Prova 3

Data Criado	@September 5, 2022 2:55 PM
Tipo	
Finalizado	☑
Fazer em	

Prova 3 - Grafos

Introdução

- Grafos **armazenam relações**, é um conjunto de vértices e arestas.
- G = (V, E)
- **Vértices** são interseções que geram diferentes caminhos/**arestas**. Numerados de 0 a V-1.
- É possível calcular a **distância** entre **vértices** a partir das **arestas**.
- Um vértice não precisa de arestas para ser um grafo. Por outro lado, uma aresta precisa de dois vértices para existir.
- **Grafo completo:** todos os vértices tem arestas se conectando.
- **Caminho**: sequência de vértices em que cada vértice sucessivo é adjacente ao predecessor. **Ciclo:** primeiro e último vértice são os mesmos.
- Um grafo com V vértices pode ter no máximo V*(V-1)/2 arestas.
- **Grafo conexo:** há um caminho de cada vértice para outro vértice no grafo. **Componentes conexas:** grafo com grafos menores conexos.

Implementações

- **Matriz de adjacência**: índice é um vértice e o elemento é uma aresta (1) ou não (0). A linha está conectada com a coluna. Custo inicial de 16 bytes (V = 4, E = 4, **adj = 8).
- **Lista de adjacência:** ao adicionar uma nova aresta, existe a premissa de que ela não existe, não se verifica toda a lista. Listas encadeadas com as arestas sendo novos elementos na chave do vértice.
- **Edge (v, w):** ligação entre dois vértices.
- **IMPLEMENTAR MATRIZ DE ADJACÊNCIA:**

```
// Estrutura do grafo
typedef struct graph{
    int V;
    int E;
    int **adj;
} graph;

// Estrutura da Edge
typedef struct Edge{
    int u;
    int w;
} Edge;

// Retorna Edge
Edge EDGE(int u, int w){
    return (Edge){u, w};
}

// Inicializa matriz
```

```
int **MATRIXinit(int V, int init){
    int **matrix = malloc(V*sizeof(int*));

    for(int i = 0; i < V; i++){
        matrix[i] = malloc(V*sizeof(int));
    }

    for(int i = 0; i < V; i++){
        for(int j = 0; j < V; j++){
            matrix[i][j] = init;
        }
    }

    return matrix;
}

// Inicializa grafo
graph *graphInit(int V){
    graph *g = malloc(sizeof(graph));
    g->V = V;
    g->E = 0;
    g->adj = MATRIXinit(V, 0);

    return g;
}

void graphInsertE(graph *G, Edge e){
    int u = e.u, w = e.w;

    if(G->adj[u][w] == 0){
        G->E++;
    }

    G->adj[u][w] = 1;
    G->adj[w][u] = 1;

    return;
}

void graphRemoveE(graph *G, Edge e){
    int u = e.u, w = e.w;

    if(G->adj[u][w] == 1){
        G->E--;
    }

    G->adj[u][w] = 0;
    G->adj[w][u] = 0;

    return;
}

int graphEdges(Edge a[], Graph *G){
  int v, w, E = 0;

  for(v = 0; v < G->V; v++){
    for(w = v+1; w < G->V; w++){ //como é v+1, diminui o custo n²
      if(G->adj[v][w] == 1)
        //pode imprimir também as arestas, se quiser
        a[E++] = EDGE(v, w);

  return E;
}
```

- **IMPLEMENTAR LISTA DE ADJACÊNCIAS:**

```
typedef struct no *link;
struct no{
  int v;
  link prox;
} no;

typedef struct graph{
  int V, E;
  link *adj;
} graph;
```

```
link new(int v, link prox){
  link x = malloc(sizeof*link);

  x->v = v;
  x-> prox = prox;

  return x;
}

graph graphInit(int v){
  graph G = malloc(sizeof*graph);
  G-> V = v;
  G-> E = 0;
  G->adj = malloc(V*sizeof(link));

  for(int i = 0; i < v; i++)
    G->adj[i] = NULL;

  return G;
}

// adiciona no início da lista
void graphInsertE(graph G, Edge E){
  int v = E.v, w = E.w;

  G->adj[v] = new(w, G->adj[v]);
  G->adj[w] = new(v, G->adj[w]);

  G->E++;

  return;
}

int graphEdges(Edge a[], Graph *G){
  int v, E = 0;
  link t;

  for(v = 0; v < G->V; v++){
    for(t = G->adj[v]; t != NULL; t=t->prox){
      if(v < t->V)
        a[E++] = EDGE(v, t->V);
    }
  }

  return E;
}
```

Custos

- Matriz pode gastar **MENOS MÉMORIA**, uma vez que só precisa gastar um char (0 ou 1) para representar relação.
- Vetor de arestas é o retornado na função GraphEdges.

