Grafos

Oficinas de Programação Competitiva

Prof. Daniel Saad Nogueira Nunes

Instituto Federal de Brasília, Câmpus Taguatinga

Sumário

Sumário

Introdução

- Muitos problemas em Ciência da Computação são modelados em formas de relacionamento entre objetos, no sentido amplo da palavra.
- Precisamos de formalismo que consegue modelar relações presentes desde problemas envolvendo interações entre pessoas à problemas envolvendo redes gigantescas de computadores.
- A chave para resolução de muitos problemas computacionais pode residir em um único formalismo, os **grafos**.

Introdução

- A Teoria dos Grafos provê uma linguagem para falar de propriedades e relacionamentos dos objetos mencionados.
- Veremos que projetar um algoritmo novo usando grafos é extremamente complicado, muita das vezes, precisamos apenas utilizar um algoritmo já conhecido.
- Às vezes o mais difícil é modelar o problema em termos de grafos!

Introdução

Definição (Grafo)

Um grafo é uma dupla $G(V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas.

Introdução

- Repare que as arestas formam uma relação sobre o conjunto dos pares de vértices.
- Por exemplo, os vértices $v \in V$ poderiam representar cidades, enquanto uma aresta (u, v) informaria que estive uma rodovia entre a cidade u e v.
- As arestas representam relacionamentos entre os objetos!

Introdução

Tipos de Grafos

- Existem diversas especialidades de grafos.
- Cada qual com suas propriedades distintas, o que faz o seu uso mais adequado em determinados problemas:
  1. Simples × Não-simples.
  2. Dirigido × Não-dirigido.
  3. Com peso × Sem peso.
  4. Esparso × Denso.
  5. Cíclico × Acíclico.
  6. Incorporado × Topológico.
  7. Implícito × Explícito.
  8. Rotulado × Não-rotulado.

Sumário

Tipos de Grafos

Simples $\times$ Não-simples

- Grafos simples não possuem estruturas complexas, tais como:
  - Loops: arestas que ligam o vértice nele mesmo.
  - Multiarestas: podemos ter várias arestas ligando dois vértices.

Tipos de Grafos

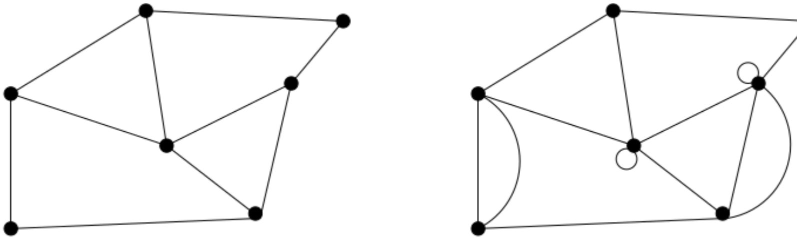


Figura: Simples × Não-simples.

Tipos de Grafos

Dirigido $\times$ Não-dirigido

- Um grafo é não-dirigido se, existe uma aresta (x, y), logo, também existe a aresta (y, x), temos arestas nas duas direções.
- Um grafo é dirigido se podemos ter arestas em uma única direção.
  - Muito útil para modelas problemas específicos.
  - Exemplo: uma via de mão única.

Tipos de Grafos

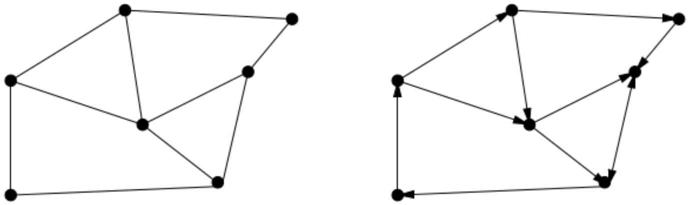


Figura: Dirigido × Não-dirigido.

Tipos de Grafos

Com Peso × Sem Peso

- Em um grafo com peso nas areas, para cada aresta (u, v), temos um peso relacionado a ela. que pode ser por exemplo números inteiros ou reais.
    - Muito utilizado em problemas de otimização, como o problema do menor caminho.

Tipos de Grafos

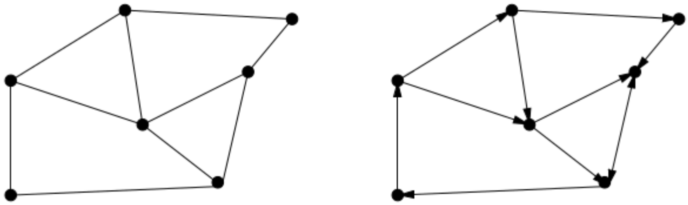


Figura: Dirigido × Não-dirigido.

Tipos de Grafos

Esparso $\times$ Denso

- Ao todo podemos ter $\binom{n}{2}$ pares de vértices.
- Grafos são esparsos se temos apenas uma pequena fração de arestas sobre os possíveis pares de vértice
- Grafos densos possuem uma grande porção de ligações entre os vértices.
- Não há uma regra geral, geralmente dizemos que um grafo é denso se $|E| \in \Theta(n^2)$.

Tipos de Grafos

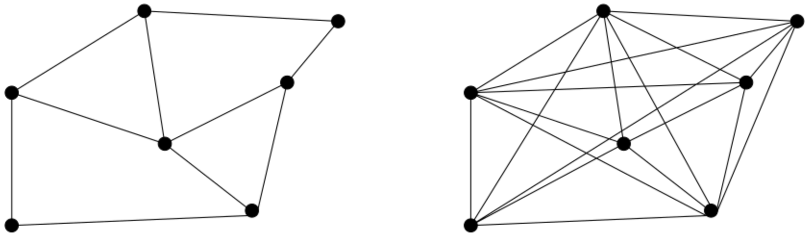


Figura: Esparso × Denso.

Tipos de Grafos

Cíclicos × Acíclicos

- Um grafo acíclico são grafos que não possuem ciclos.

