapsulamento do contexto em um objeto que também faz papel de contexto de execução e executável. Será chamado de ação para simplificar a explicação;
2. Preparação dos parâmetros da instrução;
3. Chamada da função que contem a estratégia de busca.

A Figura 36 apresenta os parâmetros e como são passados para a função.

Dentro da execução da função a ação fica guardada na memória até o momento em que é chamada pela primitiva 'executarAcaoDePercorrerOGrafo'. Nesse momento a ação recebe em sua memória os parâmetros passados pela instrução e é executada. Após a finalização do processo a função continua normalmente até que sua execução seja finalizada.

3.3.2.4 Importação de bibliotecas

A importação de bibliotecas permite a utilização de funções definidas em outros arquivos. O mapa de funções é obtido durante o processo de criação do contexto de execução principal, logo após a fase de interpretação.

Durante a fase de interpretação é criado um construtor que possui o caminho indicado para o arquivo com as funções. Na criação do contexto de execução principal o caminho é validado em seguida é feita a interpretação desse arquivo, obtenção das funções encontradas e finalmente adição das funções no contexto em criação. A partir desse ponto as funções já estão disponíveis para serem utilizadas nas primitivas.

3.3.2.5 Testes

Foram criados testes unitários para todas as instruções da linguagem. A criação deles foi feita conforme as novas instruções estavam sendo implementadas, sempre seguidas pela execução de todos os testes anteriores, garantindo que novas alterações não causassem erros nos comandos já implementados. Os testes criados estão disponibilizados no APÊNDICE C.

Os testes foram criados utilizando JUnit que é um framework utilizado para a geração de testes automatizados. Além desse framework foi criada a primitiva ‘despejar’ que executa a cópia de uma variável e adiciona na lista de despejo. A classe responsável pela execução dos testes utiliza o JUnit para comparar a lista de despejo do programa com uma lista que contem o despejo esperado de cada programa de teste.

3.3.3 IDE

A IDE foi desenvolvida para possibilitar a criação e execução de programas de usuário e fornecer algumas funcionalidades que facilitassem essa tarefa. Através dela que ocorre a saída e entrada de dados da linguagem. Esta seção irá explicar o objetivo dos componentes na tela da IDE e apresentar o processo de desenvolvimento de algumas funcionalidades.

A Figura 43 apresenta a tela da IDE e sobrepõe a imagem com números conectados aos componentes da tela, essa figura será utilizada em conjunto com o Quadro 6 para apresentar os componentes. Esse quadro ira apresentar na primeira coluna o número do componente correspondente na imagem, e na coluna descrição uma explicação sobre o objetivo de uso do componente.

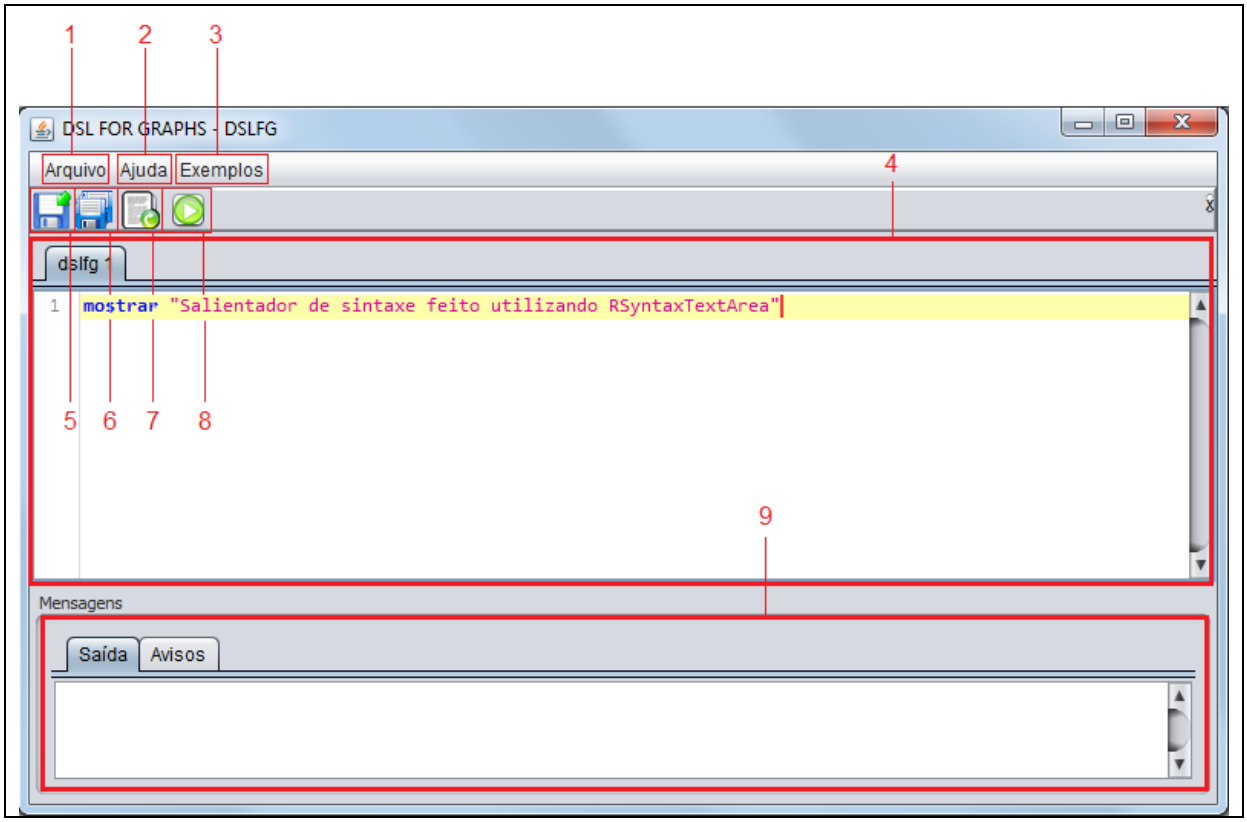


Figura 43. Tela da IDE

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.

Quadro 6. Descrição dos componentes da tela

As mensagens de erro que são comentadas no item 9 do Quadro 6 são lançadas no tratamento de erros que foi citado na seção 3.3.2.2 Tratamento de erros. A exceção lançada pelo decorador da instrução chega a tela onde é obtida a mensagem e apresentada para o usuário.

No comando que está escrito no painel código pode ser observado a diferenciação de cor entre dois termos. Essa característica é chamada de salientador de sintaxe, do inglês *syntax highlight*. Tem o objetivo de destacar os termos mais importantes da linguagem, melhorando a visualização do código.

A implementação do salientador de sintaxe foi feita utilizando o componente RSyntaxTextArea, disponibilizado para uso gratuito e com código aberto. Esse componente permite a definição de coloração através dos tipos pré-definidos como java ou a definição de novos padrões através da extensão da classe AbstractTokenMaker, método que foi utilizado para a implementação do salientador da linguagem.

Além dos elementos gráficos a IDE fornece uma biblioteca padrão que contem os métodos de busca em profundida e em largura. Essa biblioteca pode ser importada através do comando importar e utilizada normalmente, conforme explicado na seção referente a importação de biblioteca.

3.4 Experimentos

Nesta seção serão apresentados os experimentos criados para verificar o potencial da linguagem de resolver os problemas propostos.

3.4.1 Algoritmos de exemplo

Os exemplos foram criados para apresentar a utilização da linguagem em casos reais, ou seja, aplicando em algoritmos tradicionalmente aplicados em grafos. Os exemplos escolhidos então disponíveis no APÊNDICE D.

