PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Ciências Exatas e de Informática

As diferentes abordagens para o problema do caminho mais curto*

Projeto e Análise de Algoritmos

Guilherme Reis Barbosa de Oliveira
Ian Rodrigues dos Reis Paixão
Jorge Allan de Castro Oliveira

Resumo

O documento tem o intuito de apresentar ao leitor três diferentes abordagens utilizadas pelos autores para solucionar o problema do caminho mais curto em um grafo com peso não negativo, para determinar se existe um ciclo negativo em um grafo e, por fim, para resolver o problema do caminho mais curto em um grafo com aresta negativa, mas sem ciclo negativo. Todas as implementações foram realizadas por meio da linguagem C++ e seus recursos, bem como as técnicas de algoritmo guloso e programação dinâmica. Ao final é disponibilizado as análises de complexidade e os respectivos testes.

Palavras-chave: Algoritmo de Dijkstra. Algoritmo em Grafos. C++. Programação Dinâmica. Algoritmos Gulosos. Análise de Complexidade.

Abstract

This document aims to introduce three different approaches used by the authors to solve the problem of the shortest path in a non-negative weight graph, an algorithm to determine if there is a negative cycle in a graph and an algorithm to solve the problem of the shortest path in a negative weight graph, but without negative cycles. All the implementations was realized in C++ language and their resources, as well the greedy algorithms and dynamic programming techniques. The complexity analysis and their respective tests is available at the end of this article.

Keywords: Dijkstra Algorithm. Graph Algorithm. C++. Dynamic Programming. Greedy Algorithm. Complexity Analysis.

*Artigo apresentado à professora Raquel Aparecida de Freitas Mini do Instituto de Ciências Exatas e Informática da Pontifícia Universidade Católica de Minas Gerais como um trabalho da disciplina Projeto e Análise de Algoritmos do curso Ciência da Computação no primeiro semestre do ano de 2019.

Sumário

1 Introdução **3**
1.1 O problema do caminho mais curto 3
1.2 Objetivos do trabalho prático 4

2 As técnicas de programação **4**
2.1 Algoritmos gulosos 4
2.2 Programação dinâmica 4

3 Implementação **5**
3.1 Heurística gulosa para resolver o problema do caminho mais curto em um grafo com peso não negativo 5
3.2 Algoritmo para determinar se existe um ciclo negativo em um grafo 8
3.3 Algoritmo de programação dinâmica para resolver o problema do caminho mais curto em um grafo com aresta negativa, mas sem ciclo negativo 11

4 Análise de complexidade **15**
4.1 Complexidade da heurística gulosa para resolver o problema do caminho mais curto em um grafo com peso não negativo 15
4.2 Complexidade do algoritmo para determinar se existe um ciclo negativo em um grafo 15
4.3 Complexidade do algoritmo de programação dinâmica para resolver o problema do caminho mais curto em um grafo com aresta negativa, mas sem ciclo negativo 16

5 Testes **17**
5.1 Testes com a heurística gulosa para resolver o problema do caminho mais curto em um grafo com peso não negativo 17
5.2 Testes com o algoritmo para determinar se existe um ciclo negativo em um grafo 18
5.3 Testes com o algoritmo de programação dinâmica para resolver o problema do caminho mais curto em um grafo com aresta negativa, mas sem ciclo negativo . 18
5.4 Gráficos referentes aos tempos de cada algoritmo abordado em relação ao tamanho de suas entradas 18

6 Conclusão **20**

7 Referências **21**

8 Anexos **22**

1 INTRODUÇÃO

1.1 O problema do caminho mais curto

Este problema, como o próprio nome indica, procura encontrar a menor rota entre dois pontos distintos, também conhecidos como nós. Dessa forma, resolver o problema significa determinar o caminho entre dois nós com um custo mínimo, com um menor tempo de viagem ou com a máxima capacidade, (Ahuja et al., 1993). É muito utilizado em problemas de redes de comunicações.

Muitos autores consideram o problema do caminho mais curto como uma das mais importantes áreas de pesquisa devido à grande quantidade de aplicações práticas. As aplicações estão relacionadas com a otimização de atividades como, por exemplo, o tráfego de estradas, linhas de transmissão elétrica, conexão de redes, problemas de programação planejamento de movimentos de um robô e outros.

Esses problemas podem ser resolvidos por meio de algoritmos muito eficientes como, por exemplo, o algoritmo de Dijkstra ou o de Bellman-Ford. É possível entender como funciona o algoritmo por meio de um simples caso do cotidiano, como o deslocamento entre diferentes cidades.

