UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE - UFRN
CENTRO DE ENSINO SUPERIOR DO SERIDÓ - CERES
BACHARELADO EM SISTEMAS DE INFORMAÇÃO - BSI

JHONATAS ISRAEL DA COSTA LAURENTINO

Algoritmos para Grafos

Um Relatório Sobre Os Alguns dos Principais Algoritmos para serem trabalhados com Grafos vistos em Estrutura de Dados

CAICÓ/RN
2019

JHONATAS ISRAEL DA COSTA LAURENTINO

Algoritmos para Grafos

Um Relatório Sobre Os Alguns dos Principais Algoritmos para serem trabalhados com Grafos vistos em Estrutura de Dados

Relatório entregue à matéria Estrutura de Dados, no curso de Bacharelado em Sistemas de Informação, matéria está ministrada pelo Prof. Dr. João Paulo de Souza Medeiros, na Universidade Federal do Rio Grande do Norte (UFRN).

CAICÓ/RN
2019

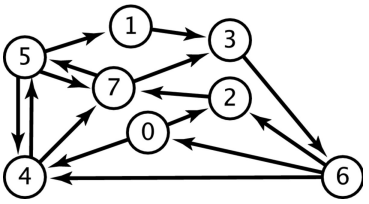
Resumo

O presente relatório tem como objetivo apresentar estudos coletados durante a segunda unidade da disciplina de estruturas de dados ministrada pelo professor Dr. João Paulo de Souza Medeiros na Universidade Federal do Rio Grande no Norte. Serão apresentados gráficos e informações acerca de alguns algoritmos de busca em representação por matriz de adjacências e listas de adjacência; e busca em profundidade e largura; será demonstrado também o problema caminho mínimo e algoritmo de Dijkstra; e o problema da árvore de cobertura mínima e algoritmos de Kruskal e Prim.

1 Introdução

O seguinte relatório tende a trazer um resumo sobre o que são e como os grafos funcionam, além de também mostrar algumas aplicações deles e algumas de suas implementações.

Este capítulo define um objeto combinatório conhecido como grafo. Grafos são importantes modelos para uma grande variedade de problemas de engenharia, computação, matemática, economia, biologia, etc.



1.1 O que é um Grafo?

Um grafo (= graph) é um par de conjuntos: um conjunto de coisas conhecidas como vértices e um conjunto de coisas conhecidas como arcos. Cada arco é um par ordenado de vértices. O primeiro vértice do par é a ponta inicial do arco e o segundo é a ponta final.

Os arcos dos grafos são dirigidos: cada arco começa na sua ponta inicial e termina na sua ponta final. Para enfatizar esse fato, usamos ocasionalmente a expressão grafo dirigido (= directed graph) no lugar de grafo, embora o adjetivo dirigido seja redundante[1].

A ponta final de todo arco é diferente da ponta inicial. Um arco com ponta inicial **v** e ponta final **w** será denotado por

v-w

Dizemos que o arco **v-w** sai de **v** e entra em **w**. A presença de um arco **v-w** é independente da presença do arco **w-v**: o grafo pode ter os arcos **v-w** e **w-v**, pode ter apenas um deles, ou pode não ter nenhum deles.

Dizemos que um vértice **w** é vizinho de um vértice v se **v-w** é um arco do grafo. Dizemos também, nessa circunstância, que **w** é adjacente a **v**. A relação de vizinhança não é simétrica: **w** pode ser vizinho de **v** sem que **v** seja vizinho de **w**.

O tamanho de um grafo com **V** vértices e **A** arcos é a soma **V + A**.

Exemplo A. Uma boa maneira de especificar um grafo é exibir o seu conjunto de arcos. Por exemplo, o conjunto de arcos

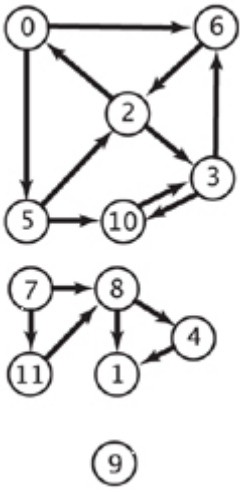
0-5 0-6 2-0 2-3 3-6 3-10 4-1 5-2 5-10
6-2 7-8 7-11 8-1 8-4 10-3 11-8

define um grafo sobre o conjunto de vértices 0..11. [2]

---

[1] (Alguns livros usam o termo digrafo. Esse termo é uma tradução de digraph, que resultou da contração de directed e graph.)