Tipos de Grafos

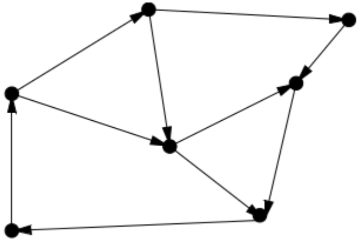


Figura: Cíclico $\times$ Acíclico.

Tipos de Grafos

Incorporado × Topológico

- Um grafo é incorporado se seus vértices estão mapeados em posições geométricas, como em um grid.
- Isso pode ter relevância em alguns problemas.
- Em problemas que isto não é importante, nos importamos apenas com a topologia do grafo (o esqueleto).

Tipos de Grafos

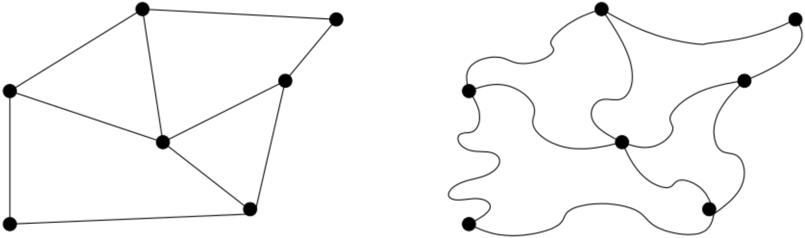


Figura: Incorporado × Topológico.

Tipos de Grafos

Implícitos × Explícitos

- Grafos ímplicitos vão sendo construídos conforme vamos utilizando eles.
  - Backtracking, simulação...
- Em outros casos, precisamos do grafo já construído para resolver certos problemas.

Tipos de Grafos

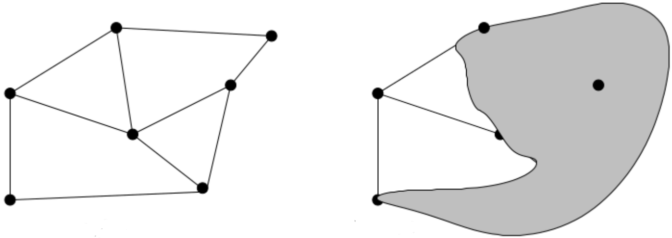


Figura: Implícito $\times$ Explícito.

Tipos de Grafos

Rotulado $\times$ Não rotulados

- Se o grafo é rotulado, a cada vértice é atribuído um rótulo que o identifica unicamente.
- Em grafos sem rótulo, não temos essa distinção.

Tipos de Grafos

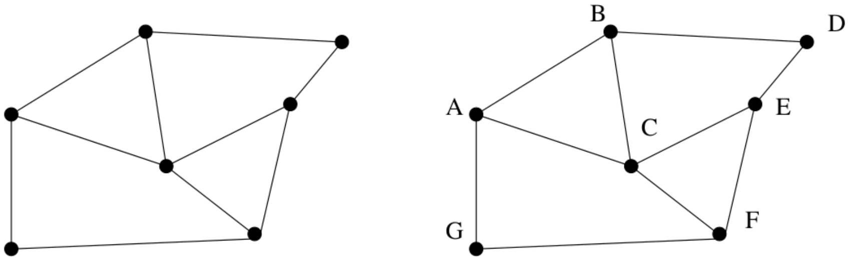


Figura: Rotulado × Não-rotulado.

Sumário

Aplicações

- Usando esse formalismo, podemos resolver problemas reais!
- Desde problemas biológicos como problemas em rede de computadores!

Aplicações

Figura: Navegação.

Aplicações

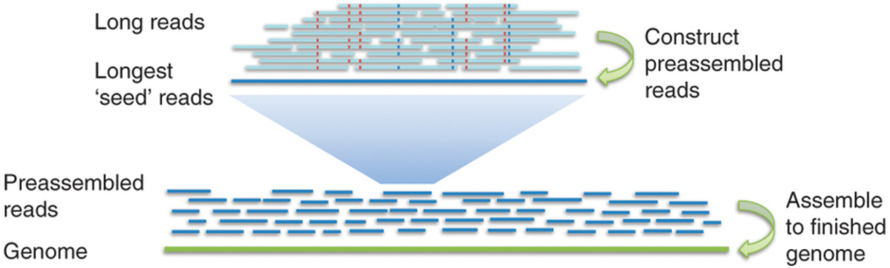


Figura: Montagem de Genomas.

Aplicações

Figura: Análise de Tráfego.

Aplicações

Figura: Redes de Computadores.

Aplicações

Exemplo

- Vamos começar nosso estudo desse incrível formalismo com uma modelagem simples.
- O grafo de relacionamento de pessoas!
  - Nosso Facebook.

Aplicações

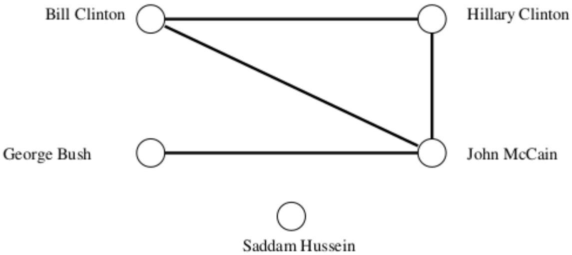


Figura: Grafo de relacionamento de pessoas.

Aplicações

Exemplo

- Através do grafo de relacionamento, podemos responder várias perguntas interessantes:
  - Meu amigo também me considera como amigo?
  - Qual é o nível da nossa amizade?
  - Eu sou amigo de mim mesmo?
  - Quem tem mais amigos?
  - Meus amigos moram perto de mim?
  - Você conhece esta pessoa?
  - Você é um indivíduo ou apenas um rosto?

Aplicações

Exemplo

- Meu amigo também me considera como amigo?

Aplicações

Exemplo

- Meu amigo também me considera como amigo?
- Existe uma aresta do seu amigo pra você?

Aplicações

Exemplo

- Qual é o nível da nossa amizade?

Aplicações

Exemplo

- Qual é o nível da nossa amizade?
- Quanto é o peso sobre a aresta que nos liga?

Aplicações

Exemplo

- Eu sou amigo de mim mesmo?

