Teoria de Grafos:
Uma exploração de conceitos e algumas aplicabilidades

Asafe Felipe dos Santos, E00731

Felipe Pereira Alves, E00900

Onofre Junior Rezende Salgado, E00917

Escola de Engenharia de Minas Gerais

Belo Horizonte - 2021

Escola de Engenharia de Minas Gerais, 2021, Páginas 1-13

Resumo: Este relatório visa expor as aplicações recorrentes de Teoria de Grafos nos dias atuais e a implementação de quatro algoritmos e seu funcionamento.

Palavras-chave: Teoria de Grafos, Algoritmos, dijkstra, bellman ford, kruskal, bfs, busca em largura

1. INTRODUÇÃO

O tema grafos nada tem a ver com gráficos, é um tema focado na aplicabilidade abrangente e impressionante com ênfase em modelagem.

Entendemos que o conhecimento matemático modelo e a capacidade de elaboração de testes para cenários complexos é extremamente importante para os alunos de graduação da atualidade. Neste contexto, a teoria de grafos dá um embasamento fundamental para que possamos criar mecanismos de identificação e solução de problemas de alto valor. É uma disciplina dinâmica que permite a aproximação com a realidade de nossas vivências.

Escolhemos alguns algoritmos que possam ser complementar na elaboração de teses na solução de problemas, sendo Dijkstra, Bellman Ford, Kruskal e BFS, Cada algoritmo possui a sua complementaridade e ataca problemas específicos fundamentais.

1.1. MOTIVAÇÃO

A Elaboração de modelos matemáticos para solução de problemas complexos reais de modo a facilitar a aplicação e diminuição do custo em sua execução é satisfatório para que se gaste tempo na criação e projeto de tais.

O conhecimento profundo dos trabalhos anteriormente criados é já existente e faz com que este tempo seja menor do que o necessário para conclusão de tais projetos. Sendo tal uma motivação forte para aquisição de tal conhecimento.

1.2. Objetivos

Este trabalho visa, contextualizar a aplicações práticas do que foi apresentado na disciplina de Teoria de grafos, implementação de quatro algoritmos e a aplicabilidade real de cada algoritmo.

2. ALGORITMOS DA TEORIA DE GRAFOS

Escolhemos quatro algoritmos, sendo eles, Dijkstra e Bellman Ford, que são utilizados para caminhamento mínimo em grafos, Krustal que é usado para encontrar árvores geradoras mínimas, e por último, o algoritmo BFS ou busca em largura, que é aplicado a busca em amplitude em grafos.

2.1. Algoritmo de Dijkstra

O objetivo desse algoritmo é: encontrar o caminho de custo mínimo entre os vértices de um grafo com arestas de peso não negativo. Para isso o Algoritmo recebe um grafo G com custos positivos nos arcos e um vértice s, é a partir desse vértice que ele faz crescer uma subárvore baseada em G que segue até englobar todos os vértices ao alcance de s.

O pseudocódigo a seguir contém o modo de operar do algoritmo que*:* recebe um grafo G com custos positivos nos arcos e um vértice s. Calcula e armazena em um vetor com raízes. As distâncias a partir de s são armazenadas no vetor em um outro vetor.

```
void GRAPHcptD1( Graph G, vertex s,
                 vertex *pa, int *dist)
{
   bool mature[1000];
   for (vertex v = 0; v < G->V; ++v)
      pa[v] = -1, mature[v] = false, dist[v] = INT_MAX;
   pa[s] = s, dist[s] = 0;

   while (true) {
      // escolha de y:
      int min = INT_MAX;
      vertex y;
      for (vertex z = 0; z < G->V; ++z) {
         if (mature[z]) continue;
         if (dist[z] < min)
            min = dist[z], y = z;
      }
      if (min == INT_MAX) break;
      // atualização de dist[] e pa[]:
      for (link a = G->adj[y]; a != NULL; a = a->next) {
         if (mature[a->w]) continue;
         if (dist[y] + a->c < dist[a->w]) {
            dist[a->w] = dist[y] + a->c;
            pa[a->w] = y;
         }
      }
      mature[y] = true;
   }
}
```

A ordem desse algoritmo é: **[O(m + n log n)]**

Quando se trata de problemas reais, o algoritmo pode ser muito bem aplicado no tráfego urbano e para a solução de problemas de logística.

