UNIVERSIDADE FEDERAL DO PARANÁ

SETOR DE CIÊNCIAS EXATAS

DEPARTAMENTO DE INFORMÁTICA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GRAFO – graph editor

CURITIBA

JULHO 2008

ULISSES CORDEIRO PEREIRA

GRAFO – graph editor

Trabalho de graduação apresentado como requisito parcial à obtenção de grau de Bacharel em Ciência da Computação. Departamento de Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: *André Luiz Pires Guedes*, Departamento de Informática, UFPR.

CURITIBA

JULHO 2008

Sumário

Lista de Figuras

Lista de Tabelas

AGRADECIMENTOS

Primeiramente gostaria de agradecer aos meus pais pela formação que eles me deram. O que proporcionou, apesar das dificuldades, concluir a universidade. Gostaria de agradecer também a minha namorada *Gizelle Marques* pelo incentivo, motivação e carinho sempre nos momentos certos. Agradeço também ao *Oliver Matias van Kaick* que mesmo fazendo doutorado no Canada foi prestativo em responder dúvidas e em dar dicas para que eu pudesse iniciar o trabalho com o **GRAFO**. E por final agradeço ao professor *André Luiz Pires Guedes* por ter aceitado realizar a minha orientação mesmo não tendo realizado orientação do meu trabalho de graduação I.

1 INTRODUÇÃO

Esse documento tem a intenção de ser um tutorial para se trabalhar com o **GRAFO - graph editor**. O **GRAFO** é uma ferramenta de ensino e de testes para trabalhos com *grafos*. Ele proporciona a construção, edição, manipulação de *grafos* e testes de algoritmos. Através da sua interface de *plugins* pode-se facilmente construir um novo algoritmo e testa-lo. O usuário não precisa se preocupar com a interface gráfica, esta é fornecida pelo **GRAFO**. Com isso ganha-se em tempo e produtividade. Também neste trabalho foi introduzido a possibilidade de construção de algoritmos *passo a passo*. Esta funcionalidade fornece ao usuário a possibilidade de interagir e depurar os algoritmos por ele construídos.

1.1 Histórico

O **GRAFO** surgiu em 2003 com uma idéia de *Murilo Vicente Goncalves da Silva* de construir um programa que facilitasse os testes de algoritmos. A primeira versão do programa possuía uma interface OpenGL e alguns poucos algoritmos nativos.

Algum tempo depois a idéia foi encampada por *Oliver Matias van Kaick*. Nesta época o *Murilo Vicente* e o *Oliver Matias* eram do PET (Programa de Educação Tutorial) do curso de Ciência da computação da UFPR (Universidade Federal do Paraná). A dedicação dos dois somadas a tutoria e experiência do professor *André Luiz Pires Guedes*

permiti-os a construção de toda base que existe atualmente: Interface GTK e OpenGL, comunicação com o GraphViz, utilização de plugins, diferentes layouts para desenho de *grafos*, rótulos e pesos em vértices e arestas, entre outros.

Em 2008, motivado por esse trabalho e junto com o professor *André Guedes* decidimos colocar o **GRAFO** de novo em ação realizando algumas manutenções e implementando as estruturas que possibilitassem a execução de algoritmos *Passo a Passo*. Entretanto as primeiras investigações sobre o **GRAFO** abriram outras possibilidades até então não exploradas que se refletem na parte de documentação que também está sendo coberta por este trabalho.

1.2 Organização deste trabalho

O que veremos nos próximos capítulos dará ao leitor uma noção de teoria de grafos, um detalhamento da interface e funcionalidades e um *HowTo - Como Fazer* para construir plugins e implementar algoritmos *passo a passo*. Além disso esta documentação serve como ponto de referência para futuras manutenção e ou implementações nas funcionalidades do **GRAFO**.

Este trabalho está organizado em 7 capítulos:

- Introdução

Este capítulo

- Conceitos

Os conceitos introduzem um pouco de teoria de grafos ao leitor com o intuito de contextualizá-lo em nomenclaturas e conceitos para que melhor consiga utilizar os recursos do **GRAFO**.

- Interface

O capítulo de interface detalha todas as opções, configurações, menus, barra de ferramentas e recursos existentes no **GRAFO**.

- Plugins

O capítulo de plugins mostra através de exemplos práticos e evolutivos como construir plugins e como faze-los interagir com a interface do **GRAFO**.

- Passo a Passo

Complementando a construção de plugins o capítulo de *passo a passo* detalha como o usuário pode tirar proveito deste novo recurso.

- Considerações finais

Este capítulo detalha o que foi realizado por este trabalho e expõem algumas idéias na intenção de convidar ao leitor a novas possibilidades de implementações e/ou manutenções.

2 CONCEITOS

Neste capítulo será dada uma breve introdução sobre teoria dos grafos. Essa introdução tem por objetivo contextualizar o leitor para trabalhar com a nomenclatura utilizada no **GRAFO**.

Tanto em matemática quanto em ciência da computação, a teoria dos grafos estuda a estrutura matemática *grafo* que são utilizados para modelar a relação entre objetos de uma certa coleção. Neste contexto um grafo é uma coleção de vértices (nodos) e arestas conectando um par de vértices. WIKIPEDIA (2008).

2.1 Grafo

Segundo GAGNON (2001) e SZWARCFITER (1984) um grafo ***G*** é um par ordenado ***G*** = *(V, E)* que está sujeito as seguintes condições:

- ***V*** é um conjunto não-vazio de elementos chamados de vértices ou nodos. Vértice é a unidade fundamental da composição de um *grafo* sendo este indivisível.

  Como exemplo temos o conjunto ***V***, onde, ***V*** = *{v0, v1, v2, v3, v4, v5}* e $v0..v_n$ são vértices.

- ***E*** é um conjunto de arestas que por sua vez são pares de vértices distintos e não ordenados.

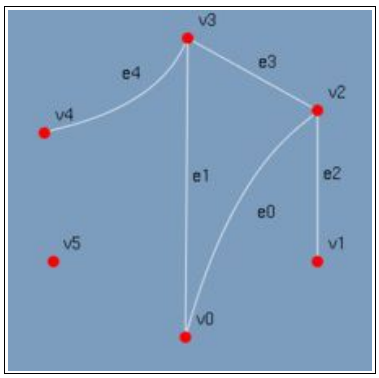
  Como exemplo temos o conjunto ***E***, onde, ***E*** = *{e0, e1, e2, e3, e4}* e $e0..e_n$ são arestas. Cada aresta de ***E*** é um par não ordenado e

distinto de vértices sendo: ***e0*** = *(v0, v2)*, ***e1*** = *(v0, v3)*, ***e2*** = *(v1, v2)*, ***e3*** = *(v2, v3)*, ***e4*** = *(v3, v4)*.

2.2 Representação gráfica de um grafo

A figura 2.1 traz a representação gráfica do *grafo* ***G*** acima, onde, ***G*** = *({v0, v1, v2, v3, v4, v5}, {(v0, v3), (v0, v2), (v0, v3), (v1, v2), (v2, v3), (v3, v4)})*.

Os seus vértices correspondem a pontos distintos do plano em posição aleatória e suas arestas correspondem a retas ou curvas que unem estes pontos. SZWARCFITER (1984).



(figura 2.1)

2.3 Incidência, grau e adjacência

Uma aresta incide aos dois vértices que conecta. Um vértice é incidente a uma aresta a ele conectada. O grau de um vértice é dado pelo número de arestas incidentes a ele. SZWARCFITER (1984). Na figura 2.1 temos como grau dos vértices o seguinte: ***v0*** = *2*, ***v1*** = *1*, ***v2*** = *3*, ***v3*** = *3*, ***v4*** = *1*, ***v5*** = *0*.

Os vértices adjacentes são vértices unidos por uma aresta, na figura 2.1 os pares de vértices não ordenados *(v0, v2), (v0, v3), (v1,*

v2), (v2, v3) e (v3, v4) são adjacentes. Duas arestas são adjacentes quando compartilham um mesmo vértice. No exemplo os pares de arestas não ordenadas *(e0, e1), (e0, e2), (e0, e3), (e1, e3), (e1, e4), (e2, e3) e (e3, e4)* são adjacentes.

2.4 Peso de vértices e arestas

Peso é um valor associado a vértices e arestas de um grafo. Por exemplo um custo, uma distância, uma capacidade ou algo que mensure ir de um vértice a outro utilizando uma determinada aresta ou passando por um determinado vértice. SZWARCFITER (1984).

2.5 Caminho, trajeto, ciclo

É denominado *caminho* em um *grafo* uma seqüencia de vértices onde para cada vértice existe uma aresta que o conecta ao vértice seguinte. Um *caminho* de *K* vértices possui um caminho de *comprimento K-1*. Se os vértices do caminho não se repetirem a seqüencia recebe o nome de *caminho simples*. Se as arestas forem únicas a seqüencia recebe o nome de *trajeto*. Um *ciclo* é um *caminho* que começa em um vértice e retorna ao mesmo vértice. SZWARCFITER (1984).

2.6 Tipos de Grafos

Neste tópico serão exemplificado alguns tipos de grafos citados na interface do **GRAFO**. São eles: *Grafo completo*, *Grafo bipartite*, *Grafo regular*, *Grafo estrela*.

2.6.1 Grafo completo

Em um *grafo completo* todo par de vértice é ligado por uma aresta, isto significa que o grafo possui o máximo de arestas possíveis.GAGNON (2001).

Um grafo com *n* vértices é nomeado K_n. *K1* um grafo com um único vértice e chamado *trivial*. *K2* é um grafo com 2 vértices e uma aresta. *K3* é um grafo com 3 vértices e 3 arestas e assim sucessivamente. O número de arestas de um *grafo completo* é dado por *n(n-1)/2*.

A figura 2.2 traz o exemplo de um *K8* que é um *grafo* com 8 vértices e 28 arestas.

2.6.2 Grafo bipartite

Em um *grafo bipartite* os vértices são divididos em dois grupos. Sendo que cada vértice do primeiro grupo é adjacente a somente vértices do segundo grupo e vice-versa. De uma maneira mais formal em um *grafo* bipartite não existe *ciclo* de comprimento ímpar. SZWARCFITER (1984).

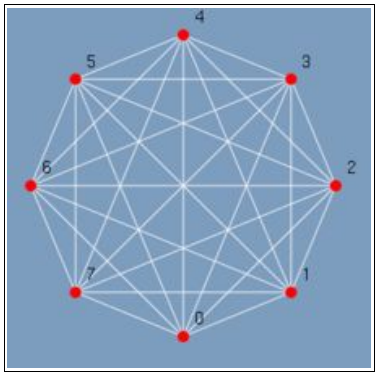
A figura 2.3 traz um exemplo de um *grafo bipartite* com 5 vértices no primeiro grupo e 3 vértices no segundo.

2.6.3 Grafo regular

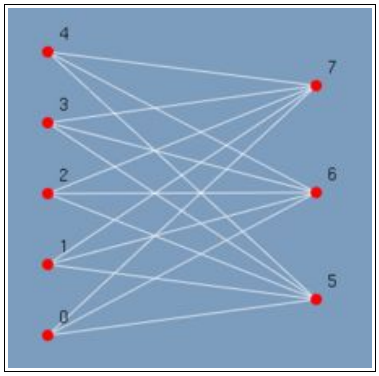
É denominado um *grafo regular* um *grafo* onde todos os vértice tem o mesmo *grau*. Como no exemplo da figura 2.4 que traz um *grafo regular* com 10 vértices e onde todos os vértices possuem grau 2. GAGNON (2001).

2.6.4 Grafo estrela

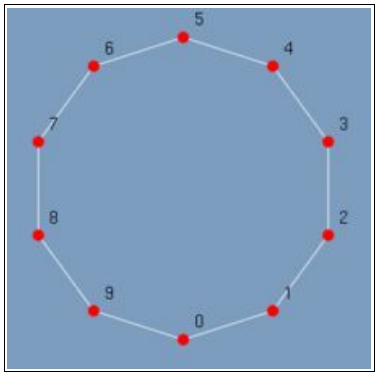
Um *grafo estrela* é um *grafo* onde existe um vértice central que é adjacente a todos os outros vértices do *grafo*. Como na figura 2.5.



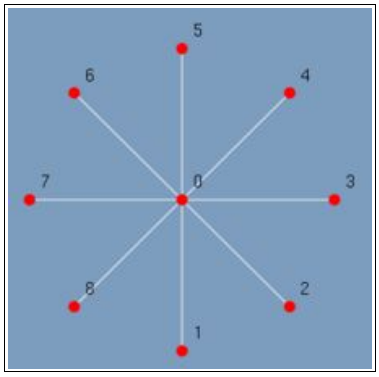
(figura 2.2)



(figura 2.3)



(figura 2.4



(figura 2.5)

2.7 Representação computacional de um grafo

Neste tópico será abordado uma das maneiras de se representar um *grafo* em um computador. Esta não é a única maneira de representação de um grafo, porem, é a maneira utilizada internamente pelo **GRAFO**. Por GAGNON (2001) e SZWARCFITER (1984).

2.7.1 Matriz de Adjacência

Dado o *grafo* ***G*** *= (V, E)* de *n* vértices *v0, v1, v2..*v_n a *matriz de adjacências* ***M*** *=* (m_{jk}) é uma matriz *n x n* sujeito as seguintes condições:

1. $\boldsymbol{m}_{jk}$ = 1 se o par de vértices (v_j, v_k) pertence a ***E***.
2. $\boldsymbol{m}_{jk}$ = 0 se não pertence.

Como exemplo a tabela 2.1 que representa a *matriz de adjacência* do grafo da figura 2.1.

--	v0	v1	v2	v3	v4	v5
v0	0	0	1	1	0	0
v1	0	0	1	0	0	0
v2	1	1	0	1	0	0
v3	1	0	1	0	1	0
v4	0	0	0	1	0	0
v5	0	0	0	0	0	0

(tabela 2.1)

3 INTERFACE

Neste capitulo será detalhado a interface que o **GRAFO** provem ao seu usuário, como menus, botões, opções, configurações e tudo que possa sofrer a interação do usuário. A expansão desta interface, com a criação de novos plugins, será abordada nos capítulos 4 e 5.

A interface do grafo, figura 3.1, possui quatro divisões básicas. São elas:

1 **Menus**

Oferece ao usuário a acesso operações de entrada e saída, edição, manipulação de *grafos*, vértices e arestas, execução de algoritmos e configurações do ambiente do **GRAFO**. Destacado em amarelo na figura 3.1.

2 **Barra de ferramentas**

Contem alguns botões de atalhos para operações básicas de entrada e saída e botões para interação em algoritmos com suporte a *passo a passo*. Destacado em vermelho na figura 3.1.

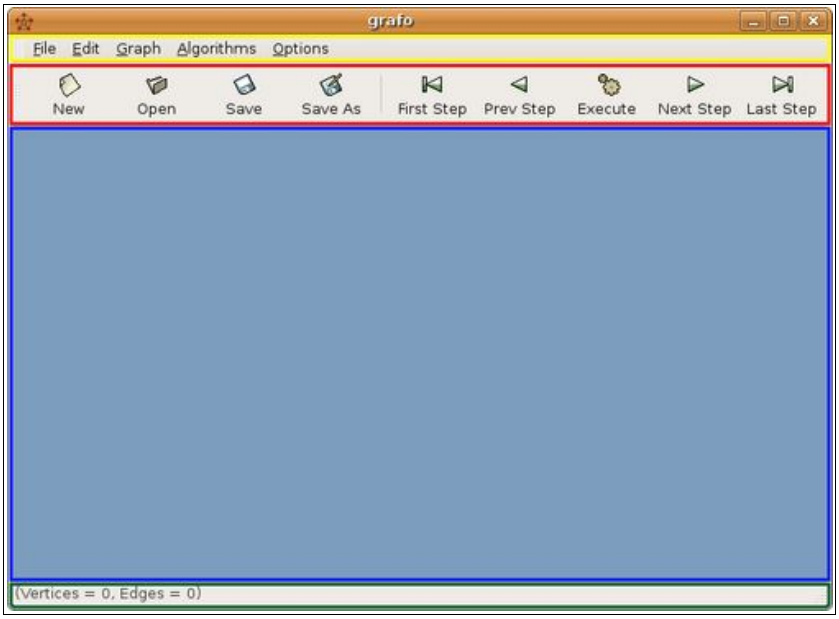
3 **Área de trabalho**

Área onde o usuário desenha e interage com o seu *grafo*. Como a inclusão e exclusão de vértices e arestas, edição de propriedades como rótulos e pesos entre outros. Também e a área em que o usuário vê o resultado da ação de um algoritmo em um *grafo*. Como destaque de vértice e arestas, a inclusão e exclusão de

componentes, a alterações de layout e outros ações sofridas pelo *grafo* da área de trabalho. Destacado em azul na figura 3.1.

4 **Barra de status**

Exibe instruções e informações ao usuário e também o resultado de algoritmos. Destacado em verde na figura 3.1.



(figura 3.1)

3.1 Menus

Os menus do **GRAFO** são divididos em cinco categorias: *File*, *Edit*, *Graph*, *Algorithms* e *Options*. Estas por sua vez estão subdivididas de maneira a conter operações correlatas.

3.1.1 File

Esta opção de menu, figura 3.2, contém as operações de entrada e saída para o **GRAFO**.

(figura 3.2)

- **New**

Limpa as estruturas internas e a área de trabalho preparando o ambiente para uma nova edição. O programa não pede confirmação para essa operação caso algum *grafo* esteja em edição este será perdido. Se um arquivo estiver aberto a sua edição também será encerrada.

