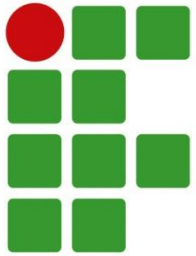


INSTITUTO FEDERAL
Goiás

Instituto Federal de Goiás
Câmpus Goiânia
Bacharelado em Sistemas de Informação
Disciplina: Estruturas de Dados II

Grafos

Prof. Ms. Renan Rodrigues de Oliveira
Goiânia - GO

Introdução a Grafos

Grafos são estruturas de dados bem parecidas com árvores.

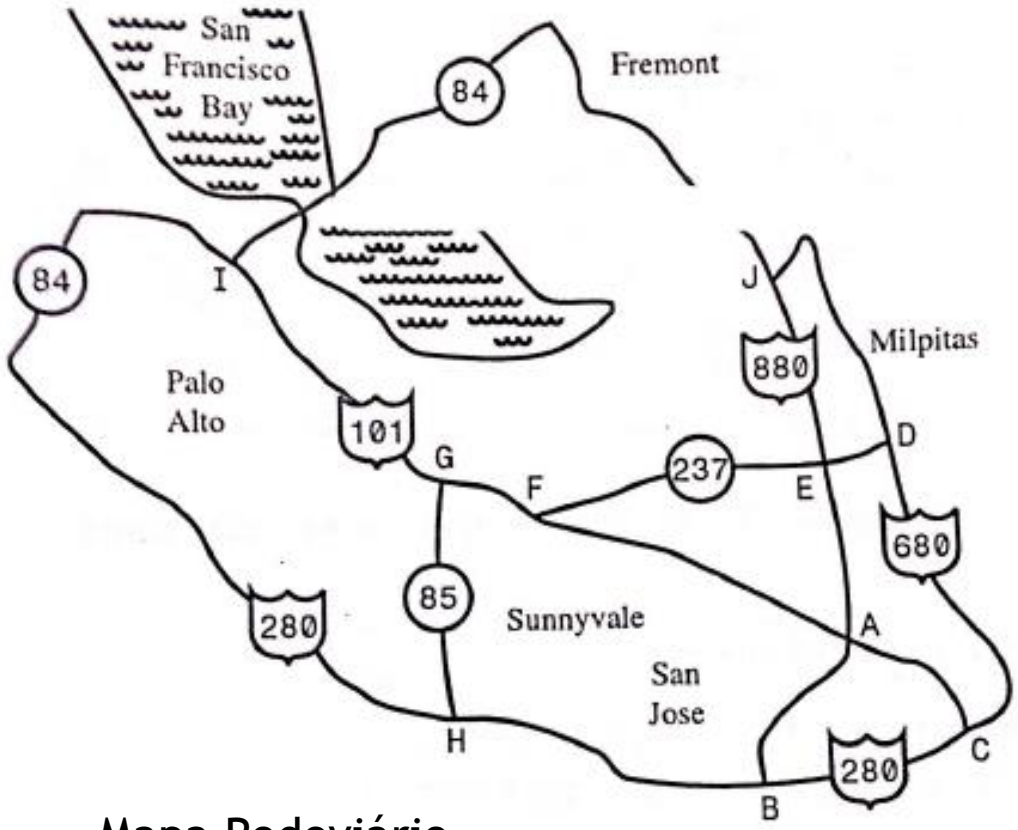
Em um sentido matemático, uma árvore é um tipo de grafo.

- No entanto, em programação de computadores, grafos são usados de maneiras diferentes de árvores.
- Se você estiver lidando com tipos gerais de problemas de armazenamento de dados, provavelmente você não precisará de uma grafo.
- Mas para alguns problemas – e eles tendem a ser bem interessantes – um grafo é indispensável.

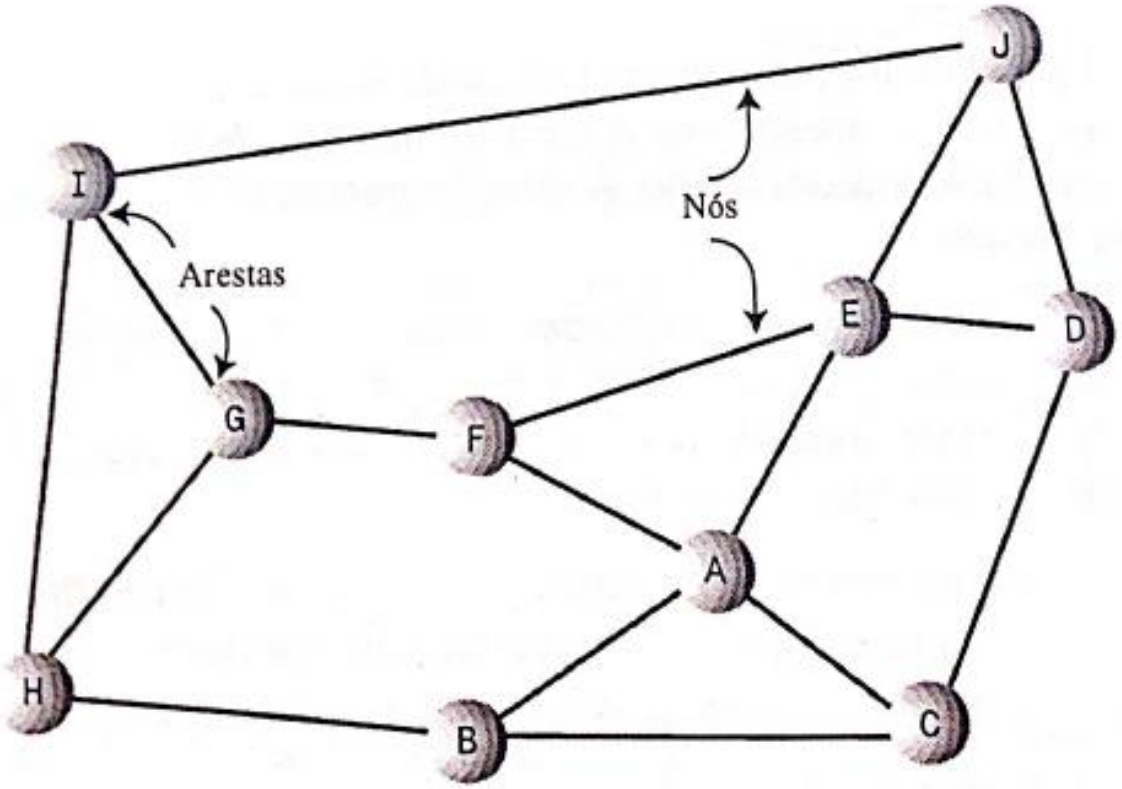
Introdução a Grafos

- **As estruturas de dados examinadas anteriormente têm uma arquitetura ditada pelos algoritmos nelas usadas.**
  - Por exemplo, uma árvore binária é formada do modo que possibilite uma maneira fácil de buscar dados e inserir novos dados.
  - As arestas de uma árvore representam maneiras rápidas de ir de um nó para outro.
- **Geralmente, grafos têm uma forma ditada por um problema físico ou abstrato.**
  - Por exemplo, nós em um grafo podem representar cidades e arestas podem representar rotas de voos de linhas aéreas entre cidades.
  - Outro exemplo mais abstrato é um grafo representando tarefas individuais necessárias para completar um projeto.

