Universidade Estadual do Ceará – UECE – Campus Itaperi
Mestrado Acadêmico em Ciências da Computação – Turma 2013
Disciplina de Projeto e Análise de Algoritmos
Prof. Leonardo Sampaio
Alunos: Nécio de Lima Veras e Robert Marinho da Rocha Júnior

Relatório de Implementação do Trabalho 1 de Projeto e Análise de Algoritmos

1 Introdução

O objetivo deste relatório é documentar a implementação de alguns algoritmos que operam sobre uma estrutura de dados baseada na Teoria dos Grafos. Estes foram projetados e criados de forma a atender os requisitos elencados na descrição textual do trabalho definido pelo prof. Leonardo durante a disciplina de Projeto e Análise de Algoritmos do corrente ano e curso.

Foi definido anteriormente que um **grafo** consiste em um conjunto de elementos denominados **vértices** e um conjunto de relações entre pares de vértices, que são denominados **arestas**. Por exemplo, poderíamos ter o conjunto de vértices V = {1, 2, 3, 4, 5}, e o conjunto de arestas E = {{1, 5}, {1, 2}, {2, 3}, {2, 5}, {3, 5}, {5, 4}, {3, 4}}. Em geral, usamos a notação G = (V, E) para denotar o grafo G cujo conjunto de vértices e V e conjunto de arestas E. Com isso, nossa implementação possui uma estrutura (detalhada à frente) capaz de armazenar um grafo com quaisquer quantidades de vértices com suas respectivas arestas e os reconhece a partir da leitura de um arquivo de texto cujo formato respeita a estrutura abaixo:

```
n
a       b       c
…
```

onde:

n = número de vértices
a e **b** = são pares de vértices que representam a existência de uma aresta entre eles
c = peso desta aresta

Os algoritmos codificados que operam sob a estrutura supracitada são:

- Busca em largura (bfs - *breadth-first search*);
- Busca em profundidade (dfs - *depth-first search*);
- Cálculo de uma árvore geradora mínima (algoritmos de Prim); e
- Cálculo do caminho mínimo (Algoritmo de Bellman-Ford).

Para a execução das rotinas relacionadas, nossa solução é capaz de reconhecer comandos através da própria chamada ao programa executável, recebendo parâmetros que especificam o(s) comando(s) via console em modo texto do sistema operacional. Assim, é possível, inclusive, executar todas as rotinas para uma mesma instância de um grafo a partir de um único comando ou, ainda, realizar as mesmas tarefas carregando diversas instâncias consecutivas. A seguir o detalhamento do funcionamento da codificação e seus recursos.

2 Funcionamento e recursos

Esta seção visa detalhar o correto funcionamento e utilização das implementações realizadas, bem como, os casos de uso (recursos) englobados no programa.

Como forma de ilustrar os casos **mais gerais** de uso da aplicação criamos um diagrama de casos de uso.

