Grafos

- Nesta aula é fornecido um breve histórico sobre a teoria dos grafos
- São também introduzidos conceitos sobre grafos e algoritmos que os manipulam

Algoritmos e Estruturas de Dados II

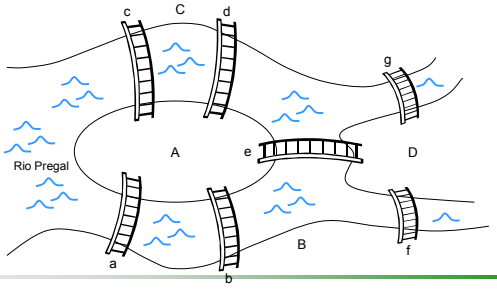
José Augusto Baranauskas
Departamento de Física e Matemática – FFCLRP-USP
augusto@ffclrp.usp.br
http://dfm.ffclrp.usp.br/~augusto

Histórico

- A primeira evidência sobre **grafos** (*graphs*) remonta a 1736, quando Euler fez uso deles para solucionar o problema clássico das pontes de Koenigsberg
- Na cidade de Koenigsberg (na Prússia Oriental), o rio Pregal flui em torno da ilha de Kneiphof, dividindo-se em seguida em duas partes
- Assim sendo, existem quatro áreas de terra que ladeiam o rio: as áreas de terra (A-D) estão interligadas por sete pontes (a-g)
- O problema das pontes de Koenigsberg consiste em se determinar se, ao partir de alguma área de terra, é possível atravessar todos os pontos exatamente uma vez, para, em seguida, retornar à área de terra inicial

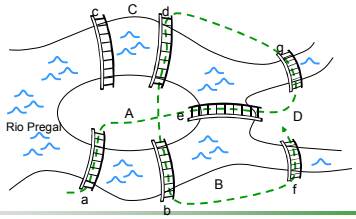
Histórico

- É possível caminhar sobre cada ponte exatamente uma única vez e retornar ao ponto de origem?



Histórico

- Um caminho possível consistiria em iniciar na área de terra **B**, atravessar a ponte **a** para a ilha **A**; pegar a ponte **e** para chegar à área **D**, atravessar a ponte **g**, chegando a **C**; cruzar a ponte **d** até **A**; cruzar a ponte **b** até **B** e a ponte **f**, chegando a **D**



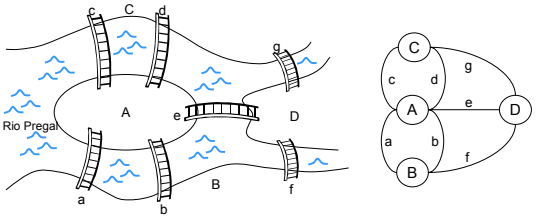
Histórico

- Um caminho possível consistiria em iniciar na área de terra **B**, atravessar a ponte **a** para a ilha **A**; pegar a ponte **e** para chegar à área **D**, atravessar a ponte **g**, chegando a **C**; cruzar a ponte **d** até **A**; cruzar a ponte **b** até **B** e a ponte **f**, chegando a **D**
- Esse caminho não atravessa todas as pontes uma vez, nem tampouco retorna à área inicial de terra **B**
- Euler provou que não é possível o povo de Koenigsberg atravessar cada ponte exatamente uma vez, retornando ao ponto inicial
- Ele resolveu o problema, representando as áreas de terra como vértices e as pontes como arestas de um grafo (na realidade, um multigrafo)

Definição

- Um grafo G(**V**,**E**) é composto de
  - **V** é um conjunto não-vazio de **vértices**
    - vértice (***vertex***); vértices (***vertices, vertexes***)
  - **E** é um conjunto de **arestas** (***edges***), conectando os **vértices** em **V**
    - Uma **aresta** (**u**,**v**) é um par de vértices, ou seja, $u \in V$ e $v \in V$
- Usaremos a notação
  - **V** ou **V(G)** para representar o conjunto de vértices de G
  - **E** ou **E(G)** para representar o conjunto de arestas de G
  - G ou G=(**V**,**E**) ou G(**V**,**E**) para representar um grafo
  - **n** = |**V**| é o número de vértices
  - **e** = |**E**| é o número de arestas
- Em geral, |**A**| indica a cardinalidade (número de elementos) do conjunto **A**

Problema das Pontes de Koenigsberg como um Grafo

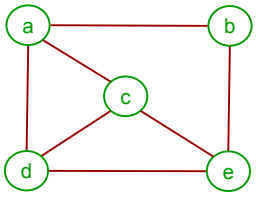


Problema das Pontes de Koenigsberg como um Grafo

- A solução é elegante e tem aplicação a todos os grafos
- Definindo o grau de um vértice como sendo o número de arestas que lhe são incidentes, Euler mostrou que existe um caminho com ponto de início em qualquer vértice, que passa através de cada aresta exatamente uma vez e termina no vértice inicial contanto que o grau de cada vértice seja par
- O caminho que cumprir com essas condições é denominado **Euleriano**
- Não existe nenhum caminho Euleriano nas pontes de Koenigsberg, uma vez que todos os quatro vértices têm grau ímpar

Exemplo

- **V**={a,b,c,d,e}
- **E**={(a,b),(a,c),(a,d), (b,e),(c,d),(c,e), (d,e)}
- Número de vértices, **n**
  - **n** = |**V**| = 5
- Número de arestas, e
  - **e** = |**E**| = 7



Aplicações

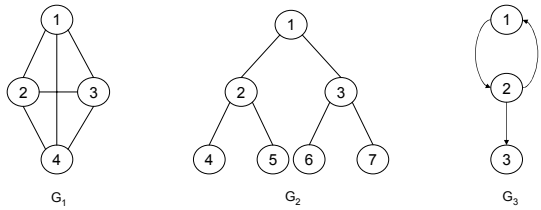
- Análise de circuitos elétricos
- Verificação de caminhos mais curtos
- Análise de planejamento de projetos (*scheduling*)
- Identificação de compostos químicos
- Genética
- Cibernética
- Lingüística
- Ciências Sociais, etc
- Pode-se afirmar que de todas as estruturas matemáticas, grafos são as que se encontram em uso mais amplo

Terminologia

- Em um **grafo não-orientado** (ou **não-dirigido**), não há ordenação especial no par de vértices que representam qualquer aresta
  - Assim sendo, os pares **(u,v)** e **(v,u)** representam a mesma aresta
- Num **grafo orientado** (ou **dirigido** *ou* **dígrafo**) cada aresta é representada por um par dirigido **(u,v)**, onde **u** é o início e **v** o término da aresta
  - Assim sendo **(u,v)** e **(v,u)** representam duas arestas distintas

Terminologia

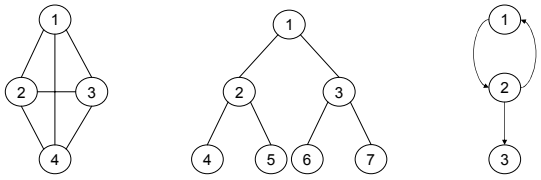
- Os grafos G_1 e G_2 são não-orientados
- O grafo G_3 é um grafo orientado (dígrafo)
- Observe que as arestas de um grafo orientado são desenhadas com uma seta que vai do início até o término



G_1 G_2 G_3

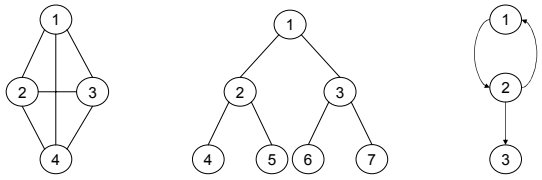
Terminologia

- $V(G_1)=\{1,2,3,4\}$; $E(G_1)=\{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$
- $V(G_2)=\{1,2,3,4,5,6,7\}$; $E(G_2)=\{(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)\}$
- $V(G_3)=\{1,2,3\}$; $E(G_3)=\{(1,2), (2,1), (2,3)\}$



Terminologia

- O grafo G_2 é também uma árvore, ao passo que os grafos G_1 e G_3 não são



Terminologia

- As árvores podem ser definidas como sendo casos especiais de grafos (veremos isso mais adiante)
- Vamos precisar que se (v_i,v_j) ou (v_j,v_i) é uma aresta em E(G), então $v_i \neq v_j$
- Uma vez que E(G) é um conjunto, um grafo não pode ter ocorrências múltiplas da mesma aresta
  - Quando há mais de uma ocorrência de uma mesma aresta, o objeto de dados é denominado **multigrafo**
- O mesmo é valido para V(G), ou seja, não há ocorrências múltiplas de um mesmo vértice

Número Máximo de Arestas

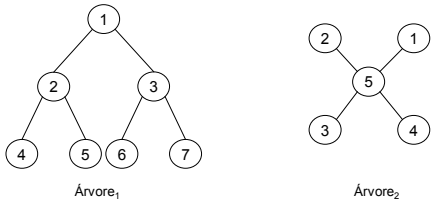
- O número de pares diferentes não-ordenados (v_i,v_j) com $v_i \neq v_j$ em um grafo com **n** vértices é **n***(**n**-1)/2
- Assim, o número máximo de arestas em qualquer grafo não-orientado com **n** vértices é **n***(**n**-1)/2
- Um grafo não-orientado com **n** vértices e com exatamente **n***(**n**-1)/2 arestas é denominado **completo**; caso contrário é denominado **incompleto** (ou **não-completo**)
- Para o caso de um grafo orientado com **n** vértices, o número máximo de arestas é **n***(**n**-1)

Número Máximo de Arestas

- Intuitivamente, em um grafo completo com **n** vértices, cada um dos **n** vértices é incidente a (**n**-1) arestas; assim, cada aresta é contada duas vezes, ou seja, resultando em **n***(**n**-1)/2

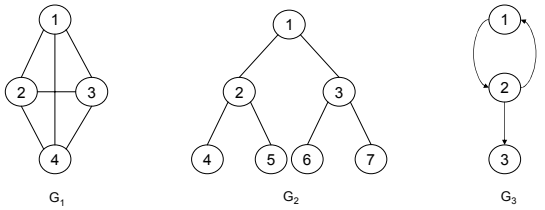
Número Máximo de Arestas em uma Árvore

- Uma árvore com **n** vértices possui exatamente (**n**-1) arestas



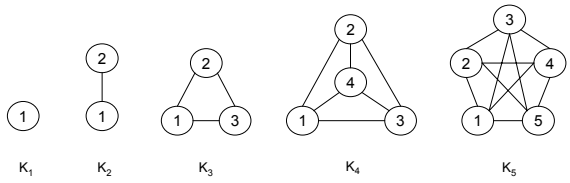
Grafos Completo e Incompleto

- G_1 é o grafo completo com 4 vértices, enquanto que G_2 e G_3 são grafos incompletos



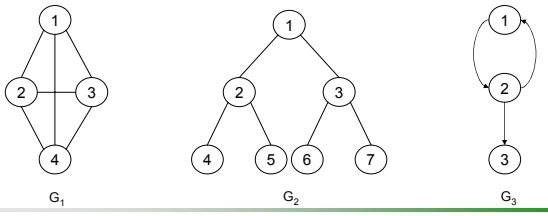
Grafos Completos

- Em geral, **K_n** denota o grafo completo com **n** vértices



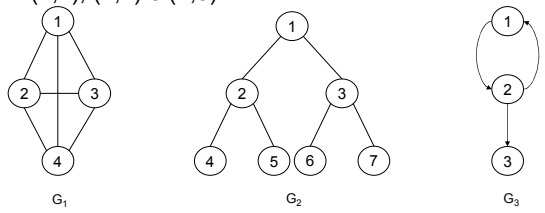
Vértices Adjacentes

- Sendo **(u,v)** uma aresta em E(G), dizemos que os vértices **u** e **v** são **adjacentes** e que a aresta **(u,v)** é **incidente** nos vértices **u** e **v**



Vértices Adjacentes

- Os vértices adjacentes ao vértice 3 em G_2 são 1, 6 e 7
- Em G_3, as arestas incidentes ao vértice 2 são (1,2), (2,1) e (2,3)



Grau

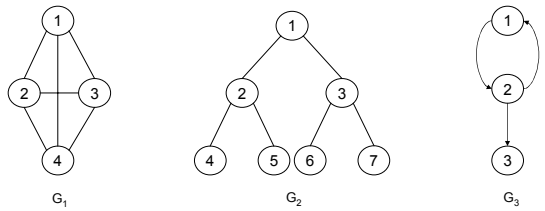
- O **grau** de um vértice **v,** escrito como **grau(v)**, é o número de arestas incidentes no vértice **v**
- Caso G seja um grafo orientado:
  - o **grau de entrada** de um vértice **v** é definido como sendo o número de arestas para as quais **v** seja o término
  - o **grau de saída** é o número de arestas para as quais **v** é o início
- Em grafo G com **n** vértices $\{v_1, v_2, ..., v_n\}$ e **e** arestas, é fácil perceber que

$$e = \frac{1}{2}\sum_{i=1}^{n} grau(v_i)$$

- No restante desta apresentação vamos nos referir a um grafo orientado como dígrafo; um grafo não-orientado será, por vezes, chamado simplesmente de um grafo

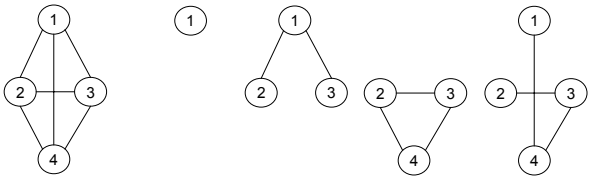
Grau

- O grau de vértice 1 em G_1 é 3
- O vértice 2 de G_3 tem grau de entrada igual a 1, grau de saída igual a 2 e grau igual a 3



Subgrafo

- Um **subgrafo** de G é um grafo G' tal que $V(G') \subseteq V(G)$ e $E(G') \subseteq E(G)$

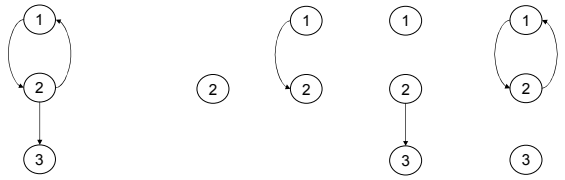


G_1

Subgrafos de G_1

Subgrafo

- Um **subgrafo** de G(V,E) é um grafo G'(V',E') tal que $V(G') \subseteq V(G)$ e $E(G') \subseteq E(G)$

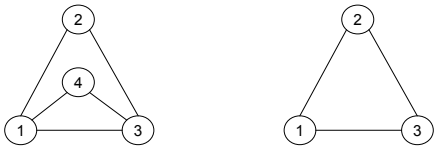


G_3

Subgrafos de G_3

Clique

- O **clique** de um grafo G é um subgrafo de G que seja completo



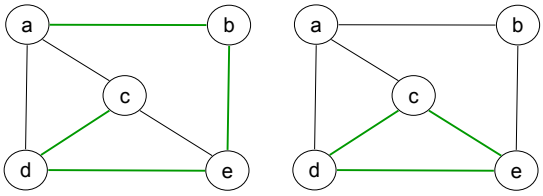
G

clique de G

Caminho

- Um **caminho** (*path*) do vértice v_p para o vértice v_q no grafo G é uma seqüência de vértices v_p, v_{i1}, v_{i2}, ...,v_{in}, v_q de tal maneira que (v_p,v_{i1}), (v_{i1},v_{i2}), ...,(v_{in},v_q) são arestas em E(G)
- O **comprimento** (ou **tamanho**) de um caminho é o número de arestas que ele contém
- Um **caminho simples** é um caminho em que são diferentes todos os vértices, com a possível exceção do primeiro e o do último
- Um **ciclo** é um caminho simples em que o primeiro e o último vértices são iguais
- Um **trajeto** é um caminho no qual todas as arestas são distintas

Caminho

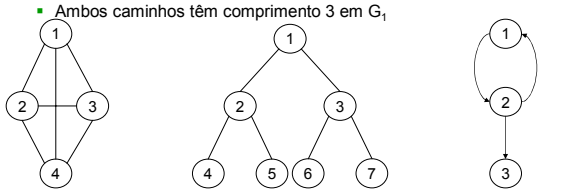


Caminho: a,b,e,d,c
Tamanho: 4

Ciclo: c,e,d,c
Tamanho: 3

Caminho

- O caminho (1,2),(2,4),(4,3) em G_1, também escrito como 1,2,4,3, é caminho simples ao passo que (1,2),(2,4),(4,2) escrito como 1,2,4,2 também é um caminho em G_1, mas não um caminho simples
  - Ambos caminhos têm comprimento 3 em G_1