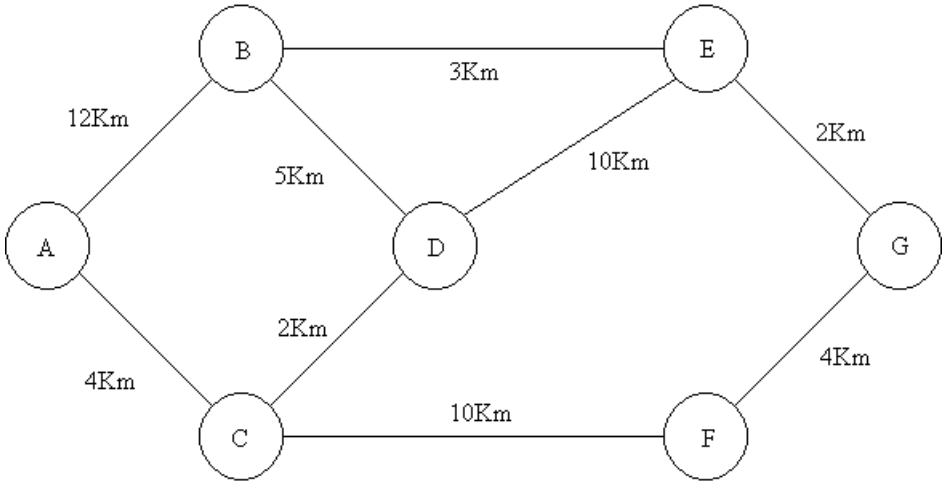

Figura 1 - Mapa de estrada

Utilizando o algoritmo de Dijkstra, ao escolher um vértice como a raiz da busca, este calcula o custo mínimo do vértice escolhido para todos os outros vértices do grafo por meio dos pesos das arestas que compõem o caminho. É uma lógica bastante simples e com um bom nível de performance. O Dijkstra não garante, contudo, a exatidão da solução caso haja a presença de arcos com valores negativos. Este algoritmo parte de uma estimativa inicial para o custo mínimo e vai sucessivamente ajustando esta estimativa.

1.2 Objetivos do trabalho prático

O artigo tem em vista expandir, o máximo possível, os conceitos sobre o problema do menor caminho, apresentando soluções mais complexas que os algoritmos tradicionais já conhecidos e que não são capazes de solucionar, por exemplo, um sistema financeiro de transações, onde existem arestas de peso negativo.

2 AS TÉCNICAS DE PROGRAMAÇÃO

O primeiro programa foi desenvolvido sob a estratégia gulosa de solução de problemas, enquanto no último é utilizado o método da programação dinâmica. A seguir é possível ver mais detalhes sobre como cada técnica funciona.

2.1 Algoritmos gulosos

Para resolver um problema, o algoritmo guloso sempre realiza a escolha que parece ser a melhor no momento, ou seja, uma escolha ótima local na esperança de que esta escolha leve até a solução ótima global.

Ele toma decisões com base nas informações disponíveis na iteração corrente, sem olhar as consequências que essas decisões terão no futuro. Um algoritmo guloso jamais se arrepende ou volta atrás, já que as escolhas que faz em cada iteração são definitivas.

Vantagens: São simples e fáceis de implementar.

Desvantagens: Nem sempre leva à soluções ótimas globais.

2.2 Programação dinâmica

A Programação Dinâmica procura resolver o problema de otimização através da análise de uma seqüência de problemas mais simples do que o problema original, afim de prevenir queda de performance e recálculos desnecessários. Podemos associar a ela recursos como recursividade de um código.

Um exemplo de programação dinâmica é a solução para o problema do caixeiro viajante que utiliza a ideia de subconjunto de cidades para a solução de subproblemas. Assim como o algoritmo guloso, existem pontos positivos e negativos em relação ao uso desse método de construção.

Vantagens: Útil para aplicar em problemas que exigem teste de todas as possibilidades.

Desvantagens: A complexidade espacial pode vir a ser exponencial.

3 IMPLEMENTAÇÃO

Nessa seção será especificada a implementação das três soluções propostas para o problema, bem como a forma como é feita a execução dos algoritmos. Uma breve descrição sobre a finalidade de cada função está em forma de comentário no próprio código apresentado.

Todos os algoritmos foram implementados na linguagem *C++* e compilados com o *G++* na versão *8.3.0*. As entradas do programa são grafos e as saídas encontram-se em formato texto no terminal que foi executado. Para compilar e executar os programas deve-se usar as diretivas abaixo:

g++ nomePrograma.cpp -o programa

./programa

O Visual Studio Code foi utilizado como ambiente de desenvolvimento para implementação dos códigos e todos eles foram testados na mesma máquina, sob o sistema operacional *Ubuntu 16.04 LTS* que possui as seguintes configurações:

- Intel(R) Core(TM) i5-7600K CPU @ 3.80GHz
- 8GB RAM

3.1 Heurística gulosa para resolver o problema do caminho mais curto em um grafo com peso não negativo

O algoritmo de Dijkstra é, em muitos aspectos, semelhante ao algoritmo de Prim para a árvore geradora mínima. Assim como a árvore geradora mínima de Prim, foi decidido criar uma árvore de caminho mais curto em que existe uma determinada fonte como origem.