O apêndice D.1 apresenta a busca em profundidade, um algoritmo que possui o objetivo de verificar todos os vértices atingíveis de um grafo através de uma busca que sempre que chega a um vértice já escolhe um de seus adjacentes para prosseguir a busca até o momento em que não há vértices adjacentes não visitados, nesse momento retorna o caminho verificando outros vértices adjacentes.

O apêndice D.2 apresenta a busca em largura. Diferentemente da busca em profundidade, sempre que esse algoritmo chega a um vértice explora todos os vértices daquele nível e monta um conjunto contendo todos os adjacentes do próximo nível para serem

verificados em seguida, repetindo o processo até que todos os vértices atingíveis do grafo tenham sido verificados.

O apêndice D.3 apresenta a implementação do algoritmo de dijkstra. O objetivo desse algoritmo é definir o menor caminho para todos os vértices do grafo a partir de um vértice inicial. Os dados gerados na execução desse algoritmo podem fornecer a distância para ir do vértice inicial até um vértice final e o caminho percorrido.

3.4.1.1 Execução do algoritmo de busca em profundidade

A Figura 44 apresenta a execução de um algoritmo na linguagem proposta. O algoritmo executado é o mesmo presente no apêndice D.1, responsável pela busca em profundidade. Na aba “dslfg 1” está o código de entrada digitado pelo usuário e na aba “Saída” o resultado após a execução do programa. Conforme a lógica do algoritmo especifica o código percorre o grafo “exemplo” e manda mostrar na tela conforme passa nos vértices.

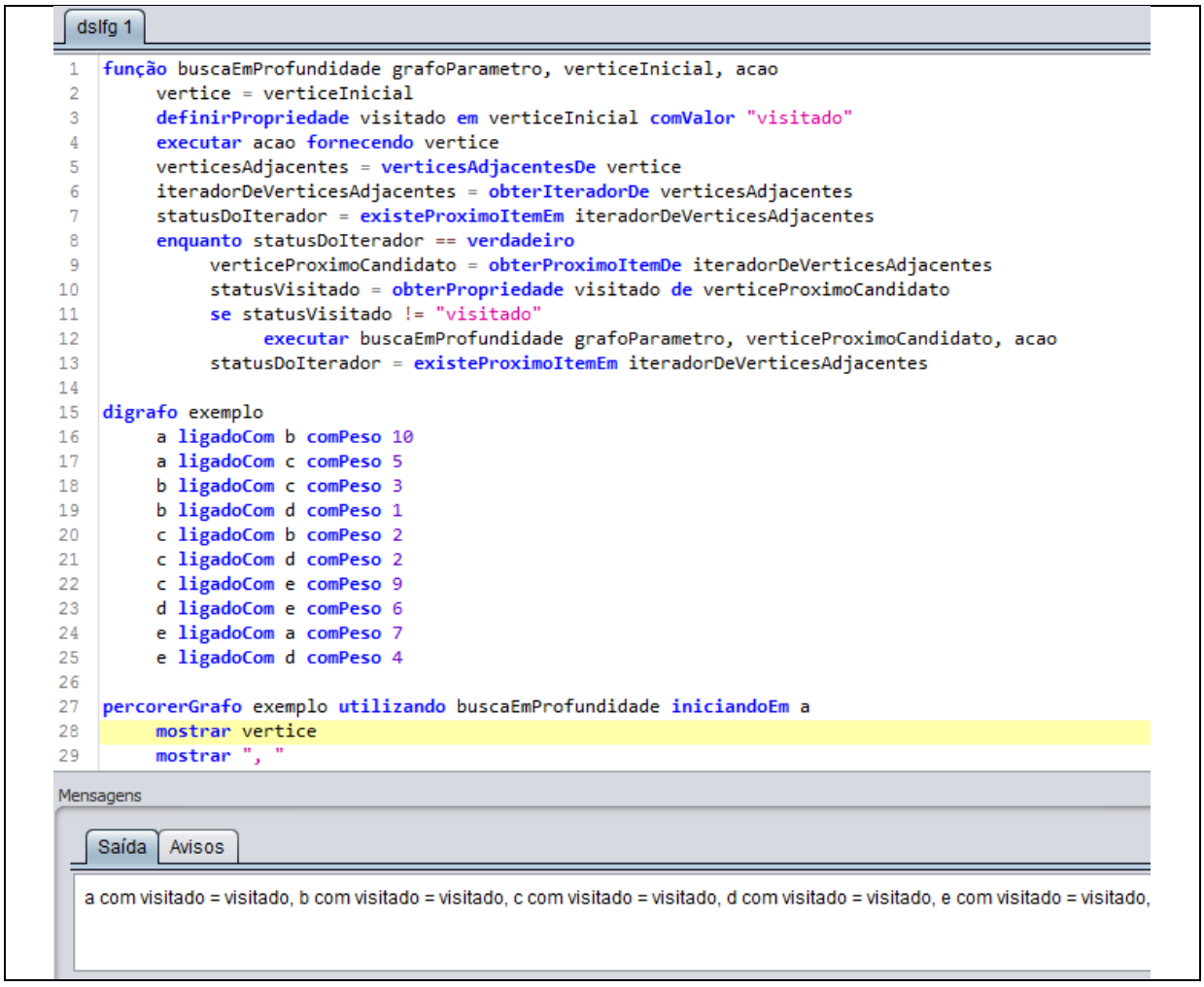


Figura 44. Algoritmo executado na IDE

3.4.2 Apresentação em aula

Para avaliar o potencial didático foi programada uma apresentação para alunos da disciplina de grafos no curso de Ciência da Computação. O experimento foi executado no dia 17 de junho de 2014 com sete alunos em sala e contou com a seguinte estrutura de apresentação:

1. Apresentação da linguagem: Essa apresentação contou com uma introdução teórica sobre DSL, demonstração das operações de conjunto e por final as instruções e estrutura da linguagem.

2. Atividade: Foi apresentado um algoritmo na linguagem proposta com um erro na lógica e solicitado para a turma que apontasse o erro e solução. O objetivo foi fazer com que olhassem os comandos da linguagem com atenção para poderem responder perguntas referente à linguagem.

3. Questionário: Foram elaboradas perguntas para avaliar a efetividade da linguagem na opinião dos avaliados. As perguntas e opções são apresentadas no Quadro 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.

Quadro 7. Perguntas do questionário

Na primeira pergunta todos os avaliados marcaram a opção 'Sim', indicando que todos os usuários consideram que a lógica dos algoritmos fica clara na linguagem. Na segunda pergunta houveram respostas diferentes conforme o gráfico na Figura 45 apresenta.

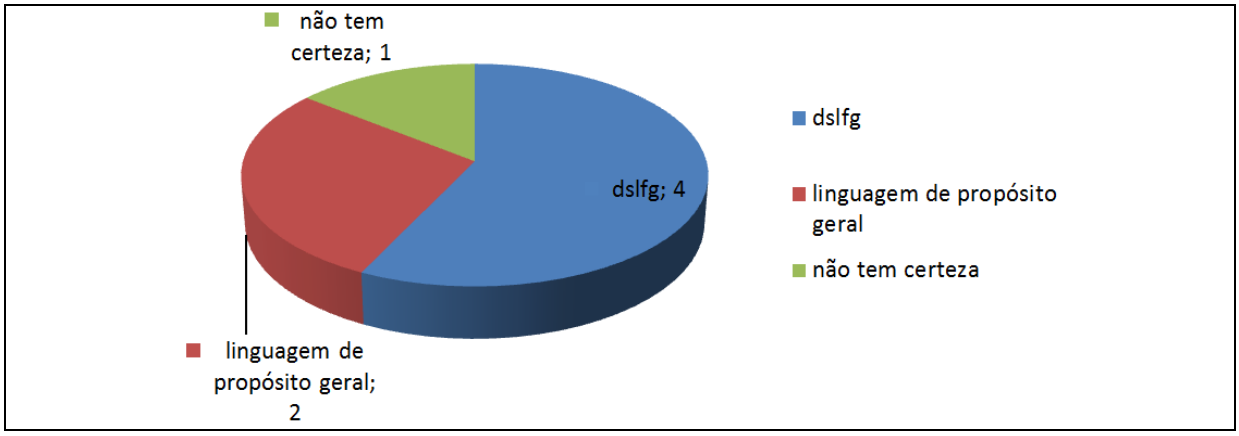


Figura 45. Respostas da segunda pergunta do questionário

4 CONCLUSÕES

O objetivo desse trabalho foi a criação de uma linguagem que possibilitasse a execução de algoritmos direcionados a grafos com uma sintaxe que facilitasse a leitura e compreensão da lógica contida no código. Para isso foi necessário o estudo de duas áreas em especifico, grafos para conseguir projetar uma linguagem que atendesse as necessidades do usuário e a área de compiladores para possibilitar a implementação do projeto.