Aplicações

Exemplo

- Eu sou amigo de mim mesmo?
- O grafo é simples? Possuo um loop pra mim mesmo?

Aplicações

Exemplo

- Quem tem mais amigos?

Aplicações

Exemplo

- Quem tem mais amigos?
- Qual é o vértice que tem mais arestas saindo dele?

Aplicações

Exemplo

- Meus amigos moram perto de mim?

Aplicações

Exemplo

- Meus amigos moram perto de mim?
- Dado que o grafo é incorporado, qual a distância do seu vértice ao dos seus amigos?

Aplicações

Exemplo

- Você é um indivíduo ou apenas um rosto?

Aplicações

Exemplo

- Você é um indivíduo ou apenas um rosto?
- O grafo é rotulado?

Sumário

Representação de Grafos

- Como representar grafos computacionalmente?
- Temos que escolher uma representação eficiente.
- Escolhas mais comuns:
  1. Listas de Adjacências.
  2. Matrizes de Adjacências.

Representação de Grafos

Listas de Adjacência

- As listas de adjacências consistem de um vetor de tamanho $|V|$ de listas encadeada.
- Cada elemento do vetor, aponta para uma lista encadeada.
- Suponha o i-ésimo elemento deste vetor. Ele apontará para uma lista encadeada que contém as arestas que saem do nó i.

Listas de Adjacência

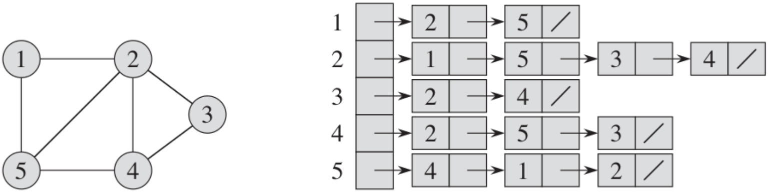


Figura: Lista de adjacências.

Listas de Adjacência

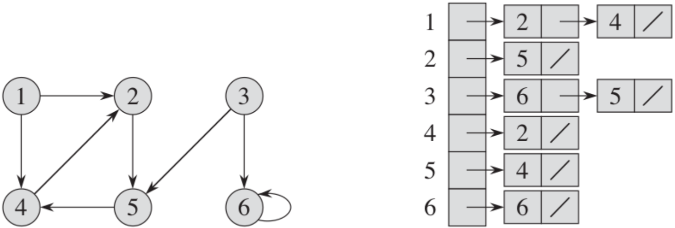


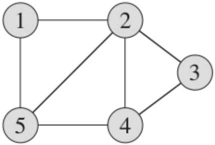
Figura: Lista de adjacências.

Representação de Grafos

Matrizes de Adjacências

- As matrizes de adjacências, como um nome diz, é uma matriz.
- O elemento $M[i][j]$, indica se existe uma aresta entre os nós i e j.

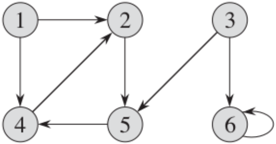
Matrizes de Adjacências



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

Figura: Matriz de Adjacências.

Matrizes de Adjacências



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

Figura: Matriz de Adjacências.

Listas vs Matrizes de Adjacências

- Cada abordagem tem seus pontos fortes e fracos.
- Listas de adjacência são mais econômicas em espaço quando o grafo é esparso.
- Matrizes de adjacência permitem acesso em tempo constante a qualquer aresta.
- Qual utilizar?

Listas vs Matrizes de Adjacências

Tabela: Comparação entre listas e matrizes de adjacências.

Critério	Ganhador
Tempo de acesso em arestas	Matriz
Verificar o grau do vértice	Lista
Consumo de memória em grafos esparsos	Lista
Consumo de memória em grafos densos	Matriz
Inserção/remoção de arestas	Matriz
Percurso do grafo	Listas

Sumário

Conceitos Fundamentais

Definição (Grau de Entrada)

- Definido sobre um nó v.
- Representa o número de arestas que chegam em um nó v.

Conceitos Fundamentais

Definição (Grau de Saída)

- Definido sobre um nó v.
- Representa o número de arestas que saem de um nó v.
- **OBS:** em um grafo não direcionado, o grau de entrada de cada vértice é igual ao grau de saída.

Conceitos Fundamentais

Definição (Caminho)

- Um caminho é uma sequência de arestas que conecta vértices distintos.

Conceitos Fundamentais

Definição (Conectividade)

- Um grafo não-dirigido é dito conexo se existe um caminho para qualquer dois pares de vértices.
- Um grafo com apenas um vértice também é considerado conexo.

Conceitos Fundamentais

Definição (Componente Conexa)

- Uma componente conexa de um grafo não dirigido é um subgrafo maximal conexo do grafo original.

Conceitos Fundamentais

Definição (Conectividade Fraca)

- Um grafo dirigido é dito fracamente conexo se ao trocarmos suas arestas pela versão não dirigida, obtemos um grafo conexo.

Conceitos Fundamentais

Definição (Conectividade Forte)

- Um grafo **dirigido** é dito fortemente conexo se para quaisquer par u e v de vértices, existe um caminho de u para v e um de v para u.

Conceitos Fundamentais

Definição (Corte)

- Um corte é um conjunto de vértices que separa o grafo, isto é, que o deixa com mais de uma componente conexa.

Conceitos Fundamentais

Definição (k-conectividade)

- Um grafo não-dirigido é dito k-conexo se não existe um conjunto de $k - 1$ vértices, que desconecta o grafo.

Sumário

2 Percurso

Percurso em Grafos

- Talvez o problema mais elementar em grafos seja percorrê-los.
- O percurso deve certificar de não visitar os mesmos nós várias vezes.
- Precisamos marcar o nó após visitá-lo, para não voltar nos mesmos.
- Duas estratégias elementares:
  1. Busca em largura (BFS - Breath-First-Search).
  2. Busca em profundidade (DFS - Depth-First-Search).

