# Relatório Tarefa 5 INF1010 - Algoritmos em Grafos

João Gabriel Peixoto Drumond Lopes - 2210168

## 1 Introdução

A tarefa 5 envolve a representação de um grafo utilizando lista de adjacências e a implementação de dois algoritmos nesse grafo. Os dois algoritmos são: Algoritmo de Dijkstra, a partir do vértice A, para calcular o menor caminho; e a busca em profundidade ou DFS, também a partir do vértice A.

## 2 Estrutura do Programa

O programa está estruturado em dois módulos. O primeiro, *grafo.c*, contém as definições de tipos de dados estruturados usados, Grafo, Vertice, Aresta, Pilha e Fila. O segundo, *main.c*, é o módulo principal, que chama as funções de *grafo.c*.

### Definições de tipo de *grafo.c*

- typedef struct grafo Grafo: estrutura de cabeça de grafo.

  Campos: Vertice** vertices, array de Vertice*, contendo os vértices pertencentes ao grafo; int ocupados, que armazena a quantidade de vértices já inseridos; e int num_vertices, número máximo de vértices que o grafo pode ter.

- typedef struct vertice Vertice: estrutura de vértice de grafo.

  Campos: int chave, valor de 0 a num_vertices - 1, indica o local físico de armazenamento do vértice no grafo; int valor, valor armazenado pelo vértice, no programa utilizado como label; int num_arestas, número de arestas ligados a cada vértice; Aresta* arestas, lista encadeada de estrutura de aresta; int menor_caminho, menor caminho do vértice até algum outro, calculado pelo Algoritmo de Dijsktra; Vertice* downprox, utilizado para quando o Vertice é usado como estrutura de nó de pilha ou de fila.

- typedef struct aresta Aresta: estrutura de aresta de grafo.

Campos: int vertice2, vertice para o qual a aresta aponta, outro vértice descoberto por meio da estrutura de Vertice a qual a aresta encontra-se ligada; int peso, peso da aresta; Aresta* prox, ponteiro para próxima estrutura de aresta na lista de adjacências.

- typedef struct pilha Pilha: estrutura de cabeça de pilha.

  Campos: Vertice* topo, vertice no topo da pilha.

- typedef struct fila Fila: estrutura de cabeça de fila. Campos: Vertice* inicio, ponteiro para o vértice no início da fila; Vertice* fim, ponteiro para o vértice no fim da fila.

- typedef int(*comp_func)(Aresta*, Aresta*): ponteiro para função que retorna inteiro e recebe dois ponteiros para estrutura de aresta (usado para funções de comparação).

## Funções de acesso de $grafo.c$:

- Grafo* criar_grafo(int num_vertices): A função de criar grafo aloca memória na heap para uma cabeça de estrutura de grafo. Retorna o espaço de memória alocado, com um vetor de vértice de tamanho $num\_vertices$.

- void inserir_vertice(int chave, char valor, Grafo* grafo): Insere um vértice de *chave* e *valor* passados como argumento em um grafo. *chave* deve ser um valor entre 0 e $grafo.num\_vertices - 1$.

- void inserir_aresta(int v1, int v2, int peso, Grafo* grafo): Insere uma aresta entre $v1$, vértice 1, e $v2$, vértice 2, com um determinado peso.

- void libera_grafo(Grafo* grafo): Libera toda a memória alocada na função criar_grafo. Inclui o vetor de vértices e as listas encadeadas de arestas.

- void imprime_grafo(Grafo* grafo): Imprime a estrututura de grafo em sua totalidade, sem preocupação com o percurso.

- void dijkstra_algorithm(Grafo* grafo, int origem, int keep_values): Executa o Algoritmo de Dijkstra no grafo passado para a função, com o vértice de origem definido. A flag *keep_values* determina se os valores calculados devem ser apagados após a execução da função ou não.

- void dfs(Grafo* grafo, int origem): Executa um percurso de busca em profundidade no grafo.

