Grafos

Professor: Elton Sarmanho[1]
E-mail: eltonss@ufpa.br

[1]Faculdade de Sistemas de Informação - UFPA/CUTINS

7 de outubro de 2025

Roteiro

Roteiro

Representação computacional
- Matriz de Adjacências
- Matriz de Incidências
- Lista de Adjacências

Algoritmos de Busca
- Busca em Largura
- Busca em Profundidade
- Trabalho I.

Objetivos de Aprendizagem

Ao final desta aula, você será capaz de:

- **Conceituar** o que é um grafo e identificar seus componentes fundamentais (vértices e arestas).
- **Diferenciar** os principais tipos de grafos e suas terminologias (simples, dígrafo, ponderado, etc.).
- **Modelar** problemas do mundo real utilizando a estrutura de grafos.
- **Implementar** as três principais representações computacionais de um grafo: Matriz de Adjacências, Lista de Adjacências e Matriz de Incidência.
- **Analisar** as vantagens e desvantagens de cada representação, sabendo escolher a mais adequada para cada cenário.
- **Aplicar** algoritmos de travessia fundamentais: Busca em Largura (BFS) e Busca em Profundidade (DFS).
- **Resolver** o problema do caminho mínimo com o Algoritmo de Dijkstra.

Grafos Estão em Todo Lugar!

- Pense em grafos como uma forma de representar **conexões**. Quase tudo no mundo pode ser visto como um conjunto de "coisas" e as relações entre elas.

Exemplos do Mundo Real

- **Redes Sociais:** Você e seus amigos são os "pontos" (vértices), e as amizades são as "linhas" (arestas) que os conectam.
- **Mapas e GPS:** Cidades são vértices e as estradas entre elas são arestas. O GPS usa algoritmos em grafos para encontrar o caminho mais rápido!
- **A Internet:** Páginas da web são vértices, e os links de uma página para outra são as arestas.
- **Logística e Transportes:** Aeroportos são vértices, e os voos diretos são arestas. Como uma empresa aérea otimiza suas rotas? Com grafos.

- A Teoria dos Grafos nos dá a linguagem e as ferramentas para analisar e resolver esses problemas de conexão de forma estruturada.

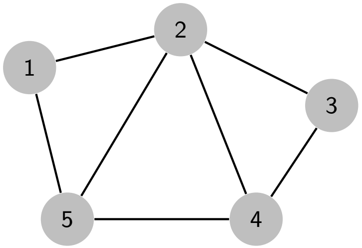
A Definição Formal de um Grafo

- Formalmente, um grafo **G** é um par ordenado $\mathbf{G} = (V, E)$, onde:
  - **V** é um conjunto de **Vértices** (ou nós). São os "pontos" ou as "entidades" do nosso modelo.
  - **E** é um conjunto de **Arestas** (ou arcos). São as "linhas" que conectam pares de vértices, representando a relação entre eles.

- Uma aresta $\mathbf{e} \in E$ que conecta os vértices u e v é representada como um par: $\mathbf{e} = (u, v)$.
  - Dizemos que os vértices u e v são **adjacentes** (ou vizinhos).
  - A aresta (u, v) é **incidente** aos vértices u e v.

Representação de um Grafo

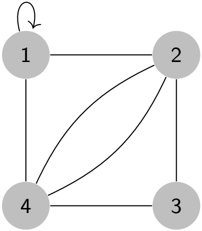
Vértices	1, 2, 3, 4, 5
Arestas	(1,2),(1,5),(5,2),(5,4), (2,4),(2,3),(4,3)
Tamanho $\lvert V \rvert$	Número de elementos em G
$\lvert E \rvert$	Número de arestas em G



Tipos Especiais de Arestas e Grafos

Laços e Arestas Paralelas

- **Laço (Loop):** Uma aresta que conecta um vértice a si mesmo, ex: (v, v).
- **Arestas Paralelas (ou Múltiplas):** Quando existe mais de uma aresta conectando o mesmo par de vértices.



Tipos Especiais de Arestas e Grafos

Classificações Importantes

- **Multigrafo:** Um grafo que pode conter laços e/ou arestas paralelas.
- **Grafo Simples:** Um grafo que **não** possui laços nem arestas paralelas. *(Este é o tipo mais comum que estudaremos!)*
- **Grafo Trivial:** Um grafo com apenas um vértice e nenhuma aresta.

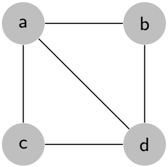
Subgrafo

Definição

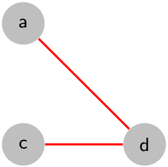
Um grafo $G' = (V', E')$ é um **subgrafo** de $G = (V, E)$ se, e somente se, $V' \subseteq V$ e $E' \subseteq E$.

- Em outras palavras, um subgrafo é formado pegando um "pedaço" dos vértices e arestas do grafo original.
- **Exemplo prático:** A rede de amizades dos alunos de uma única turma é um subgrafo da rede de amizades de toda a universidade.

Grafo G

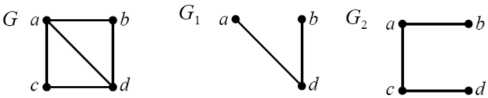


Subgrafo G'



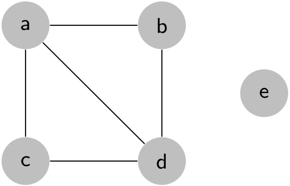
Subgrafo

$$
\begin{array}{ll}
\color{red}{G} & V = a, b, c, d \\
 & E = \{ab, ac, ad, bd, cd\} \\
\color{red}{G_1} & V_1 = a, b, d \\
 & E_1 = \{ad, bd\} \\
\color{red}{G_2} & V_2 = a, b, c, d \\
 & E_2 = \{ab, ac, cd\}
\end{array}
$$



Grau de um Vértice

- O **Grau** (ou valência) de um vértice v, denotado por $deg(v)$, é o número de arestas incidentes a ele. Em grafos com laços, um laço conta como 2 no grau.

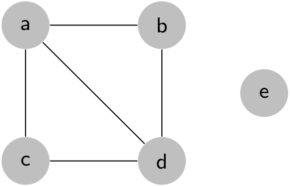


Exemplos de Grau

- $deg(a) = 3$
- $deg(b) = 2$
- $deg(d) = 3$
- $deg(e) = 0$

Grau de um Vértice

- O **Grau** (ou valência) de um vértice v, denotado por $deg(v)$, é o número de arestas incidentes a ele. Em grafos com laços, um laço conta como 2 no grau.



Terminologia

- **Vértice Isolado:** Um vértice com grau 0.
- **Vértice Folha (ou Terminal):** Um vértice com grau 1.

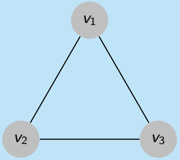
Theorem (Handshaking Lemma)

A soma dos graus de todos os vértices de um grafo é igual a duas vezes o número de arestas: $\sum_{v \in V} deg(v) = 2|E|$.

Grafos Regulares e Completos

Grafo Regular

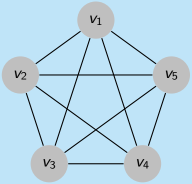
Um grafo é **k-regular** se todos os seus vértices possuem grau k.