Percurso em Grafos

- Talvez o problema mais elementar em grafos seja percorrê-los.
- O percurso deve certificar de não visitar os mesmos nós várias vezes.
- Precisamos marcar o nó após visitá-lo, para não voltar nos mesmos.
- Duas estratégias elementares:
  1. Busca em largura (BFS - Breath-First-Search).
  2. Busca em profundidade (DFS - Depth-First-Search).

Sumário

2 Percurso
- Breath-First-Search
- Depth-First-Search
- Problemas

Busca em Largura

BFS

- Consiste em, a partir de um nó, descobrir todos os seus vizinhos.
- O procedimento é repetido para cada vizinho do nó original em ordem de descoberta.
- Cada nó visitado é marcado para evitar loops.
- Implementável com uma fila!

Busca em Largura

Algorithm 1: BFS

Input: G, v

foreach $u \in V$ **do**
  $u.color \leftarrow WHITE$
  $u.\pi \leftarrow$ **NIL**

$Q \leftarrow \emptyset$
$Q.\text{PUSH}(v)$
$v.color \leftarrow GREY$
while $\neg Q.\text{EMPTY}()$ **do**
  $u \leftarrow Q.\text{POP}()$
  foreach $v \in (u, v)$ **do**
    if($v.color = WHITE$)
      $v.color \leftarrow GREY$
      $v.\pi \leftarrow u$
      $Q.\text{PUSH}(v)$

  $u.color \leftarrow BLACK$

Busca em Largura

Complexidade

- $O(|V| + |E|)$ com listas de adjacências.
- $O(|V|^2)$ com matrizes de adjacências.

Busca em Largura

Complexidade

- $O(|V| + |E|)$ com listas de adjacências.
- $O(|V|^2)$ com matrizes de adjacências.
- Por que?

Busca em Largura

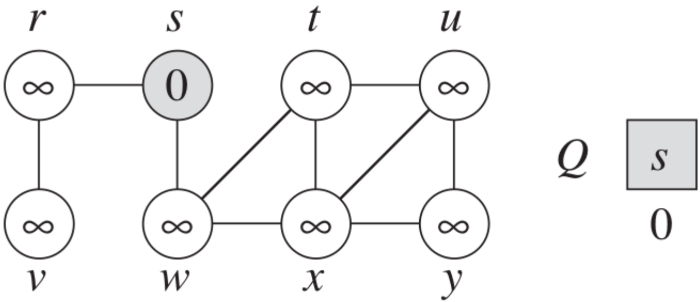


Figura: Como ficaria a busca em largura para este grafo?

Busca em Largura

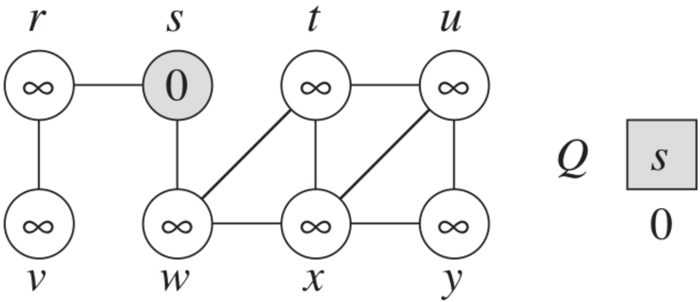


Figura: Busca em largura partindo do nó s.

Busca em Largura

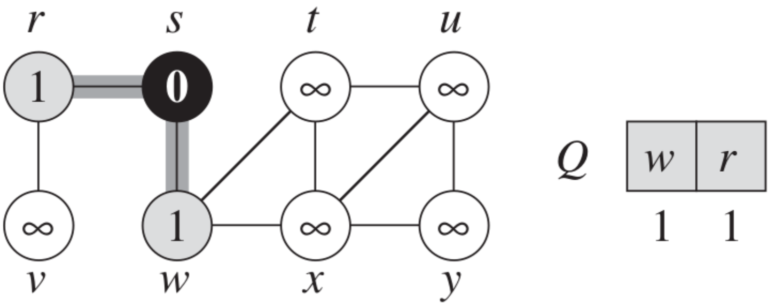


Figura: Busca em largura partindo do nó s.

Busca em Largura

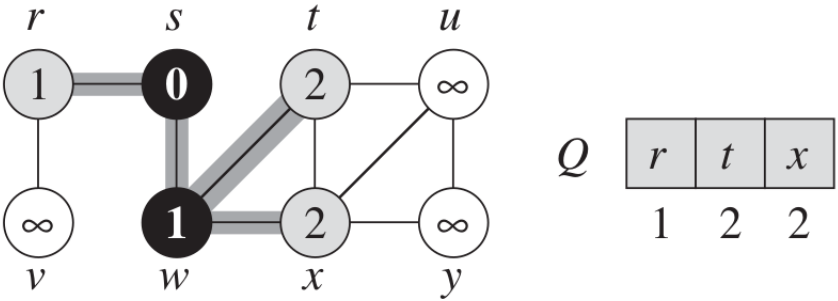


Figura: Busca em largura partindo do nó s.

Busca em Largura

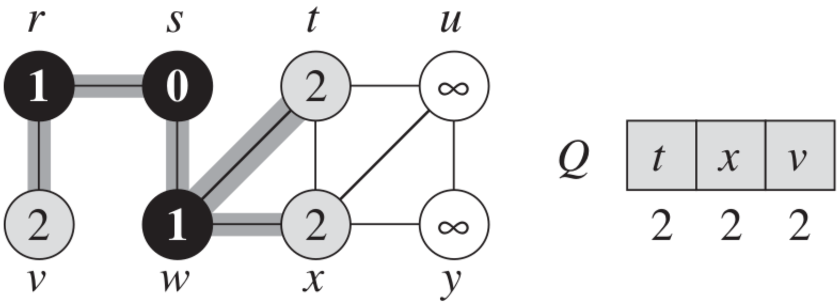


Figura: Busca em largura partindo do nó s.