[2] Este exemplo foi copiado do livro Algorithms, de Sedgewick e Wayne.



2 Lista Adjacência

O vetor de listas de adjacência de um grafo tem uma lista encadeada (= linked list) associada com cada vértice do grafo. A lista associada com um vértice v contém todos os vizinhos de v. Portanto, a lista do vértice v representa o leque de saída de v. Por exemplo, eis o vetor de listas de adjacência do grafo cujos arcos são 0-1 0-5 1-0 1-5 2-4 3-1 5-3 :

0: 5 1
1: 5 0
2: 4
3: 1
4:
5: 3

```
int main (void){
	Table table;
	int arestas = quantidade_arestas();
	table.b = malloc(sizeof(Bloco*)*(arestas));
	int vetor_posicoes[arestas];
	for(int i = 0; i < arestas; i++){
		table.b[i] = NULL;
	}
	preencher_vetor(vetor_posicoes, arestas);
	adicionar(&table, arestas, vetor_posicoes);
	exibiTabela(&table, arestas, vetor_posicoes);
	return 0;
}
```

A função demonstrada acima é a principal do programa que será abordado. Ela inicia-se verificando a quantidade de arestas (elementos) no grafo para alocar memória/criar vetor auxiliar, logo em seguida a tabela e o vetor auxiliar de posições é iniciado, preenchendo assim a tabela com com valores null e também adicionando as ligações (vértices) na tabela, por fim, será exibida a tabela criada.

A função QUANTIDADE_ARESTAS() é uma das implementações do algoritmo. Ela é a responsável por abrir o arquivo escolhido para realizar a contagem de linhas (arestas) existentes.

```
int quantidade_arestas(){
            FILE *arq;
            arq = fopen("grafo.txt", "r");
            char linha[100], *resultado;
            int contador = 0;

            while(!feof(arq)){
               resultado = fgets(linha, 100, arq);
               if(strcmp (linha, "#\n") == 0) break;
               else contador++;
            }

            fclose(arq);
            return contador;
         }
```

```
void preencher_vetor(int *vetor, int arestas){

            FILE *arq;
            arq = fopen("grafo.txt", "r");
            char linha[100], *resultado;
            int contador = 0;

            while(contador < arestas){
               resultado = fgets(linha, 100, arq);
               vetor[contador] = atoi(linha);
               contador++;
            }
            fclose(arq);
         }
```

A função ADICIONAR(), realiza a abertura do arquivo, assim que aberto ele vai realizar a comparação se o contador for maior que o número de arestas ele irá quebrar os dois elementos

da linha. Após isso, ele entrará no laço que é responsável por guardar os dois elementos da linha. Quando o while for executado pela primeira vez ele irá pegar o primeiro elemento da linha, na segunda vez que o while rodar ele vai sair pegando o segundo elemento da linha.

Logo em seguida ele irá verificar a posição correspondente da chave no vetor auxiliar, e realiza a verificação de que se a ligação existe ou não. Após isso será criado um novo bloco a qual será adicionado na chave correspondente.

```
void adicionar (Table *table, int arestas , int *vetor){

   FILE *arq;
   arq = fopen("grafo.txt", "r");
   char linha[100], *resultado, *split;
   int  contador = 0, contador_linha = 0, chave, valor, posicao_chave;

   while(!feof(arq)){
      resultado = fgets(linha, 100, arq);
      if(contador > arestas){
         contador_linha = 0;
         split = strtok(linha, " ");

         while (split != NULL){
            if(contador_linha == 0){
               chave = atoi(split);
            }else{
               valor = atoi(split);
            }
            contador_linha++;
            split = strtok (NULL, " ");
         }

         posicao_chave = verifica_posicao_chave(vetor, chave , arestas);
         if(busca_ligacao(table, valor, posicao_chave ) == 0){

         Bloco *bloco = novo(valor);

           if(table->b[posicao_chave] == NULL){
               table->b[posicao_chave] = bloco;
            }else{
               bloco->proximo = table->b[posicao_chave];
               table->b[posicao_chave] = bloco;
            }
         }
      }
      contador++;
   }
   fclose(arq);
}
```