Como se vê, o algoritmo de árvore geradora mínima tem caráter guloso, pois a cada iteração, esse algoritmo escolhe a aresta com o menor custo sem se preocupar com o efeito global dessa escolha. Abaixo estão os detalhes de funcionamento do algoritmo para encontrar o caminho mais curto de um único vértice de origem para todos os outros vértices existentes no grafo não-dirigido:

1. Existe um array booleano **arvCaminho** que monitora os vértices incluídos na árvore de caminho mais curto, ou seja, cuja distância mínima da fonte é calculada e finalizada. Inicialmente este conjunto está vazio.

2. Se um valor **arvCaminho[v]** for verdadeiro, então o vértice v será incluído na árvore, caso contrário, não será. Array **distancia[]** é usado para armazenar os menores valores de distância de todos os vértices.

3. É atribuído um valor de distância a todos os vértices no grafo de entrada. Inicializa-se todos os valores de distância como **INFINITO** e também atribui-se valor de distância como zero para o vértice de origem que foi escolhido.

4. Enquanto a **arvCaminho** não inclui todos os vértices:

   (a) Escolhe um vértice u que não esteja em **arvCaminho** e tem valor mínimo de distância.

   (b) Inclui o vértice u em **arvCaminho**.

   (c) Atualiza o valor da distância de todos os vértices adjacentes de u. Para atualizar os valores de distância, percorra todos os vértices adjacentes. Para cada vértice adjacente v, se a soma do valor da distância de u (da origem) e o peso da aresta uv for menor que o valor da distância de v, então atualiza o valor da distância de v.

- Abaixo é possível verificar o código na íntegra:

```
/**
 * Algoritmo Guloso para o problema do menor caminho
 * @author Guilherme Reis Barbosa de Oliveira
 * @author Ian Rodrigues dos Reis Paixao
 * @author Jorge Allan de Castro Oliveira
 * @version 2 05/2019
 */

#include <limits.h>
#include <stdio.h>

// Numero de vertices no grafo
#define V 4

// Funcao para encontrar o vertice com valor minimo de distancia a
   partir do conjunto de vertices ainda nao incluidos na arvore de
   caminho mais curto
int distanciaMinima(int distancia[], bool arvCaminho[]) {
    // Inicializa o valor da variavel min
    int min = INT_MAX, min_ind;

    for (int v = 0; v < V; v++) {
        if (arvCaminho[v] == false && distancia[v] <= min) {
            min = distancia[v], min_ind = v;
        }
    }
    return min_ind;
```

```
26 }
27
28 // Funcao para imprimir a matriz de distancia que foi construida
29 int imprimirMatriz(int distancia[], int n) {
30     printf("Vertice  || Dist. origem\n");
31     for (int i = 0; i < V; i++) {
32         printf("%d || %d\n", i, distancia[i]);
33     }
34 }
35
36 // Funcao que implementa o Dijkstra para solucao do menor caminho
      para um grafo que utiliza matriz de adjacencia
37 void dijkstra(int grafo[V][V], int orig) {
38     int distancia[V]; // O array de saida. O distancia[i] vai manter
          a menor distancia de orig ate i
39
40     bool arvCaminho[V]; // arvCaminho[i] sera verdadeiro se o vertice
          i for incluido na arvore de caminho mais curto ou a distancia
          mais curta de orig ate i for finalizada
41
42     // Inicializa todas as distancias como INFINITO e arvCaminho[]
          como falso
43     for (int i = 0; i < V; i++) {
44         distancia[i] = INT_MAX, arvCaminho[i] = false;
45     }
46
47     // Distancia do vertice de origem de si mesmo eh sempre igual a
          zero
48     distancia[orig] = 0;
49
50     // Encontra o caminho mais curto para todos os vertices
51     for (int i = 0; i < V - 1; i++) {
52         // Escolhe o vertice de distancia minima do conjunto de
              vertices ainda nao processados. u eh sempre igual a orig
              na primeira iteracao
53         int u = distanciaMinima(distancia, arvCaminho);
54
55         // Marca o vertice escolhido como processado
56         arvCaminho[u] = true;
57
58         // Atualiza o valor de distancia dos vertices adjacentes do
              vertice escolhido.
59         for (int v = 0; v < V; v++) {
```

```
60              // Atualiza distancia[v] somente se nao estiver em
                   arvCaminho, existe uma aresta de u para v, e o peso
                   total do caminho de orig para v ate u eh menor que o
                   valor atual de distancia[v]
61              if (!arvCaminho[v] && grafo[u][v] && distancia[u] !=
                   INT_MAX
62                  && distancia[u] + grafo[u][v] < distancia[v])
63                  distancia[v] = distancia[u] + grafo[u][v];
64          }
65      }
66
67      // Imprime a matriz de distancia
68      imprimirMatriz(distancia, V);
69  }
70
71  int main() {
72      //Valores de entrada do grafo
73      int grafo[V][V] = { { 0, 1, 100, 1000},
74                          { 1, 0, 6, 5},
75                          { 100, 6, 0, 10},
76                          { 1000, 5, 10, 0 } };
77
78      //Chamada da funcao que calcula menor caminho
79      dijkstra(grafo, 0);
80
81      return 0;
82  }
```

- A saída deste programa é:

```
1  Vertice || Dist. origem
2     0    ||    0
3     1    ||    1
4     2    ||    7
5     3    ||    6
```

3.2 Algoritmo para determinar se existe um ciclo negativo em um grafo

Para a resolução do problema em questão, foi utilizado o algoritmo de Floyd-Warshall que resolve o problema de calcular o caminho mais curto entre todos os pares de vértices em um grafo orientado e valorado. O algoritmo de Floyd-Warshall utiliza ideias de programação dinâmica.

