UNIVERSIDADE FEDERAL DE MINAS GERAIS
Escola de Engenharia
Departamento de Engenharia Elétrica
Programação e Desenvolvimento de Software II

Aluno: Leandro Emílio de Paula

Matrícula: 2012022221

Programação e Desenvolvimento de Software II

Trabalho Prático

Neste trabalho foi criada uma classe para representar o conceito de grafo (classe Graph), com vértices representados por números inteiros. Como os vértices são muito simples, não foi necessário cria uma classe para representa-los.

Graph utilizará uma representação interna por matriz de adjacência. A classe Graph possui:

a) Um construtor, que recebe como parâmetro um inteiro indicando o número de vértices do grafo;

b) Um destrutor, que se incumbe de fazer a desalocação de memória eventualmente utilizada na representação do grafo;

c) Função que insete uma aresta no grafo: bool Graph::insert(const Edge&). A função retorna true se a inserção ocorrer com sucesso e false caso a aresta que se está tentando inserir já exista no grafo.

d) Função que retira uma aresta do grafo: bool Graph::remove(const Edge&). A função retorna true se a remoção ocorrer com sucesso e false caso a aresta que se está tentando remover não exista no grafo.

e) Funções para buscar o número de vértices e o número de arestas do grafo. Para que a função que retorna o número de arestas seja eficiente, é interessante que a classe mantenha um atributo interno que faça esta contagem. O atributo deve ser atualizado em todas as inserções e remoções de aresta que ocorrerem com sucesso;

f) Função para verificar a existência de uma aresta do grafo: bool Graph::edge(const Edge&) const . A função retornará true se a aresta estiver presente no grafo e false em caso contrário.

g) Função booleana para verificar se o grafo desenhado é completo.

h) Função para completar o grafo desenhado.

i) Função para realizar a busca em largura (Breadth First Search - BFS). Essa função deve receber o índice de um vértice e apresentar os índices dos vértices na ordem do caminhamento em largura a partir do vértice recebido como parâmetro. Este caminhamento deve ser feito apenas no componente do vértice inicial.

j) Função para realizar a busca em profundidade (Depth First Search – DFS). Essa função deve receber o índice de um vértice e apresentar os índices dos vértices na ordem do caminhamento em profundidade a partir do vértice recebido como parâmetro. Este caminhamento deve ser feito em todos os componentes do grafo.

k) Função para retornar o número de componentes conectados do grafo. A determinação do número de componentes conectados pode ser feita usando busca em profundidade no grafo.

l) Função para encontrar o menor caminho através do Algoritmo de Dijkstra. Essa função deverá receber o índice do vértice inicial e final e retornar os vértices contidos no menor caminho bem como o comprimento desse menor caminho.

Implementação:

Graph.h:

As funções públicas são:

Graph(const int n) : Constrói a classe Graph, construindo a matriz de adjacência de tamanho nxn.