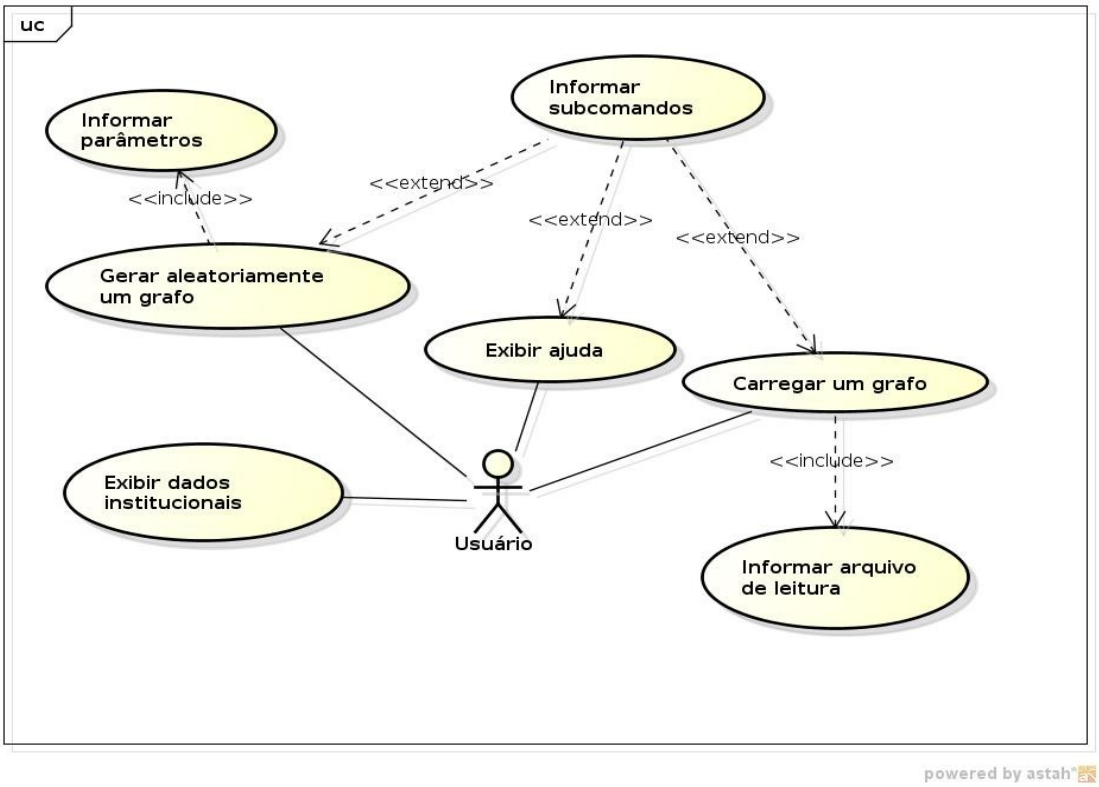

Figura 1: diagrama de casos de uso

Percebe-se que o usuário poderá executar quatro **comandos** básicos definidos sob a forma de um caso de uso. O comando de carregar um grafo requer obrigatoriamente um parâmetro que informará o nome do arquivo a ser lido, enquanto que o comando gerar aleatoriamente um grafo requer também obrigatoriamente alguns parâmetros generalizados pelo caso de uso "Informar parâmetros". Os **subcomandos** estão diretamente relacionados aos três comandos (exibir ajuda; gerar um grafo; gerar aleatoriamente um grafo) básicos e representam parâmetros ligados aos **recursos** oferecidos pelo programa. O quadro abaixo traz uma breve descrição dos quatro comandos básicos.

Quadro 1: mapeamento descritivo entre comando e caso de uso

Comando	Caso de uso	Descrição
read	Carregar um grafo	Serve para informar ao programa que um arquivo texto será informado e nele existirá um grafo. O comando contém um parâmetro que é o nome do arquivo a ser carregado.
gerar	Gerar aleatoriamente um grafo	Serve para solicitar ao programa que um grafo seja gerado aleatoriamente. O comando recebe três parâmetros numéricos, são eles: <número_máximo_de_vértices> <número_máximo_de_arestas> <valor_máximo_do_peso_de_uma_aresta>

sobre	Exibir dados institucionais	Serve para que o programa exiba as informações de cabeçalho do programa mostrando as informações sobre instituição, curso, disciplina e alunos.
help	Exibir ajuda	Serve para pedir um auxílio ao programa em relação à utilização do mesmo. Este comando deve ser usado com outros subcomandos especificados à frente.

2.1 Os subcomandos reconhecidos

O programa é capaz de reconhecer **sete** subcomandos que podem ser utilizados em conjunto com os comandos básicos ***read***, ***gerar*** e ***help***, são eles: mza, pesos, bfs, dfs, mst, sp e exporta. O quadro abaixo organiza as informações de descrição e parâmetros que cada subcomando possui.

