UNIVERSIDADE PRESBITERIANA MACKENZIE
Faculdade de Computação e Informática
Prof. Dr. Ivan Carlos Alcântara de Oliveira
Teoría dos Grafos

Relatório do Projeto – Parte 1

Nome do Integrante	TIA
Enzo Ryo Maeda Kanbayashi	41928369
Gustavo Aragão Guedes	32089678
Gustavo Fernandes Costa	32161093

Descrição:

A cidade de São Paulo, uma das maiores metrópoles do mundo, enfrenta diariamente desafios relacionados ao trânsito congestionado, aumento do consumo de combustível e poluição do ar. Em busca de soluções inovadoras para abordar esses problemas críticos, foi desenvolvida uma rede de distância mínima. Essa rede é um grafo cuidadosamente construído, conectando os locais-chave em São Paulo, como bairros residenciais, centros comerciais, escritórios e áreas industriais, de forma a otimizar o consumo de gasolina e reduzir a emissão de poluentes. Este grafo representa um modelo eficiente de transporte que beneficia tanto os habitantes da cidade quanto o meio ambiente.

Locais do estado de São Paulo que foram usados no grafo na ordem do arquivo grafo.txt:

1) Parque Ibirapuera
2) Catedral da Sé
3) Pinacoteca de São Paulo
4) Beco Do Batman
5) Estádio Morumbi
6) Bairro da Liberdade
7) Avenida Paulista
8) Estádio Allianz Parque
9) Neo Química Arena
10) Mercado Municipal
11) Universidade Presbiteriana Mackenzie
12) MASP
13) Museu do Futebol
14) Teatro Municipal
15) 25 de Março
16) Jardim Botânico
17) Edificio Italia
18) Copan
19) Estação da Sé
20) Estação Pinheiros
21) Estação da Luz
22) Ponte Estaiada
23) Viaduto do Chá

24) Viaduto Santa Ifigênia
25) Praça da República
26) Vale do Anhangabaú
27) Pateo do Colégio
28) Museu Paulista
29) Palácio dos Bandeirantes
30) Aquário de São Paulo
31) Zoológico de São Paulo
32) Monumento dos Bandeirantes
33) Shopping Iguatemi
34) Shopping JK
35) Shopping Morumbi
36) Shopping Ibirapuera
37) Shopping Aricanduva
38) Zoo Safari
39) Shopping Anália Franco
40) Shopping El Dorado
41) Shopping Tatuapé
42) Shopping Tamboré
43) Shopping Cidade de São Paulo
44) Farol Santander
45) Parque Ibirapuera
46) Parque Villa-Lobo
47) Praia Grande
48) Edifício Altino Arantes
49) Teatro Municipal de São Paulo
50) Museu Catavento

Como o arquivo grafo.txt funciona:

A primeira linha é o número de vértices.

A segunda linha é o número de arestas.

A terceira é a linha que contém as informações da aresta onde o primeiro número é o número do vértice inicial, o segundo o vértice final e o terceiro o peso da aresta ou distância entre as duas vértices.

UNIVERSIDADE PRESBITERIANA MACKENZIE
Faculdade de Computação e Informática
Prof. Dr. Ivan Carlos Alcântara de Oliveira
Teoría dos Grafos

Exemplo do grafo.txt:

```
50
150
1 2 5
1 3 8
1 4 6
1 5 10
1 6 5
1 7 9
1 8 9
1 9 22
1 10 8
1 11 7
```

Exemplo de uma fração do grafo feita no graph online

Tecnicamente, se pode ir de qualquer lugar para qualquer outro lugar, porém para facilitar a representação do grafo e não exceder o limite de 150 arestas nós montamos o programa para mostrar as distâncias em km de 50 lugares a partir de apenas 4 lugares de referência (Parque Ibirapuera, Catedral da Sé, Pinacoteca de São Paulo e Beco do Batman). Cada um desses 4 lugares faz ligação com os outros 50 lugares do grafo (A exceção é o 4º, o Beco do Batman, que apenas faz ligação com outros 6 para não exceder muito a quantidade de arestas).