~Graph(): Destrói a classe Graph, deletando todos os elementos da matriz de adjacência.
int **matriz: Matriz que recebe os valores 1 e 0 para construir a matriz de adjacência.
bool ExisteAresta(const int a, const int b: Função que compara vértices com índices da matriz para verificar existência de aresta.
bool InserirAresta(const int a, const int b): Função que insere uma aresta na matriz, se a mesma não existe.
bool RetirarAresta(const int a, const int b): Função que retira uma aresta da matriz se a mesma existe.
void BuscarArestas(): Procedimento que verifica quantidade de arestas e vértices da matriz.
void VerificaCompleto(): Procedimento que verifica se matriz está completa, ou seja, todos elementos igual a 1, menos os da diagonal principal.
void CompletaGrafo(): Procedimento que faz com que matriz fique completa, todos os elementos igual a 1, menos os da diagonal principal.
void BuscarLargura(int a, int b,int c, int v): Procedimento que busca conexões e distância entre o vértice escolhido e os vértices da matriz que estão conectados.
void BuscarLargura2(int a, int b,int c, int v): Procedimento complementar do BuscaLargura, e o procedimento que faz a recursão com este procedimento.
void BuscarProfundidade(): Procedimento que busca níveis de todos os vértices da matriz, suas distâncias e conexões.
void MenorCaminho(int a , int b): Procedimento que verifica distância entre dois vértices da matriz.
void ImprimeGrafo(): Procedimento que imprime a matriz de adjacência na tela.
void ListaGrafo(int a , int b, int c): Procedimento que usa lista para armazenar os vértices nos procedimentos de busca.
bool ListaVazia(): Procedimento que verifica se a lista do vértices está vazia.
void ImprimirLista(): Procedimento que imprime os vértices da listagrafo da busca em largura.
void ImprimirLista2(): Procedimento que imprime os vértices da listagrafo da busca em profundidade.
void ImprimirLista3(int a , int b): Procedimento que imprime os vertices da listagrafo do menor caminho.
void ConstroiLista(): Procedimento que constrói lista, já que a mesma é deletada nos procedimentos de impressão.
int ContaLinhasArquivo(FILE* entrada);
int num: Valor da quantidade de vértices inicial.
No* prox(No* no): Apontador da lista usada em ListaGrafo.
int vert: Valor do vértice armazenado na lista.
int dist: Valor do vértice de conexão
int dist2: Valor da distância do vértice com os demais vértices
int cont: Contador que é usado para comparações nos procedimentos de busca
int componentes: Valor dos componentes conectados na matriz.
As funções privadas são:
int vertices: Valor da quantidade de vertices da matriz.
int arestas: Valor da quantidade de arestas da matriz.
int Tam: Valor do tamanho da lista usada para armazenar os vértices nas buscas
No* Cabeca: Nó inicial da lista de listagrafo.
No* Ultimo: Nó final da lista de listagrafo.

Graph.cpp:

No arquivo Lista.cpp foram implementadas as funções declaradas em Graph.h possibilitando a manipulação da matriz de adjacência do grafo e a chamada destas funções no main.

Main.cpp:

No main.cpp foi implementado um programa teste que possibilitasse a manipulação da matriz de adjacência através de uma interface com o usuário.
Esse programa teste foi feito utilizando um menu, onde o o usuário recebe um arquivo de entrada com dados para criação de uma matriz de adjacência e após isso as funções switch/case, onde o usuário selecionava uma das opções indicadas na tela e aciona algumas das funções da lista.

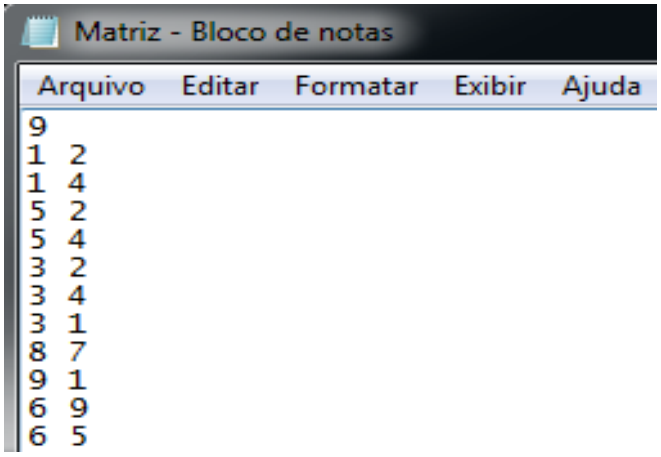

Figura 1: Arquivo de entrada, matriz 9x9. Os números em sequência determinam as aretas do grafo.