- **Open**

Abre um *grafo* previamente salvo. Figura 3.3. O **GRAFO** trabalha com dois tipos de arquivos de entrada:

1. *Grafo format (.gr)* - Formato Nativo
2. *GraphViz format (.dot)* - Formato do GraphViz

(figura 3.3)

Para maiores detalhes sobre o formato de arquivo GraphViz visite o site (http://www.graphviz.org/).

- **Save**

Salva o *grafo* em edição. Figura 3.4. O **GRAFO** trabalha com três tipos de arquivo saída:

1. *Grafo format (.gr)* - Formato Nativo
2. *GraphViz format (.dot)* - Formato do GraphViz
3. *PostScript format (.ps)* - Formato PostScript

(figura 3.4)

- **Save As**

Salva o arquivo previamente carregado com um novo nome. O diálogo é o mesmo da opção *save*.

- **About**

Exibe o nome dos autores do **GRAFO** e as informações de copyright. Figura 3.5.

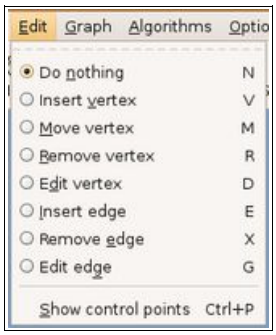
(figura 3.5)

- **Exit**

Sai do **GRAFO**. O programa não pede confirmação para esta operação caso algum *grafo* esteja em edição ele será perdido.

3.1.2 Edit

Esta opção de menu contem as operações para construção de um *grafo* e a edição de atributos e propriedades. Figura 3.6.



(figura 3.6)

- **Do nothing - N**

Encerra qualquer operação de edição que esteja ocorrendo.

- **Insert vertex - V**

Insere um vertice na posição do cursor quando o botão esquerdo do mouse é pressionado. Os vértices inseridos são numerados de 0 até N.

- **Move vertex - M**

Move o vértice selecionado ao movimentar o mouse com o botão esquerdo pressionado sobre ele. Ao soltar o botão o vértice permanecerá no local escolhido.

- **Remove vertex - R**

Exclui o vértice selecionado ao pressionar com o botão esquerdo do mouse sobre ele. Os números dos vértices são rearranjados de forma que não fiquem espaços entre as numerações. Os *Ids - Identificadores* dados aos vértices continuam os mesmos. Vide opção *Edit vertex*.

- **Edit vertex - D**

Edita as propriedades do vértice selecionado ao pressionar com o botão esquerdo do mouse sobre ele. A figura 3.7 mostra o diálogo de edição de vértices.

As propriedades disponíveis são:

1. *Id* - Número único que identifica o vértice (Não editável)
2. *Label* - Rótulo dado ao vértice
3. *Weight* - Peso deste vértice
4. *Color* - Cor que ele será impresso na tela

- **Insert edge - E**

Insere uma aresta entre os vértices ao pressionar com o botão esquerdo do mouse primeiramente em um vértice e em seguida em um segundo vértice. Em um *Digrafo* a ordem que os vértices

são selecionados é importante, pois determinam o sentido do arco.

- **Remove edge - X**

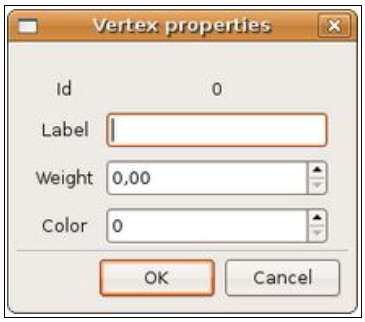
Remove uma aresta entre dois vértices ao pressionar com o botão esquerdo do mouse primeiramente em um vértice e em seguida em um segundo vértice.

- **Edit edge - G**

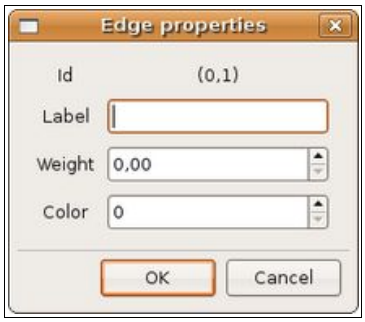
Edita as propriedades de uma aresta entre dois vértices ao pressionar com o botão esquerdo do mouse primeiramente em um vértice e em seguida em um segundo vértice. A figura 3.8 mostra o diálogo de edição de arestas. A figura 3.9 traz um *grafo* de exemplo onde foram utilizados atributos para vértices e arestas.

As propriedades disponíveis são:

1. *Id* - Par ordenado único que identifica a aresta (Não editável)
2. *Label* - Rótulo dado a aresta
3. *Weight* - Peso desta aresta
1. *Color* - Cor que ela será impressa na tela



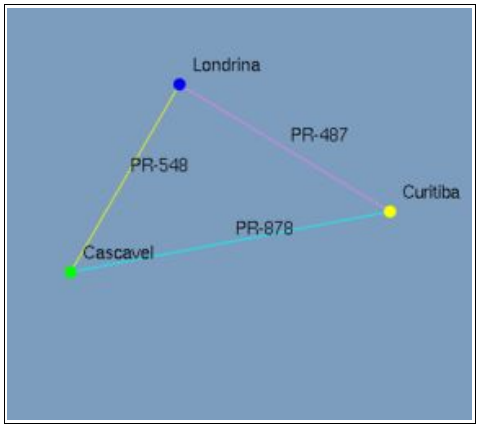
(figura 3.7)



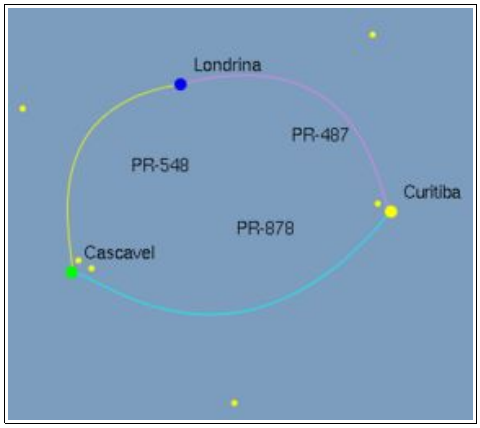
(figura 3.8)

- **Show control points**

Exibe os pontos de controle de uma aresta. Com eles é possível desenhar arestas curvas. Para isso após ativar os pontos de controle deve-se selecionar a opção *Move vertex* e em seguida pressionar o botão esquerdo do mouse arrastando os pontos de controle até formar a curva desejada. A figura 3.10 mostra um *grafo* com arestas curvas geradas utilizando os *control points*. É importante ressaltar que a curvatura das arestas não são salvas pelo grafo. Ao abrir um (.gr) o grafo exibirá sempre arestas retas.



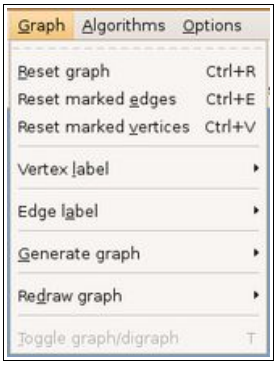
(figura 3.9)



(figura 3.10)

3.1.3 Graph

Esta opção de menu oferece manipulações sobre o *grafo* já construído através do menu *Edit*. Também traz um *wizard* para geração de novos *grafos* e opções para redesenhar *grafo* em edição com um novo layout. A figura 3.11 traz o menu de opções.



(figura 3.11)

3.1.3.1 Reset Graph - Ctrl+R

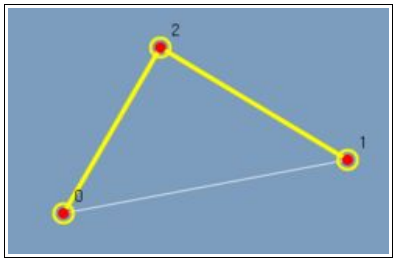
Limpa as estruturas internas e a área de trabalho preparando o ambiente do **GRAFO** para uma nova edição.

3.1.3.2 Reset marked edges - Ctrl+E

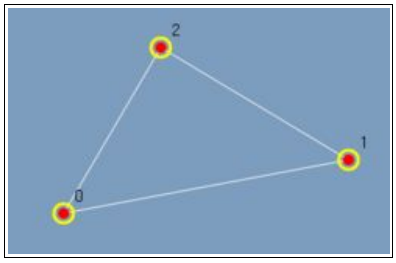
Limpa as arestas destacadas. Como exemplo temos a figura 3.13 quando executado o *Reset marked edges* sobre o grafo da figura 3.12.

3.1.3.3 Reset marked vertices - Ctrl+V

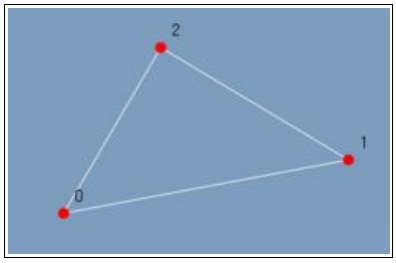
Limpa os vertices destacados. Como exemplo temos a figura 3.14 quando executado o *Resete marked vertices* sobre o grafo da figura 3.13.



(figura 3.12)



(figura 3.13)



(figura 3.14)

3.1.3.4 Vertex label

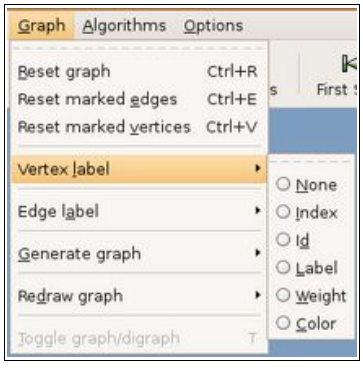
Exibe os vértices nomeados de acordo com a opção selecionada. Figura 3.15. Nas figuras 3.9 e 3.10 foram utilizadas a opção de exibição de *label* ou rótulos nos vértices. Estes vértices representam as cidades *Londrina*, *Curitiba*, *Cascavel*.

- *None* - Não exibe label para o vértice
- *Index* - Exibe o indice para o vértice
- *Id* - Exibe o identificador do vértice
- *Label* - Exibe o nome dado ao vértice
- *Weight* - Exibe o peso do vértice
- *Color* - Exibe o número da cor do vértice

3.1.3.5 Edge label

Exibe as arestas nomeadas de acordo com a opção selecionada. Figura 3.16. Nas figuras 3.9 e 3.10 foram utilizadas a opção de exibição de *label* ou rótulos nas arestas. Estas arestas representam nomes de rodovias, *PR-487*, *PR-548* e *PR-878*, que ligam as cidades.

- *None* - Não exibe nome para a aresta
- *Index* - Exibe o índice para a aresta
- *Id* - Exibe o identificador da aresta
- *Label* - Exibe o nome dado a aresta
- *Weight* - Exibe o peso da aresta
- *Color* - Exibe o número da cor da aresta

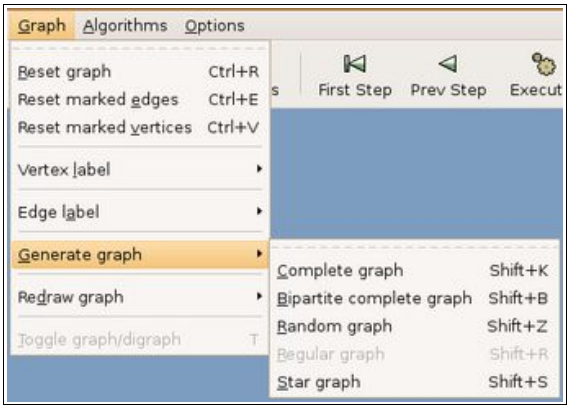


(figura 3.15)

(figura 3.16)

3.1.3.6 Generate Graph

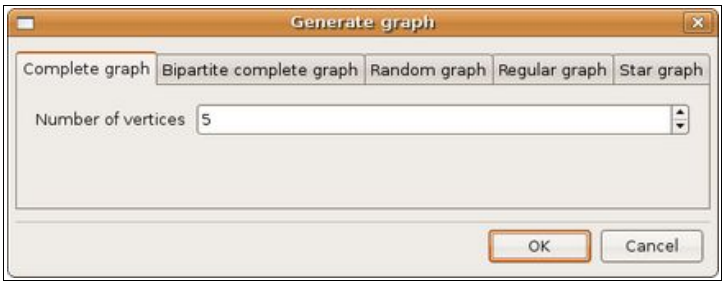
Gera *grafos* pré determinados de acordo com um conjunto de parâmetros informados. A figura 3.17 mostra o menu de opções.



(figura 3.17)

- **Complete Graph - Shift+K**

Gera um *grafo completo* de acordo com o número de vértices informados. A figura 3.18 mostra o dialogo de parâmetros. Como exemplo de execução temos a figura 3.23 que é um *grafo* completo de 5 vértices.



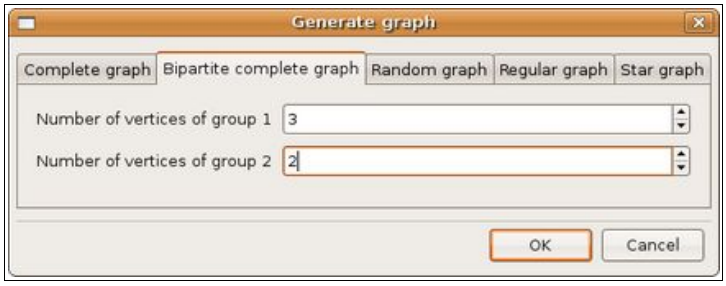
(figura 3.18)

- Parâmetros:
  1. *Number of vertices* - Número de vértices

- **Bipartite complete graph - Shift+B**

Gera um *grafo bipartite* de acordo com o número de vértices informados para o grupo 1 e grupo 2. A figura 3.19 mostra o dialogo de parâmetros. Como exemplo de execução temos a

figura 3.24 que é um *grafo bipartite* com 5 vértices no grupo 1 e 2 vértices no grupo 2.



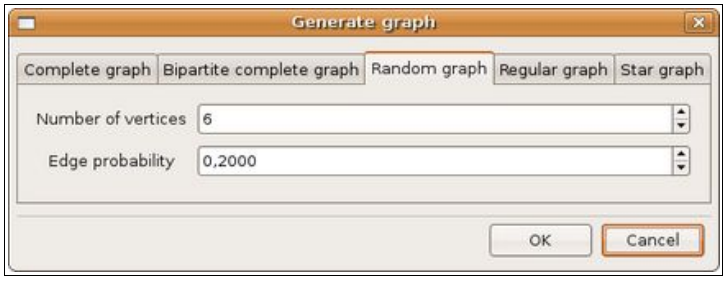
(figura 3.19)

- Parâmetros:

    1. *Number of vertices of group 1* - Número de vertices do grupo 1
    2. *Number of vertices of group 2* - Número de vertices do grupo 2

- **Random graph - Shift+Z**

Gera um *grafo* aleatório de acordo com o número de vértices informados e a probabilidade de existência de uma aresta entre eles. A figura 3.20 mostra o dialogo de parâmetros. Como exemplo de execução temos a figura 3.25 que é um *grafo* gerado de maneira aleatória com 6 vértices e uma probabilidade de existência de aresta de 20%.



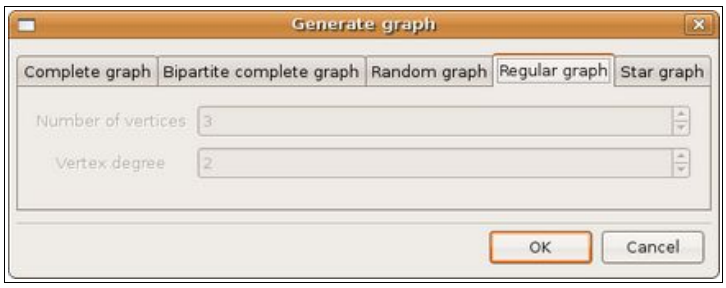
(figura 3.20)

- Parâmetros:

    1. *Number of vertices* - Número de vértices
    2. *Edge probability* - Probabilidade de uma aresta

- **Regular graph - Shift+R**

Gera um *grafo* regular de acordo com o número de vértices e grau informados. A figura 3.21 mostra o dialogo de parâmetros. A opção de *grafo* regular está presente no menu, porem a geração não foi implementada. No capítulo7 será abordado a possibilidade de conclusão desta opção. Como exemplo de *grafo* regular vide a figura 2.4 do capítulo 2.



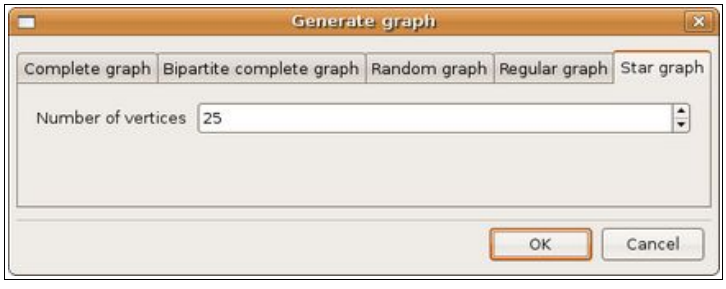
(figura 3.21)

- Parâmetros:

1. *Number of Vertices* - Número de vértices
2. *Vertex degree* - Grau do vértice

- **Star graph - Shift+S**

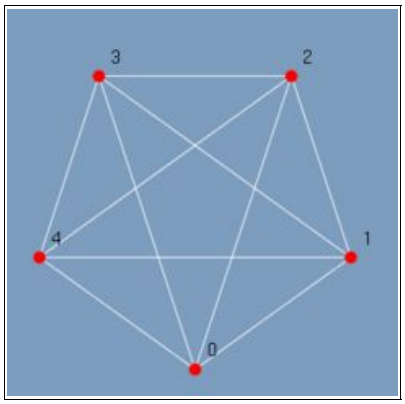
Gera um *grafo estrela* de acordo com o número de vértices informados. A figura 3.22 mostra o dialogo de parâmetros. Como exemplo de execução temos a figura 3.26 que é um *grafo estrela* com 25 vértices.