Quadro 2: detalhamento dos subcomandos reconhecidos

Subcomando	Parâmetro(s)	Descrição
mza	<sentido>	É usado para exibir o resultado da leitura promovida por um comando básico. A visualização se dá por uma matriz de adjacência onde os vértices são representados por uma matriz de 0 e 1 (lógica) indicando a presença ou não de uma aresta entre os vértices envolvidos. O parâmetro sentido pode receber os valores 0, 1 e 2, onde: 0 = grafo direcionado. Sentido de IDA 1 = grafo direcionado. Sentido de VOLTA 2 = grafo não direcionado.
pesos		É usado para exibir os vértices do grafo com seus respectivos pesos.
bfs	<x>	É usado para executar a técnica de Busca em Largura, do inglês *Breadth-first search*. O Parâmetro x indica o vértice inicial da busca. Saída: será exibido uma sequência de vértices visitados, iniciando no vértice x.
dfs	<x>	É usado para executar a técnica de Busca em Profundidade, do inglês *Depth-first search.* O Parâmetro x indica o vértice inicial da busca. Saída: será exibido uma sequência de vértices visitados, iniciando no vértice x.
mst	<arquivo_saida>	É usado para calcular e gerar a árvore geradora mínima (*Minimal Spanning Tree*) por meio do algoritmo Prim. O parâmetro: arquivo_saida indica o nome do arquivo que será gerado pelo algoritmo com a árvore geradora mínima. Este arquivo conterá o mesmo formato de arquivo definido no início deste relatório.
sp	<s> <t>	É usado para executar a técnica do caminho mínimo (*short path*) entre dois vértices por meio do algoritmo de Bellman Ford. O parâmetro s indica o vértice inicial enquanto que o t indica o vértice final. Saída : d x1 x2 … xp, onde: d = valor do caminho mínimo entre s e t

		cada x representa os vértices que formam o caminho mínimo entre x1 e xp. Vale lembrar que x1 = s e xp = t.
exporta	<arquivo_saida>	É usado para exportar o grafo atual em um arquivo de saída. O parâmetro arquivo_saida indica o nome do arquivo que o programa irá gravar os dados do grafo. Este arquivo conterá o mesmo formato de arquivo definido no início deste relatório.

2.1 Makefile

Objetivando facilitar o processo de compilação criamos um arquivo de configuração para o programa utilitário make. Make é um programa que auxilia alguns processos do desenvolvimento de software, entre eles a compilação de bibliotecas e programas. Os comandos são especificados em um arquivo de configuração denominado de Makefile. Dessa forma criamos um *makefile* com os seguintes comandos: **compila**, **executa** e **limpa**.

O comando **compila** otimiza as etapas antecessoras à compilação verificando a existência dos arquivos necessários para o processo e o finaliza passando-os como parâmetro para o compilador g++ (GNU Compiler para C e C++). Já o comando **executa** garante uma chamada prévia ao comando compila e informa ao usuário a maneira correta de executar o programa. Por fim, o comando **limpa** serve apenas para excluir os programas-objeto gerados por compilações anteriores.

2.2 Execução

Para uma boa utilização do programa devem ser chamados e repassados de maneira correta os comandos, subcomandos e parâmetros conforme a sintaxe de uso de cada um deles. Abaixo reunimos a título de exemplificação um conjunto de formas de utilização do programa considerando que o programa executável foi gerado a partir do uso do Makefile:

a) execução para leitura e visualização de um grafo contido no arquivo “inst-10”.

- Usando a visualização de vértices, arestas e pesos: **./grafo.o read inst-10 pesos**
- Usando a visualização com uma matriz de adjacência não direcionada:
  ./grafo.o read inst-10 mza 2

b) execução para geração aleatória de um grafo com no máximo 50 vértices, 60 arestas e pesos com valores máximo de 10: **./grafo gerar 50 60 10**

c) para gerar o mesmo grafo anterior e exportar em um arquivo texto chamado de gsaida.g:
./grafo gerar 50 60 10 exporta gsaida.g

d) para ler um grafo do arquivo inst-15 e descobrir o caminho mínimo entre os vértices 1 e 4:
./grafo read inst-15 sp 1 4

e) para ler um grafo do arquivo inst-15, visualizar a matriz de adjacência não direcionada e realizar as buscas em largura e profundidade iniciando no vértice 1: **./grafo read inst-15 mza 2 bfs 1 dfs 1**

f) para ler um grafo do arquivo inst-30, visualizar a matriz de adjacência não direcionada, realizar as buscas em largura e profundidade iniciando no vértice 1, calcular a árvore gerados mínima e gravá-la no arquivo grafo_saida.g, calcular e mostrar a caminho mínimo entre os vértices 1 e 20:
./grafo read inst-30 mza 2 bfs 1 dfs 1 mst grafo_saida.s sp 1 20

3 Especificações técnicas

Esta seção visa explicar tecnicamente a implementação criada sob o ponto de vista das estruturas internas da solução. Iniciamos demonstrando os componentes idealizados e, posteriormente, as classes e estruturas definidas para a manipulação do problema.