Introdução a Grafos



Mapa Rodoviário

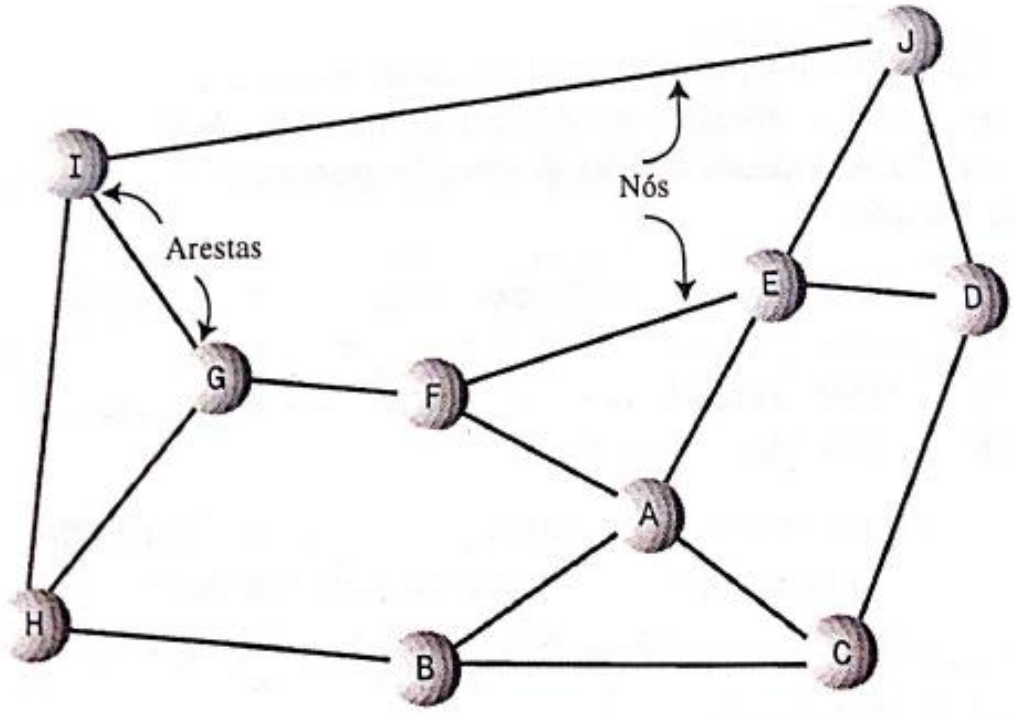


Grafo

Definições de Grafos

Adjacência

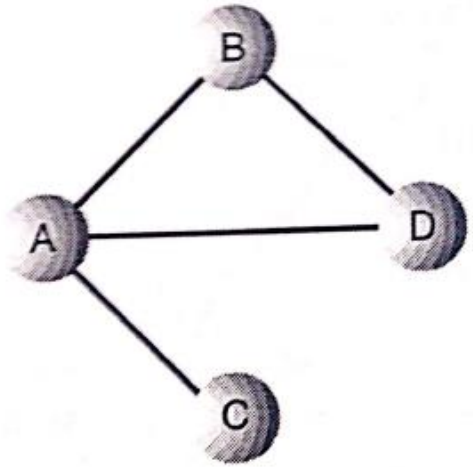
- Dois nós são ditos adjacentes um ao outro se forem conectados por uma única aresta.
- Assim, os nós I e G são adjacentes, mas os nós I e F não são.
- Os nós adjacentes a um determinado nó são algumas vezes ditos como vizinhos. Por exemplo, os vizinhos de G são I, H e F.



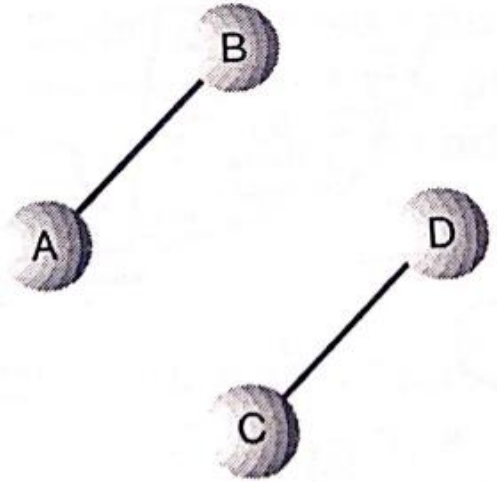
Definições de Grafos

Grafos Conectados

- Um grafo é dito conectado se houver pelo menos um caminho de cada nó para cada outro nó.
- “Se você não puder chegar lá a partir daqui”, o grafo é conectado.



Grafo Conectado

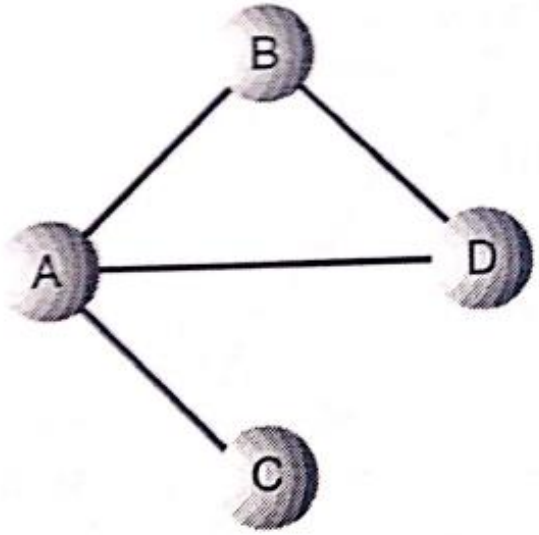


Grafo Não Conectado

Definições de Grafos

Grafos Não Orientados

- As arestas de um grafo não orientado não têm uma direção.
- Isto significa que você pode ir para qualquer lado nelas.

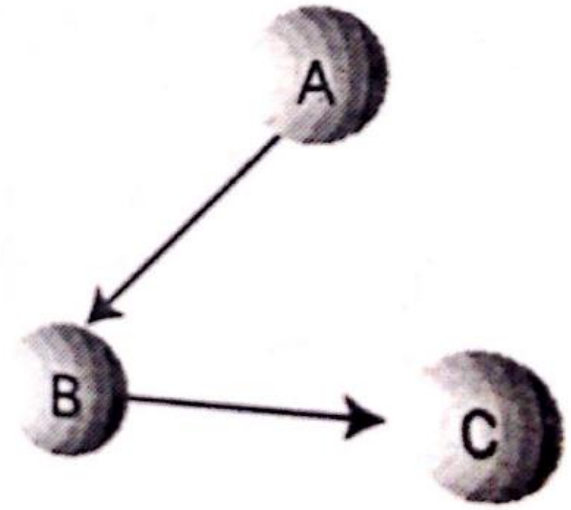


Grafo Não Orientado

Definições de Grafos

Grafos Orientados

- Grafos orientados são geralmente usados para modelar situações nas quais você pode ir em apenas uma direção em uma aresta.
- No grafo abaixo, você pode ir de A para B, mas não de B para A.
- A direção permitida é geralmente mostrada com uma seta na ponta da aresta.