Busca em Largura

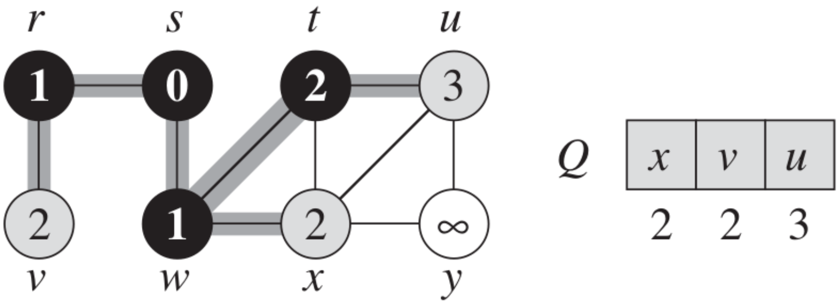


Figura: Busca em largura partindo do nó s.

Busca em Largura

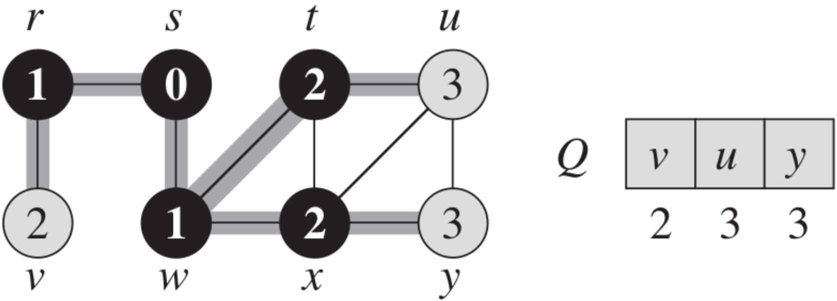


Figura: Busca em largura partindo do nó s.

Busca em Largura

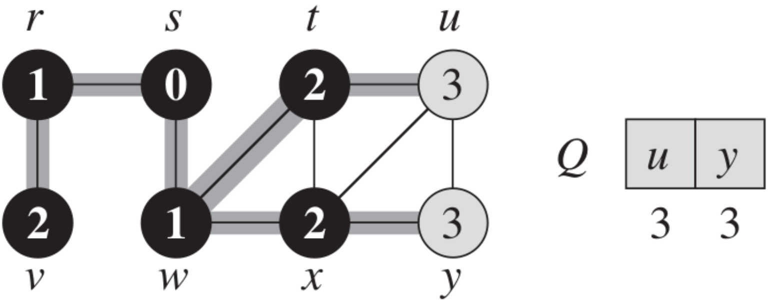


Figura: Busca em largura partindo do nó s.

Busca em Largura

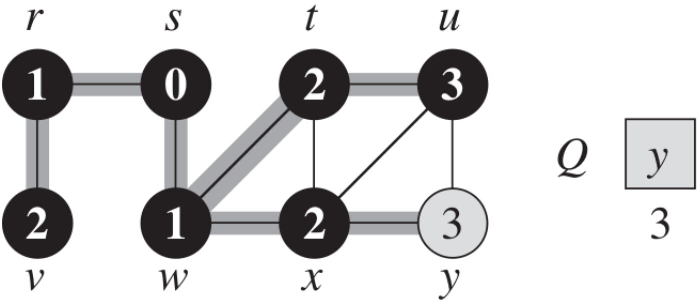


Figura: Busca em largura partindo do nó s.

Busca em Largura

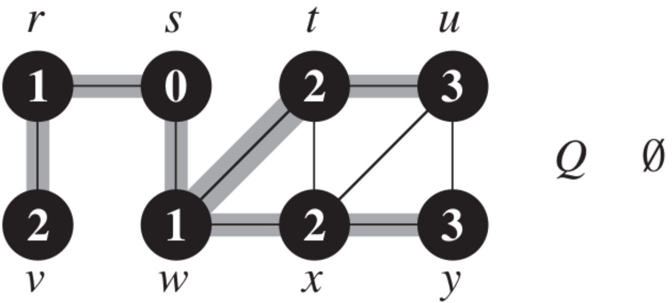


Figura: Busca em largura partindo do nó s.

Busca em Profundidade

DFS

- A busca em profundidade parte de um determinado nó e avança para o seu vizinho imediato.
- Recursivamente, repetimos a mesma ideia para o vizinho imediato.
- Apenas após ter ido à profundidade máxima, passamos para o próximo vizinho.

Depth-First-Search

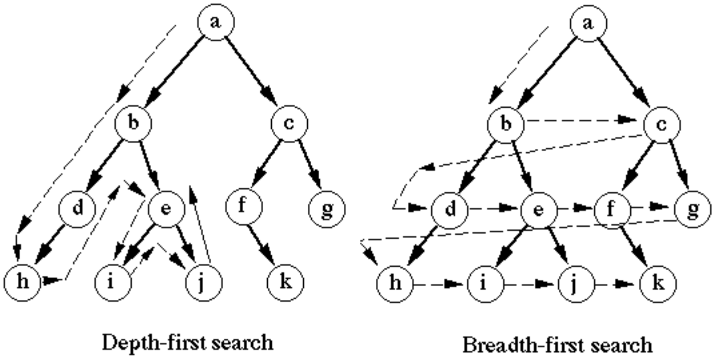


Figura: Busca em Largura vs Busca em Profundidade.

Busca em Profundidade

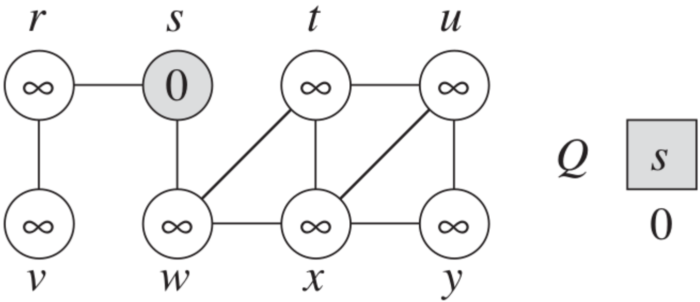


Figura: Como ficaria a busca em profundidade para este grafo?

Busca em Profundidade

- Como implementar a busca em profundidade?

Busca em Profundidade

- Como implementar a busca em profundidade?
- Busca em largura trocando fila por pilha!