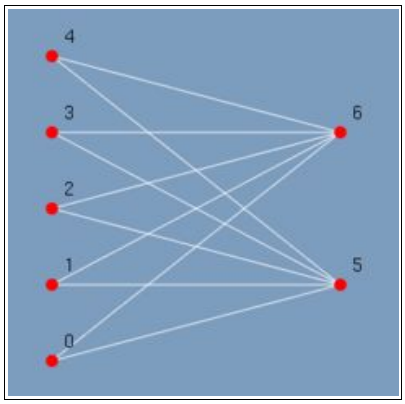
(figura 3.22)

- Parâmetros:

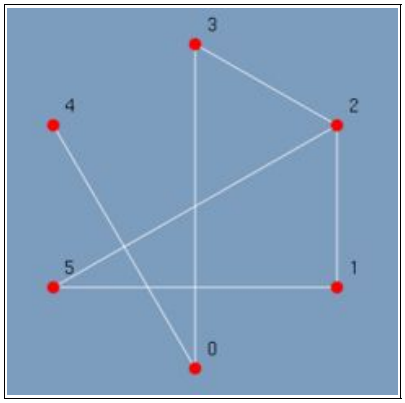
1. *Number of vertices* - Número de vértices



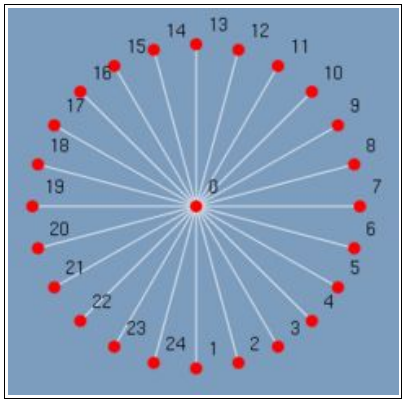
(figura 3.23)



(figura 3.24)



(figura 3.25)



(figura 3.26)

3.1.3.7 Redraw graph

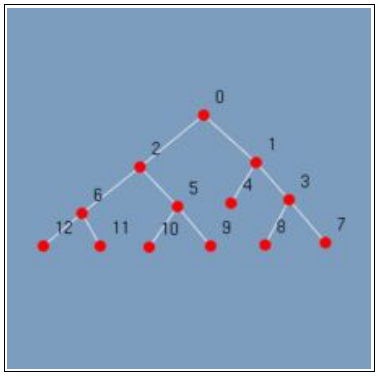
Redesenha o *grafo* em um novo *layout* porem mantendo inalterado o seu conjunto de vértices e arestas.

- **Fit to windows - Shift+W**

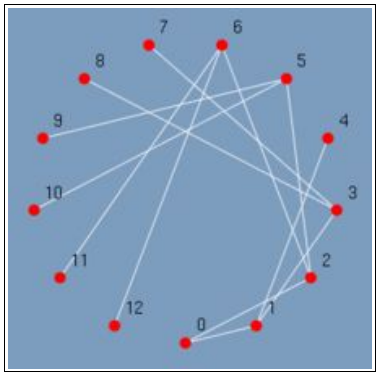
Redesenha o *grafo* fazendo que ele ocupe o espaço máximo disponível na área de trabalho.

- **As circle - Shift+C**

Redesenha o *grafo* em circulo. Com exemplo temos o *grafo* em árvore da figura 3.27 redesenhado em circulo na figura 3.28.



(figura 3.27)



(figura 3.28)

- **Using neato - Shift+N**

Redesenha o grafo utilizando o *neato*.

Vale observar que o *grafo* não está sendo corretamente redesenhado quando utilizado o *neato* e o *dot*. No capítulo 6 serão abordadas as possibilidades de manutenções nesta funcionalidade. Para informações sobre o *netao* e o *dot* visite o site (http://www.graphviz.org/).

- **Using dot - Shift+D**

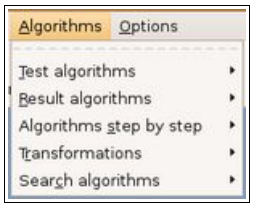
Redesenha o grafo utilizando o *dot*.

3.1.3.8 Toggle graph/digraph - T

Altera o tipo do *grafo* entre Grafo e Digrafo. Esta opção está desabilitada no menu por não estar implementada. No capítulo 6 será abordada a possibilidade de finalização ao suporte a digrafos.

3.1.4 Algorithms

Esta opção de menu, figura 3.29, exibe o elenco de algoritmos disponíveis para utilização no **GRAFO**. Os algoritmos estão distribuídos nas categorias de *Teste*, *Resultado*, *Passo a Passo*, *Transformação* e *Busca*. O **GRAFO** possui algoritmos nativos ou fornecidos através de plugins. Os algoritmos disponibilizados através de plugins são uma expansão da interface e estão destacados com ícone de plug ( ).



(figura 3.29)

- **Test algorithms**

Este menu é destinado a algoritmos que efetuam algum teste sobre o grafo da área de trabalho. Como por exemplo verificar se ele é *cordal* ou não, se ele é conexo ou se possui um *caminho de Euler*. Também neste menu estão dispostos dois algoritmos de testes da interface de plugins que são o *Hello World!* e o *Get some parameters*. A construção destes dois plugins serão detalhadas no capítulo 4.

- **Chordal**

Retorna à *barra de status* se o *grafo* é ou não cordal.

- **Hello World!** - ( )

Ilustra a utilização da *Barra de status* escrevendo "Hello World!".

- **Get some parameters** - ( )

Ilustra a utilização de entrada de parâmetros para plugins. Solicita que o usuário informe através de dialogos dados dos tipos *inteiro*, *double* e *string*. Em seguida imprime a entrada dada na *barra de status*.

- **Conectivity** - ()

Retorna à *barra de status* se o *grafo* é conexo.

- **Euler Graph** - ()

Retorna à *barra de status* se o *grafo* possui ou não um *caminho de Euler*.

- **Result algorithms**

Neste menu estão agrupados algoritmos que podem ser utilizados para análise do resultado de *grafos* gerados por outros algoritmos. Como por exemplo destacar a árvore de busca de *grafo* ou analisar quantos componentes conexos ele possuí. Neste menu também estão presentes o os plugins *Highest Degree* e *Some Edge* que terão as suas construções detalhadas no capítulo 4.

- **Span, Tree**

Executa o algoritmo *Span, Tree* no *grafo* destacando a sua árvore de busca.

- **Conected Components** - ()

Retorna quantos componentes conexos existem no *grafo*.

- **Highest Degree** - ()

Destaca o vértice de maior grau de um *grafo*.

- **Some Edge** - ()

Destaca a aresta entre o vértice de maior grau e o de menor grau adjacente a ele.

- **Algorithms step by step**

Agrupados neste menu estão os algoritmos com suporte a *passo a passo*. A utilização destes algoritmos serão detalhada no capítulo 5 bem como a construção do plugin *Select odd*.

- **Depth First**

Executa o algoritmo *Depth First* no modo passo a passo.

- **Select odd** - ()

Destaca todos os vértices ímpares de um *grafo* e todas as arestas que conectam dois vértices ímpares.

- **Tranformations**

Agrupados neste menu estão os algoritmos que realizam transformações físicas nos *grafos* da área de trabalho ou criam novos *grafos* com determinadas características. Como por exemplo transformar um *grafo* em *cordal* ou gerar um *grafo de Paley*. Também está presente o algoritmo *Generate Star Graph* que é utilizado no capitulo 4 para exemplificar a criação de vértices e arestas através de um plugin.

- **Line**

Executa o algoritmo *Line* em um *grafo*.

- **Complement**

Retorna o *grafo* de complemento.

- **Generate Star Graph** - ()

Cria um *grafo estrela* com o número de arestas solicitadas.

- **Generate Paley Graph** - ( )

Cria um *grafo de Paley* com o número de arestas solicitadas.

- **Chordalize a Graph** - ( )

Transforma um *grafo* não cordal em cordal.

- **Delaunary triangulation** - ( )

Executa o algoritmo de *triangulação Delaunary* em um *grafo*.

- **Search algorithms**

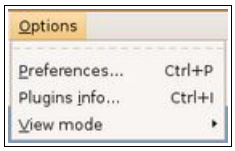
Este menu é destinado aos algoritmos de busca em grafos como por exemplo o algoritmo de busca em profundidade *Depth First*.

- **Depth First**

Executa o algoritmo *Depth First* em um *grafo* destacando a árvore e os vértices buscados.

3.1.5 Options

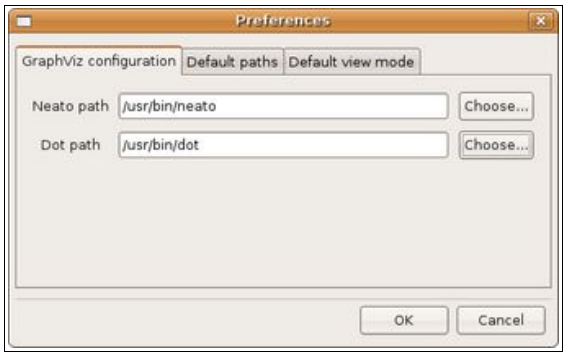
Este menu agrupa as preferências do usuário, configurações, informações sobre plugins e modo de visualização



(figura 3.30)

- **Preferences... - Ctrl+P**
  - **GraphViz Configuration**

Exibe as opções de configuração do *GraphViz* no **GRAFO**. Figura 3.31. Ao clicar em *Choose...* abre o dialogo de seleção de arquivo.



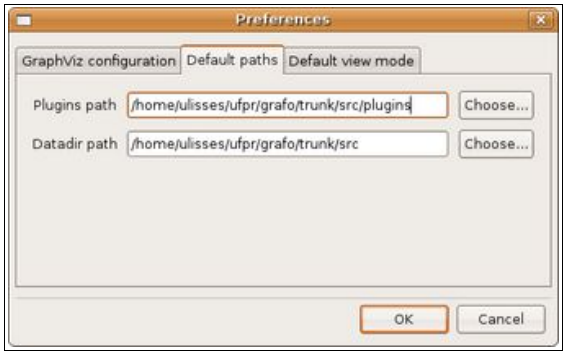
(figura 3.31)

- Opções:

    1. *Neato path* - Caminho completo para o programa neato
    2. *Dot path* - Caminho completo para o programa dot

- **Default paths**

Exibe as opções de configuração dos caminhos utilizados pelo **GRAFO**. Figura 3.32.



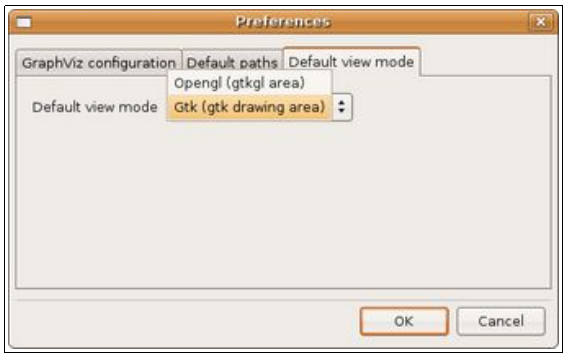
(figura 3.32)

- Opções:

    1. *Plugins path* - Caminho completo da localização dos plugins (*.so*)
    2. *Datadir path* - Caminho padrão para os diálogos *Open*, *Save* e *Save as*

- **Default view mode**

Padrão gráfico para a renderização da área de trabalho. Figura 3.33.



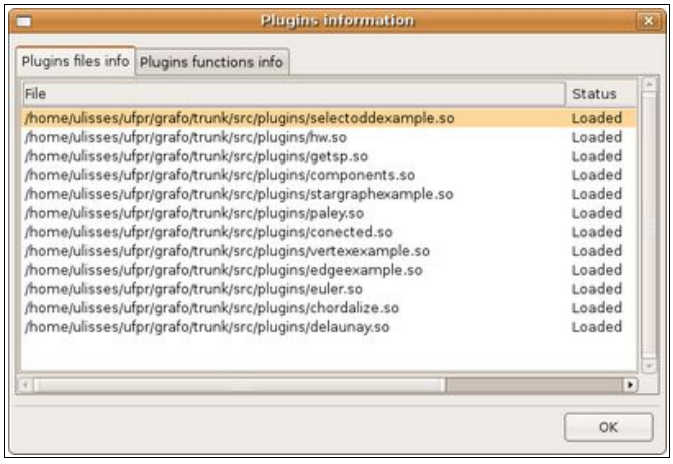
(figura 3.33)

- Opções:

1. *Opengl (gtkgl area)* - Utilizando OpenGL
2. *Gtk (gtk drawing area)* - Utilizando GTK

- **Plugins info... - Ctrl+I**

- **Plugins files info**

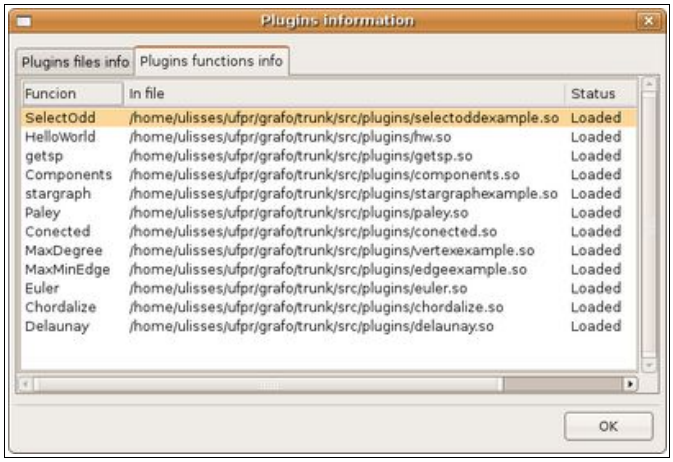


(figura 3.34)

- Colunas:

1. *File* - Informa o nome e caminho das bibliotecas dinâmicas dos plugins *(.so)*
2. *Status* - Informa se o plugin foi ou não carregado corretamente

- **Plugins functions info**



(figura 3.35)

- Colunas:

    1. *Function* - Informa as funções disponíves em cada plugin
    2. *In file* - Informa o nome e caminho das bibliotecas dinâmicas dos plugins *(.so)*
    3. *Status* - Informa se o plugin foi ou não carregado corretamente

- **View mode**

Altera o modo de renderização da área de trabalho entre *OpenGL* e *GTK*. Contudo ao entrar no **GRAFO** o modo configurado como padrão será restaurado.

- Opções:

    1. *Opengl* - *Shift*+*O* - Utilizando OpenGL
    2. *Gtk drawing area Shift*+*G* - Utilizando GTK

3.2 Barra de Ferramentas

A *barra de ferramentas*, figura 3.36, traz os comandos básicos de entrada e saída e opções para a manipulação dos algoritmos *passo a passo*. A utilização dos botões do *passo a passo* serão detalhadas no capitulo 5.

(figura 3.36)

- **New**

Atalho para o menu *New*.

- **Open**

Atalho para o menu *Open*.

- **Save**

Atalho para o menu *Save*.

- **Save As**

Atalho para o menu *Save As*.

- **First Step**

Retorna ao primeiro passo gravado.

- **Prev Step**

Retorna ao passo anterior.

- **Execute**

Inicia a execução passo a passo em modo automático.

- **Next Step**

Avança para o próximo passo.

- **Last Step**

Avança para o último passo gravado.

3.3 Área de trabalho

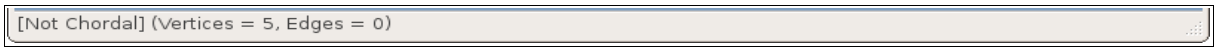
É na área de trabalho que o usuário edita os seus *grafos*, recebe o resultado de execução dos algoritmos como o destaque de vértices e arestas e a criação e exclusão de componentes do *grafo*.

3.4 Barra de Status

A *barra de status* serve como canal de comunicação entre o **GRAFO** e o usuário. É nela que o **GRAFO** dá instruções, mostra resultados e interage de forma escrita com o usuário. Como por exemplo a figura 3.37 onde o usuário é instruído a clicar na área de trabalho para criar um vértice. Na figura 3.38 o usuário é instruído a clicar em um vértice e arrasta-lo para que este seja movido. E na figura 3.39 o usuário é informado que o *grafo* por ele escolhido não é *cordal*.

(figura 3.37)

(figura 3.38)



(figura 3.39)

4 PLUGINS

Este capítulo é um tutorial prático para construção de plugins. Ele começa explorando os *headers* e estruturas internas ao **GRAFO** que precisão ser conhecidas antes da construção. Depois através de um exemplo simples, *Hello World*, é explorado a estrutura de um plugin: *headers* necessários, protótipo das funções e corpo das funções. Utilizando-se do *Hello World* também é abordo compilação, instalação e execução de um plugin. O capitulo termina detalhando através do exemplo *Get some parameter* como se faz para receber parâmetros. Pelo exemplo *Generate Star Graph* como se cria vértices e arestas. E finalmente utilizando-se dos plugins *Highest Degree* e *Some Edge* como se faz respectivamente para destacar os vértices e arestas na interface do grafo.

Os plugins são bibliotecas carregadas dinamicamente pelo **GRAFO** no momento da inicialização. Cada plugin pode ter um ou mais algoritmos. A cada algoritmo criado recebe uma nova entrada no menu *Algorithms* de acordo com a sua categoria. São elas:

1. Test
2. Result
3. Step by step
4. Tranformations
5. Search

Sem dúvida a construção de plugins é que torna o **GRAFO** realmente poderoso. Com eles pode-se facilmente implementar um novo algoritmo e acompanhar visualmente o seu resultado.