## Funções internas de $grafo.c$:

- Vertice* criar_vertice(int chave, char valor): Função de criação e alocação de memória para uma estrutura de vértice, de chave e valor passado. de parâmetro.

- void print_vertice(Vertice* vertice): Realiza a impressão de parte dos conteúdos de uma estrutura vértice. Utilizada para debugging e testagem.
- Aresta* criar_aresta(int v2, int peso): Função de criar e alocar espaço para uma estrutura de aresta de grafo.
- int compara_vertice(Aresta* aresta, Aresta* cursor): Função que retorna se o campo de vértice da primeira estrutura é maior que o da segunda.
- int compara_peso(Aresta* aresta, Aresta* cursor): Função que retorna se o campo de peso da primeira estrutura é menor que o da segunda.
- Aresta* inserir_ordenado_aresta(Aresta* aresta, Aresta* head): Função de inserção em lista encadeada de Aresta de forma a manter a ordenação, para lista de adjacências de grafo.
- void push_vertice(Pilha* pilha, Vertice* vertice): Insere um vértice no topo de uma estrutura de pilha.
- Vertice* pop_vertice(Pilha* pilha): Retira um vértice do topo de uma estrutura de pilha.
- Pilha* criar_pilha(void): Função para criar e alocar espaço de memória para estrutura de cabeça de pilha.
- void libera_pilha(Pilha* pilha): Libera memória alocada de cabeça de pilha.
- void enqueue_vertice(Fila* fila, Vertice* vertice): Insere vértice em fila.
- Vertice* dequeue_vertice(Fila* fila): Remove vértice de fila.
- Fila* criar_fila(void): Função para criar e alocar espaço de memória para estrutura de cabeça de fila
- void libera_fila(Fila* fila): Libera memória alocada de cabeça de fila.

# 3 Solução

Inicialmente, é inicializada uma estrutura de grafo, com número de vértices igual a 7. São inseridos os vértices de 0 a 6, 'A' a 'F', mais 'H'.

A função de criação de vértice cria uma estrutura de vértice e insere na lista de adjacências do grafo, na posição apropriada passada como parâmetro.

```c
// Cria grafo
Grafo* grafo = criar_grafo(MAX_VERTICES);

// Insere os vértices
inserir_vertice(0, 'A', grafo);
inserir_vertice(1, 'B', grafo);
inserir_vertice(2, 'C', grafo);
inserir_vertice(3, 'D', grafo);
inserir_vertice(4, 'E', grafo);
inserir_vertice(5, 'F', grafo);
inserir_vertice(6, 'H', grafo);
```

Em seguida, são inseridas as arestas, que recebem como parâmetro o vértice1 e o vértice2. A aresta é inserida por criar_aresta tanto na lista de adjacências de v1 quanto de v2.

```c
// Insere as arestas
inserir_aresta(0, 1, 5, grafo);
inserir_aresta(0, 2, 4, grafo);
inserir_aresta(0, 3, 2, grafo);
inserir_aresta(1, 2, 6, grafo);
inserir_aresta(1, 4, 6, grafo);
inserir_aresta(1, 6, 9, grafo);
inserir_aresta(2, 3, 3, grafo);
inserir_aresta(2, 4, 4, grafo);
inserir_aresta(3, 4, 5, grafo);
inserir_aresta(3, 5, 9, grafo);
inserir_aresta(4, 6, 6, grafo);
inserir_aresta(4, 5, 2, grafo);
inserir_aresta(5, 6, 3, grafo);
```

O grafo é impresso, para verificar a inserção adequada das estruturas no passo anterior. O percurso em profundidade, DFS, é realizado em sequência.

```
// Imprime os contéudos do grafo
imprime_grafo(grafo);

// Realiza percurso dfs do grafo
// com o vértice A de origem
printf("DFS: Origem A\n");
dfs(grafo, 0);
```

Esse percurso ocorre da seguinte forma: uma estrutura de pilha é inicializada e recebe o primeiro vértice. A partir desse vértice, são postos os próximos vértices adjacentes marcados como não visitados na pilha, em ordem crescente de valor de vertice (mesmo comportamento que uma ordem alfabética). Isso se repete até a pilha estar vazia. Os vértices em ordem de acesso são impressos, durante a execução do algoritmo.