3.1 Componentes

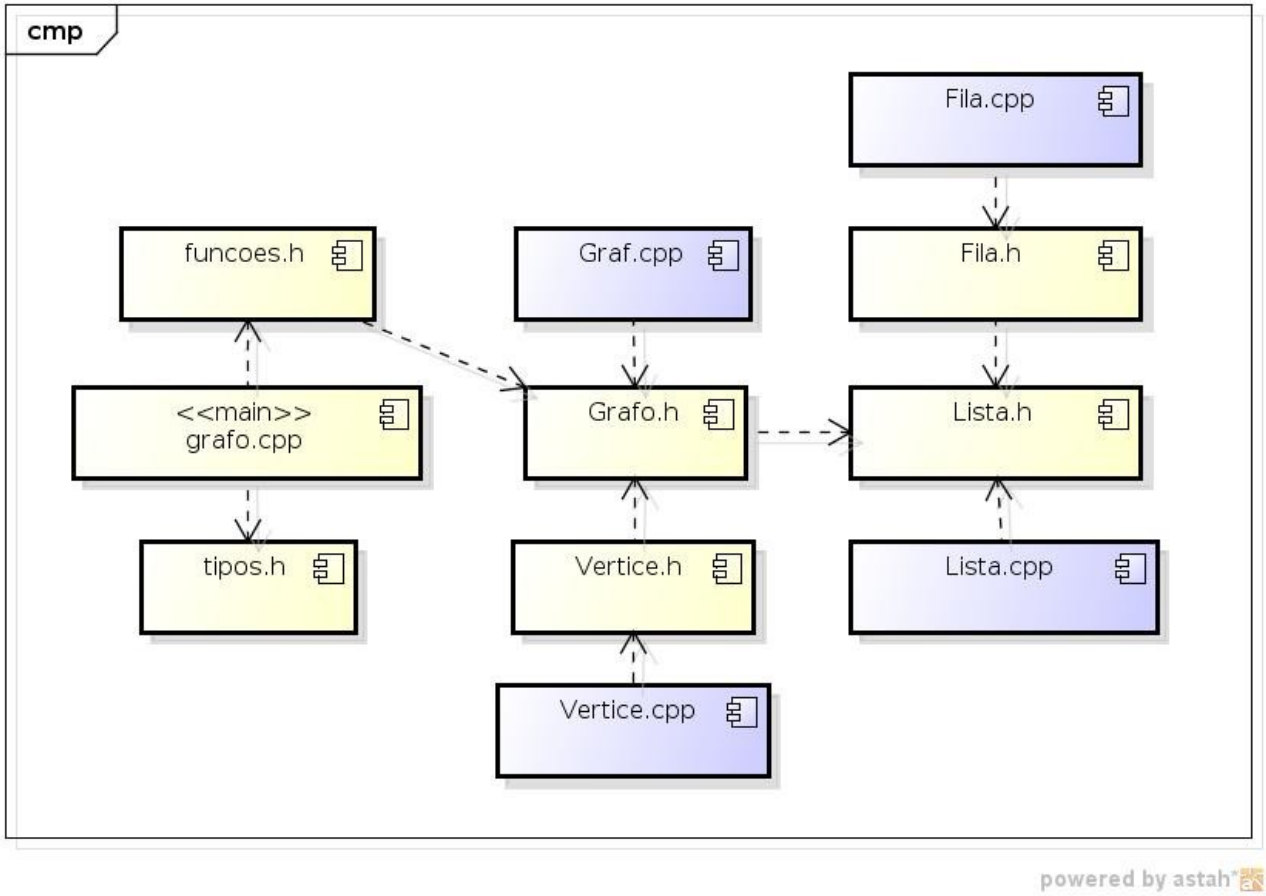

Figura 2: diagrama de componentes

Criamos uma biblioteca de funções (*funcoes.h*), outra biblioteca com alguns tipos de dados e constantes para facilitar a programação da solução (*tipos.h*) e optamos pela codificação usando o paradigma da Orientação a Objetos concretizando quatro classes que norteiam a resolução do problema (*Grafo.h*, *Vertice.h*, *Fila.h* e *Lista.h*). O diagrama mostra as relações de dependências e as relações entre as classes e suas respectivas implementações (representadas no diagrama pela cor azul). É possível perceber que o bloco principal (*grafo.cpp*) requer uma dependência entre as bibliotecas e as funções dependem da classe Grafo para realizar suas tarefas. Dessa forma, procuramos nesse diagrama evidenciar as interações entre os componentes existentes na nossa implementação.