4.1 Headers

Para a construção de plugins é preciso conhecer algumas estruturas internas do **GRAFO** e os *headers* onde estão declaradas. São Elas:

graph.h

No *header graph.h* estão declaradas as estruturas que representam um vértice, uma aresta e um *grafo* dentro do ambiente. Bem como as funções utilizadas para manipula-los.

Vertex struct - Estrutura de um vértice

A estrutura *Vertex* (tabela 4.1) contém as informações necessárias para a representação de um vértice são elas: Identificador, nome dado ao vértice, o seu peso, a sua posição (x,y) no plano cartesiano, sua cor, informações se o vértice está conectado, se está marcado, grau do vértice, grau de entrada e grau de saída.

Obs.: A propriedade *degree* só é utilizada em *grafos* enquanto que propriedades *indegree* e *outdegree* em *Digrafos*.

Nome	Tipo	Descrição
id	int	Identificador auto incremental e único
label	char *	Texto (nome) associado ao vértice
weight	Weight	Peso da aresta - typedef float
x	float	Posição X em 2D
y	float	Posição Y em 2D
color	Color	Cor do vértice - typedef int
concomp	int	Se 1 o vértice está conectado
mark	char	Se 1 o vértice está marcado
degree	int	Grau do vértice
indegree	int	Grau de entrada
outdegree	int	Grau de saída

(tabela 4.1)

Edge - Matriz de adjacência

Um aresta é representada pela matriz de adjacência e sua unidade básica é `typedef char Edge`. Neste caso assume *0* (zero) ou *1* (um). Na tabela 4.3 onde descrito as propriedades de um grafo pode ser visto que o *Edge* é declarado como `Edge **`, ou seja uma matriz bidimensional.

Edge_prop struct - Estrutura das propriedades de uma aresta

A estrutura *Edge_prop* (tabela 4.2) vem complementar a matriz de adjacência *Edge*. As informações complementares são: Nome associado a aresta, o seu peso, a sua cor, informações se a aresta está marcada e os pontos de controle utilizados para gerar arestas curvas.

Nome	Tipo	Descrição
label	char *	Texto (nome) associado a aresta
weight	Weight	Peso da aresta - typedef float
color	Color	Cor da aresta - typedef int
mark	char	Se 1 a aresta está marcada
ctrlpoints	float [4][3]	Pontos de controle

(tabela 4.2)

Graph struct - Estrutura de um grafo

A estrutura *Graph* (tabela 4.3) contém as informações necessárias para a representação de um *grafo*. São elas: O seu tipo, a quantidade de vértices, o identificador do próximo vértice, a quantidade

de vértices para o tipo *grafo*, a quantidade de arestas para o tipo *digrafo*, o espaço em memória alocado, a lista de vértices, a matriz de adjacência, a matriz de propriedade dos vértices, o número de componentes conexos e o número de cores utilizadas.

Obs.: As propriedades ncomp, ncolor e encolor não são inicializadas ou atualizadas automaticamente. Caso deseje utiliza-las é necessário que as faça manualmente.

Nome	Tipo	Descrição
type	GraphType	Informa o tipo do *grafo* - typedef enum (GRAPH,DIGRAPH)
size	int	Quantidade de vértices do *grafo*
nextid	int	Identificador a ser recebido pelo próximo vértice
nedges	int	Quantidade de vértices (*Grafos*)
narcs	int	Quantidade de arcos (*Digrafos*)
allocated_size	int	Espaço alocado
vertex	Vertex *	Lista de vértices
edge	Edge **	Matriz de adjacência - typedef char
prop	Edge_prop **	Propriedades da matriz de adjacência
ncomp	int	Número de componentes conexos
ncolor	int	Número de cores utilizadas (Não é número cromático)
encolor	int	Número de cores utilizadas pelas arestas

(tabela 4.3)

SEdge struct - Estrutura de uma aresta simples

A estrutura *SEdge* (tabela 4.4) contem a informação do par ordenado que forma uma aresta. Esta estrutura será e incluída será utilizada na lista de destaque de aresta. Vide tabela 4.18 para maiores detalhes.

Obs.: As estruturas SEdge e SVertex são auxiliares e são necessárias para destaque de vértices e arestas na interface do **GRAFO**.

Nome	Tipo	Descrição
u	SVertex	Vértice U - typedef int
v	SVertex	Vértice V - typedef int

(tabela 4.4)

Functions - Funções para manipulação de um *grafo*

A tabela 4.5 traz a lista de funções para manipulação do grafo. Com elas pode-se alocar e liberar memória para um *grafo*, inicializar as estruturas internas, copiar um grafo em outro, adicionar e remover vértices, arestas e arcos. E também limpar o status de marcado (*mark*) de vértices e arestas.

Obs.: Funções internas de controle não estão documentadas. A princípio elas não são necessárias para a construção de plugins.

Declaração	Descrição
`void AllocGraph(Graph *G)`	Aloca espaço para um *grafo* e as estruturas que o compõem (vértices, arestas, matriz de adjacência, etc)
`void FreeGraph(Graph *G)`	Libera espaço alocado para *grafo* e suas estruturas
`void InitGraph(Graph *G, GraphType type)`	Inicializa um *grafo* e suas estruturas
`void InitEdges(Graph *G)`	Inicializa somente a matriz de adjacência e suas propriedades
`void CopyGraph(Graph *H, Graph *G)`	Aloca o *grafo* H e copia o G nele
`void AddPoint(Graph *G, float x, float y)`	Adiciona a vértice na posição (x,y)
`void RemovePoint(Graph *G, int p)`	Remove o vértice o P vértice do *grafo*
`void AddEdge(Graph *G, int x, int y)`	Adiciona um aresta entre os vértices X e Y
`void RemoveEdge(Graph *G, int x, int y)`	Remove a aresta entre os vértices X e Y
`void AddArc(Graph *G, int x, int y)`	Adciona o arco entre os vértice X para Y
`void RemoveArc(Graph *G, int x, int y)`	Remove a aresta direcionada do vértice X para Y
`void ClearMarkedVerts(Graph *G)`	Limpa a propriedade *mark* dos vértices
`void ClearMarkedEdges(Graph *G)`	Limpa a propriedade *mark* das arestas

(tabela 4.5)

pgin.h

No *header pgin.h* estão declarados a estrutura para a lista de algoritmos bem como funções para entrada de parâmetros.

Pgin struct - Estrutura da lista de funções

A estrutura *Pgin* é uma lista de funções (algoritmos) disponíveis em um plugin. Cada plugin deve possuir no mínimo uma função sempre terminada pelo tipo `PGIN_LIST_END` ao final do capítulo 4 a criação de

plugins de exemplo irão abordar a correta construção da lista de algoritmos disponíveis em um plugin. A tabela 4.6 detalha os campos da estrutura. São eles: Tipo do algoritmo, nome para o menu, nome da função principal do algoritmo e flags de controle para o **GRAFO**.

Nome	Tipo	Descrição
`type`	`int`	Determina o tipo do plugin na lista de algoritmos
`label`	`char *`	Determina o nome do algoritmo. Este é nome que irá aparecer na entrada do menu *Algorithms*
`name`	`char *`	Determina o nome da função do algoritmo dentro do plugin
`flags`	`int`	Passa informações ao **GRAFO** de como tratar esse algoritmo após a sua execução

(tabela 4.6)

A tabela 4.7 detalha o valor para o campo `type` na lista de algoritimos.

Nome	Descrição
`PGIN_TEST_ALG`	Informa que o algoritmo é de testes
`PGIN_RESULT_ALG`	Informa que o algoritmo é de resultado
`PGIN_STEP_ALG`	Informa que o algoritmo é de passo a passo
`PGIN_TRANSF_ALG`	Informa que o algoritmo é de transformação
`PGIN_SEARCH_ALG`	Informa que o algoritmo é de busca
`PGIN_LIST_END`	Final da lista de algoritmos

(tabela 4.7)

O campo `flag` detalhado na tabela 4.8 informa ao **GRAFO** como ele deve redesenhar o *grafo* resultante ao termino da execução de um algoritmo de transformação. Os demais tipos de *plugins* não são sensíveis a estes *flags* e os grafos não serão redesenhados ao término da execução destes *plugin*.

Nome	Descrição
`PGIN_FLAG_NOREDRAW`	Não redesenha o *grafo*. O construtor do plugin é responsável por determinar a posição dos vértices
`PGIN_FLAG_REDRAWASCIRCLE`	Redesenha o *grafo* automaticamente em circulo
`PGIN_FLAG_REDRAWWITHCENTEREDVERTEX`	Redesenha o *grafo* com o primeiro vértice no centro

(tabela 4.8)

Functions - Funções de entrada de dados a um *plugin*

A tabela 4.9 traz a lista de funções para manipulação de entrada de dados para os plugins. Com elas é possível fazer a leitura dos tipos de dados: Inteiro, Double e String.

Declaração	Descrição
`int pgin_read_int(char *name, char *label, int def, int min, int max, int inc)`	Cria janela para leitura de um tipo de dado inteiro
`double pgin_read_double(char *name, char *label, double def, double min, double max, double inc, int digits)`	Cria janela para leitura de um tipo de dado double
`char *pgin_read_string(char *name, char *label, char *def, int max_len)`	Cria janela para leitura de uma tipo de dado string

(tabela 4.9)

A tabela 4.10 detalha os parâmetros das funções de entrada de dados descritas na tabela 4.9.

Parâmetros	Descrição
`name`	Nome da janela
`label`	Texto informativo
`def`	Valor padrão
`min`	Mínimo valor permitido
`max`	Máximo valor permitido
`inc`	Incremento para `int` e `double`
`digits`	Número de casas decimais
`max_len`	Tamanho máximo da string

(tabela 4.10)

alghandler.h

No *header alghandler.h* estão declarados as estruturas das listas de algoritmos bem como as funções utilizadas para manipula-las.

A tabela 4.11 detalha a função *RunAlgorithm* que pode ser utilizada para chamar qualquer algoritmo existente no grafo sendo ele nativo ou não. As listas especiais de destaque *arc_edges* e *verts* serão abordadas neste capítulo no tópico *Estrutura de um plugin*.

Declaração	Descrição
`int RunAlgorithm(char *name, Graph *G, char *s,int x, List *arc_edges, List *verts)`	Chama qualquer algoritmo disponível sendo ele nativo ou por plugins

(tabela 4.11)

list.h

No *header list.h* estão declarados as estruturas de pilha e lista. Bem como as funções utilizadas para manipula-las. Elas são necessárias no destaque de vértices e arestas. Vide o tópico *Destacando vértices e arestas* neste capítulo.

Node struct - Estrutura de um nodo

A estrutura *Node* na tabela 4.12 contém um nodo genérico onde o campo `data` pode apontar para qualquer tipo de dado. O nodo pode ser utilizado tanto em uma lista ou em uma pilha.

Nome	Tipo	Descrição
`data`	`void *`	Aponta para o dado do Node
`size`	`int`	Tamanho alocado para o dado apontado
`next`	`Node *`	Aponta para o próximo nodo
`prev`	`Node *`	Aponta para o nodo anterior

(tabela 4.12)

Functions - Funções para manipulação de um nodo

A tabela 4.13 traz a lista de funções para criação e destruição de um nodo.

Declaração	Descrição
`Node MakeNode(void *data, int size)`	Cria um nodo apontando para um dado informado com o tamanho de `size`
`void DestroyNode(Node n)`	Destrói o nodo informado

(tabela 4.13)

List struct - Estrutura de uma lista

A estrutura *List* (tabela 4.14) contém os dados necessários para se trabalhar com o tipo de dado lista.

Nome	Tipo	Descrição
`size`	`int`	Número de nodes
`head`	`Node`	Primeiro node
`tail`	`Node`	Último node

(tabela 4.14)

Functions - Funções para manipulação de uma lista

A tabela 4.15 contém a lista de funções para manipulação da estrutura *list* podendo-se com elas: Criar e inicializar uma lista, inserir e remover um nodo, mover, copiar e limpar uma lista.

Declaração	Descrição
`List *CreateList(void)`	Cria uma lista
`void InitList(List *list)`	Inicializa a lista informada
`void InsertNode(List *list, Node node)`	Insere um nodo na lista informada
`Node RemoveNode(List *list, Node node)`	Remove o nodo na lista informada
`void MoveList(List *l1, List *l2)`	Copia a L2 em L1 destruindo L2; L1 deve estar inicializada
`void CleanList(List *list)`	Remove todos os nodos de uma lista
`void CopyList(List *l1, List *l2)`	Copia a L2 em L1; L1 deve estar inicializada

(tabela 4.15)

Stack struct - Estrutura de uma pilha

A estrutura *Stack* (tabela 4.16) contém os dados necessários para se trabalhar com o tipo de dado pilha no **GRAFO**.

Nome	Tipo	Descrição
`size`	`int`	Número de nodes
`top`	`Node`	Topo da pilha

(tabela 4.16)

Functions - Funções para manipulação de uma pilha

A tabela 4.17 contém a lista de funções para manipulação da estrutura *Stack* podendo-se com elas: Criar e inicializar uma pilha, inserir e remover um nodo e limpar a pilha.

Declaração	Descrição
`Stack *CreateStack(void)`	Cria uma pilha
`void InitStack(Stack *stk)`	Inicializa a pilha informada
`void PushNode(Stack *stk, Node node)`	Insere o nodo na pilha
`Node PopNode(Stack *stk)`	Remove o nodo da pilha
`void CleanStack(Stack *stk)`	Remove todos os nodos da pilha

(tabela 4.17)

4.2 Estrutura de um plugin

Neste tópico será abortado a criação do plugin *Hello World* e também o detalhamento da estrutura e divisões de um plugin. O código fonte completo do plugin *Hello Word* encontra-se no Apêndice B.

O plugin é dividido em 3 partes básicas. São elas:

1. Inclusão de *headers*
2. Protótipo das funções
3. Corpo das funções

4.2.1 headers

Na primeira parde deve-se incluir os *headers* da linguagem C e os *headers* obrigatórios para os plugins:

```
/* headers da linguagem C */
#include <stdio.h>
#include <stdlib.h>

/* headers do GRAFO  */
/* Informacoes da estrutura de plugins */
#include "../pgin.h"

/* Informacoes da estrutura do GRAFO */
#include "../graph.h"
```

Obs.: Os *headers* do grafo foram incluídos pressupondo que a diretório *plugins* está um nível abaixo do diretório do **GRAFO**.

4.2.2 Protótipo das funções

Na segunda parte deve-se declarar o protótipo para a função obrigatória *pgin_info* o protótipo para as funções de algoritmos e outras funções internas ao plugin. Como função de algoritmo foi declarado o tradicional *HelloWorld*.

```
/* Funcao obrigatoria pgin_info */
Pgin *pgin_info(void);

/* Funcoes de algoritmos */
int HelloWorld(Graph *G, char *mess);

/* Funcoes internas ao plugin*/
...
...
```

As funções de algoritmos possuem uma assinatura diferente para a lista de parâmetros de acordo com o seu tipo. Essa diferença deve ser observada ao incluir o algoritmo na lista de funções do *plugin*.

1. Test

```
int TestAlgorithm(Graph *G, char *mess)
```

2. Result

```
int ResultAlgorithm(Graph *G, char *mess, List *arc_edges, List *verts)
```

3. Step by step

```
int StepAlgorithm(Graph *G, char *mess, int indx, List *arc_edges, List *verts)
```

4. Tranformations

```
int Transformation(Graph *G, char *mess)
```

5. Search

```
int SearchAlgorithm(Graph *G, char *mess, int indx, List *arc_edges, List *verts)
```

A tabela 4.18 traz o detalhamento dos parâmetros passados as funções de algoritmos.

Parâmetros	Descrição
`Graph *G`	Aponta para *grafo* na área de trabalho
`char *mess`	Aponta para a mensagem na barra de status
`int indx`	Número do vértice inicial de buscas
`List *arc_edges`	Lista com as arestas destacadas do *grafo* na área de trabalho
`List *verts`	Lista com os vértices destacados do *grafo* na área de trabalho

(tabela 4.18)

É importante destacar que o *grafo G* passado como parâmetro aponta para a estrutura do grafo que será desenhada na área de trabalho. E as listas *arc_edges* e *verts* apontam para as estruturas de destaque de vértices e arestas. Ao iniciar o **GRAFO** estas estruturas são alocadas e disponibilizadas para utilização. Um plugin pode reinicializa-las com *InitGraph*, *InitEdges* e *CleanList*. Porem **NÃO** deve liberar a memória alocadas a elas.

