Jonatas Teixera
Luan Haddad Ricardo dos Santos

Um editor de grafos para simulação de algoritmos

Curitiba - PR, Brasil

27 de dezembro de 2011

Jonatas Teixera
Luan Haddad Ricardo dos Santos

Um editor de grafos para simulação de algoritmos

Trabalho de conclusão de curso apresentado para obtenção do Grau de Bacharel em Ciência da Computação pela Universidade Federal do Paraná.

Orientador: Alexandre Ibrahim Direne

Curitiba - PR, Brasil

27 de dezembro de 2011

Agradecimentos

Somos gratos aos nossos pais, Fabio Ricardo dos Santos, Christina Bady Haddad Ricardo dos Santos e Attilio Teixeira Junior, Mariluz Teixeira, que foram os grandes patrocinadores da nossa graduação. Agradecemos também aos nossos amigos: Anderson, Andrio, Danilo, Derik, Diego, Felipe, Fernando e Raphael, alguns por terem estado junto de nós em projetos na universidade, outros por terem partilhado de bons momentos acadêmicos. Não podemos deixar de lembrar ao bom e velho labJoe, o qual sem ele esse trabalho não seria possível. Gostaríamos de lembrar o nome do nosso orientador Alexandre Ibrahim Direne que aceitou nos orientar na execução deste trabalho. Além disto agradecemos à Kauana de Oliveira Grola, namorada do Luan pelo apoio moral.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Esse trabalho tem a intenção de mostrar o funcionamento e a utilização do Grape que é uma ferramenta para edição de grafos e simulação de algoritmos. O Grape fornece uma interface de autoria para implementação de algoritmos e os executa de maneira gráfica através de passos iterativos, os quais facilitam a construção, compreensão e depuração dos algoritmos, dispõe de uma interface personalizável para diferentes representações dos grafos. Esta ferramenta foi projetada sendo multiplataforma, e seus algoritmos podem ser incluídos facilmente sem a necessidade de recompilações. Aqui será apresentado um breve resumo a respeito de teoria dos grafos, para que haja familiaridade com os termos e para que demais detalhes do grape possam ser expostos.

Palavras-chave: Grafos, editor de grafos, algoritmos, simulador.

1 Introdução

Durante as disciplinas de grafos e inteligência artificial ministrados no curso de graduação percebemos a carência de representações visuais de determinadas estruturas de dados, para tal fim iniciamos a implementação do Grape, que sugere uma alternativa similar ao que é comumente visto em sala de aula, onde o professor representa grafos e árvores com nós representados com pequenas esferas com algum identificador, e arestas que ligam esses nós seguindo uma lógica qualquer.

Por grafo ser um modelo matemático mais abrangente, este foi escolhido para ser abordado neste trabalho. Após a necessidade de utilizar a ferramenta GRAFO (8) na disciplina de algoritmos e teoria dos grafos enfrentamos algumas limitações, como não ser possível a confecção de multi-grafos, di-grafos e multi-di-grafos, não haver a possibilidade de adição de diferentes atributos a cada vértice ou aresta assim como a seleção de múltiplos vértices e ainda a abertura de vários grafos simultaneamente em abas ou janelas.

Apesar de grande interesse na ferramenta após uma análise da possibilidade de continuar o GRAFO, percebemos que a estrutura de dados interna causaria alguns problemas na representação de multi-grafos e a reestruturação da interface seria algo demasiadamente trabalhoso, então aproveitamos a oportunidade para introduzir novas ferramentas tecnológicas e iniciar a implementação do Grape.

2 Teoria dos Grafos

A origem da teoria dos grafos se deu a partir do problema ocorrido na cidade Königsberg aonde a população desejava saber como atravessar as sete pontes da cidade sem que a mesma ponte fosse atravessada mais de uma vez. Este problema foi estudado pelo matemático Euler que provou não haver solução.

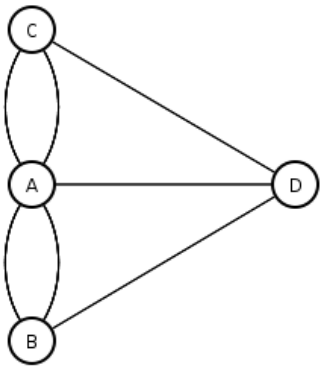

Figura 1: Grafo de Euler

Para simplificar o problema Euler ignorou alguns aspectos como largura dos rios e forma das ilhas e continentes, mantendo apenas a existência ou não de conexões entre dois pontos da cidade. Esse novo campo da matemática se tornou uma ferramenta importante para analisar problemas reais. Para compreender um pouco melhor vamos definir um grafo **G** sendo um par ordenado $\boldsymbol{G = (V, E)}$ tal que:

- **V** é um conjunto não vazio de **vértices** ou nodos.
- **E** é um conjunto de **arestas** ou arcos que por sua vez são pares de vértices distintos.

Quando dois vértices são conectados por uma aresta, estes são chamados de **adjacentes**. O **grau** de um vértice é o número de arestas que terminam no vértice, um vértice é chamado **pendant** quando o seu grau é um.