A seguir é possível visualizar como funciona este algoritmo e no código, encontra-se comentado de forma mais detalhada como o Floyd-Warshall é utilizado para encontrar e determinar existência de ciclo negativo no grafo.

1. No início **dist[][]** só tem em conta os caminhos diretos *(usando uma aresta do grafo)*.
2. No final da primeira iteração **(com k = 1)**, tem em conta todos os caminhos diretos ou que usam o nó 1 como ponto intermédio.
3. No final de *i* iterações **(com k <= i)**, tem em conta todos os caminhos diretos ou que usam quaisquer nós menores ou iguais a *i*.
4. Quando chegamos ao final, todos os caminhos possíveis são tidos em conta.
5. Se existir um **ciclo negativo**, vamos ter uma entrada **dist[v][v]** com valor negativo durante a execução do algoritmo.

- O código encontra-se abaixo:

```
/**
 * Programa que verifica a existencia de um ciclo negativo
 * utlizando o algoritmo de Floyd Warshall
 * @author Guilherme Reis Barbosa de Oliveira
 * @author Ian Rodrigues dos Reis Paixao
 * @author Jorge Allan de Castro Oliveira
 * @version 2 05/2019
 */

#include <iostream>
using namespace std;

// Numero de vertices do grafo
#define V 4

// Define INF como o maior valor possivel
//Este valor sera usado para vertices nao conectados a nenhum outro
#define INF 99999

// Imprime solucao da matriz
void printSolucao(int dist[][V]);

// Retorna se existe ciclo negativo ou nao
bool cicloNegfloydWarshall(int grafo[][V]) {
    // dist[][] eh o vetor que armazena a menor distancia entre um
       par de vertices
```

```
26      int dist[V][V], i, j, k;
27
28      // Inicializa a matriz dist com os mesmos valores do grafo
29      // Podemos dizer que o menor caminho inicial eh a distancia entre
            os vertices sem nenhum no entre eles.
30      for (i = 0; i < V; i++) {
31          for (j = 0; j < V; j++) {
32              dist[i][j] = grafo[i][j];
33          }
34      }
35
36      /* Adiciona todos os vetices no conjunto de vertices
           intermediarios.
37      -> Antes de inicializar, temos as menores distancias entre todos
           os pares de vertices de tal modo que as menores distancias
           considerem apenas um conjunto de vertices intermediarios {0,
           1, 2, .. k}
38      -> Apos o final da iteracao, vertice de numero k eh adicionado ao
            conjunto de vertices intermediarios {0, 1, 2, .. k} */
39      for (k = 0; k < V; k++) {
40          // Pega todos os vertices pela origem um por um
41          for (i = 0; i < V; i++) {
42              // Pega todos os vertices destinos acima do origem
                   selecionado
43              for (j = 0; j < V; j++) {
44                  // Se o vertice k esta no menor caminho de i para j,
                       entao o valor eh atualizado em dist[]i[j]
45                  if (dist[i][k] + dist[k][j] < dist[i][j]) {
46                      dist[i][j] = dist[i][k] + dist[k][j];
47                  }
48              }
49          }
50      }
51
52      // Se a distancia de qualquer vertice para ele mesmo se torna
           negativa, entao existe um ciclo negativo
53      for (int i = 0; i < V; i++) {
54          if (dist[i][i] < 0) {
55              return true;
56          }
57      }
58      return false;
59  }
```

```
60
61 int main() {
62     //Valores de entrada do grafo
63     int grafo[V][V] = { {0 , -1 , INF , INF},
64                         {INF , 0 , -1 , INF},
65                         {INF , INF , 0 , -1},
66                         {-1 , INF , INF , 0} };
67
68     //Imprime se existe ciclo negativo ou nao
69     if (cicloNegfloydWarshall(grafo)) {
70         cout << "Sim";
71     } else {
72         cout << "Nao";
73     }
74     return 0;
75 }
```

- A saída deste programa é:

```
1 Sim
```

3.3 Algoritmo de programação dinâmica para resolver o problema do caminho mais curto em um grafo com aresta negativa, mas sem ciclo negativo

Para essa solução, foi utilizado uma função baseada em programação dinâmica que calcula o menor caminho ao armazenar em uma tabela o peso do caminho mais curto de um vértice **i** até um vértice **j** com **k** arestas. Trata-se de um código de programação dinâmica graças a característica de transformar um problema em problemas menores, a fim de atingir o objetivo.

A função **cicloNegfloydWarshall()** da abordagem anterior também se faz presente para que possa ser verificado se existe ou não um ciclo negativo no grafo. Antes da execução do **menorCaminho()**, é necessário saber se o grafo de entrada possui ou não um ciclo negativo.