4.2.3 Corpo das funções

Na terceira parte deve-se criar o corpo para a função obrigatória *pgin_info* e implementar as funções do algoritmo.

pgin_info - Função obrigatória

A *pgin_info* é responsável por inicializar a lista de funções existentes no *plugin*. No exemplo abaixo incluímos a função *HelloWorld* do tipo *PGIN_TEST_ALG*.

```
/* A funcao pgin_info() deve sempre estar presente. O GRAFO ira chama-la
 * para recuperar as informacoes do plugin.
 */
Pgin *pgin_info(void){
Pgin *pgin;

    /* Aloca espaco para o HelloWorld e o PGIN_LIST_END */
    pgin = (Pgin *) malloc(2 * sizeof(Pgin));

    /* Inclui HelloWorld como primeiro algoritmo do plugin */
    pgin[0].type = PGIN_TEST_ALG;
    pgin[0].label = "Hello World!";
    pgin[0].name = "HelloWorld";
    pgin[0].flags = 0;
```

```
    /* Indica o final da lista de algoritmos */
    pgin[1].type = PGIN_LIST_END;
    pgin[1].label = 0;
    pgin[1].name = 0;
    pgin[1].flags = 0;

    return pgin;
}
```

Como já foi dito um plugin pode conter vários algoritmos, para isso basta incluí-los na lista. Por exemplo se o algoritmo *HelloUniverse* estivesse no mesmo plugin que o *HelloWorld* a função *pgin_info* deveria ser:

```
/* A funcao pgin_info() deve sempre estar presente. O GRAFO ira chama-la
 * para recuperar as informacoes do plugin.
 */
Pgin *pgin_info(void){
Pgin *pgin;

   /* Aloca espaco para o HelloWorld, HelloUniverse e o PGIN_LIST_END */
   /* Observe que agora o malloc aloca um espaco a mais para o HelloUniverse */
   pgin = (Pgin *) malloc(3 * sizeof(Pgin));

   /* Inclui HelloWorld como primeiro algoritimo do plugin */
   pgin[0].type = PGIN_TEST_ALG;
   pgin[0].label = "Hello World!";
   pgin[0].name = "HelloWorld";
   pgin[0].flags = 0;

   /* Inclui HelloUniverse como primeiro algoritmo do plugin */
   pgin[1].type = PGIN_TEST_ALG;
   pgin[1].label = "Hello Universe!!!";
   pgin[1].name = "HelloUniverse";
   pgin[1].flags = 0;

   /* Indica o final da lista de algoritmos */
   pgin[2].type = PGIN_LIST_END;
   pgin[2].label = 0;
   pgin[2].name = 0;
   pgin[2].flags = 0;

   return pgin;
}
```

HelloWorld - Funções de algoritmos

E finalmente a implementação da função de algoritmo. Quando o nosso exemplo for executado será escrito na barra de status *Hello World!*.

```
/* Esta e a funcao principal do plugin */
int HelloWorld(Graph *G, char *mess){

    sprintf(mess, "Hello World!");

    return 1;
}
```

4.3 Compilação e Instalação

Para que o plugin seja reconhecido pelo **GRAFO** devemos criar uma biblioteca dinâmica e inclui-lo no diretório destinado aos plugins.

1. Compilação

Para compilar o plugin de exemplo *HelloWorld* que está no arquivo *hw.c* utilizamos a linha de comando abaixo:

```
gcc -Wall -fPIC -shared hw.c -o hw.so
```

2. Instalação

Para instalar o plugin basta copiar a biblioteca construída (*hw.so*) para o diretório de plugins. O diretório configurado pode ser consultado/alterado em *Options* > *Preferences...* > *Default paths* > *Plugins path*. Vide o capítulo 3 para maiores detalhes.

4.4 Execução

Ao abrir o **GRAFO** os plugins serão carregados e um item de menu criado para cada função neles contidas. No nosso exemplo o plugin *HelloWorld* criou a entrada de menu abaixo. Figura 4.1:

(figura 4.1)

Ao clicar no menu *Hello World!* o **GRAFO** chama a função de algoritmo associado a ele. E ao termino atualiza a *área de trabalho* e a *barra de status* com os resultados. No nosso exemplo simplesmente escreve na *barra de status*. Figura 4.2.

(figura 4.2)

4.5 Recebendo parâmetros

Neste tópico será abordado a criação do plugin *Get some parameters* que ilustra a utilização das funções de entrada de dados para o **GRAFO**. O código fonte completo do plugin *Get some parameters* encontra-se no Apêndice B.

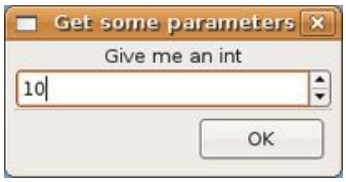
Os plugins podem receber 3 tipos de dados.

1. Inteiro
2. Double
3. String

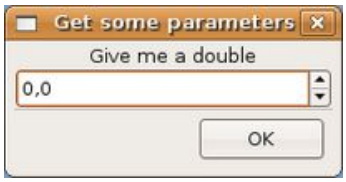
Abaixo segue a função principal no arquivo *getsp.c*. O detalhamento das funções *pgin_read_int*, *pgin_read_double* e *pgin_read_string* pode ser visto na tabela 4.9.

```
int getsp(Graph *G, char *mess){
int i;
double d;
char *s;

    i = pgin_read_int("Get some parameters", "Give me an int",
                10, 1, 2000000, 1);
    d = pgin_read_double("Get some parameters", "Give me a double",
                0.0, -10.0, 10.0, 0.1, 1);
    s = pgin_read_string("Get some parameters", "Give me a string",
                "I'm a string", 40);

    sprintf(mess, "I got: %d, %f and '%s'. Thanks!", i, d, s);
    free(s);

    return 1;
}
```

Como resultado obtemos os seguintes diálogos de entrada de dados. Para um inteiro figura 4.3. Para um double figura 4.4. Para uma string figura 4.5.



(figura 4.2)



(figura 4.3)

(figura 4.4)

E os valores recebidos pelos diálogos são escritos na *barra de status*. Vide figura 4.6.

(figura 4.6)

4.6 Criando vértices e arestas

Neste tópico será abordado a criação do plugin *Generate Star Graph* que ilustra a utilização das funções *AddPoint* e *AddEdge*

detalhadas na tabela 4.5. Abaixo segue a função principal no arquivo *stargraphexample.c*. O código fonte completo do plugin *Generate Star Graph* encontra-se no Apêndice B.

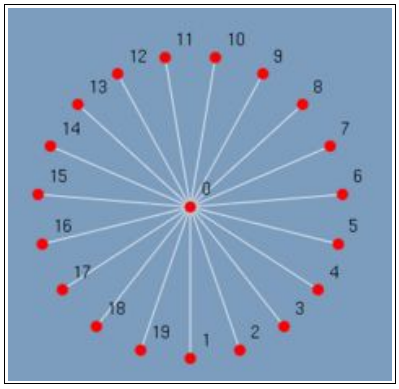
```
int stargraph(Graph *G, char *mess){
int i, n;

    /* Le o numero de vertices do grafo */
    n = pgin_read_int("Generate Star Graph", "Number of vertices",
                5, 1, 2000000, 1);

    /* Inicializar o grafo da area de trabalho */
    InitGraph(G, G->type);

    /* Adiciona N vertices na posicao (0,0) */
    for (i = 0; i < n; i++)
        AddPoint(G, 0, 0);

    /* Adiciona N arestas comecando no vertice 0 ate N-1 */
    for (i = 1; i < n; i++) {
        AddEdge(G, 0, i);
    }

    return 1;
}
```

Como pode ser observado o algoritmo não se preocupa em posicionar de maneira elegante os vértices na tela. Adicionando todos os vértices sempre na posição *(0,0)*.

```
    ...

    /* Adiciona N vertices na posicao (0,0) */
    for (i = 0; i < n; i++)
        AddPoint(G, 0, 0);

    ...
```

Isto é possível porque o plugin de transformação criado está utilizando o *flag PGIN_FLAG_REDRAWWITHCENTEREDVERTEX*. E com isso o *grafo* gerado será redesenhado automaticamente com o primeiro vértice no centro. Se estivéssemos utilizando o *flag PGIN_FLAG_NOREDRAW* o plugin deve responsável por posicionar os vértices corretamente na *área de trabalho*.

Como resultado a geração de um grafo estrela com 20 vértices. Figura 4.7:



(figura 4.7)

4.7 Destacando vértices e arestas

Neste tópico será abordado a construção do plugin *Highest Degree* que ilustra a utilização da lista especial *verts* e a construção do plugin *Some Edge* que ilustra a utilização da lista especial *arc_edges*. O destaque de vértices e arestas utilizam as listas especiais passadas como parâmetros aos plugins. A lista *verts* contém os vértices destacados em um grafo. A lista *arc_edges* contém as arestas e arcos destacados em um *{di,}grafo*. Para o destaque deve-se simplesmente inserir um nodo nas listas a ao encerrar a execução do algoritmo o **GRAFO** se encarrega de destacar na interface os nodos inseridos. O detalhamento sobre as funções para manipulação de nodos e listas estão respectivamente nas tabelas 4.13 e 4.15.

4.7.1 Destacando um vértice

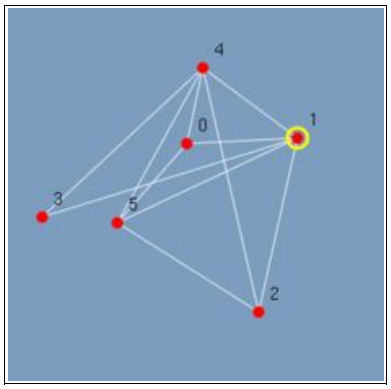
Abaixo segue a função principal no arquivo *vertexexample.c* responsável pelo plugin *Highest Degree* que ilustra a utilização da lista

especial *verts*. O código fonte completo do plugin *Highest Degree* encontra-se no Apêndice B.

```
int MaxDegree(Graph *G, char *mess, List *arc_edges, List *verts){
Node N;
int i,maxdeg;

    /* Testa se o grafo esta vazio */
    if (G->size < 1) {
        sprintf(mess, "%s", "Empty graph!");
        return 1;
    }

    /* Procurar pelo vertice de maior grau */
    maxdeg = 0;
    for (i = 1; i < G->size; i++) {
        if (G->vertex[i].degree > G->vertex[maxdeg].degree) {
            maxdeg = i;
        }
    }

    /* Coloca em um novo nodo o vertice de maior grau */
    N = MakeNode(&maxdeg, sizeof(int));

    /* Coloca o nodo na lista de vertices especiais */
    InsertNode(verts, N);

    return 1;
}
```

Utiliza-se da função *MakeNode* para criar um novo nodo e depois da função *InsertNode* para inserir um nodo com o vértice de maior grau na lista *verts*.

Como resultado o destaque do vértice 1 de grau 5 no grafo abaixo. Figura 4.8:



(figura 4.8)

4.7.2 Destacando uma aresta

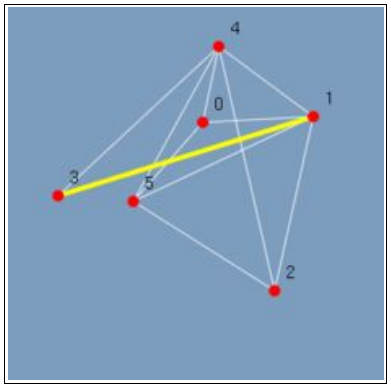
Abaixo segue a função principal no arquivo *edgeexample.c* responsável pelo plugin *Some Edge* que ilustra a utilização da lista especial *arc_edges*. O código fonte completo do plugin *Some Edge* encontra-se no Apêndice B.

```
int MaxMinEdge(Graph *G, char *mess, List *arc_edges, List *verts){
SEdge E; /* Aresta simples */
Node N;
int i,maxdeg,mindeg;

    /* Testa se o grafo esta vazio */
    if (G->size < 1) {
        sprintf(mess, "%s", "Empty graph!");
        return 1;
    }

    /* Procurar pelo vertice de maior grau */
    maxdeg = 0;
    for (i = 1; i < G->size; i++) {
        if (G->vertex[i].degree > G->vertex[maxdeg].degree) {
            maxdeg = i;
        }
    }

    /* Procura pelo vertice de menor grau adjacente ao de maior */
    mindeg = -1;
    for (i = 0; i < G->size; i++) {
        if (G->edge[maxdeg][i]) {
            if (mindeg == -1) {
                mindeg = i;
            } else if (G->vertex[i].degree < G->vertex[mindeg].degree) {
                mindeg = i;
```

```
                    }
                }
            }

            /* Atribui maxdeg e o mindeg a aresta simples */
            E.u = maxdeg;
            E.v = mindeg;

            /* Coloca em um novo nodo a aresta simples  */
            N = MakeNode(&E, sizeof(SEdge));

            /* Coloca o nodo na lista de arestas especiais */
            InsertNode(arc_edges, N);

            return 1;
        }
```

Como no exemplo do vértice também se utiliza a função *MakeNode* para criar um novo nodo. Porem o nodo criado contém como dado uma aresta simples (*SEdge*). E depois utiliza-se da função *InsertNode* para inserir o nodo com a aresta encontrada na lista *arc_edges*.

Como resultado o destaque da aresta entre o vértice 1 de grau 5 e o vértice 3 de grau 2. Figura 4.9:



(figura 4.9)

5 PASSO A PASSO

Neste capítulo, inicialmente, é feito um comparativo entre a execução de um algoritmo normal; recurso já existente no **GRAFO**; e a execução de um algoritmo *passo a passo*, novo recurso introduzido por este trabalho. também são explicados os recursos existentes no *passo a passo* e como o usuário pode interagir com um algoritmo. Em seguida é abordado como o este foi implementado e o que motivou esta abordagem. O capítulo termina mostrando um tutorial prático de como se realiza a implementação um plugin com funcionalidade *passo a passo*. Para tanto abordou-se neste tema também os *headers* necessários, as estruturas internas e os cuidados que devem ser tomados para sua construção.

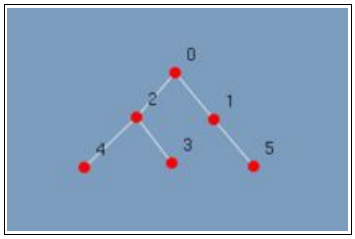
5.1 Como estava

O que é o *passo a passo*? É a possibilidade de interagir com a execução do algoritmo. E com isso conseguir uma visão melhor de como e o que ele faz do seu início ao fim.

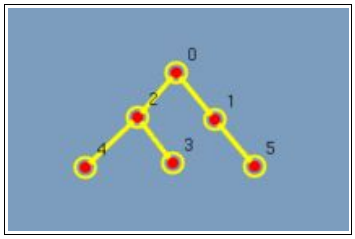
A possibilidade de execução de algoritmos *passo a passo* veio preencher uma lacuna deixada pelo **GRAFO**. Na estrutura de execução que existia até então era difícil observar o que se passava entre o início (click no menu) e o fim do algoritmo (atualização da área de trabalho).

Observemos a execução do algoritmo *Depth First - Busca em profundidade*, sem suporte a *passo a passo*, no menu *Algorithms > Search algorithms > Depth First*.

Na figura (5.1) observa-se um *grafo* em árvore com 6 vértices. Ao executar o algoritmo em segundos todos os vértices e arestas da árvore são destacados. Vide figura (5.2). Como a árvore foi completamente destacada? Qual a ordem que os vértices foram destacados? Quantos passos têm esse algoritmo? Sem o conhecimento prévio de como funciona a busca em profundidade em uma árvore é impossível responder a tais perguntas.



(figura 5.1)



(figura 5.2)

5.2 Como ficou

Foi justamente com o objetivo de responder a estas perguntas que entra a execução de algoritmo *passo a passo* na interface do **GRAFO**. Ele vem auxiliar a compreensão e o entendimento da execução de algoritmos de uma maneira visual e interativa.

Observemos a execução do mesmo algoritmo *Depth First - Busca em profundidade*, porém com suporte a *passo a passo*, no menu *Algorithms* > *Algorithms step by step* > *Depth First*.

Na figura 5.3 observa-se o mesmo *grafo* em árvore com 6 vértices. Ao executar o algoritmo o *grafo* vai lentamente sendo destacado. Vide figura 5.4 até figura 5.9. Com esta visualização mesmo um leitor leigo consegue responder as perguntas feitas.

Como a árvore foi completamente destacada?

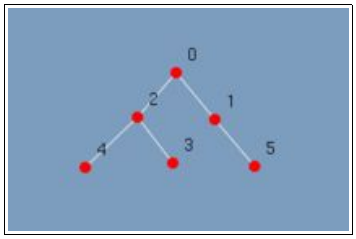
- De cima para baixo da esquerda para a direita.

Qual a ordem que os vértices foram destacados?

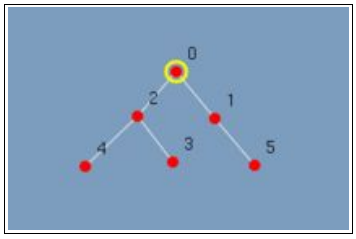
- 0, 2, 4, 3, 1, 5

Quantos passos tem esse algoritmo?

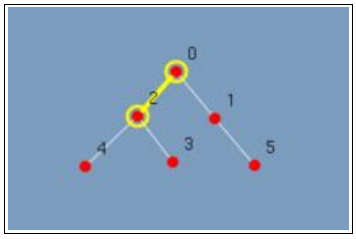
- 6 passos desde a marcação do primeiro vértice.



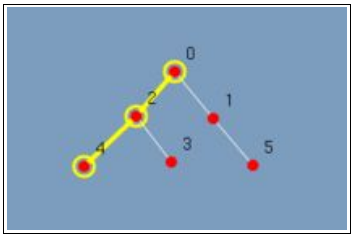
(figura 5.3)



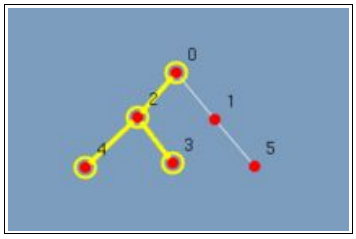
(figura 5.4)



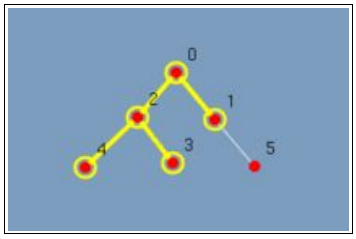
(figura 5.5)



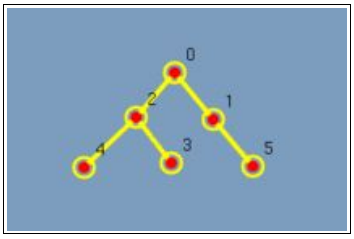
(figura 5.6)



(figura 5.7)



(figura 5.8)



(figura 5.9)

5.3 Interagindo com o algoritmo

Através da *barra de ferramentas*, figura 5.10, é possível uma interação básica com o algoritmo. Ao acionar o algoritmo pelo menu ele lentamente vai sendo executado, porem é possível através dos botões ir direto ao resultado final, voltar ao começo, ir somente um passo a diante, retroceder um passo ou colocá-lo para executar *passo a passo* novamente.