Busca em Profundidade

Algorithm 2: DFS

```
Input: G, v
foreach u ∈ V do
    u.color ← WHITE
    u.π ← NIL
S ← ∅
S.PUSH(v)
v.color ← GREY
while ¬S.EMPTY() do
    u ← S.POP()
    foreach v ∈ (u, v) do
        if( v.color = WHITE )
            v.color ← GREY
            v.π ← u
            S.PUSH(v)
    u.color ← BLACK
```

Busca em Profundidade

- Podemos implementar recursivamente também.
- Mais simples e mais elegante.
- Pilha implícita.

Busca em Profundidade

Algorithm 3: DFS

Input: G, v

```
v.color ← GREY
foreach w ∈ (v, w) do
    if( w.color = WHITE )
        w.color ← GREY
        w.π ← v
        DFS(G, w)
v.color ← BLACK
```

Problemas

- Veremos uma serie de problemas em que estes simples problemas de busca são aplicáveis.

Sumário

2 Percurso
- Breath-First-Search
- Depth-First-Search
- Problemas

Problemas

Menor Distância

- Dado um grafo **sem peso**, determine a distância de um vértice para todos os outros vértices.
- Neste caso, a distância de u e v, denotada por $D(u, v)$ é dada pela quantidade de arestas do menor caminho estes dois vértices.
- Extremamente aplicável em roteamento! Queremos minimizar o número de saltos.

Menor Distância

- Para computara menor distância, podemos recorrer à uma simples busca em largura.
- A cada passo da busca em largura, estamos descobrindo nós à uma distância de uma unidade maior.

Busca em Largura

Algorithm 4: BFS para computar a menor distância entre os vértices.

Input: G, v

foreach $u \in V$ **do**
- $u.color = WHITE$
- $u.\pi =$ **NIL**
- $u.d = \infty$

$Q \leftarrow \emptyset$

$v.d \leftarrow 0$

$Q.\textsc{push}(v)$

$v.color \leftarrow GREY$

while $\neg Q.\textsc{empty}()$ **do**
- $u \leftarrow Q.\textsc{pop}()$
- **foreach** $v \in (u, v)$ **do**
  - **if**($v.color = WHITE$)
    - $v.\pi = u$
    - $v.d = u.d + 1$
    - $Q.\textsc{push}(v)$
    - $v.color \leftarrow GREY$
- $u.color \leftarrow BLACK$

Busca em Largura

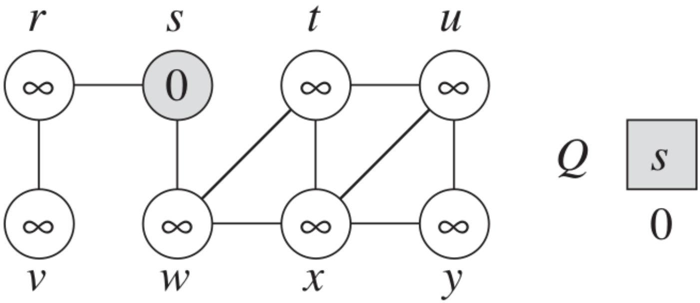


Figura: Busca em largura partindo do nó s.

Busca em Largura

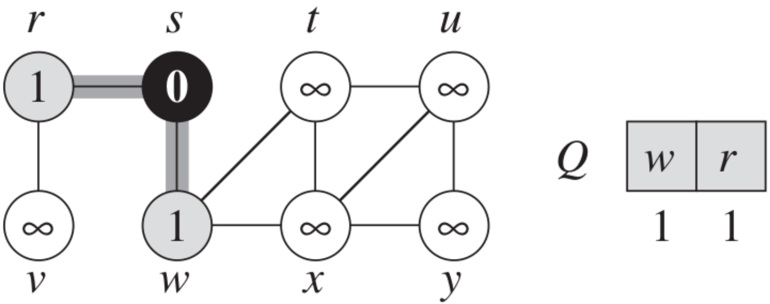


Figura: Busca em largura partindo do nó s.

Busca em Largura

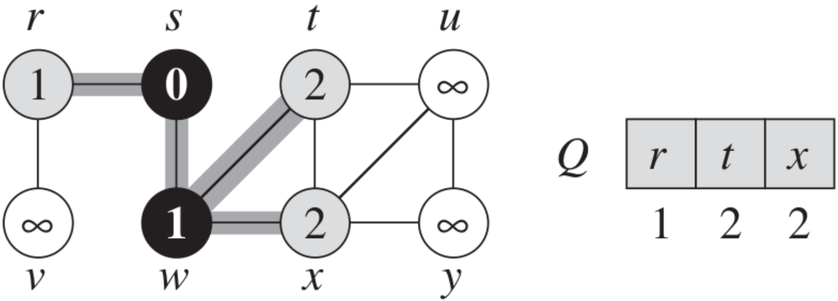


Figura: Busca em largura partindo do nó s.

Busca em Largura

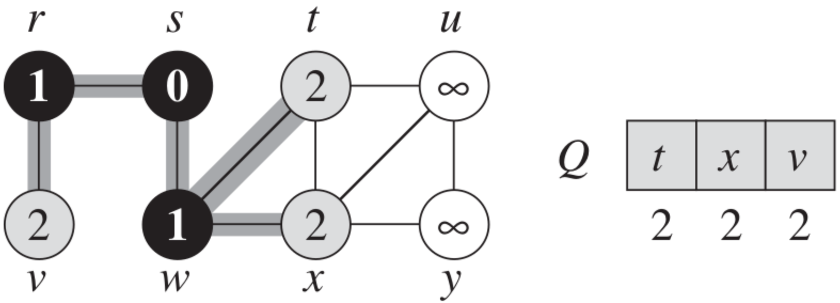


Figura: Busca em largura partindo do nó s.