	Vetor de Arestas	Matriz Adj.	Lista Adj.
Espaço (Memória)	E	V^2	V+E
Inicializar	1	V^2	V
Copy	E	V^2	E
Destruir	1	V (free em cada V)	E (free em todas as E)
InsertE	1	1	1
Encontrar/RemoverE	E	1	V (ver todas as arestas de cada V)
V isolado?	E	V	1 (só ver NULL)
Caminho de u para v	E*lgV	V^2	V+E

- Grafo quase ou completa: lista de adjacências tem espaço quase igual ao da matriz.

Buscas

DFS (Depth = Profundidade):

- Vértice em vértice para visitar os que ainda não foram visitados (vai visitando até o final da conexidade). → Continuidade.
- Permite saber quantos componentes conexos e se um grafo é complementamente conexo.
- Similar à pilha.
- **Implementação:**

```
// Uma chamada sem o componentes conexos, fica sabendo quais vértices são conexos, começando por um vértice.
void dfsR_Matriz(graph *G, int v, int u){ // V de onde eu vim e W onde estou. Pode ser uma aresta, também
    pre[u] = 1;

    for(v = 0; v < G->V; v++){
        if(pre[v] == 0 && G->adj[u][v] == 1){
            // tamComp ++; -> Calcular o tamanho da componente atual (Var Global)
            dfsR_Matriz(G, u, v);
        }
    }

    return;
}

void dfsR_Lista(graph *G, int v, int u){ // V de onde eu vim e W onde estou
    vis[u] = 1;

    no *aux = G->adj[u].prox;

    while(aux != NULL){
      if(pre[aux->V] == -1)
        dfsR(G, w, V);
      aux = aux -> prox;
    }

    return;
}

int vis[5000];

// BUSCA GERAL
int componentesConexos(graph *G){
    int conexos = 0;

    memset(pre, 0, sizeof(int)*5000); // ou - 1

    for(int i = 0; i < G->V; i++){ // visita todos os vértices, até de desconexos
        if(vis[i] == 0){ // ou - 1
            // tamComp = 1;
            dfsR(G, i, i);
            // Componentes[conexos] = tamComp; -> vai registrando o tamanho de cada componente no grafo
            // Salva em Componentes o tamanho e o vértice de início dessa componente
            conexos++;
        }
    }

    return conexos;
}
```

BFS (Largura):

- Vê todas as arestas de um vértice antes de passar para o próximo.
- Gastos: V^2 em matriz e V*E em lista de adjacência.
- Ideal para encontrar o menor caminho/distância.
- Uso com fila (não recursiva).
- **Implementação:**

```
void bfsR_Matriz(graph *G, Edge E){
  filaInit(f);
  enfileira(E);

  while(!filaVazia(f)){
    E = desenfileira(f);
    v = E.v, w = E.w, pre[w] = v;
    for(int i = 0; i < G->V; i++){
      if(G->adj[w][i] == 1 && pre[i] == 0){
        enfileira(EDGE(w, i));
      }
    }
}

int pre[5000];

void bfsR_Matriz(graph *G, Edge E){
  filaInit(f);
  enfileira(E);

  while(!filaVazia(f)){
    E = desenfileira(f);
    v = E.v, w = E.w, pre[w] = 1;;
    for(l = G->adj[w]; l != NULL; l = l -> prox){ // todas as arestas conectadas ao vértice
      if(pre[l->V] == 0){
        enfileira(EDGE(w, i));
        pre[l->V] = 1;
      }
    }
}

// busca geral
int componentesConexos ou GraphSearch(graph *G){
    int conexos = 0;

    memset(vis, 0, sizeof(int)*5000);

    for(int i = 0; i < G->V; i++){
        if(vis[i] == 0){
            bfsR(G, i, i);
            conexos++;
        }
    }

    return conexos;
}

// BFS para BUSCA DE CAMINHO MÍNIMO
void graphSearch(graph *G, int s, int *dist){
    int INFINITY = G->V;

    for(int i = 0; i < G->V; i++){
        dist[i] = INFINITY;
    }

    dist[s] = 0;

    Fila *f;
    criarFila(f, G->V);
    inserir(f, s);

    while(!estaVazia(f)){
        int v = remover(f);
        for(link a = G->adj[v]; a != NULL; a = a->prox){
            int w = a->v;
            if(dist[w] == INFINITY){
                dist[w] = dist[v]+1;
                inserir(f, w);
            }
        }
    }
}
```

Grafos dirigidos

- Conjunto de vértices e arestas dirigidas (vão apenas em uma direção). Uma aresta vai DE um vértice PARA outro vértice.
- **Caminho dirigido:** é uma lista de vértices no qual existe uma aresta dirigida conectando cada vértice da lista a seu sucessor. T é alcançável de S se existe um caminho dirigido de S para T.
- Quantidade máxima de arestas: 2^V²
- A diferença na implementação é que não se coloca sempre que os dois vértices se conectam.
- **Grafo dirigido Acíclico (DAG):** implementações de árvores são feitas dessa forma. Não há ciclos.
- **Dirigido fortemente conexo**: se todos os vértices são alcançáveis a partir de todos os vértices.