(figura 5.10)

- **First Step**

Retorna o primeiro passo gravado.

- **Prev Step**

Retorna o passo anterior.

- **Execute**

Inicia a execução passo a passo em modo automático.

- **Next Step**

Avança para o próximo passo.

- **Last Step**

Avança para o último passo gravado.

5.4 Como foi implementado

A idéia inicial foi implementar a interface *passo a passo* de uma forma genérica sem que fosse necessária intervenção explícitas nos algoritmo. E assim todo e qualquer algoritmo no **GRAFO** poderia ser executado com ou sem o *passo a passo*, porem devido a modularidade do programa fez com que essa abordagem não fosse possível. No **GRAFO** existe uma clara divisão entre o que é interface com o usuário, estrutura de desenho, estrutura de *grafos* e estrutura de plugins e algoritmos. Os algoritmos nativos ou por plugins não têm acesso direto a interface com o usuário e a estrutura de desenho. Ao serem executados eles se preocupam em manipular grafos, arestas e vértices.

O reflexo desta alterações ao usuário é feito pela estrutura de desenho em conjunto com a interface.

A estratégia adotada foi de que os algoritmos continuassem a manipular somente o que eles conhecem, que são grafos, arestas e vértices, para isto, foi criado um mecanismo de gravação e reprodução de estados. A cada estado interessante o algoritmo explicitamente grava a situação do grafo e das listas de destaque de vértices e arestas. Ao concluir a execução temos um filme com os passos relevantes para a compreensão do algoritmo. E este filme é disponibilizado ao usuário.

5.5 Headers

Para criar o mecanismo de gravação e reprodução utilizou-se uma lista onde cada nodo contém em que contexto está o *grafo*, arestas e vértices em destaque no momento que se instrui a gravação. Abaixo segue as estruturas e funções implementadas para tratar o *passo a passo*.

step.h

O *header step.h* deve ser incluído em um plugin sempre que se desejar que este tenha os recursos do *passo a passo*

Step struct - Estrutura de um passo

A estrutura *Step* (tabela 5.1) contém as informações necessárias a um passos que são: O seu identificador, uma cópia do grafo, os vértices em destaque, as arestas em destaque e uma mensagem associada ao passo salvo.

Nome	Tipo	Descrição
id	int	Identificador do passo. Auto incremental e único
G	Graph	*Grafo* associado ao passo
SpecEdges	List *	Lista de destaque de arestas associadas ao passo
SpecVerts	List *	Lista de destaque de vértices associados ao passo
status_string	char[1000]	Mensagem para a barra de status associada ao passo

(tabela 5.1)

StepStruct struct - Estrutura de múltiplos passos

A estrutura *StepStruct* (tabela 5.2) contém as informações necessárias para a reprodução dos passo: A quantidade de passos salvos, o passo atual, a lista de passos salvos e o status da execução do *passo a passo*.

Nome	Tipo	Descrição
size	int	Número de passos gravados
index	int	Contador de passo. Informa o passo atual
steps	Step[MAX_STEPS]	Lista contendo os passos gravados
exec_state	char	Status de execução (EXECUTING_STEPS, NOT_EXECUTING_STEPS)

(tabela 5.2)

Functions - Funções para manipulação do *passo a passo*

A tabela 5.3 traz a lista de funções para manipulação do passo a passo. A função *AddStep* é a única de uso externo, através de plugins, as demais são de uso interno a implementação do *passo a passo*. Um algoritmo somente deve-se preocupar em salvar os passos através do *AddStep* a inicialização da estrutura *Step* e *StepStruct* e o início automático da execução são atribuições da interface do **GRAFO**.

Declaração	Descrição
void InitStepStruct(void)	Inicializa a estrutura passo a passo
void AddStep(Graph *G, List *arc_edges, List *verts)	Adiciona o *grafo* G e as listas arc_edges e verts no próximo passo livre
void ShowStep(Graph *G, List *arc_edges, List *verts, int step)	Vai para o passo informado e o copia para a interface o *grafo* G e as listas arc_edges e verts
void FirstStep(Graph *G, List *arc_edges, List *verts)	Vai para o primeiro passo gravado e o copia para a interface o *grafo* G e as listas arc_edges e verts
void PreviousStep(Graph *G, List	Vai para o passo anterior e o copia para a

*arc_edges, List *verts)	interface o *grafo* G e as listas arc_edges e verts
void NextStep(Graph *G, List *arc_edges, List *verts)	Vai para o próximo passo e o copia para a interface o *grafo* G e as listas arc_edges e verts
void LastStep(Graph *G, List *arc_edges, List *verts)	Vai para o último passo e o copia para a interface o *grafo* G e as listas arc_edges e verts
int GetLastStep(void)	Retorna o número de passos gravados
char GetExecStateStep(void)	Retorna se o *passo a passo* está em execução (EXECUTING_STEPS, NOT_EXECUTING_STEPS)
void SetExecStateStep(char state)	Inicia e termina a execução do *passo a passo* (START_EXEC_STEPS, STOP_EXEC_STEPS)

(tabela 5.3)

5.6 Construindo um algoritmo passo a passo

Como exemplo prático será construído o plugin *Select odd* que tem como característica destacar todos os vértices ímpares e todas as arestas que conectam vértices ímpares. O código fonte completo do plugin *Select odd* encontra-se no Apêndice B.

No capítulo 4 foi abordada a construção de plugins. Neste capítulo iremos complementá-la com a possibilidade de plugins *passo a passo*. Como no plugin comum o *passo a passo* também é dividido em 3 partes. São elas:

1. Inclusão de *headers*
2. Protótipo das funções
3. Corpo das funções

5.6.1 headers

Na primeira parte deve-se incluir os *headers* da linguagem C e os *headers* obrigatórios para todos os plugins e mais o *header step.h* obrigatório para o *passo a passo*:

```
/* headers da linguagem C */
#include <stdio.h>
#include <stdlib.h>

/* headers do GRAFO  */
/* Informacoes da estrutura de plugins */
#include "../pgin.h"

/* Informacoes da estrutura do GRAFO */
#include "../graph.h"

/* Informacoes da estrutura do Passo a Passo */
#include "../step.h"
```

5.6.2 Protótipo das funções

Na segunda parte declara-se o protótipo para a função obrigatória *pgin_info* e o protótipo para as funções de algoritmos e outras funções. Como função de algoritmo foi declarada a *SelectOdd*.

```
/* Funcao obrigatoria pgin_info */
Pgin *pgin_info(void);

/* Funcoes de algoritmos */
int SelectOdd(Graph *G, char *mess, int indx, List *arc_edges, List *verts);

/* Funcoes internas ao plugin*/
...
...
```

Como mencionado no capítulo 4 cada tipo de algoritmo possui uma assinatura diferente para a lista de parâmetros (tabela 5.4). Para algoritmos *passo a passo* temos:

1. Step by step

```
int StepAlgorithm(Graph *G, char *mess, int indx, List *arc_edges, List *verts)
```

Parâmetros	Descrição
`Graph *G`	Aponta para *grafo* na área de trabalho
`char *mess`	Aponta para a mensagem na barra de status
`int indx`	Número do vértice inicial de buscas
`List *arc_edges`	Lista com as arestas destacadas do *grafo* na área de trabalho
`List *verts`	Lista com os vértices destacados do *grafo* na área de trabalho

(tabela 5.4)

5.6.3 Corpo das funções

Análogo a algoritmos tradicionais a terceira parte deve-se criar o corpo para a função obrigatória *pgin_info* e implementar as funções do algoritmo.

pgin_info - Função obrigatória

A *pgin_info* deve adicionar a lista de algoritmos com o tipo correto para algoritmos *passo a passo* que é *PGIN_STEP_ALG*.

```
/* A funcao pgin_info() deve sempre estar presente. O GRAFO ira chama-la
 * para recuperar as informacoes do plugin.
 */
Pgin *pgin_info(void){
Pgin *pgin;

    /* Aloca espaco para o SelectOdd e o PGIN_LIST_END */
    pgin = (Pgin *) malloc(2 * sizeof(Pgin));

    /* Inclui SelectOdd como primeiro algoritmo do plugin */
    pgin[0].type = PGIN_STEP_ALG;
    pgin[0].label = "Select odd";
    pgin[0].name = "SelectOdd";
    pgin[0].flags = 0;

    /* Indica o final da lista de algoritmos */
    pgin[1].type = PGIN_LIST_END;
    pgin[1].label = 0;
    pgin[1].name = 0;
    pgin[1].flags = 0;
```

```
    return pgin;
}
```

SelectOdd - Funções de algoritmos

A função *SelectOdd* deve-se preocupar em gravar passos que ilustram a execução do algoritmo, os passos gravados que serão disponibilizado para a interação do usuário.

```
int SelectOdd(Graph *G, char *mess, int indx, List *arc_edges, List *verts) {
SEdge E; /* Single Edge */
Node N;
int i, j,oddvertex;

    /* Testa se o grafo e vazio */
    if (G->size < 1) {
        sprintf(mess, "%s", "Empty graph!");
        return 1;
    }

    /* Seta para falso (0) a propriede 'mark' em todos os vertices */
    ClearMarkedVerts(G);

    /* Adiciona o primeiro passo - Nada destacado */
    AddStep(G, arc_edges, verts);

    /* Procura e destaca os vertices impares */
    oddvertex = 0;
    for (i = 1; i < G->size; i++) {

        /* Verifica se i e um numero impar */
        if ((i % 2)) {
            /* Marca o vertices para uso futuro */
            G->vertex[i].mark = 1;
            oddvertex = i;

            /* Coloca em um novo nodo o vertice impar */
            N = MakeNode(&oddvertex, sizeof(int));

            /* Coloca o nodo na lista de vertices especiais */
            InsertNode(verts, N);

            /* Adiciona novo passo - Salva o vertice destacado */
            AddStep(G, arc_edges, verts);
        }
    }

    /* Seta para falso (0) a propriede 'mark' em todas as arestas */
    ClearMarkedEdges(G);
```

```
            /* Procura por um vertice impar adjacente a outro vertice impar */
            for (i = 0; i < G->size; i++) {
                for (j = 0; j < G->size; j++) {

                     /* Evita o salvamento do passo [j,i] porque o passo [i,j] ja foi
salvo */
                    if (!G->prop[j][i].mark && \

                        /* Verdadeiro se o vertex[i] e adjacente ao vertex[j] */
                        G->edge[i][j] && \

                        /* Verdadeiro se o vertex[i] e um vertice impar */
                        G->vertex[i].mark && \

                        /* Verdadeiro se o vertex[j] e um vertice impar */
                        G->vertex[j].mark) {

                        /* Marca a aresta para uso futuro */
                        G->prop[i][j].mark = 1;

                        /* Atribui um vertice impar adjacente a outro vertice impar no
vertice simples */
                        E.u = i;
                        E.v = j;

                        /* Coloca em um novo nodo a aresta simples  */
                        N = MakeNode(&E, sizeof(SEdge));

                        /* Coloca o nodo na lista de arestas especiais */
                        InsertNode(arc_edges, N);

                        /* Adiciona novo passo - Salva a aresta destacada */
                        AddStep(G, arc_edges, verts);
                    }
                }
            }

            return 1;
        }
```

No algoritmo *Select odd* foi utilizado três vezes a chamada da função *AddStep*. A primeira vez se preocupa em salvar contexto do *grafo* sem o destaque de vértices e arestas. A segunda chamada se preocupa em salvar o momento que vértices ímpares são inseridos na lista especial *verts*. A terceira chamada se preocupa em salvar o momento em que as arestas são inseridas na lista especial *arc_edges*.

A segunda chamada da função *AddStep* poderia ser suprimida. E com isso os vértices e arestas apareceriam em destaque no *passo a passo* em conjunto. Com isso concluímos que o posicionamento da chamada *AddStep* no algoritmo é essencial para sua compreensão. No exemplo o que se deseja é mostrar que primeiramente todos os vértices impares são destacados e depois que todas as arestas entre vértices ímpares são destacadas.

Um outro cuidado que se deve ter é de não salvar passos adicionais ou repetidos que não acrescentem em nada a compreensão do algoritmo. No exemplo utiliza-se a propriedade *mark* para informar que uma aresta já foi destacada. Por exemplo se foi salvo o passo quando a aresta (1,3) é destacada não existe a necessidade de salvar o passo (3,1) pois a aresta é a mesma. Abaixo segue o trecho de código que faz este controle.

```
                    ...
                    ...

                     /* Evita o salvamento do passo [j,i] porque o passo [i,j] ja foi
salvo */
                    if (!G->prop[j][i].mark && \
                        ...
                        ...
                        /* Marca a aresta para uso futuro */
                        G->prop[i][j].mark = 1;

                        ...
                        ...
```

5.7 Compilação e Instalação

O procedimento de compilação deve ser reproduzido para os plugins *passo a passo*.

2. Compilação

Para compilar o plugin de exemplo *SelectOdd* que está no arquivo *selectoddexample.c* utilizamos:

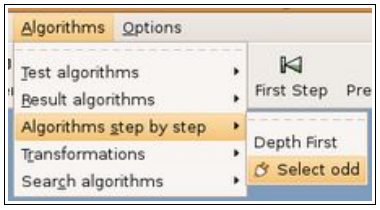
```
gcc -Wall -fPIC -shared selectoddexample.c -o selectoddexample.so
```

3. Instalação

Para instalar copiamos a biblioteca gerada (*selectoddexample.so*) para o diretório de plugins.

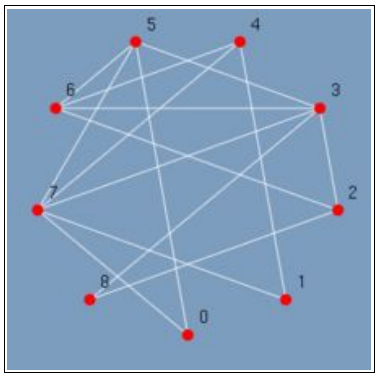
5.8 Execução

Ao abrir o **GRAFO** o novo plugin estará disposto abaixo do menu *Algorithms > Algorithms step by step* vide figura 5.11.

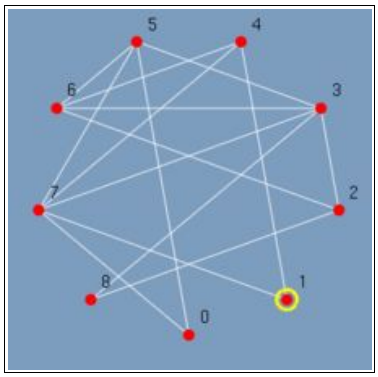


(figura 5.11)

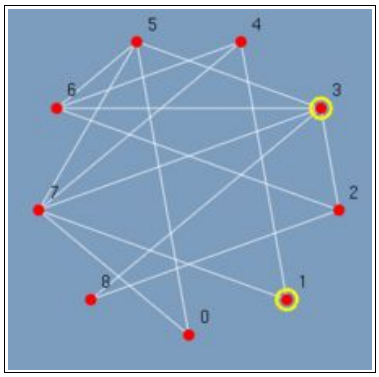
Ao clicar no menu *Select odd* o **GRAFO** chama a função de algoritmo associado a ele, o algoritmo é executado gravando todos os passos relevantes, e ao término o **GRAFO** exibe lentamente o resultado (Figura 5.12 até 5.20). Observando-se os passos pode-se ter uma idéia de como o algoritmo *Select odd* foi construído. Primeiro são destacados todos os vértices ímpares do menor para o maior *{1, 3, 5, 7}*. E depois todas as arestas entre vértices ímpares do menor vértice para o maior *{(1,7), (3,5), (3,7), (5,7)}*.



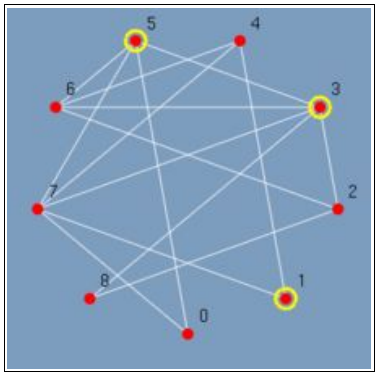
(figura 5.12)



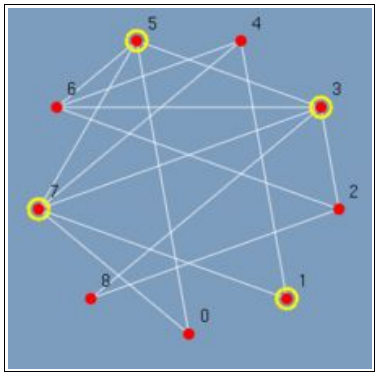
(figura 5.13)



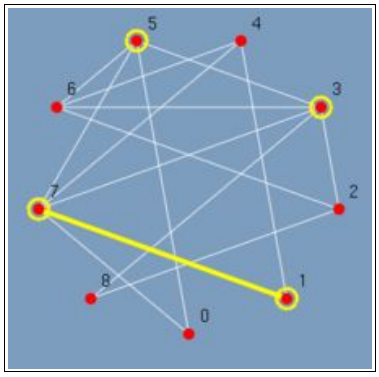
(figura 5.14)



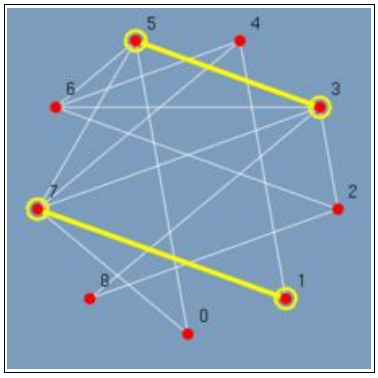
(figura 5.15)



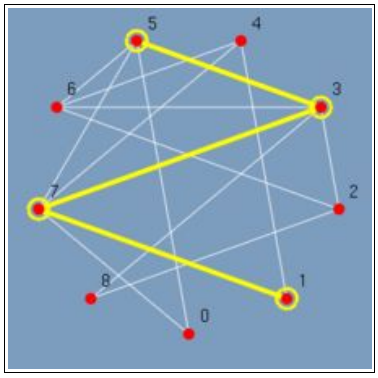
(figura 5.16)



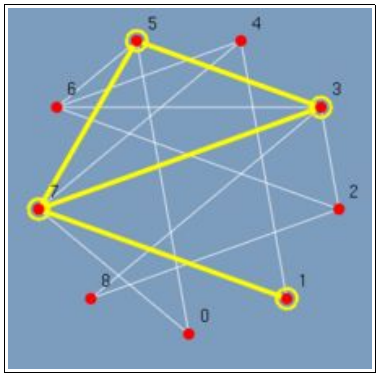
(figura 5.17)



(figura 5.18)



(figura 5.19)



(figura 5.20)

6 CONSIDERAÇÕES FINAIS

A evolução de um software é um processo contínuo que envolve manutenções corretivas, implementação de novas funcionalidades e documentação. Este capítulo detalha o que foi realizado por este trabalho e expõem algumas idéias na intenção de convidar ao leitor a novas possibilidades de implementações e/ou manutenções no **GRAFO**.

