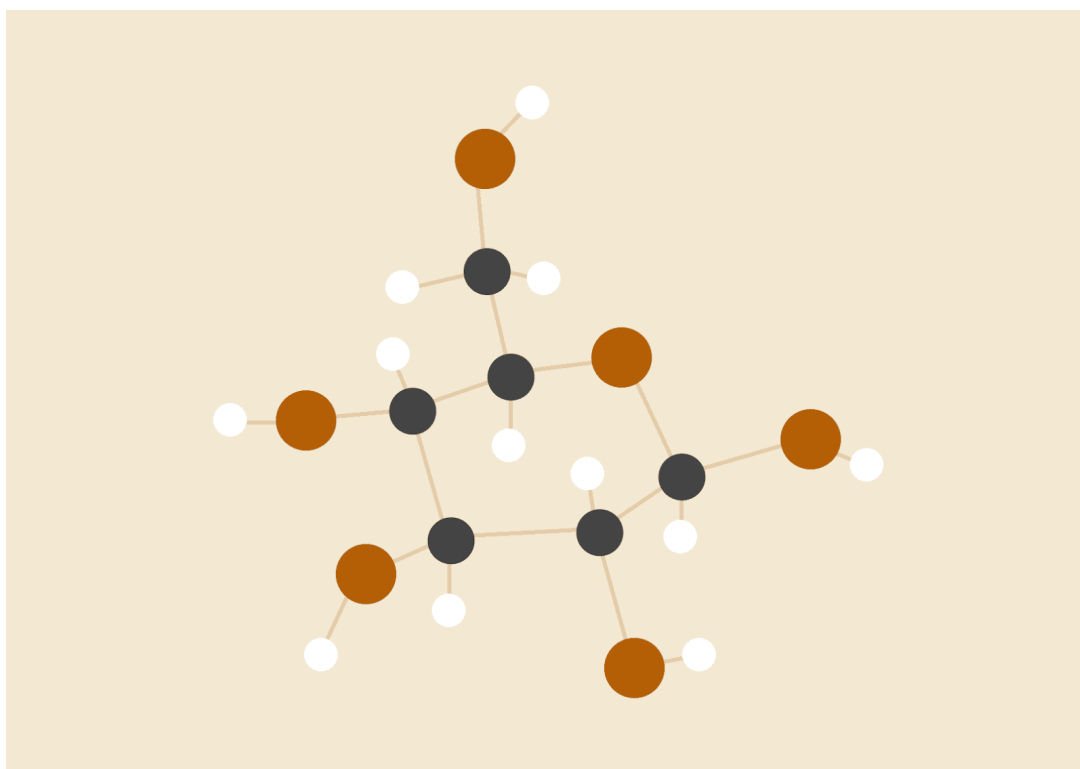


ESTRUTURA DE DADOS II

Implementação de grafos e seus principais algoritmos.