A função abaixo é responsável por criar um novo bloco é a seguinte:

```
Bloco* novo(int c){

   Bloco* bloco;
   bloco = (Bloco*)malloc(sizeof(Bloco));
   bloco->valor = c;
   bloco->proximo = NULL;
   return bloco;
}
```

As funções exibiTabela e exibiLista são responsáveis por “printar” os valores.

```
void exibiTabela(Table *t, int tam, int *vetor){
   int i;
   for (i = 0; i < tam; i++){
      printf("[%d]", vetor[i]);
      printf(" --> ");
      exibiLista(t->b[i]);
      printf("\n");
   }
}

void exibiLista(Bloco *x){
   while (x != NULL){
      printf("%d", x->valor);
      printf(" ");
      x = x->proximo;
   }
}
```

A função VERIFICA_POSICAO_CHAVE() é a responsável por realizar a validação e a verificação da posição da chave.

```
int verifica_posicao_chave(int *vetor, int valor, int arestas) {
   int i;
   for(i = 0; i < arestas; i++){
      if(vetor[i] == valor){
         return i;
      }
   }
}
```

Por fim, mas não menos importante a função BUSCA_LIGACAO() é a grande responsável por realizar todo o processo de busca que ocorre no algoritmo.

```
int busca_ligacao(Table *table, int valor, int posicao_chave){

   Bloco *bloco = table->b[posicao_chave];
   while(bloco != NULL){
      if(bloco->valor == valor) return 1;
      bloco = bloco->proximo;
   }
   return 0;
}
```

3 Matriz Adjacência

A matriz de adjacências de um grafo é uma matriz de 0's e 1's com colunas e linhas indexadas pelos vértices. Se adj[][] é uma tal matriz então, para cada vértice v e cada vértice w,

adj[v][w] = 1 se v-w é um arco e
adj[v][w] = 0 em caso contrário.

Assim, a linha v da matriz adj[][] representa o leque de saída do vértice v e a coluna w da matriz representa o leque de entrada do vértice w. Por exemplo, veja a matriz de adjacências do grafo cujos arcos são 0-1 0-5 1-0 1-5 2-4 3-1 5-3 :

	0	1	2	3	4	5
0	0	1	0	0	0	1
1	1	0	0	0	0	1
2	0	0	0	0	1	0
3	0	1	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	1	0	0

Como nossos grafos não têm laços, os elementos da diagonal da matriz de adjacências são iguais a 0. Se o grafo for não-dirigido, a matriz é simétrica: adj[v][w] ≡ adj[w][v].

A implementação do código inicia-se realizando a verificação da quantidade de arestas (elementos) e logo em seguida realiza a iniciação da matriz e do vetor auxiliar de posições.

Logo em seguida a matriz é preenchida com os 0 e o vetor com os elementos, após ocorrer isso é adicionado as ligações e por fim é exibido a matriz concluída.

```
int main (void){
   int i, j, arestas;
   arestas = quantidade_arestas();
   int matriz[arestas][arestas];
   int vetor_posicoes[arestas];
         for(i = 0; i < arestas; i++)
```

```
                    for(j = 0; j < arestas; j++)
            matriz[i][j] = 0;
    preencher_vetor(vetor_posicoes, arestas);
    adicionar(arestas, vetor_posicoes, matriz);
    exibir(arestas, vetor_posicoes, matriz);
    return 0;
}
```

A função QUANTIDADE_ARESTAS() é a responsável por realizar a abertura do arquivo escolhido para realizar a contagem de linhas (arestas) existentes. Se ela encontrar o # ela dá um break e encerra o while.

```
int quantidade_arestas(){
    FILE *arq;
    arq = fopen("grafo.txt", "r");
    char linha[100], *resultado;
    int contador = 0;
    while(!feof(arq)){
        resultado = fgets(linha, 100, arq);
        if(strcmp (linha, "#\n") == 0) break;
        else contador++;
    }
    fclose(arq);
    return contador;
}
```

A função PREENCHER_VETOR() é a responsável realizar o preenchimento de todo o vetor, para isso ele realiza a abertura do arquivo e realiza a comparação se o valor que está na variável contador é menor que o número de arestas.

```
void preencher_vetor(int *vetor, int arestas){
    FILE *arq;
    arq = fopen("grafo.txt", "r");
    char linha[100], *resultado;
    int contador = 0;
    while(contador < arestas){
        resultado = fgets(linha, 100, arq);
        vetor[contador] = atoi(linha);
        contador++;
    }
    fclose(arq);
}
```

A função ADICIONAR() é a responsável por realizar a adição dos valores no vetor, ele começa a realizar a leitura após achar o # que é o separador, se o valor que está na variável contador for maior que o valor de arestas ele vai quebrar os dois elementos da linha.