6.1 O que foi realizado

Durante a confecção deste trabalho foram realizadas algumas manutenções corretivas e a implementação de novas funcionalidades como o *passo a passo* e este documento que serve como documentação para os trabalhos futuros. Tais implementações serão citadas uma a uma nos itens *6.1.1*, *6.1.2* e *6.1.3* para que o leitor possa acompanhar e perceber a dimensão deste trabalho.

6.1.1 Manutenções corretivas

- Inclusão da diretiva *-Wl,-export-dynamic* no arquivo Makefile.in para fazer que os símbolos definidos no *main* fossem vistos pelos plugins. Isso corrigiu o problema do tipo *"plugin.so: undefined symbol: FOO"* quando o plugin chamava alguma função do programa principal.

- Remoção da referência da macro *msh* do arquivo *aclocal.m4*. Apesar de não gerar erro na compilação essa macro não existia e não foi encontrado referência sobre ela na internet. Para remover os *warnings* gerados na compilação ela foi removida.

- Correção da compilação do algoritmo *Delaunay triangulation* e incluindo-o na lista de plugins disponíveis.

- Correção da compilação do algoritmo *Chordalize a Graph* e incluindo-o na lista de plugins disponíveis.

- Remoção de alguns arquivos de "Lixo" como "src/plugins/Makefile.x"

- Execução da operação de limpar os vértices e arestas destacados antes de abrir um *grafo* de um arquivo. O destaque do *grafo* que estava na área de trabalho era assumida pelo *grafo* carregado de um arquivo gerando distorções.

- Alteração dos Captions e Labels nos diálogos de geração de *grafos* para manter a coerência de nomes. Em alguns lugares utilizava a palavra *Create* e em outros *Generate*.

6.1.2 Novas funcionalidades

- Melhoramento a infraestrutura do **GRAFO** com a sua inclusão em controle de versão utilizando o software *Subversion - SVN*.

- Geração de *tag* no *SVN* da versão original do programa disponibilizada pelos autores originais. E com isso manter o histórico e a evolução do *software*.

- Inclusão do tratamento para algoritmos *passo a passo* na lista de algoritmos disponíveis no **GRAFO**. Até então o menu *passo a passo* existia porem não tinha funcionalidade implementada.

- Inclusão de ícones para o passo a passo na toolbar e dos arquivos

- Inclusão *step.h* e *step.c* para manipular a estrutura *passo a passo*.

- Inclusão do algoritmo *Depth Search* com suporte a *passo a passo*.

- Inclusão dos flag PGIN_FLAG_REDRAWASCIRCLE e PGIN_FLAG_REDRAWWITHCENTEREDVERTEX para redraw automáticos nos plugins de transformação que respectivamente redesenha o *grafo* em circulo ou com um vértice central.

- Inclusão do plugin stargraphexample.c que serve para exemplificar a utilização do AddPoint e AddEdge.

- Inclusão do plugin selectoddexample.c. O plugin destaca todos os vértices ímpares e todas as arestas que conectam dois vértices ímpares. O objetivo do plugin é ilustrar a construção de um algoritmo com os recursos do *passo a passo*.

6.1.3 Documentação

- Inclusão de novos comentários e organização de tabulações dos plugins *selectoddexample.c*, *getsp.c*, *vertexexample.c*. *edgeexample.c, hw.c*.

- Criação da pasta *docs* e disponibilização deste trabalho no formato *HTML* além do formato *txt2tags*, *PDF* e *ODT*.

- Criação da pasta *examples* contendo os *grafos* no formato *.gr* dos exemplos utilizados neste trabalho.

6.2 O que há por vir

Além do que já foi realizado por este trabalho, foram observados outros pontos onde existe a possibilidade de manutenção corretiva e oportunidades de melhorias.

6.2.1 Manutenções Corretivas

- O *redraw* utilizando o *neato* e o *dot* não está funcionando. Quando utilizado o *neato* e o *dot* para redesenhar o *grafo* da área de trabalho este fica como os vértices alinhados em uma reta e não como o esperado do retorno do *neato* e o *dot*.

- Executando *Valgrid (Debugger de memória)* observa-se pontos de *memory leaks* que podem ser eliminados para melhorar a estabilidade do programa.

- Ao sair do programa pelo botão fechar (X) da janela ocorre falha de segmentação.

- Quando uma aresta e vértices são rotulados (*labels*) e/ou são atribuídos pesos (*weight*) o programa fica instável e quebra facilmente gerando falha de segmentação.

6.2.2 Novas funcionalidades

- Finalizar o suporte de *digrafos*. O **GRAFO** já possui boa parte do suporte a *digrafos* implementada como a matriz de adjacência, a possibilidade de inserção de *arcos*, porem a interface não está preparada para desenhar e manipular as flechas e nem fazer a alteração de tipo entre *grafo* e *digrafo*.

- Concluir a funcionalidade de gerar grafos regulares. Esta opção está no menu porem não existe o algoritmo implementado.

- Salvar nos arquivos *.gr* a curvatura de arestas. Quando se utiliza dos *control points* para curvar arestas esta curvatura não é salva no *.gr* e ao carregar novamente o *grafo* as arestas são desenhadas com retas.

- Melhorar as estruturas internas otimizando o consumo de memória. Algumas estruturas do programa não fazem alocação dinâmica de memória. Alocam ao iniciar uma quantidade fixa e pressupõem que esta seja o suficiente.

- Melhorar a edição vértices e arestas. É burocrático ter que clicar em cada um dos vértices para se editar uma aresta. Seria suficiente clicar na própria aresta. Em vértices quando se está utilizando *control points* ocorre uma sobreposição entre a área de ação sobre um vértice e sobre os *control points*.

- Possibilidade de arrastar um grafo inteiro. Muitas vezes é desejável arrastar todo o *grafo* e manter o seu *layout*. Hoje isto não é possível sendo necessário mover vértice a vértice.

- Implementar novos algoritmos como *Dual*, *Planar*, *Coloração Ótima*, *Djsktra*, *Prim* entre outros.

- Possibilitar salvar e reproduzir algoritmos passo a passo. Pode ocorrer de um algoritmo demorar muito para ser executado, a possibilidade de salvar os passos de execução pode evitar ter que roda-lo toda vez.

- Possibilitar salvar como imagem a lista de reprodução passo a passo. É a possibilidade de salvar a lista de reprodução acima para por exemplo incluir os passos de execução do

algoritmo em um documento e assim evitar ter que salvar passo por passo manualmente com um editor de imagens.

- Possibilitar carregar e descarregar plugins sem necessidade de sair do ambiente. Esta funcionalidade é extremamente útil quando está se codificando um novo algoritmo com isto evita-se ter que ficar entrando toda vez na interface do *grafo* para realizar um novo teste.

- Possibilitar trabalhar com mais de um grafo de maneira simultânea. Hoje o **GRAFO** esta limitado a um só *grafo* por interface. Com isso não é possível realizar por exemplo a operação entre dois *grafos* como união ou decomposição.

6.3 Software livre

O **GRAFO** é um software livre sobre a GPL versão 2. Porem ele está restrito ao ambiente da UFPR. Como um software livre qualquer um pode contribuir para o seu desenvolvimento. Encontrar um repositório público e disponibiliza-lo para a comunidade dará longevidade ao software. O Apêndice A detalha como é possível obter os fontes a partir do repositório na *UFPR*.

6.4 Conclusão

Com este trabalho conseguiu-se cumprir com algumas estapas do processo de evolução do **GRAFO**. Fazendo-se manutenções corretivas, implementando-se novas funcionalidades e melhorando a documentação. A expectativa é que agora outros venham a cumprir outras etapas no processo, como realizado por *Murilo Vicente*, *Oliver Matias* e por mim.

REFERENCIAS BIBLIOGRÁFICA

1. Bodnar, Jan. **The GTK+ tutorial**. http://zetcode.com/tutorials/gtktutorial/, Programação com o Toolkit GTK+. Acessado em maio de 2008.

2. GAGNON, Michel. **CI065 - Algoritmos e teoria dos grafos - Notas de Aula**, http://www.inf.ufpr.br/andre/Disciplinas/BSc/CI065/michel/grafos.html. Algoritmos e teoria dos grafos. Acessado em junho de 2008.

3. SZWARCFITER, Jayme Luiz. **Grafos e Algoritmos Computacionais**. 1° Edição. Rio de Janeiro: Campus, 1984.

4. WIKIPEDIA. **Graph theory**, http://en.wikipedia.org/wiki/Graph_theory. Teoria dos Grafos. Acessado em junho de 2008.

5. Wheeler, David A. **Program Library HOWTO**, http://www.dwheeler.com/program-library/Program-Library-HOWTO/index.html. Bibliotecas Dinâmicas. Acessado em abril de 2008.

APÊNDICES

Apêndice A - Compilando o GRAFO

Esse capitulo tem a intenção de auxiliar a quem vier a fazer uma futura manutenção ou melhoria no **GRAFO**. Mostra de maneira simplificada a estrutura de diretórios e o conteúdo de cada arquivo e os pacotes de dependência do **GRAFO** e onde pode-se obter os arquivos fontes.

Obtenção dos Fontes

Os fontes devem ser obtidos através do *Subversion* do *AlgLab* para isto deve-se solicitar a inclusão de sua chave pública ao professor *André Guedes* <andre@inf.upfr.br>.

- **Gerando chave pública**

Para geração de chave pública *RSA* execute o comando abaixo:

```
ssh-keygen -t rsa
```

Enviar o arquivo `{$HOME}\.ssh\id_rsa.pub` ao professor *André Guedes*.

- **Fazendo *Checkout* dos fontes**

```
mkdir <DIRETÓRIO DO GRAFO>
```

```
svn checkout svn+ssh://andre@fradim.inf.ufpr.br/alglab/grafo/ <DIRETÓRIO
DO GRAFO>
```

Pacotes necessários

Os pacotes abaixo relacionados são necessários para a compilação. Em muitos ambientes alguns destes já virão instalados por padrão. Esta lista foi obtida a partir de um *Linux Ubuntu 7.10* com a sua instalação padrão.

```
sudo apt-get install g++
sudo apt-get install build-essential
sudo apt-get install libtool
sudo apt-get install libgtk2.0-dev
sudo apt-get install libgtkgl2.0-dev
sudo apt-get install xlibmesa-gl-dev
sudo apt-get install libgl1-mesa-dev
sudo apt-get install gtkglarea5-dev
sudo apt-get install freeglut3-dev
```

Compilação, Execução e Instalação

- **Compilação**

Para gerar o programa executável a partir dos fontes obtidos do *Subversion* faça:

```
cd <DIRETÓRIO DO GRAFO>

cd src
```

```
./configure
```

```
make
```

- **Execução**

Para executar o programa a partir do compilação acima faça:

```
./grafo
```

- **Instalação**

O arquivo executável será instalado no diretório `/usr/bin`. É necessário privilégio de *root* para fazer esta operação.

```
make install
```

Estrutura e conteúdo de diretórios

A seguir segue a estrutura de diretório e uma breve descrição do conteúdo dos arquivos que compõem o **GRAFO**.

- **/grafo**
  - AUTHORS
  - CHANGES
  - COPYING
  - README
  - **/icons**
    - k3.xpm
    - k5.xpm
  - **/docs**
    - <Arquivos para composição deste documento>
  - **/examples**
    - <Arquivos de exemplos de grafos>

- **/src**
    - aclocal.m4
    - alghandler.c
    - alghandler.h
    - color.h
    - color.c
    - config.h.in
    - configure
    - configure.in
    - draw.c
    - draw.h
    - geometry.c
    - geometry.h
    - gr.c
    - grafodoc.t2t
    - graph.c
    - graph.h
    - list.c
    - list.h
    - Makefile.in
    - pgin.c
    - pgin.h
    - plugins.c
    - plugins.h
    - rc.c
    - rc.h
    - rw.c
    - rw.h
    - step.c

- step.h
- **/iface**
    - callbacks.c
    - callbacks.h
    - factory.c
    - factory.h
    - iface.c
    - iface.h
    - pixmaps.h
    - pixmaps2.h
- **/plugins**
    - chcolor.c
    - chordalize.c
    - components.c
    - conected.c
    - delaunay.c
    - edgeexample.c
    - euler.c
    - getsp.c
    - hw.c
    - Makefile
    - paley.c
    - selectoddexample.c
    - stargraphexample.c
    - vertexexample.c

Descrição e conteúdo dos arquivos

- **/grafo**

AUTHORS

Lista de autores do GRAFO.

CHANGES

Descrição de correções, melhorias em cada versão.

COPYING

Cópia da GNU GENERAL PUBLIC LICENSE

README

Breve apresentação e instruções para compilação.

- **icons**

k3.xpm

Ícone do GRAFO com 3 vértices.

k5.xpm

Ícone do GRAFO com 5 vértices.

- **/docs**

Contém os arquivos necessários para confeccionar este documento

- **/examples**

Contém os exemplos de grafos no formato (.gr) dos exemplos utilizados neste documento

- **/src**

alghandler.{c,h}

Responsável por manipular a lista de algoritmos disponíveis para o GRAFO. A lista pode incluir algoritmos nativos ou em plugins.

algorithms.{c,h}

Contem a lista de algoritmos nativos do GRAFO.

color.{c,h}

Definição de cores disponíveis.

draw.{c,h}

Responsável por tratar a estrutura que destaca vértices e arestas na interface do GRAFO.

geometry.{c,h}

Redesenha o grafo em uma nova forma (em circulo, ocupando o máximo da janela) ou utiliza o Neato e o DOT para redesenhar o grafo.

gr.c

Programa principal; aloca espaço para a estruturas do grafo e desenho; instala algoritmos nativos; chama instalador de plugins; chama leitura de configurações;

graph.{c,h}

Manipula estrutura do grafo; insere, remove, move e edita vértices e arestas; copia; aloca; inicializa estrutura para o grafo.

list.{c,h}

Implementação de lista e pilha para ser utilizado pelo grafo. Ex Lista de algoritmos, pilha de vértices visitados etc.

pgin.{c,h}

Responsável por criar e exibir os diálogos de entrada de parâmetros.

plugins.{c,h}

Manipula carga e descarga de plugins.

rc.{c,h}

Manipula leitura e escrita do arquivo de configuração (.graforc).

rw.{c,h}

Manipula leitura e escrita dos formatos de arquivos suportados pelo grafo. (*.gr*, *.dot*, *.ps*)

step.{c,h}

Estruturas para implementação de algoritmos passo a passo.

- **/iface**

callbacks.{c,h}

Responsável por tratar todas as chamadas da interface. Menus, Toolbar e etc.

factory.{c,h}

Responsável por criar os componentes da interface.

iface.{c,h}

Responsável pela declaração dos componentes da interface.

pixmaps{,2}.h

Conjunto de imagens utilizadas em botões e menus.

- **/plugins**

chcolor.c

Colore um grafo cordal usando algoritmo descrito em Gavril [72].

Obs.: Plugin não finalizado pelos autores originais.

chordalize.c

Insere arestas de maneira a transformar um grafo em um grafo cordal.

components.c

Retorna o número de componentes do grafo.

conected.c

Testa a conectividade do grafo.

delaunay.c

Cria triangulação Delaunay de um conjunto de vértices.

edgeexample.c

Exemplo de como utilizar a lista de arestas especiais *special edge* dento do GRAFO. Esta lista que é responsável por destacar as arestas dentro da interface do GRAFO.

euler.c

Testa se o grafo tem um *caminho de Euler*.

getsp.c

Exemplo de como trabalhar com passagem de parâmetros para o grafo. Como inteiros, ponto flutuante e strings.

hw.c

Exemplo de plugin minimalista *Hello World!*.

Makefile

Arquivo de makefile para compilar os plugins.

paley.c

Gera um grafo de *Paley* a partir de um número de vértices informado.

selectoddexample.c

Seleciona todos os vértices ímpares e todas as arestas que conectam dois vértices ímpares

stargraphexample.c

Gera um grafo em estrela de acordo com o número de vértices informado.

vertexexample.c

Exemplo de como utilizar a lista de vértices especiais *special vertex* dento do GRAFO. Esta lista que é responsável por destacar os vértices dentro da interface do GRAFO.

