



UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação Orientada a Objetos

Trabalho Prático

Autor

Gabriel Meneses vieira

Rúbia Reis Guerra

Professora

Raquel Mini

ELE078

1 Introdução

2 Parte 1: Grafos

2.1 Classe

```
class Graph{
    int** adjM; // matriz de adjacencia
    int num_vert; // numero de vertices
    int num_edge; // numero de arestas
public:
    Graph(const int V = 3);
    // construtor, aloca memoria
    ~Graph();
    // destrutor, desaloca memoria
    /* Funcoes de estrutura do grafo */
    bool insert(const Edge& E, const int w = 1);
    // insere aresta, retorna true se aresta inserida
    bool remove(const Edge& E);
    // remove aresta, retorna true se aresta removida
    bool edge(const Edge& E);
    // verifica se aresta pertence ao grafo, true caso positivo
    int getNumVertex();
    // retorna num_vert
    int getNumEdge();
    // retorna num_edge
    bool isComplete();
    // verifica se o grafo e completo, true se completo
    void completeGraph();
    // completa grafico
    void randEdge();
    // completa grafo com pesos aleatorios nas arestas
    void print();
    // Imprime a matriz de adjacencia

    /* Funcoes de busca no grafo */
    void BFS(const int V);
    // Breadth First Search
    void DFS(const int V);
    // Depth First Search
    void DFSUtil(int v, bool visited[]);
    // Funcao auxiliar de DFS

    /* Funcoes especiais */
    int connComponents();
    // Retorna o numero de componentes conectados
    void Dijkstra(const int I, const int F);
    // Retorna menor caminho de I a F por Dijkstra
```

```

void AGM(bool pcv = false);
// Retorna a Arvore Geradora Minima (AGM)
// Se PCV = true, retorna solucao do PCV
int DUtil(int dist[], bool finalizado[]);
int AGMUtil(int dist[], bool finalizado[]);
// Funcoes auxiliares de Dijkstra e AGM
void printAGM(int parent[]);
// Imprime a AGM
void PCV();
// Completa o grafo e resolve o Problema do Caixeiro Viajante
};

```

2.2 Funções de Busca

2.2.1 Breadth First Search

Para a busca em largura no grafo, criou-se um vetor booleano de vértices visitados, inicializado em *false* para todas as posições, exceto para o vértice fornecido. Em seguida, criou-se uma pilha para busca em largura, de forma que cada vértice marcado como visitado é empilhado. Até que a pilha fosse esvaziada, retirou-se o primeiro vértice da pilha e verificou-se se todos seus vizinhos já foram visitados. Caso negativo, marcou-os como visitados e colocou-os na fila. Complexidade: $O(V + E)$, onde $V = numVert$ e $E = numEdge$.

2.2.2 Depth First Search

Para a busca em profundidade no grafo, criou-se um vetor booleano de vértices visitados, inicializado em *false* para todas as posições. Em seguida, foi chamada a função auxiliar recursiva *DFSUtil*, tendo como parâmetros o vértice fornecido para DFS e o vetor de visitados.

DFSUtil A função auxiliar marca o vértice fornecido como visitado e percorre todos os seus vértices adjacentes. Ao encontrar um vizinho não visitado, ocorre uma chamada recursiva de *DFSUtil*, tendo como parâmetros o índice do vértice vizinho e o vetor de visitados. A função recorre até que todos os vértices tenham sido visitados. Complexidade: $O(V + E)$, onde $V = numVert$ e $E = numEdge$.

2.3 Funções Especiais

2.3.1 Componentes Conectados

Para encontrar o número de componentes conectados no grafo, criou-se um vetor booleano de vértices visitados, inicializado em *false* para todas as posições. Em seguida, iterou-se sobre todos os vértices do grafo, chamando a função auxiliar recursiva *DFSUtil* para cada vértice marcado como não visitado. A quantidade de componentes conectados é incrementada a cada chamada de *DFSUtil*, visto que novas chamadas indicam que todos os vértices alcançáveis a partir de um dado vértice i foram visitados, formando um dos componentes do grafo. Como a implementação da solução utiliza uma busca em profundidade em todos os vértices, a complexidade final é $O(V + E)$, onde $V = numVert$ e $E = numEdge$.