O estudo referente a grafos incluiu uma revisão referente a sua estrutura e a avaliação de algoritmos para conhecer melhor as formas utilizadas para resolução dos problemas da área. A área de compiladores teve seu estudo focado no aprendizado das melhores praticas no desenvolvimento de uma linguagem, focando em linguagens especificas de domínio e conhecer os métodos e ferramentas utilizadas para desenvolver linguagens.

O ANTLR foi utilizado como ferramenta para a definição da sintaxe e recepção dos eventos da árvore sintática. No inicio do trabalho, a versão quatro da ferramenta tinha sido recentemente lançada, e embora isso a principio tenha dificultado a aquisição de material ideal, fez-se a opção por continuar com a versão atualizada devido a diversas melhorias entre a versão três e quatro da ferramenta, como a forma de descrever a gramática por exemplo.

Durante o desenvolvimento foi dada muita atenção na arquitetura do software, tentando criar um código de fácil leitura e manutenção. Para isso foi adotado o conceito da metodologia ágil de desenvolvimento iterativo, definindo pequenos objetivos em cada parte do desenvolvimento.

Após a finalização do desenvolvimento das estruturas principais da arquitetura, foram formados pacotes com as instruções restantes. Cerca de 70% gasto no projeto foi tomado para formação da arquitetura, o restante para a implementação das instruções em si.

Por fim, após a construção da ferramenta, todos os testes das instruções foram executados para garantir que funcionavam corretamente. Com o resultado positivo na execução das instruções, foram implementados três algoritmos dentro da linguagem com o objetivo de garantir o funcionamento com algoritmos reais de grafos. Os resultados mostraram sucesso para a classe de algoritmos em que foi testado e pode ser considerada uma alternativa positiva para o desenvolvimento de algoritmos voltados a grafos. Além disso o

experimento realizado em sala de aula demonstrou que a linguagem possui potencial como ferramenta didática na disciplina de grafos.

Como trabalhos futuros foram previstos os itens descritos a seguir:

- Flexibilização das instruções: A flexibilização contaria com alterações como permitir expressões aritméticas envolvendo mais de dois elementos por vez, múltiplas expressões lógicas dentro das condicionais, entre outras extensões desse tipo;

- Criar instrução para importar e salvar grafos prontos na linguagem ‘graphml’ ou outra linguagem similar. A linguagem citada é uma linguagem de marcação que foi construída para simplesmente descrever grafos, não seus algoritmos (Graphml Team, 2013);

- Criar instrução para apresentar o grafo com mais opções, dando a possibilidade de mostrar somente as propriedades desejadas;

- Criar instruções para apresentar o grafo graficamente, possivelmente com a utilização de frameworks prontos como, por exemplo, o ‘yED’ (yWorks, 2014);

- Desenvolver a integração da linguagem com outras GPLs, possibilitando o uso em sistemas maiores.

REFERÊNCIAS

AMARAL, Daniela. **Introdução a teoria dos grafos.** [S.I]: Wikidot. Disponível em: <http://danielamaral.wikidot.com/introducao-a-teoria-dos-grafos>. Acesso em: 10 nov. de 2012.

BALAKRISHNAN, V. K. **Schaum's outline of theory and problems of graph theory**. New York: McGraw-Hill, 2005.

BOAVENTURA NETTO, Nome. **Grafos:** teoria, modelos, algoritmos. 4.ed. São Paulo: Edgard Blücher, 2006.

DEURSEN, Arie Van; KLINT, Paul. **Little languages: little maintenance?** Amsterdam: *University of Amsterdam*, 1998. Disponível em: http://www.google.com.br/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CDAQFjAA&url=http%3A%2F%2Fciteseerx.ist.psu.edu%2Fviewdoc%2Fdownload%3Fdoi%3D10.1.1.13.8061%26rep%3Drep1%26type%3Dpdf&ei=m6jEUMCkKoXe8wTljIHoAQ&usg=AFQjCNH9AABLkPOusqZIGBYs1oBYN-nuvw, Acesso em: 23 out. 2012.

DIESTEL, Reinhard. **Graph Theory**. 3. ed. Berlin: Springer-Verlag Heidelberg, 2005.
EUC.FC.UL. **Conexos.** [S.I],[S.d.]. Disponível em: <http://www.educ.fc.ul.pt/icm/icm2001/icm33/conexos.htm>. Acesso em: 03 nov. 2012.
FISTER JR, Iztok; MERNIK, Marjan; BREST, 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. Disponível:< http://graphml.graphdrawing.org/ >. Acesso em 06/06/2014.

GULWANI, Sumit; TARACHANDANI, Asha; GUPTA, Deepak; SANGHI, Dheeraj; BARRETO, Luciano Porto; MULLER, Gilles; CONSEL, Charles.In.: **WebCal : a domain specific language for web caching**. [S.I.], 2001. Disponível em: http://research.microsoft.com/en-us/um/people/sumitg/pubs/webcal_wcw00.pdf. Acesso em: 25 out. 2012.

HARTSFIELD, Nora; RINGEL, Gerhard. **Pearls in graph theory.** Boston: Academic Press, 1990.

IME.USP. **Teoria dos grafos.** <Ano que a informação foi gerada. Esse ano que vai na citação>**.** Disponivel em: <http://www.ime.usp.br/~pf/teoriadosgrafos/texto/TeoriaDosGrafos.pdf>. Acesso em: 14 nov. 2012

KLARLUND, Nils; SCHWARTZBACH, Michael I. **A domain-specific language for regular sets of strings and trees**. Artigo apresentado na Conference on Domain-Specific Languages (DSL). Santa Barbara: USENIX, 1997. Disponível em: http://www.google.com.br/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CDAQFjAA&url=http%3A%2F%2Fciteseerx.ist.psu.edu%2Fviewdoc%2Fdownload%3Fdoi%3D10.1.

1.30.446%26rep%3Drep1%26type%3Dpdf&ei=4rDEUOumMofM9gTt8oC4DA&usg=AFQjCNFXux456b45WHjSqIqptU-_RfH4IA. Acesso em 07 set. 2012.

KLOTZ, Walter. **Graph coloring algorithms**. Clausthal: University of Technology, 2002. Disponível em: http://www.math.tu-clausthal.de/Arbeitsgruppen/Diskrete-Optimierung/publications/2002/gca.pdf. Acesso em: 01 nov. 2012.

KREHER, Donald L.; STINSON, Douglas R. **Combinatorial algorithms:** generation, enumeration, and search. Boca raton, Fla: CRC Press, 1999.

KRUEGER, Charles W. Software Reuse. AC Computing Surveys, v. 24, n. 2, Jun. 1992, p. 131-183. Disponível em: http://www.biglever.com/papers/Krueger_AcmReuseSurvey.pdf . Acesso em: 15 out. 2012.