O grafo acima feito no Graph Online mostra apenas as 5 primeiras ligações que os 4 lugares principais fazem com outros lugares e as suas respectivas distâncias. No grafo completo do programa há 10 vezes mais arestas e lugares que cada um dos 4 fazem ligações.

*O grafo 100% realístico teria 50 vértices com 50 arestas cada um, ou seja, 2500 arestas no total e portanto ficaria grande demais para a quantidade de arestas pedidas para o projeto) .

Interface do Menu do programa

```
Menu Grafo dos locais de SP:
1. Inserir vértice
2. Inserir aresta
3. Remover vértice
4. Remover aresta
5. Mostrar Grafo
6. Verificar conexividade
7. Mostrar conteudo do arquivo
8. Salvar no arquivo grafo2.txt
9. Sair
Escolha uma opção:
```

Opções

Entradas inválidas: O programa rejeita qualquer valor de entrada que não seja os que estão no menu.

```
Escolha uma opção: 0
Opção inválida. Tente novamente.
```

```
Escolha uma opção: 99
Opção inválida. Tente novamente.
```

1. **Inserir Vértice:** Pede para o usuário digitar um novo vértice para inserir no grafo. Não haverá inserção se o vértice já existir no grafo.

```
Escolha uma opção: 1
Digite o número do novo vértice: 13
O vértice 13 já existe.
```

```
Escolha uma opção: 1
Digite o número do novo vértice: 151
Vértice 151 inserido com sucesso.
```

2. **Inserir aresta:** Pede para o usuário digitar o número dos dois vértices que serão interligados pela nova aresta e pede para digitar o valor (nesse caso em km) dessa nova aresta.

```
Escolha uma opção: 2
Digite o vértice v: 13
Digite o vértice w: 14
Digite o número da aresta: 10
```

3. **Remover vértice:** Pede para o usuário digitar o número do vértice a ser removido, caso ele exista no grafo. O vértice removido não aparecerá no novo grafo gerado.

```
Escolha uma opção: 3
Digite o número do vértice a ser removido: 13
Vértice 13 removido com sucesso.
```

```
Escolha uma opção: 3
Digite o número do vértice a ser removido: 200
O vértice 200 não existe.
```

4. **Remover aresta:** Pede para o usuário digitar o número dos dois vértices cuja aresta que os interliga será removida, caso eles existam no grafo. A aresta removida não aparecerá no novo grafo gerado.

```
Escolha uma opção: 4
Digite o vértice de origem da aresta a ser removida: 14
Digite o vértice de destino da aresta a ser removida: 15
```

