INF1010 - ESTRUTURAS DE DADOS AVANÇADAS - 2022.1 - 3WB

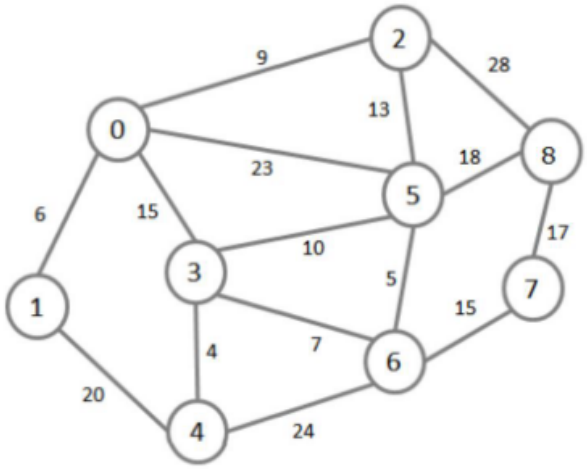
Lab5 - Grafos

Nome: Eric Leão Matrícula: 2110694
Nome: Marina Schuler Martins Matrícula: 2110075

Relatório do Laboratório

(Enunciado) Gerar relatório de seu trabalho contendo o grafo de entrada para o algoritmo, a lista de adjacências inicializada em seu programa e a saída gerada para o grafo indicado. Entregar o código-fonte do algoritmo em linguagem C e o relatório em pdf.

1. **Grafo de entrada para o algoritmo**



2. **Lista de adjacências inicializada no programa**

(0-1=6) -> (0-2=9) -> (0-3=15) -> (0-5=23) -> NULL
(1-0=6) -> (1-4=20) -> NULL
(2-0=9) -> (2-5=13) -> (2-8=28) -> NULL
(3-0=15) -> (3-4=4) -> (3-5=10) -> (3-6=7) -> NULL
(4-1=20) -> (4-3=4) -> (4-6=24) -> NULL
(5-0=23) -> (5-2=13) -> (5-3=10) -> (5-6=5) -> (5-8=18) -> NULL
(6-3=7) -> (6-4=24) -> (6-5=5) -> (6-7=15) -> NULL
(7-6=15) -> (7-8=17) -> NULL
(8-2=28) -> (8-5=18) -> (8-7=17) -> NULL

3. **Saída gerada para o grafo indicado (Árvore Geradora Mínima)**

vertice 3 com o vertice 4 com aresta 4 esta na agm
vertice 5 com o vertice 6 com aresta 5 esta na agm

vertice 0 com o vertice 1 com aresta 6 esta na agm
vertice 3 com o vertice 6 com aresta 7 esta na agm
vertice 0 com o vertice 2 com aresta 9 esta na agm
vertice 2 com o vertice 5 com aresta 13 esta na agm
vertice 6 com o vertice 7 com aresta 15 esta na agm
vertice 7 com o vertice 8 com aresta 17 esta na agm