LOUDEN, Kenneth C. **Compiladores principios e práticas**. São Paulo, SP : Pioneira Thomson Learning, 2004.

MARTIN, Robert C. **Clean code A handbook of agile software craftsmanship**. Boston, MA: Pearson Education, Inc, 2009.

MARTINS, José Carlos Cordeiro. **Técnicas para gerenciamento de projetos de software.** São Paulo: Brasport, 2007.

PARR, Terence. **The definitive ANTLR 4 Reference**. Dallas, Texas: The Pragmatic Bookshelf, 2012.

PETER, Jandl Jr; ZUCHINI, Márcio Henrique. **IC:** um interpretador de linguagem. **Revista Projeções**. v. 19/20, Jan./Dez. 2001/2002, p. 29-36. Disponível em: http://www.saofrancisco.edu.br/edusf/publicacoes/RevistaProjecoes/Volume_03/uploadAddress/proje%C3%A7oes-6[6374].pdf. Acesso em: 02 out. 2012.

REACTIVE-SEARCH. **Clique.** Trento: Reactive Search, [S.d.]. Disponível em: <http://reactive-search.com/clique.php>. Acesso em: 13 nov. 2012.
RICARTE, Ivan. **Introdução a compilação.** Rio de Janeiro: Campus, 2008.
ROSEN, Kenneth H. **Discrete mathematics and its applications.** Boston: WCB/McGraw-Hil**,** 1999.

SAID, Ricardo. **Curso de lógica de programação.** São Paulo: Universo dos livros, 2007.

SCIENCEBLOGS. **Graph contraction.** <Ano que a informação foi gerada. Esse ano que vai na citação>**.** Disponível em: <http://scienceblogs.com/goodmath/2007/07/08/graph-contraction-and-minors-1/>. Acesso em: 10 nov. 2012.

SEBESTA, Robert W. **Conceitos de linguagens de programação.** Porto Alegre: Bookman, 2002.

SHI, Hongchi;GADER, Paul; LI, Hongzheng. **Parallel Mesh Algorithms for Grid Graph Shortest Paths with Application to Separating of Touching Chromosomes.** Kluwer Academic Pubishers. Boston, 1998.

SZWARCFITER, Jayme Luiz. **Grafos e algoritmos computacionais**. Rio de Janeiro: Campus, 1988.

VASIR. **Dijkstra.** Disponível em: <http://vasir.net/blog/game_development/dijkstras_algorithm_shortest_path/>. Acesso em: 11 nov. 2012.

WIKIPÉDIA. **Laço.** <Ano que a informação foi gerada. Esse ano que vai na citação**>.** Disponível em: <http://pt.wikipedia.org/wiki/Laço_(teoria_dos_grafos)>. Acesso em: 14 nov. 2012.

yWorks. **yEd Graph Editor**. 2014. Disponível:< http://www.yworks.com/en/products_yed_about.html/ >. Acesso em 06/06/2014.

APÊNDICE A. EXEMPLOS DE ALGORITMOS

A.1. PROTÓTIPO INICIAL

```
importar "caminhoDoArquivo"
mostrar 3
menorAresta = MAX
mostrar menorAresta
grafo grafoExemploUmDoLivro
        Navegantes
                com Populacao = 60000
                com efetivoPolicial = 300
                ligadoCom B
                comPeso 2
        A ligadoCom C comPeso 3
        B ligadoCom D comPeso 2
        B ligadoCom E comPeso 1
        B ligadoCom C comPeso 3
        C ligadoCom E comPeso 2
DFS grafo, verticeInicial, acao
        verticeAtual = verticeInicial
        verticesVisitados = {}
        enquanto verticesVisitados subconjuntoProprioDe grafo
                adicionar verticeAtual em verticesVisitados
                verticesAdjacentesDoVerticeAtual = verticesAdjacentesDe verticeAtual
                verticesAdjacentesNaoVisitadosDoVerticeAtual = retirar verticesVisitados de
verticesAdjacentesDoVerticeAtual
                se verticesAdjacentesNaoVisitadosDoVerticeAtual diferenteDe {}
        verticeAtual = obterUmElementoDe verticesAdjacentesNaoVisitadosDoVerticeAtual
                executar acao fornecendo verticeAtual arestaPercorrida

obterConjuntoEnquanto parametro2
        conjuntoRetorno = {9}
        enquanto conjuntoRetorno subconjuntoProprioDe parametro2
                elemento = obterUmElementoDe parametro2
                adicionar elemento em conjuntoRetorno
        retornar conjuntoRetorno
        mostrar "Não deveria ter mostrado isso em: obterConjuntoEnquanto"

percorrerUtilizando DFS oGrafo grafoExemploUmDoLivro iniciandoEm Navegantes
        aplicando
                verticeAtual
                arestapercorrida
                população = obter populacao de Navegantes
                população = 5+5+5+5+5+5+5
                mostrar população
```

A.2. BUSCA EM PROFUNDIDADE

dslfg 1

```
1  função buscaEmProfundidade grafoParametro, verticeInicial, acao
2       vertice = verticeInicial
3       definirPropriedade visitado em verticeInicial comValor "visitado"
4       executar acao fornecendo vertice
5       verticesAdjacentes = verticesAdjacentesDe vertice
6       iteradorDeVerticesAdjacentes = obterIteradorDe verticesAdjacentes
7       statusDoIterador = existeProximoItemEm iteradorDeVerticesAdjacentes
8       enquanto statusDoIterador == verdadeiro
9            verticeProximoCandidato = obterProximoItemDe iteradorDeVerticesAdjacentes
10           statusVisitado = obterPropriedade visitado  de verticeProximoCandidato
11           se statusVisitado != "visitado"
12                executar buscaEmProfundidade grafoParametro, verticeProximoCandidato, acao
13           statusDoIterador = existeProximoItemEm iteradorDeVerticesAdjacentes
14
15
16 digrafo exemploUfsc
17      a ligadoCom b comPeso 10
18      a ligadoCom c comPeso 5
19      b ligadoCom c comPeso 3
20      b ligadoCom d comPeso 1
21      c ligadoCom b comPeso 2
22      c ligadoCom d comPeso 2
23      c ligadoCom e comPeso 9
24      d ligadoCom e comPeso 6
25      e ligadoCom a comPeso 7
26      e ligadoCom d comPeso 4
27
28 percorerGrafo exemploUfsc utilizando buscaEmProfundidade iniciandoEm a
29      mostrar vertice
30
31
```

