Apostila Resumida - Teoria dos Grafos

Fonte: http://faculty.ycp.edu/~dbabcock/PastCourses/cs360/lectures/
Adaptação: **Prof. Acauan C. Ribeiro**

Alunos: GUILHERME LUCAS PEREIRA BERNARDO

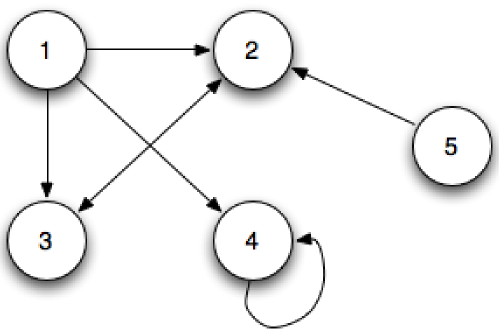
Este trabalho se propõe a fazer uma breve revisão sobre os principais assuntos relacionados a Grafo vistos na disciplina DCC 405 – Estrutura de Dados II.

1. Definição de Grafo

Assumimos que um **grafo G(V, E)** é representado como um par de conjuntos finitos onde **V é o conjunto de vértices e E é o conjunto de arestas**.

1.1 Gráfico dirigido (dígrafo)

Em um *grafo direcionado*, as arestas são representadas **por pares ordenados de vértices (u,v)** e mostrados de maneira gráfica como setas direcionadas (um vértice pode ser conectado a si mesmo por meio de um auto-loop).



Uma aresta *(u,v)* é **incidente** de(**sai**) *u* e é incidente de(**entra**) v. Se um grafo contém uma aresta (u,v), então v é adjacente a u e é representado notadamente como u→v. Note que v ser adjacente a u não implica que u seja adjacente a v, a menos que a aresta (v,u) ∈ E. Assim (u,v) e (v,u) são arestas distintas em um grafo direcionado.

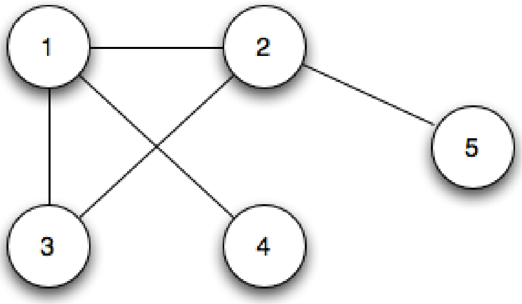
Dizemos que u e v são vizinhos se (u,v) ∈ E ou (v,u) ∈ E.

Para cada vértice definimos o **grau** de saída como o número de arestas que saem do vértice, o grau de entrada como o número de arestas que entram no vértice e o grau como o grau desaída + o grau de entrada (ou seja, o número total de arestas no vértice). Se um vértice tem grau= 0, então o vértice é isolado.

Se o grafo direcionado não tem auto-loops, então é um **grafo direcionado simples**.

1.2 Gráfico não direcionado

Em um grafo não direcionado, as arestas são representadas por pares não ordenados de vértices. Assim (u,v) e (v,u) representam a mesma aresta e são mostrados de maneira gráfica como simplesmente uma linha de conexão (observe que grafos não direcionados podem não conter auto-loops).



Uma aresta (u,v) é incidente em u e v, e u e v são adjacentes um ao outro.

O grau é o número de arestas incidentes em um vértice.

Para converter um grafo não direcionado em um direcionado, basta substituir cada aresta (u,v) por (u,v) e (v,u). Por outro lado, para converter um grafo direcionado em um não direcionado, substitua cada aresta (u,v) ou (v,u) por (u,v) e remova todos os auto-loops.

1.3 Caminhos

Um caminho de comprimento k de **u** para **u'** é uma sequência de vértices $< v_0, v_1, \ldots, v_k >$ com **u** $= v_0$, $u' = v_k$, e $(v_{i-1} , v_i) \in E$.

Se existe um caminho p de u para u', então u' é alcançável a partir de u (denotado $u \rightsquigarrow u'$ se G for um grafo direcionado).

O **caminho** é simples se todos os vértices forem distintos.

Um subcaminho é uma subsequência contígua $< v_i , v_{i+1} , \ldots, v_j >$ com $0 \leq i \leq j \leq k$.

Um **ciclo** é um caminho com $v_0 = v_k$ (e também é simples se todos os vértices, exceto os pontos finais, forem distintos). Um grafo **acíclico** é um grafo sem ciclos.

1.4 Componentes conectados

Em um grafo não direcionado, um **componente conectado** é um *subconjunto* de vértices que são todos alcançáveis uns pelos outros. O **grafo é conexo** se contiver exatamente um componente conexo, ou seja, todos os vértices são alcançáveis a partir de todos os outros.

Em um grafo direcionado, um componente fortemente conectado é um subconjunto de vértices mutuamente alcançáveis, ou seja, existe um caminho entre quaisquer dois vértices no conjunto.

1.5 Gráficos Especiais

Um **grafo completo** é um grafo não direcionado onde todos os vértices são adjacentes a todos os outros vértices, ou seja, existem arestas entre cada par de vértices.

Um **grafo bipartido** é um grafo não direcionado que pode ser particionado em V_1 e V_2 tal que para cada aresta $(u, v) \in E$ ou

$u \in V_1$ e$v \in V_2$ OU $u \in V_2$ e$v \in V_1$ _ _

ou seja, o grafo pode ser separado de forma que as únicas arestas estejam entre vértices em diferentes subconjuntos.

Uma **floresta** é um grafo acíclico não direcionado. Se também estiver conectado, então é uma **árvore** .

Um grafo **acíclico direcionado é conhecido como DAG** .