Com a próxima chamada, é executado o algoritmo de Dijkstra. Ele ocorre da seguinte forma: é criada uma estrutura de fila. Lá é posto o primeiro vértice, marcado como adicionado. Existem dois vetores, adicionados e visitados. Enquanto a fila não estiver vazia, um vértice é retirado. Então, todos os vértices adjacentes são visitados pelas arestas, e o valor é somado no valor do caminho para vertice 2, se a soma for menor que o já presente. Os vértices ainda não visitados são adicionados na fila. No fim, são impressos os menores caminhos. Se indicado, os valores calculados são apagados da estrutura. Caso contrário, lá persistem.

```
// Executa o algoritmo de Dijkstra
// com o vértice A como origem
printf("Dijkstra: Origem A\n");
dijkstra_algorithm(grafo, 0, 0);
```

Toda a memória alocada é liberada ao fim da execução do programa, incluindo as auxiliares, dentro das funções de DFS e de Dijkstra (também lá liberadas).

```
// Libera memória do grafo
libera_grafo(grafo);
```

### Saída do Programa

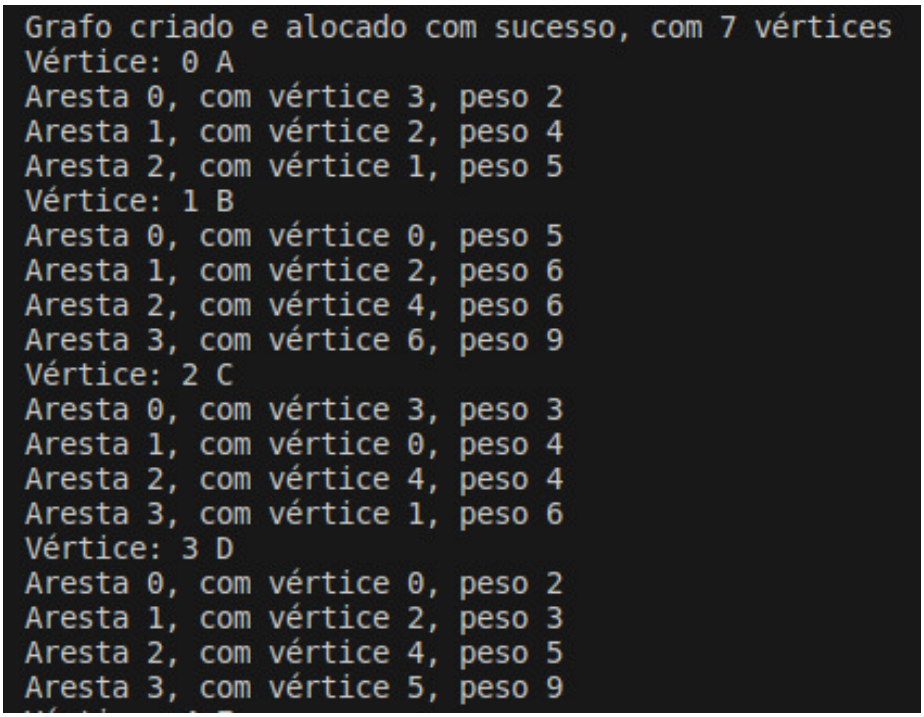
As imagens da saída do programa tiveram que ficar no final do pdf por conta de problemas de espaço e de organização do formato.

