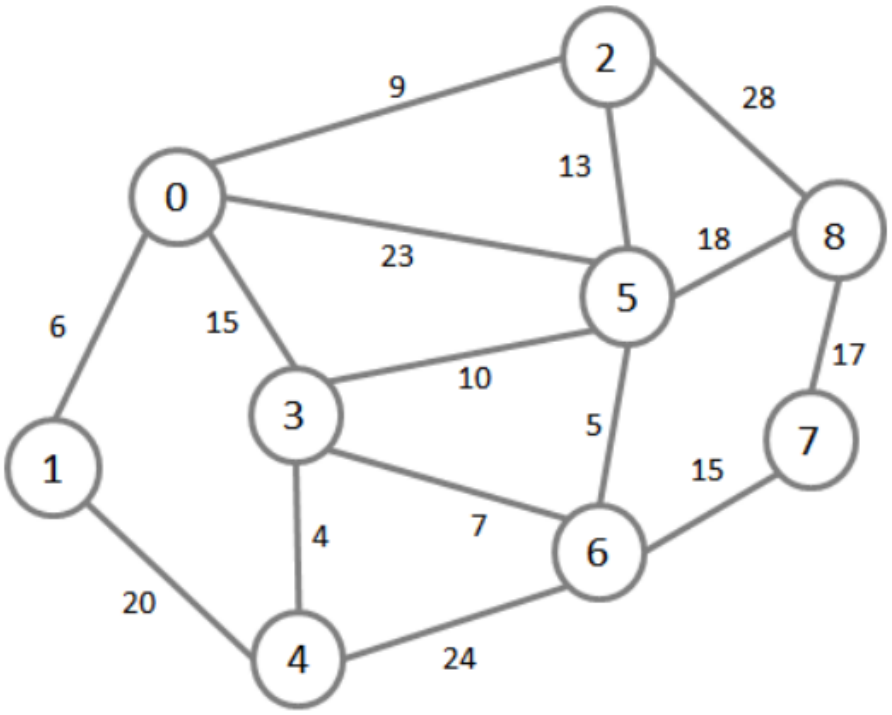
Relatório de INF1010 - Trabalho 5 (Grafos)

Gustavo Molina Soares (2020209)
Leo Land Bairros Lomardo (2020201)

Tarefa 5 e 6 de INF 1010 – EDA – 2022-2

1. Dado o grafo:



Represente o grafo acima usando lista de adjacências. Programe em linguagem C os seguintes algoritmos, para, a partir do vértice 0 (quando for o caso):

a) Calcular o caminho mais curto utilizando o algoritmo de Dijkstra.

b) Indicar uma árvore geradora de custo mínimo, usando o algoritmo de Kruskal.

c) Indicar um caminho de uma busca em profundidade.

d) Indicar um caminho de uma busca em amplitude.

Durante o último laboratório tivemos que criar um grafo com custos, então começando a implementação, criei structs de dados para representá-los.

Separei as funções e questões por header para ser de mais fácil manutenção e visualização do código.

```c
typedef struct vertex{
    int value;
    int cost;
    struct vertex *next;
}Vertex;

typedef struct graph{
    int num_vertex;
    int *visited;
    Vertex **adjList;
}Graph;
```

Inicialmente temos essas duas structs, uma que representa um ponto e o outro que representa o grafo, no ponto temos o valor e o custo e o próximo ponto, e no grafo temos o número de pontos, a lista de pontos visitados (usado posteriormente nos algoritmos de busca) e a lista de grafos.

Para a criação do grafo primeiramente criamos e alocamos tanto os pontos quanto o grafo.

```c
Vertex *createVertex(int value){
    Vertex *newVertex = malloc(sizeof(Vertex));
    newVertex->value = value;
    newVertex->cost = 0;
    newVertex->next = NULL;
    return newVertex;
}

Graph *createGraph(int numVertex){
    Graph *newGraph = malloc(sizeof(Graph));

    newGraph->num_vertex = numVertex;
    newGraph->adjList = malloc(numVertex * sizeof(Vertex));
    newGraph->visited = malloc(numVertex * sizeof(int));

    for (int i = 0; i < numVertex; i++){
        newGraph->adjList[i] = NULL;
        newGraph->visited[i] = 0;
```

```
    }
    return newGraph;
}
```

Aqui temos as funções que realizam isso respectivamente.