Um grafo é chamado **multi-grafo** quando existem mais de uma aresta conectando o mesmo par de vértices, e um grafo é completo quando para todos pares de vértices existe

pelo menos uma aresta conectando-os, denota-se Kn um grafo completo com n vértices, como podemos ver na figura 2 um exemplo de grafo **completo** com $n <= 6$

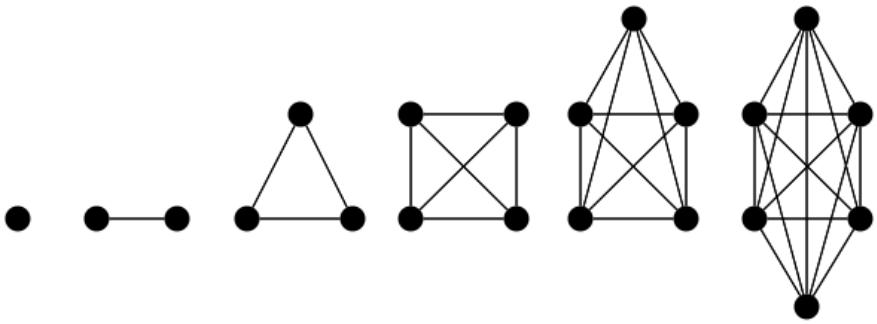

Figura 2: Grafos completos

Além dos grafos comuns, podemos ter grafos dirigidos ou digrafos, nestes grafos as arestas tem direção, ou seja, saem de um vértice e chegam a outro, sem necessariamente ter volta. Nos di-grafos o grau é muitas vezes dividido em grau de entrada e grau de saída, quantidade de arestas que chegam e que saem do vértice, respectivamente. Chamamos uma sequência de arestas em um grafo de caminho e o tamanho do caminho é dado pelo número de arestas nesta sequência.

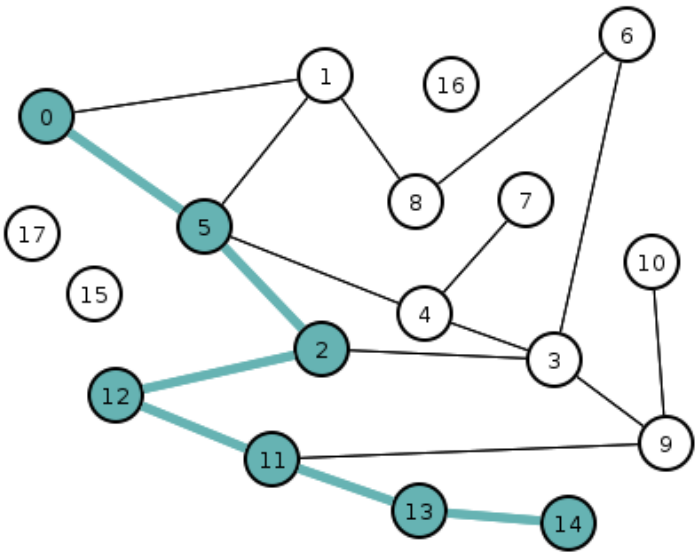

Figura 3: Caminho

Um caminho é uma sequência de arestas consecutivas em um grafo e o tamanho do caminho é o número de arestas atravessadas. A figura 3 demonstra um caminho em um grafo com início no vértice 0 e fim no vértice 14.

Um circuito ou **ciclo** é um caminho que tem início e fim no mesmo vértice, a ilustração na figura 4 representa um ciclo em um grafo, um laço é um ciclo de tamanho

um, ou seja, uma aresta que conecta um vértice a ele mesmo, grafos que possuem laços são chamados e pseudografos.

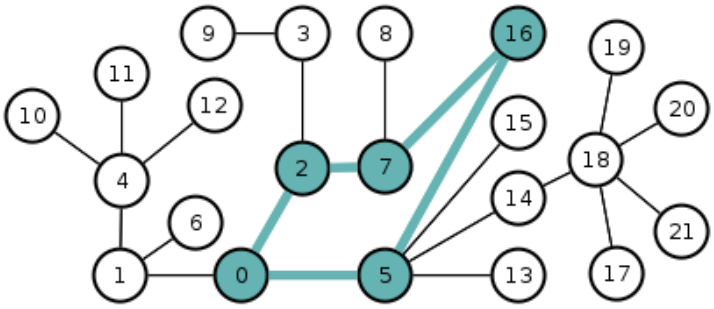

Figura 4: Ciclo

Um grafo onde existe pelo menos um caminho conectando todo par de vértice é chamado de grafo conexo, caso um grafo não seja conexo, é possível dividi-lo em **componentes conexas**, que são subgrafos conexos disjuntos, como pode ser visto na figura 5.

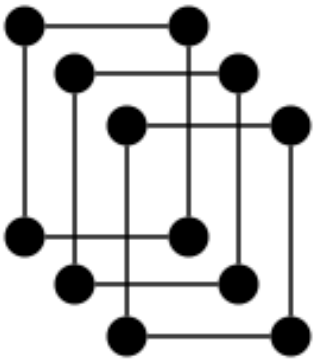

Figura 5: Grafo com três componentes conexas

Chamamos de vértice de corte ou ponto de articulação um vértice que se removido do grafo, produz um novo grafo com mais componentes conexas que anteriormente.

Um grafo planar é um grafo que pode ser representado no plano de tal forma que suas arestas não se cruzem. Segundo o Teorema de Kuratowski (7), um grafo planar não pode apresentar nem o grafo completo K5 nem o grafo bipartido K3,3 como subgrafos.

O problema explicado anteriormente das sete pontes da cidade de Königsberg ficou conhecido como caminho de Euler, ou seja, um Caminho Euleriano é um caminho em um grafo que visita cada aresta sem que a mesma seja revisitada, em casos especiais um Circuito Euleriano é um caminho Euleriano que começa e termina no mesmo vértice, dessarte, grafos que possuem um circuito Euleriano são chamados Grafos Eulerianos.

O matemático Willian Hamilton (6) criou o jogo icosiano na época da Guerra Civil

Americana que consistia em um dodecaedro sólido que forma um grafo planar de vinte vértices e trinta arestas onde cada vértice representa uma cidade (figura 6).