3.2 Classes e tipos definidos

Objetivando visualizar os atributos internos e serviços oferecidos pelas classes idealizadas, modelamos o diagrama de classe conforme a Linguagem de Modelagem Unificada. Incluímos também no diagrama, para efeitos de percepção e entendimento, algumas estruturas adicionais tais como: Taresta e ponteiros. A primeira representa uma aresta que parte de um vértice e possui

adjacência com outro vértice. A segunda pode ser percebida em relações com estruturas não elementares como Vertice e Grafo e, sempre veem com o sufixo “p” para indicar sua natureza de ***ponteiro***, por exemplo Pvertice e Pgrafo.

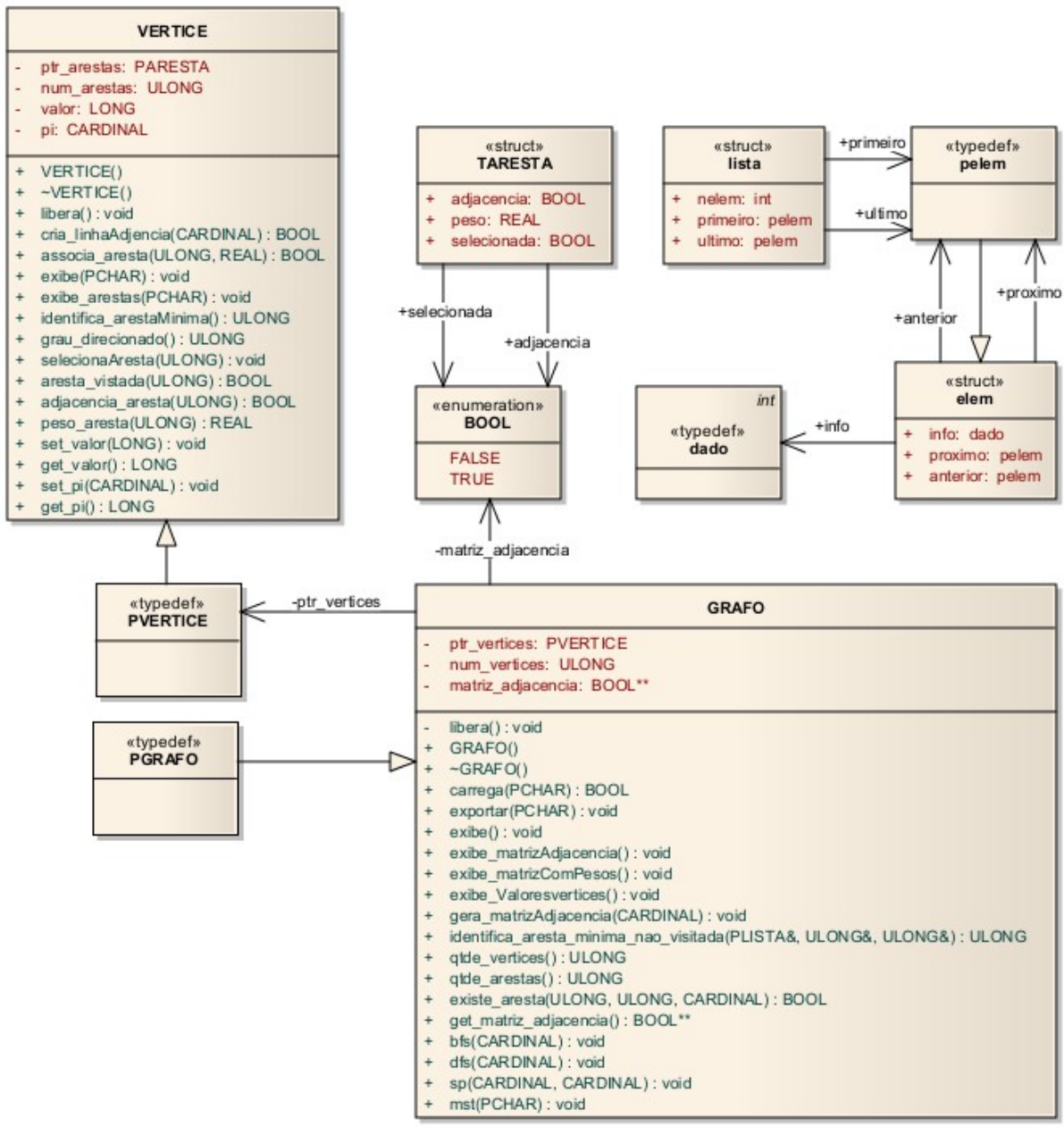

Figura 3: diagrama de classes

3.3 Algoritmos

O trabalho exigiu a codificação de quatro algoritmos sob a estrutura dos grafos, a saber: buscas em largura (bfs) e profundidade (dfs), cálculos e geração de árvore geradora mínima (mst) e caminho mínimo (sp). O quadro a seguir relaciona alguns detalhes sobre a implementação de cada um dos quatro algoritmos.

Quadro 3: detalhamento das implementações dos quatro algoritmos solicitados

Algoritmo	Observações sobre a implementação
bfs	A ideia partiu em inicialmente armazenar em uma estrutura os vértices não visitados. Depois criamos um pilha para empilhar todos os vértices adjacentes ao vértice inicial, chamamos isso de primeiro nível e desempilhamos o próprio vértice inicial armazenando-o em uma lista que determinará mais tarde a sequência dos vértices visitados. Em seguida, tomamos como novo vértice inicial um vértice adjacente ao anterior e repetimos o mesmo processo até que a pilha estivesse completamente vazia.
dfs	Esse algoritmo de busca objetiva explorar um grafo por meio das adjacências consecutivas até que não exista mais nenhum vértice "profundo" a ser explorado. Quando isso acontecer deve-se retornar e explorar as arestas adjacentes não visitadas. Partindo disso, nosso algoritmo começa marcando todos os vértices como não visitados. Depois ele marca o primeiro como visitado e inicia a exploração de "ida" em busca das arestas mais profundas. Em um determinado ponte ele poderá encontrar um vértice final e então nesse ponto deve ser retornado pelo mesmo caminho de ida em busca de novas arestas.