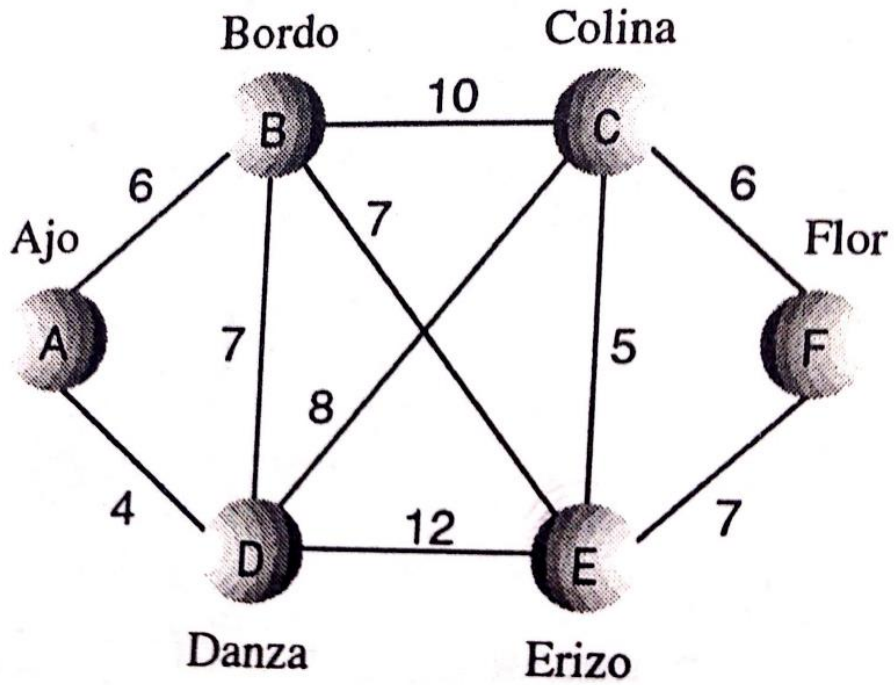


Grafo Orientado

Definições de Grafos

Grafos Ponderados

- Nos grafos ponderados, as arestas recebem pesos.
- O peso pode representar a distância física entre dois nós, o tempo que leva para ir de um nó ao outro ou quanto custa viajar de um nó para outro.

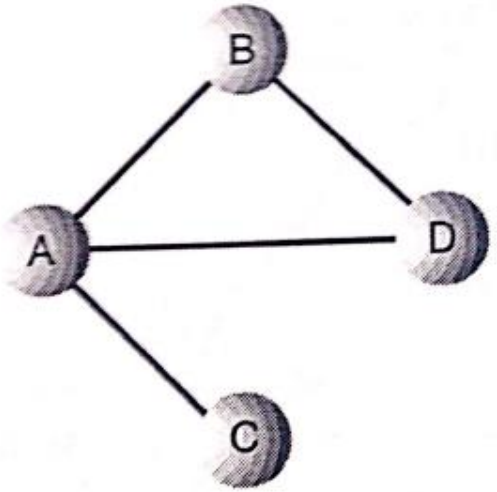


Grafo Ponderado

Definições de Grafos

Arestas

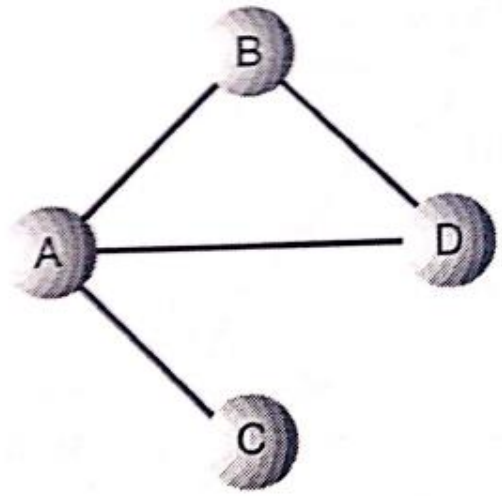
- Em um grafo, cada nó pode ser conectado a um número arbitrário de outros nós.
- No grafo abaixo, o nó A é conectado a três outros nós, ao posso que C é conectado a apenas um.
- Para modelar esse tipo de estrutura livre de forma, uma abordagem diferente para representar arestas é preferível àquela utilizada em árvores.



Definições de Grafos

Matriz de Adjacência

- Uma matriz de adjacência é uma matriz na qual os elementos indicam se uma aresta está presente entre dois nós.
- Se uma matriz tiver N nós, a matriz de adjacência será uma matriz com dimensões NxN.
- Uma aresta entre dois nós é indicado por 1; a ausência de uma aresta é um 0.

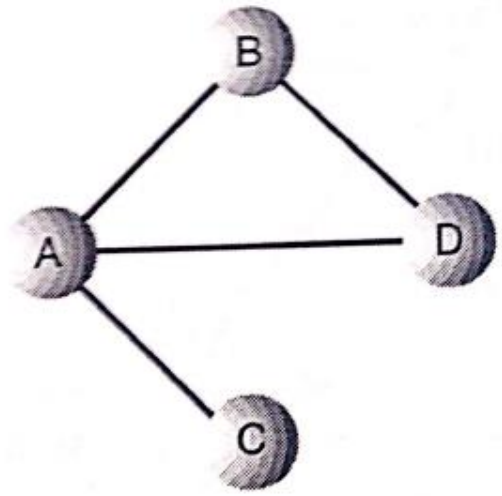


	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

Definições de Grafos

Matriz de Adjacência

- Como o grafo abaixo não existe conexão de um nó consigo mesmo, a diagonal de identidade (AA a DD) é toda composta por 0.
- Observe que a parte triangular da matriz acima da diagonal de identidade é uma imagem espelhada da parte abaixo. As duas partes contém a mesma informação.

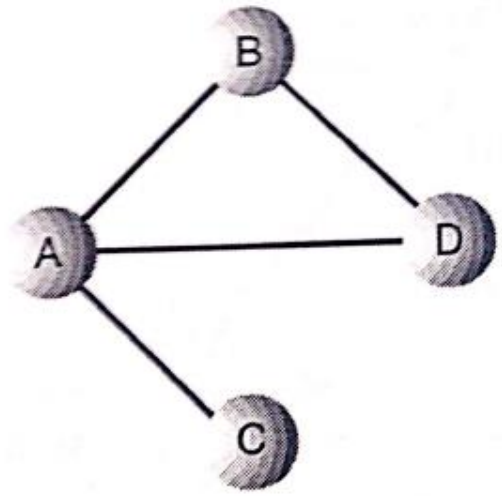


	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

Definições de Grafos

Matriz de Adjacência

- Essa redundância pode parecer ineficiente, mas não há uma maneira de criar um vetor triangular na maioria das linguagem de computador.
- Consequentemente, quando você adicionar uma aresta ao grafo, terá que criar duas entradas na matriz de adjacência, ao invés de uma.

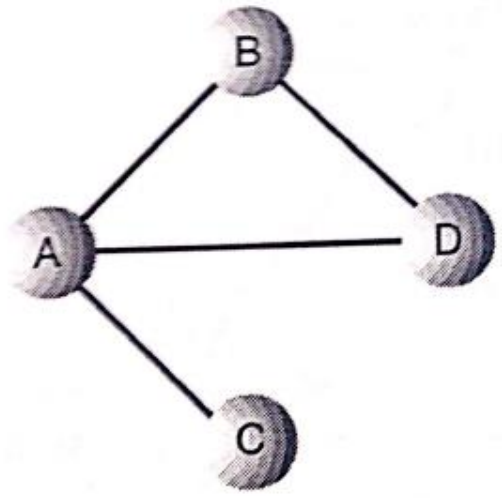


	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

Definições de Grafos

Lista de Adjacências

- Um lista de adjacência é um vetor de listas.
- Cada lista individual mostra a quais nós um dado nó é adjacente.