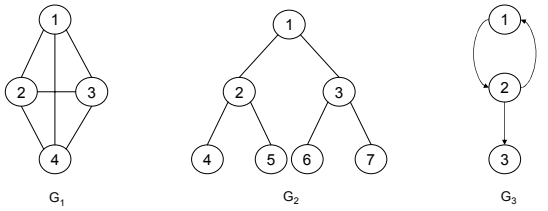
G_1

G_2

G_3

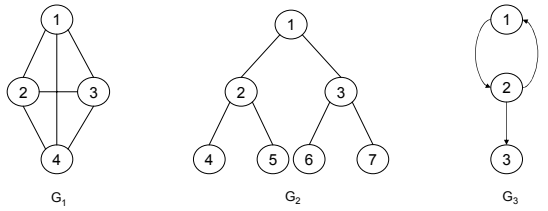
Caminho

- 1,2,3 é um caminho simples orientado em G_3
- 1,2,3,2 não é um caminho em G_3, uma vez que a aresta (3,2) não se encontra em $E(G_3)$



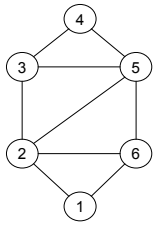
Ciclo

- 1,2,3,1 é um ciclo em G_1
- 1,2,1 é um ciclo orientado em G_3
- Normalmente, para grafos orientados, acrescentamos o termo "orientado" aos termos ciclo e caminho



Caminhos Hamiltoniano e Euleriano

- Caminho ou Ciclo Hamiltoniano
  - É um caminho que contém cada vértice do grafo exatamente uma vez.
  - Um ciclo $v_1, ..., v_k, v_{k+1}$ é hamiltoniano quando o caminho $v_1, ..., v_k$ o for
- Caminho ou Ciclo Euleriano
  - É um caminho que contém cada aresta do grafo exatamente uma vez
- caminho hamiltoniano 3, 4, 5, 2, 1, 6
- caminho euleriano 3, 4, 5, 3, 2, 5, 6, 2, 1, 6

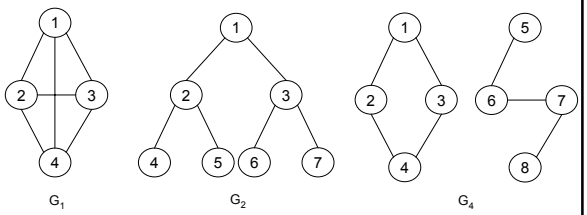


Vértices e Grafos Interligados

- Em um grafo não-orientado G dois **vértices** v_p e v_q se dizem **interligados** se houver um caminho em G desde v_p até v_q (uma vez que G não é orientado, isto significa que há também um caminho desde v_q até v_p)
- Um **grafo** não-orientado se diz **interligado** (**conexo**) se para cada par de vértices individuais v_i e v_j em V(G) existe um caminho desde v_i até v_j em G; caso contrário G é **não-interligado** (**desconexo**)
- Assim, se **e**<(**n**-1) então G é desconexo, onde **e**=|E(G)|, **n**=|V(G)|

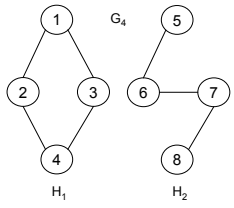
Vértices e Grafos Interligados

- Os grafos G_1 e G_2 são conexos (interligados) ao passo que G_4 não o é



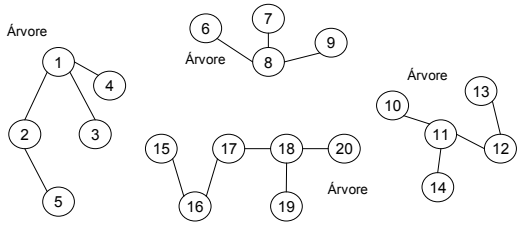
Componente Conexo

- Um **componente conexo**, ou simplesmente um **componente** de grafo não-orientado, é um subgrafo interligado ao máximo
- No exemplo, G_4 tem dois **componentes** H_1 e H_2



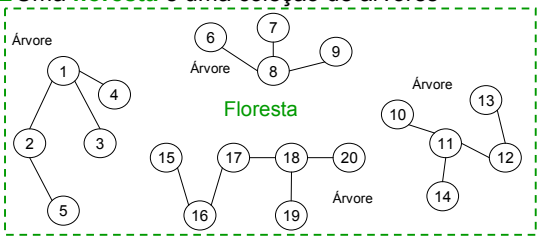
Componente Conexo

- Uma **árvore** é um grafo conexo sem ciclos (acíclico)
- Uma **floresta** é uma coleção de árvores



Componente Conexo

- Uma **árvore** é um grafo conexo sem ciclos (acíclico)
- Uma **floresta** é uma coleção de árvores



Árvore é um Grafo Conexo Acíclico

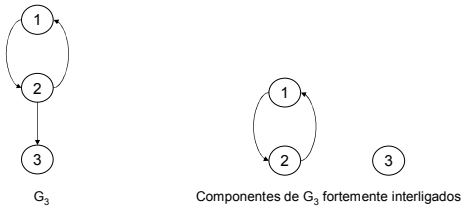
- Teorema:
  - Um grafo G é uma árvore se e somente se existir um único caminho entre cada par de vértices G
- Prova:
  - Se G uma árvore então G é conexo
  - Portanto existe pelo menos um caminho entre cada par de vértices **v** e **w** de G
  - Suponha que existem dois caminhos distintos (**v**,P_1,**w**) e (**v**,P_2,**w**) entre **v** e **w**; então o caminho (**v**,P_1,**w**,P_2,**v**) forma um ciclo, o que contradiz G ser acíclico
  - Reciprocamente, se existe exatamente um caminho entre cada par de vértices de G, então o grafo é obviamente conexo, e além disso não pode conter ciclos
  - Portanto, G é uma árvore

Componente Conexo

- Um **grafo** orientado G se diz **fortemente conexo** (**fortemente interligado**) se para cada par de vértices diferentes v_i e v_j em V(G) existe um caminho orientado desde v_i até v_j e também de v_j para v_i
- Um **componente fortemente interligado** é um subgrafo máximo fortemente interligado

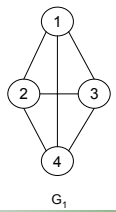
Componente Conexo

- O grafo G_3 não está fortemente interligado, uma vez que não existe nenhum caminho de v_3 até v_2
- G_3 tem dois componentes fortemente interligados



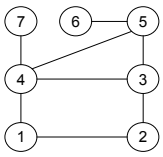
Distância

- A **distância d(v,w)** entre dois vértices **v** e **w** de um grafo é o tamanho do menor caminho entre **v** e **w**
- No exemplo, d(1,4) = 1



Excentricidade

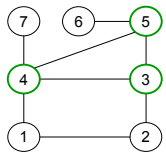
- A excentricidade de um vértice **v** é a distância máxima entre **v** e **w**, para todo **w** de V(G)



Vértice	Excentricidade
1	3
2	3
3	2
4	2
5	2
6	3
7	3

Centro

- O **centro** de um grafo G é o subconjunto de vértices de excentricidade mínima
- No exemplo, centro(G)={3,4,5}



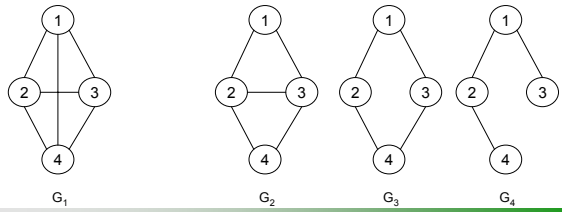
Vértice	Excentricidade
1	3
2	3
3	2
4	2
5	2
6	3
7	3

Subgrafo e Árvore de Cobertura

- O **subgrafo de espalhamento** (*spanning graph*), ou **subgrafo gerador** ou **subgrado estendido**, de um grafo $G_1(V_1,E_1)$ é um subgrafo $G_2(V_2,E_2)$ de G_1, onde $V_1 = V_2$
- Quando o subgrafo de espalhamento é uma árvore, ele recebe o nome de **árvore de espalhamento**, ou **árvore geradora** ou **árvore estendida** (*spanning tree*)
- Dessa forma, a **árvore de geradora** de um grafo G é uma árvore contendo todos os vértices de G
- Um grafo G pode possuir várias árvores de cobertura

Subgrafo e Árvore de Cobertura

- G_2, G_3 e G_4 são subgrafos de cobertura de G_1
- G_3 e G_4 são subgrafos de cobertura de G_2
- G_4 é subgrafo de cobertura de G_3
- G_4 é árvore de cobertura de G_1, G_2 e G_3



Árvore de Cobertura

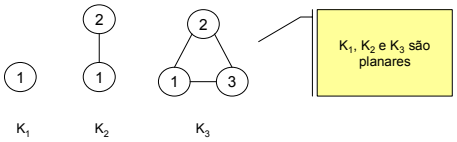
- Todo grafo conexo G(V,E) possui uma árvore de cobertura que pode ser obtida da seguinte forma
  - Para cada aresta (u,v) $\in$ E(G)
    - Se G(V,E-{(u,v)}) for conexo então remova (u,v) de G
- Quando todas as arestas que permanecerem tiverem sido consideradas, o grafo resultante é uma árvore de cobertura de G

Planaridade

- Seja G um grafo e uma representação geométrica de G em um plano (duas dimensões)
- A representação é denominada **plana** quando não houver cruzamento de linhas (exceto nos vértices)
- Um grafo é denominado **planar** se possuir pelo menos uma representação plana
- As linhas de representação dividem o plano em regiões, denominadas **faces**
  - Existe exatamente uma região ilimitada (chamada face externa)
  - Duas representações planas de um mesmo grafo possuem sempre o mesmo número de faces
- Todo grafo planar admite uma representação plana em que todas as linhas são retas

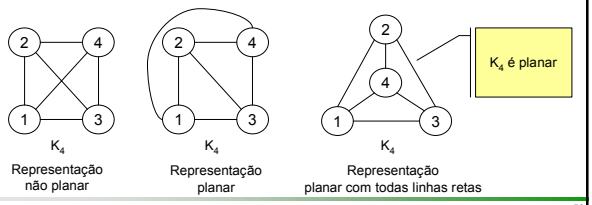
Planaridade

- Se G é um grafo planar então **n**+**f** = **e**+2
  - **n**=|V|, **e**=|E|, **f** é o número de faces
- Se G é um grafo planar então **e** ≤ 3***n**-6
- Assim, quanto maior o número de arestas em relação ao número de vértices, mais difícil se torna obter representações planas



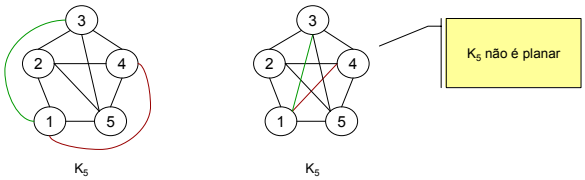
Planaridade

- Se G é um grafo planar então **n**+**f** = **e**+2
  - **n**=|V|, **e**=|E|, **f** é o número de faces
- Se G é um grafo planar então **e** ≤ 3***n**-6



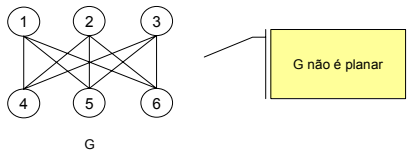
Planaridade

- Se G é um grafo planar então **n**+**f** = **e**+2
  - **n**=|V|, **e**=|E|, **f** é o número de faces
- Se G é um grafo planar então **e** ≤ 3***n**-6



Planaridade

- Se G é um grafo planar então **n**+**f** = **e**+2
  - **n**=|V|, **e**=|E|, **f** é o número de faces
- Se G é um grafo planar então **e** ≤ 3***n**-6

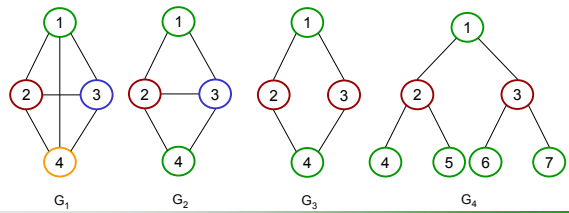


Coloração

- Uma **coloração** de G(V,E) é uma atribuição de alguma cor para cada vértice de V, de forma que dois vértices adjacentes possuam cores distintas
- Uma **k-coloração** é uma coloração que utiliza um máximo de **k** cores
- O **número cromático** de um grafo G é número mínimo de cores **k** para o qual existe uma **k**-coloração de G
- Colorir um grafo é simples, entretanto não é trivial encontrar um algoritmo eficiente para obtenção do número cromático; na realidade ainda é desconhecido um algoritmo eficiente para isso

Coloração

- Numero cromático de $G_1 = 4$; $G_2 = 3$, $G_3 = 2$; $G_4 = 2$

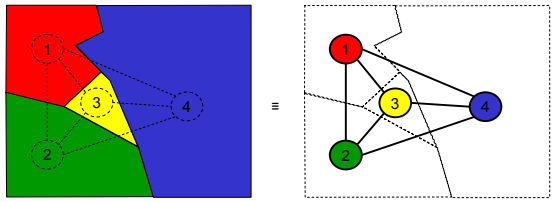


Coloração

- O **problema das 4 cores**, ou **coloração de mapas usando quatro cores**, data de 1852 quando o inglês Francis Guthrie observou que apenas quatro cores eram suficientes para colorir o mapa dos condados da Inglaterra
- O **problema das 4 cores** consiste em colorir os países de um mapa arbitrário plano, cada país com uma cor, de tal forma que países fronteiriços possuam cores diferentes, usando no máximo 4 cores, teorema provado em 1977 por Appel e Haken
- Desde a prova do teorema, os algoritmos mais eficientes encontrados para 4-coloração de mapas quererem $O(n^2)$, onde **n** é o número de vértices

Coloração

- Cada região do mapa é substituída por um vértice
- Dois vértices são conectados por uma aresta se e somente se as duas regiões são fronteiriças (compartilham um segmento de borda)



Especificação

- bool Graph::Empty();
  - retorna **true** se o grafo está vazio; **false** caso contrário
- int Graph::numVertices();
  - Retorna o número de vértices
- int Graph::numEdges();
  - Retorna o número de arestas
- int Graph::Size();
  - Retorna o número de vértices mais arestas
- vertex Graph::Vertex(int i)
  - Retorna o **i**-ésimo vértice, 1 <= **i** <= numVertices()
- edge Graph::Edge(int j)
  - Retorna a **j**-ésima aresta , 1 <= **j** <= numEdges()

Especificação

- float Graph::distance(vertex v, vertex w)
  - Retorna a distância entre os vértices **v** e **w**
- int Graph::degree(vertex v)
  - Retorna o grau de **v**
- int Graph::inDegree(vertex v)
  - Retorna o grau de entrada de **v**
- int Graph::outDegree(vertex v)
  - Retorna o grau de saída de **v**
- edges Graph::incidentEdges(vertex v)
  - Retorna uma enumeração de todas as arestas incidentes ao vértice **v**
- edges Graph::inIncidentEdges(vertex v)
  - Retorna uma enumeração de todas as arestas que chegam no vértice **v**
- edges Graph::outIncidentEdges(vertex v)
  - Retorna uma enumeração de todas as arestas que partem do vértice **v**

Especificação

- vertices Graph::adjacentVertices(vertex v)
  - Retorna uma enumeração dos vértices adjacentes a **v**
- vertices Graph::inAdjacentVertices(vertex v)
  - Retorna uma enumeração dos vértices adjacentes a **v** considerando arestas que chegam a **v**
- vertices Graph::outAdjacentVertices(vertex v)
  - Retorna uma enumeração dos vértices adjacentes a **v** considerando arestas que partem de **v**
- bool Graph::areAdjacent(vertex v, vertex w)
  - Retorna **true** se os vértices **v** e **w** são adjacentes, **false** caso contrário
- vertex Graph::insertVertex(object o)
  - Insere e retorna um novo (isolado) vértice, armazenando **o** na sua posição
- edge Graph::insertEdge(vertex v, vertex w, object o)
  - Insere e retorna uma aresta não orientada entre v e w, armazenando **o** na sua posição
- edge Graph::insertDirectedEdge(vertex v, vertex w, object o)
  - Insere e retorna uma aresta orientada entre **v** e **w**, armazenando **o** na sua posição
- void Graph::removeEdge(edge e)
  - Remove aresta **e**

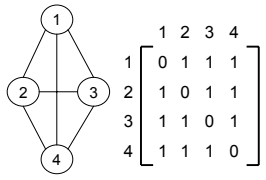
Representação

- Embora sejam possíveis diversas representações dos grafos, vamos estudar duas mais utilizadas comumente
  - matriz de adjacência e
  - lista de adjacências
- A escolha de determinada representação dependerá da aplicação que se tem em vista e das funções que se espera realizar no grafo

Matriz de Adjacências

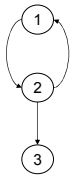
- Seja G=(V,E) um grafo com **n** vértices, n≥1
- A **matriz de adjacências** A de G é um arranjo bidimensional **n** x **n** com a propriedade de que
  - A[i,j] = 1 se a aresta (v_i,v_j) pertence a E(G)
  - A[i,j] = 0 caso contrário
- A matriz de adjacências para um grafo não-orientado é simétrica, pois a aresta (v_i,v_j) está em E(G), se a aresta (v_j,v_i) também está em E(G)
- A matriz de adjacências de um grafo orientado não é necessariamente simétrica
- O espaço necessário para representar um grafo usando a matriz de adjacências é de n^2 bits
  - Aproximadamente metade desse espaço pode ser poupado no caso dos grafos não-orientados, armazenando apenas o triângulo superior ou inferior da matriz