Figura 46. Algoritmo de busca em profundidade na linguagem dslfg

APÊNDICE B. GRAMÁTICA

A.3. DSLFGLEXICO.G4

```
lexer grammar DSLFGLexico;

TIPOS_BOOLEANOS: 'verdadeiro' | 'falso';

CONSTANTES: 'numeroMaximo' | 'numeroMinimo';

OPERADOR_ARITMETICO: '+' | '-' | '*' | '/' | 'div' | 'mod' ;

OPERADOR_CONDICIONAL_DE_CONJUNTO : ' subconjuntoProprioDe ' | ' subconjunto ' |
' eUmElementoDe ' | ' naoEUmElementoDe ';

OPERADOR_CONDICIONAL: ' == ' | ' <= ' | ' >= '| ' < '| ' > ' | ' != ';

NUMERO : DIGITO+ ('.' DIGITO+)?;

STRING:     ('“'|'"')
           (' ' |'!' | [\u0001-\u0021] | [\u0023-\u30FF] | 'Á')*
            ('”'|'"')
      ;

DIGITO : [0-9];

PALAVRA : ([A-Z]|[a-z]|[À-ü]|[\u0168-\u0169]|'Ã'|'ã'|'ú'|'Ú'|'_') ([A-Z]|[a-
z]|[À-ü]|[\u0168-\u0169]|'Ã'|'ã'|'ú'|'Ú'|'_'|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;

inicio :
           importar* codigo*
;
codigo:
          QUEBRA_DE_LINHA |
          construcaoDeGrafo |
          construcaoDeDigrafo |
          funcaoMostrar |
          declaracaoDeConjunto |
          atribuicaoPorExpressao |
          adicionarAConjunto |
          definirPropriedade |
          se |
          enquanto |
          despejar |
          funcao |
          chamadaFuncao |
          retorno |
          funcaoPercorerGrafo |
          executarAcaoDePercorrerGrafo

      ;
se:
      TAB* 'se' (condicionalDeConjunto | condicional) QUEBRA_DE_LINHA
  ;

enquanto:
            TAB* 'enquanto' (condicionalDeConjunto | condicional)
QUEBRA_DE_LINHA
        ;

condicionalDeConjunto:
                          PALAVRA OPERADOR_CONDICIONAL_DE_CONJUNTO PALAVRA
                    ;
condicional:
               multiTipos OPERADOR_CONDICIONAL multiTipos
           ;
importar :
             'importar' STRING
         ;
construcaoDeGrafo:
                    TAB* 'grafo' PALAVRA QUEBRA_DE_LINHA
                       ligacaoDeVertice*
                ;
construcaoDeDigrafo:
                    TAB* 'digrafo' PALAVRA QUEBRA_DE_LINHA
                       ligacaoDeVertice*
                         ;
ligacaoDeVertice:
                  TAB+ declaracaoDeVertice 'ligadoCom' declaracaoDeVertice
pesoDaAresta? QUEBRA_DE_LINHA
               ;
```

```
declaracaoDeVertice:
                    PALAVRA (QUEBRA_DE_LINHA adicaoDePropriedade* TAB+)?
                ;
adicaoDePropriedade:
                    TAB+ 'com' PALAVRA '=' multiTipos QUEBRA_DE_LINHA
                ;
pesoDaAresta:
                'comPeso' NUMERO
            ;
funcaoMostrar:
                TAB* 'mostrar' multiTipos QUEBRA_DE_LINHA
            ;
declaracaoDeConjunto:
                    TAB* PALAVRA '=' conjunto  QUEBRA_DE_LINHA
                ;
atribuicaoPorExpressao:
                       TAB* PALAVRA '=' funcoes QUEBRA_DE_LINHA
                    ;
conjunto:
            '{'((NUMERO',')* NUMERO)? '}'
        ;
adicionarAConjunto:
                    TAB* 'adicionar' multiTipos 'em' PALAVRA QUEBRA_DE_LINHA
                ;
definirPropriedade:
                  TAB* 'definirPropriedade' PALAVRA 'em' PALAVRA 'comValor'
multiTipos QUEBRA_DE_LINHA
                ;
funcoes:
            obterVerticesAdjacentes | obterUmElemento | obterUmVertice| retirar
| uniao | interseccao | diferenca |
            chamadaFuncao | obterVertice | obterPesoDaAresta | obterPropriedade
| obterIterador | existeProximoItem |
            obterProximoItem | obterTodosOsVertices | obterTodasAsArestas |
copiarGrafo | expressaoAritmetica    ;

expressaoAritmetica:
                     multiTipos (OPERADOR_ARITMETICO multiTipos)?
                ;
retirar:
            PALAVRA 'retirando' PALAVRA
        ;
uniao:
         PALAVRA 'unindo' PALAVRA
     ;
interseccao:
                PALAVRA 'interseccao' PALAVRA
            ;
diferenca:
             PALAVRA 'diferenca' PALAVRA
         ;
chamadaFuncao:
               TAB*  'executar' PALAVRA parametros
            ;
copiarGrafo:
              'copiarGrafo' PALAVRA
          ;
```

```
obterTodosOsVertices:
                        'obterTodosOsVerticesDe' PALAVRA
                    ;
obterTodasAsArestas:
                       'obterTodasAsArestasDe' PALAVRA
                   ;
obterIterador:
                'obterIteradorDe' PALAVRA
             ;
existeProximoItem:
                    'existeProximoItemEm' PALAVRA
                ;
obterProximoItem:
                   'obterProximoItemDe' PALAVRA
                ;
obterPropriedade:
                   'obterPropriedade' PALAVRA 'de' PALAVRA
                ;
obterPesoDaAresta:
                    'obterPesoDaAresta' 'de' PALAVRA 'para' PALAVRA
                ;
obterVerticesAdjacentes:
                            'verticesAdjacentesDe' PALAVRA
                       ;
obterUmElemento :
                'obterUmElementoDe' PALAVRA
             ;
obterUmVertice :
                'obterUmVerticeDe' PALAVRA
             ;
obterVertice :
                 'obterVertice' PALAVRA 'de' PALAVRA
              ;
multiTipos:
               NUMERO | PALAVRA | STRING | CONSTANTES | TIPOS_BOOLEANOS
           ;
despejar:
             TAB* 'despejar' PALAVRA
         ;
funcao:
          'função' PALAVRA parametros
       ;
parametros:
              (PALAVRA ','?)*
           ;
retorno:
                TAB+ 'retornar' PALAVRA
          ;
funcaoPercorerGrafo:
                    TAB* 'percorerGrafo' PALAVRA 'utilizando' PALAVRA
'iniciandoEm' PALAVRA
                    ;
executarAcaoDePercorrerGrafo:
                                TAB* 'executar' PALAVRA 'fornecendo' parametros

                           ;
```

APÊNDICE C. PROGRAMAS PARA TESTE

C.1. CONJUNTO SIMPLES

```
xpto = {1.5,2,3,4,5}
mostrar xpto
```

C.2. CONJUNTO SIMPLES 2

```
xpto = {}
mostrar xpto
xpto = {1.5,2,3,4,5}
mostrar xpto
adicionar 8 em xpto
mostrar xpto
```

C.3. COMANDO ENQUANTO

```
xpto = {1,2,3,4,5}
xpto2 = {1}
mostrar xpto2
enquanto xpto2 subconjuntoProprioDe xpto
       novoElemento = obterUmElementoDe xpto
       adicionar novoElemento em xpto2
mostrar xpto2
```

C.4. COMANDO ENQUANTO COM SE

```
xpto = {1,2,3,4,5}
mostrar xpto
xpto2 = {1}
mostrar xpto2
xpto3 = {1,2,3}
se xpto3 subconjuntoProprioDe xpto
       mostrar xpto3
       adicionar 13 em xpto3
       xpto4 = {8,9,10}
       xpto5 = {}
       enquanto xpto2 subconjuntoProprioDe xpto
              xpto5 = {}
              novoElemento = obterUmElementoDe xpto
              adicionar novoElemento em xpto2
mostrar xpto2
```

C.5. COMANDO SE

```
xpto = {1,2,3,4,5}
mostrar xpto
xpto2 = {4,5}
mostrar xpto2
se xpto2 subconjuntoProprioDe xpto
       adicionar 8 em xpto
mostrar xpto
```

C.6. COMANDO SE 2

```
xpto = {1,2,3,4,5}
mostrar xpto
xpto2 = {4,5}
mostrar xpto2
se xpto2 subconjuntoProprioDe xpto
       xpto3 = {6,7,8}
       adicionar 8 em xpto
       mostrar xpto3
mostrar xpto
```

C.7. CONTEXTO DE FUNÇÃO RECURSIVA

```
função teste numeroParametro
       numero = numeroParametro
       mostrar numero
       se numero < 6
              numero = numero + 1
              executar teste numero
       mostrar numero

função testeConjunto numeroParametro
       numero = numeroParametro
       conjunto = {}
       adicionar numero em conjunto
       mostrar conjunto
       se numero < 4
              numero = numero + 1
              executar testeConjunto numero
       mostrar conjunto

numero1 = 0
executar testeConjunto numero1
executar teste numero1
```