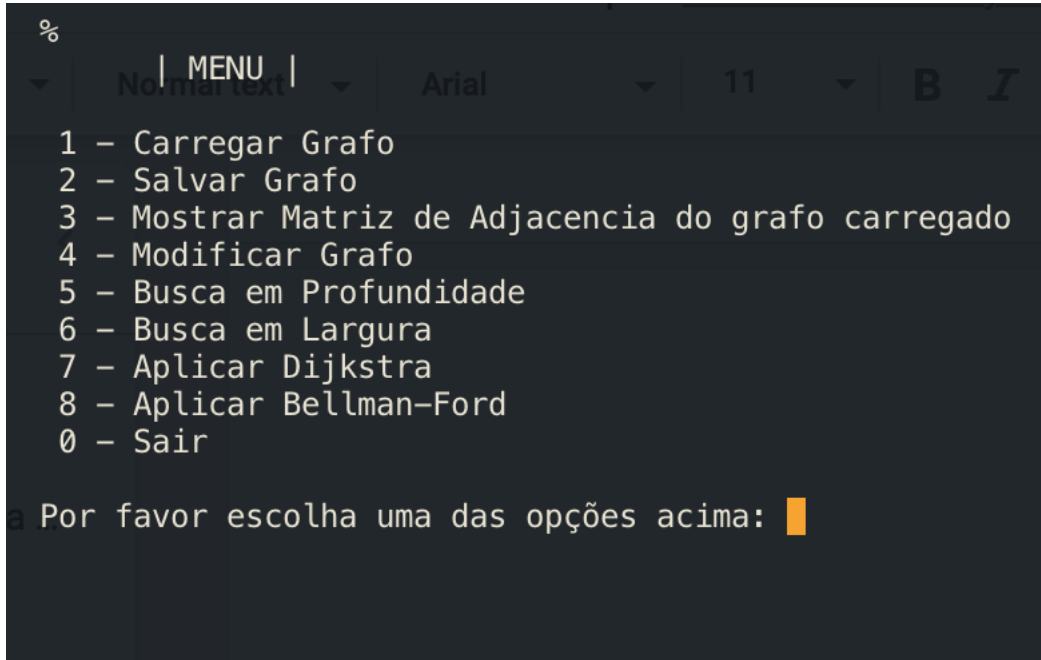


Marcelo Eduardo Rodrigues da Silva Filho

CIÊNCIA DA COMPUTAÇÃO 2º ANO

Como Utilizar a Aplicação:

O uso da aplicação desenvolvida para esse trabalho, se baseia na navegação pelo seguinte menu, que será exibido após a execução do arquivo “**main.c**” na pasta **src** do trabalho:



```
%  
| MENU |  
Normal text Arial 11 B I  
1 - Carregar Grafo  
2 - Salvar Grafo  
3 - Mostrar Matriz de Adjacencia do grafo carregado  
4 - Modificar Grafo  
5 - Busca em Profundidade  
6 - Busca em Largura  
7 - Aplicar Dijkstra  
8 - Aplicar Bellman-Ford  
0 - Sair  
a Por favor escolha uma das opções acima: █
```

Cada opção do menu será correspondente a uma funcionalidade diferente, segue abaixo as funcionalidades:

1. **Carregar Grafo:** Esta opção pede ao usuário, o caminho do arquivo de grafo o qual o usuário quer carregar, e faz o processamento deste para transformá-lo numa matriz de adjacência. Exemplo de caminho: "arquivos/1.grafo"
 - a. A função responsável por esse item do menu é a função:
le_matriz_arquivo(caminho_arquivo, matriz_adj, &tipo, &tamanho)
Essa função recebe o caminho do arquivo e passa 3 variáveis que irão receber retornos da função, uma que receberá a matriz de adjacência gerada, outra receberá o tipo do grafo, e outra a quantidade de vértices do mesmo.
A função pode ser encontrada na **linha 79** do arquivo “**matrizAdjacencia.h**”.
2. **Salvar Grafo:** Faz o salvamento de um grafo previamente carregado (pode ter sido modificado ou não). Irá pedir o caminho para salvar o arquivos, como por exemplo, "arquivos/1.grafo".

- a. A função responsável por esse item é a seguinte:
escreve_matriz_arquivo(arquivo, matriz_adj, tipo, tamanho)
Essa função recebe o arquivo de salvamento, a matriz que será salva, e o tipo e tamanho do grafo.
A função pode ser encontrada na **linha 61** do arquivo “**matrizAdjacencia.h**”.
- 3. **Mostrar Matriz de Adjacência do grafo carregado:** Basicamente vai mostrar o grafo carregado na memória na forma de matriz de adjacência, para o usuário ter controle de qual grafo está utilizando.
 - a. Chama a função **print_matriz**(matriz_adj, tamanho), que simplesmente pega a matriz e o tamanho do grafo, e realiza um loop de prints, para mostrar a matriz referente ao grafo carregado.
A função pode ser encontrada na **linha 31** do arquivo “**matrizAdjacencia.h**”.
- 4. **Modificar Grafo:** Após carregar um grafo, o usuário pode utilizar essa opção e modificar (adicionar/remover/modificar arestas) o seu grafo. Será pedido, a linha e coluna da matriz de adjacência onde se deseja fazer a alteração, e o peso que será gravado na posição especificada.
 - a. Nenhuma função é utilizada, apenas uma mudança simples de item em array bidimensional.
- 5. **Busca em Profundidade:** Será requisitado do usuário, o caminho do arquivo de grafo no qual se deseja fazer a busca em profundidade, o caminho onde se deseja salvar a tabela resultante e a raiz onde se iniciará a busca.
 - a. Chama a função **busca_profundidade**(raiz, entrada, saída), que recebe a raiz de onde será iniciada a busca, o arquivo de origem do grafo onde será feita a busca e o arquivo de salvamento do grafo.
A função pode ser encontrada na **linha 59** do arquivo “**buscas.h**”.
- 6. **Busca em Largura:** Segue o mesmo padrão da opção de "Busca em Profundidade".
 - a. Chama a função **busca_largura**(raiz, entrada, saída), que recebe a raiz de onde será iniciada a busca, o arquivo de origem do grafo onde será feita a busca e o arquivo de salvamento do grafo.
A função pode ser encontrada na **linha 165** do arquivo “**buscas.h**”.
- 7. **Aplicar Dijkstra:** Opção que irá mostrar o grafo carregado na memória e perguntar se o usuário deseja realmente aplicar naquele grafo, caso sim, é requisitado a raiz para iniciar o algoritmo e o caminho para salvar os resultados. Caso o usuário não deseje

prosseguir com a aplicação do algoritmo naquele grafo, ele é redirecionado ao menu principal, para que possa carregar o grafo que deseja.