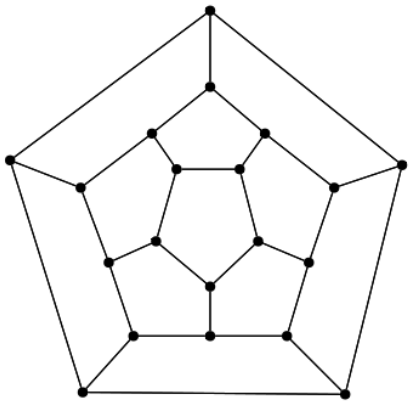

Figura 6: Grafo jogo icosiano

Este jogo tem como objetivo passar por todas as cidades uma única vez sendo que o percurso deve iniciar e terminar no mesmo vértice, formando um circuito que recebeu o nome de **circuito Hamiltoniano**. Em outras palavras um caminho Hamiltoniano é um caminho que passa por todos os vértices uma única vez, e um circuito Hamiltoniano é um caminho Hamiltoniano que inicia e termina no mesmo vértice.

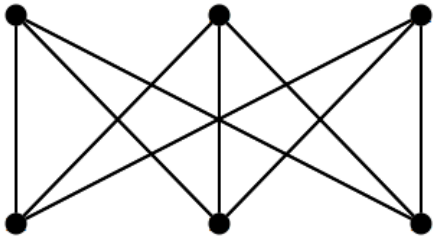

Figura 7: Grafo bipartido completo (K3,3)

Chamamos um grafo de bipartido quando seus vértices podem ser divididos em dois subconjuntos U e V. Cada aresta conecta um vértice de U a um vértice de V, nunca um vértice de V a outro de V ou de um vértice de U a outro de U.
Grafo bipartido completo é um grafo bipartido onde todos os vértices do subconjunto U são adjacentes a todos os vértices do conjunto V (figura 7). O número cromático de um grafo é o menor número de cores que são necessárias para colorir todos os vértices de modo que nenhum vértice adjacente tenha a mesma cor. Todo grafo planar pode ser colorido com no máximo quatro cores, essa propriedade chama-se Teorema das quatro cores, e podemos ver um exemplo disso na figura 8.

Figura 8: Grafo dos países e fronteiras da América do Sul

3 Estrutura de dados

Como já foi discutido neste trabalho um grafo é um modelo matemático, e para este modelo existem algumas alternativas de representações computacionais, tais como matrizes e listas de adjacência.

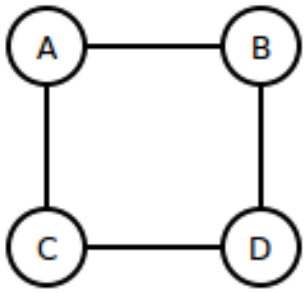

Figura 9: Exemplo de grafo

3.1 Matriz de adjacência

Para um grafo G com n vértices, podemos representá-lo em uma matriz A_{nxn}, onde cada entrada da matriz pode ser definida de acordo com as propriedades do grafo a ser representado, porém é comum o valor a_{ij} conter a informação de como os vértices se relacionam. Isto é, para representar um grafo não direcionado simples e sem pesos podemos simplesmente assumir que em cada entrada tenha 1 ou 0, tal que 1 representa a existência da conexão entre os vértices e 0 que a mesma não há. Em outras palavras, representa a presença ou não de uma aresta. Por exemplo a tabela 1 é uma matriz $5x5$ que representa o grafo da figura 9.

—	**A**	**B**	**C**	**D**
A	0	1	1	0
B	1	0	0	1
C	1	0	0	1
D	0	1	1	0

Tabela 1: Matriz de adjacência do grafo representado na figura 9.

3.2 Lista de adjacência

Uma lista de adjacência (estrutura de adjacência ou dicionário) é a representação das arestas de um grafo em uma lista. Em outras palavras, cada vértice contem uma lista de adjacência a qual representa os vértices vizinhos deste.

O grafo da figura 9 pode ser representado pela seguinte estrutura demonstrada na tabela 2

A	adjacente a	**[B, C]**
B	adjacente a	**[A, D]**
C	adjacente a	**[A, D]**
D	adjacente a	**[B, C]**

Tabela 2: Lista de adjacência do grafo representado na figura 9.

Para uma representação utilizando listas de adjacência, nós devemos manter para cada vértices uma lista de todos os outros vértices com os quais ele tem aresta.

Um contra desta estrutura é não haver um lugar para manter informações relativas à aresta, como por exemplo peso, cumprimento, cor ou custo.

3.3 Lista de incidência

Percebendo que em uma Matriz de adjacência apesar de haver a possibilidade de manter uma informação referente à aresta ainda existem muitas limitações para armazenamento de informações referentes a cada vértice. Por outro lado, na representação de Lista de adjacência é possível armazenamento de informações em cada vértice, porém não existem a possibilidade de representar demais informações para as arestas. Tal abordagem é defendida em alguns textos como o de Goodrich e Tamassia em (3) e (4).

3.4 Representação no Grape

Como no Grape, precisamos que cada vértice e aresta tenham os seus próprios atributos, utilizamos uma abordagem orientada a objetos e descrevemos uma estrutura de dados onde temos uma classe nomeada Vertex para os vértices e outra Edge para as arestas.

Um objeto Vertex tem uma lista de objetos Edge e cada objeto Edge mantem um ponteiro para os vértices de de início em fim que para grafos direcionados dão noção de direção às arestas.

Com uma abordagem orientada a objetos existe a possibilidade de armazenar mais informações para as arestas e vértices, como por exemplo: peso, cor, etc. Como no Grape existe a grande preocupação com a representação gráfica deste grafo, é dentro dessas classes o lugar propício para manter tais informações, como tamanho, espessura, posição e demais informações gráficas.