2.3.2 Algoritmo de Dijkstra

Para resolver o algoritmo de Dijkstra, cria-se um vetor de finalizados para guardar todos os vértices já processados, ou seja, cuja distância mínima da fonte já foi calculada e finalizada. O

vetor é vazio, inicialmente. Em seguida, um vetor com as distâncias dos demais vértices ao vértice fornecido I é criado, de forma que a distância de I a I é igual a zero e de I aos demais é infinita. Enquanto o vetor de finalizados não incluir todos os vértices: escolhe-se o próximo vértice (u) não processado e que tenha valor mínimo de distância (calculado pela função *DAGMUtil*); inclui-se o vértice no vetor de finalizados; e atualiza-se a distância de todos os vértices adjacentes a u . A atualização da distância ocorre iterando-se em todos os vértices adjacentes e verificando-se se para cada adjacente v , a soma da distância de I até u e da aresta (u,v) é menor que a distância de I até v . Ao final, é retornado o valor da distância de I ao vértice fornecido F . Caso os vértices pertençam a componentes desconectados, a distância retornada é infinita. Complexidade: $O(V^2)$, onde $V = numVert$.

2.3.3 Árvore Geradora Mínima

Para encontrar a árvore geradora mínima do grafo, foi utilizado o algoritmo de Prim assumindo que o grafo possui apenas um componente conectado. A solução é parecida com a abordagem utilizada para o algoritmo de Dijkstra: cria-se um vetor de finalizados para guardar todos os vértices já processados, ou seja, que já foi inserido na AGM. O vetor encontra-se vazio, inicialmente. A cada vértice, é atribuído um valor de chave (inicialmente infinito). Atribui-se uma chave igual a 0 ao vértice inicial, para que seja escolhido primeiro. Enquanto o vetor de finalizados não incluir todos os vértices: escolhe-se um vértice u ainda não processado e que tenha um valor mínimo de chave; inclui-se u no vetor de finalizados e atualiza-se as chaves de todos os vértices adjacentes. Atualização dos valores das chaves é feita iterando-se através de todos os vértices adjacentes e, para adjacente v , se o peso da aresta (u,v) for menor que o valor da chave atual de v , atualiza-se o valor da chave como o peso de (u,v) . Complexidade: $O(V^2)$, onde $V = numVert$.

2.3.4 Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante (PCV) é conhecidamente da classe NP-Hard, e diz que dado um conjunto de cidades e a distância entre cada par de cidades, precisa-se encontrar a rota mais curta possível que visita cada cidade exatamente uma vez e retorna ao ponto de partida. Existem algoritmos aproximados para resolver o problema que funcionam somente se a instância do problema satisfizer a desigualdade triangular a seguir: o menor caminho para alcançar um vértice j a partir de i é ir a j diretamente de i , em vez de algum passar por outro vértice (ou vértices) k , ou seja, $dist(i, j)$ é sempre menor ou igual $dist(i, k) + dist(k, j)$. Quando a função de distância satisfaz a desigualdade do triângulo, podemos projetar um algoritmo aproximado para o PCV que retorna um roteiro cuja distância total nunca é maior que o dobro da distância percorrida em um roteiro ótimo. Dadas as considerações, pode-se resolver o PCV utilizando-se o algoritmo de Prim modificado: encontrada a árvore geradora mínima (com raiz no vértice 0, ponto inicial e final do vendedor), todos os vértices do encaminhamento da AGM são listados e o vértice raiz é adicionado ao final, formando um ciclo. A complexidade final é igual a complexidade para a AGM: $O(V^2)$, onde $V = numVert$.

3 Parte 2: Sequências

3.1 Classe mãe

```

/***** Class Seq *****/
class Seq{

```

```

    public:
        std::vector<unsigned long int>elements;
        Seq();
        Seq(unsigned);
        ~Seq();
        unsigned length();
        unsigned long elem(unsigned i);
        void print(ostream&);
        friend ostream& operator<<(ostream&, Seq&);
        void printS(unsigned, unsigned);
        void empty();
    protected:
        void gen_elems(unsigned i);
};

```

A classe mãe é possui todos os métodos para gerar, armazenar e manipular sequencias em geral, a sequencia escolhida para ser a padrão foi a sequencias dos números naturais (0, 1, 2, 3, 4, 5, ...). Esse classe é composta por um elemento da classe vector, responsável por armazenar os elementos calculados.

Além disso a classe possui os seguintes métodos:

- Seq() - Construtor de uma sequencia vazia;
- Seq(unsigned i) - construtor de uma sequencia com i elementos;
- ~Seq() - Destrutor, libera a memoria ocupada pela sequencia;
- unsigned length() - Retorna o número de elementos da sequencia;
- unsigned long elem(unsigned i) - retorna o elemento i, caso não exista, calcula-se ate esse elementos para retorná-lo;
- void print(ostream& os) - coloca toda a sequencia no stream os;
- friend ostream& operator<<(ostream&, Seq&) - sobrecarga do operador << para lidar com objetos da classe Seq;
- void printS(unsigned,unsigned) - imprime uma secção da sequencia na saída padrão;
- void empty() - remove todos elementos de uma sequencia;
- void gen_elems(unsigned i) - gera elementos até o elemento i.