(Código Fonte)

```
class GraphDijkstra:

    def __init__(self, vertices):
        self.vertices = vertices
        self.graph = [[0] * self.vertices for i in range(self.vertices)]

    def add_edge(self, origin, coming, weight):
        self.graph[origin][coming] = weight
        self.graph[coming][origin] = weight

    def view_dijkstra(self):
        print('Visualização da matriz')
        for i in range(self.vertices):
            print(self.graph[i])

    def dijkstra(self, origin):
       # minus one represents infinite | Origin
        result = [[-1, 0, 0] for i in range(self.vertices)]

        # Add value the origin
        result[origin - 1] = [0, origin, origin]

        # Starting heap minimum
        heapMinimum = HeapMinimum()

        # Add node the heap
        heapMinimum.add_nodes(0, origin)

        # Receiving the start heap, and verify which the minimum path
        while heapMinimum.size() > 0:
            distance, vertice = heapMinimum.remove_node()
            for i in range(self.vertices):
                if self.graph[vertice - 1][i] != 0:
                    if (result[i][0] == -1) or (result[i][0] > distance + self
.graph[vertice - 1][i]):
                        result[i] = [distance + self.graph[vertice - 1
][i], vertice, i+1]

                        heapMinimum.add_nodes(distance + self.graph[vertice - 1
][i], i + 1)


        for k, v in enumerate(result):
            print(f'saindo do vertice: {v[1]}, indo para vertice {v[2]}
, o custo atual é: {v[0]}')
        print(f'A menor distância do vertice {result[0][1]} até o vertice {result[-1
][2]} é: {result[-1][0]}')
```

2.2. Algoritmo de Bellman Ford

O objetivo do algoritmo é encontrar o caminho mais curto de uma única origem no qual os custos das arestas podem ser negativos. Para isso, o algoritmo recebe um grafo G,que tem custos nas arestas, e um vértice s, a partir daí ele procura encontrar o caminho mais barato de G que tenha s como raiz.

6

O pseudocódigo a seguir contém o modo de operar do algoritmo que: Recebe um vértice s de um grafo G com custos nos arcos e supõe que todos os vértices estão ao alcance de s. Caso G tenha um ciclo negativo ele responde que é falso. Caso não tenha, ele vai armazenar em um vetor.

```
bool GRAPHoptBF( Graph G, vertex s, vertex *pa, int *dist)
{
   QUEUEinit( G->A);
   bool onqueue[1000];
   for (vertex u = 0; u < G->V; ++u)
      pa[u] = -1, dist[u] = INT_MAX, onqueue[u] = false;
   pa[s] = s, dist[s] = 0;
   QUEUEput( s);
   onqueue[s] = true;
   vertex V = G->V; // pseudovértice
   QUEUEput( V); // sentinela
   int k = 0;

   while (true) {
      vertex v = QUEUEget( );
      if (v < V) {
         for (link a = G->adj[v]; a != NULL; a = a->next) {
            if (dist[v] + a->c < dist[a->w]) {
               dist[a->w] = dist[v] + a->c;
               pa[a->w] = v;
               if (onqueue[a->w] == false) {
                  QUEUEput( a->w);
                  onqueue[a->w] = true;
               }
            }
         }
      } else {
         if (QUEUEempty( )) return true; // A
         if (++k >= G->V) return false; // B
         QUEUEput( V); // sentinela
         for (vertex u = 0; u < G->V; ++u)
            onqueue[u] = false;
      }
   }
}
```

A ordem desse algoritmo é: **[O(VxE)]**

Quando se trata de problemas reais, o algoritmo pode ser muito bem aplicado em assuntos relacionados a movimentações financeiras assim como em protocolos de roteamento em redes de dados.