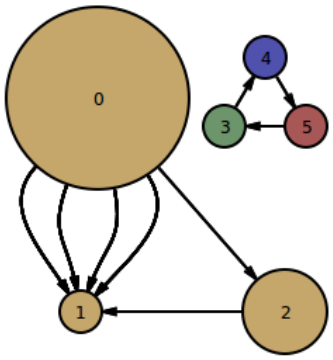

Figura 10: Exemplo de possibilidade de edição do grafo

4 Interface

Neste capítulo iremos detalhar a interface do Grape, tais como menus, atalhos, opções, ferramentas, configurações, logs, e todas as outras maneiras de interagir com o usuário. Será apresenta do a barra de menus, ferramentas, divisão e múltipla edição distribuída em abas, e área de pintura.

Nota: O Grape é multi-plataforma, isto é, sua interface pode conter leves divergências ao que esta sendo mostrado de acordo com o sistema em que esta sendo executado. Neste trabalho iremos demonstrar o software usando Linux.

4.1 Barra de menus

A barra de menu é responsável por gerenciar entradas e saídas, configuração do ambiente do Grape, edição do grafo, edição do que deve ou não ser mostrado na tela, execução de algorítimos e créditos.

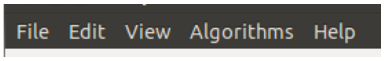

Figura 11: Menu

4.1.1 Menu File (Arquivo)

No menu do arquivo existem opções de entrada e saída do Grape, ou seja, possibilidade abrir, salvar, fechar, reverter para ultima versão salva, sair; ver figura 12

- **New (Novo)**: Fornece a possibilidade de criar um novo grafo, inicialmente vazio.
- **Open (Abrir)**: Esta opção permite que um grafo criado no Grape seja aberto novamente, permitindo reedições.
- **Save (Salvar)**: Opção responsável por salvar o grafo aberto na aba atual, em um local a ser especificado posteriormente ao usuário. (figura 12)

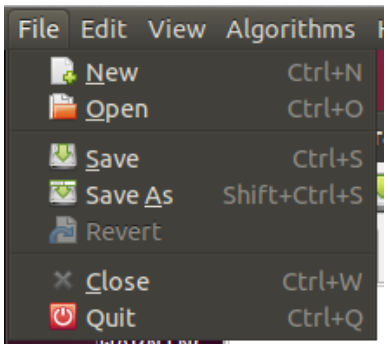

Figura 12: Menu arquivo

- **Save As (Salvar Como)**: Esta opção permite que um arquivo que já tenha sido salvo anteriormente possa ser reescrito em um caminho diferente, mantendo uma cópia no local anterior.
- **Revert (Reverter)**: Descarta todas as ações efetuadas no grafo, e retorna o seu estado para o mesmo que está salvo em disco.
- **Close (Fechar)**: Fecha a aba atual. Caso o grafo não tenha sido salvo, um diálogo irá aparecer para auxiliar nesta tarefa.
- **Quit (Sair)**: Essa ação irá fechar todas as abas e encerrar o grape, salvando os seus logs e suas configurações.

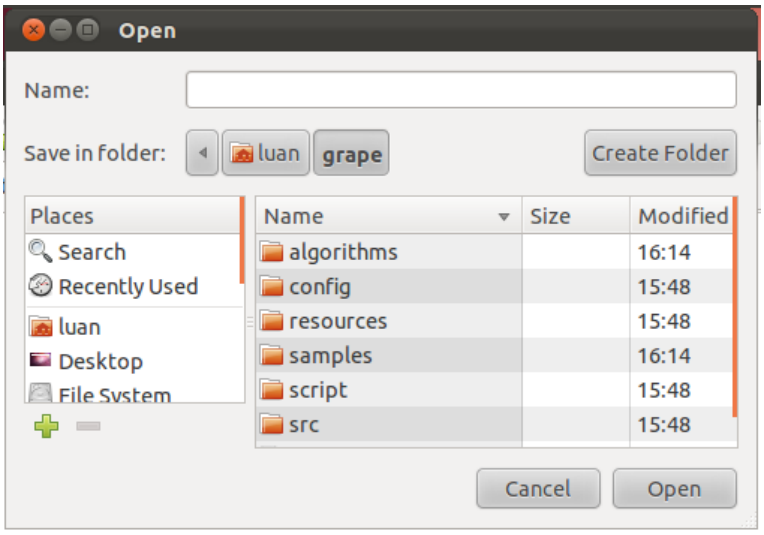

Figura 13: Abrir arquivo

4.1.2 Menu Edit (Editar)

Esse menu é responsável por todas as ações de manipulação do grafo, e preferências globais. Todas as edições e modificações afetarão o grafo da aba que estiver com foco, com

exceção da opção Preferências que irá atualizar as preferências globais do Grape.

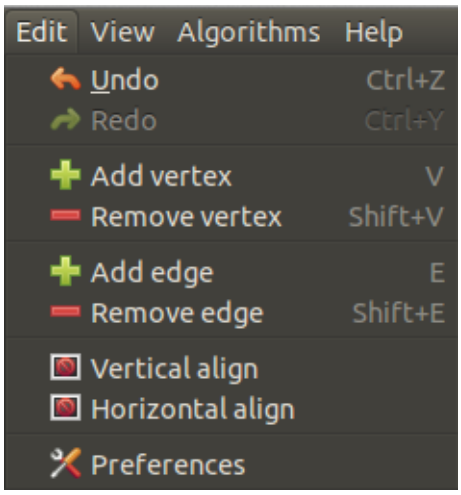

Figura 14: Menu editar

- **Undo (Desfazer)**: Toda ação efetuada em cima de um grafo é salva em uma pilha de ações, para que possa ser desfeita em caso de acidentes. A ação desfazer, desfaz a ultima ação.
- **Redo (Refazer)**: Essa ação refaz uma ação desfeita com o Desfazer.
- **Add Vertex (Adicionar Vértice)**: Habilita o modo de adição de vértices, em outras palavras, um vértice será criado assim que houver um clique com o botão esquerdo do mouse na área de pintura.
- **Remove Vertex (Remover Vértice)**: Remove o/os vértices que estejam selecionados e todas as arestas que tenham alguma conexão com os estes.
- **Add Edge (Adicionar Aresta)**: Quando esta opção for acionada, caso haja somente um vértice selecionado surgirá uma "aresta" tracejada entre o vértice selecionando e o ponteiro do mouse, desde que este esteja sobre a área de pintura, então será necessário que o usuário clique sobre um segundo vértice para que a aresta seja inserida. Em casos de múltiplos vértices selecionados, serão adicionadas arestas conectando todos os vértices entre si.
- **Vertical Align (Alinhar verticalmente)**: Esta ação calculará a média aritmética entre todos os vértices selecionados e os reposicionará na área de pintura de modo que estejam todos alinhados verticalmente.
- **Horizontal Align (Alinhar horizontalmente)**: Ação análoga ao Alinhar verticalmente, porém agora o alinhamento é vertical.