Depois criei uma função para adicionar conexão entre dois pontos.

```
void addConnections(Graph *graph, int s, int d, int cost){
    Vertex *newVertex = createVertex(d);
    newVertex->next = graph->adjList[s];
    newVertex->cost = cost;
    graph->adjList[s] = newVertex;

    newVertex = createVertex(s);
    newVertex->next = graph->adjList[d];
    newVertex->cost = cost;
    graph->adjList[d] = newVertex;
}
```

Aqui podemos ver que ele pega um ponto, e faz eles serem adjacentes, adicionando um custo entre si.

Para visualização, criei uma função de print, que pode ser usada em outras questões, mas não é necessária.

```
void printGraph(Graph* graph) {
  int v;
  for (v = 0; v < graph->num_vertex; v++) {
    Vertex* temp = graph->adjList[v];
    while (temp) {
      if (temp->next == NULL){
        printf(" (%d -> %d) weight: %d", v, temp->value, temp->cost);
      }
      else{
        printf(" (%d -> %d) weight: %d |", v, temp->value, temp->cost);
      }
      temp = temp->next;
    }
    printf("\n");
  }
```

```
}
```

Na letra a, criei o header djistrika.h, que vai armazenar as funções para realizar o djistrika, que são duas, respectivamente, minDistance e djistrika.

```
int minDistance(int *dist, bool *isPath){
  int min = INT_MAX;
  int min_index;

  for (int i = 0; i < numNodes; i++){
    if (isPath[i] == false && dist[i] <= min){
      min = dist[i];
      min_index = i;
    }
  }

  return min_index;
}
```

minDistance é uma função que verifica a menor distância entre os pontos selecionados, e devolve o index.

```
void djistrikaPath(int startNode, Graph *graph, int endNode){
  int dist[numNodes];

  bool isPath[numNodes];

  for (int i = 0; i < numNodes; i++){
    dist[i] = INT_MAX;
    isPath[i] = false;
  }

  dist[startNode] = 0;

  for (int j = 0; j < numNodes - 1; j++){
    int aux = minDistance(dist, isPath);

    isPath[aux] = true;

    for (int k = 0; k < numNodes; k++){
```

```
      Vertex *temp = graph->adjList[aux];
      while(temp){
        if ((dist[aux] + temp->cost < dist[k]) && temp->value == k){
            dist[k] =  dist[aux] + temp->cost;
        }
        temp = temp->next;
      }

    }
  }
  printf("The shortest path between the starting node %d and the ending
node %d is costing %d\n", startNode, endNode, dist[endNode]);
}
```

E a função djistrika devolve o custo entre os dois pontos selecionados, onde ela basicamente vai pegar o ponto inicial e vai calcular os custos entre qualquer ponto do grafo, depois é só selecionar o ponto final que ele irá devolver o custo do caminho.

OBS: Acredito que a implementação do Djistrika acabou sendo mais simples, já que já fizemos a matéria Inteligência Artificial INF1771, então isso ajudou bastante nessa parte.

Na letra b criamos o header kruskal.h, onde minha dupla acabou realizando a sua própria implementação de como criar um grafo, então ficou duas implementações de grafo.

```
typedef struct Edge{
  int src;
  int dest;
  int cost;
  struct Edge *next;
}Edge;

Edge *createEdgeList(Graph *grafo);
Edge* insertEdge(Edge* conectionList, int src, int dest, int cost);
Edge* createEdge(int src, int dest, int cost);
```

Decidimos manter porque como já tinha sido feito e testado, acabamos deixando assim já que só vai ser utilizado na função do kruskal, mas podemos ver que ele usa a struct de grafo, apenas criou uma nova struct para os pontos do grafo, adicionando o parâmetro destino.

```
void kruskal(Edge* conectionList, Graph *kruskalTree){
  Edge *auxEdge = conectionList;
  int vRep[numNodes];
  int x, y;

  sentNegative(vRep, kruskalTree->num_vertex);

  while(auxEdge){
    x = find(vRep, auxEdge->src);
    y = find(vRep, auxEdge->dest);

    if (x != y){
      Union(vRep, x, y);
      addConnections(kruskalTree, auxEdge->src, auxEdge->dest,
auxEdge->cost);
    }
    auxEdge = auxEdge->next;
  }
}
```