Dessa forma, pegamos o valor de retorno da função **cicloNegfloydWarshall()** e ocorre os seguintes passos descritos abaixo:

1. Verifica se o retorno da função **cicloNegfloydWarshall()** é verdadeiro ou falso.
   (a) Se for **falso**, imprime na tela o valor inteiro que a função **cicloNegfloydWarshall()** retorna.
   (b) Se for **verdadeiro**, imprime na tela uma mensagem dizendo que existe ciclo negativo.

- Segue o código com comentários:

```
/**
 * Algoritmo de programacao dinamica para encontrar o caminho
 * mais curto sem ciclo negativo
 * @author Guilherme Reis Barbosa de Oliveira
 * @author Ian Rodrigues dos Reis Paixao
 * @author Jorge Allan de Castro Oliveira
 * @version 3 05/2019
 */

#include <iostream>
#include <climits>
using namespace std;

// Define o numero de vertices do grafo e valor infinito
#define V 4
#define INF INT_MAX

// Funcao baseada em programacao dinamica para definir o menor
   caminho de u ate v com exatamente k arestas
int menorCaminho(int grafo[][V], int u, int v, int k) {
// Tabela para ser preenchida usando programacao dinamica. A variavel
    sp[i][j][e] vai armazenar o peso do caminho mais curto de i ate j
    com exatamente k arestas
    int sp[V][V][k+1];

    // Loop para o numero de arestas de 0 ate k
    for (int e = 0; e <= k; e++) {
        for (int i = 0; i < V; i++) { // Origem
            for (int j = 0; j < V; j++) { // Destino
                // Inicializa variavel com valor INF
                sp[i][j][e] = INF;

                // Casos base
                if (e == 0 && i == j) {
                    sp[i][j][e] = 0;
                }
                if (e == 1 && grafo[i][j] != INF) {
                    sp[i][j][e] = grafo[i][j];
                }

                // Vai para o adjacente somente quando o numero de
                   arestas eh maior do que 1
                if (e > 1) {
```

```
40                      for (int a = 0; a < V; a++) {
41                          // Deve haver uma aresta de i ate a e a nao
                               deve ser igual i ou j
42                          if (grafo[i][a] != INF && i != a &&  j!= a &&
                                sp[a][j][e-1] != INF) {
43                              sp[i][j][e] = min(sp[i][j][e], grafo[i][a
                                   ] + sp[a][j][e-1]);
44                          }
45                      }
46                  }
47              }
48          }
49      }
50      return sp[u][v][k];
51  }
52
53  // Retorna se existe ciclo negativo ou nao
54  bool cicloNegfloydWarshall(int grafo[][V]) {
55      // dist[][] eh o vetor que armazena a menor distancia entre um
           par de vertices
56      int dist[V][V], i, j, k;
57
58      // Inicializa a matriz dist com os mesmos valores do grafo
59      // Podemos dizer que o menor caminho inicial eh a distancia entre
            os vertices sem nenhum no entre eles
60      for (i = 0; i < V; i++) {
61          for (j = 0; j < V; j++) {
62              dist[i][j] = grafo[i][j];
63          }
64      }
65
66      /* Adiciona todos os vetices no conjunto de vertices
           intermediarios.
67      -> Antes de inicializar, temos as menores distancias entre todos
           os pares de vertices de tal modo que as menores distancias
           considerem apenas um conjunto de vertices intermediarios {0,
           1, 2, .. k}
68      -> Apos o final da iteracao, vertice de numero k eh adicionado ao
            conjunto de vertices intermediarios {0, 1, 2, .. k} */
69      for (k = 0; k < V; k++) {
70          // Pega todos os vertices pela origem um por um
71          for (i = 0; i < V; i++) {
72              // Pega todos os vertices destinos acima do origem
```

```
                selecionado
            for (j = 0; j < V; j++) {
                // Se o vertice k esta no menor caminho de i para j,
                   entao o valor eh atualizado em dist[]i[j]
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    // Se a distancia de qualquer vertice para ele mesmo se torna
       negativa, entao existe um ciclo negativo
    for (int i = 0; i < V; i++) {
        if (dist[i][i] < 0) {
            return true;
        }
    }
    return false;
}

// Funcao principal para testar o programa
int main() {
    // Criacao do grafo
    int grafo[V][V] = { { 0, 1, 100, 1000},
                        { 1, 0, 6, 5},
                        { 100, 6, 0, 10},
                        { 1000, 5, 10, 0 } };

    int u = 0, v = 3, k = 2;
    if(cicloNegfloydWarshall(grafo) == false) {
        cout << menorCaminho(grafo, u, v, k);
    } else {
        cout << "Existe ciclo negativo";
    }

    return 0;
}
```

- A saída deste programa é:

```
6
```

4 ANÁLISE DE COMPLEXIDADE

Essa seção é destinada para demonstrar matematicamente o custo de cada uma das soluções apresentadas nesse artigo. Para calcular a complexidade, foi analisado o número de vezes que os laços *(estruturas de repetição)* são executados nos algoritmos. Todos os algoritmos apresentados tem como operação mais relevante o número de comparações entre elementos do grafo.