mst	Nossa implementação para o cálculo e geração de árvore mínima foi espelhada pelo algoritmo de **Prim**. Ela se dá criando inicialmente uma lista que servirá para inserir os vértices selecionados. Armazenamos em uma outra lista as arestas destes vértices que nos servirá para montar a árvore ao final do algoritmo. O núcleo do algoritmo está em um método da classe Grafo chamado de "identifica_aresta_minima_nao_visitada". Este método recebe uma lista de vértices e sua função é descobrir, dentre esses vértices, qual aresta mínima. Na medida que vamos selecionando novos vértices a execução da função vai tornando-se mais importante para a reolução do problema em si e com ela nossa implementação mostrou-se eficiente nos testes que realizamos.
sp	Para a implementação do cálculo e geração do caminho mínimo utilizamos o algoritmo de **Bellman-Ford**. Inicialmente inicializamos todos os valores dos vértices com a constante INFINITO e atribuímos como antecessor de cada um o próprio vértice. Depois zeramos o valor do vértice inicial e, para cada aresta de cada vértice aplicamos a técnica do relaxamento. Após isso, verificamos se existe algum ciclo negativo e caso seja verdade isso, o programa aborda o algoritmo mostrando que há presença de um ciclo negativo. Feito isso, nosso código passa a trabalhar na geração do arquivo de saída com os vértices, arestas e pesos.

4 Resultados e testes

Esta seção visa demonstrar alguns resultados obtidos a partir de instâncias geradas manualmente. Para tanto usamos instâncias com cinco, sete, nove e treze vértices e apresentaremos em seguida os resultados das execuções de cada um dos quatro algoritmos nas instâncias citadas.