4. **Código fonte em C**

```c
/*
INF1010 - Estrutura de Dados - 2022.1 - 3WB - Prof. Luis Fernando Seibel
Laboratório 5 - Grafos
Nome: Eric Leao     Matrícula: 2110694
Nome: Marina Schuler Martins    Matrícula: 2110075
*/

#define MAXN 9
#define MAXE 15
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

/* estruturas */
typedef struct Node {
  int vertex;
  int weight;
  struct Node *next;
} node;

typedef struct List {
  node *head;
} list;

typedef struct MInheapnode {
  int noi;
  int noj;
  int edge;
} minheapnode;

typedef struct Forestnode {
  int value;
  struct Forestnode *parent;
  int rank;
} forestnode;
```

```c
/* protótipos TAD */
forestnode *MakeSet(int value);
forestnode *_union(forestnode *node1, forestnode *node2);
forestnode *find(forestnode *node);
void addNode(int s, int t, int w);
void printlist();
void up(minheapnode vet[], int i, int tam);
void makeheap(minheapnode vet[], int l);
void down(minheapnode vet[], int i, int tam);
void libera_lista(node *no);

/* lista global */
list *adj[MAXN];

int main(void) {
  /* criando o vetor para lista de adjacências */
  for (int c = 0; c < MAXN; c++) {
    adj[c] = (list *)malloc(sizeof(list));
    assert(adj[c]);
    adj[c]->head = NULL;
  }
  /* inicializando a lista de adjacências */
  {
    addNode(0, 1, 6);
    addNode(0, 2, 9);
    addNode(0, 3, 15);
    addNode(0, 5, 23);
    addNode(1, 0, 6);
    addNode(1, 4, 20);
    addNode(2, 0, 9);
    addNode(2, 5, 13);
    addNode(2, 8, 28);
    addNode(3, 0, 15);
    addNode(3, 4, 4);
    addNode(3, 5, 10);
    addNode(4, 1, 20);
    addNode(3, 6, 7);
    addNode(4, 3, 4);
    addNode(4, 6, 24);
    addNode(5, 2, 13);
    addNode(5, 0, 23);
    addNode(5, 6, 5);
```

```
    addNode(5, 8, 18);
    addNode(5, 3, 10);
    addNode(6, 3, 7);
    addNode(6, 4, 24);
    addNode(6, 5, 5);
    addNode(6, 7, 15);
    addNode(7, 6, 15);
    addNode(7, 8, 17);
    addNode(8, 2, 28);
    addNode(8, 5, 18);
    addNode(8, 7, 17);
  }
  printf("Lista inicializada:\n");
  printlist();

  /* construindo o algoritmo de Kruskal para obter uma árvore geradora
mínima
usando union-find */
  minheapnode vet[MAXE];
  makeheap(vet, MAXE);
  forestnode *sets[MAXN];
  for (int c = 0; c < MAXN; c++) {
    sets[c] = MakeSet(c);
  }
  printf("Encontrando Árvore Geradora Mínima:\n");
  for (int c = 0; c < MAXE; c++) {
    /* obtenção das arestas via union-find */
    int a = find(sets[vet[0].noi])->value;
    int b = find(sets[vet[0].noj])->value;
    if (a != b) {
      /* representação da árvore geradora mínima indicando as arestas que a
compõem */
      printf("vertice %d com o vertice %d com aresta %d esta na agm\n",
             vet[0].noi, vet[0].noj, vet[0].edge);
      _union(find(sets[vet[0].noi]), find(sets[vet[0].noj]));
    }
    minheapnode temp = vet[MAXE - 1 - c];
    vet[MAXE - 1 - c] = vet[0];
    vet[0] = temp;
    down(vet, 0, MAXE - 1 - c);
  }
  /* liberando memória alocada */
      for (int c = 0; c < MAXN; c++) {
```

```
            libera_lista(adj[c]->head);
    free(adj[c]);
            free(sets[c]);
  }
  return 0;
}

/* funções utilizadas */
void printlist() {
  for (int c = 0; c < MAXN; c++) {
    node *n = adj[c]->head;
    while (n) {
      printf("(%d-%d=%d) -> ", c, n->vertex, n->weight);
      n = n->next;
    }
    printf("NULL \n");
  }
}

void addNode(int s, int t, int w) {
  node *newV = (node *)malloc(sizeof(node));
  assert(newV);
  newV->vertex = t;
  newV->weight = w;
  newV->next = NULL;
  if (adj[s]->head == NULL) {
    adj[s]->head = newV;
  } else {
    node *temp = adj[s]->head;
    while (temp->next != NULL && temp->next->vertex < newV->vertex) {
      temp = temp->next;
    }
    if (newV->vertex > temp->vertex) {
      newV->next = temp->next;
      temp->next = newV;
    } else {
      newV->next = temp;
      adj[s]->head = newV;
    }
  }
  return;
}
```

```
forestnode *MakeSet(int value) {
  forestnode *node = malloc(sizeof(forestnode));
  node->value = value;
  node->parent = NULL;
  node->rank = 0;
  return node;
}

forestnode *_union(forestnode *node1, forestnode *node2) {
  if (node2->rank > node1->rank) {
    node1->parent = node2;
    return node2;
  } else if (node1->rank > node2->rank) {
    node2->parent = node1;
  } else {
    node2->parent = node1;
    node1->rank++;
  }
  return node1;
}

forestnode *find(forestnode *node) {
  forestnode *root = node;
  while (root->parent != NULL) {
    root = root->parent;
  }
  return root;
}

void makeheap(minheapnode vet[], int l) {
  int aux = 0;
  for (int c = 0; c < MAXN; c++) {
    node *n = adj[c]->head;
    while (n) {
      if (c <= n->vertex) {
        vet[aux].noi = c;
        vet[aux].noj = n->vertex;
        vet[aux].edge = n->weight;
        up(vet, aux, l);
        aux++;
      }
      n = n->next;
    }
```

```
  }
}

void up(minheapnode vet[], int i, int tam) {
  int j;
  minheapnode aux;
  j = (i - 1) / 2;
  if (j >= 0)
    if (vet[i].edge < vet[j].edge) {
      aux = vet[i];
      vet[i] = vet[j];
      vet[j] = aux;
      up(vet, j, tam);
    }
}

void down(minheapnode vet[], int i, int tam) {
  int j;
  minheapnode aux;
  j = (2 * i) + 1;
  int temp = i;
  if (j < tam)
    if (vet[j].edge < vet[temp].edge)
      temp = j;
  if (j + 1 < tam)
    if (vet[j + 1].edge < vet[temp].edge)
      temp = j + 1;
  if (i != temp) {
    aux = vet[i];
    vet[i] = vet[temp];
    vet[temp] = aux;
    down(vet, temp, tam);
  }
}

void libera_lista(node * no) {
      if (no == NULL) return;
      node * aux = no->next;
  free(no);
  libera_lista(aux);
}
```