- **Preferences (Preferências)**: Ação para edição das configurações globais do Grape, tais configurações são salvas e carregadas do arquivo .grape.conf que fica na home do usuário que está rodando o Grape.
  - **Graph (Grafo)**: Aba para edições globais do grafo, tais como título, plano de fundo e tipo (Grafo, Multi-Grafo, Di-Grafo e Multi-Di-Grafo). Tais configurações só entrarão em vigor após a criação ou aberturo de um próximo grafo.
  - **Vertex (Vértice)**: Aba para edições de opções gerais para os vértices, cor de preenchimento, cor da borda, tamanho do vértice, tamanho da borda.
  - **Edge (Aresta)**: Aba para edições das propriedades das aresta, como cor e espessura. Properties (Propriedades): Atributos que serão comuns em todos os vértices ou arestas.

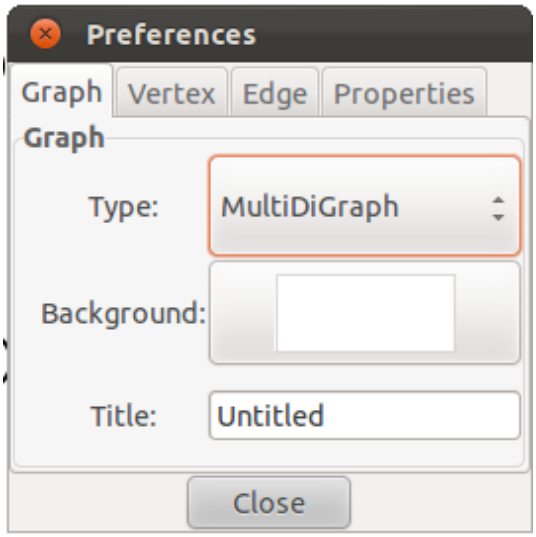

Figura 15: Preferências

4.1.3 Menu View (Ver)

Esse nenu é utilizado para personalizar itens da interface do Grape, itens que devem ou não ser exibidos e opções de aproximar ou afastar o observador da área de pintura.

- **Zoom In (Ampliar)**: Ação responsável por deixar a imagem da área de pintura mais próxima do observador.
- **Zoom Out (Afastar)**: Ação análoga a Ampliar, porém desta vez deixando a imagem da área de pintura mais distante do observador.

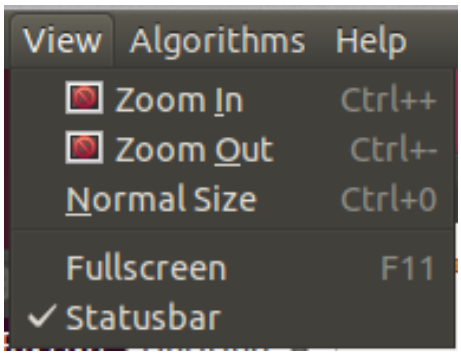

Figura 16: Menu ver

- **Normal Size (Tamanho normal)**: Restaura o tamanho da imagem da área de pintura para o tamanho normal.
- **Fullscreen (Tela cheia)**: Coloca ou remove a janela do Grape do modo tela cheia.
- **Statusbar (Barra de status)**: Mostra ou oculta a barra de status.

4.1.4 Menu Algorithms (algoritmos)

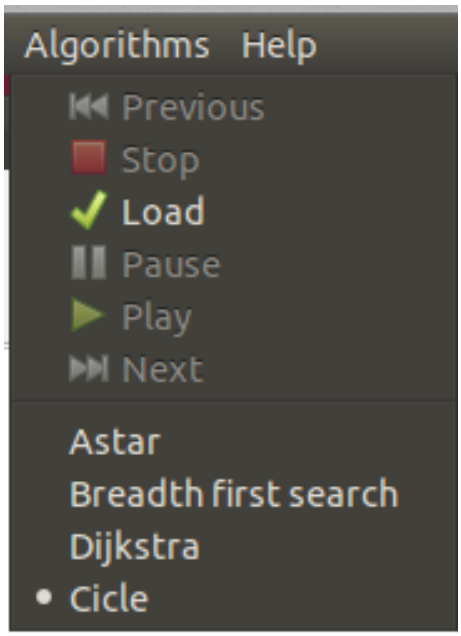

Figura 17: Menu algoritmos

Esse menu é utilizado para manipulação dos algoritmos. Quando o Grape é carregado, ele preenche automaticamente este menu com os algorítimos presentes em seu determinado diretório. E as demais ações desse menu são para manipulação destes algorítimos.

- **Previous (Anterior)**: Ação utilizada em casos de execução iterada, permite que o estado do algorítimo retroceda em uma iteração.