5. **Mostrar grafo:** Fornece ao usuário a impressão da iteração mais recente do grafo.

```
Escolha uma opção: 5
n: 50
m: 300

Adj[0,0]=- Adj[0,1]=5 Adj[0,2]=8 Adj[0,3]=6 Adj[0,4]=10 Adj[0,5]=5 Adj[0,6]=9 Adj[0,7]=9 Adj[0,8]
=22 Adj[0,9]=8 Adj[0,10]=7 Adj[0,11]=4 Adj[0,12]=6 Adj[0,13]=7 Adj[0,14]=8 Adj[0,15]=10 Adj[0,16]
=7 Adj[0,17]=7 Adj[0,18]=5 Adj[0,19]=8 Adj[0,20]=8 Adj[0,21]=7 Adj[0,22]=7 Adj[0,23]=7 Adj[0,24]=
8 Adj[0,25]=6 Adj[0,26]=6 Adj[0,27]=7 Adj[0,28]=9 Adj[0,29]=5 Adj[0,30]=14 Adj[0,31]=5 Adj[0,32]=
1 Adj[0,33]=3 Adj[0,34]=8 Adj[0,35]=4 Adj[0,36]=22 Adj[0,37]=13 Adj[0,38]=16 Adj[0,39]=6 Adj[0,40
]=13 Adj[0,41]=23 Adj[0,42]=4 Adj[0,43]=8 Adj[0,44]=1 Adj[0,45]=11 Adj[0,46]=72 Adj[0,47]=8 Adj[0
,48]=6 Adj[0,49]=9
Adj[1,0]=5 Adj[1,1]=- Adj[1,2]=3 Adj[1,3]=8 Adj[1,4]=13 Adj[1,5]=1 Adj[1,6]=3 Adj[1,7]=10 Adj[1,8
]=17 Adj[1,9]=2 Adj[1,10]=3 Adj[1,11]=4 Adj[1,12]=6 Adj[1,13]=2 Adj[1,14]=2 Adj[1,15]=15 Adj[1,16
]=3 Adj[1,17]=2 Adj[1,18]=1 Adj[1,19]=10 Adj[1,20]=3 Adj[1,21]=15 Adj[1,22]=2 Adj[1,23]=2 Adj[1,2
4]=3 Adj[1,25]=2 Adj[1,26]=1 Adj[1,27]=6 Adj[1,28]=12 Adj[1,29]=7 Adj[1,30]=16 Adj[1,31]=6 Adj[1,
32]=8 Adj[1,33]=15 Adj[1,34]=15 Adj[1,35]=10 Adj[1,36]=20 Adj[1,37]=27 Adj[1,38]=15 Adj[1,39]=3 A
dj[1,40]=3 Adj[1,41]=5 Adj[1,42]=4 Adj[1,43]=13 Adj[1,44]=6 Adj[1,45]=23 Adj[1,46]=87 Adj[1,47]=3
9 Adj[1,48]=5 Adj[1,49]=2
Adj[2,0]=8 Adj[2,1]=3 Adj[2,2]=- Adj[2,3]=8 Adj[2,4]=16 Adj[2,5]=3 Adj[2,6]=4 Adj[2,7]=6 Adj[2,8]
=21 Adj[2,9]=2 Adj[2,10]=3 Adj[2,11]=5 Adj[2,12]=5 Adj[2,13]=3 Adj[2,14]=3 Adj[2,15]=17 Adj[2,16]
=2 Adj[2,17]=3 Adj[2,18]=3 Adj[2,19]=9 Adj[2,20]=1 Adj[2,21]=17 Adj[2,22]=3 Adj[2,23]=1 Adj[2,24]
=2 Adj[2,25]=3 Adj[2,26]=4 Adj[2,27]=10 Adj[2,28]=13 Adj[2,29]=10 Adj[2,30]=11 Adj[2,31]=7 Adj[2,
32]=9 Adj[2,33]=16 Adj[2,34]=15 Adj[2,35]=12 Adj[2,36]=18 Adj[2,37]=20 Adj[2,38]=12 Adj[2,39]=9 A
dj[2,40]=10 Adj[2,41]=9 Adj[2,42]=5 Adj[2,43]=4 Adj[2,44]=8 Adj[2,45]=20 Adj[2,46]=80 Adj[2,47]=4
2 Adj[2,48]=3 Adj[2,49]=4
```

6. **Mostrar conectividade:** Fornece ao usuário o tipo de conectividade do grafo atual.

```
Escolha uma opção: 6
O grafo é fortemente conexo (C3).
```