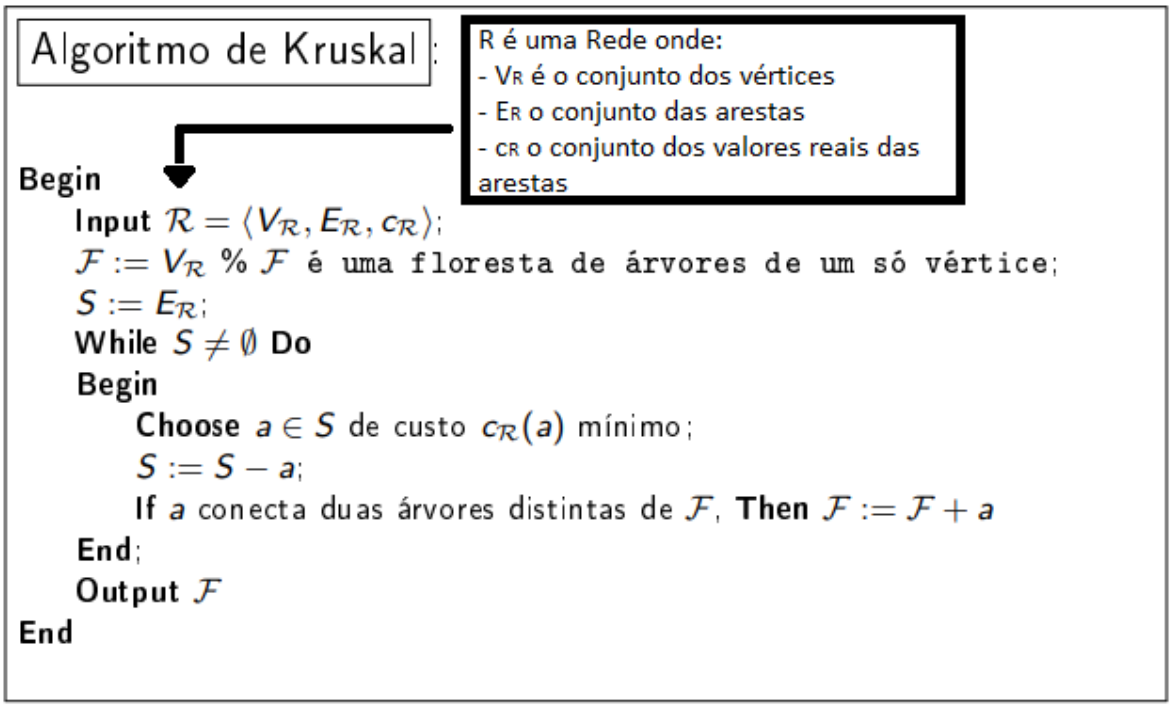
(Código Fonte)

```python
class BellmanFord:

    def __init__(self, vertices):
        self.V = vertices
        self.graph = []


    def add_edge(self, s, d, w):
        if w != 0:
            self.graph.append([s, d, w])


    def print_solution(self, dist):
        print("Vertice, distancia ate o vertice de saida")
        for i in range(self.V):
            print(f"{i}\t\t{dist[i]}")

    def bellman_ford(self, src):

        dist = [float("Inf")] * self.V
        # Mark the source vertex
        dist[src] = 0

        for _ in range(self.V - 1):
            for s, d, w in self.graph:
                if dist[s] != float("Inf") and dist[s] + w < dist[d]:
                    dist[d] = dist[s] + w


        for s, d, w in self.graph:
            if dist[s] != float("Inf") and dist[s] + w < dist[d]:
                print("Graph contains negative weight cycle")
                return

        self.print_solution(dist)
```

2.3. Algoritmo de kruskal

O objetivo desse algoritmo é: Determinar a árvore geradora mínima de um grafo que não é dirigido e no qual as arestas têm custos. Para isso ele vai selecionar sucessivamente as arestas de menor custo do grafo, sem gerar circuitos, até ter produzido uma árvore.

O pseudocódigo a seguir contém o modo de operar do algoritmo que: Escolhe uma aresta que localmente tem o menor custo e acrescenta ela a F sem se importar com o resultado global disso.



A ordem desse algoritmo é: [O(|E| log(|V|))]

Quando se trata de problemas reais, o algoritmo pode ser muito bem aplicado em esquemas de bancos de dados e algoritmos genéticos.

(Código Fonte)

```python
class GraphKrustal:
    def __init__(self, vertex):
        self.V = vertex
        self.graph = []

    def add_edge(self, u, v, w):
        if w != 0:
            self.graph.append([u, v, w])


    def search(self, parent, i):
        if parent[i] == i:
            return i
        return self.search(parent, parent[i])

    def apply_union(self, parent, rank, x, y):
        xroot = self.search(parent, x)
        yroot = self.search(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1


    def kruskal(self):
        result = []
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.search(parent, u)
            y = self.search(parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.apply_union(parent, rank, x, y)
        for u, v, weight in result:
            print("Edge:",u, v,end =" ")
            print("-",weight)
```

2.4. Algoritmo de Busca em amplitude

O objetivo desse algoritmo é: Visitar todos os vértices uniformemente. Para isso ele visita primeiro todos os vértices adjacentes ao vértice inicial e depois visita os vértices de outros níveis.