Alcançabilidade e Fecho Transitivo

- **Fecho transitivo:** é um grafo dirigido com os mesmos vértices, mas com uma aresta de S a T no fecho transitivo, se existe um caminho dirigido de S a T no grafo dirigido. Basicamente, se S chega a T passando por Y, no fecho transitivo, faz-se uma ligação direta de S a T.
- Bom para um caso em que seja necessário ficar fazendo muitas dsfR ou bfsR, porque fazer o fecho transitivo é caro, mas após ter feito, saber se chega é constante.
- **Floyd Warshall** (cria o grafo transitivo, cúbico, caro):

```
void Graphtc(Graph G){
  int i, t;
  G -> tc = MATRIXInit(G->V, G->V, 0); // coloca na estrutura do grafo o tc
  for(int s = 0; s < G->V; s++){
    for(t = 0; t < G->V; t++){
      G->tc[s][t] = G->adj[s][t]; // copia todos os vértices
    }
  }

  for(int s = 0; s < G->V; s++) G->t[s][s]=1; // vértice chega nele mesmo

  for(i = 0; i < G->V; i++){
    for(int s = 0; s < G->V; s++){
      if(G->tc[s][i]==1)
        for(t=0; t < G->V; t++)
          if(G->tc[i][t] == 1)
            G->tc[s][t]=1;
    }
  }
}

int GraphReach(Graph G, int S, int T){ //saber se chega S a T
  return G->tc[s][t];
}

// Pode usar esse algoritmo e rodar toda a matrix conferindo se adj[i][j] == 1, assim seria totalmente conexo
```

- Funciona em uma lógica de ir calculando quem chega em quem, e se S chega a T e Y chega a S, então Y chega a T.

- **Peso nas arestas**: custo de locomoção entre vértices (distância em km, por exemplo).
  - Implementar peso em **matrix de adj.:** coloca o peso na posição (ao invés de 1, apenas).
  - Implementar peso em **lista de adj.:** coloca peso dentro da struct de aresta.
- **Caminho barato:** buscar o caminho mais rápido/”barato” entre duas arestas. Esses algoritmos calcula para todos os pontos, não é para um vértice apenas (por isso é meio caro):
  - **Bellmon Ford (custos negativos podem gerar ciclos → sentinela resolve)**

```
// a->c = custo da aresta;
int GraphBF(Graph G, int origem, int *pa, int *dist){ // pa é o "pre" (ja visitados)
  int naFila[1000]; // pelo menos a quantidade de vértices
```

```
  for(int v = 0; v < G->V; v++){
    pa[v] = -1, dist[v] = INT_MAX, naFila[v]=0;
  }

  pa[origem] = origem; dist[origem] = 0;

  InicializaFila(G->E);

  Enfileira(origem);
  naFila[origem] = 1;
  Enfileira(sentinela); // sentinela é um número maior que a quantidade de vértices

  while(1){
    int v = Desenfileira();
    if(v < sentinela){
      for(link a = G->adj[v]; a != NULL; a = a -> prox){ // percorre todas as arestas de v
        if(dist[v] + a -> c < dist[a->v]){ // se a distância (custo) para chegar é menor que o tamanho da distância atual, atualiza
          dist[a->v] = dist[v] + a->c;
          pa[a->v] = v;

          if(naFila[a->v] == 0){
            Enfileira(a->v);
            naFila[a->v] = 1;
          }

        }
      }
    } else{
      if(filaVazia) return 1; // acabou

      if(++k >= G->V) return 0; //muitas sentinelas, problema e para

      Enfileira(sentinela);

      for(int t = 0; t < G->V; t++){
        naFila[t] = 0;
      }

    }
  }
}
```

- **Dijkstra (desempenho melhor, mas somente para pesos positivos)**

```
void GraphDikstra(Graph G, int origem, int *pa, int *dist){
  bool mature[1000]; // pelo menos a quantidade de vértices

  for(int v = 0; v < G->V; v++){
    pa[v] = -1, mature[v] = 0, dist[v] = INT_MAX;
  }

  pa[origem]=origem; dist[origem]=0;

  while(1){
    int min = INT_MAX;
    int y;

    // custo alto -> para melhorar, faz uma PQ (fila de prioridades)
    for(int z = 0; z < G->V; z++){
      if(mature[z]) continue; // já analisou o vértice todo
      if(dist[z] < min){
        min = dist[z], y = z;
      }
    }

    if(min == INT_MAX) break; // ou todos maduros, ou não alcançável pela origem

    for(link a = G->adj[y]; a != NULL; a = a -> prox){ //todos as arestas e compara a distância
      if(mature[a->v]) continue;

      if(dist[y] + a -> c < dist[a->v]){
        dist[a->v] = dist[y] + a -> c;
        pa[a->v] = y;
      }
    }
```

```
  }
}
```