Aqui geramos a árvore miníma de kruskal.

```
void sentNegative(int *vRep, int numVertice){
  for(int i = 0; i <numVertice; i++){
    vRep[i] = -1;
  }
}
```

A função sentNegative preenche o vetor com números negativos.

```
int find(int vRep[], int i){
  if (vRep[i] == -1){
    return i;
  }
  return find(vRep, vRep[i]);
}
```

A função find acha um vetor negativo, para verificar se ele foi visitado.

```
void Union(int vRep[], int x, int y){
  vRep[x] = y;
```

```
}
```

A função Union vai unir dois pontos do vetor.

E a addConnections foi explicada anteriormente, ela vai unir dois pontos no grafo.

Agora partiremos para o item c, onde vamos realizar o DFS.

```
void DFS( Graph* graph, int src) {
  Vertex* temp = graph->adjList[src];

  graph->visited[src] = 1;
  printf("Visited:%d\n", src);

  while (temp != NULL) {
    int connectedVertex = temp->value;

    if (graph->visited[connectedVertex] == 0) {
      DFS(graph, connectedVertex);
    }
    temp = temp->next;
  }
}
```

A DFS foi bem simples, apenas usamos o vetor de visitas do grafo, e vamos visitar tudo que não foi visitado anteriormente, e ali printamos a visita atual.

E o último item, o d, precisamos implementar algumas coisas a mais, primeiramente um sistema de fila, usando o arquivo queue.h

Onde eu terei funções básicas de uma fila, adicionar na fila, remover da fila, verificar se ela está vazia e outros.

```
typedef struct queue{
  int items[numNodes];
  int front;
  int rear;
}Queue;

Queue *createQueue();
void enqueue(Queue *q, int src);
int dequeue(Queue *q);
```

```c
void display(Queue *q);
int isEmpty(Queue *q);
void printQueue(Queue *q);

Queue* createQueue() {
  Queue* q = malloc(sizeof(Queue));
  q->front = -1;
  q->rear = -1;
  return q;
}

int isEmpty(Queue* q) {
  if (q->rear == -1)
    return 1;
  else
    return 0;
}

void enqueue(Queue* q, int value) {
  if (q->rear == numNodes - 1)
    printf("\nQueue is Full!!");
  else {
    if (q->front == -1)
      q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
  }
}

int dequeue(Queue* q) {
  int item;
  if (isEmpty(q)) {
    printf("Queue is empty");
    item = -1;
  } else {
    item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
      q->front = q->rear = -1;
    }
```

```
  }
  return item;
}

void printQueue(Queue* q) {
  int i = q->front;

  if (isEmpty(q)) {
    printf("Queue is empty");
  } else {
    printf("\nQueue contains: ");
    for (i = q->front; i < q->rear + 1; i++) {
      printf("%d ", q->items[i]);
    }
    printf("\n");
  }
}
```

Uso isso no BFS:

```
void BFS(Graph *graph, int src) {

    Queue *q = createQueue();


    graph->visited[src] = 1;
    enqueue(q, src);

    while(!isEmpty(q)) {
      printQueue(q);
      int curr = dequeue(q);
      printf("Visited: %d\n", curr);

      Vertex *temp = graph->adjList[curr];

      while(temp){
        int adjVertex = temp->value;
```

```
        if (graph->visited[adjVertex] == 0) {
          graph->visited[adjVertex] = 1;
          enqueue(q, adjVertex);
        }
        temp = temp->next;
      }
    }


}
```

Aqui vou adicionar na fila todos os vértices adjacentes, e depois vou acessando eles na fila, assim que acesso, retiro da fila e adiciono os adjacentes do vértice atual.

Em geral esse laboratório foi bem tranquilo, trabalhar com algoritmos de busca não apresenta nenhuma grande dificuldade para nenhum de nós dois, mas é bem legal implementar, desde que tivemos a experiência com a outra matéria, que implementamos A* e outros algoritmos de busca, tive bastante interesse em desenvolver mais e pesquisar mais sobre.