Estrutura de Dados II – 2021
Trabalho Prático Final – Individual – Entrega até 14/02/2022
Documentação – Relatório

NOME: Daniel Henrique Serezane Pereira	**RA:** 201250047

1. Uso do programa

O programa deve ser usado por linha de comando, com as entradas de usuário sendo passadas em forma de argumentos ao comando. O padrão é o seguinte:

```
<nome do programa> <entrada> <forma> <origem> <saída>
```

sendo <entrada> o nome do arquivo de entrada e <saída> o nome do arquivo log a ser gerado. <forma> é a forma na qual os caminhos mínimos devem ser obtidos, sendo **1 - Dijkstra** e **2 - via ordenação topológica (GAD)**. <origem> é o vértice de origem.

Por exemplo, para processar o arquivo “grafo.txt” com o algoritmo de Dijkstra a partir do vértice 1, e depositar o log de processamento no arquivo “log_grafo.txt”, deve-se executar, usando o executável anexado ao trabalho:

```
./dssp grafo.txt 1 1 log_grafo.txt
```

(comando de PowerShell).

Para compilar o programa, basta utilizar o CMake ou IDE compatível, lembrando que o CMakeLists.txt incluído em entrega foi configurado para Windows. Alternativamente, pode-se compilar o projeto com o gcc (MinGW se Windows), sendo necessário C++20.

```
gcc main.cpp Graph.cpp -o dssp
```

O programa deve funcionar corretamente em sistemas Unix-like (Linux, macOS), porém não foi testado nestes ambientes.

2. Implementação

2.1 Softwares utilizados

Foi utilizado o IDE CLion 2021.3.3 para desenvolvimento do projeto, na linguagem C++ (versão 20), utilizando o CMake para organizar a compilação e o gcc via MinGW (versão embutida no CLion) como compilador. O programa foi desenvolvido e testado no sistema Windows 10 versão 21H1.

2.2 Status dos processamentos

Foram implementados ambos os processamentos requisitados, isto é:

- Algoritmo de Dijkstra
- Usando ordenação topológica

Ambos estão funcionando integralmente.

Contudo, pressupõe-se bom uso do programa, isto é, erros podem ocorrer caso processamentos incorretos sejam realizados. Por exemplo, não foi definido um comportamento caso tente-se processar um grafo cíclico via ordenação topológica, ou um grafo com pesos negativos com Dijkstra.

Também não é verificada a corretude dos arquivos. Isto é, pressupõe-se que o arquivo de entrada está devidamente organizado, como definido em enunciado.

2.3 Lista de adjacências

A lista de adjacências foi implementada a partir de um mapa não-ordenado de inteiros, com cada chave indicando um outro inteiro. A chave é equivalente ao identificador de um vértice dentro do grafo, que por sua vez também tem seus vértices definidos em formato de mapa (chave inteira e valor ponteiro para vértice). O valor indicado pela chave da lista de adjacências é o peso da adjacência. Segue um esboço para clarificar, utilizando o mesmo grafo visto no enunciado do trabalho. Neste exemplo, ocultam-se os demais atributos dos vértices (distância, peso e cor) e do grafo.

```
Grafo {
    ... atributos
    V :{
        1: ptr(1)
        2: ptr(2)
        3: ptr(3)
        4: ptr(4)
        5: ptr(5)
    }
}
```

```
Vértice 3 {
    ... atributos
    adj: {
        2: 3
        4: 9
        5: 2
    }
}
```

Simplificadamente

```
G (
    (1, (2, 10), (3, 5)),
    (2, (3, 2), (4,1)),
    (3, (2, 3), (4, 9)),
    (4, (5, 6)),
    (5, (1, 7), (4,4))
)
```

3. Funções

```
explicit Graph(std::istream &f);
```

Classe para criação dos grafos. Recebe um arquivo.

```
void Graph::dijkstra(int s);
```

Determina os caminhos mínimos do grafo via algoritmo de Dijkstra, a partir da origem s.

```
void Graph::dag_shortest_paths(int s);
```

Determina os caminhos mínimos do grafo via ordenação topológica, a partir da origem s.

```
std::string Graph::get_log();
```

Retorna o log de processamento do grafo.

```
void Graph::addEdge(int u, int v, int w);
```

Adiciona o vértice u ao grafo, definindo v como seu adjacente (u é predecessor de v), com a aresta que os liga tendo peso w.

```
void Graph::initialize_single_source(int s);
```

Inicializa os vértices do grafo usando s como origem.

```
int Graph::extract_min(std::unordered_map<int, Vertex*> *queue);
```

Retira do mapa queue (usado no lugar da fila Q) o vértice com menor distância à origem. Retorna a chave do vértice removido (para imprimir no log).

```
void Graph::relax(Vertex *u, int uid, Vertex *v, int vid, int w);
```

Relaxa os vértices u e v, com peso w. uid e vid são as chaves destes vértices no grafo, passadas à esta função apenas para fins de impressão (gerar o log).

```
std::vector<int> Graph::dfs();
```

Faz busca em profundidade no grafo, a partir do último vértice inserido. O vetor de inteiros retornado corresponde às chaves dos vértices no grafo, em ordem de finalização (primeiro elemento do vetor foi o primeiro vértice a ser finalizado).

```
void Graph::dfs_visit(int uid, Vertex *u, std::vector<int> *f);
```

Vista o vértice u, adicionando-o a f assim que for finalizado (para isso é necessário sua chave, uid).

```
std::vector<int> Graph::topological_sort();
```

Realiza ordenação topológica no grafo, retornando as chaves dos vértices nesta ordem (o grafo em si não é alterado).

```
void Graph::log_graph();
```

Adiciona ao log de processamento uma visão geral do estado atual do grafo.

```
void Graph::log_q(std::unordered_map<int, Vertex*> *queue);
```

Adiciona ao log de processamento o estado atual da fila Q.

Todas as funções foram implementadas usando como referência o livro CLRS.