O pseudocódigo a seguir contém o modo de operar do algoritmo que: Visita todos os vértices do grafo G que estão ao alcance do vértice e registra em um vetor a ordem em que os vértices são descobertos.

```
void GRAPHbfs( Graph G, vertex s)
{
   int cnt = 0;
   for (vertex v = 0; v < G->V; ++v)
      num[v] = -1;
   QUEUEinit( G->V);
   num[s] = cnt++;
   QUEUEput( s);

   while (!QUEUEempty( )) {
      vertex v = QUEUEget( );
      for (link a = G->adj[v]; a != NULL; a = a->next)
         if (num[a->w] == -1) {
            num[a->w] = cnt++;
            QUEUEput( a->w);
         }
   }
   QUEUEfree( );
}
```

A ordem desse algoritmo é: [O(|V|)]

Quando se trata de problemas reais, o algoritmo pode ser aplicado na teoria dos 6 graus de comparação.

(Código Fonte)

```python
class GraphBfs:
    def __init__(self):
        self.graph = DefaultDict(list)


    def add_edge(self, index, vertex):
        self.graph[index].append(vertex)


    def BFS(self, node):
        visited = [False] * (len(self.graph))

        listQueue = []

        listQueue.append(node)
        visited[node] = True

        while listQueue:
            node = listQueue.pop(0)
            print(f'visitando o vertice: {node}', ' ')

            for i in self.graph[node]:
                if visited[i] == False:
                    listQueue.append(i)
                    visited[i] = True
```

3. APLICAÇÃO DA TEORIA DE GRAFOS NO CONTEXTO ATUAL

Vemos de modo claro em nosso mundo diversas aplicações que utilizam Teoria de Grafos mesmo que esta seja de modo implícito e de difícil visualização, Assim podemos identificar de modo mais claro aplicações em dimensionamentos de redes de fibra ótica, escolhas de locais ideias para colocação de antenas, softwares disponibilizando a mesma banda para diversos números distintos sem provocar linha cruzada entre outras.

Todavia a nossa comunidade exploratória dedicada à elaboração deste conteúdo decidiu por duas aplicações em conceitos de softwares que explora muito bem as teorias discutidas sendo o Git e Neo4J.

O Git forma um grafo direcionado caracterizado por nós que guarda informações importantes e pontos de junções fundamentais para o usuário. Podemos até ter uma falsa impressão da formação de uma árvore com alguns galhos ramificados que indicam trabalhos paralelos ao tronco principal facilitando assim o trabalho em times. Esta falsa sensação vem pelo modo ao qual vemos a exibição do trabalho no decorrer do tempo nos softwares gráficos tal como GitHub, em sua essência o git é bem mais próximos das teorias utilizadas para formação de redes sociais tradicionais, porém o foco desta rede que ser destoar das demais. assim ele forma nós com poucos seguidores e o conteúdo destes nós e códigos fontes.

O Neo4j e um banco de dados NoSQL baseado em teoria de grafos ao qual podemos utilizar modelagem como estrutura principal na formação de sua estrutura de armazenamento sendo os vértices ou nós nossas entidades é as arestas nossos relacionamentos onde os relacionamentos guardam dados e indicar uma direção, podendo ser ela direcionada ou não.

4. CONCLUSÃO

Diante de tudo que foi exposto neste relatório, vimos a importância da disciplina de Teoria de Grafos em aplicações reais, onde por meio das modelagens baseadas em grafos conseguimos entender e resolver problemas, vimos também como funcionam quatro algoritmos, Dijkstra, Bellman Ford, Kruskal e BFS e como cada um deles resolve um problema real. Assim sendo, este trabalho foi de suma importância para cada integrante da comunidade selecionada para exploração do tema para o entendimento da disciplina e suas aplicações práticas.

5. REFERÊNCIAS

DEO, N. Graph Theory with Applications to Engineering and Computer Science. Prentice-Hall, 1974.

NETTO, P. O. B. Grafos: Teoria, Modelos, Algoritmos, 4ª edição, Edgard Blucher, 2006.

DIESTEL, R. Graph Theory, Springer, 2006.

CORMEN, T.H., LEISERSON, C.E., RIVEST, R.L, STEIN, C. Algoritmos, Teoria e Prática, Campus, 2002.

Resumos LEIC-A

Material disponibilizado pelo professor Marcos Rodrigues