Busca em Largura

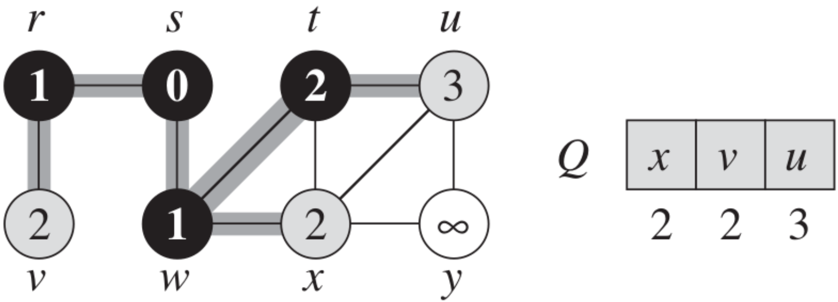


Figura: Busca em largura partindo do nó s.

Busca em Largura

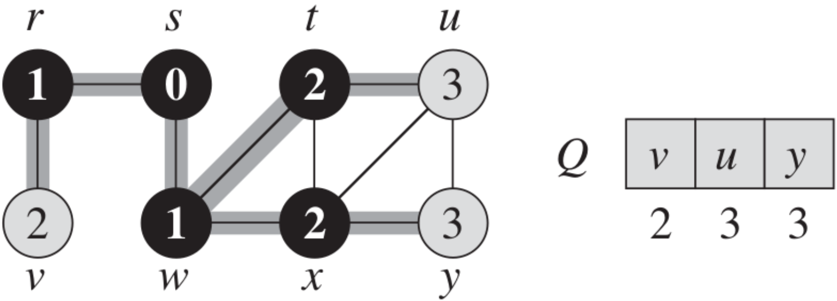


Figura: Busca em largura partindo do nó s.

Busca em Largura

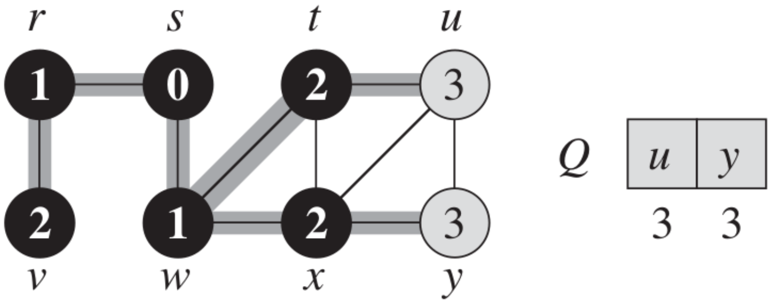


Figura: Busca em largura partindo do nó s.

Busca em Largura

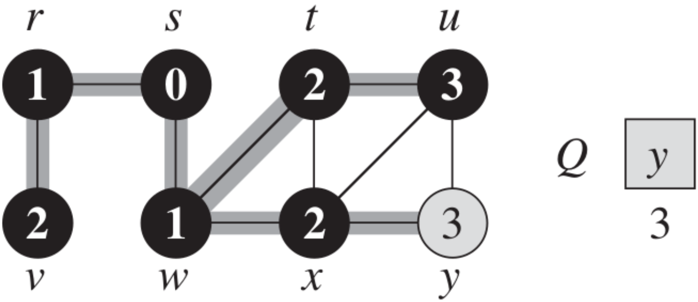


Figura: Busca em largura partindo do nó s.

Busca em Largura

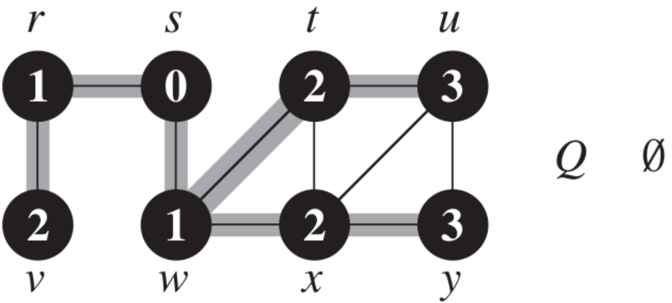


Figura: Busca em largura partindo do nó s.

Problemas

Ordenação Topológica

- Dado um grafo **acíclico** dirigigo (DAG), produzir uma ordenação topológica é interessante em algumas aplicações.
- A ordenação topológica produz uma ordenação dos vértices tal que, se existe uma aresta (u, v), então u deve estar antes de v no resultado da ordenação.
- Na prática, podemos usar DAGs para indicar precedência de eventos.
- Exemplo: verificar quais disciplinas são pré-requisito de outras.

Ordenação Topológica

- Para produzir a ordenação topológica, podemos usar a busca em profundidade para marcar os tempos em que um nó é visitado pela primeira e segunda vez (tempo de início e tempo de término).
- Conforme a busca, adicionamos o nó com tempo de término mais recente no início de uma lista.
- Isso quer dizer que o nó com tempo mais recente obrigatoriamente vem antes do nó com tempo menos recente na ordenação.

Ordenação Topológica

Algorithm 5: TOPOLOGICAL-SORT

Input: G

Output: Lista contendo os nós em ordem topológica.

$time \leftarrow 0$
$list \leftarrow [\,]$
for all($v \in V$)
  $time \leftarrow \mathrm{DFS}(G, v, time, list)$
return $list$

Ordenação Topológica

Algorithm 6: DFS que computa os tempos de início e fim de visitação

Input: $G, v, time, list$

if($v.color \neq BLACK$)
- $v.color = BLACK$
- $time++$
- $v.d \leftarrow time$
- **foreach** $w \in (v, w)$ **do**
  - **if**($w.color = WHITE$)
    - $w.\pi = v$
    - $time \leftarrow \text{DFS}(G, w, time, list)$
- $time++$
- $v.f \leftarrow time$
- $list.\text{PUSH-FRONT}(v)$
- **return** $time$

Ordenação Topológica

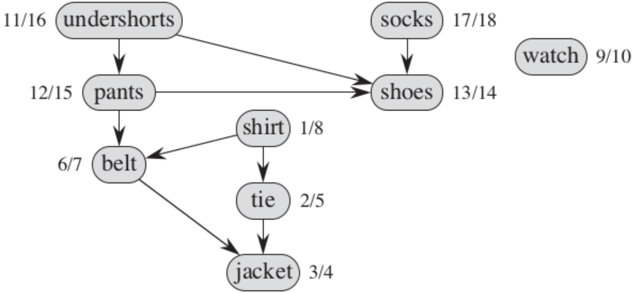


Figura: Ordenação topológica da sequência de vestimento.