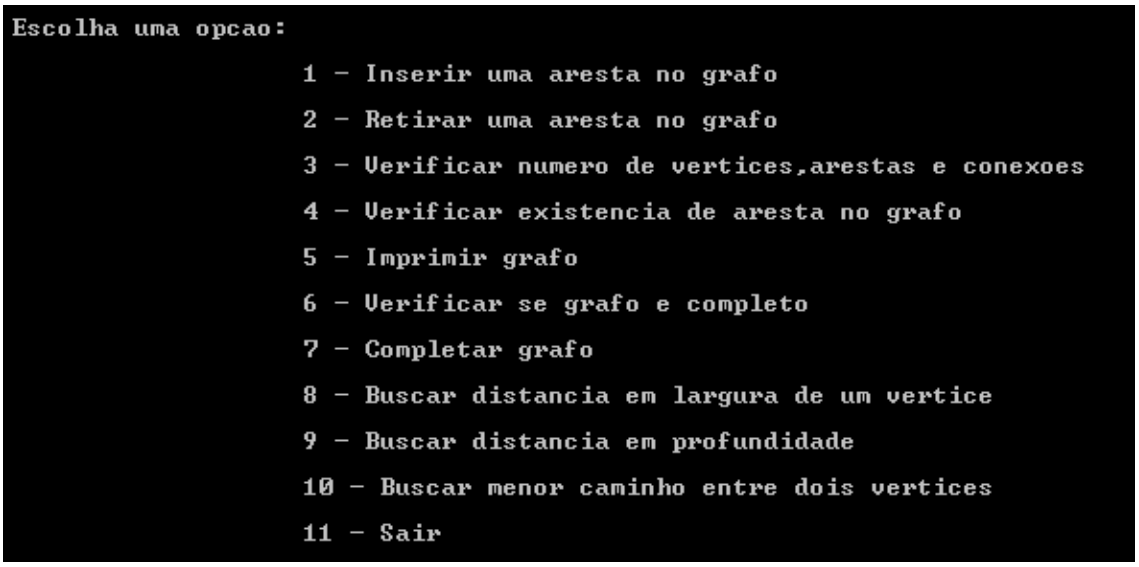

Figura 2: Interface inicial com o usuário.

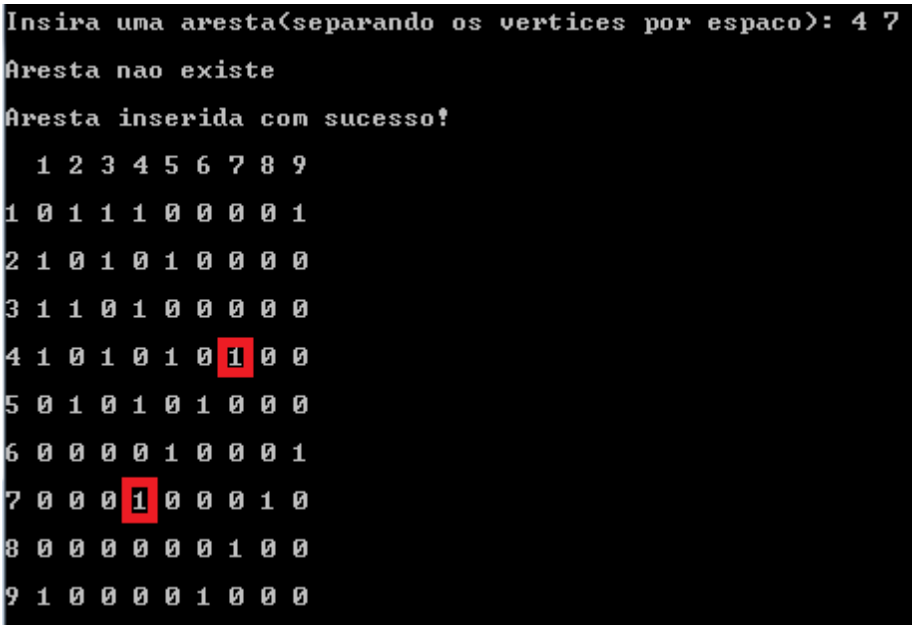

Figura 3: Opção1, foi inserida na matriz a aresta 4 e 7.