Logo em seguida ele irá verificar a posição correspondente da chave no vetor auxiliar, e realiza a verificação de que se a ligação existe ou não. Após isso será criado um novo bloco a qual será adicionado na chave correspondente.

```
void adicionar (int arestas , int *vetor, int matriz[arestas][arestas]){
   FILE *arq;
   arq = fopen("grafo.txt", "r");
   char linha[100], *resultado, *split;
   int  contador = 0, contador_linha = 0, chave, valor, posicao_chave, posicao_valor;

   while(!feof(arq)){
      resultado = fgets(linha, 100, arq);
      if(contador > arestas){
         contador_linha = 0;
         split = strtok(linha, " ");
         while (split != NULL){
            if(contador_linha == 0){
               chave = atoi(split);
            }else{
               valor = atoi(split);
            }
            contador_linha++;
            split = strtok (NULL, " ");
         }
         posicao_chave = verifica_posicao_chave(vetor, chave , arestas);
         posicao_valor = verifica_posicao_chave(vetor, valor , arestas);
         matriz[posicao_chave][posicao_valor] = 1;
      }
      contador++;
   }
   fclose(arq);
}
```

A função VERIFICA_POSICAO_CHAVE() realiza a verificação da posição de cada elemento no vetor.

```
int verifica_posicao_chave(int *vetor, int valor, int arestas) {
   int i;
   for(i = 0; i < arestas; i++){
      if(vetor[i] == valor){
         return i;
      }
   }
}
```

Por fim, a matriz é printada na tela através da função EXIBIR().

```
void exibir(int arestas, int *vetor, int matriz[arestas][arestas]){
   int i, j, k;
   printf("     ");
   for(k=0; k < arestas; k++) printf("[%d] ", vetor[k]);
   printf("\n");
   for(i = 0; i < arestas; i++){
      printf("[%d]", vetor[i]);
      for(j = 0; j < arestas; j++){
         printf("   %d ", matriz[i][j]);
      }
      printf("\n");
   }
}
```

4 Busca em Profundidade

Um algoritmo de busca (ou de varredura) é um algoritmo que visita todos os vértices de um grafo andando pelos arcos de um vértice a outro. Há muitas maneiras de fazer uma tal busca. Cada algoritmo de busca é caracterizado pela ordem em que os vértices são visitados.

O algoritmo de busca em profundidade (= depth-first search), ou busca DFS, tem por objetivo visitar todos os vértices e numerá-los na ordem em que são descobertos.

A busca em profundidade não resolve um problema específico. Ela é apenas um arcabouço, ou pré-processamento, para a resolução eficiente de vários problemas concretos. A busca DFS nos ajuda a compreender o grafo com que estamos lidando, revelando sua forma e reunindo informações (representadas pela numeração dos vértices) que ajudam a responder perguntas sobre o grafo.

A busca em profundidade está relacionada com expressões como backtracking, pré-ordem, exploração de labirintos, exploração Trémaux, fio de Ariadne (no mito de Teseu e o Minotauro), etc.

O algoritmo de busca DFS visita todos os vértices e todos os arcos do grafo numa determinada ordem e atribui um número a cada vértice: o k-ésimo vértice descoberto recebe o número k .

```
#include<stdio.h>
#define max 50
#define nulo 0
#define infinito max-1

enum booleano {false,true};

int grafo[max][max];
int distancia[max];
int vertices, arestas;
```

```
int custo=0;

int main(){
  int i;
  limpa_grafo();
  ler_grafo();
  mostra_grafo();
  DFS(0);
  for (i=0;i<vertices;i++){printf("[%d]",distancia[i]);}
  printf("\n");
}
```