7. **Mostrar conteúdo do arquivo:** Fornece ao usuário o conteúdo do arquivo do grafo atual em uma outra topologia.

```
Escolha uma opção: 7
50
150
1 2 5
1 3 8
1 4 6
1 5 10
1 6 5
1 7 9
1 8 9
1 9 22
1 10 8
1 11 7
1 12 4
1 13 6
1 14 7
1 15 8
1 16 10
1 17 7
1 18 7
1 19 5
1 20 8
1 21 8
1 22 7
1 23 7
1 24 7
1 25 8
1 26 6
1 27 6
1 28 7
1 29 9
1 30 5
1 31 14
1 32 5
1 33 1
1 34 3
1 35 8
1 36 4
1 37 22
1 38 13
1 39 16
1 40 6
1 41 13
1 42 23
1 43 4
1 44 8
1 45 1
1 46 11
1 47 72
1 48 8
1 49 6
1 50 9
2 3 3
2 4 8
2 5 13
2 6 1
2 7 3
2 8 10
2 9 17
2 10 2
2 11 3
2 12 4
2 13 6
2 14 2
2 15 2
2 16 15
2 17 3
2 18 2
2 19 1
2 20 10
2 21 3
2 22 15
2 23 2
2 24 2
2 25 3
2 26 2
2 27 1
2 28 6
2 29 12
2 30 7
2 31 16
2 32 6
2 33 8
2 34 15
2 35 15
2 36 10
2 37 20
2 38 27
2 39 15
2 40 3
2 41 3
2 42 5
2 43 4
2 44 13
2 45 6
2 46 23
2 47 87
2 48 39
2 49 5
2 50 2
3 4 8
3 5 16
3 6 3
3 7 4
3 8 6
3 9 21
3 10 2
3 11 3
3 12 5
3 13 5
3 14 3
3 15 3
3 16 17
3 17 2
3 18 3
3 19 3
3 20 9
3 21 1
3 22 17
3 23 3
3 24 1
3 25 2
3 26 3
3 27 4
3 28 10
3 29 13
3 30 10
3 31 11
3 32 7
3 33 9
3 34 16
3 35 15
3 36 12
3 37 18
3 38 20
3 39 12
3 40 9
3 41 10
3 42 9
3 43 5
3 44 4
3 45 8
3 46 20
3 47 80
3 48 42
3 49 3
3 50 4
4 5 16
4 6 3
4 7 4
4 8 6
4 9 21
4 10 3
```

8. **Salvar no arquivo grafo2.txt :** Salva a iteração atual do grafo em um arquivo .txt separado.

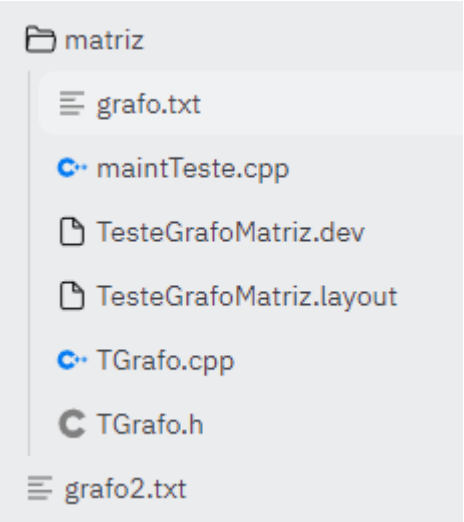
```
Escolha uma opção: 8
Informações do grafo salvas em grafo2.txt com sucesso.
```

9. **Sair :** Encerra o programa e libera o espaço que ele ocupou.

```
Escolha uma opção: 9
espa◆o liberado:
```

Códigos

Pasta "Matriz" :



grafo.txt : arquivo de texto original do grafo sem alterações,

matriz > grafo.txt

```
50
150
1 2 5
1 3 8
1 4 6
1 5 10
1 6 5
1 7 9
1 8 9
1 9 22
1 10 8
1 11 7
1 12 4
```

...