Matriz de Adjacências



G_1

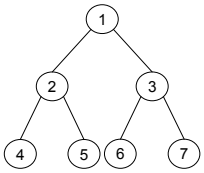
	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0



G_3

	1	2	3
1	0	1	0
2	1	0	1
3	0	0	0

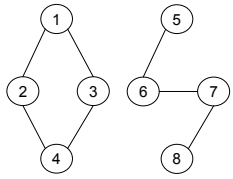
Matriz de Adjacências



G_2

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

Matriz de Adjacências



G_4

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

Matriz de Adjacências

- A partir da matriz de adjacências é possível determinar se existe uma aresta que liga quaisquer dois vértices v_i e v_j
- Para um grafo não-orientado, o grau de qualquer vértice v_i vem a ser a soma de sua linha:

$$\sum_{j=1}^{n} A[i,j]$$

- Para um grafo orientado, a soma da linha é o grau de saída ao passo que a soma da coluna vem a ser o valor grau de entrada

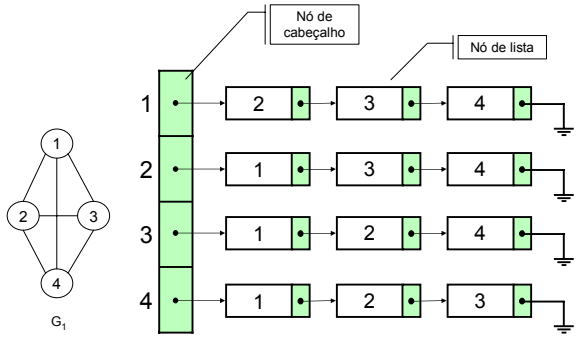
Matriz de Adjacências

- Suponha que queremos responder perguntas sobre grafos tais como:
  - Quantas arestas existem em G?
  - Será que G está interligado?
- Usando as matrizes de adjacências, todos os algoritmos vão exigir, pelo menos, $O(n^2)$ de tempo uma vez que (n^2-**n**) entradas da matriz (a diagonal principal possui somente zeros) têm de ser examinadas
- Se o grafo for esparso, isto é, quando a maioria dos termos na matriz de adjacências for zero, é possível responder as perguntas acima em tempo O(**n**+**e**), onde **e** é o número de arestas em G e **e** << $n^2/2$
- Esse aceleramento pode ser possibilitado mediante a utilização de listas ligadas, nas quais são representadas apenas as arestas existentes em G, o que nos leva à próxima representação para grafos

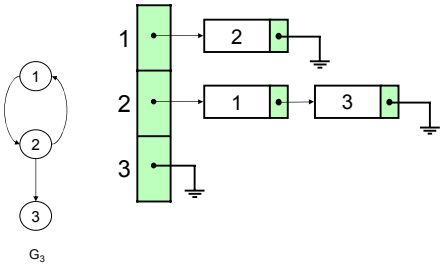
Lista de Adjacências

- Nesta representação as **n** linhas da matriz de adjacências são representadas como **n** listas encadeadas, ou seja, existe uma lista para cada vértice em G
- Os nós na lista **i** representam os vértices que são adjacentes ao vértice v_i
- Cada nó possui, pelo menos, dois campos
  - Um campo que contém o índice do vértice adjacente ao vértice v_i
  - Um campo de ligação com o próximo vértice adjacente ao vértice v_i

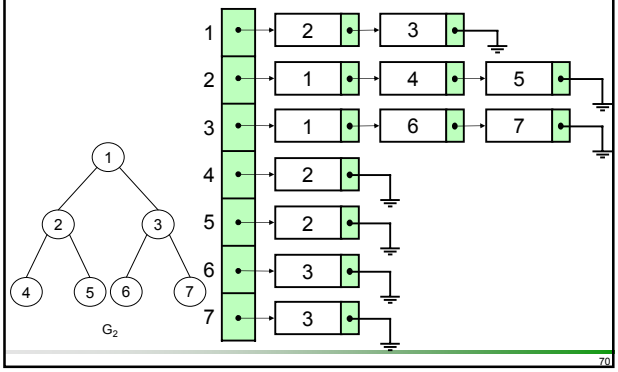
Lista de Adjacências



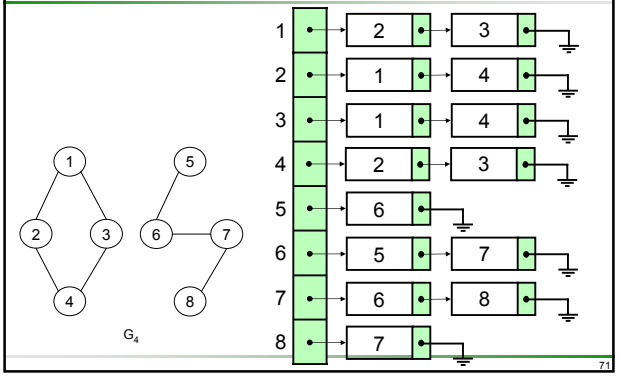
Lista de Adjacências



Lista de Adjacências



Lista de Adjacências



Lista de Adjacências

- Cada lista possui um nó de cabeçalho (início ou *head* da lista)
  - Os nós de cabeçalho são seqüenciais, permitindo fácil acesso aleatório à lista de adjacências de determinado vértice
- No caso de um grafo não-orientado com **n** vértices e **a** arestas, essa representação requer **n** nós de cabeçalho e 2***a** nós de lista, sendo que cada nó de lista possui 2 campos (vértice adjacente + ligação)
- Em termos de quantidade de bits de memória necessária, essa contagem deve ser multiplicada por $\log_2(\mathbf{n})$ para os nós de cabeçalho e por $\log2(\mathbf{n})+\log2(\mathbf{a})$ para os nós de lista, uma vez que são necessários $\log_2(\mathbf{x})$ bits para representar um número de valor **x**
- Em alguns casos os nós podem ser condensados seqüencialmente nas listas de adjacências eliminando-se os campos de ligação

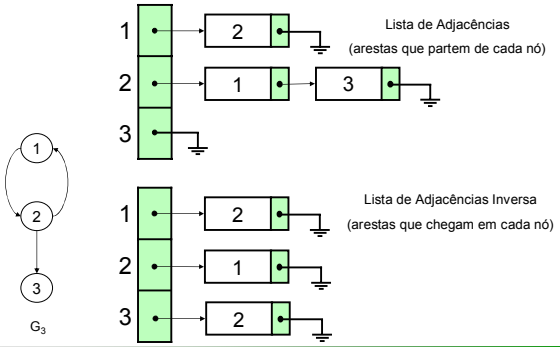
Lista de Adjacências

- O grau de qualquer vértice em um grafo não-orientado pode ser determinado simplesmente contando o número de nós na respectiva lista de adjacências
- Portanto, o número total de arestas pode ser determinado em tempo O(**n**+**e**)
- No caso de um dígrafo, o número total de arestas corresponde ao número de nós de lista
- O grau-de-saída de qualquer vértice pode ser determinado contando o número de nós na lista de adjacências
- O número total de arestas em G pode, portanto, ser determinado em O(**n**+**e**)

Lista de Adjacências

- Determinar o grau de entrada de um vértice é uma tarefa um pouco mais complexa, havendo necessidade de ter acesso repetidamente a todos os vértices adjacentes a outro vértice
- Assim, talvez seja conveniente o esforço de manter outro conjunto de listas, além das listas de adjacências
- Esse conjunto de listas, chamado de **listas de adjacência inversa**, conterá uma lista para cada vértice
  - Cada vértice conterá um nó para cada vértice adjacente ao vértice que representa
- Alternativamente, pode-se adotar Listas Cruzadas, estrutura também utilizada para a representação de matrizes esparsas

Lista de Adjacências Inversa



Representação

- Em algumas situações são atribuídos pesos às arestas de um grafo
- Esses pesos podem representar a distância de um vértice para outro ou o custo de se passar de um vértice para outro adjacente
- Neste caso as entradas da matriz de adjacências A[i,j] mantêm também essas informações
- No caso das listas de adjacências, essa informação sobre pesos pode ser conservada nos nós de listas, incluindo um campo adicional
- Um grafo com arestas dotadas de pesos é denominado **rede** ou **grafo ponderado**

Busca

- Partindo-se do nó de raiz de uma árvore binária, uma das coisas que se efetua de forma muito freqüente é atravessar a árvore e visitar os nós em alguma seqüência
  - Por exemplo, as três maneiras básicas de fazer isto são: pré-ordem, em-ordem e pós-ordem
- Um problema análogo ocorre em grafos: sendo G(V,E) um grafo não-orientado e um vértice **v** em V(G), estamos interessados em visitar todos os vértices em G que são alcançáveis a partir de **v** (isto é, todos os vértices interligados a **v**)

Busca em Grafos

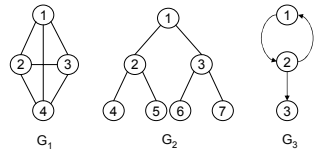
- Seja G um grafo conexo em que todos os seus vértices se encontram **desmarcados**
- Inicialmente, **marca-se** um vértice arbitrariamente escolhido
- Após isso, seleciona-se algum vértice **v** que esteja marcado e seja incidente a alguma aresta (**v**,**w**) ainda não selecionada
- A aresta (**v**,**w**) torna-se então **selecionada** e o vértice **w** é **marcado** (caso ainda não o seja)
- O processo termina quando todas as arestas de G tiverem sido selecionadas

Busca em Grafos

- Quando a aresta (**v**,**w**) é selecionada a partir do vértice marcado **v**, diz-se que (**v**,**w**) foi **visitada** (ou **explorada**) e o vértice **w** foi alcançado
- Um vértice torna-se **visitado** (ou **explorado**) quando todas as arestas incidentes a ele tiverem sido exploradas
- Assim, durante o processo de exploração de um vértice é possível que este venha a ser alcançado diversas vezes
- O vértice inicial é denominado **raiz** da busca

Busca em Grafos

```
procedure GeneralSearch
  escolher e marcar um vértice inicial (raiz da
  busca);
  while existe algum vértice v marcado e incidente a
  uma aresta (v,w) não explorada do
    escolher o vértice v;
    visitar a aresta (v,w);
    if w não está marcado then
      marcar w;
    endif
  endwhile
end GeneralSearch
```



G_1 G_2 G_3

Busca em Grafos

- Na busca geral, a escolha do próximo vértice e da aresta a ser visitada é arbitrária
- Nos critérios de busca em profundidade e busca em largura a escolha do próximo vértice torna-se única
  - Entretanto, a escolha do vértice inicial e aresta incidente permanece arbitrária

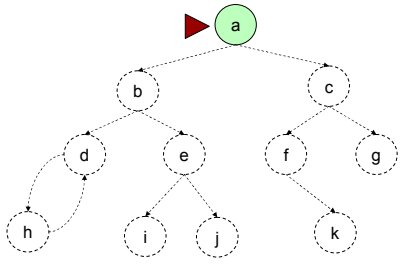
Busca em Grafos

- Busca em Profundidade (*depth first search*)
  - Uma busca é dita em profundidade quando o critério de escolha do vértice marcado (a partir do qual será realizada a próxima visita de aresta) obedecer ao seguinte: *"Dentre todos os vértices marcados e incidentes a alguma aresta ainda não visitada, escolher aquele **mais** recentemente alcançado na busca"*
- Busca em Largura (*breadth first search*)
  - Uma busca é dita em largura quando o critério de escolha do vértice marcado (a partir do qual será realizada a próxima visita de aresta) obedecer ao seguinte: *"Dentre todos os vértices marcados e incidentes a alguma aresta ainda não visitada, escolher aquele **menos** recentemente alcançado na busca"*

Busca em Profundidade

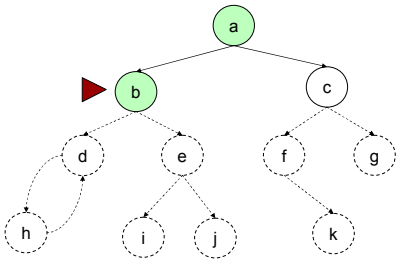
- A pesquisa em profundidade de um grafo se realiza da seguinte maneira:
  - Visita-se o vértice inicial **v**
  - Em seguida um vértice não visitado **w**, adjacente a **v**, é selecionado iniciando-se uma pesquisa em profundidade, a partir de **w**
  - Atingindo-se um vértice **u** tal que tenham sido visitados todos seus vértices adjacentes, voltamos para o último vértice visitado que tem vértice **w** não visitado adjacente ao mesmo e iniciamos uma pesquisa em profundidade a partir de **w**
- A busca termina quando nenhum vértice não visitado pode ser atingido de qualquer um daqueles que foram visitados

Busca em Profundidade



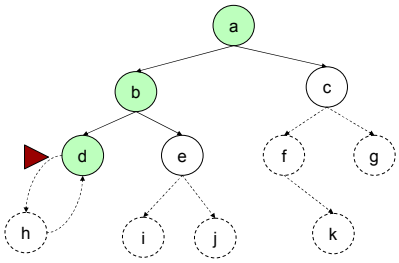
Inserir na frente, remover da frente: a

Busca em Profundidade



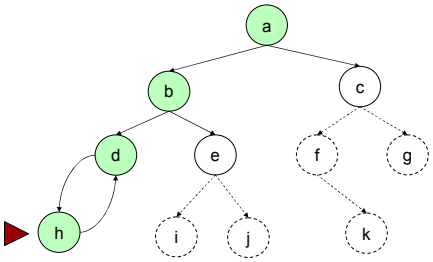
Inserir na frente, remover da frente: b, c

Busca em Profundidade



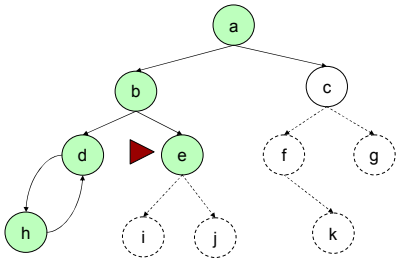
Inserir na frente, remover da frente: d, e, c

Busca em Profundidade



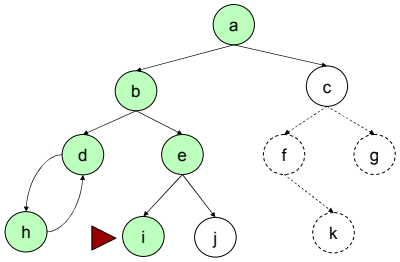
Inserir na frente, remover da frente: h, e, c

Busca em Profundidade



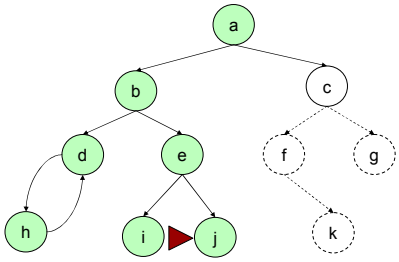
Inserir na frente, remover da frente: e, c

Busca em Profundidade



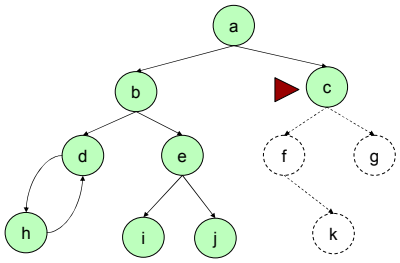
Inserir na frente, remover da frente: i, j, c

Busca em Profundidade



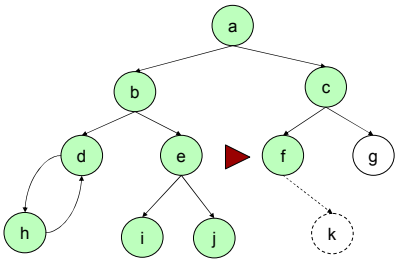
Inserir na frente, remover da frente: j, c

Busca em Profundidade



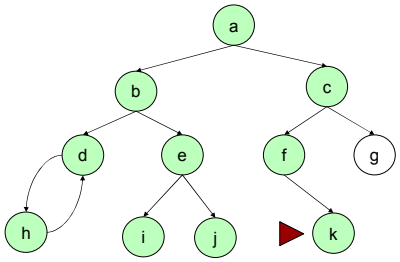
Inserir na frente, remover da frente: c

Busca em Profundidade



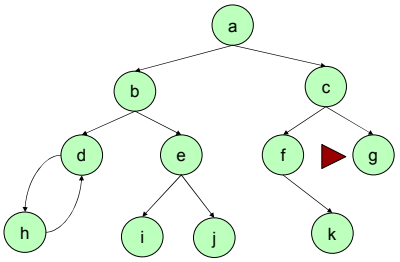
Inserir na frente, remover da frente: f, g

