João Ricardo Malta De Oliveira - 2112714
Vinícius Machado Da Rocha Viana - 2111343

Definição de Estruturas:

Nesta seção, são definidas duas estruturas: NO para representar um nó da lista de adjacência, contendo informações sobre o vértice, o peso da aresta e um ponteiro para o próximo nó na lista; e Grafo para representar o grafo, com informações sobre o número total de vértices, o número de arestas e uma matriz de ponteiros para listas de adjacência

```c
typedef struct no NO;
struct no {
  int no;
  float peso;
  NO *prox;
};

typedef struct grafo Grafo;
struct grafo {
  int numNo; /* numero de nos do grafo */
  int na;    /* numero de arestas */
  NO **no;   /* viz[i] aponta para a lista de arestas incidindo em i */
};
```

Declaração das estruturas de nó e de grafo

Função para Inicializar o Grafo (criarGrafo):

A função criarGrafo aloca dinamicamente memória para a estrutura Grafo e para a matriz de listas de adjacência. Os campos da estrutura são inicializados com valores iniciais, incluindo o número total de vértices e o número de arestas, ambos inicializados como zero.

```c
// Função para inicializar o grafo
Grafo *criarGrafo(int numNo) {
  Grafo *grafo = (Grafo *)malloc(sizeof(Grafo));
  if (grafo == NULL) {
    printf("Erro ao alocar memoria para o grafo\n");
    exit(1);
  }
  grafo->numNo = numNo;
  grafo->na = 0;
  grafo->no = (NO **)malloc(numNo * sizeof(NO *));
  if (grafo->no == NULL) {
    printf("Erro ao alocar memoria de inicializacao do nó");
    exit(1);
  }
  return grafo;
}
```

Função de criação do grafo

Função para Adicionar Arestas (adicionarAresta):

A função adicionarAresta adiciona uma aresta ao grafo. Ela aloca dinamicamente memória para um novo nó NO e atribui os valores do vértice de destino (fim), o peso da aresta e o ponteiro para o próximo nó na lista de adjacência. O novo nó é então inserido no início da lista de adjacência do vértice de início (ini) e o número total de arestas no grafo é incrementado de 1.

```c
// Função para adicionar uma aresta ao grafo
void adicionarAresta(Grafo *grafo, int ini, int fim, float peso) {
  NO *novoNo = (NO *)malloc(sizeof(NO));
  if (novoNo == NULL) {
    printf("Erro ao alocar memoria para o novo No\n");
    exit(1);
  }
  novoNo->no = fim;
  novoNo->peso = peso;
  novoNo->prox = grafo->no[ini];
  grafo->no[ini] = novoNo;
  grafo->na++;
}
```

Função de adicionar arestas

Função de Busca em Profundidade (DFS):

No início da função, o. Em seguida, a função itera sobre os vizinhos não visitados desse nó, acessando a lista de adjacência correspondente. Durante essa iteração, a função verifica se o vizinho já foi visitado. Se não foi, imprime uma mensagem indicando uma aresta não visitada e chama recursivamente a função dfs para explorar esse vizinho. Se o vizinho já foi visitado, a função imprime uma mensagem indicando que a aresta foi visitada.
O processo é repetido para todos os vizinhos do nó atual, utilizando a recursividade para explorar em profundidade. As mensagens de console geradas durante a execução fornecem informações sobre os nós visitados e o estado das arestas durante o percurso.
Essa implementação é modular e utiliza eficientemente a recursividade para percorrer o grafo em profundidade. A busca em profundidade é uma técnica fundamental em grafos, sendo aplicada em várias áreas, como roteamento de redes e análise de conectividade em sistemas complexos. A estrutura clara do código contribui para uma compreensão sólida do algoritmo e pode ser adaptada para diferentes contextos de aplicação.