- **Stop (Parar)**: Interrompe a execução do algoritmo.
- **Load (Carregar)**: Carrega o algorítimo selecionado e o deixa pronto para iniciar a execução.
- **Pause (Pausa)**: Em caso de execução contínua esta ação irá passar para o modo iterativo, pausando a execução e aguardando pela próxima ação do usuário.
- **Play (Tocar)**: Inicia a execução contínua de um algorítimo que deve estar carregado com a ação Carregar.
- **Next (Próximo)**: Análogo à ação Anterior, porém agora, permite que o estado do algorítimo avance em uma iteração, para casos de execução iterada.

Os demais campos são os algorítimos de exemplo que foram carregados ao executar o Grape.

4.1.5 Menu Help (Ajuda)

Este menu apenas dirige o usuário para detalhes simples do Grape, na tela de about onde há um link para o código fonte e os créditos dos desenvolvedores.

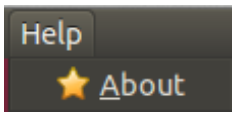

Figura 18: Menu Ajuda

Figura 19: Tela About

4.2 Atalhos e usabilidade

O grape possui uma variedade de teclas de atalhos para acelerar o processo de criação e edição dos grafos, tais atalhos serão listados e brevemente descritos abaixo.

- **Ctrl + N**: Abre um novo grafo em uma nova aba
- **Ctrl + O**: Abre diálogo para abrir um grafo
- **Ctrl + S**: Salvar grafo
- **Ctrl + Shift + S**: Salvar grafo especificando um local para ser salvo
- **Ctrl + Z**: Desfaz uma ação
- **Ctrl + Y**: Refaz uma ação
- **Ctrl + A**: Seleciona todos os vértices
- **V**: Inserir vértice
- **Shift + V**: Remover vértice
- **E**: Inserir aresta
- **Shift + E**: Remover aresta
- **Ctrl + +**: Aproximar
- **Ctrl + -**: Afastar
- **Ctrl + 0**: Voltar ao tamanho original
- **F11**: Exibir o grape um tela cheia

Setas direcionais: Selecionam o próximo vértice mais próximo do seguindo a direção da seta pressionada.

Uma forma alternativa de editar um vértice é simplesmente clicar com o botão direito do mouse sobre um vértice selecionado, E então será exibido um menu com opções de ações a serem efetuadas sobre este vértice, como mostrado na figura 20.

Ao escolher editar as configurações de um vértice quer tenha sido pela barra de ferramentas ou pelo menu de propriedades do vértice será mostrado um diálogo que possibilita a personalização do vértice selecionado, onde há a possibilidade de editar o título do

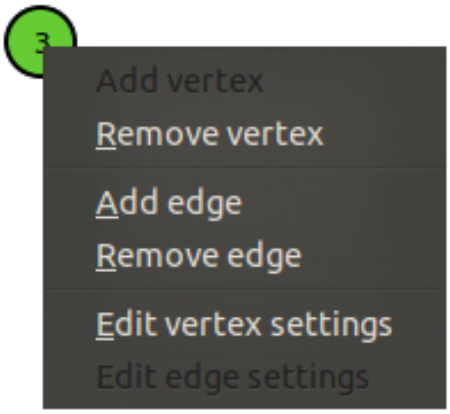

Figura 20: Menu Vértice

vértice, (o qual é exibido na área de pintura), as coordenadas, a cor de preenchimento e da borda, o tamanho e a espessura da borda, também tem a possibilidade de listar todas as arestas que pertencem a esse vértice e de adicionar atributos personalizados. Por exemplo, na sessão Properties da edição de configurações de um vértice há um campo aonde existe uma tabela de atributos personalizados o qual possibilita ao usuário adicionar atributos quaisquer, que poderão ser acessados e/ou alterados pelos métodos set_attribute e get_attribute, no momento da execução dos algoritmos que serão vistos e explicados ao decorrer deste trabalho, os atributos personalizados funcionam como variáveis, ou seja, possuem um identificador e armazenam um valor, [identificador, valor].

Para que seja possível a edição de atributos personalizados das arestas é necessário que no menu de configurações do vértice "dono" da aresta, na aba edge uma aresta seja selecionada, e com o clique direito do mouse a opção de edição de aresta seja escolhida, só então abrirá uma nova tela para edições de atributos e propriedades da aresta escolhida.