C.8. TESTE DE CONTEXTO

```
função teste2
	mostrar y
	despejar y
função teste
	x = {2, 3}
	y = 5
	executar teste2
	retornar x
x = {0, 1}
executar teste
despejar x
mostrar x
```

C.9. TESTE DE CONTEXTO 2

```
função teste
	x = {2, 3}
executar teste
despejar x
mostrar x
```

C.10. OBTER ELEMENTO DE CONJUNTO

```
xpto = {1.5,2,3,4,5}
xpto2 = obterUmElementoDe xpto
mostrar xpto2
```

C.11. OPERADOR RETIRANDO

```
conjuntoPrincipal = {1, 2, 3, 4, 5}
mostrar conjuntoPrincipal
conjuntoRetirar = {1, 5, 3}
mostrar conjuntoRetirar
conjuntoResultado = conjuntoPrincipal retirando conjuntoRetirar
mostrar conjuntoResultado
```

C.12. CONSTANTES

```
grafo grafoExemploUmDoLivro
       Navegantes ligadoCom Itajai
       Itajai ligadoCom BalnearioCamboriu
       Itajai ligadoCom Brusque
vertice = obterVertice Navegantes de grafoExemploUmDoLivro
definirPropriedade valorMaximo em vertice comValor numeroMaximo
definirPropriedade valorMinimo em vertice comValor numeroMinimo

valorMinimo = obterPropriedade valorMinimo de vertice
valorMaximo = obterPropriedade valorMaximo de vertice

despejar valorMaximo
despejar valorMinimo

mostrar valorMaximo
mostrar valorMinimo
mostrar "Fim"
```

C.13. DEFINIR PROPRIEDADE EM VÉRTICE

```
grafo grafoExemploUmDoLivro
       Navegantes ligadoCom Itajai
       Itajai ligadoCom BalnearioCamboriu
       Itajai ligadoCom Brusque
vertice = obterVertice Navegantes de grafoExemploUmDoLivro
mostrar grafoExemploUmDoLivro
mostrar vertice
definirPropriedade valor em vertice comValor 5
despejar vertice
mostrar vertice
mostrar grafoExemploUmDoLivro
mostrar "Fim"
```

C.14. DESPEJAR

```
conjuntoPrincipal = {1, 2, 3, 4, 5}
despejar conjuntoPrincipal
conjuntoRetirar = {1, 5, 3}
despejar conjuntoRetirar
```

C.15. FUNÇÃO COPIAR GRAFO

```
grafo grafoOriginal
	Navegantes
			com populacao = 2
		ligadoCom Itajai
			com populacao = 4
		comPeso 2
	Itajai ligadoCom BalnearioCamboriu comPeso 3.5
	Itajai ligadoCom Brusque comPeso 5
grafoCopia = copiarGrafo grafoOriginal
despejar grafoOriginal
despejar grafoCopia

mostrar "grafoOriginal"
mostrar grafoOriginal

verticeOriginal = obterVertice Navegantes de grafoOriginal
definirPropriedade valor em verticeOriginal comValor 5

mostrar "grafoOriginal"
mostrar grafoOriginal

mostrar "grafoCopia"
mostrar grafoCopia
```

C.16. FUNÇÃO DE USUARIO

```
mostrar "fora da funçao"

função escreverFuncao conteudo1, conteudo2
	se conteudo1 subconjuntoProprioDe conteudo2
		mostrar "dentro do if que está dentro da função"
	mostrar "dentro da funçao"
	conjuntoRetorno = {1,2,3,4}

mostrar "fora da função de novo"
conjunto1 = {1, 3}
conjunto2 = {1, 2}

executar escreverFuncao conjunto1, conjunto2
```

C.17. FUNÇÃO DE USUARIO COM RETORNO

```
conjunto = executar obterConjunto
mostrar conjunto
despejar conjunto

conjuntoParametro1 = {6,7}
conjuntoParametro2 = {6,7,8}
conjunto2 = executar obterConjuntoSe conjuntoParametro1, conjuntoParametro2
mostrar conjunto2
despejar conjunto2

conjuntoParametro1 = {9,10,11}
conjunto3 = executar obterConjuntoEnquanto conjuntoParametro1
mostrar conjunto3
despejar conjunto3

mostrar "Fim"

função obterConjunto
	conjuntoRetorno = {1,2,3,4}
	retornar conjuntoRetorno

função obterConjuntoSe parametro1, parametro2
	se parametro1 subconjuntoProprioDe parametro2
		conjuntoRetorno = {6,7,8}
		retornar conjuntoRetorno
	mostrar "Não deveria ter mostrado isso em: obterConjuntoSe"

função obterConjuntoEnquanto parametro2
	conjuntoRetorno = {9}
	enquanto conjuntoRetorno subconjuntoProprioDe parametro2
		elemento = obterUmElementoDe parametro2
		adicionar elemento em conjuntoRetorno
	retornar conjuntoRetorno
	mostrar "Não deveria ter mostrado isso em: obterConjuntoEnquanto"
```

C.18. FUNÇÃO OBTER PESO DA ARESTA

```
grafo grafoExemploUmDoLivro
	Navegantes ligadoCom Itajai comPeso 1
	Itajai ligadoCom BalnearioCamboriu comPeso 2
	Itajai ligadoCom Brusque comPeso 3

vertice = obterVertice Navegantes de grafoExemploUmDoLivro
vertice2 = obterVertice Itajai de grafoExemploUmDoLivro
pesoDaAresta = obterPesoDaAresta de vertice para vertice2
despejar pesoDaAresta
mostrar pesoDaAresta
mostrar "Fim"
```

C.19. FUNÇÃO OBTER PROPRIEDADE

```
grafo grafoExemploUmDoLivro
	Navegantes ligadoCom Itajai
	Itajai ligadoCom BalnearioCamboriu
	Itajai ligadoCom Brusque
vertice = obterVertice Navegantes de grafoExemploUmDoLivro
definirPropriedade valor em vertice comValor 5
mostrar "Vertice:"
mostrar vertice
mostrar "valor:"
valor = obterPropriedade valor de vertice
mostrar valor
despejar valor
mostrar "Fim"
```

C.20. FUNÇÃO OBTER VÉRTICE

```
grafo grafoExemploUmDoLivro
	Navegantes ligadoCom Itajai
	Itajai ligadoCom BalnearioCamboriu
	Itajai ligadoCom Brusque

vertice = obterVertice Navegantes de grafoExemploUmDoLivro
despejar vertice
mostrar vertice
mostrar "Fim"
```

C.21. FUNÇÃO PERCORRER GRAFO

```
função teste graf, init, action
	conjunto1 = {1,2, 1000}
	conjunto2 = {1}
	enquanto conjunto2 subconjuntoProprioDe conjunto1
		verticeRetorno = obterUmVerticeDe graf
		executar action fornecendo verticeRetorno
		elemento = obterUmElementoDe conjunto1
		adicionar elemento em conjunto2

função buscaTeste grafoTeste, verticeInicial, acao
	retorno = executar teste grafoTeste, verticeInicial, acao

grafo grafoExemploUmDoLivro
	Navegantes ligadoCom Itajai
	Itajai ligadoCom BalnearioCamboriu
	Itajai ligadoCom Brusque

percorerGrafo grafoExemploUmDoLivro utilizando buscaTeste iniciandoEm Navegantes
	mostrar verticeRetorno
	despejar verticeRetorno
mostrar "Fim"
```