3.2 Classes filhas

```

/***** Class Fibonacci *****/
class Fibonacci: public Seq{
    public:
        unsigned long elem(unsigned i);
        Fibonacci();
        Fibonacci(unsigned);
        ~Fibonacci();

```

```

        protected:
            void gen_elems(unsigned i);
};

/***** Class Lucas *****/
class Lucas: public Seq{
public:
    unsigned long elem(unsigned i);
    Lucas();
    Lucas(unsigned);
    ~Lucas();
protected:
    void gen_elems(unsigned i);
};

/***** Class Pell *****/
class Pell: public Seq{
public:
    unsigned long elem(unsigned i);
    Pell();
    Pell(unsigned);
    ~Pell();
protected:
    void gen_elems(unsigned i);
};

/***** Class Triangle *****/
class Triangle: public Seq{
public:
    unsigned long elem(unsigned i);
    Triangle();
    Triangle(unsigned);
    ~Triangle();
protected:
    void gen_elems(unsigned i);
};

/***** Class Square *****/
class Square: public Seq{
public:
    unsigned long elem(unsigned i);
    Square();
    Square(unsigned);
    ~Square();
protected:
    void gen_elems(unsigned i);
};

/***** Class Pentagon *****/

```

```

class Pentagon: public Seq{
public:
    unsigned long elem(unsigned i);
    Pentagon();
    Pentagon(unsigned);
    ~Pentagon();
protected:
    void gen_elems(unsigned i);
};

```

Todas as classes filhas possuem uma estrutura similar, herdando todos métodos públicos da classe mãe além de possuir os seguintes métodos específicos de cada sequência implementada:

- Construtor genérico - Inicializa a sequência com o mínimo de elementos necessários para sua continuidade;
- Construtor específico - Inicializa a sequência com um número específico de elementos;
- Destrutor - libera a memória ocupada pela sequência;
- Gerador de elementos - gera os elementos da sequência até o elemento escolhido.

3.3 Classe Container e Composição

```

/***** Class Container *****/
class Container{
public:
    std::vector<Seq>sequences;
    Container();
    ~Container();
    void insert(Seq);
    unsigned length();
    void print();
    void printS(unsigned, unsigned);
    void printE(unsigned);
};

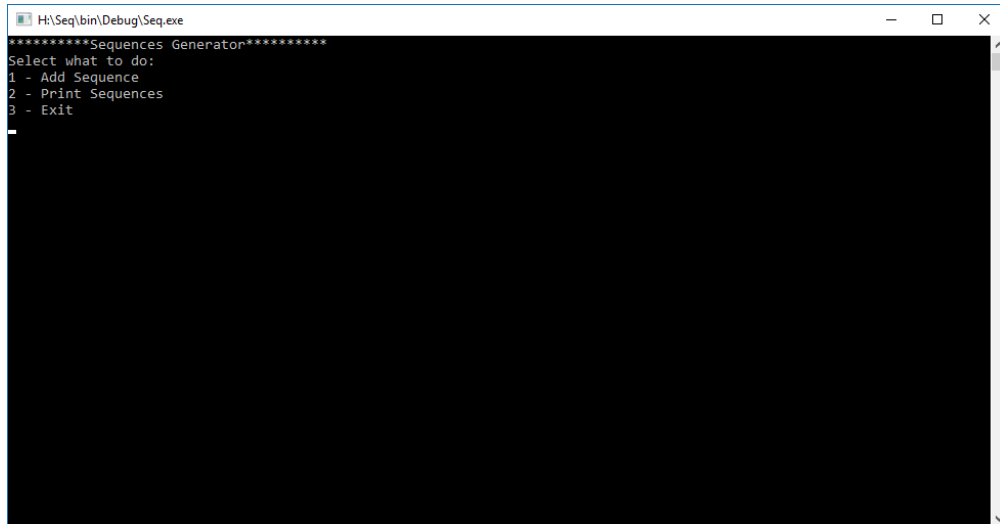
```

A classe Container é formada pela composição da classe Seq, ela é responsável por armazenar vários objetos do tipo sequência e manipular esses objetos. As sequências são armazenadas em um objeto da classe vector e os seguintes métodos foram implementados para manipular as sequências:

- void insert(Seq) - insere a sequência S no container;
- unsigned length() - retorna o número de sequências no container;
- void print() - imprime todas as sequências armazenadas na saída padrão;
- void printS(unsigned, unsigned) - imprime uma seção de todas as sequências armazenadas na saída padrão;
- void printE(unsigned) - imprime um elemento específico de todas as sequências armazenadas na saída padrão.