- a. Chama a função **dijkstra**(matriz_adj, raiz, tamanho, arquivo_destino), que recebe a matriz de adjacência onde será aplicado o algoritmo, a raiz de início, o tamanho do grafo que está sendo trabalhado e o arquivo de salvamento dos resultados.

A função pode ser encontrada na **linha 40** do arquivo “**caminhos.h**”.

8. **Aplicar Bellman-Ford:** Segue a mesma linha de interação que a opção "7 - Aplicar Dijkstra".

- a. Chama a função **bellman_ford**(matriz_adj, raiz, tamanho, tipo, arquivo_destino), que recebe a matriz de adjacência onde será aplicado o algoritmo, a raiz de início, o tamanho e o tipo do grafo que está sendo trabalhado e o arquivo de salvamento dos resultados.

A função pode ser encontrada na **linha 128** do arquivo “**caminhos.h**”.

9. **Sair:** Sair do programa.

Extensões de arquivos:

.grafo: Extensão utilizada para representação de grafos em arquivo texto.

.matriz: Os arquivos são iguais aos **.grafo** porém, essas são originadas de matrizes de adjacência, e não escritas pelo usuário.

.tbl: Extensão utilizada para representar as tabelas de busca em largura, “tbl” pode-se ler “tabela de busca em largura”.

.tbp: Extensão utilizada para representação das tabelas de busca em profundidade, “tbp” pode-se ler “tabela de busca em profundidade”.

.dijkstra: Extensão utilizada nos arquivos que guardam a tabela gerada pelo algoritmo de Dijkstra.

.bellman: Extensão utilizada nos arquivos que guardam a tabela gerada pelo algoritmo de Bellman-Ford.

.log: Extensão dos arquivos de log gerados pela execução do programa.

OBS - 1: As extensões são apenas uma convenção, não utilizá-las não muda em nada no funcionamento do programa.

OBS - 2: Para ativar a geração de log do programa, é necessário ir no arquivo "matrizAdjacencia.h" e trocar o valor da variável global para 1.

***PS:** Vale citar que, como não é o foco do trabalho, o menu pode e provavelmente irá, apresentar algumas inconsistências, como tratamento de entradas por exemplo, portanto, caso ocorra algum erro do tipo, recomenda-se a reinicialização do programa por completo.

Estruturas de Dados utilizadas:

A estrutura de dados mais explorada nesse trabalho foi o **arranjo** (array), que são uma estrutura de armazenamento na qual, qualquer item lá dentro pode ser encontrado utilizando um índice. As aplicações de arranjos dentro do projeto, consistiram basicamente para representação e manuseio das matrizes de adjacência, caso em que foi trabalho o uso de arranjos bidimensionais, e na organização dos vértices representativos dos nossos grafos. No segundo uso, foram feitas **structs** para os tipos de vértices e essas foram armazenadas dentro de arranjos, de forma que, esses arranjos acabaram tendo dentro de si, todas as informações necessárias para realizar operações sobre os grafos. Segue abaixo as duas structs feitas para vértices, chamadas de "struct node_largura" e "struct node_profundidade".

```
struct node_profundidade{
    int vertice; //contem o numero do vertice
    char cor; // B = Branco, R = Vermelho
    int descoberta;
    int finalizacao;
};
```

```
struct node_largura{
    int vertice;
    char cor; // B = Branco, R = Vermelho
    int distancia;
    int pai;
};
```

Fora os arranjos, foi utilizada também a estrutura de dados **Fila**, estrutura de dados que segue o conceito FIFO (first in first out). A fila, foi utilizada para auxiliar na busca em largura, pois nela, enfileiramos os vértices adjacentes ao qual estamos processando, para que sejam processados posteriormente e na ordem correta.