No código acima é possível ver o código da classe principal do problema de busca em profundidade. No código abaixo é possível ver a função que é utilizada para limpar o grafo.

```
void limpa_grafo(){
          int i,j;
          for (i=0;i<vertices;i++)
            for (j=0;j<vertices;j++) grafo[i][j]=nulo;
}
```

```
void mostra_grafo(){
          int i,j;
          for (i=0;i<vertices;i++){
            for (j=0;j<vertices;j++){
               if (grafo[i][j]==nulo)printf(". ");
          else printf("* ");
            }
            printf("\n");
          }
}
```

Na implementação acima é mostrada a função MOSTRAR_GRAFO() que fica responsável por fazer a impressão, e no código abaixo tem a função que realizará a leitura do grafo.

```
void ler_grafo(){
          int i,j,a,b;
          scanf("%d %d",&vertices, &arestas);
       for (j=0;j<arestas;j++){
         scanf("%d %d",&a,&b);
         grafo[a-1][b-1]=1;
         grafo[b-1][a-1]=1;
       }
}
```

```
void visitaDFS(int n){
        int i;
        distancia[n]=custo;
        for (i=0;i<vertices;i++)
        if ( (grafo[n][i]) && (custo<distancia[i]) ){
                custo++;
                printf("visitando %d a partir de %d. custo= %d \n",i,n,custo);
                visitaDFS(i);
                custo--;
        }
}
```

A implementação exemplificada logo acima, mostra o código de como funciona a visita aos valores no DFS, o código abaixo é a implementação dos graus.

```
int grau(int n){
        int i,ocorrencias;
        for (i=0;i<vertices;i++)
          if (grafo[n][i]) ocorrencias++;
        return(ocorrencias);
}
```

```
void DFS(int source){
  int i;
  printf("Inicializando uma DFS em %d\n",source);
  for(i=0;i<vertices;i++){distancia[i]=infinito;}
  distancia[source]=0;
  custo=0;
  for(i=0;i<vertices;i++)
  if  ( (grafo[source][i]) && (custo<distancia[i])){
    printf("visitando %d a partir de %d . custo= %d \n",i,source,custo);
    custo++;
    visitaDFS(i);
    custo--;
  }
}
```

A implementação acima é o código responsável por inicializar todo o processo da DFS.

5 Busca em Largura

Um algoritmo de busca é um algoritmo que esquadrinha um grafo andando pelos arcos de um vértice a outro. Depois de visitar a ponta inicial de um arco, o algoritmo percorre o arco e visita sua ponta final. Cada arco é percorrido no máximo uma vez.

Há muitas maneiras de organizar uma busca. Cada estratégia de busca é caracterizada pela ordem em que os vértices são visitados. Este capítulo introduz a busca em largura (= breadth-first search = BFS), ou busca BFS. Essa estratégia está intimamente relacionada com os conceitos de distância e caminho mínimo.

A busca em largura começa por um vértice, digamos s, especificado pelo usuário. O algoritmo visita s, depois visita todos os vizinhos de s, depois todos os vizinhos dos vizinhos, e assim por diante.

O algoritmo enumera os vértices, em sequência, na ordem em que eles são descobertos (ou seja, visitados pela primeira vez). Para fazer isso, o algoritmo usa uma fila de vértices. No começo de cada iteração, a fila contém vértices que já foram numerados mas têm vizinhos ainda não numerados. O processo iterativo consiste no seguinte:

```
enquanto a fila não estiver vazia
        retire um vértice v da fila
        para cada vizinho w de v
                se w não está numerado
                        então numere w
                         ponha w na fila
```

No começo da primeira iteração, a fila contém o vértice s, com número 0, e nada mais.

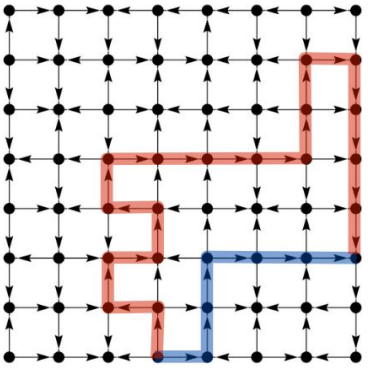
Na implementação do algoritmo é possível ver no início de cada iteração são ressaltadas as seguinte propriedades:

1. todo vértice que está na fila já foi numerado;
2. se um vértice v já foi numerado mas algum de seus vizinhos ainda não foi numerado, então v está na fila.