2. Representação do gráfico

Dois métodos comuns para implementar um gráfico em software é usar uma **lista de adjacências** ou uma **matriz de adjacências** .

2.1 Lista de Adjacência

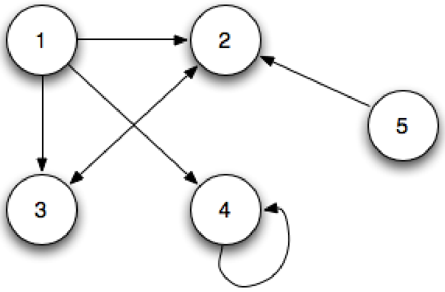
Em uma implementação de lista de adjacência, nós simplesmente armazenamos os vértices adjacentes (ou seja, arestas) para cada vértice em uma lista encadeada denotada por Adj [u]. Se somarmos os comprimentos de todas as listas de adjacência, obtemos

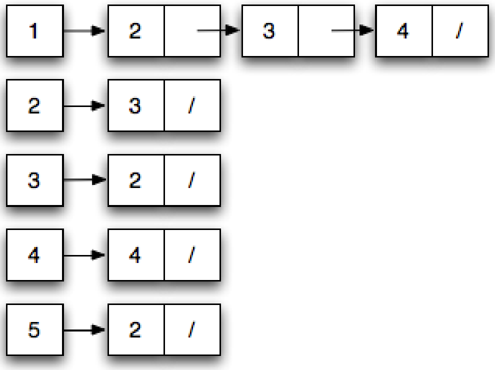
$$\sum |\mathrm{Adj}[u]| = \begin{cases} |E| & \text{if directed} \\ 2|E| & \text{if undirected} \end{cases}$$

⇒ O armazenamento $\Theta(V + E)$ é necessário.

Esta representação é boa para gráficos esparsos onde $|E| \ll |V|^2$. Uma desvantagem é que determinar se uma aresta $(u, v) \in E$ requer uma busca de lista ⇒ $\Theta(V)$.

Para o grafo direcionado original, a lista de adjacências seria





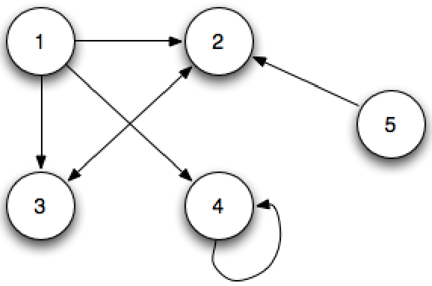
2.2 Matriz de adjacência

Em uma implementação de matriz de adjacência, armazenamos as arestas em uma matriz VxV A como valores binários ou números reais para arestas ponderadas.

⇒ O armazenamento $\Theta(V^2)$ é necessário (independente de E).

Esta representação é boa para gráficos densos onde $| E | \approx | V |^2$. A vantagem é que leva apenas $\Theta(1)$ tempo para determinar se uma aresta $(u , v) \in E$, pois é simplesmente um acesso a um elemento da matriz. Se o gráfico não é direcionado, então $\mathbf{A = A^T}$, portanto, apenas a metade triangular superior precisa ser armazenada.

Para o grafo direcionado original, a matriz de adjacência seria



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

3.1 Breadth-First Search (BFS) – Busca em Largura

Agora que revisamos a terminologia básica associada aos grafos, o primeiro algoritmo que investigaremos é ***a busca em largura*** (BFS). Este algoritmo é usado para encontrar os caminhos mais curtos (por número de arestas) para cada vértice alcançável a partir de um determinado vértice dado.

3.1.1 Problema

Dado um vértice de origem **s**, encontre os *caminhos mais curtos* (em termos de número de arestas) para cada vértice alcançável por **s**.

3.1.2 Solução

O procedimento que usaremos encontrará todos os vértices alcançáveis a uma **distância k** antes de descobrir os vértices alcançáveis a uma **distância k+1**. Em última análise, o algoritmo produzirá uma árvore em largura com **s** como **raiz**.

Durante a execução do algoritmo, os vértices serão coloridos (denotados por u.color). As cores representam o estado atual do vértice da seguinte forma

- branco - o vértice não foi descoberto (ou seja, atualmente nenhum caminho foi encontrado para o vértice)
- cinza - o vértice foi descoberto e está na *fronteira*, ou seja, pode haver outros vértices que podem ser descobertos
- preto - o vértice foi descoberto e foi completamente pesquisado

O algoritmo também usa dois campos adicionais para cada vértice

u.π - vértice predecessor

u.d - distância quando o vértice é descoberto pela primeira vez (e subsequentemente é a distância mais curta da fonte)

Empregaremos uma **fila Q** que rastreará quais vértices estão atualmente sob descoberta. Assim, os vértices que ainda não foram colocados em Q serão brancos, os que estiverem em Q serão cinzas e os que foram removidos de Q serão pretos .