4.1 Complexidade da heurística gulosa para resolver o problema do caminho mais curto em um grafo com peso não negativo

Ao analisar o código é possível observar que existem três estruturas de repetição. Para descobrir a complexidade deve-se observar o funcionamento desses laços e verificar a quantidade de vezes que cada um deles é executado.

O primeiro laço, mais externo, é executado **n** vezes, já que vai de 0 até V. No segundo laço, a variável *i* é iniciada como 0 e vai até (V - 1), portanto é executada **(n - 1)** vezes. Por último, temos o terceiro laço que se encontra dentro do segundo e assim como o primeiro, é executado **n** vezes. Com isso obtemos a seguinte função:

$$\textbf{T(n):}\ (n-1)*(n)+n$$
$$\textbf{T(n):}\ n^2-n+n$$
$$\textbf{T(n):}\ n^2$$

Conclui-se que a ordem de complexidade do algoritmo é definido por $O(n^2)$.

4.2 Complexidade do algoritmo para determinar se existe um ciclo negativo em um grafo

Para o algoritmo de determinação do ciclo negativo, verifica-se as seguintes estruturas de repetição que permite o cálculo da complexidade :

Primeiro trecho

```
1 for (i = 0; i < V; i++) {
2     for (j = 0; j < V; j++) {
3         dist[i][j] = grafo[i][j];
4     }
5 }
```

Para esse trecho de código, tanto o laço interno quanto o externo nos são executados **n** vezes cada um. Logo é possível afimar que o laço como um todo será executado n^2 vezes.

Segundo trecho

```
for (k = 0; k < V; k++) {
    for (i = 0; i < V; i++) {
        for (j = 0; j < V; j++) {
            if (dist[i][k] + dist[k][j] < dist[i][j]) {
                dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

Para o segundo trecho de código, temos um laço externo e dois mais internos. O laço externo é executado **n** vezes, enquanto o primeiro e segundo laços internos também são executados **n** vezes. Dessa forma, o trecho todo será executado n^3 vezes.

Terceiro trecho