### Saída do Valgrind

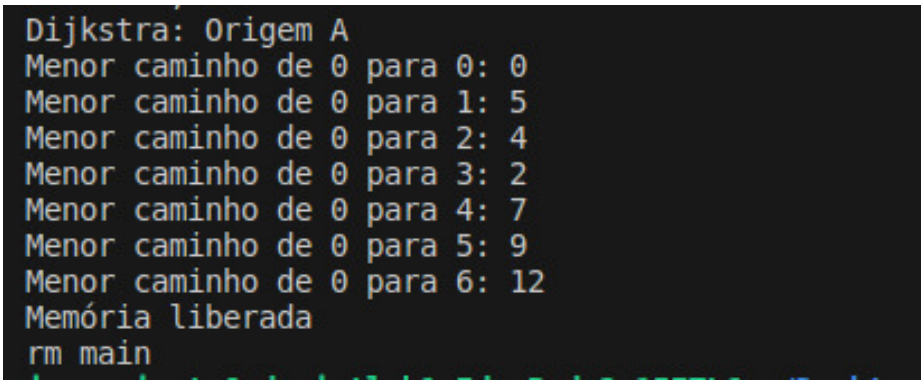
```
Memória liberada
==17107==
==17107== HEAP SUMMARY:
==17107==     in use at exit: 0 bytes in 0 blocks
==17107==   total heap usage: 44 allocs, 44 frees, 1,912 bytes allocated
==17107==
==17107== All heap blocks were freed -- no leaks are possible
==17107==
==17107== For lists of detected and suppressed errors, rerun with: -s
==17107== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

## 4 Observações e Conclusões

O trabalho encontra-se funcional, eu acredito. O percurso de Depth First Search foi feito levando em conta a ordenação dos vértices adjacentes. Não tenho certeza se é exatamente o correto, mas segui a lógica de utilizar uma estrutura de pilha, além de seguir a ordem falada em sala. Para contornar o problema de não poder acessar adjacentes por peso, utilizei um ponteiro de função para determinar duas funções distintas de comparação.


Grafo criado e alocado com sucesso, com 7 vértices
Vértice: 0 A
Aresta 0, com vértice 3, peso 2
Aresta 1, com vértice 2, peso 4
Aresta 2, com vértice 1, peso 5
Vértice: 1 B
Aresta 0, com vértice 0, peso 5
Aresta 1, com vértice 2, peso 6
Aresta 2, com vértice 4, peso 6
Aresta 3, com vértice 6, peso 9
Vértice: 2 C
Aresta 0, com vértice 3, peso 3
Aresta 1, com vértice 0, peso 4
Aresta 2, com vértice 4, peso 4
Aresta 3, com vértice 1, peso 6
Vértice: 3 D
Aresta 0, com vértice 0, peso 2
Aresta 1, com vértice 2, peso 3
Aresta 2, com vértice 4, peso 5
Aresta 3, com vértice 5, peso 9

```
Vértice: 4 E
Aresta 0, com vértice 5, peso 2
Aresta 1, com vértice 2, peso 4
Aresta 2, com vértice 3, peso 5
Aresta 3, com vértice 1, peso 6
Aresta 4, com vértice 6, peso 6
Vértice: 5 F
Aresta 0, com vértice 4, peso 2
Aresta 1, com vértice 6, peso 3
Aresta 2, com vértice 3, peso 9
Vértice: 6 H
Aresta 0, com vértice 5, peso 3
Aresta 1, com vértice 4, peso 6
Aresta 2, com vértice 1, peso 9
DFS: Origem A
Chave: 0, Valor: A
Chave: 1, Valor: B
Chave: 4, Valor: E
Chave: 5, Valor: F
Chave: 6, Valor: H
Chave: 2, Valor: C
Chave: 3, Valor: D
```


Dijkstra: Origem A
Menor caminho de 0 para 0: 0
Menor caminho de 0 para 1: 5
Menor caminho de 0 para 2: 4
Menor caminho de 0 para 3: 2
Menor caminho de 0 para 4: 7
Menor caminho de 0 para 5: 9
Menor caminho de 0 para 6: 12
Memória liberada
rm main