TGrafo.h :

matriz > TGrafo.h > ...

```
1  #include <iostream>
2  #include <fstream>
3  #include <cstdio>
4  #include <vector>
5  #include <queue>
6
7  /*
8  Implementa□o de uma Classe para grafos denominada TGrafo,
9  usando Matriz de Adjac□ncia
10 e m□todos para utiliza□o de um grafo dirigido.
11 */
12 #ifndef ___GRAFO_MATRIZ_ADJACENCIA___
13
14 #define ___GRAFO_MATRIZ_ADJACENCIA___
15
16 // defini□o de uma estrutura para armezanar um grafo
17 // Tamb□m seria poss□vel criar um arquivo grafo.h
18 // e fazer a inclus□o "#include <grafo.h>"
19 class TGrafo{
20   private:
21     int n; // quantidade de v□rtices
22     int m; // quantidade de arestas
23     int **adj; //matriz de adjac□ncia
24   public:
25     TGrafo( int n);
26     void insereA(int v, int w, int num);
27     void removeA(int v, int w);
28     void show();
29     void insereV(int v);
30     void removeV(int v);
31     void lerArquivo(const std::string& nomeArquivo);
32     void verificarTipoGrafo();
33     void salvarArquivo(const std::string& nomeArquivo);
34     ~TGrafo();
35 };
36
37 #endif
```

TGrafo.cpp :

matriz > TGrafo.cpp > ...

```cpp
#include <iostream>
#include <fstream>
#include <cstdio>
#include <vector>
#include <queue>
#include "TGrafo.h"

// Construtor do TGrafo, respons�vel por
// Criar a matriz de adjac�ncia v x v do Grafo
TGrafo::TGrafo( int n ){
    this->n = n;
    // No in�cio dos tempos n�o h� arestas
    this->m = 0;
    // aloca da matriz do TGrafo
    int **adjac = new int*[n];
    for(int i = 0; i < n; i++)
      adjac[i]= new int[n];
    adj = adjac;
    // Inicia a matriz com zeros
  for(int i = 0; i< n; i++)
    for(int j = 0; j< n; j++)
      adj[i][j]=0;
}

// Destructor, respons�vel por
// liberar a mem�ria alocada para a matriz
TGrafo::~TGrafo(){
  n = 0;
  m = 0;
  delete [] *adj;
  std::cout << "espa�o liberado";
}
```

```
33
34  // Insere uma aresta no Grafo tal que
35  // v é adjacente a w com o número da aresta especificado
36  void TGrafo::insereA(int v, int w, int numeroAresta) {
37      // Verifica se o número da aresta já foi usado
38      if (adj[v][w] == 0) {
39          adj[v][w] = numeroAresta;
40          m++; // Atualiza a quantidade de arestas
41      } else {
42          std::cout << "A aresta entre os vértices " << v << " e " << w << " já existe
    com o número " << adj[v][w] << "." << std::endl;
43      }
44  }
45
46  // remove uma aresta v->w do Grafo
47  void TGrafo::removeA(int v, int w){
48      // testa se temos a aresta
49      if(adj[v][w] == 1 ){
50          adj[v][w] = 0;
51          m--; // atualiza qtd arestas
52      }
53  }
54
55  // Apresenta o Grafo contendo
56  // n◆mero de v◆rtices, arestas
57  // e a matriz de adjac◆ncia obtida
58  // Apresenta o Grafo contendo
59  // número de vértices, arestas
60  // e a matriz de adjacência com números de aresta
61  void TGrafo::show( ) {
62      std::cout << "n: " << n << std::endl;
63      std::cout << "m: " << m << std::endl;
64      for (int i = 0; i < n; i++) {
65          std::cout << "\n";
66          for (int w = 0; w < n; w++) {
67              if (adj[i][w] > 0) {
68                  std::cout << "Adj[" << i << "," << w << "]=" << adj[i][w] << " ";
69              } else {
70                  std::cout << "Adj[" << i << "," << w << "]=- ";
71              }
72          }
73      }
74      std::cout << "\nfim da impressao do grafo." << std::endl;
75  }
76
77
```

```cpp
// Insere um vértice no Grafo
void TGrafo::insereV(int novoVertice) {
    // Verifica se o vértice já existe
    if (novoVertice >= 0 && novoVertice < n) {
        std::cout << "O vértice " << novoVertice << " já existe." << std::endl;
    } else if (novoVertice < 0) {
        std::cout << "O número do vértice deve ser não negativo." << std::endl;
    } else {
        // Cria uma nova matriz de adjacência com um vértice a mais
        int **novaMatriz = new int*[n + 1];
        for (int i = 0; i < n + 1; i++)
            novaMatriz[i] = new int[n + 1];

        // Copia os valores da matriz anterior para a nova matriz
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                novaMatriz[i][j] = adj[i][j];
            }
        }

        // Inicializa as novas entradas da matriz com zeros
        for (int i = 0; i <= n; i++) {
            novaMatriz[i][n] = 0;
            novaMatriz[n][i] = 0;
        }

        // Libera a matriz anterior e atualiza o ponteiro
        for (int i = 0; i < n; i++) {
            delete[] adj[i];
        }
        delete[] adj;

        adj = novaMatriz;
        n++; // Atualiza o número de vértices
        std::cout << "Vértice " << novoVertice << " inserido com sucesso." << std::endl;
    }
}

// Remove um vértice do Grafo
void TGrafo::removeV(int verticeRemover) {
    if (verticeRemover < 0 || verticeRemover >= n) {
        std::cout << "O vértice " << verticeRemover << " não existe." << std::endl;
    } else {
        // Cria uma nova matriz de adjacência com um vértice a menos
        int **novaMatriz = new int*[n - 1];
        for (int i = 0; i < n - 1; i++)
            novaMatriz[i] = new int[n - 1];

        int novaI = 0;
        int novaJ = 0;

        // Copia os valores da matriz anterior para a nova matriz,
        // ignorando a linha e coluna correspondentes ao vértice a ser removido
        for (int i = 0; i < n; i++) {
            if (i != verticeRemover) {
                novaJ = 0;
                for (int j = 0; j < n; j++) {
                    if (j != verticeRemover) {
                        novaMatriz[novaI][novaJ] = adj[i][j];
                        novaJ++;
                    }
                }
                novaI++;
            }
        }