3.1.3 Algoritmo

O algoritmo para busca em largura é

```
BFS(G,s)
para cada vértice u ∈ GV - {s}
u.cor == BRANCO
ud = INF
u.pi = NIL
s.cor = CINZA
dp = 0
s.pi = NIL
Q = ∅
ENFILEIRA(Q,s)
enquanto Q ≠ ∅
u = DEQUEUE(Q)
para cada v ∈ G.Adj[u]
se v.cor == BRANCO
v.cor = CINZA
vd = ud + 1
v.pi = u
```

```
17. ENFILEIRA(Q,v)
18. u.cor = PRETO
```

Basicamente, o algoritmo realiza as seguintes operações:

1. Inicialize Q com o vértice de origem s
2. Retire o vértice principal u de Q e marque como **preto**
3. Enfileire todos os vértices brancos adjacentes a u marcando-os como **cinza** , defina sua distância para a distância de u + 1 e defina seu π para u
4. Repita 2-3 até Q = ∅

3.1.4 Análise

Como nenhum vértice é enfileirado/retirado da fila mais de uma vez ⇒ O(V)

Cada lista de adjacências é escaneada apenas uma vez (quando o vértice é desenfileirado) com tamanho máximo o número total de arestas ⇒ O(E)

Sobrecarga de inicialização ⇒ O(V)

Assim, o tempo total de execução para BFS é

$$\Rightarrow O(V)+O(E)+O(V)=O(V+E)$$

Pode-se provar que o algoritmo produz os **caminhos mais curtos** (em termos do número mínimo de arestas) para todos os vértices alcançáveis da fonte s. Esses caminhos podem ser representados por uma árvore em largura que é dada pelo subgrafo predecessor

$$\begin{aligned}&G_\pi(V_\pi, E_\pi)\,\text{where}\\&V_\pi=\{v\in V:\pi(v)\neq nil\}\cup\{s\}\\&E_\pi=\{(\pi(v),v):v\in V_\pi-\{s\}\}\end{aligned}$$

Em outras palavras, o grafo predecessor contém todos os vértices com predecessores alcançáveis mais a fonte e todas as arestas predecessoras.

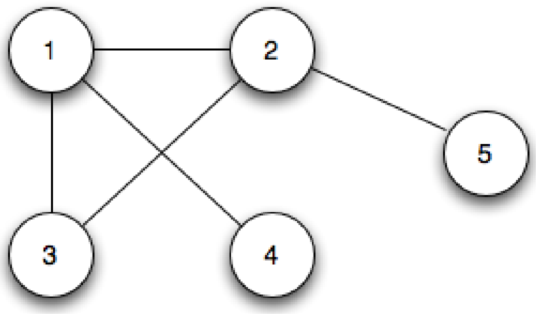
Além disso, pode-se mostrar que como o subgrafo predecessor é uma árvore , pelo teorema B.2 do CLRS

$$|E_\pi|=|V_\pi|-1$$

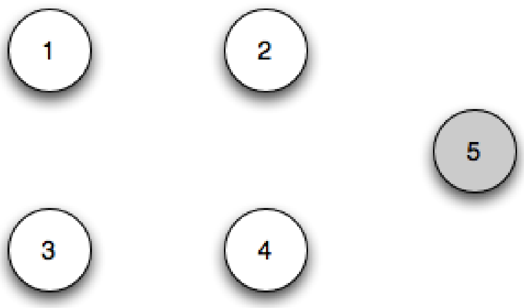
A árvore predecessora pode ser percorrida (usando os π's) para fornecer o caminho mais curto de s para v .

3.1.5 Exemplo

Considere o gráfico de cinco nós (não direcionado)



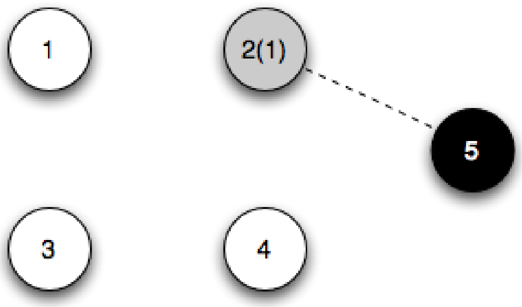
Se selecionarmos o vértice 5 como fonte, então d[5]=0, π[5]=/, Q={5}, então a inicialização nos dá



Q = {5}

	1	2	3	4	5
d	∞	∞	∞	∞	0
π	/	/	/	/	/

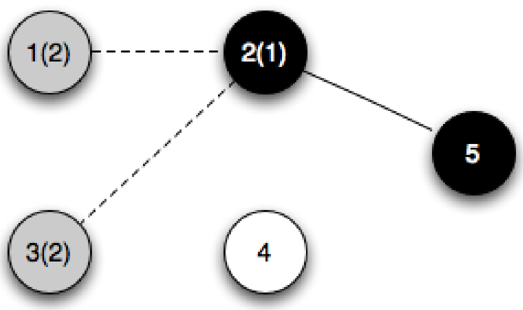
Iteração 1 : desenfileirar o vértice 5 e enfileirar o vértice 2



Q = {2}

	1	2	3	4	5
d	∞	1	∞	∞	0
π	/	5	/	/	/

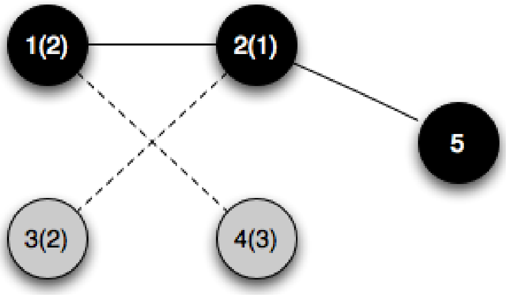
Iteração 2 : desenfileirar vértice 2 e enfileirar vértices 1,3



Q = {1,3}

	1	2	3	4	5
d	2	1	2	∞	0
π	2	5	2	/	/

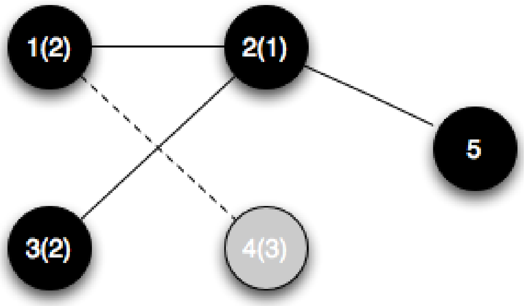
Iteração 3 : desenfileirar o vértice 1 e enfileirar o vértice 4