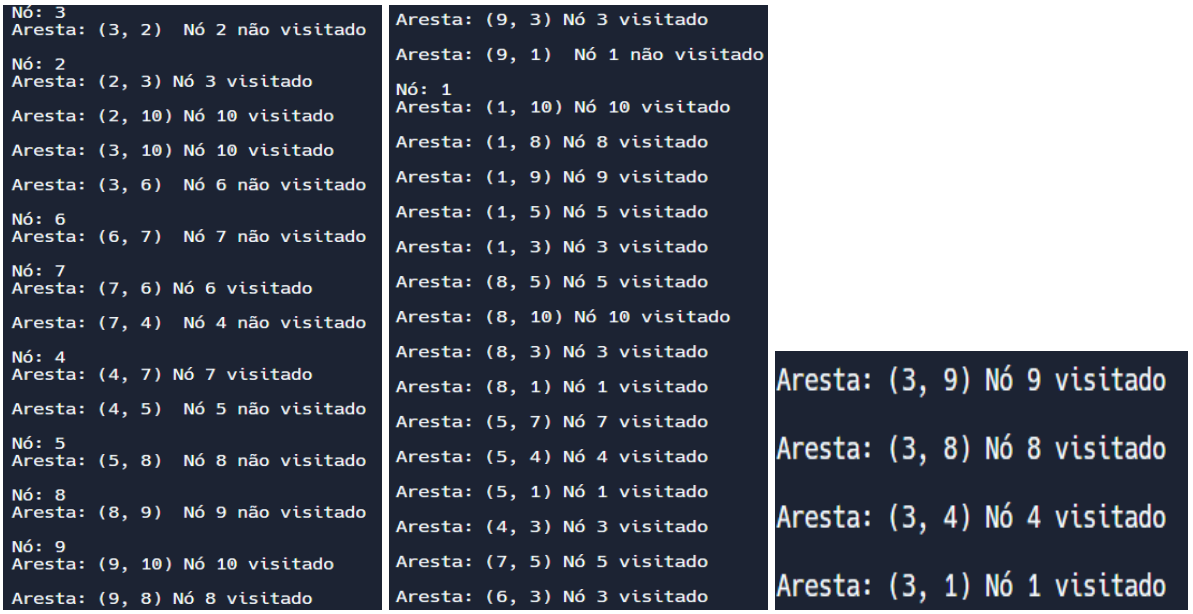
```c
void dfs(Grafo* grafo, int no_inicial, int* visitado)
{
  visitado[no_inicial] = 1;
  printf("Nó: %d\n", no_inicial);

  NO* atual = grafo->viz[no_inicial];
  while (atual)
  {
    if (!visitado[atual->no])
    {
      printf("Aresta: (%d, %d)  Nó %d não visitado\n\n", no_inicial, atual->no,
atual->no);
      dfs(grafo, atual->no, visitado);
    }
    else
    {
      printf("Aresta: (%d, %d) Nó %d visitado\n\n", no_inicial, atual->no,
atual->no);
    }
    atual = atual->prox;
  }
}
```

Função de busca por profundidade

Saída da Busca em Profundidade (DFS):

A saída gerada pelo código de busca por profundidade exibe a saída a seguir:



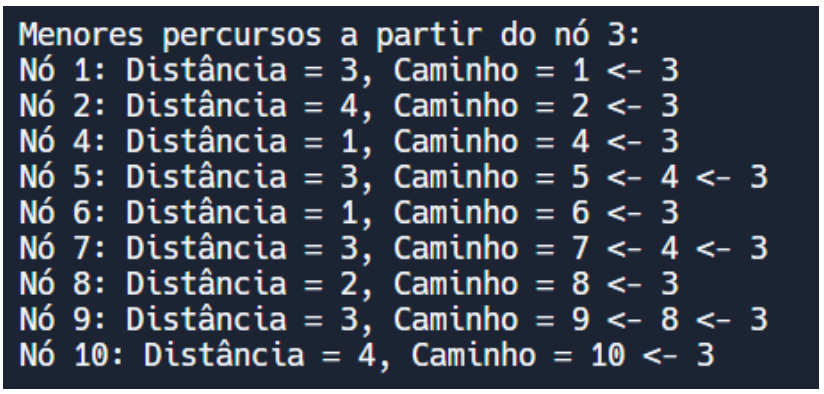
Função de menor percurso (dijkstra):

Primeiro declaramos 3 vetores dinamicamente alocados: distância, visitado e anterior. Esses vetores armazenam informações sobre a distância acumulada, o estado de visitado ou não e os nós anteriores no caminho mais curto.

Em seguida inicializamos os vetores com valores seus valores: as distâncias são inicializadas como infinito (INT_MAX), todos os nós são marcados como não visitados (0), e os nós anteriores são inicializados como -1.

Após isso utilizamos um for, onde a cada iteração o nó não visitado com a menor distância acumulada é escolhido. Esse nó é marcado como visitado, e as distâncias dos nós outros nós não visitados são atualizadas considerando o caminho mais curto até o momento.

Por fim, como foi pedido no trabalho, printamos os menores caminhos e exibimos a distância acumulada e o caminho mais curto para cada nó de destino a partir do nó de origem, o resultado obtido se encontra a seguir.



Saída do código de dijkstra

Função principal (main):

Para a função main primeiro criamos um grafo utilizando a função criarGrafo com todos os 10 nós (vértices). Em seguida adicionamos todos os pares de arestas, nos sentidos de ida e volta (3,1) (1,3) representando as conexões entre os nós, tendo cada aresta um peso associado a ela.

Após a construção do grafo, criamos um vetor visitado alocado dinamicamente e inicializamos com zeros. Esse vetor é utilizado para acompanhar quais nós foram visitados durante as operações de busca no grafo.

A função de busca em profundidade (dfs) é chamada com o grafo, o nó inicial (no caso, 3) e o vetor visitado. Em seguida, a função de Dijkstra (dijkstra) também é chamada, utilizando o mesmo nó inicial (3).

Por fim, realizamos a liberação das memórias alocadas dinamicamente. Isso é feito iterando sobre os nós do grafo, liberando as listas de adjacência associadas a cada nó. O vetor visitado, bem como a estrutura do grafo.