```
 for (j = 0; j < V; j++) {
    if (dist[i][k] + dist[k][j] < dist[i][j]) {
        dist[i][j] = dist[i][k] + dist[k][j];
    }
}
```

Por se tratar de um laço único, diferentemente dos anteriores que possuem laços internos, esse trecho será executado pela quantidade de **n** vezes.

Complexidade do algoritmo

Após a análise dos laços, podemos somar a quantidade de execuções de cada trecho e a seguinte função é obtida:

T(n): $n^3 + n^2 + n$

Conclui-se que a ordem de complexidade do algoritmo é definido por $O(n^3)$.

4.3 Complexidade do algoritmo de programação dinâmica para resolver o problema do caminho mais curto em um grafo com aresta negativa, mas sem ciclo negativo

Antes de apresentar a complexidade do algoritmo dinâmico, é necessário mostrar a complexidade do algoritmo, sem a utilização da técnica de programação dinâmica,no pior caso. A complexidade na notação Big-O é dada por O(V^k), ou seja, é exponencial para o tamanho k de arestas do grafo, o que não é de forma alguma uma solução interessante e eficiente.

Com a solução desenvolvida e apresentada na secão anterior, onde é utilizada a técnica de programação dinâmica, a complexidade cai de forma considerável. Com essa implementação, chegamos a O(V^3K), que é muito melhor se comparado a solução trivial.

Dessa forma, podemos concluir que o uso de programação dinâmica se torna interessante em casos como o descrito nessa seção. A lógica mais complicada é compensada ao obtermos uma complexidade fora do tempo exponencial, que na computação é considerada péssima graças ao tempo de execução que atinge resultados astronômicos a cada aumento no tamanho de entrada.

5 TESTES

A seção de testes tem o intuito de apresentar, em valores, diferenças no tempo de execução dos algoritmos quando os grafos de entrada possuem diferentes quantidades de vértices. Percebe-se que em determinados algoritmos, uma quantidade maior de vértices afeta a performance de maneira consideravel.

Cada tópico dessa seção vem acompanhada de uma tabela para comparação dos tempos de execução *(em milissegundos)* para cada quantidade de entrada testada. Os testes estão divididos pelo tipo de abordagem utilizada e as entradas usadads para o teste são matrizes quadradas de tamamnhos 64x64, 128x128, 256x256 e 512x512 contendo apenas números inteiros.

Após as tabelas, será apresentado um gráfico de comparação do tempo de execução entre as diferentes abordagens com as mesmas matrizes de entrada. O gráfico permitirá visualizar facilmente qual abordagem obeteve melhor desempenho para as diferentes entradas do teste.

5.1 Testes com a heurística gulosa para resolver o problema do caminho mais curto em um grafo com peso não negativo

Tabela 1 – algoritmoGuloso.cpp

Matriz de entrada	Tempo de execução
64x64	122 ms
128x128	334 ms
256x256	1.265 ms
512x512	3.000 ms

5.2 Testes com o algoritmo para determinar se existe um ciclo negativo em um grafo

Tabela 2 – cicloNegativo.cpp

Matriz de entrada	Tempo de execução
64x64	1.645 ms
128x128	11.079 ms
256x256	63.715 ms
512x512	469.000 ms

5.3 Testes com o algoritmo de programação dinâmica para resolver o problema do caminho mais curto em um grafo com aresta negativa, mas sem ciclo negativo

Tabela 3 – programDinamica.cpp

Matriz de entrada	Tempo de execução
64x64	7.460 ms
128x128	33.100 ms
256x256	283.742 ms
512x512	3.034.096 ms

5.4 Gráficos referentes aos tempos de cada algoritmo abordado em relação ao tamanho de suas entradas

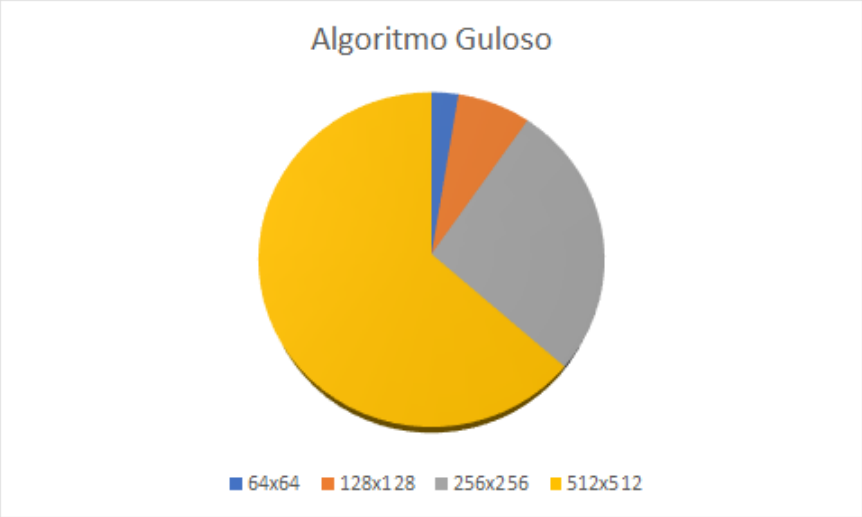

Figura 2 - Gráfico de setores para a abordagem gulosa

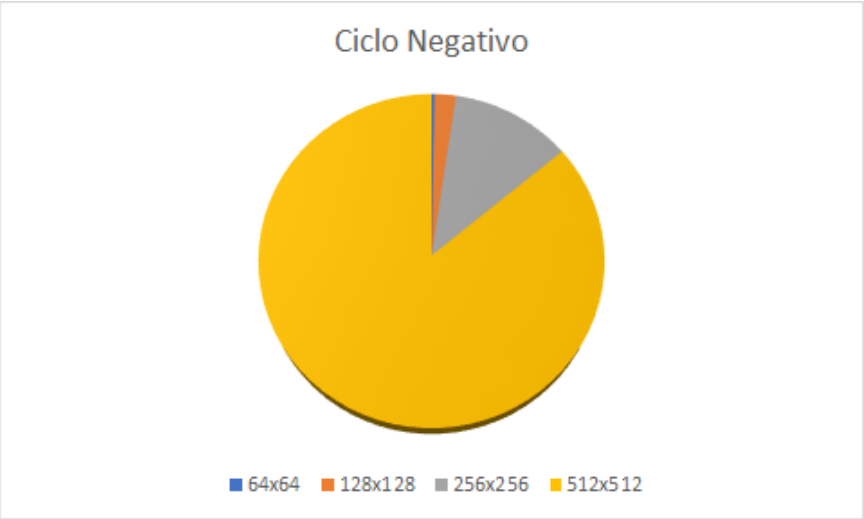

Figura 3 - Gráfico de setores para o ciclo negativo

Figura 4 - Gráfico de setores para a abordagem dinâmica

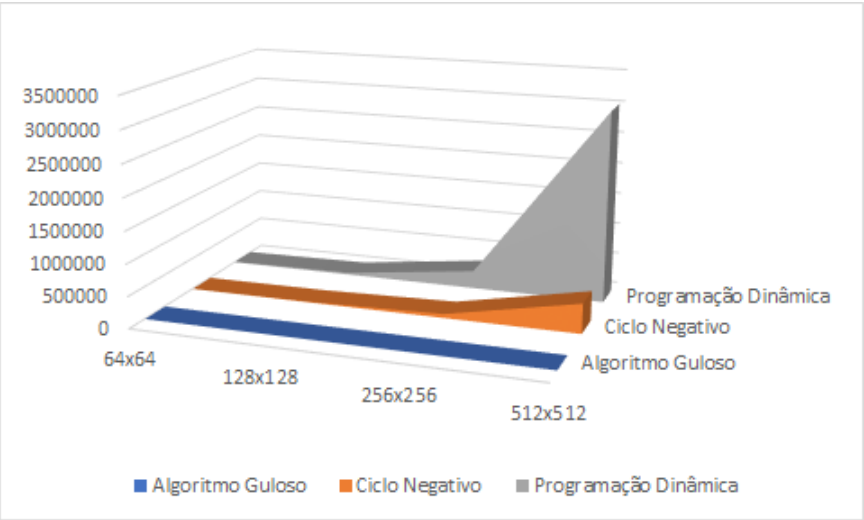

Figura 5 - Gráfico 3D para comparação entre todas as abordagens

6 CONCLUSÃO