Q = {3,4}

	1	2	3	4	5
d	2	1	2	3	0
π	2	5	2	1	/

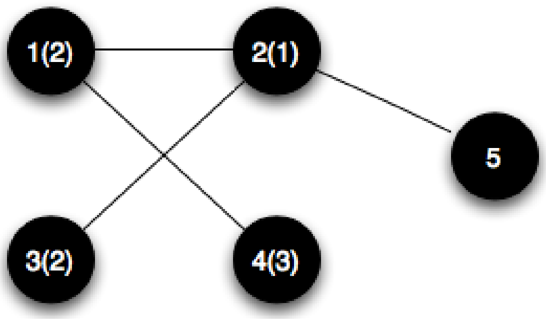
Iteração 4 : desenfileirar o vértice 3 e enfileirar nenhum vértice



Q = {4}

	1	2	3	4	5
d	2	1	2	3	0
π	2	5	2	1	/

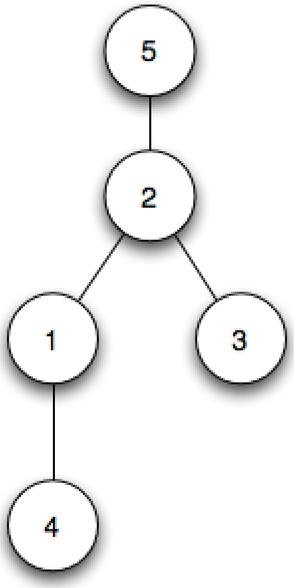
Iteração 5 : desenfileira o vértice 4 e não enfileira nenhum vértice (deixando assim a fila vazia)



Q = ∅

	1	2	3	4	5
d	2	1	2	3	0
π	2	5	2	1	/

O grafo predecessor final, ou seja, árvore em largura, para este BFS é



Prova 02 – DCC 405 – Estrutura de Dados II
----------------- FAZER A PARTIR DAQUI -----------------

3.2 Depth-First Search (DFS) – Busca em Profundidade

Enfim vimos como funciona a busca em largura, chegou a vez de vermos como funciona o algoritmo de **busca em profundidade**(DFS), usado para buscar o vértice “mais profundo” no grafo sempre que possível.

3.2.1 Problema

O problema da Busca em Profundidade é que ela só serve para nos mostrar como o grafo é feito, nos ajudando apenas a compreendê-lo e não necessariamente resolvendo algum problema.

3.2.2 Solução

A lógica básica implica em primeiro criar uma pilha e empilhar o nó raiz no topo. Enquanto a pilha não estiver vazia, percorra os nós adjacentes do nó raiz. Caso um nó não seja o nó buscado, e ainda não tenha sido visitado, empilhe-o.

3.2.3 Algoritmo

```
DFS(G)
for ∀u ∈ V[G]  do
        cor[u] ← BRANCO
        π[u] ← NIL
tempo ← 0
for ∀u ∈ V[G] do
        if cor[u] = BRANCO then
                VisitaDFS(u)
```

```
VisitaDFS(u)
cor[u] ← CINZA
d[u] ← tempo ← tempo + 1
for ∀v ∈ Adj[u] do
        if cor[v] = BRANCO then
                π[v] ← u
                VisitaDFS(v)
cor[u] ← PRETO
F[u] ← tempo ← tempo + 1
```

3.2.4 Análise

Branco, cinza e preto são cores utilizadas como sentinelas para mostrar o estado de cada nó durante a execução do algoritmo. Branco significa que o nó ainda não foi visitado. Cinza significa que o nó está sendo processado, ou está na fila pronto para um posterior processamento. Preto significa que todos os adjacentes daquele nó já foram pintados de Cinza.

3.2.5 Exemplo

Aqui temos o exemplo de um grafo com 6 vértices em uma classe chamada Graph, temos a função addEdge que faz um append na lista de adjacência do grafo toda vez q é chamada quando queremos adicionar uma aresta.

```
class Graph:

    def __init__(self,vertices):
        self.V=vertices #numero de vertices
           self.graph=[]#vetor vazio definido para armazenar as arestas entre vértices

    def addEdge(self, u, v):
        self.graph.append([u,v])

    def DFSUtil(self, v, visited):
       visited[v]=True
       print(v, end = ' ')

       for i in self.graph:
          if v==i[0]:
             if visited[i[1]] == False:
                self.DFSUtil(i[1], visited)
```

```
    def DFS(self, v):
       visited = [False] * (self.V)

       #ordena conforme primeiro vértice das arestas
       self.graph.sort()

       self.DFSUtil(v, visited)

g=Graph(6)

#A=0
g.addEdge(0,1)
g.addEdge(0,2)
g.addEdge(0,3)
g.addEdge(0,4)

#B=1
g.addEdge(1,2)
g.addEdge(1,3)
g.addEdge(1,4)

#C=2
g.addEdge(2,3)
g.addEdge(2,4)

#D=3
g.addEdge(3,4)
g.addEdge(3,5)
#E=4

g.DFS(0)
```

4. Árvore geradora mínima

Seja G um grafo não direcionado com custos nas arestas. O custo de cada aresta pode ser positivo ou negativo. O custo de um subgrafo não direcionado T de G é a soma dos custos das arestas de T.
A árvore geradora mínima de G é qualquer árvore geradora de G que tenha o menor custo. Em outras palavras, a árvore geradora T de G é mínima se não houver outras árvores geradoras que custem menos que T. O Minimum Spanning Tree também é conhecido como a abreviação MST para Minimum Spanning Tree.
Problema MST: Dado um grafo não direcionado com custos nas arestas, encontre a árvore geradora mínima para o grafo.
Claramente, este problema tem solução se e somente se o grafo for conexo. Outra observação óbvia: se todas as arestas têm o mesmo custo, então toda árvore geradora é um MST.

4.1 Algoritmo de Prim

Dado um grafo conectado não direcionado G com custo nas arestas, o algoritmo de Prim cresce uma subárvore de G até se tornar uma árvore geradora. Ao final do processo, a árvore é um MST.

Para discutir os detalhes, precisamos de alguma terminologia. Suponha que T seja uma subárvore (não necessariamente abrangente) de G . Uma aresta de T é um corte, e sua aresta é o conjunto de vértices de T. Em outras palavras, uma aresta de T é o conjunto de todas as arestas de G com uma extremidade fora de T e a outra de T.

Agora podemos descrever o algoritmo com precisão. Cada iteração começa com uma subárvore T. No início da primeira iteração, T consiste em um vértice. O processo iterativo consiste no seguinte: enquanto a aresta de T não estiver vazia,

1. escolha uma aresta da franja que tenha custo mínimo,
2. seja x-y a aresta escolhida, com x em T,
3. acrescente a aresta x-y e o vértice y a T.

Como se vê, o algoritmo tem caráter *guloso*: em cada iteração, abocanha a aresta mais barata da franja sem se preocupar com o efeito global, a longo prazo, dessa escolha. A prova de que essa estratégia está correta decorre do critério de minimalidade baseado em cortes.

4.1.1 Algoritmo

```
Função Prim(Grafo): inteiro
{
    Declare:
        V[]: inteiro
        E[]: inteiro
        i: inteiro
        k: inteiro
        n_acabou: booleano
        menor: inteiro
        custo: inteiro
        aux: inteiro
        total: inteiro
    i<-0
    k<-1
    V[0]<-0
    montaMatrizAdjacência();
    Enquanto(n_acabou) faça
    {
        menor<- +∞
        aux <- 0
        Enquanto(aux < k) faça
        {
            para j<-V[aux]+1 até tam-1
            {
                se((E[V[aux]][j] < menor) E (j não pertencer a V) então
                {
                    menor<-j
                    custo<-A[V[aux]][j]
                }fim-se
                aux<-aux + 1
```

```
            }fim-para
        V[k]<-menor
        E[k-1]<-custo
        k<-k + 1
        se(k = tam) então
        {
            n_acabou<-falso
        } senão
        {
            i<-menor
        } fim-se
        }fim-enquanto
    }fim-enquanto
    para n<-0 até tam-2 faça
    {
        total <- total + E[n]
        retorne total;
    } fim-para
}FIM
```

Algoritmo de Prim	Características
	-Possui um ponto de partida; -Não pode ser orientado; -Os pesos podem ser iguais; -Combinação linear entre os vértices; -Grafo Conexo; -Acíclico;

4.1.2 Exemplo

```python
# Prim's Algorithm in Python

INF = 9999999
# number of vertices in graph
N = 5
#creating graph by adjacency matrix method
G = [[0, 19, 5, 0, 0],
     [19, 0, 5, 9, 2],
     [5, 5, 0, 1, 6],
     [0, 9, 1, 0, 1],
     [0, 2, 6, 1, 0]]

selected_node = [0, 0, 0, 0, 0]

no_edge = 0

selected_node[0] = True

# printing for edge and weight
print("Edge : Weight\n")
while (no_edge < N - 1):
    minimum = INF
```

```
    a = 0
    b = 0
    for m in range(N):
        if selected_node[m]:
            for n in range(N):
                if ((not selected_node[n]) and G[m][n]):
                    # not in selected and there is an edge
                    if minimum > G[m][n]:
                        minimum = G[m][n]
                        a = m
                        b = n
    print(str(a) + "-" + str(b) + ":" + str(G[a][b]))
    selected_node[b] = True
    no_edge += 1
```

4.2 Algoritmo de Kruskal

O algoritmo de Kruskal funciona encontrando um subconjunto das arestas de um dado grafo, cobrindo todos os vértices presentes no grafo de modo que formem um MST e a soma dos pesos das arestas seja a menor possível.

Algoritmo de Kruskal	Característica
	-Não possui um ponto de partida; -Não pode ser orientado; -Acíclico;

4.2.2 Algoritmo

```
Kruskal (G=(V,E), custos c):
        # ordene as arestas em ordem crescente de custo
    T = {}
    Para cada aresta e = (u, v) em ordem crescente de custo:
        se T unida com ‘e’ não forma um ciclo:
            Adicione e em T
    devolva T
```

4.2.3 Exemplo

```
# Uma classe para representar um conjunto disjunto
class DisjointSet:
    parent = {}

    # executa a operação MakeSet
    def makeSet(self, n):
        # cria conjuntos disjuntos `n` (um para cada vértice)
        for i in range(n):
            self.parent[i] = i

    # Encontre a raiz do conjunto ao qual o elemento `k` pertence
```

```
    def find(self, k):
        # se `k` for root
        if self.parent[k] == k:
            return k

        # recorrente para o pai até encontrarmos a raiz
        return self.find(self.parent[k])

    # Realiza união de dois subconjuntos
    def union(self, a, b):
        # encontra a raiz dos conjuntos aos quais os elementos `x` e `y`
pertencem
        x = self.find(a)
        y = self.find(b)

        self.parent[x] = y


# Função para construir MST usando o algoritmo de Kruskal
def runKruskalAlgorithm(edges, n):

    # armazena as arestas presentes no MST
    MST = []

    # Inicializa a classe `DisjointSet`.
    # Crie um conjunto singleton para cada elemento do universo.
    ds = DisjointSet()
    ds.makeSet(n)

    index = 0

    # classifica as arestas aumentando o peso
    edges.sort(key=lambda x: x[2])

    # MST contém exatamente bordas `V-1`
    while len(MST) != n - 1:

        # considera a próxima aresta com peso mínimo do gráfico
        (src, dest, weight) = edges[index]
        index = index + 1

        # encontre a raiz dos conjuntos para os quais dois terminais
        # vértices da próxima aresta pertencem
        x = ds.find(src)
        y = ds.find(dest)

        # se ambos os terminais tiverem pais diferentes, eles pertencem a
        # diferentes componentes conectados e podem ser incluídos no MST
        if x != y:
            MST.append((src, dest, weight))
```

```python
            ds.union(x, y)

    return MST


if __name__ == '__main__':

    # (u, v, w) tripleto representam a borda não direcionada de
    # vértice `u` para vértice `v` com peso `w`
    edges = [
        (0, 1, 7), (1, 2, 8), (0, 3, 5), (1, 3, 9), (1, 4, 7), (2, 4, 5),
        (3, 4, 15), (3, 5, 6), (4, 5, 8), (4, 6, 9), (5, 6, 11)
    ]

    # número total de nós no gráfico (rotulado de 0 a 6)
    n = 7

    # gráfico de construção
    e = runKruskalAlgorithm(edges, n)

    print(e)
```

5. Problema do Menor Caminho

O problema do caminho mais curto é encontrar o melhor caminho entre dois nós. Portanto, resolver este problema pode significar determinar um caminho entre dois nós com o menor custo ou menor tempo de viagem.

Em qualquer rede, dependendo de suas características, podem existir múltiplos caminhos entre um par de nós, definidos como origem e destino. Entre vários caminhos, aquele com o menor "peso" é chamado de caminho mais curto. O peso representa a soma dos valores dos arcos que compõem o caminho, podendo ser: tempo de viagem, distância percorrida ou qualquer custo do arco.

5.1 Algoritmo de Dijkstra

O algoritmo de Dijkstra é uma solução para o problema do caminho mínimo de origem única. Funciona em grafos orientados e não orientados, no entanto, todas as arestas devem ter custos não negativos. Se houver custos negativos, usa-se o algoritmo de Bellman-Ford

Entrada: Grafo ponderado G=(N,E) e nó origem O N, de modo que todos os custos das arestas sejam não negativos.

Saída: Comprimentos de caminhos mais curtos (ou os caminhos mais curtos em si) de um determinado nó origem O N para todos os outros nós.

5.1.1 Algoritmo

```
Djikstra(Grafo g, Vértice origem){
       distancia[origem] = 0
```

```
    Cria conjunto de vertices Não_Visitados

    Para cada vértice v do grafo g{
        Se v!= origem{
            distancia[v] = INFINITO
        }
        anterior[v] = INDEFINIDO
        Insere v em Não_Visitados
    }

    Enquanto Não_Visitados não for VAZIA{
        Procura vértice com menor distância, denominado u

        Remove u de Não_Visitados

        Para cada i vizinho de u{
            custo = distancia[u] + g[u, i]

            Se custo < distancia[i] {
                distancia[i] = custo
                anterior[i] = u
            }
        }
    }

    return distancia, anterior
}
```

O algoritmo de Dijkstra identifica, a partir do nó O, qual é o custo mínimo entre esse nó e todos os outros do grafo. No início, o conjunto S contém somente esse nó, chamado de origem. A cada passo, selecionamos o conjunto de nós sobrando, o que está mais perto da origem. Depois atualizamos, para cada nó que está sobrando, a sua distância em relação à origem. Se passando pelo novo nó acrescentado, a distância ficar menor, é essa nova distância que será memorizada. Escolhido um nó como origem da busca, este algoritmo calcula, então, o custo mínimo deste nó para todos os demais nós do grafo. O procedimento é iterativo, determinando, na iteração 1, o nó mais próximo do nó O, na segunda iteração, o segundo nó mais próximo do nó O, e assim sucessivamente, até que em alguma iteração todos os **n** nós sejam atingidos.

Algoritmo de Dijkstra	Caracteristicas
	-Possui um ponto de partida; -Pode ser orientado ou não; -O Custo das arestas não podem ser negativos; -Grafo Conexo

5.1.2 Exemplo

```
import sys

class Graph():

    def __init__(self, vertx):
        self.V = vertx
        self.graph = [[0 for column in range(vertx)]
                      for row in range(vertx)]

    def pSol(self, dist):
        print("Distance of vertex from source")
        for node in range(self.V):
            print(node, "t", dist[node])


    def minDistance(self, dist, sptSet):


        min = sys.maxsize


        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]
                min_index = v

        return min_index


    def dijk(self, source):

        dist = [sys.maxsize] * self.V
        dist[source] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):

            u = self.minDistance(dist, sptSet)

            sptSet[u] = True


            for v in range(self.V):
                  if self.graph[u][v] > 0 and sptSet[v] == False and dist[v] >
dist[u] + self.graph[u][v]:
                    dist[v] = dist[u] + self.graph[u][v]
        self.pSol(dist)

f=Graph(9)
f.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
           [4, 0, 8, 0, 0, 0, 0, 11, 0],
           [0, 8, 0, 7, 0, 4, 0, 0, 2],
           [0, 0, 7, 0, 9, 14, 0, 0, 0],
```

```
          [0, 0, 0, 9, 0, 10, 0, 0, 0],
          [0, 0, 4, 14, 10, 0, 2, 0, 0],
          [0, 0, 0, 0, 0, 2, 0, 1, 6],
          [8, 11, 0, 0, 0, 0, 1, 0, 7],
          [0, 0, 2, 0, 0, 0, 6, 7, 0]
          ]

f.dijk(0)
```

5.2 Algoritmo de Floyd-Warshall

O algoritmo Floyd-Warshall calcula os caminhos mais curtos entre todos os pares de vértices de um grafo ponderado direcionado que possui arcos com pesos negativos, mas sem ciclos de custo negativo. É também um algoritmo de programação dinâmica de baixo para cima. Novamente, o conceito de relaxamento do comprimento do caminho mais curto é usado: gradualmente, as estimativas usadas são melhoradas até que um valor ótimo seja alcançado.

5.2.2 Algoritmo

```
ROTINA fw(Inteiro[1..n,1..n] grafo)
# Inicialização
VAR Inteiro[1..n,1..n]
dist := grafo
VAR Inteiro[1..n,1..n]
pred
PARA i DE 1 A n
    PARA j DE 1 A n
        SE dist[i,j] < Infinito ENTÃO
            pred[i,j] := i
            # Laço principal do algoritmo
            PARA k DE 1 A n
                PARA i DE 1 A n
                    PARA j DE 1 A n
                        SE dist[i,j] > dist[i,k] + dist[k,j] ENTÃO
                            dist[i,j] = dist[i,k] + dist[k,j]
                            pred[i,j] = pred[k,j]
                            RETORNE dist
```

5.2.2 Exemplo

```
# Função recursivo para imprimir o caminho de um determinado vértice `u` do
vértice de origem `v`
def printPath(path, v, u, route):
    if path[v][u] == v:
        return
    printPath(path, v, path[v][u], route)
    route.append(path[v][u])


# Função para imprimir o menor custo com caminho
# Informação # entre todos os pares de vértices
def printSolution(path, n):
```

```
    for v in range(n):
        for u in range(n):
            if u != v and path[v][u] != -1:
                route = [v]
                printPath(path, v, u, route)
                route.append(u)
                print(f'The shortest path from {v} —> {u} is', route)


# Função para executar o algoritmo Floyd-Warshall
def floydWarshall(adjMatrix):

    # caso básico
    if not adjMatrix:
        return

    # número total de vértices no `adjMatrix`
    n = len(adjMatrix)

    # A matriz de custo e caminho # armazena o caminho mais curto
    # (custo mais curto/rota mais curta)

    # inicialmente, o custo seria o mesmo que o peso de uma aresta
    cost = adjMatrix.copy()
    path = [[None for x in range(n)] for y in range(n)]

    # inicializa custo e caminho
    for v in range(n):
        for u in range(n):
            if v == u:
                path[v][u] = 0
            elif cost[v][u] != float('inf'):
                path[v][u] = v
            else:
                path[v][u] = -1

    # roda Floyd-Warshall
    for k in range(n):
        for v in range(n):
            for u in range(n):
                 # Se o vértice `k` estiver no caminho mais curto de `v` para
`u`,
                #, em seguida, atualize o valor de cost[v][u] e path[v][u]
                if cost[v][k] != float('inf') and cost[k][u] != float('inf') \
                        and (cost[v][k] + cost[k][u] < cost[v][u]):
                    cost[v][u] = cost[v][k] + cost[k][u]
                    path[v][u] = path[k][u]

            # se os elementos diagonais se tornarem negativos, o
            # O gráfico # contém um ciclo de peso negativo
            if cost[v][v] < 0:
                print('Negative-weight cycle found')
                return

    # Imprime o caminho mais curto entre todos os pares de vértices
```

```
    printSolution(path, n)


if __name__ == '__main__':

    # define o infinito
    I = float('inf')

    # dada a representação de adjacência da matriz
    adjMatrix = [
        [0, I, -2, I],
        [4, 0, 3, I],
        [I, I, 0, 2],
        [I, -1, I, 0]
    ]

    # Executar algoritmo Floyd-Warshall
    floydWarshall(adjMatrix)
```

A complexidade de tempo do algoritmo Floyd-Warshall é O(V3), Onde V é o número total de vértices no gráfico.

5.3 Algoritmo A*

O algoritmo A* é um dos mais utilizados em situações de pathfinding, ou busca de caminhos. Ele otimiza o algoritmo de Dijkstra em dois aspectos para tornar o seu funcionamento mais eficiente:

- Ele utiliza uma estrutura de dados chamada Fila de prioridade para organizar os vértices que serão explorados;
- Além de salvar os caminhos já calculados (como Dijkstra faz), ele também utiliza heurísticas para estimar em cada ponto quanto ainda falta para o final, buscando direcionar a escolha do próximo vértice.

5.3.1 Algoritmo

```
   Busca_A*(Grafo g, Vértice inicio, Vértice fim){
        Visitados = {} //Nenhum vértice foi visitado ainda
        Abertos ={inicio} //Vértices na fila para exploração

        custo[] // Vetor que representa os custos dos caminhos para cada

        //vértice
        anterior[] // Vetor que representa o vértice anterior no caminho
        futuro[] //Vetor que representa o custo futuro do caminho

        Enquanto Abertos não for vazio{
            atual = vértice de Abertos que possui menor valor de caminho futuro

            Se atual == fim
                Retorne o caminho

            Remove atual de Abertos
            Insere atual em Visitados
```

```
            Para cada vizinho de atual{
                Se o vizinho não foi visitado{
                    Se ele não estiver em Abertos
                        Insere vizinho em Abertos

                    custo_vizinho = custo[atual] + g[atual, vizinho]

                    Se custo_vizinho < custo[vizinho]{
                        anterior[vizinho] = atual
                        custo[vizinho] = custo_vizinho
                        futuro[vizinho] = custo[vizinho]+ HEURÍSTICA
                    }
                }
            }
        }
        Retorna ERRO
    }
```

5.3.2 Exemplo

```
#Matriz visual usada pra montar a matriz de adjacência
matriz = [ [0, 1, 2],
           [3, 4, 5],
           [6, 7, 8], ]
'''
  0  |  1     2
            ---
  3     4  |  5
 ---
  6     7     8
Ir do 0 ao 5
sem bater nas paredes
'''

nos = [
    [3],#nó 0
    [4, 2],#nó 1
    [1],#nó 2
    [4, 0],#nó 3
    [7, 1, 3],#nó 4
    [8],#nó 5
    [7],#nó 6
    [6, 4, 8],#nó 7
    [5, 7] #nó 8
]

inicio = 0
final = 5


#Função para testar qual a distância espacial do nó n para o nó final
```

```
def retorna_h(destino, node):
    def h(n):
        def caminhar(pos, visitados, distancia):
            caminhos = node[pos]
            distancias = []
            visitados.add(pos)
            if pos == destino:
                return distancia
            distancia += 1
            for novo_pos in caminhos:
                if novo_pos == destino:
                    return distancia
                if not novo_pos in visitados:
                    try:
                        distancias.append(
caminhar(novo_pos,set(visitados), distancia) )
                    except:
                        pass
            return min(distancias)


        try:
            return caminhar(n,set(), 0)
        except:
            return None

    return h

meu_h = retorna_h(final, nos)

print( meu_h(0) )

def retorna_caminho(destino, node, meu_h):
    def meu_f(n):
        def calculo_f(pos, g_anterior):
            valor = meu_h(pos)
            if not valor:
                raise Exception('Não há caminho para determinado
destino')
            return g_anterior + valor
        pos = n
        g = 0
        path = [pos]
        try:
            while True:
                menor = 999999
                idx_menor = -1
                caminhos = node[pos]
                for caminho in caminhos:
                    if caminho == destino:
```

```
                        path.append(caminho)
                        return path
                    if caminho in path:
                        continue
                    heuristica = calculo_f(caminho, g+1)
                    if heuristica < menor:
                        menor = heuristica
                        idx_menor = caminho
                if idx_menor != -1:
                    pos = idx_menor
                    path.append(idx_menor)
                    g += 1
                    continue
                else:
                    break
        except:
            return []
        return path
    return meu_f

calcular_caminho = retorna_caminho(final, nos, meu_h)


texto = ''
caminhos = calcular_caminho(0)
for i in range( len(caminhos) ):
    passagem = caminhos[i]
    if i != 0:
        texto += ' => '
    texto += str(passagem)

print(texto)
```

6. Referências

—Busca em Profundidade—

https://algoritmosempython.com.br/cursos/algoritmos-python/algoritmos-grafos/busca-profundidade/

https://blog.pantuza.com/artigos/busca-em-profundidade

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/dfs.html

http://www.dsc.ufcg.edu.br/~pet/jornal/junho2012/materias/recapitulando.html#:~:text=busca%20em%20profundidade.-,Pseudoc%C3%B3digo%20do%20Algoritmo%20de%20Busca%20em%20Profundidade.,quebra%2Dcabe%C3%A7as%20como%20o%20labirinto.

—Algoritmo de Prim—

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/prim.html https://neps.academy/br/course/algoritmos-em-grafos-(codcad)/lesson/algoritmo-de-prim

https://files.cercomp.ufg.br/weby/up/666/o/algoritmo_prim.pdf?1389784036

https://favtutor.com/blogs/prims-algorithm-python

https://algotree.org/algorithms/minimum_spanning_tree/prims_python/

—Algoritmo de Kruskal—

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/kruskal.html

http://www.aloc.ufscar.br/felice/ensino/2020s2paa/aula15.pdf

https://acervolima.com/algoritmo-de-kruskal-implementacao-simples-para-array-de-adjacencia/

—Problema do Menor Caminho—

https://docs.ufpr.br/~volmir/PO_II_10_caminho_minimo.pdf

—Algoritmo de Dijkstra—

https://www.facom.ufu.br/~madriana/ED2/6-AlgDijkstra.pdf

https://materialpublic.imd.ufrn.br/curso/disciplina/5/69/10/3#:~:text=O%20Dijkstra%20%C3%A9%20um%20dos,caminho%20entre%20dois%20v%C3%A9rtices%20espec%C3%ADficos.

https://www.delftstack.com/pt/howto/python/dijkstra-algorithm-python/#:~:text=O%20algoritmo%20de%20Dijkstra%20pode,partir%20do%20v%C3%A9rtice%20de%20origem.

—Algoritmo de Floyd-Warshall—

http://www.decom.ufop.br/marco/site_media/uploads/bcc204/07_aula_07.pdf

https://www.techiedelight.com/pt/pairs-shortest-paths-floyd-warshall-algorithm/

http://dicionario.sensagent.com/Algoritmo%20de%20Floyd-Warshall/pt-pt/

https://pt.slideshare.net/lucasvinicius585/complexidade-do-algoritmo-caminho-mnimo-floyd-warshall

—Algoritmo A*—

https://materialpublic.imd.ufrn.br/curso/disciplina/5/69/10/4

https://gist.github.com/nenodias/d92b4cdbfb92ace257ff535856ba0a46