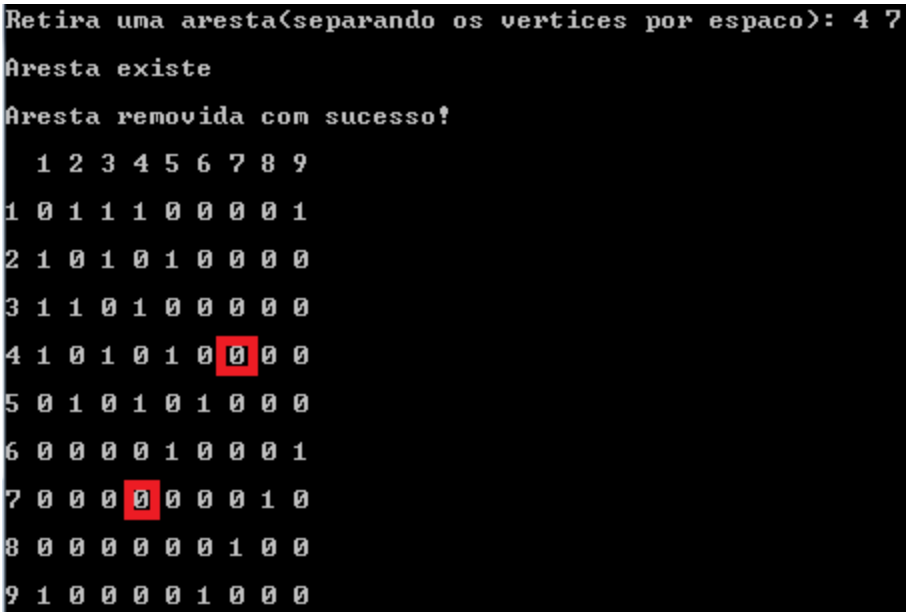

Figura 4: Opção 2, foi removida na matriz a aresta 4 e 7.

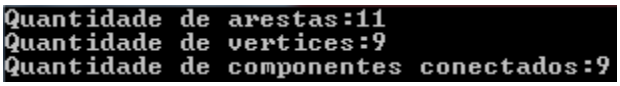

Figura 5: Opção 3, verificando os dados da matriz, quantidade de arestas, vértices e componentes conectados. Valores em conformidade com a matriz inserida no arquivo de entrada.

```
Insira a aresta a ser verificada(separando os vertices por espaco): 4 7
Aresta nao existe
```

Figura 6: Opção 4, verificando a existência da aresta 4 e 7 na matriz, aresta não existe.

```
Insira a aresta a ser verificada(separando os vertices por espaco): 8 7
Aresta existe
```

Figura 7: Opção 4, verificando a existência da aresta 8 e 7 na matriz, aresta existe.

```
  1 2 3 4 5 6 7 8 9
1 0 1 1 1 0 0 0 0 1
2 1 0 1 0 1 0 0 0 0
3 1 1 0 1 0 0 0 0 0
4 1 0 1 0 1 0 0 0 0
5 0 1 0 1 0 1 0 0 0
6 0 0 0 0 1 0 0 0 1
7 0 0 0 0 0 0 0 1 0
8 0 0 0 0 0 0 1 0 0
9 1 0 0 0 0 1 0 0 0
```

Figura 8: Opção 5, imprimindo a função de adjacência.

```
Grafo nao e completo!
```

Figura 9: Opção 6, verificando se o grafo está completo, conforme mostrado na figura anterior, o grafo não está completo.

```
  1 2 3 4 5 6 7 8 9
1 0 1 1 1 1 1 1 1 1
2 1 0 1 1 1 1 1 1 1
3 1 1 0 1 1 1 1 1 1
4 1 1 1 0 1 1 1 1 1
5 1 1 1 1 0 1 1 1 1
6 1 1 1 1 1 0 1 1 1
7 1 1 1 1 1 1 0 1 1
8 1 1 1 1 1 1 1 0 1
9 1 1 1 1 1 1 1 1 0
```

```
Grafo esta agora completo!
```

Figura 10: Opção 7, após completar o grafo, verifica-se que o mesmo se encontra completo.

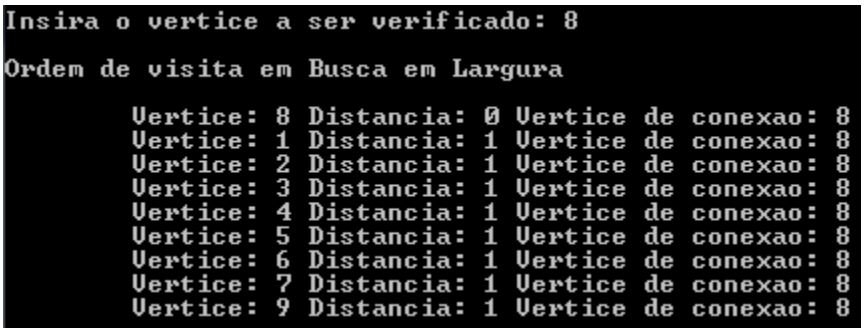

Figura 11: Opção 8, foi verificada a distância de largura para o vértice 8 em um grafo completo.