Grafo 2-Regular

Grafo Completo (K_n)

Um grafo simples é **completo** se todo par de vértices distintos é conectado por uma aresta. O grafo completo com n vértices é denotado por K_n.



Grafo Completo K_5

Theorem

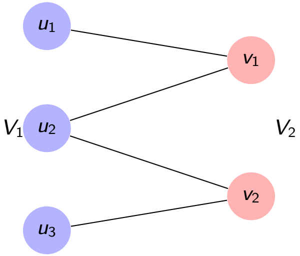
O número de arestas em um grafo completo K_n é $|E| = \frac{n(n-1)}{2}$.

Grafo Bipartido

Definição

- Um grafo $G = (V, E)$ é dito bipartido quando **V** pode ser particionado em dois subconjuntos V_1 e V_2, $\forall$ aresta de G faz ligação de vertice de V_1 a um vértice de V_2

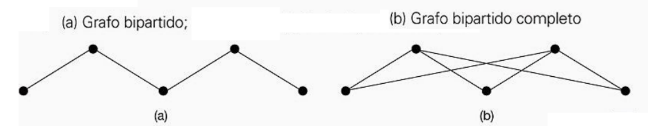
$$G(V_1 \cup V_2, E)$$



Grafo Bipartido Completo

Definição

- Um grafo $G = (V_1 \cup V_2, E)$ é dito bipartido completo quando existe uma aresta para todo par de vértice u, v sendo $u \in V_1$ e $v \in V_2$
- $K_{m,n}$
  - $m = |V_1|$
  - $n = |V_2|$



Caminhos e Ciclos

Definição

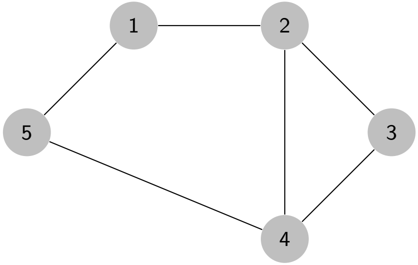
- Dado $G = (V, E)$, uma sequência de vértices $v_1, v_2, ..., v_j$, tal que $(v_i, v_{i+1}) \in E$ e $1 \leq i \leq |j - 1|$, recebe nome de **caminho** de v_1 a v_j

  Comprimento: Número de arestas do caminho
  Caminho simples: Todos os vértices são distintos
  Trajeto: Todas as arestas são distintas

Caminhos, Trajetos e Ciclos (Parte 1)

Definições Iniciais: Como "Andar" no Grafo

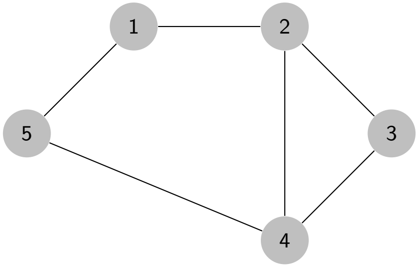
- **Caminho (Walk):** A forma mais geral de travessia. É uma sequência de vértices onde cada vértice adjacente está conectado por uma aresta. **Vértices e arestas podem ser repetidos.**. **Exemplo de Caminho (vértice 2 repetido):** $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 2$



Caminhos, Trajetos e Ciclos (Parte 1)

Definições Iniciais: Como "Andar" no Grafo

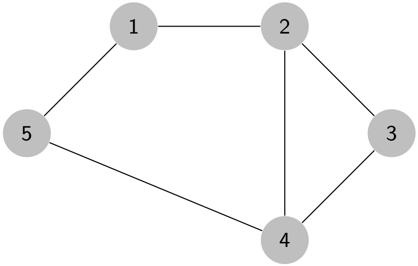
- **Trajeto (Trail):** Um tipo especial de caminho onde as **arestas não se repetem**. Vértices, no entanto, ainda podem ser visitados mais de uma vez. **Exemplo de Trajeto (vértice 2 repetido):** $1 \to 5 \to 4 \to 2 \to 3$



Caminhos, Trajetos e Ciclos (Parte 2)

Definições Fundamentais: As Travessias Mais Úteis

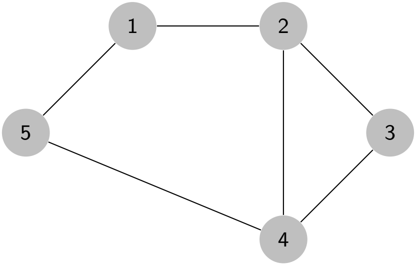
- **Caminho Simples (Path):** O conceito mais importante de "caminho". É um trajeto onde os **vértices não se repetem** (com exceção, talvez, do início e fim). Ex: **Caminho Simples de 1 a 3:** $1 \to 2 \to 3$



Caminhos, Trajetos e Ciclos (Parte 2)

Definições Fundamentais: As Travessias Mais Úteis

- **Ciclo (Cycle):** Um caminho simples de comprimento 3 ou mais que **começa e termina no mesmo vértice**. Ciclos são estruturas cruciais na análise de grafos. Ex: **Ciclo:** $2 \to 3 \to 4 \to 2$

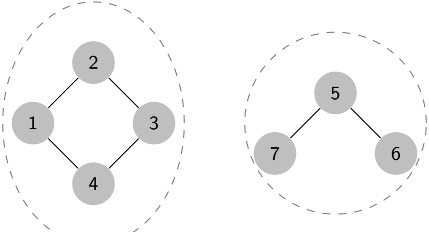


Conectividade

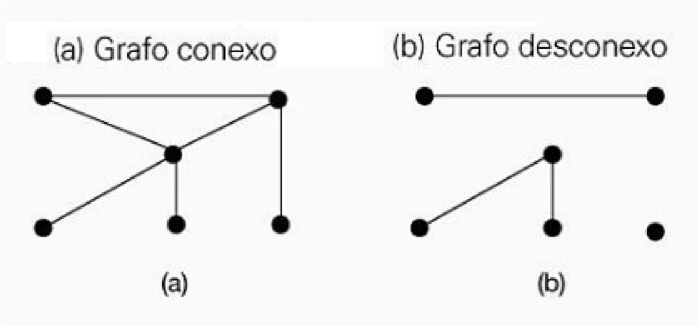
Grafo Conexo

Um grafo (não-direcionado) é **conexo** se existe um caminho entre todo par de vértices distintos.

- Se um grafo não é conexo, ele é chamado de **desconexo**.
- Um grafo desconexo é composto por duas ou mais **componentes conexas**.
- **Analogia:** Um mapa de um país com ilhas desconectadas. Cada ilha (e suas cidades/estradas) é uma componente conexa.

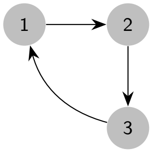


Grafo Conexo



Grafos Direcionados (Dígrafos)

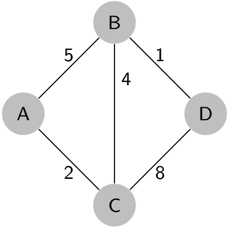
- Em um **Dígrafo**, as arestas (chamadas de **arcos**) têm uma direção. A aresta (u, v) significa que há uma conexão de u para v, mas não necessariamente de v para u.
- **Analogia:** Ruas de mão única.
- **Terminologia Adicional:**
  - **Grau de Entrada (in-degree):** Número de arcos que chegam a um vértice.
  - **Grau de Saída (out-degree):** Número de arcos que partem de um vértice.