4 Caminho Mínimo

Um caminho *C* num grafo é mínimo se não existe outro caminho que tenha a mesma origem e o mesmo término que C, mas comprimento menor que o de C.

Na figura ao lado, o caminho vermelho não é mínimo porque o caminho azul é mais curto.



Problema do caminho mínimo: Dados vértices s e t de um grafo G, encontrar um caminho mínimo em G que tenha origem s e termino t.

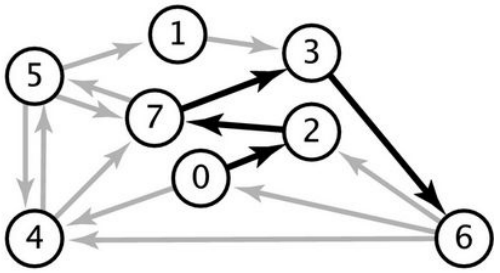
É claro que todo caminho mínimo é simples. É claro também que nem toda instância do problema tem solução: se t não está ao alcance de s então não existe caminho algum de s a t.

A distância de s a t num grafo é o comprimento de um caminho mínimo com origem s e término t. É claro que a distância de s a t é 0 se e somente se s ≡ t. É claro também que a distância de s a

t é d se e somente se (1) existe um caminho de comprimento d de s a t e (2) todo caminho de s a t tem comprimento pelo menos d.

Se não existe caminho algum de s a t, podemos dizer que a distância de s a t é ∞ (infinita).

O conceito de distância é dirigido: a distância de s a t é, em geral, diferente da distância de t a s. Em grafos não-dirigidos, entretanto, as duas distâncias são iguais.



A figura mostra um caminho mínimo em um grafo. O caminho tem comprimento 4 e vai do vértice 0 ao vértice 6. A tabela abaixo mostra as distâncias do vértice 0 a cada um dos demais vértices:

v	0	1	2	3	4	5	6	7
dist[v]	0	3	1	3	1	2	4	2

5 Minimum Spanning Tree

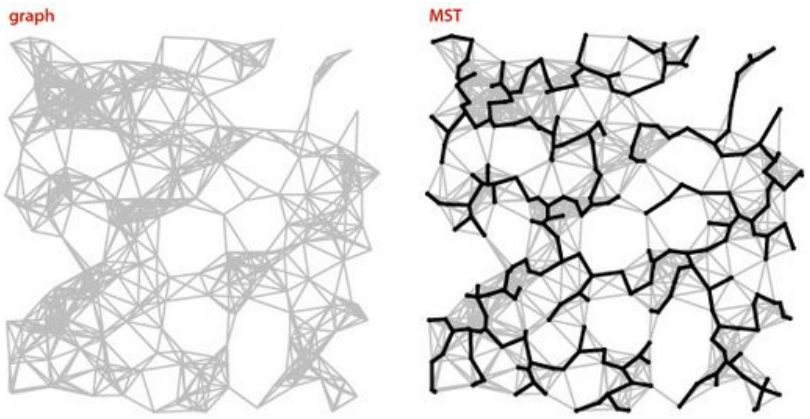
Seja G um grafo não-dirigido com custos nas arestas. Os custos podem ser positivos ou negativos. O custo de um subgrafo não-dirigido T de G é a soma dos custos das arestas de T.

Uma árvore geradora mínima de G é qualquer árvore geradora de G que tenha custo mínimo. Em outras palavras, uma árvore geradora T de G é mínima se nenhuma outra árvore geradora tem custo estritamente menor que o de T. Árvores geradoras mínimas também são conhecidas pela abreviatura MST de minimum spanning tree.

Problema da MST: Encontrar uma árvore geradora mínima de um grafo não-dirigido com custos nas arestas.

É claro que o problema tem solução se e somente se o grafo é conexo. Outra observação óbvia: se todos as arestas tiverem o mesmo custo então toda árvore geradora é uma MST.
Este capítulo faz uma introdução geral ao problema da MST. Algoritmos serão examinados em capítulos subsequentes.

A figura abaixo mostra um grafo não-dirigido com custos nas arestas e sua MST. Os 250 vértices são pontos no plano e o custo de cada aresta v-w (há 1273 delas) é igual à distância geométrica entre os pontos v e w.