4.1 Instância com cinco vértices

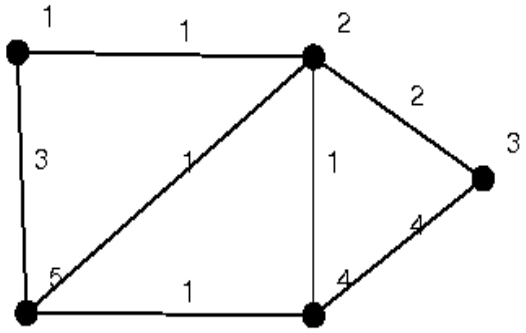

***Figura 4**: visualização do grafo com cinco vértices*

Quadro 4: resultados para uma instância com cinco vértices

Algoritmo	Parâmetro(s)	Saída
bfs	1	1 2 5 3 4
dfs	1	1 2 3 4 5
mst	<arquivo>	5 1 2 1.0 2 4 1.0 2 5 1.0 2 3 2.0
sp	1 5	2 1 2 5

4.2 Instância com sete vértices

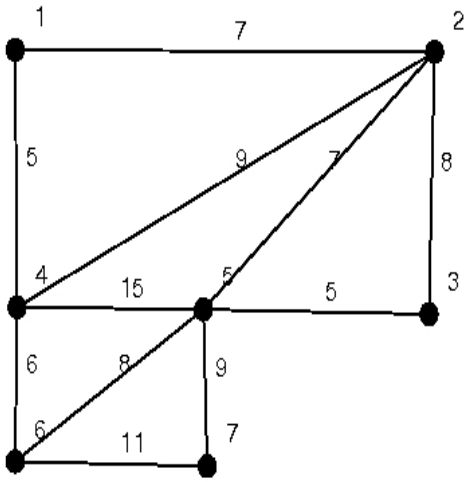

***Figura 5**: visualização do grafo com sete vértices*

Quadro 5: resultados para uma instância com sete vértices

Algoritmo	Parâmetro(s)	Saída
bfs	1	1 2 4 3 5 6 7
dfs	1	1 2 3 5 4 6 7

mst	<arquivo>	7 1 4 5.0 4 6 6.0 1 2 7.0 2 5 7.0 5 3 5.0 5 7 9.0
sp	1 7	22 1 4 6 7

4.3 Instância com nove vértices

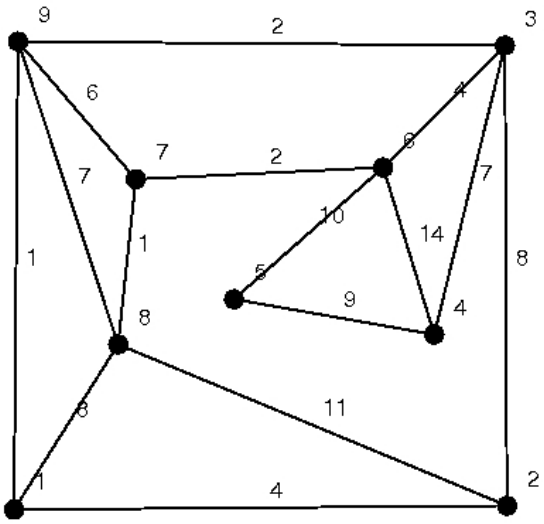

Figura 6*: visualização do grafo com nove vértices*

Quadro 6: resultados para uma instância com nove vértices

Algoritmo	Parâmetro(s)	Saída
bfs	1	1 2 8 3 7 9 4 6 5
dfs	1	1 2 3 4 5 6 7 8 9
mst	<arquivo>	9 1 2 4.0 1 8 8.0 8 7 1.0 7 6 2.0 6 3 4.0 3 9 2.0 3 4 7.0 4 5 9.0
sp	1 9	14 1 2 3 9