Vértice 2:
Grau de Entrada: 1
Grau de Saída: 1

Grafos Ponderados

- Em um **Grafo Ponderado**, cada aresta possui um valor numérico associado, chamado de **peso** ou **custo**.
- Este peso pode representar distância, tempo, custo financeiro, capacidade, etc.



Grafos Ponderados

- Em um **Grafo Ponderado**, cada aresta possui um valor numérico associado, chamado de **peso** ou **custo**.
- Este peso pode representar distância, tempo, custo financeiro, capacidade, etc.

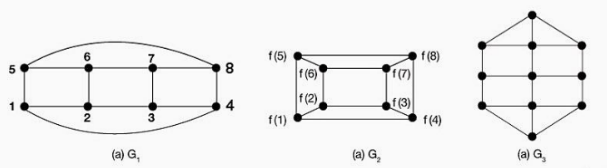
Aplicações

Essenciais para problemas de otimização. O Algoritmo de Dijkstra, que veremos mais tarde, opera em grafos ponderados para encontrar o caminho de menor custo.

Grafo Isomorfos

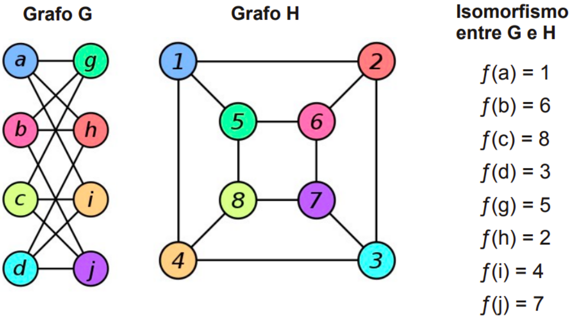
Definição

- Dois Grafos $G_1(V_1, E_1)$ e $G_2(V_2, E_2)$ são isomorfos se existir uma bijeção tal que $(u, v) \in E_1$ se e somente se $(f(u), f(v)) \in E_2$
  - $\mathbf{(u, v) \in E_1 \iff (f(u), f(v)) \in E_2}$
  - É possível re-rotular os vértices de G_1 para serem rótulos de G_2 mantendo as arestas correspondentes em G_1 e G_2



Grafo Isomorfos

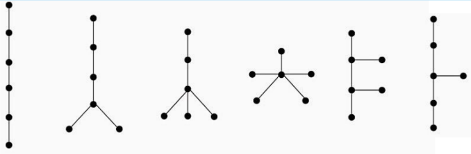
- São aplicados em grafos com mesma estrutura topológica



Árvore

Definição

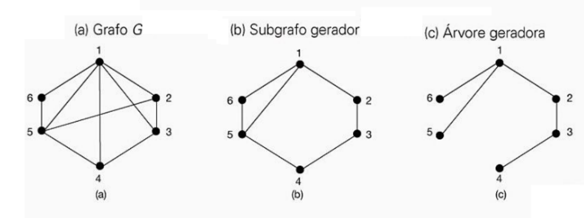
- Um grafo $T(V,E)$ que não possui ciclos e é conexo é chamado **árvore** e possui as seguintes propriedades:
  - Seja $v \in V$, se v possui grau ≤ 1, então v é **folha**. Se grau > 1, então v é **interno**
  - Uma árvore T com n vértices possui $n-1$ arestas
  - Um grafo G é uma subárvore somente se existir um único caminho entre cada par de vértices de G



Árvore geradora

Definição

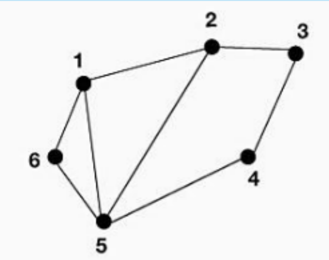
- Dado um grafo $G(V, E)$, denomina-se **subgrafo gerador** o grafo $H(V, E)$ que é subgrafo de G, tal que $V(G) = V(H)$
  - Se H é uma árvore, então é chamado de **árvore geradora**

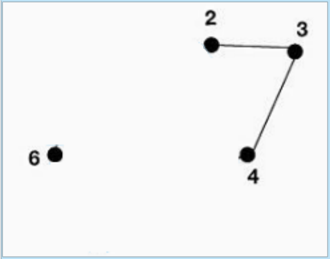


Corte de vértice e aresta

Definição

- **Corte de vértice** em $G(V, E)$ é um subconjunto **mínimo** de vértices V_1, sendo $V_1 \subseteq V$, tal que se os vértices de V_1 forem removidos de G, o grafo fica desconexo ou trivial.

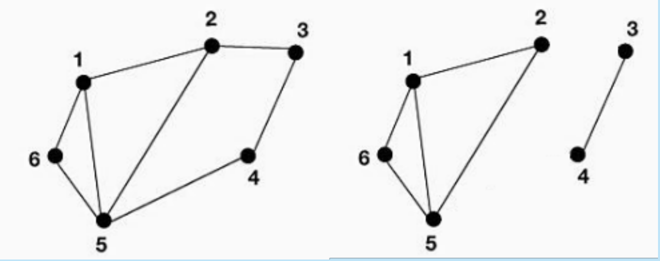




Corte de vértice e aresta

Definição

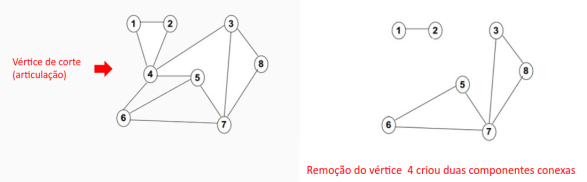
- **Corte de aresta** em $G(V,E)$ é um subconjunto **mínimo** de arestas E_1, sendo $E_1 \subseteq E$, tal que se as arestas de E_1 forem removidas de G, o grafo fica desconexo.



Vértice de corte

Definição

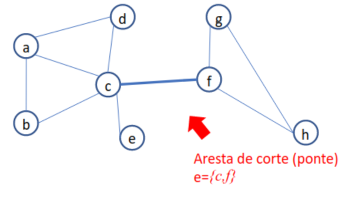
- Um vértice em um grafo G é dito **vértice de corte ou ponto de articulação**($\mathbf{v_j}$) se sua remoção (juntamente com as arestas a ele conectadas) provoca um redução na conexidade, ou seja, torna grafo desconexo.
  - Produz um grafo com mais componentes conexas que o grafo original.
  - $Componentes(G) < Componentes(G - v_j)$



Aresta de Corte

Definição

- Uma aresta em G é dita **aresta de corte ou ponte** se, ao ser retirada de G, ele torna-se desconexo.



Do Desenho ao Código

- Como traduzimos a ideia de um grafo para uma estrutura que o computador entenda? A escolha da representação é **crítica** e afeta diretamente a eficiência (tempo e memória) dos nossos algoritmos.

As Três Grandes Representações

1. **Matriz de Adjacências:** Ótima para verificar rapidamente se uma aresta existe.
2. **Lista de Adjacências:** A mais comum e eficiente em memória para grafos com poucas arestas.
3. **Matriz de Incidência:** Mais especializada, útil para certas análises de rede.