```

```
        // Libera a matriz anterior e atualiza o ponteiro
        for (int i = 0; i < n; i++) {
            delete[] adj[i];
        }
        delete[] adj;

        adj = novaMatriz;
        n--; // Atualiza o número de vértices
        std::cout << "Vértice " << verticeRemover << " removido com sucesso." << std::endl;
    }
}

// Função para ler um arquivo no formato especificado e criar um grafo
void TGrafo::lerArquivo(const std::string& nomeArquivo) {
    FILE* arquivo = fopen(nomeArquivo.c_str(), "r"); // Abra o arquivo em modo de leitura

    if (arquivo == nullptr) {
        std::cerr << "Erro ao abrir o arquivo." << std::endl;
        return;
    }

    int numVertices, numArestas;
    fscanf(arquivo, "%d %d", &numVertices, &numArestas);

    this->n = numVertices;
    this->m = 0; // Inicializa o número de arestas como zero

    // Aloca a matriz do TGrafo
    int **adjac = new int*[numVertices];
    for (int i = 0; i < numVertices; i++)
        adjac[i] = new int[numVertices];

    // Inicializa a matriz com zeros
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            adjac[i][j] = 0;
        }
    }

    adj = adjac;

    // Lê as arestas e seus pesos do arquivo e insere nas duas direções
    int vertice1, vertice2, peso;
    for (int i = 0; i < numArestas; i++) {
        fscanf(arquivo, "%d %d %d", &vertice1, &vertice2, &peso);
        insereA(vertice1 - 1, vertice2 - 1, peso); // De A para B
        insereA(vertice2 - 1, vertice1 - 1, peso); // De B para A
    }

    fclose(arquivo); // Feche o arquivo após a leitura
    std::cout << "Arquivo lido com sucesso." << std::endl;
}
```

```cpp
void TGrafo::verificarTipoGrafo() {
    // Verifica se o grafo é desconexo (C0)
    if (m == 0) {
        std::cout << "O grafo é desconexo (C0)." << std::endl;
        return;
    }

    // Verifica se o grafo é fortemente conexo (C3)
    bool fortementeConexo = true;
    for (int v = 0; v < n; v++) {
        std::vector<bool> visitado(n, false);
        std::queue<int> fila;

        // Inicia a busca em largura a partir do vértice v
        fila.push(v);
        visitado[v] = true;

        // Verifica se é possível alcançar todos os outros vértices a partir de v
        while (!fila.empty()) {
            int u = fila.front();
            fila.pop();

            for (int w = 0; w < n; w++) {
                if (adj[u][w] && !visitado[w]) {
                    visitado[w] = true;
                    fila.push(w);
                }
            }
        }

        // Se algum vértice não foi alcançado, o grafo não é fortemente conexo (C3)
        for (int i = 0; i < n; i++) {
            if (!visitado[i]) {
                fortementeConexo = false;
                break;
            }
        }
        if (!fortementeConexo) break;
    }

    if (fortementeConexo) {
        std::cout << "O grafo é fortemente conexo (C3)." << std::endl;
        return;
    }

    // Verifica se o grafo é semi-fortemente conexo (C2) e não é semi-conexo (C1)
    bool semiFortementeConexo = true;
    for (int v = 0; v < n; v++) {
        std::vector<bool> visitado(n, false);
        std::queue<int> fila;

        // Inicia a busca em largura a partir do vértice v
        fila.push(v);
        visitado[v] = true;

        // Verifica se é possível alcançar todos os outros vértices a partir de v
        while (!fila.empty()) {
            int u = fila.front();
            fila.pop();