4.4 Instância com treze vértices

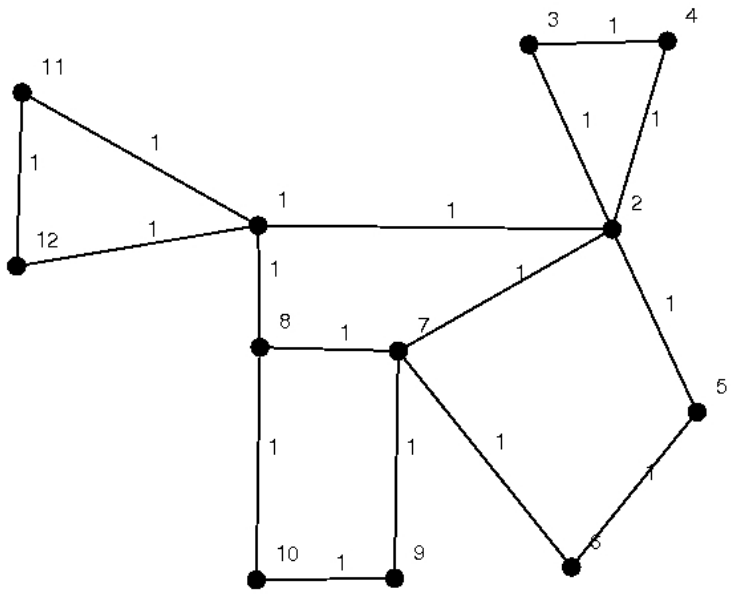

***Figura** 7: visualização do grafo com treze vértices*

Quadro 6: resultados para uma instância com treze vértices

Algoritmo	Parâmetro(s)	Saída
bfs	1	1 2 8 11 12 3 4 5 7 10 6 9
dfs	1	1 2 3 4 5 6 7 8 10 9 11 12
mst	<arquivo>	13 1 2 1.0 1 8 1.0 1 11 1.0 1 12 1.0 2 3 1.0 2 4 1.0 2 5 1.0 2 7 1.0 8 10 1.0 5 6 1.0 7 9 1.0 7 9 1.0
sp	1 9	Não há caminho mínimo, pois o grafo não é ponderado. Com isso, o algoritmo retorna: 2147483647 1 onde 2147483647 é representado pela constante INFINITO.

5 Conclusões e sugestões de melhorias

Este relatório buscou documentar a implementação de alguns algoritmos que operam sobre uma estrutura de dados baseada na Teoria dos Grafos, onde estes foram projetados e criados de forma a atender os requisitos elencados na descrição textual do trabalho supracitado. Percebemos com o desenvolvimento do mesmo a importância e aplicabilidade da teoria dos Grafos no âmbito do mundo real e visualizamos o quanto é importante o uso dos recursos computacionais para a efetivação da contribuição da Teoria dos Grafos. Acreditamos que nossas implementações

obtiveram êxito no que se refere à proposta do trabalho, no entanto, somos conhecedores que ainda é possível realizar algumas melhorias no código assim como o incremento de diversas outras funcionalidades oriundas da própria Teoria dos Grafos. Relacionamos abaixo algumas sugestões para continuidade e melhorias no código.

- Sugestões de melhorias:
    - Nos algoritmos de busca usar efetivamente o atributo PI dos vértices para marcar as visitação aos vértices. Usamos nessa versão um vetor local para a tarefa. Porém, nos algoritmos seguintes o atributo foi utilizado.
    - Melhorar os mecanismos para manipular os grafos direcionados (nos sentido de IDA e VOLTA). Isso não foi trabalhado porque não foi alvo durante o trabalho.
    - Transformar a classe de Lista para manipular QUALQUER informação usando os templates da linguagem c++. Na versão atual a lista é capaz de armazenar um dado do tipo criado chamado de "dado" que é na verdade um tipo primitivo int.
    - Manipular o conjunto de vértices e arestas com uma lista ao invés de um vetor dinâmico. Seria mais eficiente e utilizaria a memória de uma maneira mais adequada. No entanto, essa melhoria depende da anterior.

- Sugestões de continuidades:
    - Adicionar sobrecarga de operadores para as operações de inserção e subtração de vértices e arestas.
    - Codificar outros algoritmos específicos da teoria do grafos mesclando-os com heurísticas.
    - Adicionar métodos na classe Grafo que identifiquem determinados atributos do grafo como por exemplo se ele é um grafo completo, conexo, cíclico, etc.

Por fim, informamos que toda a codificação está disponível no GitHub.com na url: http://www.github.com/necioveras/grafos_MACC_UECE_PAA.

Referência

- Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, Clifford Stein, et al., *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009.

Fortaleza, 14 de Maio de 2013.