Matriz de Adjacências

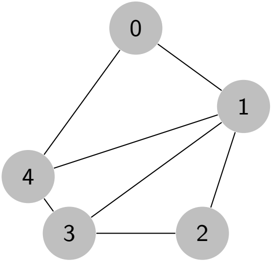
- Representamos o grafo como uma matriz quadrada M de dimensão $|V| \times |V|$.
- Se os vértices são numerados de 0 a $|V| - 1$, a entrada $M[i][j]$ da matriz nos diz sobre a aresta entre o vértice i e o vértice j.

Regra (para grafos não-ponderados)

$$M[i][j] = \begin{cases} 1, & \text{se existe a aresta } (i, j) \\ 0, & \text{caso contrário} \end{cases}$$

- Para grafos ponderados, armazenamos o peso da aresta em vez de 1.
- Para grafos **não-direcionados**, a matriz é sempre **simétrica** ($M[i][j] = M[j][i]$).

Exemplo: Matriz de Adjacências



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

Análise

- **Espaço:** $O(|V|^2)$. Ineficiente para grafos com poucas arestas (esparsos).
- **Verificar Aresta** (u, v)**:** $O(1)$. Excelente!
- **Listar Vizinhos de** u**:** $O(|V|)$. Precisa varrer uma linha inteira.

- Matriz adj é **simétrica** para Grafos não direcionados
- Dígrafo
  - Para (v_i, v_j) partindo de v_i e chegando em v_j, somente a posição M_{ij} será preenchida com 1
- Espaço de armazenamento das informações
  - $O(|V|^2)$
  - Indicado para **grafos densos**
    - Número de arestas é próximo a $|V|^2$
  - Desvantagem em **grafos esparsos**
    - Número de arestas é bem menor que $|V|^2$

Implementação em C - Matriz de adjacencias

Python Code

C Code

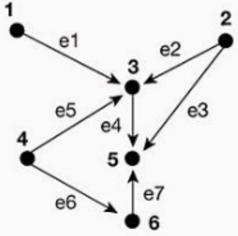
Definição

- Dado $G(V,E)$ de $|V|$ vértices e $|E|$ arestas, a matriz de incidência $M = |V| \times |E| = b_{ij}$, é definida:

$$b_{ij} = \begin{cases} 1, & se\ aresta\ j\ sai\ do\ vertice\ i \\ -1, & se\ aresta\ j\ entra\ no\ vertice\ i \\ 0, & em\ caso\ contrário \end{cases} \tag{1}$$

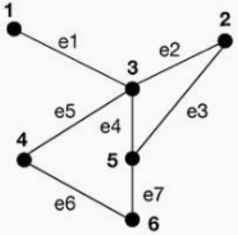
Representação computacional

Matriz de Incidências



	e1	e2	e3	e4	e5	e6	e7
1	1	0	0	0	0	0	0
2	0	1	1	0	0	0	0
3	−1	−1	0	1	−1	0	0
4	0	0	0	0	1	1	0
5	0	0	−1	−1	0	0	−1
6	0	0	0	0	0	−1	1

Dígrafo e sua Matriz de incidência



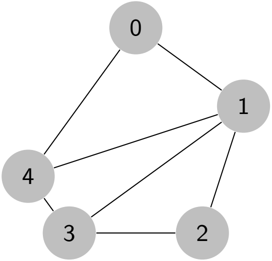
	e1	e2	e3	e4	e5	e6	e7
1	1	0	0	0	0	0	0
2	0	1	1	0	0	0	0
3	1	1	0	1	1	0	0
4	0	0	0	0	1	1	0
5	0	0	1	1	0	0	1
6	0	0	0	0	0	1	1

Grafo e sua Matriz de incidência

Lista de Adjacências

- A representação mais popular para a maioria das aplicações.
- Consiste em um array (ou vetor) de $|V|$ listas.
- Para cada vértice u, a entrada $Adj[u]$ armazena uma lista de todos os vértices v que são adjacentes a u.

Exemplo: Lista de Adjacências



$0 : \rightarrow [1, 4]$

$1 : \rightarrow [0, 2, 3, 4]$

$2 : \rightarrow [1, 3]$

$3 : \rightarrow [1, 2, 4]$

$4 : \rightarrow [0, 1, 3]$

Análise

- **Espaço:** $O(|V| + |E|)$. Muito eficiente para grafos esparsos.
- **Verificar Aresta** (u, v)**:** $O(deg(u))$. Precisa percorrer a lista de u.
- **Listar Vizinhos de** u**:** $O(deg(u))$. Excelente, o custo é proporcional ao número de vizinhos.

Implementação em C - Lista de adjacencias

Python Code

C Code

Introdução

- Objetivo mostrar um **procedimento sistemático** de como passear pelos vértices e arestas de um grafo.
- **marcar um vértice**
  - Evitar repetição
  - Distinguir daqueles que não foram processados
- Ideia geral
  - Dado $G(V, E)$ conexo e todos vértices desmarcados
  - Inicialmente marque u qualquer e $(u, v) \in E$ não foi selecionada
  - aresta (u, v) foi selecionada e o vértice v marcado

- Algoritmo de busca em profundidade (*Depth-First Search - (DFS*))
- Algoritmo de busca em largura (*Breadth-First Search (BFS))*

- Algoritmo que atende critério para explorar um vértice marcado
  - *Dentre os vários marcados e incidentes a alguma aresta ainda não explorada, escolher aquele vértice alcançado por último na busca*
- Arquétipo de muitos Algoritmos importantes
  - Algoritmo de Dijkstra.
  - Algoritmo de árvore espalhada mínima de Prim.

- **Ideia Central:** Explorar o grafo "camada por camada" a partir de um vértice de origem.
- Primeiro, visita todos os vizinhos diretos (distância 1), depois os vizinhos dos vizinhos (distância 2), e assim por diante.

Como Funciona?

Utiliza uma **Fila (Queue)** para controlar a ordem de visitação.

1. Comece em um vértice s, marque-o como visitado e coloque-o na fila.
2. Enquanto a fila não estiver vazia:
   2.1 Retire um vértice u da frente da fila.
   2.2 Para cada vizinho v de u que ainda não foi visitado:
      - Marque v como visitado.
      - Coloque v no final da fila.

Principal Aplicação

A BFS é garantida para encontrar o **caminho mais curto** (em número de arestas) entre a origem s e qualquer outro vértice em um grafo **não-ponderado**.