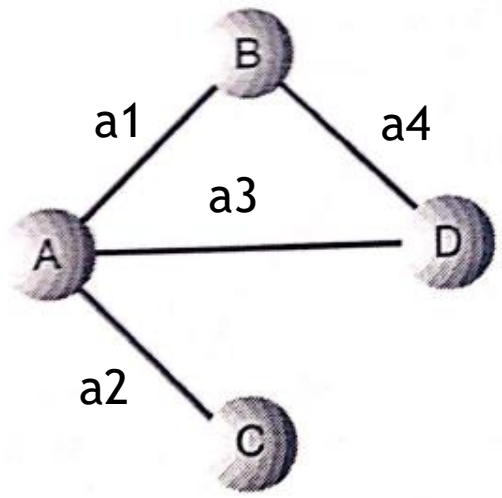


Nó	Lista Contendo Nós Adjacentes
A	B ->C -> D
B	A -> D
C	A
D	A -> B

Definições de Grafos

Matriz de Incidência

- A matriz de incidência associa vértices às linhas e arestas às colunas.
- O elemento da matriz indica se aresta incide sobre o vértice.
- Matriz n x m (n vértices, m arestas):
  - a_{ij} = 1 , se vértice i incide sobre aresta j
  - a_{ij} = 0 , caso contrário.



	a1	a2	a3	a4
A	1	1	1	0
B	1	0	0	1
C	0	1	0	0
D	0	0	1	1

Buscas

Uma das operações mais fundamentais para executar em um grafo é localizar quais nós podem ser alcançados.

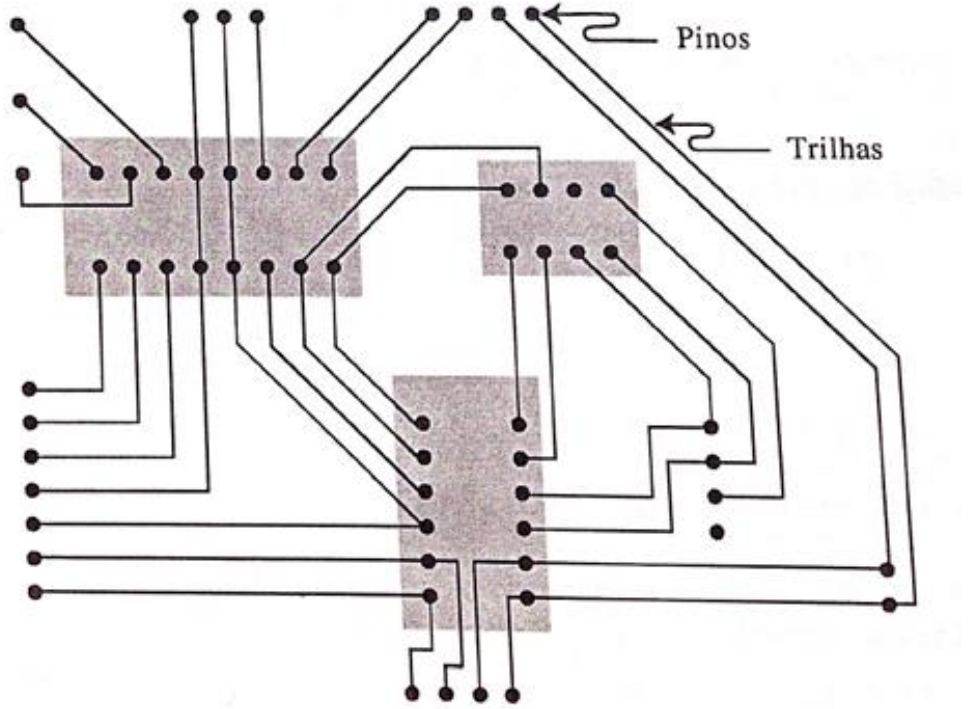
Imagine descobrir quantas cidades podem ser alcançadas em uma viagem de trem a partir de uma outra cidade de origem.

- Algumas cidades poderiam ser alcançadas.
- Outras não, porque não possui serviço de estrada de ferro, ou não está conectada a linha de trem da cidade de origem.

Buscas

Imagine que você esteja projetando uma placa de circuito impresso, onde vários circuitos integrados são colocados na placa.

- Os CI's são soldados e seus pinos são conectados eletricamente a outros pinos por trilhas.
- Em um grafo, cada pino poderia ser representado por um nó e cada trilha por uma aresta.
- Durante o processo de projeto, poderá ser útil criar um grafo e usá-lo para encontra quais pinos estão conectados ao mesmo circuito.



Grafos - Implementação em C

```c
#define MAX_VERT 20

typedef struct {
    char Cidade[30];
    char Gentilico[20];
    char Prefeito[20];
} TipoRegistro;

typedef struct {
    int Ordem;
    TipoRegistro Item;
    int FoiVisitado;
} TipoVertice;
```

```c
typedef struct {
    TipoVertice Reg[50];
    int n;
} TipoPilha;

typedef struct {
    TipoVertice ListaVertices[MAX_VERT];
    int MatrizAdj[MAX_VERT][MAX_VERT];
    TipoPilha Pilha;
    int n;
} TipoGrafo;
```

Grafos - Implementação em C

```c
void InicializaGrafo (TipoGrafo *Grafo) {
  InicializaPilha(&(Grafo->Pilha));
  Grafo->n = 0;

    for(int i=0; i<MAX_VERT; i++) {
        for(int j=0; j<MAX_VERT; j++) {
            Grafo->MatrizAdj[i][j] = 0;
        }
    }
}
```

```c
void adicionaVertice(TipoGrafo *Grafo, TipoRegistro Reg) {

    TipoVertice v;
    v.Ordem = Grafo->n;
    v.Item = Reg;
    v.FoiVisitado = 0;

    Grafo->ListaVertices[Grafo->n] = v;
    Grafo->n++;
}
```

Grafos - Implementação em C

```c
void InicializaGrafo (TipoGrafo *Grafo) {
  InicializaPilha(&(Grafo->Pilha));
  Grafo->n = 0;

    for(int i=0; i<MAX_VERT; i++) {
        for(int j=0; j<MAX_VERT; j++) {
            Grafo->MatrizAdj[i][j] = 0;
        }
    }
}
```

```c
void adicionaVertice(TipoGrafo *Grafo, TipoRegistro Reg) {

    TipoVertice v;
    v.Ordem = Grafo->n;
    v.Item = Reg;
    v.FoiVisitado = 0;

    Grafo->ListaVertices[Grafo->n] = v;
    Grafo->n++;
}
```

```c
void adicionarAresta(TipoGrafo* Grafo, int inicio, int fim) {
    Grafo->MatrizAdj[inicio][fim] = 1;
    Grafo->MatrizAdj[fim][inicio] = 1;
}
```

Grafos - Implementação em C

```c
void ListaGrafo(TipoGrafo* Grafo) {

    printf("\nGRAFO\n");
    ImprimeTituloVertice();
    for(int i=0; i < Grafo->n; i++) {
        ImprimeVertice(&(Grafo->ListaVertices[i]));
    }
    printf("---------------------------------------\n");

    printf("%-3s", "");
    for(int i=0; i<Grafo->n; i++) {
        printf("%-3d", Grafo->ListaVertices[i].Ordem);
    }
    printf("\n");

    for(int i=0; i<Grafo->n; i++) {
        printf("%-3d", Grafo->ListaVertices[i].Ordem);
        for(int j=0; j<Grafo->n; j++) {
            printf("%-3d", Grafo->MatrizAdj[i][j]);
        }
        printf("\n");
    }

}
```

Grafos - Implementação em C

```
TipoVertice* AdjNaoVisitado(TipoGrafo* Grafo, TipoVertice* v) {

    for(int j=0; j<Grafo->n; j++) {
        if ( (Grafo->MatrizAdj[v->Ordem][j] == 1) && (Grafo->ListaVertices[j].FoiVisitado == 0) ) {
            return &(Grafo->ListaVertices[j]);
        }
    }

    return NULL;
}
```

Buscas

Há duas abordagens comuns para buscar em um grafo, que resultará no grafo sendo percorrido de maneiras diferentes.