C.22. FUNÇÕES DE ITERADOR

```
conjunto = {1,2}
iterador = obterIteradorDe conjunto
status = existeProximoItemEm iterador
mostrar status
despejar status
valor = obterProximoItemDe iterador
mostrar valor

status = existeProximoItemEm iterador
mostrar status
valor = obterProximoItemDe iterador
mostrar valor

status2 = existeProximoItemEm iterador
mostrar status2
despejar status2

função teste conjuntoEsquerda, conjuntoDireita
	iterador = obterIteradorDe conjuntoEsquerda
	status = existeProximoItemEm iterador
	se status == verdadeiro
		elemento = obterUmElementoDe conjuntoEsquerda
		adicionar elemento em conjuntoDireita
		conjuntoEsquerda = conjuntoEsquerda retirando conjuntoDireita
		executar teste conjuntoEsquerda, conjuntoDireita
		mostrar elemento


função teste2 conjuntoEsquerda, conjuntoDireita
	elemento = obterUmElementoDe conjuntoEsquerda
	mostrar "elementoAntes"
	mostrar elemento
	adicionar elemento em conjuntoDireita
	se conjuntoDireita subconjuntoProprioDe conjuntoEsquerda
		executar teste conjuntoEsquerda, conjuntoDireita
		mostrar "elementoDepois"
		mostrar elemento

conjunto1 = {0,1,2,3}
conjunto2 = {}
conjunto3 = {1,2,3,4}
conjunto4 = {}
executar teste2 conjunto3, conjunto4
mostrar "Fim"
```

C.23. FUNÇÕES OBTER TODOS OS VERTICES E ARESTAS

```
digrafo grafoExemploUmDoLivro
	Navegantes ligadoCom Itajai
	Itajai ligadoCom BalnearioCamboriu
	Itajai ligadoCom Brusque
vertices = obterTodosOsVerticesDe grafoExemploUmDoLivro
arestas = obterTodasAsArestasDe grafoExemploUmDoLivro
mostrar "vertices"
mostrar vertices
mostrar "arestas"
mostrar arestas
despejar vertices
despejar arestas
```

C.24. OBTER VERTICE

```
grafo grafoExemploUmDoLivro
	Navegantes ligadoCom Itajai comPeso 2
	Itajai ligadoCom BalnearioCamboriu comPeso 3.5
	Itajai ligadoCom Brusque comPeso 5
xpto = obterUmVerticeDe grafoExemploUmDoLivro
mostrar xpto
```

C.25. OBTER VERTICES ADJACENTES

```
grafo grafoExemplo
	Navegantes ligadoCom Itajai comPeso 2
	Itajai ligadoCom BalnearioCamboriu comPeso 3.5
	Itajai ligadoCom Brusque comPeso 5
mostrar grafoExemplo
vertice = obterUmVerticeDe grafoExemplo
verticesAdjacentes = verticesAdjacentesDe vertice
mostrar vertice
mostrar verticesAdjacentes
despejar verticesAdjacentes
```

C.26. COMANDO MOSTRAR

```
conjunto = {1,2,3,4,5,6,7,8,9}
mostrar conjunto
elementoDoConjunto = obterUmElementoDe conjunto
mostrar 20
mostrar "Fim"
```

C.27. OBTER PROPRIEDADE INEXISTENTE

```
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

vertice = obterUmVerticeDe exemploUfsc
valor = obterPropriedade visitado  de vertice
mostrar valor
despejar valor
```

C.28. GRAFO COM PESO

```
grafo grafoExemploUmDoLivro
       Navegantes ligadoCom Itajai comPeso 2
       Itajai ligadoCom BalnearioCamboriu comPeso 3.5
       Itajai ligadoCom Brusque comPeso 5
despejar grafoExemploUmDoLivro
mostrar grafoExemploUmDoLivro
```

C.29. GRAFO COMPLETO

```
grafo grafoExemploUmDoLivro
       Navegantes
                     com populacao = 2
              ligadoCom Itajai
                     com populacao = 4
              comPeso 2
       Itajai ligadoCom BalnearioCamboriu comPeso 3.5
       Itajai ligadoCom Brusque comPeso 5
despejar grafoExemploUmDoLivro
mostrar grafoExemploUmDoLivro
```

C.30. GRAFO SIMPLES

```
grafo grafoExemploUmDoLivro
       Navegantes ligadoCom Itajai
       Itajai ligadoCom BalnearioCamboriu
       Itajai ligadoCom Brusque
digrafo digrafoExemploUmDoLivro
       Navegantes ligadoCom Itajai
       Itajai ligadoCom BalnearioCamboriu
       Itajai ligadoCom Brusque
mostrar "grafo"
mostrar grafoExemploUmDoLivro
mostrar "digrafo"
mostrar digrafoExemploUmDoLivro
despejar grafoExemploUmDoLivro
despejar digrafoExemploUmDoLivro
```

C.31. DIFERENÇA DE CONJUNTOS

```
conjunto1 = {0, 3}
conjunto2 = {0, 1}
conjunto = conjunto1 diferenca conjunto2
mostrar conjunto
despejar conjunto
```

C.32. DIV

```
sete = 7
tres = 3
x = sete div tres
mostrar x
despejar x
mostrar "Fim"
```

C.33. DIVISÃO

```
cinco = 5
quatro = 4
x = cinco / quatro
mostrar x
despejar x
mostrar "Fim"
```

C.34. OPERADOR É UM ELEMENTO DE

```
xpto = {1,2,3,4,5}
numero = 5
x = falso
y = falso
se numero eUmElementoDe xpto
       numero = 6
       x = verdadeiro
se xpto2 eUmElementoDe xpto
       y = verdadeiro
mostrar x
despejar x
mostrar y
despejar y
```

C.35. OPERADOR INTERSECÇÃO

```
conjunto1 = {0}
conjunto2 = {0, 1}
conjunto = conjunto1 interseccao conjunto2
mostrar conjunto
despejar conjunto
```

C.36. OPERADOR MAIOR

```
a = 0
b = 0
c = 0
se 6 > 5
      a = 1
      mostrar "certo"
se 5 > 5
      b = 1
      mostrar "errado"
se 4 > 5
      c =1
      mostrar "errado"
despejar a
despejar b
despejar c
```

C.37. OPERADOR MAIOR OU IGUAL

```
a = 0
b = 0
c = 0
se 6 >= 5
      a = 1
      mostrar "certo"
se 5 >= 5
      b = 1
      mostrar "certo"
se 4 >= 5
      c =1
      mostrar "errado"
despejar a
despejar b
despejar c
```

C.38. OPERADOR MENOR

```
a = 0
b = 0
c = 0
se 6 < 5
      a = 1
      mostrar "errado"
se 5 < 5
      b = 1
      mostrar "errado"
se 4 < 5
      c =1
      mostrar "certo"
despejar a
despejar b
despejar c
```

C.39. OPERADOR MENOR OU IGUAL

```
a = 0
b = 0
c = 0
se 6 <= 5
      a = 1
      mostrar "errado"
se 5 <= 5
      b = 1
      mostrar "certo"
se 4 <= 5
      c =1
      mostrar "certo"
despejar a
despejar b
despejar c
```

C.40. MOD

```
sete = 7
tres = 3
x = sete mod tres
mostrar x
despejar x
mostrar "Fim"
```

C.41. MULTIPLICAÇÃO

```
cinco = 5
quatro = 4
x = cinco * quatro
mostrar x
despejar x
mostrar "Fim"
```

C.42. OPERADOR NÃO É UM ELEMENTO DE

```
xpto = {1,2,3,4,5}
numero = 5
x = falso
y = falso
se numero naoEUmElementoDe xpto
      x = verdadeiro
numero = 6
se xpto2 naoEUmElementoDe xpto
      y = verdadeiro
mostrar x
despejar x
mostrar y
despejar y
```