6 Algoritmo de Dijkstra

O algoritmo de Dijkstra, concebido pelo cientista da computação holandês Edsger Dijkstra em 1956 e publicado em 1959, soluciona o problema do caminho mais curto num grafo dirigido ou não dirigido com arestas de peso não negativo, em tempo computacional O(m + n log n) onde m é o número de arestas e n é o número de vértices. O algoritmo que serve para resolver o mesmo problema em um grafo com pesos negativos é o algoritmo de Bellman-Ford, que possui maior tempo de execução que o Dijkstra.

O algoritmo considera um conjunto S de menores caminhos, iniciado com um vértice inicial I. A cada passo do algoritmo busca-se nas adjacências dos vértices pertencentes a S aquele vértice com menor distância relativa a I e adiciona-o a S e, então, repetindo os passos até que todos os vértices alcançáveis por I estejam em S. Arestas que ligam vértices já pertencentes a S são desconsideradas.

Um exemplo prático do problema que pode ser resolvido pelo algoritmo de Dijkstra é: alguém precisa se deslocar de uma cidade para outra. Para isso, ela dispõe de várias estradas, que passam por diversas cidades. Qual delas oferece uma trajetória de menor caminho?

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define FLSH gets(l)

int destino, origem, vertices = 0;
int custo, *custos = NULL;

void dijkstra(int vertices,int origem,int destino,int *custos){
```

```
 int i,v, cont = 0;
 int *ant, *tmp;
 int *z;
 double min;
 double dist[vertices];
 ant = calloc (vertices, sizeof(int *));
 tmp = calloc (vertices, sizeof(int *));
 if (ant == NULL) {
  printf ("** Erro: Memoria Insuficiente **");
  exit(-1);
 }

 z = calloc (vertices, sizeof(int *));
 if (z == NULL) {
  printf ("** Erro: Memoria Insuficiente **");
  exit(-1);
 }

 for (i = 0; i < vertices; i++) {
  if (custos[(origem - 1) * vertices + i] !=- 1) {
    ant[i] = origem - 1;
    dist[i] = custos[(origem-1)*vertices+i];
  }
  else {
    ant[i]= -1;
    dist[i] = HUGE_VAL;
  }
  [i]=0;
 }
```

```
void add(void){
 int i, j;

 do {
  printf("\nQual o numero de vertices (numero minimo = 2 ): ");
  scanf("%d",&vertices);
 } while (vertices < 2 );

 if (!custos)
  free(custos);
 custos = (int *) malloc(sizeof(int)*vertices*vertices);
 for (i = 0; i <= vertices * vertices; i++)
  custos[i] = -1;

 printf("Insira as arestas:\n");
 do {
  do {
```

```
    printf("Origem da aresta (entre 1 e %d ou '0' para sair): ", vertices);
    scanf("%d",&origem);
   } while (origem < 0 || origem > vertices);

   if (origem) {
    do {
     printf("Destino da aresta (entre 1 e %d, menos %d): ", vertices, origem);
     scanf("%d", &destino);
    } while (destino < 1 || destino > vertices || destino == origem);

    do {
     printf("Custo (positivo) da aresta do vertice %d para o vertice %d: ",
     origem, destino);
     scanf("%d",&custo);
    } while (custo < 0);

    custos[(origem-1) * vertices + destino - 1] = custo;
   }
  } while (origem);
}
```

```
void procurar(void){
 int i, j;
 printf("Lista dos Menores Caminhos no Grafo Dado: \n");

 for (i = 1; i <= vertices; i++) {
  for (j = 1; j <= vertices; j++)
   dijkstra(vertices, i,j, custos);
  printf("\n");
 }
 printf("ENTER para voltar ao menu inicial>\n");
 printf("\033[m");
}
```

```
int main(int argc, char **argv) {
 int i, j;
 char opcao[3], l[50];

 do {
  cabecalho();
  scanf("%s", &opcao);

  if ((strcmp(opcao, "d")) == 0) {
   add();
  }
  FLSH;
```

```
    if ((strcmp(opcao, "r") == 0) && (vertices > 0) ) {
      procurar();
      FLSH;
    }
  } while (opcao != "x");
  printf("\nOver & Out\n\n");
  return 0;
}
```

7 problema da árvore de cobertura mínima

8 Algoritmos de Kruskal e Prim

6 Referências

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/graphs.html
https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/dfs.html
https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/graphdatastructs.html#adjmatrix
https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/shortestpaths-intro.html
https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/mst.html