- Busca em Profundidade (DFS – Depth-First Search)
  - É implementada com uma pilha.
- Busca em Largura (BFS – Breadth-First Search)
  - É implementada com uma fila.

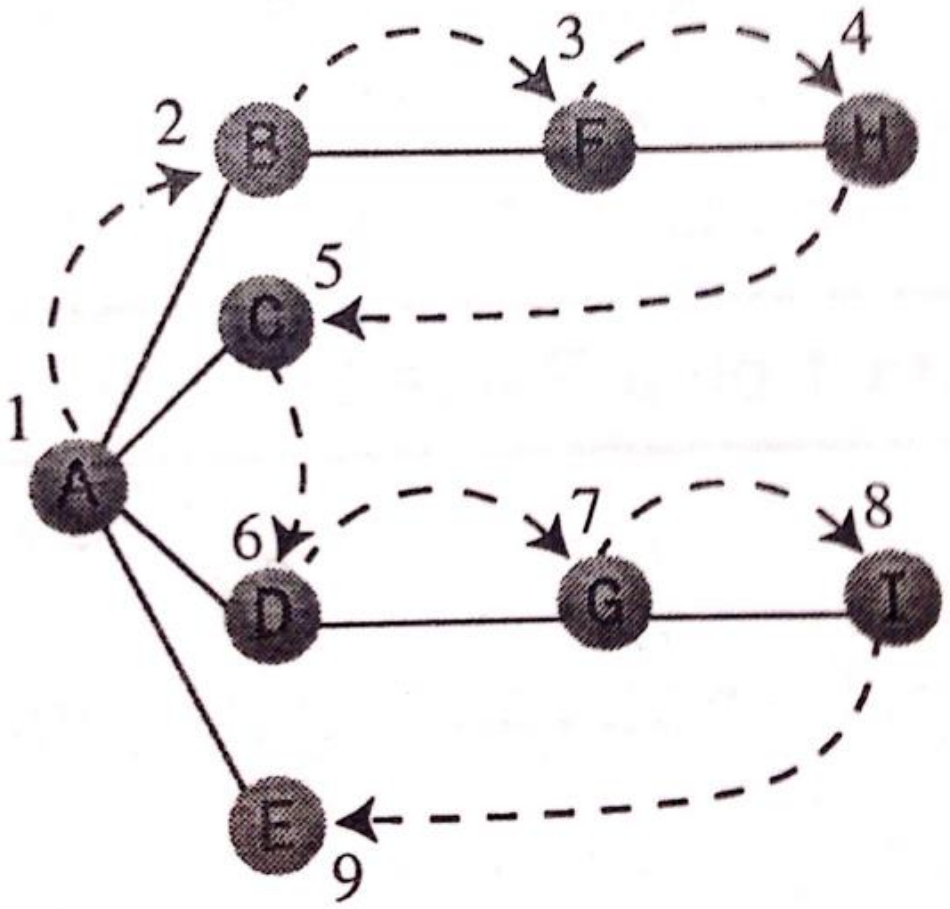
Busca em Profundidade

Na busca em profundidade, o algoritmo age como se quisesse se distanciar do ponto inicial o mais rápido possível.

A busca em profundidade usa uma pilha para lembra para onde deve ir quando atinge um ponto sem saída. Para executar a busca em profundidade, selecione um ponto de partida. Então faça três coisas:

- Regra 1: Se possível, visite um nó adjacente não visitado, marque-o e coloque na pilha.
- Regra 2: Se você não puder seguir a Regra 1, então se possível, retire um nó da pilha.
- Regra 3: Se você não puder seguir a Regra 1 ou a Regra 2, então terminou.

Busca em Profundidade



A ordem de visita é:

ABFHCDGIE

	A	B	C	D	E	F	G	H	I
A		1	1	1	1				
B	1					1			
C	1								
D	1						1		
E	1								
F		1						1	
G				1					1
H						1			
I							1		

Evento	Pilha
E(A)	A
E(B)	AB
E(F)	ABF
E(H)	ABFH
D(H)	ABF
D(F)	AB
D(B)	A
E(C)	AC
D(C)	A
E(D)	AD
E(G)	ADG
E(I)	ADGI
D(I)	ADG
D(G)	AD
D(D)	A
E(E)	AE
D(E)	A
D(A)	-
Terminado	

Busca em Profundidade

```c
void BuscaEmProfundidade(TipoGrafo* Grafo) {

    printf("\nDFS\n");
    TipoVertice* vInicio = &(Grafo->ListaVertices[0]);
    vInicio->FoiVisitado = 1;
    ImprimeVertice(vInicio);
    Empilha(&(Grafo->Pilha), vInicio);

    while(!PilhaVazia(&(Grafo->Pilha))) {

        TipoVertice *topo = VerTopo(&(Grafo->Pilha));
        TipoVertice *v = AdjNaoVisitado(Grafo, topo);

        if (v == NULL) {
           Desempilha(&(Grafo->Pilha));
        } else {
           v->FoiVisitado = 1;
           ImprimeVertice(v);
           Empilha(&(Grafo->Pilha), v);
        }
    }

    ZerarFlagsVisitado(Grafo);

}
```

```c
void ZerarFlagsVisitado(TipoGrafo* Grafo) {
    for(int j=0; j<Grafo->n; j++) {
        Grafo->ListaVertices[j].FoiVisitado = 0;
    }
}
```

```c
TipoVertice* AdjNaoVisitado(TipoGrafo* Grafo, TipoVertice* v) {

    for(int j=0; j<Grafo->n; j++) {
        if ( (Grafo->MatrizAdj[v->Ordem][j] == 1) &&
                (Grafo->ListaVertices[j].FoiVisitado == 0) ) {
            return &(Grafo->ListaVertices[j]);
        }
    }

    return NULL;
}
```