Explicação de implementações:

1. Busca em profundidade:

A ideia da busca em profundidade é buscar, sempre que for possível, o vértice mais profundo no grafo. Então, a busca em profundidade tenta, a partir de um vértice, explorar todos os adjacentes a ele, que não foram ainda explorados. Quando todos os adjacentes são explorados, a busca vai para um vértice anterior a ele (backtracking), e isso continua até que todos os vértices que são alcançáveis sejam descobertos. Segue abaixo um trecho de código onde a função que visita, "visita_profundidade", é chamada para todos os vértices:

```
int tempo = 0;
visita_profundidade(vertices[raiz], vertices ,matriz_adj, tamanho, &tempo);
for(int i = 0; i < tamanho; i++){
    if(vertices[i] -> cor == 'B' && i != raiz){ //Se a cor do vertice for branca,
                                                //iremos chamar a funcao de visita
        visita_profundidade(vertices[i], vertices ,matriz_adj, tamanho, &tempo);
    }
}
```

Essa função chamada "visita_profundidade" vai fazer o trabalho de visitar e atualizar a situação dos vértices baseando-se em um sistema de cores onde:

- **Branco** - representa os vértices ainda não visitados
- **Cinza** - representa os vértices visitados, mas que tem adjacentes brancos.
- **Preto** - representa os vértices visitados e que tem todos os seus adjacentes visitados.

Lembrando que, quando o momento em que o vértice é "pintado" de cinza, é seu tempo de descoberta, e quando é pintado de preto, é seu tempo de finalização.

Assim que a função de visita é executada sobre um vértice **v**, sua cor é trocada para cinza, e seu tempo de descoberta é atualizado, e então a própria função de visita é novamente aplicada em todos os vértices adjacentes brancos a **v**, quando não restarem mais vértices adjacentes brancos, a cor do vértice **v** é atualizada para preto, e o tempo

de finalização também é atualizado, segue o trecho de código representativo dessa função:

```
void visita_profundidade(struct node_profundidade* node, struct node_profundidade* vertices[], int matriz[TAM][TAM], int tamanho, int *tempo){
    node -> cor = 'C'; //atualizando cor do vertice visitado
    *tempo = *tempo + 1; //atualizando o tempo
    node -> descoberta = *tempo; //atualizando o tempo de descoberta
    if(logger){
        log_print(arquivo_log_buscas, "buscas.h - visita_profundidade - Linha 32", "\n [buscas.h - visita_profundidade() - Linha 32]: DESCOBERTA DE %d: %d\n", node -> vertice, node -> descoberta);
        //clearScreen();
    }
    //para cada vertice adjacente
    for(int i = 0; i < tamanho; i++){
        if(matriz[node -> vertice][i] != 0){ //SE FOR ADJACENTE AO VERTICE I
            if(vertices[i] -> cor == 'B'){ // E SE O VERTICE AO QUAL É ADJACENTE FOR BRANCO
                if(logger){
                    log_print(arquivo_log_buscas, "buscas.h - visita_profundidade - Linha 41", "\n\n [buscas.h - visita_profundidade() - Linha 41]: A PARTIR DE %d VISITANDO: %d\n", node -> vertice, i);
                }
                visita_profundidade(vertices[i], vertices, matriz, tamanho, tempo);
            }
        }
    }
    node -> cor = 'P';
    *tempo = *tempo + 1;
    node -> finalizacao = *tempo;
}
```

***PS: Vale lembrar que todos os vértices são inicializados como Brancos.**

2. Busca em largura:

A ideia da busca em largura, é visitar todos os vértices de um mesmo nível da árvore por vez, ou seja, dada uma raiz, irão ser processados primeiro, todos os vértices com distância 1 dessa raiz, depois os com distância 2, 3 e assim por diante, até que todos os vértices tenham sido processados (a “distância” é na verdade a quantidade de saltos para chegar em determinado vértice a partir da raiz).

Nessa busca, também se utiliza um esquema de cores, para que possamos sinalizar a situação dos vértices, é bem parecido com o esquema da busca em profundidade, porém com umas leves mudanças:

- **Branco** - representa os vértices ainda não visitados/não conhecidos.
- **Cinza** - representa os vértices conhecidos, mas não conhecidos (seus adjacentes não foram adicionados na fila).
- **Preto** - representa os vértices conhecidos e visitados (todos os seus adjacentes foram adicionados na fila).