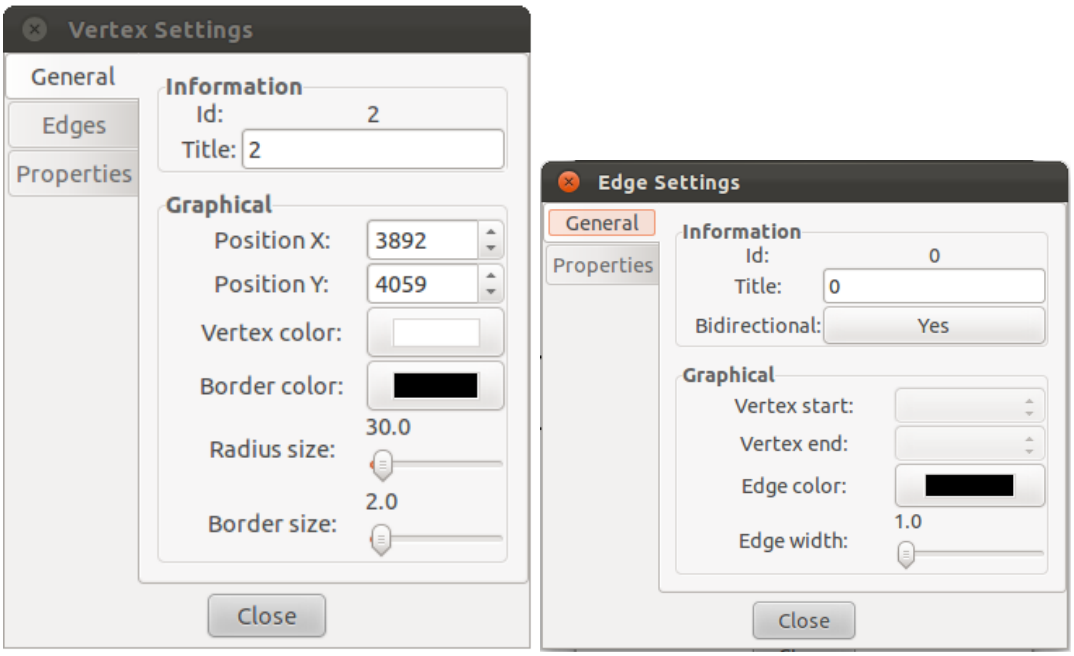

Figura 21: Configurações de vértice e aresta

5 algoritmos

Implementamos no Grape um arcabouço para implementação de algoritmos, como ele foi implementado em python usando orientação a objetos seus algoritmos seguem a mesma linha. Neste capítulo iremos explicar algumas estruturas internas e algumas funções prontas para auxiliar a construção.

Os algoritmos podem ou não ser compilados para que sejam carregados, basta que eles estejam no diretório algorithms da raiz do Grape, antes dele ser executado, pois no momento da execução ele irá carregar dinamicamente todos os algoritmos que estiverem dentro daquele diretório e irá inserir cada um deles como entradas no menu Algoritmhs, como visto no Capítulo 5.

5.1 Super Classe

Todo algoritmo deve ser uma classe e estender uma super classe Algorithm para que então tenha acesso aos métodos de manipulação do grafo e da interface gráfica da representação do algoritmo.

```
    from lib.algorithm import Algorithm
```

Métodos estendidos da super classe:

- **check(object)**: Método que dará destaque a um objeto (vértice ou aresta) no próximo ciclo de execução.
- **uncheck(object)**: Analogamente ao método check, remove o destaque do objeto no próximo ciclo de execução.
- **show(object)**: Finaliza um ciclo de execução e mostra as alterações na área de pintura.
- **find(id)**: Busca um vértice por id no grafo e o retorna seu objeto.
- **set_attribute(object, attribute, value)**: Cria ou altera um valor de um novo atributo personalizado do objeto.

- **get_attribute(object, attribute)**: Obtém o valor de um atributo personalizado criado com o set_attribute, este atributo deverá ter sido criado usando o mesmo valor attribute no mesmo objeto object.
- **remove_attribute(object, attribute)**: Remove um atributo attribute do objeto, após utilizar esse método o atributo deixará de existir, portanto não será mais acessível.
- **input_box(descricao, texto, texto_secundario)**: Cria uma caixa de texto, pedindo uma entrada para o usuário, e retorna um string contendo o valor de entrada fornecido pelo usuário.

Atributos estendidos da super classe:

- **vertex_list**: Lista com todos os vértices do grafo.
- **edge_list**: Lista com todas as arestas do grafo.

Quando o algoritmo for executado a super classe vai chamar automaticamente o método run do algoritmo, logo este dever ser implementado no código do algoritmo de usuário. Também o nome do arquivo e da classe devem ser compatíveis: o nome do arquivo usar underline_case e o da classe usar CamelCase. Por exemplo:

- Arquivo: *breadth_first_search.py*
- Classe: *BreadthFirstSearch*

Ainda, no método construtor deverá ser feita a chamada ao construtor da super classe Algorithm, como visto no código abaixo:

```
class BreadthFirstSearch(Algorithm):
    def __init__(self, graph):
        Algorithm.__init__(self, graph)
```

Dentro desta classe podem ser criados quantos atributos e métodos forem necessários.

5.2 Construindo Algoritmos

Com o intuito de facilitar a construção de algoritmos usando o arcabouço do Grape fizemos um guia passo-a-passo:

Crie um arquivo python na pasta algorithms utilizando como nome o nome de seu algoritmo, procure sempre ser descritivos nos nomes, economizando abreviações. Vamos fazer um algoritmo de busca em profundidade (Depth-First Search), como descrito no livro (2), portanto usaremos o seguinte nome:

$ touch algorithms/depth_first_search.py

Edite este arquivo usando seu editor favorito.

$ vim algorithms/depth_first_search.py

Dentro dele crie uma classe utilizando o mesmo nome do arquivo, mas desta vez em CamelCase ao inves de underline_case. Isto é muito importante pois internamente o Grape utiliza o padrão dos nomes para incluir seu algoritmo.

```
from lib.algorithm import Algorithm

class DepthFirstSearch(Algorithm):
    def __init__(self, graph):
         Algorithm.__init__(self, graph)
```

Note que está parte do algoritmo é sempre igual. Nunca esqueça de estender da classe Algorithm e de definir o método construtor dessa forma, qualquer tentativa diferente disto implicará no não funcionamento de sua implementação. O método __init__ é o método construtor, podemos implementar mais coisas nele após a primeira linha, ele será executado automaticamente quando o usuário carrega o algoritmo na interface do Grape.

O algoritmo de busca em profundidade parte de um vértice do grafo e tenta encontrar o destino expandindo cada aresta em forma de pilha, ou seja, a última aresta encontrada é a primeira a ser expandida. Para isso devemos saber o vértice origem e o vértice destino. Implementaremos no método construtor duas input_box's para que o usuário possa escolhe-los.

```
def __init__(self, graph):
        Algorithm.__init__(self, graph)
        # input box para capturarmos o vertice origem
        self.first_id =
int(self.input_box('Escreva o numero do vertice origem', 'Origem'))
        # input box para capturarmos o vertice destino
        self.goal_id =
int(self.input_box('Escreva o numero do vertice destino', 'Destino'))
```

Feito isto é hora de escrever o código do seu algoritmo, para isto, cria-se um método chamado run, o método run é executado automaticamente quando o usuário executa o algoritmo na interface do Grape. Portanto, devemos começar implementando o método run.