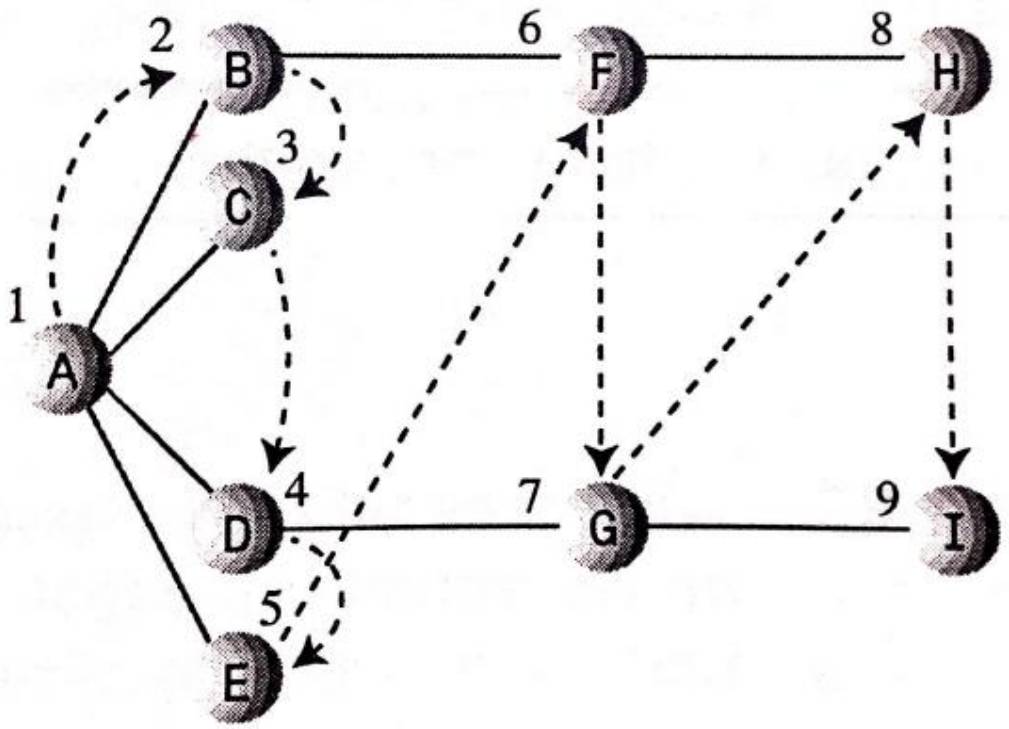
Busca em Largura

Na busca em largura, o algoritmo gosta de ficar o mais próximo possível do ponto inicial.

A busca em largura visita todos os nós adjacentes ao nó inicial e só depois vai adiante. Este algoritmo pode ser implementado usando uma fila.

- Regra 1: Visite o próximo nó não visitado (se houver um) que seja adjacente ao nó atual, marque-o e insira-o em uma fila.
- Regra 2: Se você não puder executar a Regra 1 porque não há mais nós não visitado, remova um nó da fila (se possível) e torne-o como nó atual.
- Regra 3: Se não puder executar a Regra 2, é porque a fila está vazia e o algoritmo terminou.

Busca em Largura



	A	B	C	D	E	F	G	H	I
A		1	1	1	1				
B	1					1			
C	1								
D	1						1		
E	1								
F		1						1	
G				1					1
H						1			
I							1		

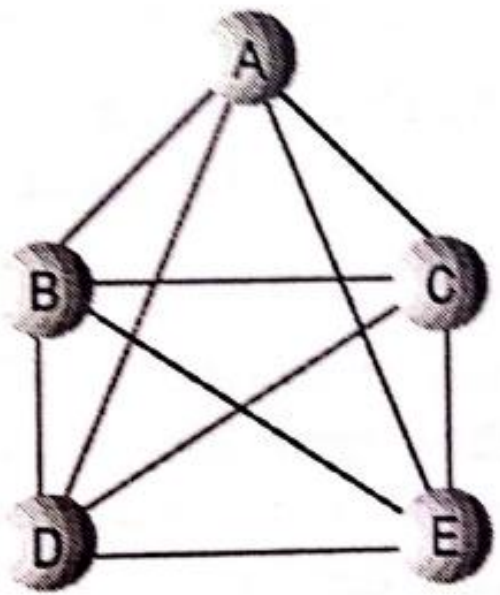
Evento	Fila
E(A)	A
E(B)	AB
E(C)	ABC
E(D)	ABCD
E(E)	ABCDE
D(A)	BCDE
E(F)	BCDEF
D(B)	CDEF
D(C)	DEF
E(G)	DEFG
D(D)	EFG
D(E)	FG
E(H)	FGH
D(F)	GH
E(I)	GHI
D(G)	HI
D(H)	I
D(I)	FIM

A ordem de visita é:

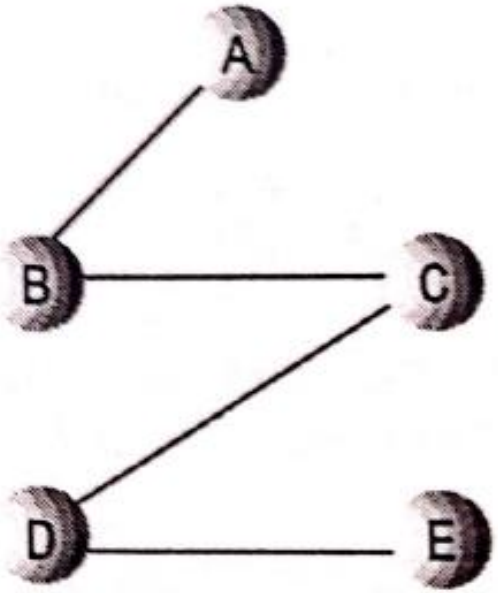
ABCDEFGHI

Árvores Geradoras Mínimas

Uma árvore geradora mínima é um grafo com um número mínimo de arestas necessárias para conectar um grafo.



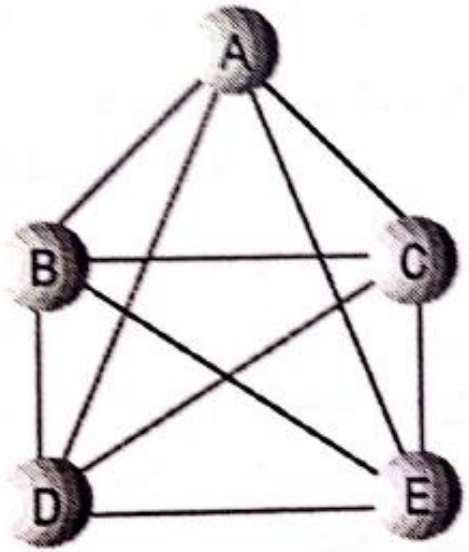
Arestas Extras



Número Mínimo de Arestas

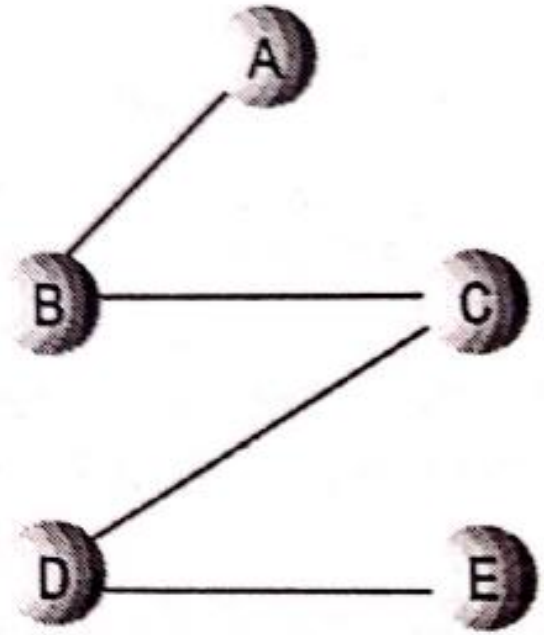
Árvores Geradoras Mínimas

Ao executar um algoritmo para buscar (profundidade ou largura) e ir registrando as arestas pela quais viajou para fazer a busca, você automaticamente criará uma árvore geradora mínima.



	A	B	C	D	E
A		1	1	1	1
B	1		1	1	1
C	1	1		1	1
D	1	1	1		1
E	1	1	1	1	

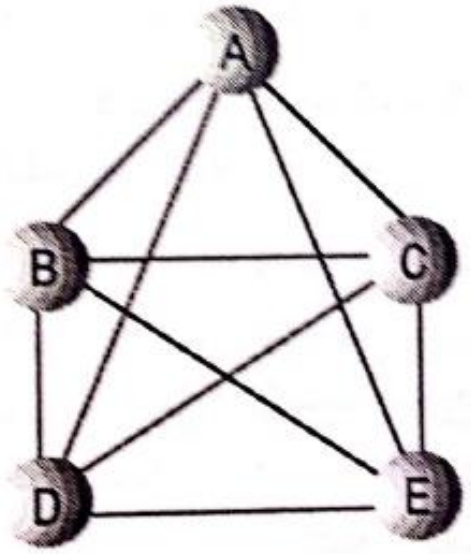
Evento	Pilha	Caminho
E(A)	A	
E(B)	AB	AB
E(C)	ABC	AB BC
E(D)	ABCD	AB BC CD
E(E)	ABCDE	AB BC CD DE
D(E)	ABCD	
D(D)	ABC	
D(C)	AB	
D(B)	A	
D(A)	-	
Terminado		



Árvores Geradoras Mínimas

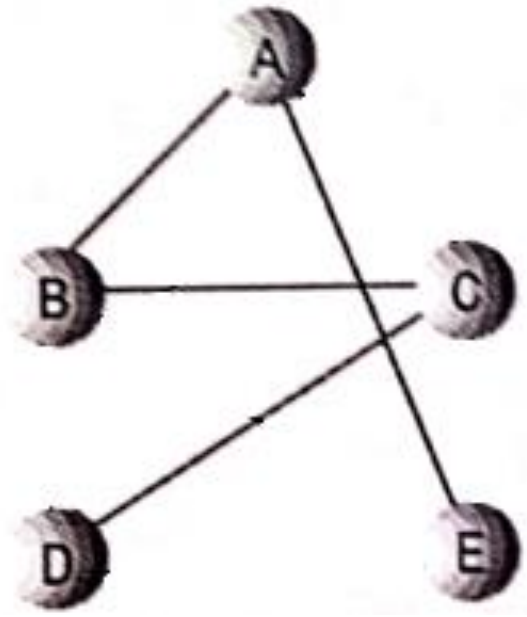
Usar um nó inicial diferente, resulta em árvores diferentes.

- O número de arestas em uma árvore geradora mínima é sempre um a menos que o números de nós.



	A	B	C	D	E
A		1	1	1	1
B	1		1	1	1
C	1	1		1	1
D	1	1	1		1
E	1	1	1	1	

Evento	Pilha	Caminho
E(E)	E	
E(A)	EA	EA
E(B)	EAB	EA AB
E(C)	EABC	EA AB BC
E(D)	EABCD	EA AB BC CD
D(D)	EABC	
D(C)	EAB	
D(B)	EA	
D(A)	E	
D(E)	-	
Terminado		



Árvores Geradoras Mínimas

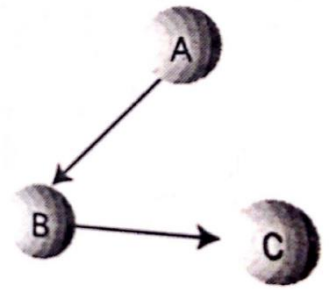
```
void AGM(TipoGrafo* Grafo, int inicio) {

    printf("\nARVORE GERADORA MINIMA\n");
    TipoVertice* vInicio = &(Grafo->ListaVertices[inicio]);
    vInicio->FoiVisitado = 1;
    Empilha(&(Grafo->Pilha), vInicio);

    while(!PilhaVazia(&(Grafo->Pilha))) {

        TipoVertice *topo = VerTopo(&(Grafo->Pilha));
        TipoVertice *v = AdjNaoVisitado(Grafo, topo);

        if (v == NULL) {
           Desempilha(&(Grafo->Pilha));
        } else {
           v->FoiVisitado = 1;
           Empilha(&(Grafo->Pilha), v);
           printf("%s -> %s\n", topo->Item.Cidade, v->Item.Cidade);

        }
    }

    ZerarFlagsVisitado(Grafo);

}
```

Grafos Orientados

Grafos orientados são geralmente usados para modelar situações nas quais você pode ir em apenas uma direção em uma aresta.

A diferença de um grafo não orientado e orientado é que uma aresta em um grafo orientado tem apenas uma entrada na matriz de adjacência.

- Os rótulos de linhas mostram onde a aresta começa e os rótulos de colunas mostram onde termina.
- Em um grafo orientado, toda célula na matriz transmite uma informação única.
- As metades não são imagens espelhadas.



	A	B	C
A		1	
B			1
C			

Ordenação Topológica

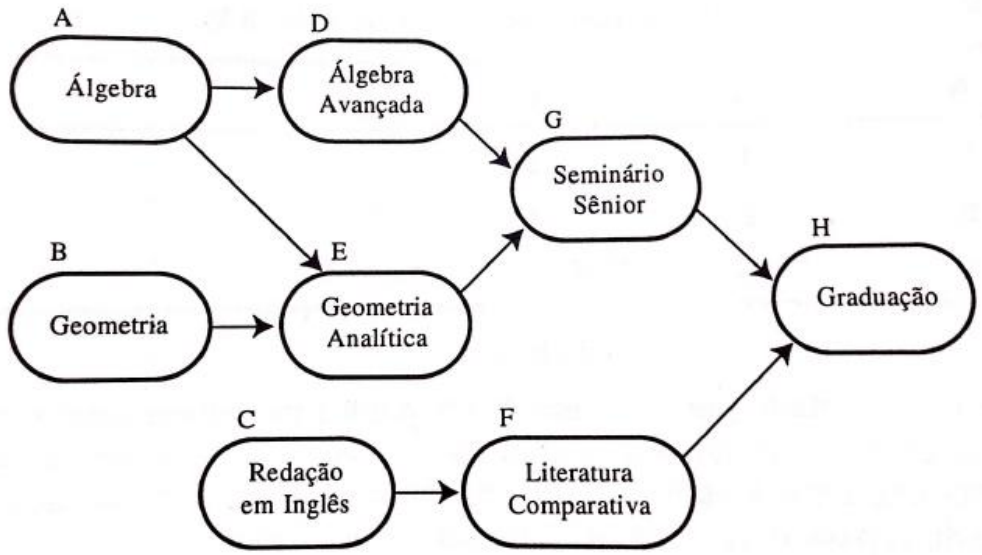
A aplicação da ordenação topológica está na programação de uma sequência de trabalhos ou tarefas a ser realizada.

No grafo abaixo, as disciplinas estão organizadas de forma que representa as disciplinas necessárias realizar a graduação em matemática, considerando seus pré-requisitos.