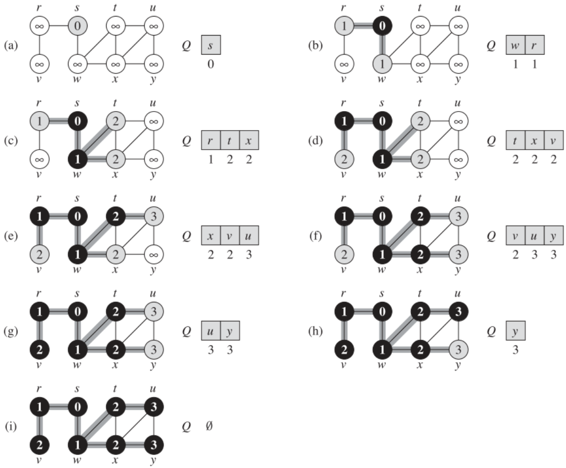
- Dado $G(V, E)$ conexo e um vértice de origem **s**, a busca em largura explora as arestas de G até explorar todos os vértices alcançáveis a partir de **s**.
- Calcula menor distância em termos de número de arestas de **s** com relação a todos os vértices de $G(V, E)$
- Origem do nome é devido a descobrir todos os vértices que se encontram a uma distância **k** de **s**, antes de descobriri os vértices que se encontram a $\mathbf{k+1}$
  - Expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da fronteira

- Produz uma **árvore** com raiz **s**
- Computa *caminho mais curto* de **s** até **v**
  - Número mínimo de arestas
- Aplicado em grafo direcionado ou não direcionado

Algoritmo

- Cada vértice é colorido de **branco, cinza ou preto**.
- Todos os vértices são inicializados **branco**
- Quando um vértice é descoberto pela primeira vez ele torna-se **cinza**
- Vértices **Cinza** e **Preto** já foram descobertos, mas são distinguidos para assegurar que a busca ocorra em largura.
- Se $(u, v) \in E$ e o vértice u é preto, então o vértice v tem que ser cinza ou preto
- Vértices **Cinza** podem ter vértices adjacentes **brancos**, e eles representam a **fronteira** entre vértices descobertos e não descobertos

```
BFS(G, s)
for cada vértice u ∈ V[G] − {s}
    do cor[u] ← BRANCO
       d[u] ← ∞
       α[u] ← NIL
cor[s] ← CINZA
d[s] ← 0
α[s] ← NIL
Q ← 0
ENQUEUE(Q, s)
 while Q ≠ 0
    do u ← DEQUEUE(Q)
       for cada v ← Adj[u]
           do if cor[v] = BRANCO
                 then cor[v] ← CINZA
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
       cor[u] ← PRETO
```



Execução do algoritmo busca em largura

Análise

- O teste da linha 13 assegura que cada vértice é colocado na fila no máximo uma vez.
- Operações de enfileirar e desenfileirar têm custo O(1), logo, o custo total com a fila é $O(|V|)$.
- Cada lista de adjacências é percorrida no máximo uma vez, quando o vértice é desenfileirado.
- Desde que a soma dos comprimentos de todas as listas adj é $\Theta(|E|)$, o tempo total gasto com as listas de adj é $O(|E|)$
- tempo total do BSF $O(|V| + |E|)$

```
BFS(G, s)
 1  for cada vértice u ∈ V[G] – {s}
 2       do cor[u] ← BRANCO
 3          d[u] ← ∞
 4          α[u] ← NIL
 5  cor[s] ← CINZA
 6  d[s] ← 0
 7  α[s] ← NIL
 8  Q ← 0
 9  ENQUEUE(Q, s)
10   while Q ≠ 0
11      do u ← DEQUEUE(Q)
12         for cada v ← Adj[u]
13             do if cor[v] = BRANCO
14                   then cor[v] ← CINZA
15                        d[v] ← d[u] + 1
16                        π[v] ← u
17                        ENQUEUE(Q, v)
18         cor[u] ← PRETO
```

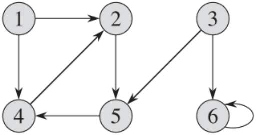
Implementação - Busca em Largura

Python Code

C Code

Busca em Largura - Exercício I

- Mostre os valores de **d** e da fila que resultam da execução da busca em largura sobre o grafo ao lado, usando o vértice **3** como origem



- A estratégia é buscar o mais profundo no grafo sempre que possível.
- As arestas são exploradas a partir do vértice **v** mais recentemente descoberto que ainda possui arestas não exploradas saindo dele.
- Quando todas as arestas adjacentes a **v** tiverem sido exploradas a busca anda para trás para explorar vértices que saem do vértice do qual v foi descoberto.
- O algoritmo é a base para muitos outros algoritmos importantes, tais como verificação de grafos acíclicos, ordenação topológica e componentes fortemente conectados.

Ideia Central

- Para acompanhar o progresso do algoritmo cada vértice é **colorido** de branco, cinza ou preto
- Todos os vértices são inicializados branco
- Quando um vértice é descoberto pela primeira vez ele torna-se **cinza**, e é tornado **preto** quando sua lista de adjacentes tenha sido completamente examinada.
- **Carimbo de Tempo**
  - $d[v]$ tempo de descoberta
  - $f[v]$ tempo de término do exame da lista de adjacentes de **v**
  - Estes registros são inteiros entre 1 e $2|V|$, pois existe um evento de descoberta e um evento de término para cada um dos $|V|$ vértices

Ideia Central

- Sempre que um vértice **v** é descoberto durante uma varredura da lista adj de um vértice já descoberto **u**
  - algoritmo registra esse evento definindo como atributo **predecessor** de **v** como $\pi[\mathbf{v}]$
  - $\pi[\mathbf{v}] = u$

  subgrafo predecessor $G_\pi = (V, E_\pi)$

  - $E_\pi = \{(\pi[v], v) : v \in V \text{ e } \pi[v] \neq \emptyset\}$
  - DFS forma várias árvores (floresta) devido busca pode ser repetida a partir de diversas origens.

Algoritmo - Busca em Profundidade

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)

DFS-VISIT(G, u)
time = time + 1              // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]        // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK              // blacken u; it is finished
time = time + 1
u.f = time
```

```
DFS(G)
for cada vértice u ← V[G]
    do cor[u] ← BRANCO
       π[u] ← NIL
tempo ← 0
for cada vértice u ∈ V[G]
    do if cor[u] = BRANCO
          then DFS-VISIT(u)

DFS-VISIT(u)
cor[u] ← CINZA               ▷ Branco, o vértice u acabou de se
tempo ← tempo + 1
d[u] ← tempo
for cada v ∈ Adj[u]          ▷ Explora a aresta (u, v).
    do if cor[u] = BRANCO
          then π[v] ← u
               DFS-VISIT(v)
cor[u] ← PRETO               ▷ Enegrece u; terminado.
f[u] ← tempo ← tempo + 1
```

Implementação em C - Busca em Profundidade

```c
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 100

typedef struct {
    int matriz[MAX_VERTICES][MAX_VERTICES];
    int numVertices;
} Grafo;
```

Implementação em C - Busca em Profundidade

```
void inicializarGrafo(Grafo* grafo, int numVertices) {
    int i, j;

    grafo->numVertices = numVertices;

    for (i = 0; i < numVertices; i++) {
        for (j = 0; j < numVertices; j++) {
            grafo->matriz[i][j] = 0;
        }
    }
}
```

Implementação em C - Busca em Profundidade

```
void adicionarAresta(Grafo* grafo, int origem, int destino) {
    grafo->matriz[origem][destino] = 1;
    // Se for um grafo nao direcionado, descomente a linha abaixo
    // grafo->matriz[destino][origem] = 1;
}
```

Implementação em C - Busca em Profundidade