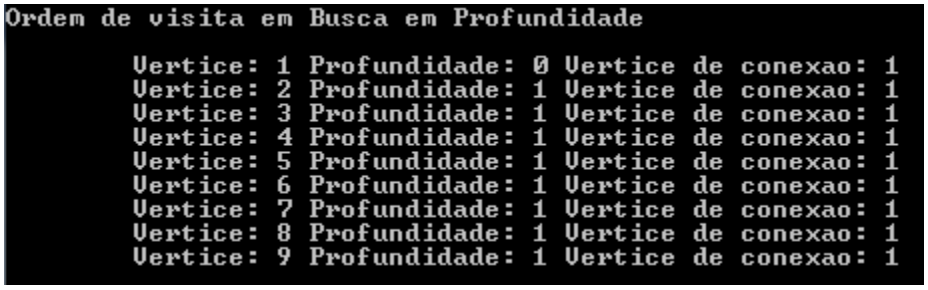

Figura 12: Opção 9, busca em profundidade para o vértice 1 do grafo.

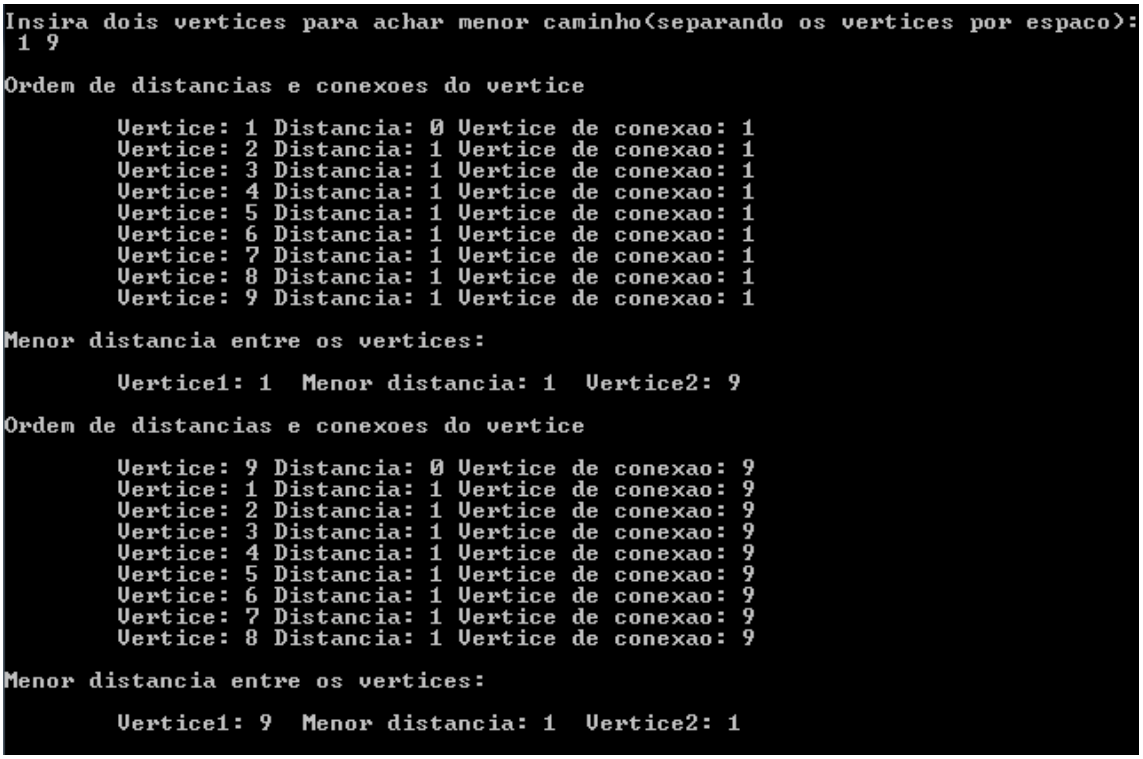

Figura 13: Opção 10, menor caminho entre as arestas 1 e 9, usando Dijkstra.

Conclusão:

Com a execução deste trabalho conclui-se que listas encadeadas são ótimas maneiras de organizar dados, e a utilização de grafos é uma ótima maneira de organizar

e relacionar dados de forma organizada. A implementação foi feita utilizando os conceitos aprendidos nas aulas de Programação e Desenvolvimento de Software II.

5. Bibliografia:

http://www.ime.usp.br/~pf/algoritmos/aulas/lista.html
http://www.inf.ufsc.br/~ine5384-hp/Capitulo4/EstruturasListaEncadeada.html
http://www.lbd.dcc.ufmg.br/colecoes/jai/2006/009.pdf