Apêndice B - Fontes dos plugins de exemplo

Hello Word! - (hw.c)

```
/*
 *  grafo - graph editor
 *  Copyright (c) 2003
 *      Murilo Vicente Goncalves da Silva <murilo@pet.inf.ufpr.br>
 *      Oliver Matias van Kaick <oliver@pet.inf.ufpr.br>
 *      Ulisses Cordeiro Pereira <ulisses@cordeiropereira.com.br>
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *  See the GNU General Public License for more details (file COPYING).
 */


/* ******** */
/* * hw.c * */
/* ******** */


/* Plugin example: hello world */

/* C language headers */
#include <stdio.h>
#include <stdlib.h>

/* Here the GRAFO headers */
/* Add plugins information */
#include "../pgin.h"

/* Add graph struct information */
#include "../graph.h"

/* Mandatory function pgin_info */
Pgin *pgin_info(void);

/* algorithm function */
```

```
int HelloWorld(Graph *G, char *mess);

/* Others functions */


/* The pgin_info() function should always be present. GRAFO will call
 * it to get the plugin information.
 */
Pgin *pgin_info(void){
Pgin *pgin;

    /* Alocate space for HelloWorld and PGIN_LIST_END */
    pgin = (Pgin *) malloc(2 * sizeof(Pgin));

    /* Includes HelloWorld as the first algorithm of plugin */
    pgin[0].type = PGIN_TEST_ALG;
    pgin[0].label = "Hello World!";
    pgin[0].name = "HelloWorld";
    pgin[0].flags = 0;

    /* It indicates the end of the list of algorithms */
    pgin[1].type = PGIN_LIST_END;
    pgin[1].label = 0;
    pgin[1].name = 0;
    pgin[1].flags = 0;

    return pgin;
}

/* This is the plugin's main function */
int HelloWorld(Graph *G, char *mess){

    sprintf(mess, "Hello World!");

    return 1;
}

/* ******* */
/* * End * */
/* ******* */
```

Get some parameters - (getsp.c)

```
/*
 *  grafo - graph editor
 *  Copyright (c) 2003
 *      Murilo Vicente Goncalves da Silva <murilo@pet.inf.ufpr.br>
 *      Oliver Matias van Kaick <oliver@pet.inf.ufpr.br>
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *  See the GNU General Public License for more details (file COPYING).
 */


/* ********** */
/* * getsp.c * */
/* ********** */


/* Plugin example: get some input parameters */

#include <stdio.h>
#include <stdlib.h>
/* Here the GRAFO headers */
#include "../pgin.h"
#include "../graph.h"


Pgin *pgin_info(void);
int getsp(Graph *G, char *mess);


Pgin *pgin_info(void){
Pgin *pgin;

    pgin = (Pgin *) malloc(2 * sizeof(Pgin));

    pgin[0].type = PGIN_TEST_ALG;
    pgin[0].label = "Get some parameters";
    pgin[0].name = "getsp";

    pgin[1].type = PGIN_LIST_END;
    pgin[1].label = 0;
```

```
    pgin[1].name = 0;

    return pgin;
}


int getsp(Graph *G, char *mess){
int i;
double d;
char *s;

    i = pgin_read_int("Get some parameters", "Give me an int",
                10, 1, 2000000, 1);
    d = pgin_read_double("Get some parameters", "Give me a double",
                0.0, -10.0, 10.0, 0.1, 1);
    s = pgin_read_string("Get some parameters", "Give me a string",
                "I'm a string", 40);

    sprintf(mess, "I got: %d, %f and '%s'. Thanks!", i, d, s);
    free(s);

    return 1;
}


/* ******* */
/* * End * */
/* ******* */
```

Generate Star Graphs - (stargraphexample.c)

```
/*
 *  grafo - graph editor
 *  Copyright (c) 2003-2008
 *      Ulisses Cordeiro Pereira <ulisses@cordeiropereira.com.br>
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *  See the GNU General Public License for more details (file COPYING).
 */


/* ****************** */
/* * stargraphexample.c * */
/* ****************** */

/* Plugin example: Generate a strar graph */

#include <stdio.h>
#include <stdlib.h>

/* Here the GRAFO headers */
/* Add plugins information */
#include "../pgin.h"

/* Add graph struct information */
#include "../graph.h"

/* Prototype functions */
Pgin *pgin_info(void);
int stargraph(Graph *G, char *mess);


/* This function returns the plugin info */
Pgin *pgin_info(void){
Pgin *pgin;

    pgin = (Pgin *) malloc(2 * sizeof(Pgin));

    pgin[0].type = PGIN_TRANSF_ALG;
    pgin[0].label = "Generate Star Graph";
    pgin[0].name = "stargraph";
```

```
    pgin[0].flags = PGIN_FLAG_REDRAWWITHCENTEREDVERTEX;

    pgin[1].type = PGIN_LIST_END;
    pgin[1].label = 0;
    pgin[1].name = 0;
    pgin[1].flags = 0;

    return pgin;
}

/* This is the plugin's main function */
int stargraph(Graph *G, char *mess){
int i, n;

    /* Get number of vertices */
    n = pgin_read_int("Generate Star Graph", "Number of vertices",
                5, 1, 2000000, 1);

    /* Generate graph */
    InitGraph(G, G->type);

    /* Adds N vertices in the position (0,0) */
    for (i = 0; i < n; i++)
        AddPoint(G, 0, 0);

    /* Adds N edges starting at the vertex 0 to N-1 */
    for (i = 1; i < n; i++) {
        AddEdge(G, 0, i);
    }

    return 1;
}

/* ******* */
/* * End * */
/* ******* */
```

Highest Degree - (vertexexample.c)

```
/*
 *  grafo - graph editor
 *  Copyright (c) 2003-2008
 *      Murilo Vicente Goncalves da Silva <murilo@pet.inf.ufpr.br>
 *      Oliver Matias van Kaick <oliver@pet.inf.ufpr.br>
 *      Ulisses Cordeiro Pereira <ulisses@cordeiropereira.com.brr>
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *  See the GNU General Public License for more details (file COPYING).
 */


/* ****************** */
/* * vertexexample.c * */
/* ****************** */

/* Plugin example: This program does not compute anything useful.
 * It is an example of how to use the "special vertex" list.
 * The vertices in this list will be highlighted in the GRAFO
 * interface.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* Here the GRAFO headers */
#include "../pgin.h"
#include "../graph.h"
#include "../list.h"

/* You have to fill the pgin struct with information about your plugins.
 * Each position of the pgin struct is related with some function you
 * have programed. Below one example where you programmed five functions:
 *
 * Pgin pgin[6] = {
 *     { PGIN_RESULT_ALG, "Find a Hamilton Path", "HPath", 0 },
 *     { PGIN_TEST_ALG, "Test Planarity", "PlanarGraph", 0 },
 *     { PGIN_TEST_ALG, "Test Bergeness", "BergeGraph", 0 },
 *     { PGIN_TRANSF, "Compute the complement", "Compl", PGIN_FLAG_NOREDRAW },
 *     { PGIN_TRANSF, "Line Graph", "LineGraph", 0 },
```

```
 *     { PGIN_LIST_END, NULL, NULL, 0 },
 * };
 *
 * In the last position you should put { PGIN_LIST_END, NULL, NULL }, to
 * end the list of plugins.
 * You have to put four pieces of information in the pgin struct:
 * -> The first is the class of your algorithm. The first entry
 * above is a Result Algorithm, so you have to put PGIN_RESULT_ALG
 * -> The second (in this case "Highest Degree") is the name that will
 * appear in the menu of GRAFO.
 * -> The third is the name of the function in the code (in this
 * case "MaxDegree").
 * -> The fourth is a set of plugin flags. In the example of the
 * fouth funtion the flag prevents that GRAFO redraws the resulting graph.
 * If you do not know what is a "plugin flag", only put 0 in this field.
 */


/* Plugin Information */
Pgin pgin[2] = {
    { PGIN_RESULT_ALG, "Highest Degree", "MaxDegree" },
    { PGIN_LIST_END, NULL, NULL },
};



/* The pgin_info() function should always be present. GRAFO will call
 * it to get the plugin information.
 */
Pgin *pgin_info(void);

Pgin *pgin_info(void){

    return pgin;
}

/* Plugin example: This program does not compute anything useful.
 * It is an example of how to use the "special vertex" list.
 * Here this list have the name verts. The vertices in this list will
 * be highlighted in the GRAFO interface.

 * In this example, the function finds the vertex with the highest
 * degree (acctualy one of them) and put it in the verts list. */

int MaxDegree(Graph *G, char *mess, List *arc_edges, List *verts);

int MaxDegree(Graph *G, char *mess, List *arc_edges, List *verts){
Node N;
int i,maxdeg;

    /* Test if graph is empty */
```

```
        if (G->size < 1) {
            sprintf(mess, "%s", "Empty graph!");
            return 1;
        }

        /* Find the vertex with max degree */
        maxdeg = 0;
        for (i = 1; i < G->size; i++) {
            if (G->vertex[i].degree > G->vertex[maxdeg].degree) {
                maxdeg = i;
            }
        }

        /* Place maxdeg in the Node */
        N = MakeNode(&maxdeg, sizeof(int));

        /* Place the node in the list of "special vertex" */
        InsertNode(verts, N);

        return 1;
    }


    /* ******* */
    /* * End * */
    /* ******* */
```

Some Edge - (edgeexample.c)

```
/*
 *  grafo - graph editor
 *  Copyright (c) 2003-2008
 *      Murilo Vicente Goncalves da Silva <murilo@pet.inf.ufpr.br>
 *      Oliver Matias van Kaick <oliver@pet.inf.ufpr.br>
 *      Ulisses Cordeiro Pereira <ulisses@cordeiropereira.com.br>
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *  See the GNU General Public License for more details (file COPYING).
 */


/* ***************** */
/* * edgeexample.c * */
/* ***************** */

/* Plugin example: This program does not compute anything useful.
 * It is an example of how to use the "special edge" list.
 * The edges in this list will be highlighted in the GRAFO
 * interface.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* Here the GRAFO headers */
#include "../pgin.h"
#include "../graph.h"
#include "../list.h"


/* You have to fill the pgin struct with information about your plugins.
 * Each position of the pgin struct is related with some function you
 * have programed. Below one example where you programmed five functions:
 *
 * Pgin pgin[6] = {
 *     { PGIN_RESULT_ALG, "Find a Hamilton Path", "HPath", 0 },
 *     { PGIN_TEST_ALG, "Test Planarity", "PlanarGraph", 0 },
 *     { PGIN_TEST_ALG, "Test Bergeness", "BergeGraph", 0 },
```

```
 *     { PGIN_TRANSF, "Compute the complement", "Compl", PGIN_FLAG_NOREDRAW },
 *     { PGIN_TRANSF, "Line Graph", "LineGraph", 0 },
 *     { PGIN_LIST_END, NULL, NULL, 0 },
 * };
 *
 * In the last position you should put { PGIN_LIST_END, NULL, NULL }, to
 * end the list of plugins.
 * You have to put four pieces of information in the pgin struct:
 * -> The first is the class of your algorithm. The first entry
 * above is a Result Algorithm, so you have to put PGIN_RESULT_ALG
 * -> The second (in this case "Some Edge") is the name that will
 * appear in the menu of GRAFO.
 * -> The third is the name of the function in the code.
 * -> The fourth is a set of plugin flags. In the example of the
 * fouth funtion the flag prevents that GRAFO redraws the resulting graph.
 */


/* Plugin Information */
Pgin pgin[2] = {
    { PGIN_RESULT_ALG, "Some Edge", "MaxMinEdge",0 },
    { PGIN_LIST_END, NULL, NULL, 0 },
};


/* The pgin_info() function should always be present. GRAFO will call
 * it to get the plugin information.
 */
Pgin *pgin_info(void);

Pgin *pgin_info(void){

    return pgin;
}


/* The plugin example: It is an example of how to use the "special edge" list.
 * Here the list have the name arc_edges. The edges in this list will
 * be highlighted in the GRAFO interface.
 *
 * The following edge will be put in the list:
 * Let the egde be called E. We will define it now: Let v be one vertex of the
 * graph such that no other vertex has degree greater than v. Let
 * u be the vertex adjacent to v with the lowest degree. The
 * edge E links u to v. */

int MaxMinEdge(Graph *G, char *mess, List *arc_edges, List *verts);

int MaxMinEdge(Graph *G, char *mess, List *arc_edges, List *verts){
SEdge E; /* Single Edge */
Node N;
```

```
        int i,maxdeg,mindeg;

            /* Test if graph is empty */
            if (G->size < 1) {
                sprintf(mess, "%s", "Empty graph!");
                return 1;
            }

            /* Find the vertex with max degree */
            maxdeg = 0;
            for (i = 1; i < G->size; i++) {
                if (G->vertex[i].degree > G->vertex[maxdeg].degree) {
                    maxdeg = i;
                }
            }

            /* Find the vertex with min degree adjacent with max degree */
            mindeg = -1;
            for (i = 0; i < G->size; i++) {
                if (G->edge[maxdeg][i]) {
                    if (mindeg == -1) {
                        mindeg = i;
                    } else if (G->vertex[i].degree < G->vertex[mindeg].degree) {
                        mindeg = i;
                    }
                }
            }

            /* Assign maxdeg and mindeg in a single edge */
            E.u = maxdeg;
            E.v = mindeg;

            /* Place the "single edge" in the Node */
            N = MakeNode(&E, sizeof(SEdge));

            /* Place the node in the list of "special edges" */
            InsertNode(arc_edges, N);

            return 1;
        }


        /* ******* */
        /* * End * */
        /* ******* */
```

Select odd - (selectoddexample.c)

```
/*
 *  grafo - graph editor
 *  Copyright (c) 2008
 *      Ulisses Cordeiro Pereira <ulisses@cordeiropereira.com.br>
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *  See the GNU General Public License for more details (file COPYING).
 */



/* ********************** */
/* * selectoddexample.c * */
/* ********************** */

/* Plugin example: This program does not compute anything useful.
 * It is an example of how to use the "step by step" structure.
 * The odd vertex and edges between odd vertex will be highlighted in the
GRAFO
 * interface.
 *
 */

/* Plugin example: Select odd */

/* C language headers */
#include <stdio.h>
#include <stdlib.h>

/* Here the GRAFO headers */
/* Add plugins information */
#include "../pgin.h"

/* Add graph struct information */
#include "../graph.h"

/* Add step by step struct information */
#include "../step.h"

/* Mandatory function pgin_info */
Pgin *pgin_info(void);
```

```
/* algorithm function */
int SelectOdd(Graph *G, char *mess, int indx, List *arc_edges, List *verts);

/* Others functions */


/* The pgin_info() function should always be present. GRAFO will call
 * it to get the plugin information.
 */
Pgin *pgin_info(void){
Pgin *pgin;

    /* Alocate space for SelectOdd and PGIN_LIST_END */
    pgin = (Pgin *) malloc(2 * sizeof(Pgin));

    /* Includes SelectOdd as the first algorithm of plugin */
    pgin[0].type = PGIN_STEP_ALG;
    pgin[0].label = "Select odd";
    pgin[0].name = "SelectOdd";
    pgin[0].flags = 0;

    /* It indicates the end of the list of algorithms */
    pgin[1].type = PGIN_LIST_END;
    pgin[1].label = 0;
    pgin[1].name = 0;
    pgin[1].flags = 0;

    return pgin;
}

/* This is the plugin's main function */
int SelectOdd(Graph *G, char *mess, int indx, List *arc_edges, List *verts) {
SEdge E; /* Single Edge */
Node N;
int i, j,oddvertex;

    /* Test if graph is empty */
    if (G->size < 1) {
        sprintf(mess, "%s", "Empty graph!");
        return 1;
    }

    /* Set to False (0) the property 'mark' in all vertices */
    ClearMarkedVerts(G);

    /* Add first step - No highlights */
    AddStep(G, arc_edges, verts);

    /* Find and mark oddvertex */
    oddvertex = 0;
```

```
            for (i = 1; i < G->size; i++) {

                /* Check if i is a odd number */
                if ((i % 2)) {
                    /* Mark the vertex for future use */
                    G->vertex[i].mark = 1;
                    oddvertex = i;

                    /* Place oddvertex in the Node */
                    N = MakeNode(&oddvertex, sizeof(int));

                    /* Place the node in the list of "special vertex" */
                    InsertNode(verts, N);

                    /* Add new step - Save highlight vertex */
                    AddStep(G, arc_edges, verts);
                }
            }

            /* Set to False (0) the property 'mark' in all vertices */
            ClearMarkedEdges(G);

            /* Find the odd vertex adjacent with other odd vertex */
            for (i = 0; i < G->size; i++) {
                for (j = 0; j < G->size; j++) {

                        /* Avoid saving the step [j,i] because [i,j] have been saved
*/
                    if (!G->prop[j][i].mark && \
                        /* True if vertex[i] is adjacent vertex[j] */
                        G->edge[i][j] && \
                        /* True if vertex[i] is a odd vertex */
                        G->vertex[i].mark && \
                        /* True if vertex[j] is a odd vertex */
                        G->vertex[j].mark) {

                        /* Mark the edge for future use */
                        G->prop[i][j].mark = 1;

                            /* Assign odd vertex adjacent with other odd vertex in a
single edge */
                        E.u = i;
                        E.v = j;

                        /* Place the "single edge" in the Node */
                        N = MakeNode(&E, sizeof(SEdge));

                        /* Place the node in the list of "special edges" */
                        InsertNode(arc_edges, N);

                        /* Add new step - Save highlight edge */
```

```
                        AddStep(G, arc_edges, verts);
                    }
                }
            }

            return 1;
        }



        /* ******* */
        /* * End * */
        /* ******* */
```