```c
void DFSVisitar(Grafo* grafo, int vertice, bool visitado[]) {
    visitado[vertice] = true;
    printf("%d ", vertice);

    int i;
    for (i = 0; i < grafo->numVertices; i++) {
        if (grafo->matriz[vertice][i] == 1 && !visitado[i]) {
            DFSVisitar(grafo, i, visitado);
        }
    }
}
```

Implementação em C - Busca em Profundidade

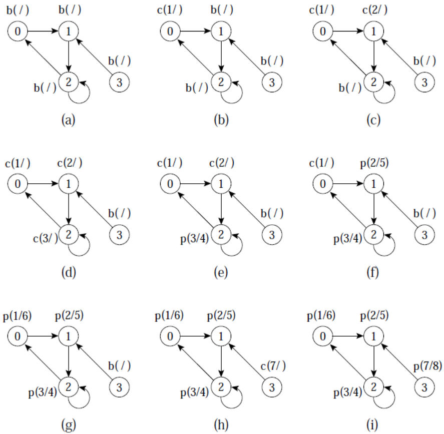
```c
void buscaEmProfundidade(Grafo* grafo, int verticeInicial) {
    bool visitado[MAX_VERTICES];
    int i;

    for (i = 0; i < grafo->numVertices; i++) {
        visitado[i] = false;
    }

    DFSVisitar(grafo, verticeInicial, visitado);
}
```

Implementação em C - Busca em Profundidade

```
int main() {
    Grafo grafo;
    int numVertices = 5;
    inicializarGrafo(&grafo, numVertices);
    adicionarAresta(&grafo, 0, 1);
    adicionarAresta(&grafo, 0, 4);
    adicionarAresta(&grafo, 1, 2);
    adicionarAresta(&grafo, 1, 3);
    adicionarAresta(&grafo, 1, 4);
    adicionarAresta(&grafo, 2, 3);
    adicionarAresta(&grafo, 3, 4);

    int verticeInicial = 0;
    printf("BFS a partir do vertice %d: ", verticeInicial);
    buscaEmProfundidade(&grafo, verticeInicial);

    return 0;
}
```

```
class DFS:
    def __init__(self,grafo):
        self.grafo= grafo;

    def run(self):
        self.cor = ['BRANCO'] * (len(self.grafo))
        self.f = [0] * (len(self.grafo))
        self.d= [0] * (len(self.grafo))
        self.time = 0
        for u in self.grafo.listVertices():
            if(self.cor[self.grafo.indexOfVertice(u)]=='BRANCO'):
                self.DFS_visit(self.grafo,u);
        for v,d,f in zip(self.grafo.listVertices(),self.d,self.f):
            print("%s␣(%s/%s)" % (v,d,f))

    def DFS_visit(self,grafo,u):
        self.time = self.time + 1
        self.d[self.grafo.indexOfVertice(u)] = self.time
        self.cor[self.grafo.indexOfVertice(u)] = 'CINZA'
        for vertice_adj in self.grafo.listAdjOf(u):
            v = self.grafo.indexOfVertice(vertice_adj)
            if self.cor[v] == 'BRANCO':
                self.DFS_visit(grafo,vertice_adj)

        self.cor[self.grafo.indexOfVertice(u)] = 'PRETO'
        self.time = self.time + 1
        self.f[self.grafo.indexOfVertice(u)] = self.time
```

Algoritmo - Busca em Profundidade



Implementação em Python

```
if __name__ == '__main__':
g = GrafoMatrix()
for v in ["0","1", "2", "3"]:
    g.add_vertice(v)