C.43. OPERADOR DIFERENTE

```
conjunto1 = {1,2}
conjunto2 = {1,2}
conjuntoEsperado1 = {3,4}
conjuntoEsperado2 = {}
se conjunto1 != conjunto2
       conjuntoEsperado1 = {5,4}
       mostrar "deu errado"
conjunto2 = {1,2,3}
se conjunto1 != conjunto2
       conjuntoEsperado2 = {3,4}
       mostrar "deu certo"
despejar conjuntoEsperado1
despejar conjuntoEsperado2
teste = "visitado"
se teste != "visitado"
       mostrar "deu errado"
mostrar "Fim"
```

C.44. OPERADOR IGUAL

```
conjunto1 = {1,2}
conjunto2 = {1,2}
conjuntoEsperado1 = {}
conjuntoEsperado2 = {3,4}
se conjunto1 == conjunto2
       conjuntoEsperado1 = {3,4}
       mostrar "deu certo"
conjunto2 = {1,2,3}
se conjunto1 == conjunto2
       conjuntoEsperado2 = {5,6}
       mostrar "deu errado"
despejar conjuntoEsperado1
despejar conjuntoEsperado2
mostrar "Fim"
```

C.45. SOMA E ATRIBUIÇÃO SIMPLES

```
cinco = 5
quatro = 4
x = cinco + quatro
mostrar x
despejar x
mostrar "Fim"
```

C.46. SUBCONJUNTO

```
xpto = {1,2,3,4,5}
mostrar xpto
xpto2 = {1,2,3,4,5}
mostrar xpto2
x = falso
y = falso
se xpto2 subconjunto xpto
       x = verdadeiro
adicionar 8 em xpto2
mostrar xpto2
se xpto2 subconjunto xpto
       y = verdadeiro
mostrar x
despejar x
mostrar y
despejar y
```

C.47. SUBTRAÇÃO

```
cinco = 5
quatro = 4
x = cinco - quatro
mostrar x
despejar x
frase = "retirar palavra daqui"
palavra = " palavra"
resultado = frase - palavra
mostrar resultado
despejar resultado
mostrar "Fim"
```

C.48. OPERADOR UNIÃO

```
conjunto1 = {0}
conjunto2 = {0, 1}
conjunto = conjunto1 unindo conjunto2
mostrar conjunto
despejar conjunto
```

APÊNDICE D. PROGRAMAS DE EXEMPLO

D.1. BUSCA EM PROFUNDIDADE

```
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 vértice
	mostrar “, “
```

D.2. BUSCA EM LARGURA

```
função adicionarVerticesAdjacentes conjunto, vertice
	verticesAdjacentes = verticesAdjacentesDe vertice
	iteradorDeVerticesAdjacentes = obterIteradorDe verticesAdjacentes
	statusDoIterador = existeProximoItemEm iteradorDeVerticesAdjacentes
	enquanto statusDoIterador == verdadeiro
		verticeAtual = obterProximoItemDe iteradorDeVerticesAdjacentes
		adicionar verticeAtual em conjunto
		statusDoIterador = existeProximoItemEm iteradorDeVerticesAdjacentes
	retornar conjunto

função buscaEmLarguraDosAdjacentes vertices, acao
	proximos = { }
	iteradorDosVertices = obterIteradorDe vertices
	existeProximoItemEmIterador = existeProximoItemEm iteradorDosVertices
	enquanto existeProximoItemEmIterador == verdadeiro
		vertice = obterProximoItemDe iteradorDosVertices
		statusVisitado = obterPropriedade visitado  de vertice
		se statusVisitado != "visitado"
			executar acao fornecendo vertice
```

```
                definirPropriedade visitado em vertice comValor "visitado"
                proximos = executar adicionarVerticesAdjacentes proximos,
vertice
        existeProximoItemEmIterador = existeProximoItemEm iteradorDosVertices
    iteradorDosProximos = obterIteradorDe proximos
    status = existeProximoItemEm iteradorDosProximos
    se status == verdadeiro
        executar buscaEmLarguraDosAdjacentes proximos, acao

função buscaEmLargura grafoParametro, verticeInicial, acao
    vertice = verticeInicial
    executar acao fornecendo vertice
    definirPropriedade visitado em vertice comValor "visitado"
    verticesAdjacentes = verticesAdjacentesDe vertice
    executar buscaEmLarguraDosAdjacentes verticesAdjacentes, acao

digrafo exemplo
    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

percorerGrafo exemplo utilizando buscaEmLargura iniciandoEm a
    mostrar vertice
```

D.3. DIJKSTRA

```
função defineVerticesComVisitadoFalso grafoParametro
    vertices = obterTodosOsVerticesDe grafoParametro
    iterador = obterIteradorDe vertices
    statusDoIterador = existeProximoItemEm iterador
    enquanto statusDoIterador == verdadeiro
        vertice = obterProximoItemDe iterador
        definirPropriedade visitado em vertice comValor "não visitado"
        statusDoIterador = existeProximoItemEm iterador

função definirPropriedadesParaDijkstra grafoParametro
    vertices = obterTodosOsVerticesDe grafoParametro
    iterador = obterIteradorDe vertices
    statusDoIterador = existeProximoItemEm iterador
    enquanto statusDoIterador == verdadeiro
        vertice = obterProximoItemDe iterador
        definirPropriedade estimativa em vertice comValor numeroMaximo
        definirPropriedade fechado em vertice comValor falso
        statusDoIterador = existeProximoItemEm iterador

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

função verificarProximoVertice grafoParametro
        menorEstimativa = numeroMaximo
        verticeRetorno = falso
        executar defineVerticesComVisitadoFalso grafoParametro
        percorerGrafo  grafoParametro utilizando buscaEmProfundidade iniciandoEm a
                statusDoVertice = obterPropriedade fechado  de vertice
                se statusDoVertice == falso
                        estimativaDoVertice = obterPropriedade estimativa  de vertice
                        se estimativaDoVertice <= menorEstimativa
                                verticeRetorno = vertice
                                menorEstimativa = estimativaDoVertice
        retornar verticeRetorno

função dijkstra grafoParametro, verticeInicial, verticeObjetivo
        executar definirPropriedadesParaDijkstra grafoParametro
        definirPropriedade estimativa em verticeInicial comValor 0
        verticeAtual = verticeInicial
        enquanto verticeAtual != falso
          definirPropriedade fechado em verticeAtual comValor verdadeiro
          verticesAdjacentes = verticesAdjacentesDe verticeAtual
          iteradorDeVerticesAdjacentes = obterIteradorDe verticesAdjacentes
          statusDoIterador = existeProximoItemEm iteradorDeVerticesAdjacentes
          enquanto statusDoIterador == verdadeiro
            verticeEmVerificacao = obterProximoItemDe iteradorDeVerticesAdjacentes
            pesoDaAresta = obterPesoDaAresta de verticeAtual para verticeEmVerificacao
            estimativaDoVerticeAtual = obterPropriedade estimativa  de verticeAtual
            estimativa = estimativaDoVerticeAtual + pesoDaAresta
            estimativaAntiga = obterPropriedade estimativa de verticeEmVerificacao




            se estimativa < estimativaAntiga
              definirPropriedade estimativa em verticeEmVerificacao comValor estimativa
              definirPropriedade precedente em verticeEmVerificacao comValor verticeAtual
            statusDoIterador = existeProximoItemEm iteradorDeVerticesAdjacentes
          verticeAtual = executar verificarProximoVertice grafoParametro
        retornar grafoParametro


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

verticeInicial = obterVertice a de exemploUfsc
verticeObjetivo = obterVertice d de exemploUfsc
saida = executar dijkstra exemplo, verticeInicial, verticeObjetivo
mostrar saida
```