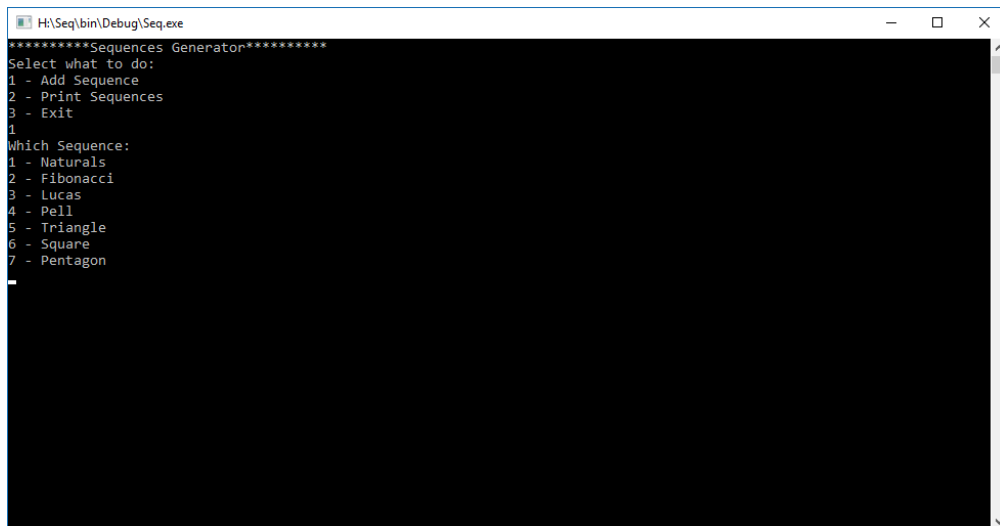
3.4 Menu

Como interface do usuário com as classes implementadas foi implementado um menu em que poderia ser selecionada a sequência desejada, o número de elementos, além de opções de como exibir essas sequências.



```
H:\Seq\bin\Debug\Seq.exe
*****Sequences Generator*****
Select what to do:
1 - Add Sequence
2 - Print Sequences
3 - Exit
```

Figura 1: Menu inicial



```
H:\Seq\bin\Debug\Seq.exe
*****Sequences Generator*****
Select what to do:
1 - Add Sequence
2 - Print Sequences
3 - Exit
1
Which Sequence:
1 - Naturals
2 - Fibonacci
3 - Lucas
4 - Pell
5 - Triangle
6 - Square
7 - Pentagon
```

Figura 2: Menu de sequencias

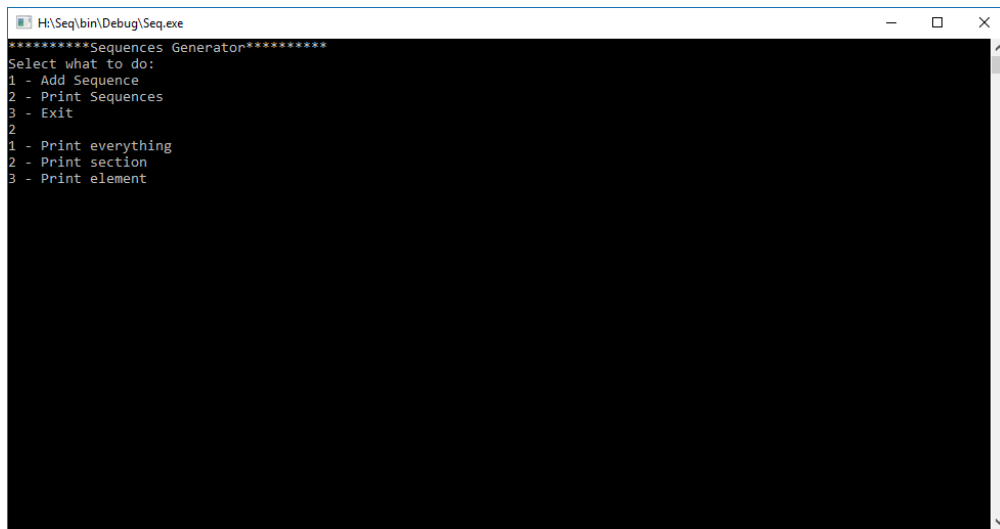


Figura 3: Menu de impressão

Referências

- [1] Mamber, Udi. *Introduction to Algorithms: A Creative Approach*. Pearson, 1ª Ed., 1989.
- [2] Cormen, Thomas H, Stein, Clifford, Rivest, Ronald L., Leiserson, Charles E. *Introduction to Algorithms*. McGraw-Hill Higher Education, 3ª Ed., 2009.
- [3] V. Sanka. Geeks for Geeks, múltiplas seções.
<http://www.geeksforgeeks.org/>