O algoritmo enfileira primeiro a raiz **u** (que já começa cinza), e a partir dela inicia a busca, desenfileirando o topo (raiz) e para cada vértice **v** adjacente branco a ela, atualiza a cor de **v** para cinza, atualiza o pai para **u**, atualiza a distância de **v** para “distância do pai + 1” e por fim, enfileira-se esse vértice adjacente **v**. Quando todos os adjacentes são conhecidos, o vértice **u**, se torna preto. Segue um trecho de código representativo:

Inicialização dos vértices, repare que a raiz recebe ‘C’, ou seja, cor cinza:

```

for(int j = 0; j < tamanho; j++){ //Criando as estruturas e Inicializando o resto dos vertice
    struct node_largura *temp = (struct node_largura*)malloc(sizeof(struct node_largura)); //
    if(j != raiz){
        temp -> vertice = j;
        temp -> cor = 'B';
        temp -> distancia = 999; //INFINITO
        temp -> pai = -1;
        vertices[j] = temp;
        if(logger == 1){
            printf("Vertice Inicializado: %d \n", temp -> vertice);
            log_print(arquivo_log_buscas, "buscas.h - Linha 262", " [buscas.h - busca_largura]");
        }
    }
    else{
        node_raiz -> vertice = raiz; //criando o nó de raiz
        node_raiz -> cor = 'C';
        node_raiz -> distancia = 0;
        node_raiz -> pai = -1;
        vertices[raiz] = node_raiz; //colocando a raiz no vetor
    }
}
//Fim da inicializacao

```

Aqui temos o trecho de código onde é feito o loop pela fila, para visitar todos os vértices enfileirados:

```

while(queue -> primeiro != NULL){ //enquanto a fila nao estiver vazia
    atual = desenfileirar(queue);
    struct node_largura* vertice_atual = atual -> conteudo;

    if(logger){
        printf("\nDesenfileirando vértice: %d\n", vertice_atual -> vertice);
        log_print(arquivo_log_buscas, "buscas.h - Linha 287", "\n [buscas.h - busca_largura]");
    }
    //para cada vertice adjacente

    if(logger){
        printf("\nVerificando os vertices adjacentes brancos ao: %d\n", vertice_atual -> vertice);
        log_print(arquivo_log_buscas, "buscas.h - Linha 304", "\n [buscas.h - busca_largura]");
    }
    for(int i = 0; i < tamanho; i++){
        if(matriz_adj[vertice_atual -> vertice][i] != 0){ //SE FOR ADJACENTE
            if(vertices[i] -> cor == 'B'){ // É SE O VERTICE AO QUAL É ADJACENTE É BRANCO
                //ATUALIZANDO VERTICES VISITADOS
                vertices[i] -> cor = 'C';
                vertices[i] -> distancia = (vertice_atual -> distancia) + 1;
                vertices[i] -> pai = vertice_atual -> vertice;
                enfileirar(queue, vertices[i]);

                if(logger){
                    printf("\nAtualizando vertice adjacente branco encontrado: %d\n", vertices[i] -> vertice);
                    log_print(arquivo_log_buscas, "buscas.h - Linha 287", "\n [buscas.h - busca_largura]");
                    printf("Cor: %c \nDistancia: %d\nPai: %d\n", vertices[i] -> cor, vertices[i] -> distancia, vertices[i] -> pai);
                    log_print(arquivo_log_buscas, "buscas.h - Linha 287", "\n [buscas.h - busca_largura]");
                    log_print(arquivo_log_buscas, "buscas.h - Linha 287", "\n [buscas.h - busca_largura]");
                    log_print(arquivo_log_buscas, "buscas.h - Linha 287", "\n [buscas.h - busca_largura]");
                }
            }
        }
    }
    //printf("DISTANCIA DO VERTICE %d: %d\n", vertices[i] -> vertice, vertices[i] -> distancia);
    vertice_atual -> cor = 'P';
}

```


3. Dijkstra:

O algoritmo de Dijkstra soluciona o problema do caminho mais curto em grafos orientados ou não orientados, ponderados, que não contém arestas de peso negativo. Em um primeiro momento, todos os vértices são inicializados:

```
for(int i = 0; i < tamanho; i++){
    //inicializando vertices
    if(logger){
        log_print(arquivo_log_caminh
    }
    struct node_largura *temp = (str
    if(i != raiz){
        temp -> vertice = i;
        temp -> distancia = 999; //
        temp -> pai = -1; //
        vertices[i] = temp;
    }
    else{
        temp -> vertice = i;
        temp -> distancia = 0; //ra
        temp -> pai = -1; //pa
        vertices[i] = temp;
    }
}
```

Todos os vértices recebem distância “infinito” e pai “NULL”, exceto a raiz, que tem distância ‘0’.

Em segundo momento, se baseando num conjunto Q, de vértices que ainda não contém custo de menor caminho, iremos realizar o “relaxamento” das arestas, que é um processo onde se verifica se o caminho que o está atribuído ao vértice é o melhor possível, ou se existe uma mudança que possa melhorar essa situação, e isso é feito até que Q esteja vazio ou até que um outro conjunto auxiliar S, que representa os vértices com custo mínimo calculado, esteja cheio:

