

**UNIVERSIDADE FEDERAL DE RORAIMA**

**CENTRO DE CIÊNCIA E TECNOLOGIA**

**DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO**

**ESTRUTURA DE DADOS II - 2022**

**PROF. ACAUAN C.RIBEIRO**

# Apostila Resumida - Teoria dos Grafos

**Fonte:** <http://faculty.ycp.edu/~dbabcock/PastCourses/cs360/lectures/>

**Adaptação:** **Prof. Acauan C. Ribeiro**

**Alunos: VALERIA ALEXANDRA GUEVARA PARRA**

**Matricula: 209047300**

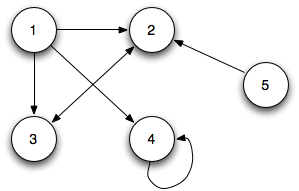
Este trabalho se propõre a fazer uma breve revisão sobre os principais assuntos relacionados a Grafo vistos na disciplina DCC 405 – Estrutura de Dados II.

# 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.

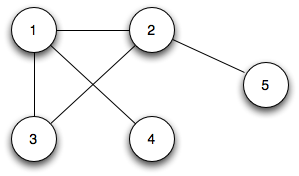
**Dizemos que u e v são vizinhos se (u,v) ∈ E ou (v,u) ∈ E.**

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 , ...,* v *k >* com **u** = v0 , u' = vk , e ( vi-1 , vi ) ∈ E .

Se existe um caminho p de u para u', então u' é alcançável a partir de u (denotado u ↝ 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 , ..., v j > com 0 ≤ i ≤ j ≤ 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 ) ∈ E ou

u ∈ V 1 ev ∈ V 2 OU u ∈ V 2 ev ∈ 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

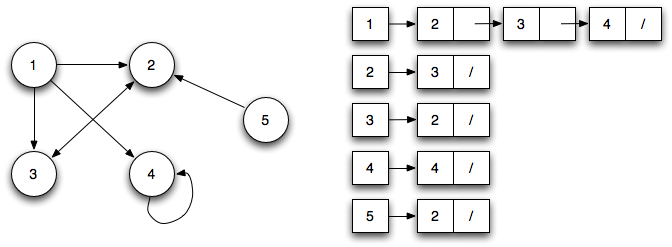
Em uma implementação de lista de adjacência, nós simplesmente armazenamos os vértices adjacentes (ou seja, arestas) para cada vértice em uma lista encadeada denotada por Adj [ u ]. Se somarmos os comprimentos de todas as listas de adjacência, obtemos



⇒ O armazenamento Θ( V + E ) é necessário.

Esta representação é boa para gráficos esparsos onde | E | ≪ | V | 2 . Uma desvantagem é que determinar se uma aresta ( u , v ) ∈ E requer uma busca de lista ⇒ Θ( V ).

Para o grafo direcionado original, a lista de adjacências seria



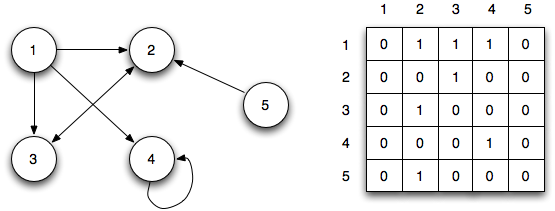
2.2 Matriz de adjacência

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 Θ( V2 ) é necessário (independente de E).

Esta representação é boa para gráficos densos onde | E | ≈ | V | 2 . A vantagem é que leva apenas Θ(1) tempo para determinar se uma aresta ( u , v ) ∈ E , pois é simplesmente um acesso a um elemento da matriz. Se o gráfico não é direcionado, então **A = AT** , portanto, apenas a metade triangular superior precisa ser armazenada.

Para o grafo direcionado original, a matriz de adjacência seria



# 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)**

1. para cada vértice u ∈ GV - {s}

2. u.cor == BRANCO

3. ud = INF

4. u.pi = NIL

5. s.cor = CINZA

6. dp = 0

7. s.pi = NIL

8. Q = ∅

9. ENFILEIRA(Q,s)

10. enquanto Q ≠ ∅

11. u = DEQUEUE(Q)

12. para cada v ∈ G.Adj[u]

13. se v.cor == BRANCO

14. v.cor = CINZA

15. vd = ud + 1

16. 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 )

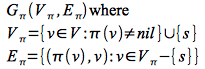
Cada lista de adjacências é escaneada apenas uma vez (quando o vértice é desenfileirado) com tamanho máximo o número total de arestas ⇒ O( E )

Sobrecarga de inicialização ⇒ O( V )

Assim, o tempo total de execução para BFS é



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



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

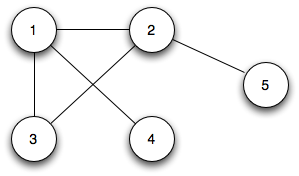
Além disso, pode-se mostrar que como o subgrafo predecessor é uma árvore , pelo teorema B.2 do CLRS



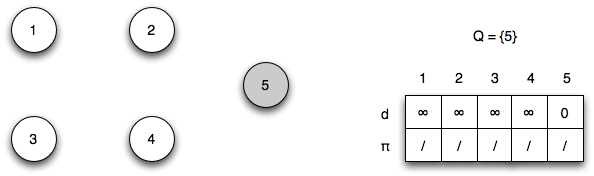
A árvore predecessora pode ser percorrida (usando os π's) para fornecer o caminho mais curto de s para v .

3.1.5 Exemplo

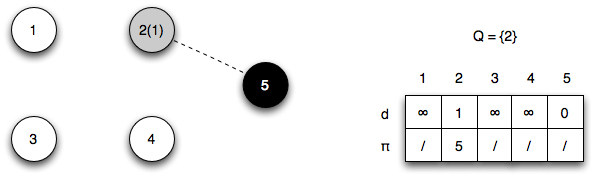
Considere o gráfico de cinco nós (não direcionado)



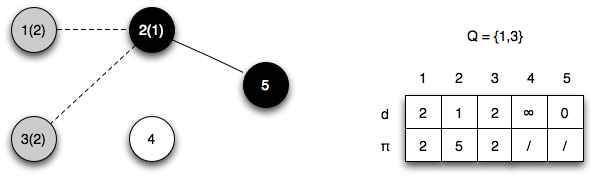
Se selecionarmos o vértice 5 como fonte, então d[5]=0, π[5]=/, Q={5}, então a inicialização nos dá



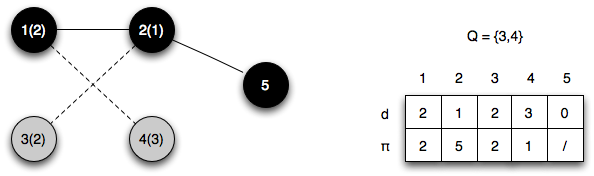
Iteração 1 : desenfileirar o vértice 5 e enfileirar o vértice 2



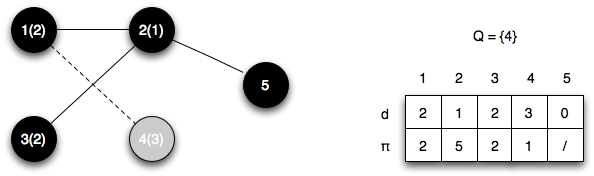
Iteração 2 : desenfileirar vértice 2 e enfileirar vértices 1,3



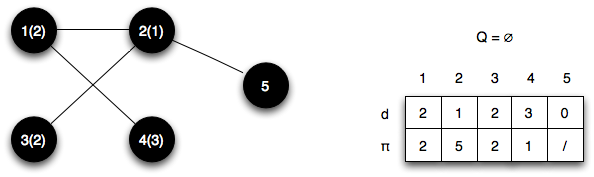
Iteração 3 : desenfileirar o vértice 1 e enfileirar o vértice 4



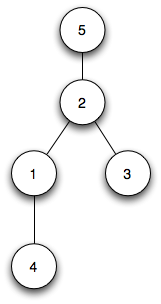
Iteração 4 : desenfileirar o vértice 3 e enfileirar nenhum vértice



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



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

 é um algoritmo usado para realizar uma busca ou travessia numa árvore, estrutura de árvore ou grafo. Intuitivamente, o algoritmo começa num nó raiz e explora tanto quanto possível cada um dos seus ramos, antes de retroceder

3.2.1 Problema

Dado um vértice de origem **s**, encontre os possíveis caminhos sem deixar de visitar nenhuma das arestas garantindo desta forma ainda que não seja na mesma ordem que nenhuma das arestas foi deixada para trás, ele pode voltar caso a aresta não tenha saída e ainda fiquem outras arestas por visitar, mas o importante a ressaltar é que o programa não pode finalizar até recorrer todo o algoritmo.

3.2.2 Solução

Uma implementação DFS padrão coloca cada vértice do gráfico em uma das duas categorias:

1. Visitou
2. Não visitado

O objetivo do algoritmo é marcar cada vértice como visitado, evitando ciclos.

O algoritmo DFS funciona da seguinte forma:

1. Comece colocando qualquer um dos vértices do gráfico no topo de uma pilha.
2. Pegue o item do topo da pilha e adicione-o à lista de visitados.
3. Crie uma lista dos nós adjacentes desse vértice. Adicione os que não estão na lista visitada ao topo da pilha.
4. Continue repetindo as etapas 2 e 3 até que a pilha esteja vazia

3.2.3 Algoritmo

DFS(G, u)

u.visitou = true

para cada v ∈ G.Adj[u]

if v.visitou == false

DFS(G,v)

iniciar() {

Para cada u ∈ G

u.visitou = false

Para cada u ∈ G

DFS(G, u)

}

3.2.4 Análise

A complexidade de tempo do algoritmo DFS é representada na forma de O(V + E), onde Vé o número de nós e Eé o número de arestas.

A complexidade de espaço do algoritmo é O(V).

1. Por encontrar o caminho
2. Para testar se o gráfico é bipartido
3. Para encontrar os componentes fortemente conectados de um grafo
4. Para detectar ciclos em um gráfico
5. Desta forma garantindo que não é parada a procura até que todos os vértices são visitados.

3.2.5 Exemplo

# DFS algorithm in Python

# DFS algorithm

def dfs(graph, start, visited=None):

if visited is None:

visited = set()

visited.add(start)

print(start)

for next in graph[start] - visited:

dfs(graph, next, visited)

return visited

graph = {'0': set(['1', '2']),

'1': set(['0', '3', '4']),

'2': set(['0']),

'3': set(['1']),

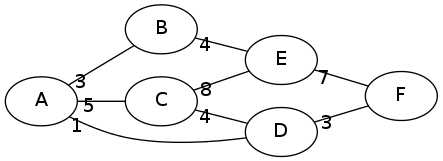
'4': set(['2', '3'])}

dfs(graph, '0')

4. Árvore geradora mínima

Dado um grafo não orientado conectado, uma árvore de extensão deste grafo é um subgrafo o qual é

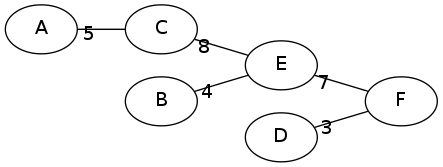
uma árvore que conecta todos os vértices. Um único grafo pode ter diferentes árvores de extensão



**Uma árvore geradora é simplesmente um conjunto de arestas do grafo que gera uma árvore.**

**Toda árvore é um grafo conexo acíclico**. Mas é mais fácil imaginar o mesmo grafo com pelo menos uma aresta entrando e no máximo uma (ou seja, não necessariamente há uma) aresta saindo de cada vértice.

Por exemplo, uma árvore geradora no grafo acima pode ser esta:



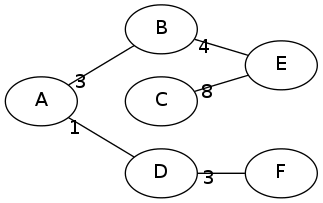
Os dois algoritmos clássicos em grafos, a **busca em profundidade** e a **busca em largura**, induzem árvores geradoras no grafo. Por isso você já as conhecia!

O **peso** dessa árvore é a soma dos valores associados as arestas dela, nesse caso:

5 + 8 + 4 + 7 + 3 = 27

Uma árvore geradora é chamada **mínima** se, dentre todas as árvores geradoras que existem no grafo, a soma dos pesos nas arestas dela é o menor possível.

Nesse grafo, uma MST (do inglês, *minimum spanning tree*) pode ser:



O peso dessa é:

1 + 3 + 3 + 4 + 8 == 19

# **4.1 Algoritmo de Prim**

**algoritmo de Prim** é um [algoritmo guloso](https://pt.wikipedia.org/wiki/Algoritmo_guloso) (*greedy algorithm*) empregado para encontrar uma [árvore geradora mínima](https://pt.wikipedia.org/wiki/%C3%81rvore_geradora_m%C3%ADnima) (*minimal spanning tree*) num grafo conectado, valorado e não direcionado. Isso significa que o algoritmo encontra um subgrafo do grafo original no qual a soma total das arestas é minimizada e todos os vértices estão interligados. O algoritmo foi desenvolvido em 1930 pelo matemático [Vojtěch Jarník](https://pt.wikipedia.org/wiki/Vojt%C4%9Bch_Jarn%C3%ADk" \o "Vojtěch Jarník) e depois pelo cientista da computação [Robert Clay Prim](https://pt.wikipedia.org/wiki/Robert_Clay_Prim) em 1957 e redescoberto por [Edsger Dijkstra](https://pt.wikipedia.org/wiki/Edsger_Dijkstra" \o "Edsger Dijkstra) em 1959.

Outros algoritmos conhecidos para encontrar [árvores geradoras mínimas](https://pt.wikipedia.org/wiki/%C3%81rvore_geradora_m%C3%ADnima) são o [algoritmo de Kruskal](https://pt.wikipedia.org/wiki/Algoritmo_de_Kruskal) e [algoritmo de Boruvka](https://pt.wikipedia.org/wiki/Algoritmo_de_Boruvka), sendo que este último pode ser empregado em grafos desconexos, enquanto o **algoritmo de Prim** e o [Algoritmo de Kruskal](https://pt.wikipedia.org/wiki/Algoritmo_de_Kruskal) precisam de um grafo conexo.

Algoritmo

Um algoritmo genérico para o **algoritmo de Prim** é dado da seguinte forma:

*Escolha um vértice****S****para iniciar o subgrafo*

***enquanto****houver vértices que não estão no subgrafo*

*selecione uma aresta segura*

*insira a aresta segura e seu vértice no subgrafo*

**Pseudocódigo**

prim(G) # G é grafo

# Escolhe qualquer vértice do grafo como vértice inicial/de partida

s ← seleciona-um-elemento(vertices(G))

para todo v ∈ vertices(G)

π[v] ← nulo

Q ← {(0, s)}

S ← ø

enquanto Q ≠ ø

v ← extrair-mín(Q)

S ← S ∪ {v}

para cada u adjacente a v

se u ∉ S e pesoDaAresta(π[u]→u) > pesoDaAresta(v→u)

Q ← Q \ {(pesoDaAresta(π[u]→u), u)}

Q ← Q ∪ {(pesoDaAresta(v→u), u)}

Q <- Q u {pesoDaArest(v->)%2, Q++}

π[u] ← v

print(Pronto)

retorna {(π[v], v) | v ∈ vertices(G) e π[v] ≠ nulo}

Exemplos

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Imagem** | **Arestas incluídas no subgrafo** | **U** | **Arestas possíveis** | **V \ U** | **Descrição** |
|  | {} | {} |  | {A,B,C,D,E,F,G} | Este é o grafo original. Os números próximos das arestas significam o seu peso. |
|  | {DA} | {D} | {D,A} = 5**V** {D,B}=9 {D,E}=15 {D,F}=6 | {A,B,C,E,F,G} | O vértice **D** foi escolhido como ponto inicial do algoritmo. Vértices **A**, **B**, **E** e **F** estão conectados com **D** através de uma única aresta. **A** é o vértice mais próximo de **D** e, portanto a aresta **AD** será escolhida para formar o subgrafo. |
|  | {DA, DF} | {A,D} | {D,B}=9 {D,E}=15 {D,F}=6**V** {A,B}=7 | {B,C,E,F,G} | O próximo vértice escolhido é o mais próximo de **D** ou **A**. **B** está a uma distância 9 de **D**, **E** numa distância 15 e **F** numa distância 6. E **A** está a uma distância de 7 de **B**. Logo devemos escolher a aresta **DF**, pois é o menor peso. |
|  | {DA, DF, AB} | {A,D,F} | {D,B}=9 {D,E}=15 {A,B}=7**V** {F,E}=8 {F,G}=11 | {B,C,E,G} | Agora devemos escolher o vértice mais próximo dos vértices **A**, **D** ou **F**. A aresta em questão é a aresta **AB**. |
|  | {DA, DF, AB, BE} | {A,B,D,F} | {B,C}= 8 {B,E}=7**V** {D,B}=9**ciclo** {D,E}=15 {F,E}=8 {F,G}=11 | {C,E,G} | Agora podemos escolher entre os vértices **C**, **E**, e **G**. **C** está a uma distância de 8 de **B**, **E** está a uma distância 7 de **B** e **G** está a 11 de **F**. **E** é o mais próximo do subgrafo e, portanto escolhemos a aresta **BE**. |
|  | {DA, DF, AB, BE, EC} | {A,B,D,E,F} | {B,C}=8 {D,B}=9ciclo {D,E}=15ciclo {E,C}=5**V** {E,G}=9 {F,E}=8 ciclo {F,G}=11 | {C,G} | Restam somente os vértices **C** e **G**. **C** está a uma distância 5 de **E** e de **G** a **E** 9. **C** é escolhido, então a aresta **EC** entra no subgrafo construído. |
|  | {DA, DF, AB, BE, EC, EG} | {A,B,C,D,E,F} | {B,C}=8ciclo {D,B}=9ciclo {D,E}=15ciclo {E,G}=9**V** {F,E}=8ciclo {F,G}=11 | {G} | Agora só resta o vértice **G**. Ele está a uma distância de 11 de **F**, e 9 de **E**. **E** é o mais próximo, então **G** entra no subgrafo conectado pela aresta **EG**. |
|  | {DA, DF, AB, BE, EC, EG} | {A,B,C,D,E,F,G} | {B,C}=8 ciclo {D,B}=9 ciclo {D,E}=15 ciclo {F,E}=8 ciclo {F,G}=11 ciclo | {} | Aqui está o fim do algoritmo e o subgrafo formado pelas arestas em verde representam a árvore geradora mínima. Nesse caso esta árvore apresenta a soma de todas as suas arestas o número 39. |

# **4.2 Algoritmo de Kruskal**

Definição teórica

é um algoritmo em teoria dos grafos que busca uma árvore geradora mínima para um grafo conexo com pesos. Isto significa que ele encontra um subconjunto das arestas que forma uma árvore que inclui todos os vértices, onde o peso total, dado pela soma dos pesos das arestas da árvore, é minimizado.

Algoritmo

Uma *sub­floresta* de um grafo não-dirigido *G* é qualquer [floresta](https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/circuits-and-forests.html#sec:forest) que seja [sub­grafo não-dirigido](https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/graphs.html" \l "subgraph-undirected) de *G*.  Uma *floresta geradora* de *G* é qualquer sub­floresta que tenha o mesmo conjunto de vértices que *G*.

Uma aresta *a* de *G* é *externa* a uma floresta geradora *F* se *a* não pertence a *F* e o grafo  *F* + *a*  é uma floresta, ou seja, um grafo sem [circuitos](https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/circuits-and-forests.html#sec:circuit).  Portanto, uma aresta é externa a *F* se tem uma ponta em uma [componente conexa](https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/components.html) de *F* e outra ponta em outra componente.

Podemos agora tratar do algoritmo.  [!] Cada iteração do algoritmo de Kruskal começa com uma floresta geradora  *F*  de *G*.  O processo iterativo é muito simples: enquanto existe alguma aresta externa,

1. escolha uma aresta externa que tenha custo mínimo;
2. seja *a* a aresta escolhida;
3. acrescente *a* a *F*.

No início da primeira iteração, cada componente conexa da floresta *F* tem apenas um vértice.  No fim do processo iterativo, *F* é conexa, uma vez que *G* é conexo e não há arestas externas a *F*.

Como se vê, o algoritmo tem caráter [*guloso*](https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/remissivo.html#greedy): em cada iteração, abocanha a aresta que parece mais promissora localmente sem se preocupar com o efeito global dessa escolha.  O algoritmo foi projetado pensando no [critério de minimalidade de MSTs baseado em circuitos](https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/mst.html#optimality1).

Pseudocodigo

procedimento Kruskal(var Grafo: TGrafo;

var T: conjunto de arestas)

variáveis

u, v:

TVertice; U1,...,

Un: conjunto de TVertice;

Q: fila de prioridade;

início

T := ∅; Q := as arestas de G ordenadas pelo seu peso;

para i:=1 até Grafo.NumVertices faça Ui := {i};

enquanto houver arestas em Q faça

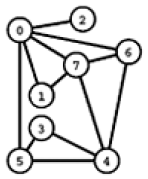
início seja (u, v) a aresta de menor peso de Q tal que (u ∈ Up) e (v ∈ Uq) e (Up ∩ Uq)= ∅ T := T ∪ {(u, v)}; Up := Up ∪ Uq; eliminar Uq;

fim

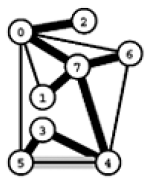
fim

Exemplos

**Exemplo A.**  O vídeo *[Kruskal's Algorithm Animation](https://www.youtube.com/watch?time_continue=16&v=o8Sqm1_3BRo)* no YouTube aplica o algoritmo de Kruskal a um grafo cujos vértices são pontos aleatórios no plano. O grafo é [completo](https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/remissivo.html#nao-dirigido-completo) e o custo de cada aresta é a distância geométrica entre suas pontas.

[](https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/figs/Sedgewick-part5-java/kruskal-fig-20-13-a.png)

**Exemplo B.**  Considere o problema de encontrar uma MST no grafo não-dirigido com custos nas arestas definido a seguir.  (Para simplificar o rastreamento, as arestas são apresentadas em ordem crescente de custo.)  Em seguida, veja o rastreamento da execução do algoritmo de Kruskal. Cada linha da tabela registra, no início de cada iteração, as arestas da floresta e o custo da floresta.

3-5 1-7 6-7 0-2 0-7 0-1 3-4 4-5 4-7 0-6 4-6 0-5 [](https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/figs/Sedgewick-part5-java/kruskal-fig-20-13-h.png)

0 1 2 3 4 5 6 7 8 9 9 11

floresta custo

- 0

3-5 0+0

3-5 1-7 0+1

3-5 1-7 6-7 1+2

3-5 1-7 6-7 0-2 3+3

3-5 1-7 6-7 0-2 0-7 6+4

3-5 1-7 6-7 0-2 0-7 3-4 10+6

3-5 1-7 6-7 0-2 0-7 3-4 4-7 16+8

A última linha dá as arestas de uma MST. Resumindo, o algoritmo de Kruskal escolhe as seguintes arestas para formar uma MST:

3-5 1-7 6-7 0-2 0-7 3-4 4-7

0 1 2 3 4 6 8

(Os espaços em branco correspondem às arestas que foram rejeitadas porque formam um circuito com as arestas escolhidas em iterações anteriores.)

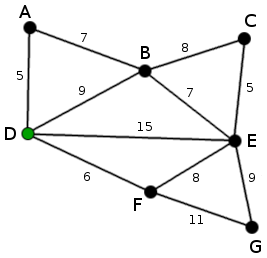
5. Problema do Menor Caminho

consiste na minimização do *custo* de travessia de um [grafo](https://pt.wikipedia.org/wiki/Grafo) entre dois nós (ou vértices); custo este dado pela soma dos pesos de cada [aresta](https://pt.wikipedia.org/wiki/Aresta) percorrida.

 problema de caminho mínimo é um dos problemas genéricos intensamente estudados e utilizados em diversas áreas como Engenharia de Transportes, Pesquisa Operacional, Ciência da Computação e Inteligência Artificial. Isso decorre do seu potencial de aplicação a inúmeros problemas práticos que ocorrem em transportes, logística, redes de computadores e de telecomunicações, entre outros.

Podem ser aplicados para encontrar automaticamente direções entre locais físicos, como instruções de direção em sites de mapeamento da web como o MapQuest ou o [Google Maps](https://pt.wikipedia.org/wiki/Google_Maps). Para esta aplicação, algoritmos especializados rápidos estão disponíveis.

Outras possibilidades de aplicação incluem quaisquer problemas envolvendo redes ou grafos em que se tenha grandezas (distâncias, tempo, perdas, ganhos, despesas) que se acumulem linearmente ao longo do percurso da rede. Se alguém representa uma [máquina abstrata](https://pt.wikipedia.org/wiki/M%C3%A1quina_abstrata) não determinística como um gráfico em que os vértices descrevem estados e arestas descrevem possíveis transições, algoritmos de caminho mínimo podem ser usados ​​para encontrar uma sequência ótima de opções para atingir um determinado estado objetivo ou para estabelecer limites mais baixos no tempo necessário para atingir um determinado estado. Por exemplo, se os vértices representam os estados de um quebra-cabeça como o [Cubo de Rubik](https://pt.wikipedia.org/wiki/Cubo_de_Rubik) e cada borda direcionada corresponde a um único movimento ou turno, os algoritmos de caminho mínimo podem ser usados ​​para encontrar uma solução que use o número mínimo possível de movimentos. Um problema relacionado é o [Problema do Caixeiro-viajante](https://pt.wikipedia.org/wiki/Problema_do_Caixeiro-viajante), que consiste em determinar o caminho mais curto que passa exatamente uma vez por cada vértice e retorna ao vértice de partida. Este é um problema [NP-Completo](https://pt.wikipedia.org/wiki/NP-Completo), para os quais não há uma solução eficiente.

[](https://pt.wikipedia.org/wiki/Ficheiro:Prim_Algorithm_0.png)

O caminho mínimo entre *D* e *E* não é D-E, mas sim D-F-E, com uma distância de 14.

5.1 Algoritmo de Dijkstra

 concebido pelo [cientista da computação](https://pt.wikipedia.org/wiki/Ci%C3%AAncia_da_computa%C3%A7%C3%A3o) holandês [Edsger Dijkstra](https://pt.wikipedia.org/wiki/Edsger_Dijkstra" \o "Edsger Dijkstra) em 1956 e publicado em 1959,[[1]](https://pt.wikipedia.org/wiki/Algoritmo_de_Dijkstra#cite_note-Dijkstra_Interview-1)[[2]](https://pt.wikipedia.org/wiki/Algoritmo_de_Dijkstra#cite_note-2) soluciona o [problema do caminho mais curto](https://pt.wikipedia.org/wiki/Problema_do_caminho_mais_curto) num [grafo dirigido](https://pt.wikipedia.org/wiki/Grafo_dirigido) ou não dirigido com arestas de peso não negativo, em tempo computacional {\displaystyle O(E+V\log(V))} onde V é o número de vértices e E é o número de arestas. O algoritmo que serve para resolver o mesmo problema em um grafo com pesos negativos é o [algoritmo de Bellman-Ford](https://pt.wikipedia.org/wiki/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.

Pseucodigo

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

}

Algoritmo

**1º passo:** iniciam-se os valores:

para todo v ∈ V[G]

d[v] ← ∞

π[v] ← -1

d[s] ← 0

V[G] é o conjunto de vértices(v) que formam o Grafo G. d[v] é o vetor de distâncias de s até cada v. Admitindo-se a pior estimativa possível, o caminho infinito. π[v] identifica o vértice de onde se origina uma conexão até v de maneira a formar um caminho mínimo.

* **2º passo:** temos que usar o conjunto *Q*, cujos vértices ainda não contém o custo do menor caminho d[v] determinado.

Q ← V[G]

* **3º passo:** realizamos uma série de **relaxamentos** das arestas, de acordo com o código:

enquanto Q ≠ ø

u ← extrair-mín(Q) //Q ← Q - {u}

para cada v adjacente a u

se d[v] > d[u] + peso(u, v) //relaxe (u, v)

então d[v] ← d[u] + peso(u, v)

π[v] ← u

peso(u, v) é o peso da aresta que vai de u a v.

u e v são vértices quaisquer e s é o vértice inicial. ffd extrair-mín(Q), pode usar um [heap](https://pt.wikipedia.org/wiki/Heap" \o "Heap) de mínimo ou uma lista de vértices onde se extrai o elemento u com menor valor d[u].

5.2 Algoritmo de Floyd-Warshall

é um algoritmo que resolve o problema de calcular o caminho mais curto entre todos os pares de vértices em um grafo orientado e valorado. O algoritmo Floyd-Warshall foi publicado por Robert Floyd em 1966. Trata-se novamente de um algoritmo de programação dinâmica bottom-up. Também de maneira similar, o conceito de relaxação do comprimento dos caminhos mais curtos é empregado: incrementalmente, aprimora-se uma estimativa utilizada, até que o valor ótimo seja atingido.

**Algoritmo**

Nós já cobrimos **fonte única os caminhos mais curtos** em postos separados. Vimos que:

* Para gráficos com pesos de aresta não negativos, [Algoritmo de Dijkstra](https://www.techiedelight.com/pt/single-source-shortest-paths-dijkstras-algorithm/) corre em O(E + V × log(V))
* Para gráficos contendo pesos de arestas negativos, [Bellman–Ford](https://www.techiedelight.com/pt/single-source-shortest-paths-bellman-ford-algorithm/) corre em O(V × E).
* Para um DAG, [um passe de Bellman-Ford](https://www.techiedelight.com/pt/cost-of-shortest-path-in-dag-using-one-pass-of-bellman-ford/) *(chamado passo de relaxamento)* é o suficiente que vai levar O(V + E) Tempo.

Aqui, V é o número total de vértices e E é o número total de arestas no grafo.

Este post vai apresentar **Caminhos mais curtos de todos os pares** que retornam os caminhos mais curtos entre cada par de vértices no gráfico contendo pesos de arestas negativos.

**Pseucodigo**

Entrada: Grafo G = (V , E) e matriz de pesos D={dij} para os arcos {i, j} 1

L ← D;//Inicializa os elementos da matriz

L 2 para k ← 1 até n faça 3

para i ← 1 até n faça 4

para j ← 1 até n faça 5

se lij > lik + lkj então 6 lij ← lik + lkj ;

fim

fim

fim

fim

5.3 Algoritmo A\*

**Algoritmo A\*** (Lê-se: A-estrela) é um algoritmo para **Busca de Caminho**. Ele busca o caminho em um grafo de um vértice inicial até um vértice final. Ele é a combinação de aproximações heurísticas como do algoritmo [Breadth First Search](https://pt.wikipedia.org/wiki/Busca_em_largura" \o "Busca em largura) (Busca em Largura) e da formalidade do [Algoritmo de Dijkstra](https://pt.wikipedia.org/wiki/Algoritmo_de_Dijkstra).

O algoritmo foi descrito pela primeira vez em 1968 por Peter Hart, Nils Nilsson, e Bertram Raphael. Na publicação deles, ele foi chamado de algoritmo A; usando este algoritmo com uma heurística apropriada atinge-se um comportamento ótimo, e passou a ser conhecido por A\*.

Sua aplicação vai desde aplicativos para encontrar rotas de deslocamento entre localidades a resolução de problemas, como a resolução de um quebra-cabeças. Ele é muito usado em jogos.

**Algoritmo**

Sejam

Q = conjunto de nós a serem pesquisados;

S = o estado inicial da busca

Faça:

Inicialize Q com o nó de busca (S) como única entrada;

Se Q está vazio, interrompa. Se não, escolha o melhor elemento de Q;

Se o estado (n) é um objetivo, retorne n;

(De outro modo) Remova n de Q;

Encontre os descendentes do estado (n) que não estão em visitados e crie todas as extensões de n para cada descendente;

Adicione os caminhos estendidos a Q e vá ao passo 2;

caminhos expandidos;

Uma estimativa que sempre subestima o comprimento real do caminho ate o objetivo é chamada de admissível. O uso de uma estimativa admissível garante que a busca de custo-uniforme ainda encontrará o menor caminho.

Pseucodigo

6. Referências

* [Depth First Search (DFS) Explained: Algorithm, Examples, and Code - YouTube](https://www.youtube.com/watch?v=PMMc4VsIacU" \o "Depth First Search (DFS) Explained: Algorithm, Examples, and Code - YouTube)www.youtube.com
* [busca em profundidade grafos python - YouTube](https://www.youtube.com/results?search_query=busca+em+profundidade+grafos+python" \o "busca em profundidade grafos python - YouTube)www.youtube.com
* [busca em profundidade grafos - YouTube](https://www.youtube.com/results?search_query=busca+em+profundidade+grafos" \o "busca em profundidade grafos - YouTube)www.youtube.com
* [BUSCA EM PROFUNDIDADE IMPLEMENTAÇÃO - YouTube](https://www.youtube.com/results?search_query=+BUSCA+EM+PROFUNDIDADE+IMPLEMENTA%C3%87%C3%83O" \o "BUSCA EM PROFUNDIDADE IMPLEMENTAÇÃO - YouTube)www.youtube.com
* [BUSCA EM PROFUNDIDADE SOLUÇÃO - YouTube](https://www.youtube.com/results?search_query=+BUSCA+EM+PROFUNDIDADE+SOLU%C3%87%C3%83O" \o "BUSCA EM PROFUNDIDADE SOLUÇÃO - YouTube)www.youtube.com
* [BUSCA EM PROFUNDIDADE RESOLUÇÃO - YouTube](https://www.youtube.com/results?search_query=+BUSCA+EM+PROFUNDIDADE+RESOLU%C3%87%C3%83O" \o "BUSCA EM PROFUNDIDADE RESOLUÇÃO - YouTube)www.youtube.com
* [PROBLEMA DE BUSCA EM PROFUNDIDADE EXXERCICIO - YouTube](https://www.youtube.com/results?sp=mAEB&search_query=PROBLEMA+DE+BUSCA+EM+PROFUNDIDADE+EXXERCICIO" \o "PROBLEMA DE BUSCA EM PROFUNDIDADE EXXERCICIO - YouTube)www.youtube.com
* [Busca em profundidade (Turma 02) - YouTube](https://www.youtube.com/watch?v=Qq2US_qubnk" \o "Busca em profundidade (Turma 02) - YouTube)www.youtube.com
* [Busca em profundidade - implementação (Turma 01) - YouTube](https://www.youtube.com/watch?v=PXEcdAkqNlM" \o "Busca em profundidade - implementação (Turma 01) - YouTube)www.youtube.com
* [PROBLEMA DE BUSCA EM PROFUNDIDADE PHYTON - YouTube](https://www.youtube.com/results?search_query=PROBLEMA+DE+BUSCA+EM+PROFUNDIDADE+PHYTON" \o "PROBLEMA DE BUSCA EM PROFUNDIDADE PHYTON - YouTube)www.youtube.com
* [Depth-First Search (DFS) - Pesquisa Google](https://www.google.com/search?q=Depth-First+Search+(DFS)&oq=Depth-First+Search+(DFS)&aqs=chrome..69i57j0i19l2j0i19i22i30l7.2566j0j7&sourceid=chrome&ie=UTF-8" \o "Depth-First Search (DFS) - Pesquisa Google)www.google.com
* [::Projeto de Pesquisa:: Introdução a Path Finding](https://web.archive.org/web/20090327141634/http:/www2.dc.uel.br/~rarosa/pixel/path.html)web.archive.org
* [algoritmo de busca de caminho pseudocodigo - Pesquisa Google](https://www.google.com/search?q=algoritmo+xde+busca+de+caminho+pseudocodigo&biw=1366&bih=625&ei=7bPmYpDFKp7y1sQPwIutqAc&ved=0ahUKEwjQ7_b3y6P5AhUeuZUCHcBFC3UQ4dUDCA4&uact=5&oq=algoritmo+xde+busca+de+caminho+pseudocodigo&gs_lcp=Cgdnd3Mtd2l6EAM6BwgAEEcQsAM6BAgAEA06BggAEB4QDToICAAQHhAIEA06BAghEAo6BQgAEKIESgQIQRgASgQIRhgAUNsDWMAfYL4iaAFwAXgAgAGvAYgB_BSSAQQwLjIwmAEAoAEByAEIwAEB&sclient=gws-wiz" \o "algoritmo de busca de caminho pseudocodigo - Pesquisa Google)www.google.com
* [Confira o artigo: Pseudocódigo para algoritmos - Embarcados](https://embarcados.com.br/pseudocodigo/" \o "Confira o artigo: Pseudocódigo para algoritmos - Embarcados)embarcados.com.br
* [RepositorioIAsc/Python/buscas at master · felipemartinsss/RepositorioIAsc](https://github.com/felipemartinsss/RepositorioIAsc/tree/master/Python/buscas" \o "RepositorioIAsc/Python/buscas at master · felipemartinsss/RepositorioIAsc)github.com
* [Algoritmo A\* – Wikipédia, a enciclopédia livre](https://pt.wikipedia.org/wiki/Algoritmo_A*" \o "Algoritmo A* – Wikipédia, a enciclopédia livre)pt.wikipedia.org
* [Buscas em Largura e A\* usando Python – IAsc – Inteligência Artificial sob controle](https://iascblog.wordpress.com/2016/05/10/buscas-em-largura-e-a-star-usando-python/" \o "Buscas em Largura e A* usando Python – IAsc – Inteligência Artificial sob controle)iascblog.wordpress.com
* =
* [Material Didático - IMD](https://materialpublic.imd.ufrn.br/curso/disciplina/5/69/10/3" \l ":~:text=O%20Dijkstra%20%C3%A9%20um%20dos,caminho%20entre%20dois%20v%C3%A9rtices%20espec%C3%ADficos." \o "Material Didático - IMD)materialpublic.imd.ufrn.br
* [Algoritmo de caminho de custo mínimo de Dijkstra - uma introdução detalhada e visual](https://www.freecodecamp.org/portuguese/news/algoritmo-de-caminho-de-custo-minimo-de-dijkstra-uma-introducao-detalhada-e-visual/" \o "Algoritmo de caminho de custo mínimo de Dijkstra - uma introdução detalhada e visual)www.freecodecamp.org
* [Algoritmo de Dijkstra – Wikipédia, a enciclopédia livre](https://pt.wikipedia.org/wiki/Algoritmo_de_Dijkstra" \o "Algoritmo de Dijkstra – Wikipédia, a enciclopédia livre)pt.wikipedia.org
* [Caminho de Custo Mínimo - Algoritmo de Dijkstra](http://www.inf.ufsc.br/grafos/temas/custo-minimo/dijkstra.html" \o "Caminho de Custo Mínimo - Algoritmo de Dijkstra)www.inf.ufsc.br
* [Minimum spanning tree - Wikipedia](https://en.wikipedia.org/wiki/Minimum_spanning_tree" \l "Applications" \o "Minimum spanning tree - Wikipedia)en.wikipedia.org