Busca em Profundidade



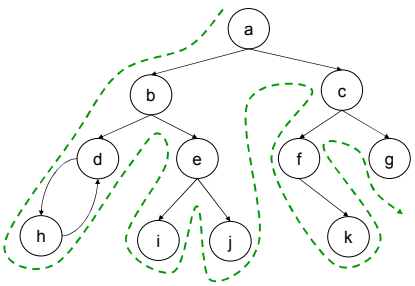
Inserir na frente, remover da frente: k, g

Busca em Profundidade



Inserir na frente, remover da frente: g

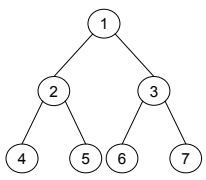
Busca em Profundidade



Ordem de visita partindo do vértice a: a,b,d,h,e,i,j,c,f,g,k

Exercício

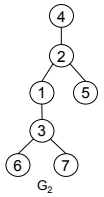
- Forneça a ordem em que são visitados os vértices do grafo ao lado usando busca em profundidade (caso um vértice tenha mais de um vértice adjacente a ele, visite primeiro os vértices de menor número), assumindo a raiz da busca como:
  - vértice 1
  - vértice 4



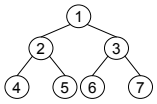
G_2

Solução

- Forneça a ordem em que são visitados os vértices do grafo ao lado usando busca em profundidade (caso um vértice tenha mais de um vértice adjacente a ele, visite primeiro os vértices de menor número), assumindo a raiz da busca como:
  - vértice 1
    - 1,2,4,5,3,6,7
  - vértice 4
    - 4,2,1,3,6,7,5



G_2



G_2

Algoritmo de Busca em Profundidade

```
// Dado um grafo G=(V,E) com n vértices e arranjo
  VISITED[1..n] zerado inicialmente, este algoritmo
  visita em profundidade todos os vértices
  alcançáveis a partir de v

procedure DepthFirstSearch(v)
  VISITED[v] ← true; // marque v como visitado
  for cada vértice w adjacente a v do
    if not VISITED[w] then
      DepthFirstSearch(w);
    endif
  next w
end DepthFirstSearch
```


Algoritmo de Busca em Profundidade

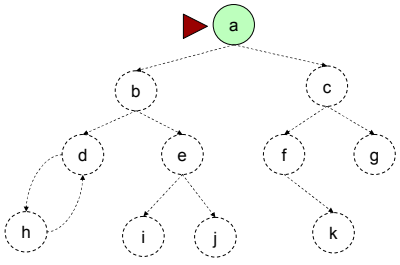
```
procedure DepthFirstSearch(v)
  VISITED[v] ← true; // marque v como visitado
  for cada vértice w adjacente a v do
    if not VISITED[w] then
      DepthFirstSearch(w)
    endif
  next w
end DepthFirstSearch

// procedimento que chama DepthFirstSearch
  for i ← 1 to n do
    VISITED[i] ← false;
  next i
  escolher vértice raiz da busca, v;
  DepthFirstSearch(v);
```


Algoritmo de Busca em Profundidade

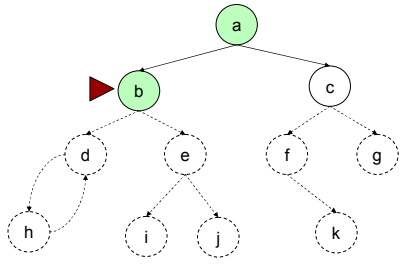
- Como pode ser observado, o algoritmo de busca em profundidade possui uma chamada recursiva no final
- Portanto o algoritmo recursivo pode ser substituído por uma versão iterativa por meio do uso de uma pilha
- Além disso, as arestas de retorno também podem ser visitadas estendendo o algoritmo fornecido
- A implementação dessas alterações é deixada como exercício

Busca em Largura



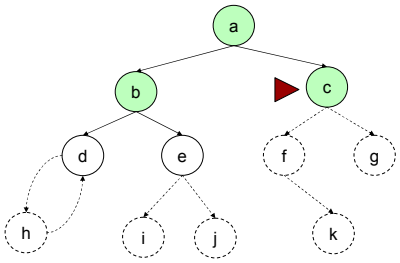
Inserir no final, remover da frente: a

Busca em Largura



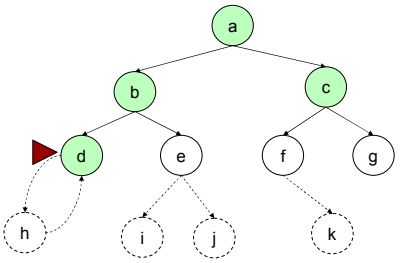
Inserir no final, remover da frente: b, c

Busca em Largura



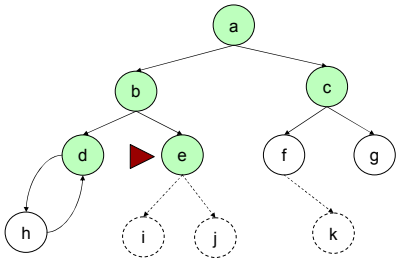
Inserir no final, remover da frente: c, d, e

Busca em Largura



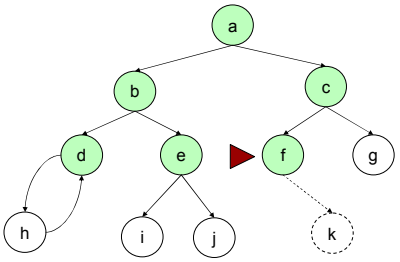
Inserir no final, remover da frente: d, e, f, g

Busca em Largura



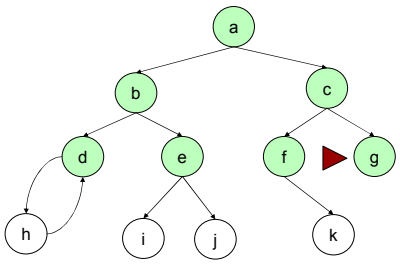
Inserir no final, remover da frente: e, f, g, h

Busca em Largura



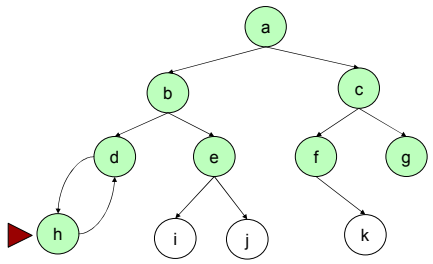
Inserir no final, remover da frente: f, g, h, i, j

Busca em Largura



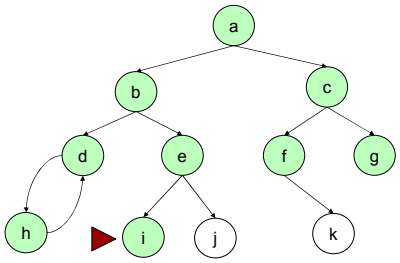
Inserir no final, remover da frente: g, h, i, j, k

Busca em Largura



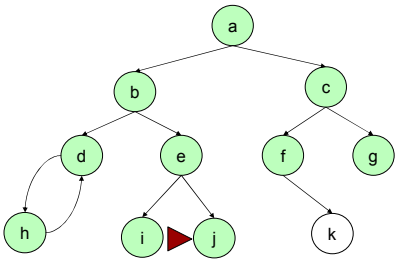
Inserir no final, remover da frente: h, i, j, k

Busca em Largura



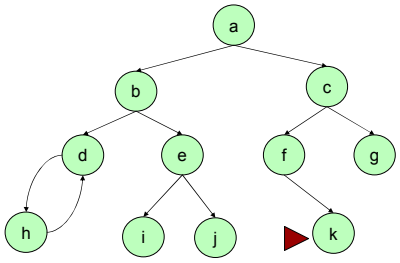
Inserir no final, remover da frente: i, j, k

Busca em Largura



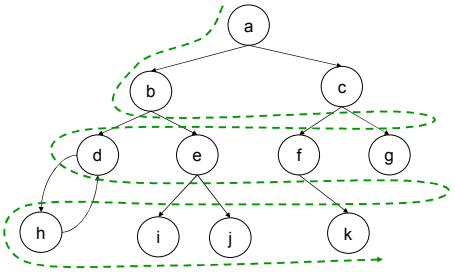
Inserir no final, remover da frente: j, k

Busca em Largura



Inserir no final, remover da frente: k

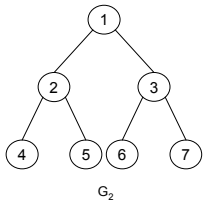
Busca em Largura



Ordem de visita partindo do vértice a: a,b,c,d,e,f,g,h,i,j,k

Exercício

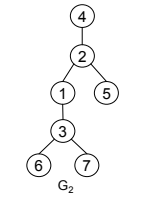
- Forneça a ordem em que são visitados os vértices do grafo ao lado usando busca em largura (caso um vértice tenha mais de um vértice adjacente a ele, visite primeiro os vértices de menor número), assumindo a raiz da busca como:
  - vértice 1
  - vértice 4



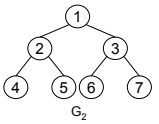
G_2

Solução

- Forneça a ordem em que são visitados os vértices do grafo ao lado usando busca em largura (caso um vértice tenha mais de um vértice adjacente a ele, visite primeiro os vértices de menor número), assumindo a raiz da busca como:
  - vértice 1
    - 1,2,3,4,5,6,7
  - vértice 4
    - 4,2,1,5,3,6,7



G_2



G_2

Algoritmo de Busca em Largura

- Para o algoritmo de busca em largura, uma fila será necessária
- Para tanto, assume-se que os métodos do ADT fila encontram-se disponíveis
  - Quando um vértice é adicionado na fila, usa-se a operação **Append** (inserir no final)
  - Quando um vértice é retirado da fila, usa-se a operação **Serve** (retirar do início)
  - O método **Empty** retorna **true** se a fila está vazia; **false** caso contrário
- Como na busca em profundidade, as arestas de retorno também podem ser visitadas estendendo o algoritmo, o que é deixado como exercício

Algoritmo de Busca em Largura

```
// Uma pesquisa em largura de G=(V,E) com n vértices é iniciada a
   partir do vértice v. Todos os vértices visitados são marcados
   como sendo VISITED[i]. O arranjo VISITED[1..n] é zerado
   inicialmente

procedure BreadthFirstSearch(v)
  VISITED[v] ← true; // marque v como visitado
  definir uma fila Q vazia
  Q.Append(v);
  while not Q.Empty() do
    Q.Serve(v); // retire v da fila Q
    for para cada vértice w adjacente a v do
      if not VISITED[w] then
         Q.Append(w); // adicione w à fila Q
         VISITED[w] ← true; // marque w como visitado
      endif
    next w
  endwhile
end BreadthFirstSearch
```


Algoritmo de Busca em Largura

```
procedure BreadthFirstSearch(v)
  VISITED[v] ← true; // marque v como visitado
  definir uma fila Q vazia
  Q.Append(v);
  while not Q.Empty() do
    Q.Serve(v); // retire v da fila Q
    for para cada vértice w adjacente a v do
      if not VISITED[w] then
         Q.Append(w); // adicione w à fila Q
         VISITED[w] ← true; // marque w como visitado
      endif
    next w
  endwhile
end BreadthFirstSearch

// procedimento que chama BreadthFirstSearch
  for i ← 1 to n do
    VISITED[i] ← false;
  next i
  escolher vértice raiz da busca, v;
  BreadthFirstSearch(v);
```


Componentes Conexos

- Sendo G um grafo não-orientado, pode-se determinar se está ou não interligado, por simplesmente executando DFS ou BFS para, em seguida, determinar se existe algum vértice não visitado
- O tempo necessário para se fazer isto é de $O(n^2)$, se forem utilizadas matrizes de adjacências e $O(e)$ se forem utilizadas listas de adjacências
- Um problema mais interessante é aquele em que se determinam todos os componentes conexo de um grafo, efetuando chamadas repetidas, a DFS(**v**) ou BFS(**v**), sendo **v** um vértice ainda não visitado
- Isso nos leva ao algoritmo **Comp**, que determina todos os componentes conexos de um grafo G
- O algoritmo faz uso de DFS, sendo possível substituí-lo por BFS, sem afetar o tempo de computação

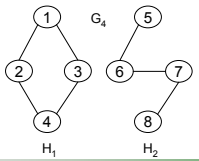
Componentes Conexos

```
// Determina os componentes conexos de G, que tem n ≥ 1
   vértices; VISITED agora é um arranjo local.

procedure COMP()
   // marcar todos os vértices como não visitados
  for i ← 1 to n do
    VISITED[i] ← false;
  next i
  for i ← 1 to n do
    if not VISITED[i] then
     DepthFirstSearch(i); // procure um componente
     dê saída em todos os vértices recém visitados,
     junto com todas as arestas que incidem nos mesmos;
    endif
  next i
end COMP
```


Componentes Conexos

```
procedure COMP()
   // marcar todos os vértices como não visitados
  for i ← 1 to n do
    VISITED[i] ← false;
  next i
  for i ← 1 to n do
    if not VISITED[i] then
     DepthFirstSearch(i); // procure um componente
     dê saída em todos os vértices recém visitados,
     junto com todas as arestas que incidem nos mesmos;
    endif
  next i
end COMP
```



Componentes Conexos

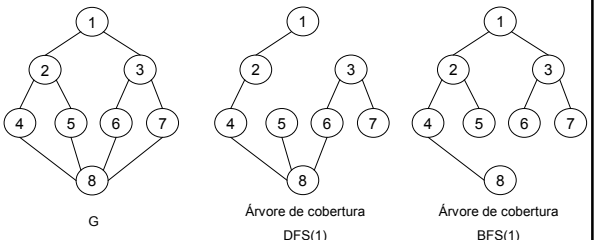
- Se G for representado por lista de adjacências, o tempo total consumido por DFS é de O(**e**)
  - A saída pode ser completada em tempo O(**e**), contando que DFS mantenha uma relação de todos os vértices recém visitados
- Uma vez que os laços **for** consomem O(**n**), o tempo total necessário para gerar todos os componentes conexos é O(**n**+**e**)
- Em virtude da definição de um componente conexo, existe um caminho entre cada par de vértices no componente e não existe caminho em G do vértice **v** para **w**, se **v** e **w** estiverem em dois componentes diferentes
- Assim, se A é matriz de adjacências de um grafo não-orientado (A é simétrica), seu fechamento transitivo A^+ pode ser determinado em tempo O(n^2) por se determinarem primeiro os componentes conexos (que será visto mais adiante)

Árvore de Cobertura

- Sendo um grafo G conexo, uma pesquisa em profundidade ou em largura, começando em qualquer vértice, visita todos os vértices em G
- Nesse caso, as arestas de G são divididas em dois conjuntos
  - T, para **arestas de árvore** e
  - B, para **arestas de retorno**, ou **frondes**
- sendo T o conjunto de arestas utilizadas ou atravessadas durante a busca e B o conjunto de arestas remanescentes
- O conjunto T pode ser determinado por se inserir o comando seguinte nas cláusulas **then** de DFS e BFS
  - $T \leftarrow T \cup \{(v, w)\}$
- As arestas em T formam uma árvore que inclui todos os vértices de G, ou seja, uma árvore de cobertura de G

Árvore de Cobertura

- Árvores de cobertura resultantes de busca em profundidade e em largura de G, começando pelo vértice 1



Árvore de Cobertura

- Se os nós de G representarem, por exemplo, cidades e as arestas representarem possíveis meios de comunicação entre duas cidades então o número mínimo necessário de elos para interligar **n** cidades é de (**n**-1) arestas
  - As árvores estendidas de G representarão todas as escolhas possíveis
- Todavia, em situações práticas são designadas às arestas determinadas ponderações
  - Essas ponderações podem representar o custo da construção, o comprimento de ligação etc
- Tendo um grafo ponderado, seria então desejável selecionar para fins de construção um conjunto de elos de comunicações que interligariam todas as cidades, tendo custo total mínimo ou extensão total mínima
  - Em qualquer dos casos os elos selecionados terão que formar uma árvore (admitindo que sejam positivas todas as ponderações)

Árvore de Cobertura

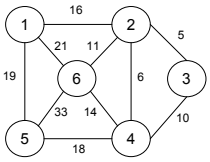
- Assim sendo, a seleção dos elos contém ciclos
- A remoção de qualquer um dos elos desse ciclo resultará numa seleção de elos de custo mais baixo, para interligar todas as cidades
- O objetivo é encontrar uma árvore de cobertura de G com custo mínimo
- O custo de uma árvore de cobertura será a soma dos custos das arestas dessa árvore

Árvore de Cobertura de Custo Mínimo

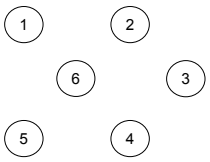
- Uma abordagem para determinação de uma árvore estendida de custo mínimo de um grafo foi proposta por Kruskal
- Nessa abordagem é construída, aresta por aresta, uma árvore estendida de custo mínimo T
- São consideradas as arestas para inclusão em T, por **ordem crescente** de seus custos
- Uma aresta é incluída em T, caso não forme ciclo com as arestas já existentes em T
- Uma vez que G está interligado e possui **n** > 0 vértices, serão selecionadas exatamente **n**-1 arestas para inclusão em T

Árvore de Cobertura de Custo Mínimo

- Quais as duas primeiras arestas a serem incluídas em T, lembrando que as arestas são selecionadas por **ordem crescente** de seus custos?



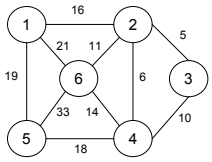
G



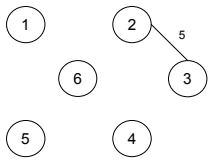
Componentes conexos: {1}, {2}, {3}, {4}, {5}, {6}

Árvore de Cobertura de Custo Mínimo

- As primeiras duas arestas (2,3) e (2,4) estão incluídas em T



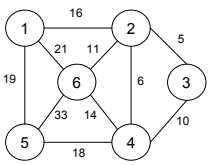
G



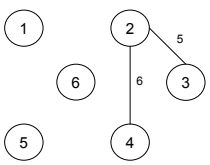
Componentes conexos: {1}, {2,3}, {4}, {5}, {6}

Árvore de Cobertura de Custo Mínimo

- As primeiras duas arestas (2,3) e (2,4) estão incluídas em T



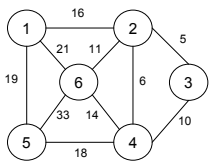
G



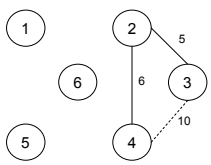
Componentes conexos: {1}, {2,3,4}, {5}, {6}

Árvore de Cobertura de Custo Mínimo

- A próxima aresta a ser considerada é (4,3), que liga entre si dois vértices já interligados em T e por isso é rejeitada



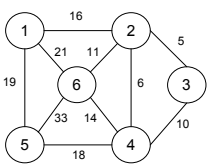
G



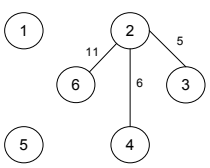
Componentes conexos: {1}, {2,3,4}, {5}, {6}

Árvore de Cobertura de Custo Mínimo

- A aresta (2,6) é selecionada...



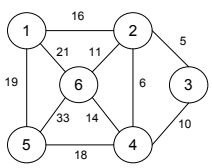
G



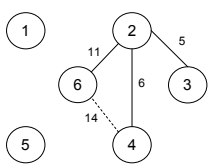
Componentes conexos: {1}, {2,3,4,6}, {5}

Árvore de Cobertura de Custo Mínimo

- A aresta (2,6) é selecionada, ao passo que (4,6) é rejeitada, uma vez que os vértices 4 e 6 já estão ligados em T, sendo que a inclusão de (4,6) resultaria num ciclo



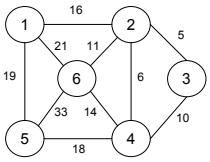
G



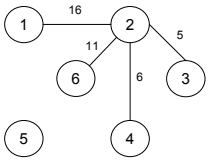
Componentes conexos: {1}, {2,3,4,6}, {5}

Árvore de Cobertura de Custo Mínimo

- Finalmente, são incluídas as arestas (1,2)...



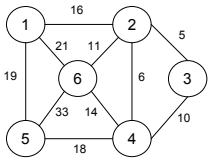
G



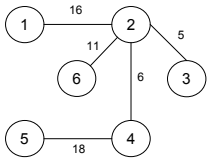
Componentes conexos: {1,2,3,4,6}, {5}

Árvore de Cobertura de Custo Mínimo

- Finalmente, são incluídas as arestas (1,2) e (4,5)



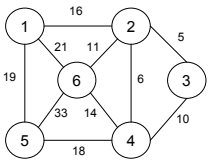
G



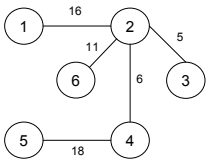
Componentes conexos: {1,2,3,4,5,6}

Árvore de Cobertura de Custo Mínimo

- Nesse ponto, T possui **n**-1 arestas, sendo uma árvore que abrange **n** vértices
- A árvore de cobertura obtida tem custo de 56



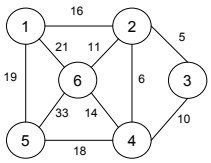
G



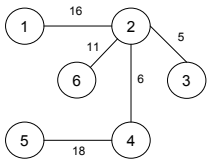
Árvore de cobertura de custo mínimo de G

Árvore de Cobertura de Custo Mínimo

- A seqüência de arestas consideradas para serem incluídas na árvore de cobertura de custo mínimo foi:
  - (2,3), (2,4), (4,3), (2,6), (4,6), (1,2) e (4,5)
  - Isto corresponde à seqüência de custos 5, 6, 10, 11, 14, 16 e 18



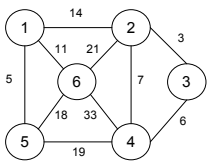
G



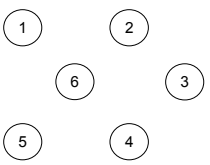
Árvore de cobertura de custo mínimo de G

Exercício

- Encontre a árvore de cobertura de custo mínimo do grafo G e o custo correspondente

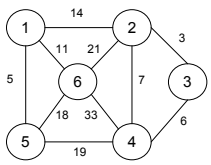


G

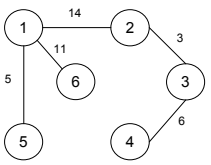


Solução

- T = {(2,3), (1,5), (3,4), (1,6), (1,2)}
- Custo(T) = 39



G



Árvore de cobertura de custo mínimo de G

Algoritmo de Kruskal

```
function Kruskal
 1  T ← ∅;
 2  while T contém menos de n-1 arestas and E não é vazio do
 3    escolha uma aresta (v,w) de E, de menor custo;
 4    E ← E - {(v,w)};  // remova (v,w) de E
 5    if (v,w) não cria um ciclo em T then
 6      T ← T ∪ {(v, w)};  // adicione (v,w) a T
 7    endif
 8  endwhile
 9  if T contém menos de n-1 arestas then
10    T ← ∅; // nenhuma árvore de cobertura encontrada
11  endif
12  return T;
end Kruskal
```


Algoritmo de Kruskal

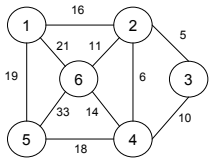
- Inicialmente, E é o conjunto de todas as arestas em G
- As únicas funções realizadas com este conjunto são:
  - (i) determinar uma aresta com custo mínimo (linha 3) e
  - (ii) remover essa aresta (linha 4)
- Ambas funções podem ser realizadas com eficiência, contanto que as arestas em E sejam mantidas como lista seqüencial ordenada por custos
- Na realidade, não é essencial classificar todas as arestas, contanto que a próxima aresta para a linha 3 possa ser determinada com facilidade
- A ordenação pelo método *heapsort* serve de maneira ideal para isto e permite que a próxima aresta seja determinada e seja removida em tempo O(**e***log(**e**))
  - A construção do *heap* em si requer tempo O(**e**)
- O tempo de computação do algoritmo é determinado pelas linhas 3 e 4 que, no pior caso, é de O(**e***log(**e**))

Algoritmo de Kruskal

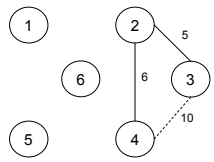
- Para se realizar eficientemente os passos 5 e 6, os vértices em G devem ser agrupados de tal maneira que se possa facilmente determinar se os vértices **v** e **w** já estão interligados pela seleção anterior de arestas
  - Nesse caso, a aresta **(v,w)** deve ser descartada
- Caso contrário, **(v,w)** deve ser incluída em T
- Um agrupamento possível consistiria em se colocarem todos os vértices do mesmo componente conexo de T num conjunto (todos os componentes conexos de T serão também árvores)
- Assim, dois vértices **v** e **w**, estarão interligados em T se estiverem no mesmo conjunto

Algoritmo de Kruskal

- No exemplo, onde se considera a aresta (4,3), os conjuntos seriam {1}, {2,3,4}, {5}, {6}
  - Os vértices 4 e 3 já estão no mesmo conjunto, de forma que a aresta (4,3) é rejeitada

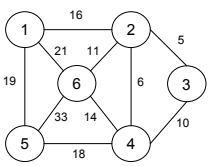


G

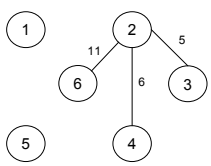


Algoritmo de Kruskal

- A próxima aresta a ser considerada é (2,6), os conjuntos ainda são {1}, {2,3,4}, {5}, {6}
  - Como os vértices 2 e 6 estão em conjuntos diferentes, a aresta é aceita interligando os dois componentes {2,3,4} e {6}
  - Esses dois componentes devem ser unidos para se obter o conjunto que representa o novo componente {2,3,4,6}



G



Caminhos Mais Curtos

- Os grafos podem ser utilizados para representar a estrutura rodoviária de um estado ou de um país, com os vértices representando cidades e as arestas representando trechos de rodovia
- As arestas podem então ter ponderações podem ser a distância entre as duas cidades interligadas pela aresta, ou o tempo médio necessário para percorrer a referida seção da rodovia ou mesmo o custo de combustível
- Um motorista que queira ir da cidade *A* para a cidade *B* estaria interessado em ter as respostas para:
  - Existe um caminho que vai de *A* para *B?*
  - Havendo mais de um caminho de *A* para *B,* qual o caminho mais curto?

Caminhos Mais Curtos

- Os grafos podem ser utilizados para representar a estrutura rodoviária de um estado ou de um país, com os vértices representando cidades e as arestas representando trechos de rodovia
- As arestas podem então ter ponderações podem ser a distância entre as duas cidades interligadas pela aresta, ou o tempo médio necessário para percorrer a referida seção da rodovia ou mesmo o custo de combustível
- Um motorista que queira ir da cidade *A* para a cidade *B* estaria interessado em ter as respostas para:
  - Existe um caminho que vai de *A* para *B?*
  - Havendo mais de um caminho de *A* para *B,* qual o caminho mais curto?

Caminhos Mais Curtos

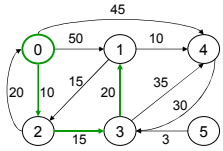
- O **comprimento** de um **caminho** agora é definido como a soma das ponderações das arestas do caminho ao invés do número de arestas
- O vértice inicial do caminho será denominado **origem** e o último vértice **destino**
- Os grafos possivelmente serão dígrafos para levar em conta as ruas de mão única
- A não ser que se indique ao contrário, admitiremos que todas as ponderações são positivas

Origem Única de Todos os Destinos

- Sejam
  - G(V,E) um grafo dirigido
  - **w(e)** uma função de ponderação para as arestas de G
  - **v_o** o vértice de origem
- O problema consiste em se determinar quais os caminhos mais curtos de **v_o** para todos os demais vértices de G
- Admite-se que todas as ponderações sejam positivas

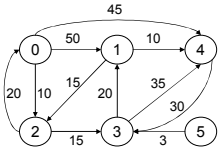
Origem Única de Todos os Destinos

- No exemplo, os números aplicados às arestas são as ponderações
- Sendo **v_o**=0 o vértice de origem, o caminho mais curto de 0 para 1 é 0,2,3,1
  - O comprimento desse caminho é 10+15+20 = 45



Origem Única de Todos os Destinos

- No exemplo, os números aplicados às arestas são as ponderações
- Sendo **v_o**=0 o vértice de origem, o caminho mais curto de 0 para 1 é 0,2,3,1
  - O comprimento desse caminho é 10+15+20 = 45
- Ainda que esse caminho apresente três arestas, ele é mais curto do que o caminho 0,1 cujo comprimento é 50
- Não existe caminho de 0 para 5



Caminhos mais curtos de v_o para todos os destinos

Caminho	Comprimento
0,2	10
0,2,3	25
0,2,3,1	45
0,4	45

Origem Única de Todos os Destinos

- Os caminhos foram relacionados por ordem crescente de comprimento
- Se elaborarmos um algoritmo que gere os caminhos mais curtos, nessa ordem, podemos então chegar a diversas observações
- Seja S o conjunto de vértices (inclusive **v_o**) para as quais já foram encontrados os caminhos mais curtos
- Para o caso de **w** $\notin$ S, seja DIST[**w**] o comprimento do caminho mais curto, partindo de **v_o** e passando unicamente por aqueles vértices que estão em S e que terminem em **w**

Origem Única de Todos os Destinos

- Observamos que
  - (i) Se o próximo caminho mais curto for aquele que vai para o vértice **u**, o caminho começa em v_o, termina em **u** e passa unicamente através daqueles vértices de S
    - Para comprovar isto temos que mostrar que todos os vértices intermediários no caminho mais curto para **u** estão em S
    - Admitamos que exista um vértice **w** nesse caminho que não se encontre em S
    - Nesse caso, o caminho de v_o para **u** contém também um caminho de v_o para **w** que é de comprimento inferior ao caminho de v_o para **u**
    - Supondo que os caminhos mais curtos estão sendo gerados em ordem crescente de comprimento, o caminho mais curto v_o para **w** já deve ter sido gerado
    - Assim sendo, não pode haver vértice intermediário que não esteja em S
  - (ii) O destino do próximo caminho gerado deve ser aquele vértice **u** que tem a distância mínima, DIST[**u**] entre todos os vértices que não estão em S
    - Isto decorre da definição de DIST e da observação (i)
    - Caso existam diversos vértices que não estejam em S, mas que tenham a mesma DIST, qualquer um deles poderá ser selecionado

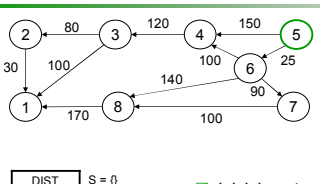
Origem Única de Todos os Destinos

- (iii) Tendo selecionado um vértice **u**, como em (ii) e tendo gerado o caminho mais curto de v_o para **u**, o vértice **u** passa a se tornar integrante de S
  - Nesse ponto, poderá diminuir o comprimento dos caminhos mais curtos começando em v_o e que passam através dos vértices em S, que terminam em um vértice **w** que não está em S
  - Em outras palavras, o valor de DIST[**w**] poderá sofrer alteração
  - Caso não mude, isto será devido a um caminho mais curto que começa em v_o e que vai a **u** e depois a **w**
  - Os vértices intermediários no caminho v_o para **u** e no caminho de **u** para **w** devem todos estar em S
  - Além do mais, o caminho de v_o para **u** deve ser o caminho mais curto (caso contrário, DIST[**w**] não foi definida corretamente)
  - O caminho de **u** para **w** pode ser escolhido para não conter quaisquer vértices intermediários
  - Podemos, portanto, concluir que se DIST[**w**] vai mudar (isto é, diminuir), em virtude de um caminho de v_o para **u** para **w**, onde o caminho de v_o para **u** é o caminho mais curto, sendo que o caminho de **u** para **w** é a aresta **(u,w)**
  - O comprimento desse caminho será DIST[**u**] + custo da aresta **(u,w)**

Origem Única de Todos os Destinos

- O algoritmo para encontrar o caminho mais curto foi enunciado pela primeira vez por Dijkstra e recorre às observações precedentes a fim de determinar o custo dos caminhos mais curtos de v_o para todos os demais vértices em G
- Admite-se que os **n** vértices de G são numerados de 1 até **n**
- O conjunto S é mantido como um arranjo de bits com S[**i**]=**false** se o vértice **i** não esteja em S e sendo S[**i**]=**true** caso contrário
- Admite-se ainda que o grafo está representado pela sua matriz de adjacências de custos, sendo COST[**i**,**j**] a ponderação da aresta (**i**,**j**)
- COST[**i**,**j**] conterá algum número grande +∞, caso a aresta (**i**,**j**) não esteja em E(G)
- Para **i=j**, COST[**i**,**j**] pode ser qualquer número não-negativo sem afetar o resultado do algoritmo
- A geração real dos caminhos é uma extensão de menor importância do próprio algoritmo, sendo deixada como exercício

Algoritmo de Dijkstra



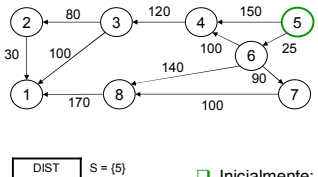
	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4			120	0				
5				**150**	**0**	**25**		
6				100		0	90	140
7							0	100
8	170							0

DIST	
1	+∞
2	+∞
3	+∞
4	150
5	0
6	25
7	+∞
8	+∞

S = {}

- Inicialmente:
  - v_o = 5 (origem)
  - S é vazio
  - DIST[i] consiste em COST[v_o,i]

Algoritmo de Dijkstra



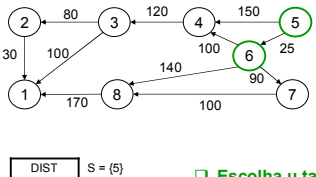
	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4			120	0				
5				**150**	**0**	**25**		
6				100		0	90	140
7							0	100
8	170							0

DIST	
1	+∞
2	+∞
3	+∞
4	150
5	0
6	25
7	+∞
8	+∞

S = {5}

- Inicialmente:
  - v_o = 5 (origem)
  - S ← ∅
  - DIST[i] consiste em COST[v_o,i]
- Coloque v_o em S
  - S ← S ∪ {v_o}
  - DIST[v_o] ← 0

Algoritmo de Dijkstra



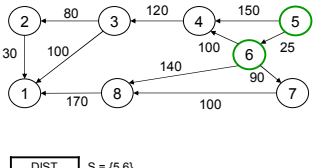
	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4			120	0				
5				150	0	25		
6				**100**		**0**	**90**	**140**
7							0	100
8	170							0

DIST	
1	+∞
2	+∞
3	+∞
4	150
5	0
6	**25**
7	+∞
8	+∞

S = {5}

- **Escolha u tal que**
  - **DIST[u] = mín{DIST[w]} para todos w ∉ S**
    - **u = 6**
- Coloque u em S
  - S ← S ∪ {u}
- Para todos w ∉ S
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra

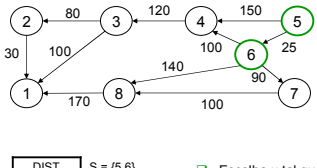


	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4			120	0				
5				150	0	25		
6				**100**		**0**	**90**	**140**
7							0	100
8	170							0

DIST	S = {5,6}
1	+∞
2	+∞
3	+∞
4	150
5	0
6	25
7	+∞
8	+∞

- Escolha u tal que
  - DIST[u] = mín{DIST[w]} para todos w ∉ S
    - u = 6
- **Coloque u em S**
  - **S ← S ∪ {u}**
- Para todos w ∉ S
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra

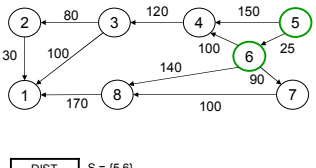


	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4			120	0				
5				150	0	25		
6				**100**		0	**90**	**140**
7							0	100
8	170							0

DIST	S = {5,6}
1	+∞
2	+∞
3	+∞
4	150
5	0
6	25
7	+∞
8	+∞

- Escolha u tal que
  - DIST[u] = mín{DIST[w]} para todos w ∉ S
    - u = 6
- Coloque u em S
  - S ← S ∪ {u}
- **Para todos w ∉ S**
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}
    - DIST[4] ← mín{DIST[4], DIST[6]+COST[6,4]}
    - DIST[4] ← mín{150, 25+100}
    - DIST[4] ← 125
    - Analogamente para demais w ∉ S

Algoritmo de Dijkstra

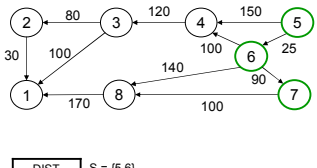


	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4			120	0				
5				150	0	25		
6				**100**		0	90	140
7							0	100
8	170							0

DIST	S = {5,6}
1	+∞
2	+∞
3	+∞
4	125
5	0
6	25
7	115
8	165

- Escolha u tal que
  - DIST[u] = mín{DIST[w]} para todos w ∉ S
    - u = 6
- Coloque u em S
  - S ← S ∪ {u}
- Para todos w ∉ S
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra

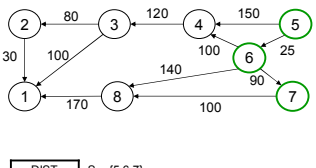


	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4			120	0				
5				150	0	25		
6				100		0	90	140
7							**0**	**100**
8	170							0

DIST	S = {5,6}
1	+∞
2	+∞
3	+∞
4	125
5	0
6	25
7	**115**
8	165

- **Escolha u tal que**
  - **DIST[u] = mín{DIST[w]} para todos w ∉ S**
    - **u = 7**
- Coloque u em S
  - S ← S ∪ {u}
- Para todos w ∉ S
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra

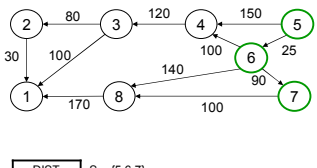


	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4			120	0				
5				150	0	25		
6				100		0	90	140
7							**0**	**100**
8	170							0

DIST	S = {5,6,7}
1	+∞
2	+∞
3	+∞
4	125
5	0
6	25
7	115
8	165

- Escolha u tal que
  - DIST[u] = mín{DIST[w]} para todos w ∉ S
    - u = 7
- **Coloque u em S**
  - **S ← S ∪ {u}**
- Para todos w ∉ S
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra

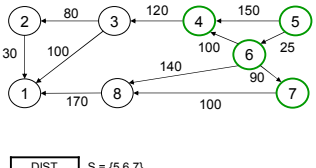


	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4			120	0				
5				150	0	25		
6				100		0	90	140
7							**0**	**100**
8	170							0

DIST	S = {5,6,7}
1	+∞
2	+∞
3	+∞
4	125
5	0
6	25
7	115
8	165

- Escolha u tal que
  - DIST[u] = mín{DIST[w]} para todos w ∉ S
    - u = 7
- Coloque u em S
  - S ← S ∪ {u}
- **Para todos w ∉ S**
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra



	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4				120	0			
5					150	0	25	
6					100		0	90	140
7								0	100
8	170								0

DIST	S = {5,6,7}
1	+∞
2	+∞
3	+∞
4	125
5	0
6	25
7	115
8	165

- **Escolha u tal que**
  - **DIST[u] = mín{DIST[w]} para todos w ∉ S**
    - **u = 4**
- Coloque u em S
  - S ← S ∪ {u}
- Para todos w ∉ S
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

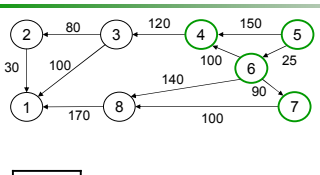
Algoritmo de Dijkstra

	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4				120	0			
5					150	0	25	
6					100		0	90	140
7								0	100
8	170								0

DIST	S = {5,6,7,4}
1	+∞
2	+∞
3	+∞
4	125
5	0
6	25
7	115
8	165

- Escolha u tal que
  - DIST[u] = mín{DIST[w]} para todos w ∉ S
    - u = 4
- **Coloque u em S**
  - **S ← S ∪ {u}**
- Para todos w ∉ S
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra



	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4				120	0			
5					150	0	25	
6					100		0	90	140
7								0	100
8	170								0

DIST	S = {5,6,7,4}
1	+∞
2	+∞
3	245
4	125
5	0
6	25
7	115
8	165

- Escolha u tal que
  - DIST[u] = mín{DIST[w]} para todos w ∉ S
    - u = 4
- Coloque u em S
  - S ← S ∪ {u}
- **Para todos w ∉ S**
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

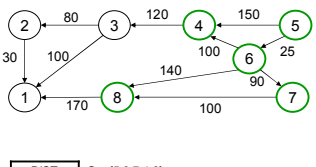
Algoritmo de Dijkstra

	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4				120	0			
5					150	0	25	
6					100		0	90	140
7								0	100
8	170								0

DIST	S = {5,6,7,4}
1	+∞
2	+∞
3	245
4	125
5	0
6	25
7	115
8	**165**

- **Escolha u tal que**
  - **DIST[u] = mín{DIST[w]} para todos w ∉ S**
    - **u = 8**
- Coloque u em S
  - S ← S ∪ {u}
- Para todos w ∉ S
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra



	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4				120	0			
5					150	0	25	
6					100		0	90	140
7								0	100
8	**170**								**0**

DIST	S = {5,6,7,4,8}
1	+∞
2	+∞
3	245
4	125
5	0
6	25
7	115
8	165

- Escolha u tal que
  - DIST[u] = mín{DIST[w]} para todos w ∉ S
    - u = 8
- **Coloque u em S**
  - **S ← S ∪ {u}**
- Para todos w ∉ S
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

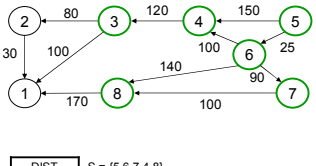
Algoritmo de Dijkstra

	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	100	80	0					
4				120	0			
5					150	0	25	
6					100		0	90	140
7								0	100
8	**170**								**0**

DIST	S = {5,6,7,4,8}
1	335
2	+∞
3	245
4	125
5	0
6	25
7	115
8	165

- Escolha u tal que
  - DIST[u] = mín{DIST[w]} para todos w ∉ S
    - u = 8
- Coloque u em S
  - S ← S ∪ {u}
- **Para todos w ∉ S**
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra



	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	**100**	**80**	**0**					
4			120	0				
5				150	0	25		
6				100		0	90	140
7							0	100
8	170							0

DIST	
1	335
2	+∞
3	**245**
4	125
5	0
6	25
7	115
8	165

S = {5,6,7,4,8}

- **Escolha u tal que**
  - **DIST[u] = mín{DIST[w]} para todos w ∉ S**
    - **u = 3**
- Coloque u em S
  - S ← S ∪ {u}
- Para todos w ∉ S
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra

	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	**100**	**80**	**0**					
4			120	0				
5				150	0	25		
6				100		0	90	140
7							0	100
8	170							0

DIST	
1	335
2	+∞
3	245
4	125
5	0
6	25
7	115
8	165

S = {5,6,7,4,8,3}

- Escolha u tal que
  - DIST[u] = mín{DIST[w]} para todos w ∉ S
    - u = 3
- **Coloque u em S**
  - **S ← S ∪ {u}**
- Para todos w ∉ S
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra

	1	2	3	4	5	6	7	8
1	0							
2	30	0						
3	**100**	**80**	**0**					
4			120	0				
5				150	0	25		
6				100		0	90	140
7							0	100
8	170							0

DIST	
1	335
2	325
3	245
4	125
5	0
6	25
7	115
8	165

S = {5,6,7,4,8,3}

- Escolha u tal que
  - DIST[u] = mín{DIST[w]} para todos w ∉ S
    - u = 3
- Coloque u em S
  - S ← S ∪ {u}
- **Para todos w ∉ S**
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra

	1	2	3	4	5	6	7	8
1	0							
2	**30**	**0**						
3	100	80	0					
4			120	0				
5				150	0	25		
6				100		0	90	140
7							0	100
8	170							0

DIST	
1	335
2	**325**
3	245
4	125
5	0
6	25
7	115
8	165

S = {5,6,7,4,8,3}

- **Escolha u tal que**
  - **DIST[u] = mín{DIST[w]} para todos w ∉ S**
    - **u = 2**
- Coloque u em S
  - S ← S ∪ {u}
- Para todos w ∉ S
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra

	1	2	3	4	5	6	7	8
1	0							
2	**30**	**0**						
3	100	80	0					
4			120	0				
5				150	0	25		
6				100		0	90	140
7							0	100
8	170							0

DIST	
1	335
2	325
3	245
4	125
5	0
6	25
7	115
8	165

S = {5,6,7,4,8,3,2}

- Escolha u tal que
  - DIST[u] = mín{DIST[w]} para todos w ∉ S
    - u = 2
- **Coloque u em S**
  - **S ← S ∪ {u}**
- Para todos w ∉ S
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra

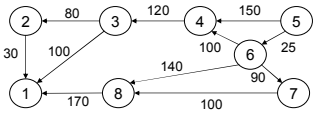
	1	2	3	4	5	6	7	8
1	0							
2	**30**	**0**						
3	100	80	0					
4			120	0				
5				150	0	25		
6				100		0	90	140
7							0	100
8	170							0

DIST	
1	335
2	325
3	245
4	125
5	0
6	25
7	115
8	165

S = {5,6,7,4,8,3,2}

- Escolha u tal que
  - DIST[u] = mín{DIST[w]} para todos w ∉ S
    - u = 2
- Coloque u em S
  - S ← S ∪ {u}
- **Para todos w ∉ S**
  - DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]}

Algoritmo de Dijkstra



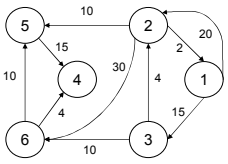
Iteração	S	Vértice Selecionado	DIST							
			1	2	3	4	5	6	7	8
inicial		5	+∞	+∞	+∞	150	0	25	+∞	+∞
1	5	6	+∞	+∞	+∞	125	0	25	115	165
2	5,6	7	+∞	+∞	+∞	125	0	25	115	165
3	5,6,7	4	+∞	+∞	245	125	0	25	115	165
4	5,6,7,4	8	335	+∞	245	125	0	25	115	165
5	5,6,7,4,8	3	335	325	245	125	0	25	115	165
6	5,6,7,4,8,3	2	335	325	245	125	0	25	115	165
final	5,6,7,4,8,3,2		335	325	245	125	0	25	115	165

Algoritmo de Dijkstra

```
// DIST[j], 1 ≤ j ≤ n é posicionado para o comprimento do caminho
   mais curto do vértice v até o vértice j num digrafo G com n
   vértices. G é representado por sua matriz de adjacências de
   custos COST[1..n, 1..n]. S é um arranjo declarado como S[1..n]
procedure ShortestPath(v, COST, DIST, n)
 1 for i ← 1 to n do  // inicialize conjunto S como vazio
 2   S[i] ← false; DIST[i] ← COST[v,i];
 3 next j
 4 S[v] ← true; DIST[v] ← 0; num ← 2 // coloque v no conjunto S
 5 while num < n do // determine n-1 caminhos a partir de v
 6   escolha u: DIST[u] = min{DIST[w]} para todos w com S[w]=false
 7   S[u] ← true; num ← num + 1; // coloque u no conjunto S
 8   for todos w com S[w]=false do  // atualize as distâncias
 9     DIST[w] ← mín{DIST[w], DIST[u]+COST[u,w]};
10  next w
11 endwhile
end ShortestPath
```

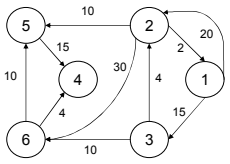

Exercício

- Utilize o algoritmo **ShortestPath** para obter, em seqüência crescente, os comprimentos dos caminhos mais curtos entre o vértice 1 e todos os vértices restantes do dígrafo



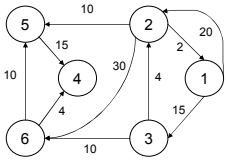
Exercício

- Utilize o algoritmo **ShortestPath** para obter, em seqüência crescente, os comprimentos dos caminhos mais curtos entre o vértice 1 e todos os vértices restantes do dígrafo



	1	2	3	4	5	6
1	0	20	15			
2	2	0			10	30
3		4	0			10
4				0		
5				15	0	
6				4	10	0

Solução



	1	2	3	4	5	6
1	0	20	15			
2	2	0			10	30
3		4	0			10
4				0		
5				15	0	
6				4	10	0

Iteração	S	Vértice Selecionado	DIST					
			1	2	3	4	5	6
inicial		1	0	20	15	+∞	+∞	+∞
1	1	3	0	19	15	+∞	+∞	25
2	1,3	2	0	19	15	+∞	29	25
3	1,3,2	6	0	19	15	29	29	25
4	1,3,2,6	4	0	19	15	29	29	25
final	1,3,2,6,4		0	19	15	29	29	25

Análise de ShortestPath

- O tempo consumido pelo algoritmo em um grafo com **n** vértices é:
  - O laço **for** da linha 1 leva um tempo de O(**n**)
  - O laço **while** é executado **n**-2 vezes
    - Cada execução deste laço requer um tempo de O(**n**) na linha 6, para selecionar o próximo vértice e novamente, nas linhas 8-10 para atualizar DIST
- Assim, o tempo total de execução é de $O(n^2)$

Análise de ShortestPath

- Qualquer algoritmo de caminho mais curto deve examinar cada aresta no grafo uma vez pelo menos, já que qualquer uma das arestas poderia estar num caminho mais curto
- Assim sendo, o tempo mínimo possível seria O(**e**)
- Uma vez que foi utilizada matriz de adjacências de custos para representar o grafo, isso toma o tempo de O(n^2) só para determinar quais as arestas que estão em G
  - Assim qualquer algoritmo de caminho mais curto que utilizar essa representação deve tomar O(n^2)
  - Dessa forma, o algoritmo ShortestPath encontra-se otimizado, dentro de um fator constante
- No caso de listas de adjacências, o tempo total para o laço **for** das linhas 8-10 pode ser reduzido para O(**e**)
  - Uma vez que DIST pode mudar somente no caso dos vértices adjacentes a **u**
- O tempo total para a linha 6 continua a ser O(n^2)

Caminhos Mais Curtos entre Todos os Pares

- O problema do caminho mais curto entre todos os pares requer encontrar os caminhos mais curtos entre todos os pares de vértices v_i e v_j, $i \neq j$
- Uma possível solução seria aplicar **n** vezes o algoritmo ShortestPath, uma vez para cada vértice em V(G) como origem
  - O tempo total consumido seria O(n^3)
- Para o problema de todos os pares, podemos obter um algoritmo mais simples em termos conceituais e que funcionará ainda que algumas arestas em G tenham ponderações negativas. contanto que G não tenha ciclos com comprimento negativo
  - O tempo de computação desse algoritmo ainda será O(n^3), embora seja menor o fator constante

Caminhos Mais Curtos entre Todos os Pares

- O grafo G é representado pela sua matriz de adjacências de custos, com
  - COST[**i**,**i**] = 0 e
  - COST[**i**,**j**] = $+\infty$ caso a aresta (**i**,**j**), $i \neq j$ não esteja em G
- Seja A^k[**i**,**j**] definido como o custo do caminho mais curto de **i** até **j** que não passa por nenhum vértice intermediário com índice superior a **k**
- Nesse caso A^n[**i**,**j**] será o custo do caminho mais curto de **i** para **j** em G, que possui **n** vértices
- A^0[**i**,**j**] é igual a COST[**i**,**j**] uma vez que os únicos caminhos de **i** até **j** permitidos não podem incluir vértices intermediários

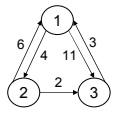
Caminhos Mais Curtos entre Todos os Pares

- A idéia básica no algoritmo de todos os pares é gerar sucessivamente as matrizes A^0, A^1, A^2, ..., A^n
- Se já tivermos gerado A^{k-1}, podemos gerar A^k, por compreender que para qualquer par de vértices **i** e **j**
  - ou (a) o caminho mais curto de **i** para **j** (que não atravessa nenhum vértice com índice superior a **k**) não passa pelo vértice **k**; assim, seu custo será A^{k-1}[**i**,**j**]
  - ou (b) o mais curto desses caminhos passa pelo vértice **k**: esse caminho é composto por uma rota de **i** até **k** e outro rota de **k** até **j**; esses caminhos devem ser os caminhos mais curtos de **i** para **k** e de **k** para **j** que não atravessam nenhum vértice com índice superior a **k**-1; assim, seus custos são A^{k-1}[**i**,**k**] e A^{k-1}[**k**,**j**]
- Isto é verdade se G não tiver nenhum ciclo com o comprimento negativo contendo o vértice **k**

Caminhos Mais Curtos entre Todos os Pares

- Assim, obtemos as seguintes fórmulas para A^k[**i**,**j**]
  - A^0[**i**,**j**] = COST[**i**,**j**]
  - A^k[**i**,**j**] = mín{A^{k-1}[**i**,**j**], A^{k-1}[**i**,**k**]+A^{k-1}[**k**,**j**]}, **k**≥1
- O algoritmo **AllCosts** calcula A^n[**i**,**j**]
- O cálculo é efetuado na própria matriz, sendo eliminado o superscrito de A
  - O cálculo pode ser feito na própria matriz uma vez que A^k[**i**,**k**]=A^{k-1}[**i**,**k**] e A^k[**k**,**j**]=A^{k-1}[**k**,**j**] de modo que a computação não altera o resultado

Exemplo



COST	1	2	3
1	0	4	11
2	6	0	2
3	3	$+\infty$	0

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	$+\infty$	0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

Algoritmo AllCosts

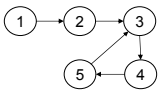
```
// COST[1..n,1..n] é a matriz de adjacências de custos de um grafo
   com n vértices; A[i,j] é o custo do caminho mais curto entre
   vértices i e j. COST[i,i]= 0, 1 ≤ i ≤ n

procedure AllCosts(n,COST,A)
1  for i ← 1 to n do
2    for j ← 1 to n do
3      A[i,j] ← COST[i,j];
4    next j
5  next i
6  for k ← 1 to n do
7    for i ← 1 to n do
8      for j ← 1 to n do
9        A[i,j] ← min{A[i,j], A[i,k]+A[k,j]};
10     next j
11   next i
12 next k
end AllCosts
```


Fechamento Transitivo

- Um problema relacionado com o problema do caminho mais curto de todos os pares é o de se determinar para cada par de vértices **i** e **j** em G, a existência de um caminho de **i** até **j**
- O comprimento do caminho é igual ao número de arestas
- Sendo A a matriz de adjacências de G, então a matriz A^+ que tiver a propriedade
  - A^+[**i**,**j**] = 1, se há um caminho de comprimento > 0 de **i** para **j**
  - A^+[**i**,**j**] = 0, caso contrário
- A^+ é denominada matriz de **fechamento transitivo** de G
- A matriz A^* com a propriedade
  - A^*[**i**,**j**] = 1, se há um caminho de comprimento ≥ 0 de **i** para **j**
  - A^*[**i**,**j**] = 0, caso contrário
- A^* é denominada matriz de **fechamento transitivo reflexivo** de G

Exemplo



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

A^+	1	2	3	4	5
1	0	1	1	1	1
2	0	0	1	1	1
3	0	0	1	1	1
4	0	0	1	1	1
5	0	0	1	1	1

A^*	1	2	3	4	5
1	1	1	1	1	1
2	0	1	1	1	1
3	0	0	1	1	1
4	0	0	1	1	1
5	0	0	1	1	1

Fechamento Transitivo

- A única diferença entre A^* e A^+ reside nos termos da diagonal
- A^+[**i**,**j**] = 1 se existe um ciclo de comprimento > 1 contendo o vértice **i**, enquanto A*[**i**,**i**] será sempre igual a 1, uma vez que existe um caminho de comprimento 0 que vai de **i** para **i**
- Se usarmos o algoritmo **AllCosts** com
  - COST[**i**,**j**]=1, se (**i**,**j**) for uma aresta em G e
  - COST[**i**,**j**]=+∞ se (**i**,**j**) não está em G
- então podemos obter facilmente A^+ da matriz final **A** deixando A^+[**i**,**j**]=1 se A[**i**,**j**] < +∞
- A* pode ser obtido de A^+, fazendo-se iguais a 1 todos os elementos da diagonal
- O tempo total é de O(n^3)

Ordenação Topológica

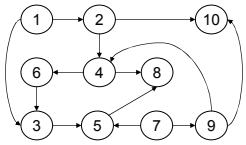
- Todos os projetos, exceto os mais simples, podem ser subdivididos em diversos subprojetos denominados **tarefas** ou **atividades**
- A terminação bem-sucedida dessas atividades resultará na conclusão do projeto na sua totalidade
  - Em projetos de engenharia ou manufatura, uma tarefa é dividida em subtarefas. Em geral, o término de certas subtarefas deve preceder a execução de outras subtarefas. Se uma subtarefa x deve preceder uma subtarefa y, isto será denotado por x < y. A ordenação topológica, neste caso, consiste em dispor as subtarefas em uma ordem tal que, antes da iniciação de cada subtarefa, todas as subtarefas de que ela depende tenham sido previamente completadas
  - Em um currículo universitário, algumas disciplinas devem ser cursadas antes de outras, uma vez que se baseiam nos tópicos apresentados nas disciplinas que são seus pré-requisitos. Se uma disciplina x é pré-requisito da disciplina y, a notação será x < y. A ordenação topológica corresponde, no caso, a arranjar as disciplinas em uma ordem tal que nenhuma delas exija como pré-requisito uma outra que não tenha sido previamente cursada
  - Em um programa, alguns procedimentos podem conter referências (chamadas) a outros procedimentos. Se um procedimento x é chamado por um procedimento y, denota-se o fato por x < y. Neste caso, a ordenação topológica implica o arranjo das declarações de procedimento em tal forma que nunca haja referências a procedimentos a serem declarados posteriormente

Ordenação Topológica

- Uma ordem parcial de um conjunto S é uma relação entre os elementos de S
- Esta relação, denotada pelo símbolo **<**, verbalmente lida como **precede**:
  - **x** < **y** é lido como "**x** precede **y**"
- Uma ordem parcial deve satisfazer as três seguintes propriedades (axiomas), para quaisquer elementos distintos **x**, **y**, **z** de S:
  - (a) se **x** < **y** e **y** < **z**, então **x** < **z** (*transitividade*)
  - (b) se **x** < **y**, então não ocorre **y** < **x** (*assimetria*)
  - (c) não ocorre **z** < **z** (*irreflexiva*)
- Admite-se, por motivos óbvios, que o conjunto S é finito
- Uma ordem parcial pode ser representada por um grafo no qual os vértices denotam os elementos de S e as arestas representam as relações de ordem

Ordenação Topológica

- Admita que o conjunto S e suas relações de ordenação sejam inicialmente representados por uma seqüência de pares de nós
- Para o exemplo ao lado, as relações de ordem entre os vértices são mostrados a seguir, no qual os símbolos **<** estão explicitados para maior clareza:
  - 1 < 2
  - 2 < 4
  - 4 < 6
  - 2 < 10
  - 4 < 8
  - 6 < 3
  - 1 < 3
  - 3 < 5
  - 5 < 8
  - 7 < 5
  - 7 < 9
  - 9 < 4
  - 9 < 10

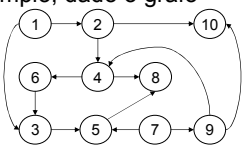


Ordenação Topológica

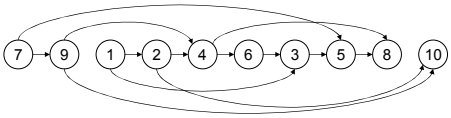
- O problema da **ordenação topológica** consiste em encontrar, em uma ordem linear, uma ordem parcial
- Graficamente, isto implica o arranjo linear dos vértices do grafo, de tal maneira que todas as arestas tenham o mesmo sentido (para a direita, por exemplo)
- As propriedades (a) e (b) da ordem parcial garantem a ausência de ciclos no grafo, o que permite encontrar uma ordem parcial em uma ordem linear

Ordenação Topológica

- Por exemplo, dado o grafo



- Uma ordenação topológica resultante seria



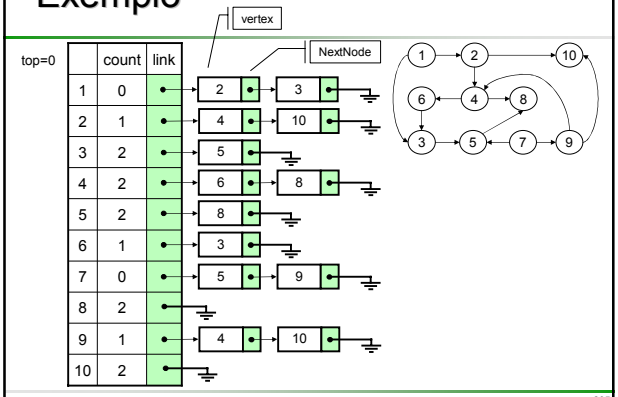
Ordenação Topológica

- O algoritmo para encontrar uma das possíveis ordenações lineares é bastante simples:
  - Escolhem-se inicialmente os nós que não sejam precedidos por quaisquer outros (é necessário que haja no mínimo um elemento nessas condições; caso contrário existiria um ciclo e o conjunto não seria parcialmente ordenado), denotando esse conjunto de nós sem predecessores por P
  - escolhe-se então um nó x de P, colocando x no início da lista de saída L; em seguida o nó x é eliminado do conjunto S
  - O conjunto S resultante ainda se encontra parcialmente ordenado, devendo então o mesmo algoritmo ser aplicado sucessivamente até que o conjunto S se esgote

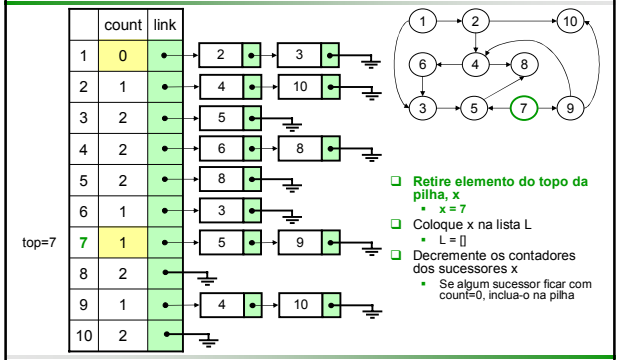
Ordenação Topológica

- Para efeitos de implementação, admite-se no algoritmo que o grafo seja representado por listas de adjacências
- Os vértices de cabeçalho dessas relações contêm dois campos:
  - **count** (contador): O campo **count** contém o grau de entrada do vértice correspondente
  - **link** um ponteiro para o primeiro vértice da lista de adjacências do vértice
- Cada vértice será representado por uma lista com 2 campos:
  - **vertex**, que mantém o número do vértice correspondente
  - **NextNode**, um ponteiro para o próximo vértice
- Os campos **count** podem ser estabelecidos facilmente por ocasião da entrada
  - Ao se dar entrada na aresta (**i**,**j**), a contagem do vértice **j** é incrementada por 1
- A lista de vértices com contagem zero é mantida sob forma de uma pilha
  - Poderia ter sido utilizada uma fila, mas a pilha é um pouco mais simples
- A pilha é ligada por meio do campo **count** dos vértices da cabeçalho, uma vez que esse campo não é de utilidade, depois que o **count** ficou igual a zero

Exemplo

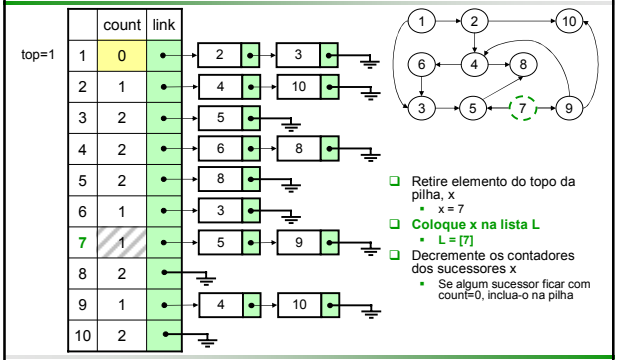


Exemplo



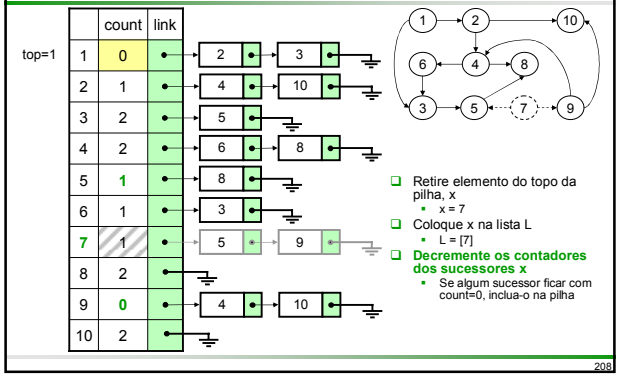
- **Retire elemento do topo da pilha, x**
  - **x = 7**
- Coloque x na lista L
  - L = []
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo



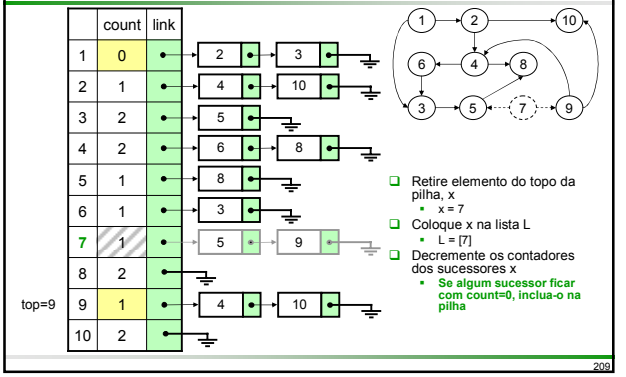
- Retire elemento do topo da pilha, x
  - x = 7
- **Coloque x na lista L**
  - **L = [7]**
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo



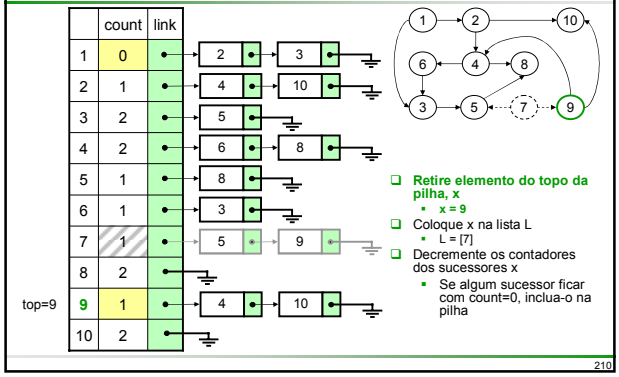
- Retire elemento do topo da pilha, x
  - x = 7
- Coloque x na lista L
  - L = [7]
- **Decremente os contadores dos sucessores x**
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo



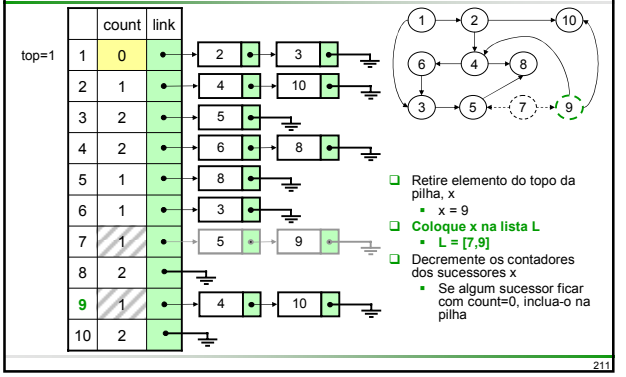
- Retire elemento do topo da pilha, x
  - x = 7
- Coloque x na lista L
  - L = [7]
- Decremente os contadores dos sucessores x
  - **Se algum sucessor ficar com count=0, inclua-o na pilha**

Exemplo



- **Retire elemento do topo da pilha, x**
  - **x = 9**
- Coloque x na lista L
  - L = [7]
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

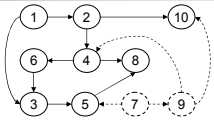


- Retire elemento do topo da pilha, x
  - x = 9
- **Coloque x na lista L**
  - **L = [7,9]**
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=1

	count	link
1	0	2 → 3
2	1	4 → 10
3	2	5
4	1	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	1	

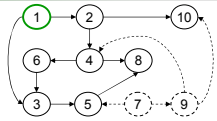


- Retire elemento do topo da pilha, x
  - x = 9
- Coloque x na lista L
  - L = [7,9]
- **Decremente os contadores dos sucessores x**
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=1

	count	link
1	0	2 → 3
2	1	4 → 10
3	2	5
4	1	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	1	

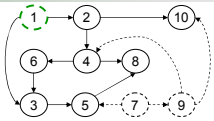


- **Retire elemento do topo da pilha, x**
  - **x = 1**
- Coloque x na lista L
  - L = [7,9]
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=0

	count	link
1	0	2 → 3
2	1	4 → 10
3	2	5
4	1	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	1	

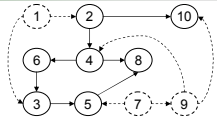


- Retire elemento do topo da pilha, x
  - x = 1
- **Coloque x na lista L**
  - **L = [7,9,1]**
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=0

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	1	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	1	

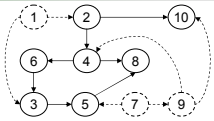


- Retire elemento do topo da pilha, x
  - x = 1
- Coloque x na lista L
  - L = [7,9,1]
- **Decremente os contadores dos sucessores x**
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=2

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	1	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	1	

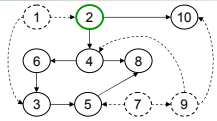


- Retire elemento do topo da pilha, x
  - x = 1
- Coloque x na lista L
  - L = [7,9,1]
- Decremente os contadores dos sucessores x
  - **Se algum sucessor ficar com count=0, inclua-o na pilha**

Exemplo

top=2

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	1	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	1	

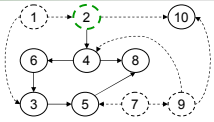


- **Retire elemento do topo da pilha, x**
  - **x = 2**
- Coloque x na lista L
  - L = [7,9,1]
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=0

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	1	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	1	

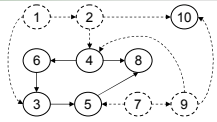


- Retire elemento do topo da pilha, x
  - x = 2
- **Coloque x na lista L**
  - **L = [7,9,1,2]**
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=0

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	**0**	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	**0**	

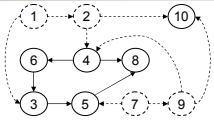


- Retire elemento do topo da pilha, x
  - x = 2
- Coloque x na lista L
  - L = [7,9,1,2]
- **Decremente os contadores dos sucessores x**
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=10

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	0	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	4	

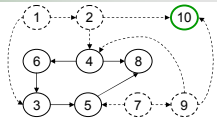


- Retire elemento do topo da pilha, x
  - x = 2
- Coloque x na lista L
  - L = [7,9,1,2]
- Decremente os contadores dos sucessores x
  - **Se algum sucessor ficar com count=0, inclua-o na pilha**

Exemplo

top=10

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	0	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	4	

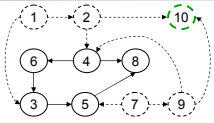


- **Retire elemento do topo da pilha, x**
  - **x = 10**
- Coloque x na lista L
  - L = [7,9,1,2]
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=4

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	0	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	4	

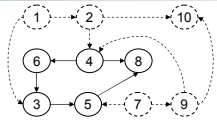


- Retire elemento do topo da pilha, x
  - x = 10
- **Coloque x na lista L**
  - **L = [7,9,1,2,10]**
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=4

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	0	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	4	

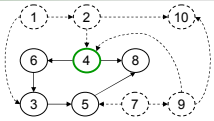


- Retire elemento do topo da pilha, x
  - x = 10
- Coloque x na lista L
  - L = [7,9,1,2,10]
- **Decremente os contadores dos sucessores x**
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=4

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	**0**	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	4	

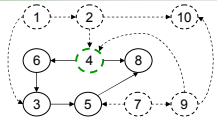


- **Retire elemento do topo da pilha, x**
  - **x = 4**
- Coloque x na lista L
  - L = [7,9,1,2,10]
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=0

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	**0**	6 → 8
5	1	8
6	1	3
7	1	5 → 9
8	2	
9	1	4 → 10
10	4	

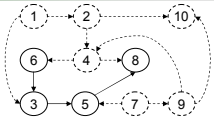


- Retire elemento do topo da pilha, x
  - x = 4
- **Coloque x na lista L**
  - **L = [7,9,1,2,10,4]**
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=0

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	**0**	6 → 8
5	1	8
6	**0**	3
7	1	5 → 9
8	**1**	
9	1	4 → 10
10	4	

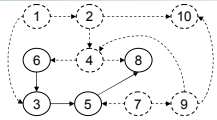


- Retire elemento do topo da pilha, x
  - x = 4
- Coloque x na lista L
  - L = [7,9,1,2,10,4]
- **Decremente os contadores dos sucessores x**
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=6

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	**0**	6 → 8
5	1	8
6	**0**	3
7	1	5 → 9
8	1	
9	1	4 → 10
10	4	

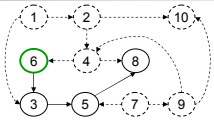


- Retire elemento do topo da pilha, x
  - x = 4
- Coloque x na lista L
  - L = [7,9,1,2,10,4]
- Decremente os contadores dos sucessores x
  - **Se algum sucessor ficar com count=0, inclua-o na pilha**

Exemplo

top=6

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	0	6 → 8
5	1	8
6	**0**	3
7	1	5 → 9
8	1	
9	1	4 → 10
10	4	

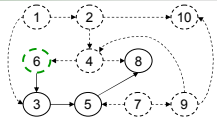


- **Retire elemento do topo da pilha, x**
  - **x = 6**
- Coloque x na lista L
  - L = [7,9,1,2,10,4]
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=0

	count	link
1	0	2 → 3
2	0	4 → 10
3	1	5
4	0	6 → 8
5	1	8
6	**0**	3
7	1	5 → 9
8	1	
9	1	4 → 10
10	4	

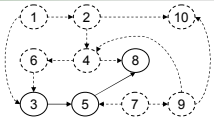


- Retire elemento do topo da pilha, x
  - x = 6
- **Coloque x na lista L**
  - **L = [7,9,1,2,10,4,6]**
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=0

	count	link
1	0	2 → 3
2	0	4 → 10
3	**0**	5
4	0	6 → 8
5	1	8
6	0	3
7	1	5 → 9
8	1	
9	1	4 → 10
10	4	

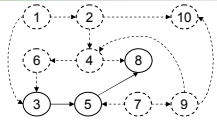


- Retire elemento do topo da pilha, x
  - x = 6
- Coloque x na lista L
  - L = [7,9,1,2,10,4,6]
- **Decremente os contadores dos sucessores x**
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=3

	count	link
1	0	2 → 3
2	0	4 → 10
3	0	5
4	0	6 → 8
5	1	8
6	0	3
7	1	5 → 9
8	1	
9	1	4 → 10
10	4	

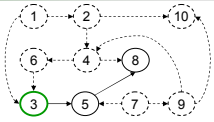


- Retire elemento do topo da pilha, x
  - x = 6
- Coloque x na lista L
  - L = [7,9,1,2,10,4,6]
- Decremente os contadores dos sucessores x
  - **Se algum sucessor ficar com count=0, inclua-o na pilha**

Exemplo

top=3

	count	link
1	0	2 → 3
2	0	4 → 10
3	0	5
4	0	6 → 8
5	1	8
6	0	3
7	1	5 → 9
8	1	
9	1	4 → 10
10	4	

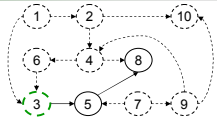


- **Retire elemento do topo da pilha, x**
  - **x = 3**
- Coloque x na lista L
  - L = [7,9,1,2,10,4,6]
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=0

	count	link
1	0	2 → 3
2	0	4 → 10
3	0	5
4	0	6 → 8
5	1	8
6	0	3
7	1	5 → 9
8	1	
9	1	4 → 10
10	4	

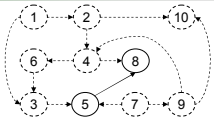


- Retire elemento do topo da pilha, x
  - x = 3
- **Coloque x na lista L**
  - **L = [7,9,1,2,10,4,6,3]**
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=0

	count	link
1	0	2 → 3
2	0	4 → 10
3	0	5
4	0	6 → 8
5	**0**	8
6	0	3
7	1	5 → 9
8	1	
9	1	4 → 10
10	4	

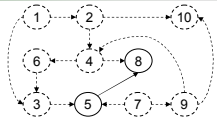


- Retire elemento do topo da pilha, x
  - x = 3
- Coloque x na lista L
  - L = [7,9,1,2,10,4,6,3]
- **Decremente os contadores dos sucessores x**
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=5

	count	link
1	0	2 → 3
2	0	4 → 10
3	0	5
4	0	6 → 8
5	**0**	8
6	0	3
7	1	5 → 9
8	1	
9	1	4 → 10
10	4	

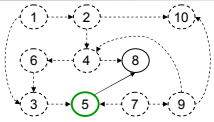


- Retire elemento do topo da pilha, x
  - x = 3
- Coloque x na lista L
  - L = [7,9,1,2,10,4,6,3]
- Decremente os contadores dos sucessores x
  - **Se algum sucessor ficar com count=0, inclua-o na pilha**

Exemplo

top=5

	count	link
1	0	2 → 3
2	0	4 → 10
3	0	5
4	0	6 → 8
5	0	8
6	0	3
7	1	5 → 9
8	1	
9	1	4 → 10
10	4	

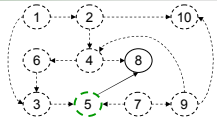


- **Retire elemento do topo da pilha, x**
  - **x = 5**
- Coloque x na lista L
  - L = [7,9,1,2,10,4,6,3]
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=0

	count	link
1	0	2 → 3
2	0	4 → 10
3	0	5
4	0	6 → 8
5	0	8
6	0	3
7	1	5 → 9
8	1	
9	1	4 → 10
10	4	

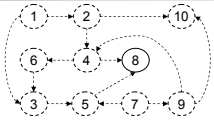


- Retire elemento do topo da pilha, x
  - x = 5
- **Coloque x na lista L**
  - **L = [7,9,1,2,10,4,6,3,5]**
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=0

	count	link
1	0	2 → 3
2	0	4 → 10
3	0	5
4	0	6 → 8
5	0	8
6	0	3
7	1	5 → 9
8	**0**	
9	1	4 → 10
10	4	

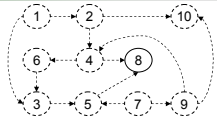


- Retire elemento do topo da pilha, x
  - x = 5
- Coloque x na lista L
  - L = [7,9,1,2,10,4,6,3,5]
- **Decremente os contadores dos sucessores x**
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

top=8

	count	link
1	0	2 → 3
2	0	4 → 10
3	0	5
4	0	6 → 8
5	0	8
6	0	3
7	1	5 → 9
8	0	
9	1	4 → 10
10	4	

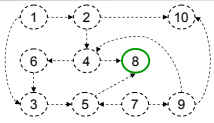


- Retire elemento do topo da pilha, x
  - x = 5
- Coloque x na lista L
  - L = [7,9,1,2,10,4,6,3,5]
- Decremente os contadores dos sucessores x
  - **Se algum sucessor ficar com count=0, inclua-o na pilha**

Exemplo

top=8

	count	link
1	0	2 → 3
2	0	4 → 10
3	0	5
4	0	6 → 8
5	0	8
6	0	3
7	1	5 → 9
8	**0**	
9	1	4 → 10
10	4	

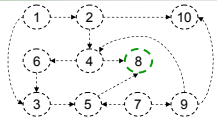


- **Retire elemento do topo da pilha, x**
  - **x = 8**
- Coloque x na lista L
  - L = [7,9,1,2,10,4,6,3,5]
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Exemplo

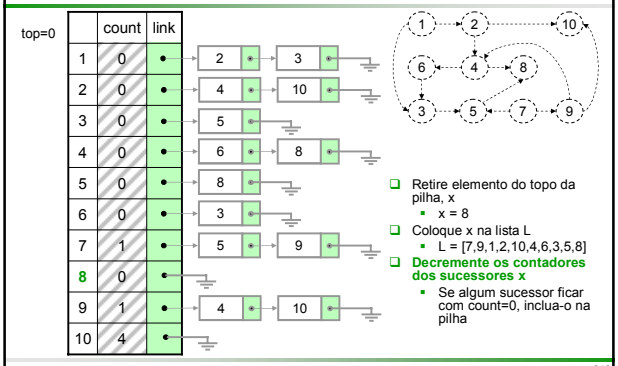
top=0

	count	link
1	0	2 → 3
2	0	4 → 10
3	0	5
4	0	6 → 8
5	0	8
6	0	3
7	1	5 → 9
8	0	
9	1	4 → 10
10	4	



- Retire elemento do topo da pilha, x
  - x = 8
- **Coloque x na lista L**
  - **L = [7,9,1,2,10,4,6,3,5,8]**
- Decremente os contadores dos sucessores x
  - Se algum sucessor ficar com count=0, inclua-o na pilha

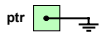
Exemplo



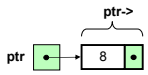
- Retire elemento do topo da pilha, x
  - x = 8
- Coloque x na lista L
  - L = [7,9,1,2,10,4,6,3,5,8]
- **Decremente os contadores dos sucessores x**
  - Se algum sucessor ficar com count=0, inclua-o na pilha

Ordenação Topológica

- No algoritmo seguinte, assume-se que:
  - Ponteiros "aterrados" são indicados por **NULL**
    - ptr ← NULL;



  - Se **ptr** é uma variável do tipo ponteiro, então **ptr->** refere-se ao que é apontado por **ptr**



Ordenação Topológica

```
// n vértices de um grafo são relacionados em ordem topológica
procedure TopologicalOrder(n,count,link)
 1  top ← 0;          // inicialize a pilha
 2  for i ← 1 to n do // crie uma pilha de ligação de vértices sem predecessores
 3    if count[i] = 0 then count[i] ← top; top ← i; endif
 4  next i
 5  for i ← 1 to n do   // imprima os vértices em ordem topológica
 6    if top = 0 then print('grafo tem um ciclo'); return; endif
 7    j ← top; top ← count[top]; print(j); // coloque vértice j na lista de saída
 8    ptr ← link[j];
 9    while ptr ≠ NULL do // decremente a contagem de vértices sucessores de j
10      k ← ptr->vertex;      // k é o sucessor de j
11      count[k] ← count[k] - 1;
12      if count[k] = 0 then // adicione vértice k à pilha
13        count[k] ← top; top ← k;
14      endif
14      ptr ← ptr->NextNode;
15    endwhile
16  next i
end TopologicalOrder
```


Ordenação Topológica

- Como resultado de uma escolha criteriosa de estruturas de dados o algoritmo é muito eficiente
- Para uma rede com **n** vértices e **e** arestas
  - o laço de linhas 2-4 requer um tempo de O(**n**)
  - as linhas 6-8 requerem O(**n**) para o algoritmo inteiro
  - o laço **while** precisa de tempo O($\mathbf{d_i}$) para cada vértice **i**, onde $\mathbf{d_i}$ é o grau de saída do vértice **i** ($1 \le \mathbf{i} \le \mathbf{n}$)
    - Uma vez que esse laço é encontrado uma vez para cada vértice de saída, o tempo total para essa parte do algoritmo é de $O((\Sigma \mathbf{d_i})+\mathbf{n}) = O(\mathbf{e}+\mathbf{n})$
- Portanto, o tempo assintótico de computação do algoritmo é O(**e**+**n**), ou seja, é linear segundo o tamanho do problema

Resumo

- Grafos são estruturas de dados que possuem um conjunto de vértices e um conjunto de arestas
- Entre outros, foram vistos algoritmos elementares de busca em grafos:
  - Profundidade
  - Largura
- Observa-se que a eficiência de algoritmos em grafos, em alguns casos, depende da escolha adequada de estruturas de dados