g.add_aresta("0", "1")
g.add_aresta("1", "2")
g.add_aresta("2", "0")
g.add_aresta("2", "2")
g.add_aresta("3", "1")
print("Seguinte Percurso usando DFS")
dfs = DFS(grafo=g)
dfs.run()
```

Resultado

Seguinte Percurso usando DFS
0(1/6)
1(2/5)
2(3/4)
3(7/8)

Análise - Busca em Profundidade

- Os dois *loops* possuem tempo $\Theta(V)$
- O procedimento DFS_VISIT é chamado exatamente uma vez para cada vértice $v \in V$, uma vez que o método chamado somente para vértices brancos e a primeira ação é pintar de cinza.
- No loop principal (4-7), o DFS_VISIT é executado $|adj|v||$ vezes
  - Custo de tempo é $\sum_{v \in V} |Adj[v]| = \Theta(E)$
- Tempo total $\Theta(V + E)$

Classificação das Arestas - Busca em Profundidade

Classificação de arestas pode ser útil para derivar outros algoritmos

Arestas de árvore são arestas de uma árvore de busca em profundidade. A aresta $(\mathbf{u}, \mathbf{v})$ é uma aresta de árvore se **v** foi descoberto pela primeira vez ao percorrer a aresta $(\mathbf{u}, \mathbf{v})$

Arestas de retorno conectam um vértice **u** com um antecessor **v** em uma árvore de busca em profundidade (inclui *self-loops*).

Arestas de avanço são as arestas $(\mathbf{u}, \mathbf{v})$ que não pertencem à árvore de busca mas conectam um vértice **u** a um descendente **v** em uma árvore de busca.

Arestas de cruzamento podem conectar vértices na mesma árvore de busca em profundidade, ou em duas árvores diferentes

Busca em Profundidade - Trabalho I

- A busca em profundidade pode ser usada para verificar se um grafo é acíclico ou contém um ou mais ciclos.
- Se uma aresta de retorno é encontrada durante a busca em $G(V, E)$, então o grafo tem ciclo
- Um grafo direcionado $G(V, E)$ é acíclico se e somente se a busca em profundidade em $G(V, E)$ não apresentar arestas de retorno
- O algoritmo pode ser alterado para descobrir arestas de retorno. Basta verificar se um vértice **v** adjacente a **u** apresenta cor cinza na primeira vez que a aresta $(\mathbf{u}, \mathbf{v})$ é pecorrida

1. Faça uma implementação para verficar se um $G(V, E)$ é acíclico.

O Problema do Caminho Mínimo

- **Objetivo:** Dado um grafo **ponderado** com pesos não-negativos, um vértice de origem s e um de destino t, encontrar o caminho de s a t com a menor soma de pesos das arestas.
- Esta é uma das aplicações mais famosas e úteis de grafos, sendo a base para sistemas de GPS e roteamento de redes.

Algoritmo de Dijkstra

- É um algoritmo **guloso (greedy)** que resolve o problema do caminho mínimo de uma única origem para todos os outros vértices.
- **Ideia principal:** É uma generalização da BFS. Em vez de uma fila normal, usa uma **Fila de Prioridade (Priority Queue)** para sempre escolher o próximo vértice a ser explorado como aquele com a menor distância *conhecida* da origem.
- **Restrição Crucial:** O algoritmo de Dijkstra **não funciona corretamente** se o grafo tiver arestas com pesos negativos.

Como Dijkstra Funciona (Intuitivamente)

1. Crie um conjunto de "distâncias" para todos os vértices, inicializando a distância da origem s como 0 e de todos os outros como infinito.
2. Adicione a origem s a uma fila, onde a **prioridade** é a distância.
3. Mantenha um conjunto de vértices já "finalizados" (cujo caminho mínimo já foi encontrado), inicialmente vazio.
4. Enquanto a fila de prioridade não estiver vazia:
   4.1 Extraia o vértice u com a menor distância da fila. Adicione u ao conjunto de finalizados.
   4.2 Para cada vizinho v de u:
      - Calcule uma nova distância para v através de u: $dist(s, u) + peso(u, v)$.
      - Se essa nova distância for menor que a distância atual de v, atualize a distância de v e sua prioridade na fila. Este passo é chamado de **relaxamento (relax)**.

- Ao final, o array de distâncias conterá o custo do caminho mínimo de s para cada outro vértice.

Problema dos Caminhos Mais Curtos

- Modelagem
  - $G = (V, E)$: Grafo direcionado ponderado.
  - Função Peso $\mathbf{w} : E \rightarrow \mathbb{R}$
    - $w(u, v)$: peso de cada aresta
  - **Peso** de um caminho $\mathbf{p} = \langle \mathbf{v_0}, \mathbf{v_1}, ..., \mathbf{v_k} \rangle$ é somatório dos pesos de suas arestas:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- Definimos **peso do caminho mais curto** desde u até v por:

$$\delta(u, v) = \begin{cases} min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{\textit{se existir caminho de u a v}} \\ \infty, & \text{\textit{caso contrário}} \end{cases} \quad (2)$$

- **Caminho mais curto** do vértice **u** ao vértice **v** é então definido como qualquer caminho **p** com peso $\mathbf{w(p)} = \delta(\mathbf{u}, \mathbf{v})$

Problema dos Caminhos Mais Curtos

- Modelagem de Problema
  - Saber rota mais curta entre Belém e Paraupebas
  - V: Estações
  - E: Segmentos de Estrada
  - w: **distâncias** entre as estações
    - **Medidas**: Quantidade que se acumule linearmente ao longo de um caminho e que deseje minimizar
  - G(V,E): Mapa rodoviário
- Algoritmo de Busca em Largura será nossa referência.
- **Caminhos mais curtos a partir de uma origem**
  - Dado um grafo ponderado $G = (V, E)$, desejamos obter o caminho mais curto a partir de um dado vértice origem $s \in V$ até cada $v \in V$

Representação de Caminhos Mais Curtos

- A representação de caminhos mais curtos pode ser realizada pela variável **predecessor** π
- Para cada vértice $v \in V$ o predecessor $\pi[\mathbf{v}]$ é outro vértice ou *Null*
- O algoritmo atribui ao predecessor os rótulos de vértices de uma cadeia de predecessores com origem em v e que anda para trás ao longo de um caminho mais curto desde s até v
- Os valores em $\pi[\mathbf{v}]$, em um passo intermediário, não indicam necessariamente caminhos mais curtos
- Entretanto, ao final do processamento, predecessor contém uma árvore de caminhos mais curtos definidos **em termos dos pesos de cada aresta** de G, ao **invés do número de arestas**.

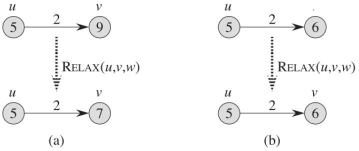
Algoritmo de Dijkstra

- Mantém um conjunto S de vértices cujos caminhos mais curtos até um vértice origem já são conhecidos.
- Produz uma árvore de caminhos mais curtos de um vértice origem s para todos os vértices que são alcançáveis a partir de s.
- Utiliza **Relaxamento**
  - Para cada $v \in V$ o atributo **d**[**v**] é limite superior sobre peso de um caminho mais curto desde origem s até v
  - O vetor **d**[**v**] contém uma estimativa de um caminho mais curto
- O primeiro passo do algoritmo é inicializar os predecessores e as estimativas de caminhos mais curtos
  - Método *INITIALIZE-SINGLE-SOURCE(G,S)*

Relaxamento

- *O relaxamento de uma aresta (u, v) consiste em verificar se é possível melhorar o melhor caminho até* **v** *obtido até o momento se passarmos por* **u**.
    - Uma etapa de relaxamento pode diminuir $d[v]$ e atualizar campo $\pi[v]$.
    - No Dijkstra as arestas são relaxadas somente 1 vez.

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```



Método *Relax*

Algoritmo de Dijkstra

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0

RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u

DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

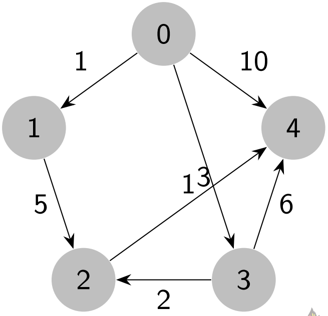
```
INITIALIZE-SINGLE-SOURCE(G, s)
for cada vértice v ∈ V[G]
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0

RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
  then d[v] ← d[u] + w(u, v)
       π[v] ← u

DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S ← ∅
Q ← V[G]
while Q ≠ ∅
   do u ← EXTRACT-MIN(Q)
      S ← S ∪ {u}
      for cada vértice v ∈ Adj[u]
          do RELAX(u, v, w)
```

Dijkstra Passo a Passo: Início no Vértice 0

- O conjunto de vértices visitados **S** começa vazio.
- A Fila de Prioridade **Q** contém todos os vértices.
- A distância $d[0]$ para a origem é **0**.
- Todas as outras distâncias $d[v]$ são inicializadas com ∞.

Vértice	**d[v]**	π**[v]**
0	0	NULL
1	∞	NULL
2	∞	NULL
3	∞	NULL
4	∞	NULL

$$\mathbf{S} = \{\}$$

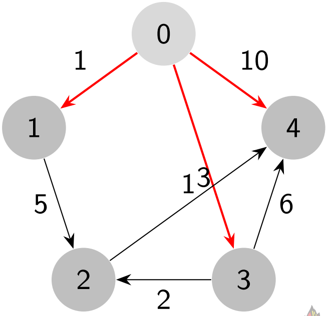


Dijkstra Passo a Passo: Início no Vértice 0

- Extraímos de Q o vértice com menor distância: **0**.
- Adicionamos **0** ao conjunto S.
- **Relaxamos** as arestas que saem de 0:
  - (0,1): $d[1] = \min(\infty, 0 + 1) = 1$
  - (0,3): $d[3] = \min(\infty, 0 + 3) = 3$
  - (0,4): $d[4] = \min(\infty, 0 + 10) = 10$

Vértice	**d[v]**	π**[v]**
0	0	NULL
1	1	0
2	∞	NULL
3	3	0
4	10	0

$\mathbf{S} = \{\mathbf{0}\}$

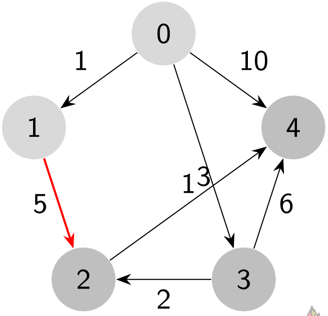


Dijkstra Passo a Passo: Início no Vértice 0

- Próximo vértice de Q com menor distância: **1** (d=1).
- Adicionamos **1** ao conjunto S.
- **Relaxamos** a aresta (1,2):
  - $d[2] = \min(\infty, d[1] + 5) = \min(\infty, 1 + 5) = 6$

Vértice	d[v]	π[v]
0	0	NULL
1	1	0
2	6	1
3	3	0
4	10	0

$\mathbf{S} = \{\mathbf{0}, \mathbf{1}\}$

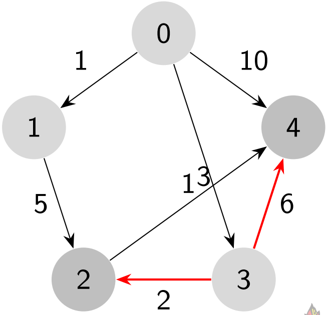


Dijkstra Passo a Passo: Início no Vértice 0

- Próximo vértice de Q: **3** (d=3).
- Adicionamos **3** ao conjunto S.
- **Relaxamos** as arestas que saem de 3:
  - (3,2): $d[2] = \min(6, d[3] + 2) = \min(6, 3 + 2) = 5$
  - (3,4): $d[4] = \min(10, d[3] + 6) = \min(10, 3 + 6) = 9$

Vértice	d[v]	π[v]
0	0	NULL
1	1	0
2	5	3
3	3	0
4	9	3

$\mathbf{S} = \{\mathbf{0}, \mathbf{1}, \mathbf{3}\}$

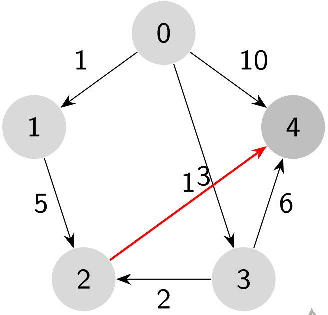


Dijkstra Passo a Passo: Início no Vértice 0

- Próximo vértice de Q: **2** (d=5).
- Adicionamos **2** ao conjunto S.
- **Relaxamos** a aresta (2,4):
  - $d[4] = \min(9, d[2] + 1) = \min(9, 5 + 1) = 6$

Vértice	**d[v]**	π**[v]**
0	0	NULL
1	1	0
2	5	3
3	3	0
4	6	2

$\mathbf{S} = \{\mathbf{0}, \mathbf{1}, \mathbf{3}, \mathbf{2}\}$

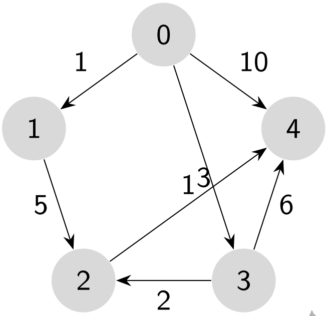


Dijkstra Passo a Passo: Início no Vértice 0

- Último vértice de Q: **4** (d=6).
- Adicionamos **4** ao conjunto S.
- Não há arestas para relaxar, pois não há vizinhos fora de S.
- A fila Q fica vazia. O algoritmo termina.

Vértice	**d[v]**	π**[v]**
0	0	NULL
1	1	0
2	5	3
3	3	0
4	6	2

$$\mathbf{S} = \{\mathbf{0}, \mathbf{1}, \mathbf{3}, \mathbf{2}, \mathbf{4}\}$$



Algoritmo de Dijkstra

- Invariante: o número de elementos da fila de prioridade (Heap) **Q** é igual a $V - S$ no início do loop *while*.
- A cada iteração do *while*, um vértice **u** é extraído da fila e adicionado ao conjunto **S**, mantendo assim o invariante.
- **EXTRACT-MIN** obtém o vértice **u** com o caminho mais curto estimado até o momento e adiciona ao conjunto **S**.
- No loop da linha 7-8, a operação de relaxamento é realizada sobre cada aresta $(\mathbf{u}, \mathbf{v})$ adjacente ao vértice **u**

Algoritmo de Dijkstra

- Para realizar de forma eficiente a seleção de uma nova aresta, todos os vértices que não estão na árvore de caminhos mais curtos residem na fila **Q** baseada no campo **d**.
- Para cada vértice **v**, **d**[**v**] é o caminho mais curto obtido até o momento, de **v** até o vértice raiz.
- A fila mantém os vértices, mas a condição da fila é mantida pelo caminho mais curto estimado até o momento através do arranjo **d**[**v**], a fila é indireta.

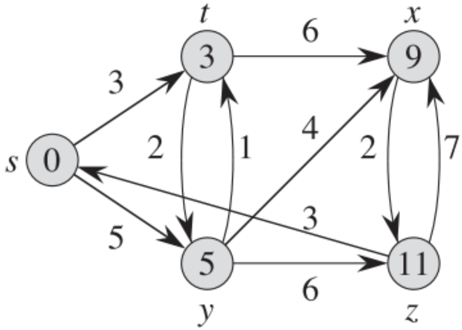
Algoritmo de Dijkstra

- O algoritmo usa uma estratégia **gulosa**: sempre escolher o vértice mais leve (ou o mais perto) em $V - S$ para adicionar ao conjunto solução **S**
- O algorimo de Dijkstra sempre obtém os caminhos mais curtos, pois cada vez que um vértice é adicionado ao conjunto **S** temos que $d[u] = \delta(s, u)$.

Algoritmo de Dijkstra - Trabalho II

1. Execute o algoritmo de Dijkstra sobre o grafo orientado do slide 91, primeiro usando o vértice **s** como origem, e depois usando o vértice **z** como origem. Imprima os valores **d** e os vértices no conjunto **S** após cada iteração do loop *while*
2. Dado um digrafo D sem ciclos direcionados, formular um algoritmo para encontrar o comprimento do maior caminho em D. A matriz de distância W de D é formada por distâncias não negativas.

Algoritmo de Dijkstra - Trabalho II



Referências I

- Lee K.D., Hubbard S. (2015) Trees. In: Data Structures and Algorithms with Python. Undergraduate Topics in Computer Science. Springer, Cham. Retrieved from https://doi.org/10.1007/978-3-319-13072-9_6
- Hubbard, J. (2007). Schaum's Outline sof Data Structures with Java. Retrieved from http://www.amazon.com/Schaums-Outline-Data-Structures-Java/dp/0071476989
- Cormen, T. H., Leiserson, C. E., & Stein, R. L. R. E. C. (2012). Algoritmos: teoria e prática. Retrieved from https://books.google.com.br/books?id=6iA4LgEACAAJ.
- Ascenio, Ana Fernanda Gomes. Estrutura de dados: Algoritmos, análise da complexidade e implementações em Java e C++. São Paulo: Pearson Prentice Hall, 2010.

Referências II

- Donald E. Knuth. Sorting and Searching, volume 3 of The Art ofComputer Programming. Addison-Wesley, 1973. Second edition, 1998
- Szwarcfiter, Jayme Luiz. Estruturas de dados e seus algoritmos / Jayme Luiz Szwarcfiter, Lilian Markenzon. 3.ed. [Reimpr.]. - Rio de Janeiro : LTC, 2015.
- Learning about defaultdict in Python. Retrieved from `https://www.educative.io/edpresso/learning-about-defaultdict-in-python`.
- How to implement a graph in Python. Retrieved from `https://www.educative.io/edpresso/how-to-implement-a-graph-in-python`.

Grafos

Professor: Elton Sarmanho[1]
E-mail: eltonss@ufpa.br

[1]Faculdade de Sistemas de Informação - UFPA/CUTINS

7 de outubro de 2025