```

```
            for (int w = 0; w < n; w++) {
                if (adj[u][w] && !visitado[w]) {
                    visitado[w] = true;
                    fila.push(w);
                }
            }
        }

        // Se algum vértice não foi alcançado, o grafo é semi-fortemente conexo (C2)
        for (int i = 0; i < n; i++) {
            if (!visitado[i]) {
                semiFortementeConexo = false;
                break;
            }
        }

        // Verifica se o grafo é semi-conexo (C1)
        if (!semiFortementeConexo) {
            bool semiConexo = true;
            for (int i = 0; i < n; i++) {
                std::vector<bool> visitado(n, false);
                std::queue<int> fila;

                // Inicia a busca em largura a partir do vértice v
                fila.push(i);
                visitado[i] = true;

                // Verifica se é possível alcançar todos os outros vértices a partir de v
                while (!fila.empty()) {
                    int u = fila.front();
                    fila.pop();

                    for (int w = 0; w < n; w++) {
                        if (adj[u][w] && !visitado[w]) {
                            visitado[w] = true;
                            fila.push(w);
                        }
                    }
                }

                // Se algum vértice não foi alcançado, o grafo é semi-conexo (C1)
                for (int j = 0; j < n; j++) {
                    if (!visitado[j]) {
                        semiConexo = false;
                        break;
                    }
                }
                if (!semiConexo) break;
            }

            if (semiConexo) {
                std::cout << "O grafo é simplesmente-conexo (C1) e não é semi-fortemente conexo (C2)." << std::endl;
                return;
            }
        }
    }

    // Se nenhuma das condições acima for atendida, o grafo é desconexo (C0)
    std::cout << "O grafo é desconexo (C0)." << std::endl;
}

```

```cpp
void TGrafo::salvarArquivo(const std::string& nomeArquivo) {
    std::ofstream arquivo(nomeArquivo);

    if (!arquivo.is_open()) {
        std::cerr << "Erro ao criar o arquivo." << std::endl;
        return;
    }

    // Escreve o número de vértices e o número de arestas na primeira e segunda linhas
    arquivo << n << std::endl;
    arquivo << m << std::endl;

    // Escreve as arestas no formato vértice1 vértice2 peso
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (adj[i][j] > 0) {
                arquivo << i + 1 << " " << j + 1 << " " << adj[i][j] << std::endl;
            }
        }
    }

    arquivo.close();
    std::cout << "Informações do grafo salvas em " << nomeArquivo << " com sucesso."
<< std::endl;
}