- Obter a graduação é o último item na lista, que poderia ficar assim:

  BAEDGCFH

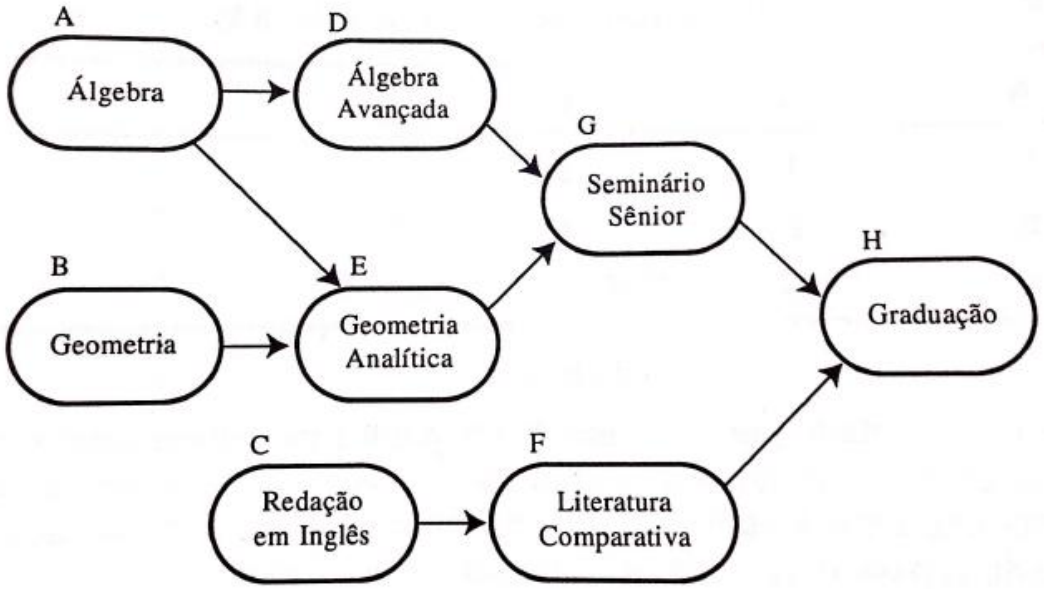
- Muitas ordens possíveis satisfariam os pré-requisitos das disciplinas.
- Para o ordenação topológica, o grafo não pode ter ciclos.



Ordenação Topológica

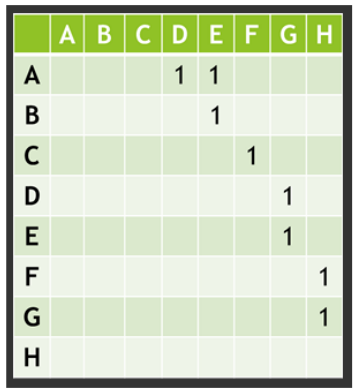
A ideia por trás do algoritmo de ordenação topológica é simples:

- Passo 1: Localize um nó que não tenha sucessores.
- Passo 2: Elimine esse nó do grafo e insira seu rótulo no início de uma lista.

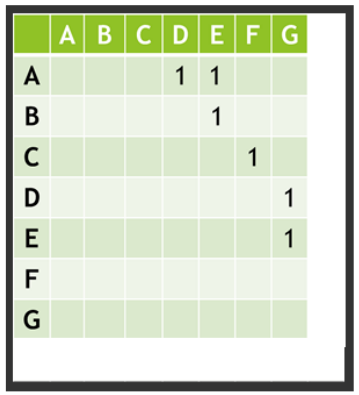


	A	B	C	D	E	F	G	H
A				1	1			
B					1			
C						1		
D							1	
E							1	
F								1
G								1
H								

Ordenação Topológica

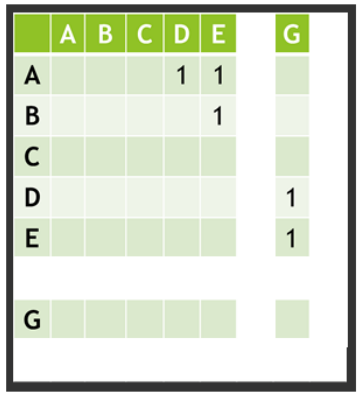


	A	B	C	D	E	F	G	H
A				1	1			
B					1			
C						1		
D							1	
E							1	
F								1
G								1
H								



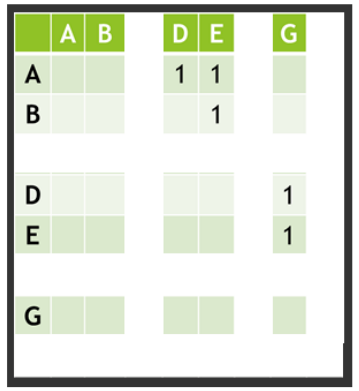
	A	B	C	D	E	F	G
A				1	1		
B					1		
C						1	
D							1
E							1
F							
G							

R(H)



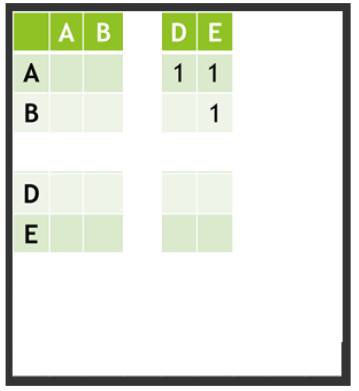
	A	B	C	D	E	G
A				1	1	
B					1	
C						
D						1
E						1
G						

R(F)



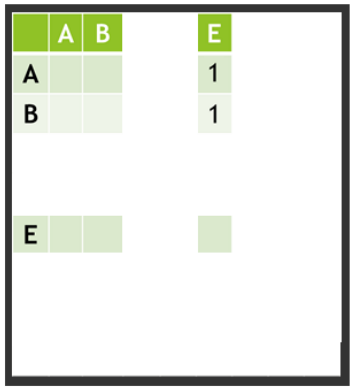
	A	B	D	E	G
A			1	1	
B				1	
D					1
E					1
G					

R(C)



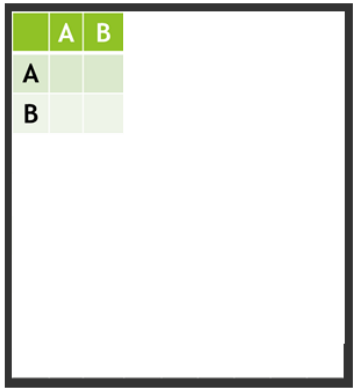
	A	B	D	E
A			1	1
B				1
D				
E				

R(G)



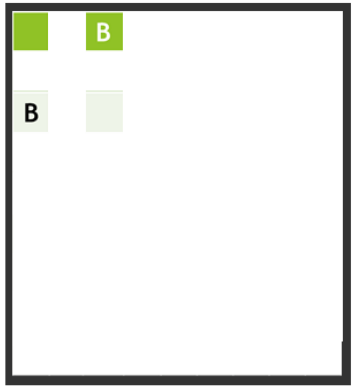
	A	B	E
A			1
B			1
E			

R(D)



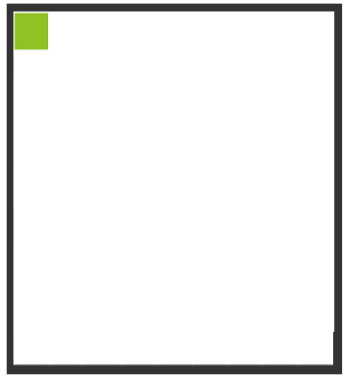
	A	B
A		
B		

R(E)



	B
B	

R(A)



R(B)

Evento	Fila
R(H)	H
R(F)	FH
R(C)	CFH
R(G)	GCFH
R(D)	DGCFH
R(E)	EDGCFH
R(A)	AEDGCFH
R(B)	BAEDGCFH

Ciclos e Árvores

Um ciclo modela uma situação sem saída. É uma caminho que terminar onde começou.

É fácil descobrir se um grafos tem ciclos: se um grafo com N nós tiver mais de N-1 arestas, terá que ter ciclos.

- Uma ordenação topológica tem que ser executada em um grafo orientado sem ciclos.
- Este tipo de gráfico é chamado de grafo acíclico orientado.