A primeira coisa que faremos é encontrar quais são os vértices baseando-se no que foi digitado pelo usuário:

```
    def run(self):
        first = self.find(self.first_id)
        goal = self.find(self.goal_id)
```

Tendo isso vamos criar nossa pilha de execução para adicionarmos as arestas e vértices a serem expandidos. Em python uma pilha é uma lista, apenas utilizada de forma correta.

```
        stack = []
# utlizaremos uma pilha para nossa busca em profundidade

        # adicionamos nosso inicio na pilha
        stack.append((first, None))
        # uma tupla (vertice, aresta). Como este e' o inicio nao
        # utilizamos nenhuma aresta para alcanca-lo
```

No início, nenhuma aresta foi visitada, portanto devemos marcar todas como "não visitadas":

```
        # marcamos todas as arestas como nao visitados
        for e in self.edge_list:
            self.set_attribute(e, 'visited', 'no')
```

Começamos agora o algoritmo, ele deverá ser executando enquanto houverem elementos na pilha:

Capturamos o elemento no topo da pilha (em python o topo da pilha é o último elemento da lista, denotado por lista[-1]), marcamos como vistado e marcamos a aresta origem e o vértice para visualização:

```
        while len(stack) > 0:
            node = stack[-1]
            if node[1]:
                self.set_attribute(node[1], 'visited', 'yes')

            self.check(node[0])
            self.check(node[1])
            self.show()

            # Caso o vertice seja o destino, paramos por aqui o
            # algoritmo:
            if node[0].id == goal.id:
                rtn = node[0]
                break
```

Caso contrário verificaremos se ainda há arestas a serem expandidas a partir deste

vértice, isto se dá caso haja alguma aresta não visitada na vizinhança do vértice. Se não houver arestas a serem expandidas o vértice é removido da pilha e chamamos a próxima iteração do while.

```
            else:
                pop = True
                for edge in node[0].edge_list:
                    if self.get_attribute(edge, 'visited') == 'no':
                        pop = False
                        break

                if pop:
                    self.uncheck(node[0])
                    self.uncheck(node[1])
                    stack.pop()
                    continue
```

Agora o algoritmo em si, na vizinhança do vértice atual procuraremos as arestas não visitadas e adicionaremos ela (juntamente com o vértice destino em questão) na pilha para ser expandida na próxima iteração.

```
            for edge in node[0].edge_list:
                if self.get_attribute(edge, 'visited') == 'no':
                    if edge.start == node[0]:
                        stack.append((edge.end, edge))
                    else:
                        stack.append((edge.start,edge))
```

6 Conclusão

Com o Grape um ambiente de desenvolvimento é oferecido, aonde um aprendiz pode implementar através de uma linguagem de alto nível algoritmos à serem executados sobre uma estrutura de grafos. Partindo do premissa ideológica que uma lista é um tipo específico de árvore, e por sua vez uma árvore é um tipo específico de grafo, assim temos um grafo sendo a estrutura mais genérica para a representação em um ambiente de simulação iterativa de algoritmos

É esperado de uma ferramenta de apoio para o aprendizado de algoritmos que ela consiga de alguma maneira expor uma ambiente de depuração para que o aprendiz visualize o funcionamento do que está implementando, vendo possíveis erros à serem corrigidos ou até entendendo melhor o fluxo do algoritmo.

Como o Grape tem o mesmo propósito do Grafo, porém com outras representações e outra interface, podemos afirmar que a utilização de ferramentas como estas é muito proveitosa para aprendizado, tendo sido nós mesmos usuários deste ambiente.

Este trabalho conseguiu cumprir os objetivos traçados inicialmente para o Grape, fazendo o desenvolvimento técnico, teórico e conceitual de tudo que fora planejado.

7 Trabalhos Futuros

Este trabalho deixa margem para variadas evoluções e em diversas vertentes.

A integração parcial com o NetworkX (5) (Biblioteca de grafos para python) já foi implementada, porém a interface ainda não faz a comunicação necessária para exportar e importar arquivos nos formatos suportados pela biblioteca.

Além da interface gráfica GTK+ planejamos criar uma interface de linha de comando, tanto interpretativa quanto em modo batch, porque num ambiente de sala de aula onde o professor passa trabalhos de implementação é interessante que haja maneira de testar várias implementações com diferentes entradas de forma a comparar apenas o resultado sem ter o overhead de carregar toda a interface gráfica e manualmente executar cada uma das simulações para cada uma das entradas previstas.

Gostaríamos de publicar nosso simulador de grafos na *GraphDrawing conference* (1), uma conferência internacional de desenhos de grafos.

Como grafo é um modelo bastante genérico há a possibilidade de representar máquinas de estado em todas as suas variações.

Referências

1 Graph drawing. http://graphdrawing.org/.

2 CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, second edition ed. MIT Press and McGraw-Hill, 2001.

3 GOODRICH, M. T., AND TAMASSIA, R. Estruturas de dados e algoritmos em java. *Porto Alegre: Bookman* (2001), 502–503.

4 GOODRICH, M. T., AND TAMASSIA, R. Projeto de algoritmos: Fundamentos, análise e exemplos da internet. *Porto Alegre: Bookman* (2004), 299–303.

5 HAGBERG, A., SCHULT, D., SWART, P., CONWAY, D., SÉGUIN-CHARBONNEAU, L., ELLISON, C., EDWARDS, B., AND TORRENTS, J. Networkx. http://networkx.lanl.gov/.

6 HAMILTON, W. R. Account of the icosian calculus. *Proceedings of the Royal Irish Academy* (1858).

7 KURATOWSKI, K. Sur le problème des courbes gauches en topologie. *Fund. Math. 15* (1930), 271–283.

8 PEREIRA, U. C. Grafo - graph editor. *Trabalho de graduação em Bacharelado em Ciência da Computação da Universidade Federal do Paraná* (2008).