```

mainTeste.cpp :

```cpp
#include <iostream>
#include <fstream>
#include <cstdio>
#include <string>
#include "TGrafo.h"
```

```cpp
void mostrarConteudoArquivo(const std::string& nomeArquivo) {
    // Abra o arquivo
    std::ifstream arquivo(nomeArquivo);

    if (!arquivo.is_open()) {
        std::cerr << "Erro ao abrir o arquivo." << std::endl;
        return;
    }

    std::string linha;

    // Leia e mostre o conteúdo linha por linha
    while (std::getline(arquivo, linha)) {
        std::cout << linha << std::endl;
    }

    // Feche o arquivo
    arquivo.close();
}
```

```cpp
int main() {
    int n;
    int v;
    int w;

    TGrafo grafo(0); // Crie um objeto TGrafo vazio

    // Leia o arquivo grafo.txt e preencha o grafo com as informações
    grafo.lerArquivo("matriz/grafo.txt");

    // Mostre o grafo
    grafo.show();

    int escolha;
    while (true) {
        std::cout << "\nMenu Grafo dos locais de SP:" << std::endl;
        std::cout << "1. Inserir vértice" << std::endl;
        std::cout << "2. Inserir aresta" << std::endl;
        std::cout << "3. Remover vértice" << std::endl;
        std::cout << "4. Remover aresta" << std::endl;
        std::cout << "5. Mostrar Grafo" << std::endl;
        std::cout << "6. Verificar conexividade" << std::endl;
        std::cout << "7. Mostrar conteudo do arquivo" << std::endl;
        std::cout << "8. Salvar no arquivo grafo2.txt" << std::endl;
        std::cout << "9. Sair" << std::endl;
        std::cout << "Escolha uma opção: ";
        std::cin >> escolha;

        switch (escolha) {
            case 1: // Inserir vértice
                int novoVertice;
                std::cout << "Digite o número do novo vértice: ";
                std::cin >> novoVertice;
                grafo.insereV(novoVertice);
                break;
            case 2: // Inserir aresta
                // Solicita ao usuário os vértices e o número da aresta
                int numeroAresta;
                std::cout << "Digite o vértice v: ";
                std::cin >> v;
                std::cout << "Digite o vértice w: ";
                std::cin >> w;
                std::cout << "Digite o número da aresta: ";
                std::cin >> numeroAresta;

                grafo.insereA(v, w, numeroAresta);
                break;
            case 3: // Remover vértice
                int verticeRemover;
                std::cout << "Digite o número do vértice a ser removido: ";
                std::cin >> verticeRemover;
                grafo.removeV(verticeRemover);
                break;
```

```
136             case 4: // Remover aresta
137                 int arestaV, arestaW;
138                 std::cout << "Digite o vértice de origem da aresta a ser removida: ";
139                 std::cin >> arestaV;
140                 std::cout << "Digite o vértice de destino da aresta a ser removida: ";
141                 std::cin >> arestaW;
142                 grafo.removeA(arestaV, arestaW);
143                 break;
144             case 5: // Mostrar Grafo
145                 grafo.show();
146                 break;
147             case 6: // Mostrar Grafo
148                 grafo.verificarTipoGrafo();
149                 break;
150             case 7:
151                 mostrarConteudoArquivo("matriz/grafo.txt");
152                 break;
153             case 8:
154                 grafo.salvarArquivo("grafo2.txt");
155                 break;
156             case 9: // Sair
157                 return 0;
158             default:
159                 std::cout << "Opção inválida. Tente novamente." << std::endl;
160                 break;
161         }
162     }
163
164     return 0;
165 }
```

grafo2.txt : Arquivo de texto da iteração mais recente do grafo (modificada).

grafo2.txt

```
50
300
1 2 5
1 3 8
1 4 6
1 5 10
1 6 5
1 7 9
1 8 9
1 9 22
1 10 8
1 11 7
1 12 4
```