Esta prática proporcionou um grande aprendizado a respeito do problema de menor caminho graças a quantidade de variações e diferentes soluções a serem colocadas em prática. Tal problema que possui grande importância na área de Ciência da Computação e foi necessário ir além do que foi apresentado em sala de aula para que o trabalho atingisse o nível desejado.

Diferentemente do imaginado, foi mais complexo a realização da heurística gulosa e do ciclo negativo do que o algoritmo dinâmico, embora este tenha demandado também um bom tempo de entendimento. Essa facilidade veio em decorrência do aproveitamento do código de ciclo negativo que foi replicado no algoritmo dinâmico, já que neste também se tornou necessário a verificação se existia ou não ciclos com pesos negativos no grafo executado.

A maior dificuldade em relação a implementação do algoritmo dinâmico foi devido a uma certa dificuldade no entendimento da técnica, o que demandou um tempo significante no decorrer do trabalho. Após o domínio do assunto, foi possível realizar o código com mais segurança e rapidez.

Foi possível ver na prática os extremos quanto ao tempo de execução dos algoritmos e compreender que um algoritmo otimizado realmente faz a diferença quando o tamanho da entrada se torna grande o suficiente. Ao mesmo tempo que um teste com a mesma entrada em algoritmos diferentes pode rodar em um tempo parecido, se mudarmos a entrada para um número maior, o tempo de execução de todos os algoritmos diferem-se em um valor considerável.

A diferença mais aparente que pôde ser visualizada foi no teste do algoritmo dinâmico para matriz 512x512 que demorou por volta de 3 segundos para terminar a execução, enquanto os demais encontraram-se na casa dos milissegundos. São essas observações que demonstram a importância de um trabalho prático para melhor compreensão de um assunto novo e, de certa forma, complexo.

7 REFERÊNCIAS

NOGUEIRA, Fernando. Problema do caminho mais curto. 2010. **Disponível em:** <http://www.ufjf.br/epd015/files/2010/06/caminhomaiscurto.pdf>. **Acesso em:** 10 mai. 2019.

ROCHA, Anderson. Algoritmos gulosos: definições e aplicações. 2004. **Disponível em:** <https://www.ic.unicamp.br/ rocha/msc/complex/algoritmosGulososFinal.pdf>. **Acesso em:** 11 mai. 2019.

FEOFILOFF, Paulo. Programação dinâmica. 2018. **Disponível em:** <https://www.ime.usp.br/ pf/analisedealgoritmos/aulas/dynamic-programming.html>. **Acesso em:** 13 mai. 2019.

RIBEIRO, Pedro. Distâncias Mínimas. 2014. **Disponível em:** <http://www.dcc.fc.up.pt/ pribeiro/aulas/daa1415/slides/8distancias06122014.pdf>. **Acesso em:** 16 mai. 2019.

FEOFILOFF, Paulo. Distâncias com pesos e o algoritmo de Dijkstra. 2016. **Disponível em:** <https://www.ime.usp.br/ pf/analisedealgoritmos/aulas/dijkstra.html>. **Acesso em:** 17 mai. 2019.

SATORU, Luiz. Algoritmo de Floyd. 2017. **Disponível em:** <http://www.ic.uff.br/ eoliveira/DisciplinasD.Sc/OtimizacaoemRedes/AlgoritmodeFloyd.pdf>. **Acesso em:** 17 mai. 2019.

SILVA, Fernando. Introdução à Complexidade de Algoritmos. 2013. **Disponível em:** <https://www.dcc.fc.up.pt/ fds/aulas/EDados/1314/Apontamentos/complexidade-1x2.pdf>. **Acesso em:** 20 mai. 2019.

CAMPELLO, Ricardo. Grafos Ponderados Caminhos Mínimos (Bellman-Ford). 2016. **Disponível em:** <http://wiki.icmc.usp.br/images/e/e9/GrafosVI.pdf>. **Acesso em:** 21 mai. 2019.

8 ANEXOS

A listagem dos programas implementados pela equipe estão citados abaixo para identificação em caso de necessidade de execução. Ambos se encontram dentro da pasta **Códigos**, sendo possível testá-los com diferentes entradas.

- **Heurística gulosa para resolver o problema do caminho mais curto em um grafo com peso não negativo:**
  algoritmoGuloso.cpp
- **Algoritmo para determinar se existe um ciclo negativo em um grafo:**
  cicloNegativo.cpp
- **Algoritmo de programação dinâmica para resolver o problema do caminho mais-curto em um grafo com aresta negativa, mas sem ciclo negativo:**
  programDinamica.cpp