Ordenação Topológica

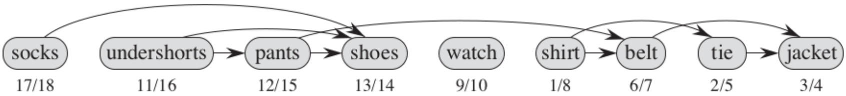


Figura: Ordenação topológica da sequência de vestimento.

Ordenação Topológica

Complexidade

- Precisamos apenas fazer uma busca em profundidade modificada.
- $O(|V| + |E|)$

Problemas

Detecção de Ciclos

- Detectar ciclos em grafos dirigidos é muito útil em algumas aplicações.
- Exemplo: detecção de deadlock pelo S.O.
- Exemplo: detecção de incompatibilidade de dependências.

Problemas

Detecção de Ciclos

- Entrada: Um grafo dirigido.
- Saída: Sim, se o grafo possui ciclos, não, caso contrário.

Detecção de Ciclos

- Estamos procurando por uma **back edge**, isto é, uma aresta que volta para um nó que ainda está sendo processado na busca em profundidade.
- Se no grafo existe uma **back edge**, então temos um ciclo.
- Basta adaptar a busca em profundidade.

Detecção de Ciclos

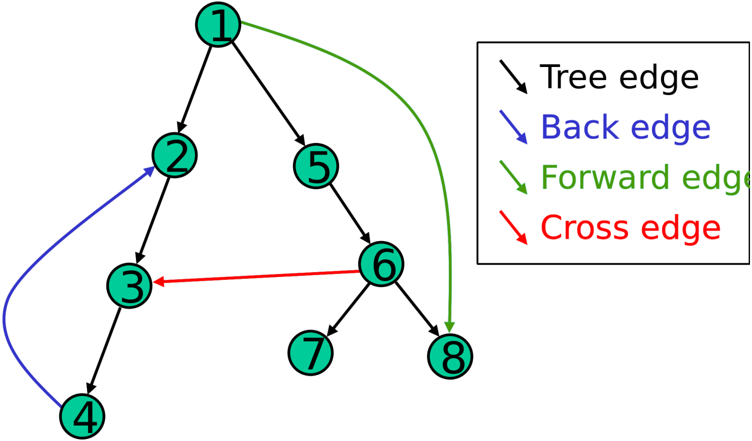


Figura: Tipos de aresta.

Detecção de Ciclos

Algorithm 7: CYCLE-DETECTION

Input: G

Output: Sim, se G tem ciclos, não, caso contrário.

```
for all( v ∈ V )
    v.color = WHITE
    v.π = NIL
for all( v ∈ V )
    if( v.color = WHITE )
        if( CYCLE-SEARCH(G, v) )
            return true
return false
```

Detecção de Ciclos

Algorithm 8: CYCLE-SEARCH

Input: G, v

Output: Sim, sse a componente conexa de v tem ciclos

```
v.color = GREY
temp ← false
for all( (v,w) ∈ E )
    if( w.color = GREY )                    // back edge
        temp ← true
    else if( w.color = WHITE )
        if( CYCLE-SEARCH(G,w) )
            temp ← true
v.color = BLACK
return temp
```

Detecção de Ciclos

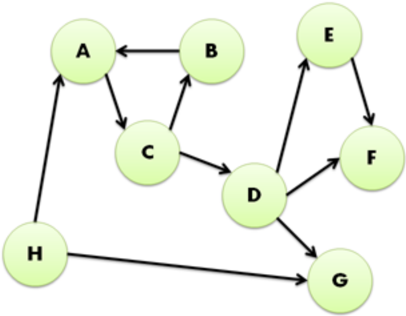


Figura: Detecção de Ciclos

Detecção de Ciclos

Complexidade

- $O(|V| + |E|)$.

Componentes Fortemente Conexas

- Todo grafo dirigido pode ser decomposto em várias componentes fortemente conexas.
- Um grafo é dito fortemente conexo se possui apenas 1 componente fortemente conexa.
- Como determinar as componentes fortemente conexas de um grafo?

Componentes Fortemente Conexas

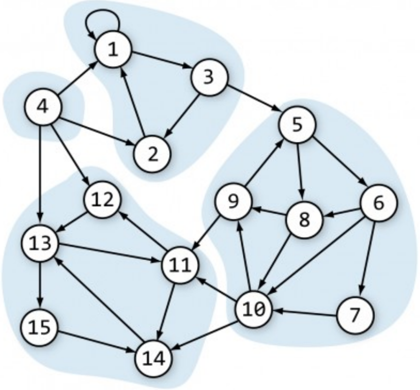


Figura: Componentes fortemente conexas.

Componentes Fortemente Conexas

Componentes Fortemente Conexas

- **Entrada:** um grafo G.
- **Saída:** suas componentes fortemente conexas.

Componentes Fortemente Conexas

- Uma componente é dita fortemente conexa se existe um caminho entre quaisquer dois pares de vértice nesta componente.
- Se a partir de um vértice v chegamos em u, temos que certificar que é possível chegar em v à partir de u.
- Podemos usar o conceito de **back edge**!

Componentes Fortemente Conexas

- Primeiramente, numeramos cada vértice com com a busca em profundidade de acordo com sua ordem de exploração ($v.index$).
- Se durante a busca em profundidade um nó u possuí um caminho para um nó v que tem um número menor que u, então sabemos que também existe um caminho de u para v, e portanto, eles estão na mesma componente conexa.
- Marcaremos cada nó com um número $v.low$, indicando o vértice com menor número que é alcançável por ele.
- Todos os nós que possuem $v.low \leq v.index$ estão na mesma componente conexa de $v.low$.

Componentes Fortemente Conexas

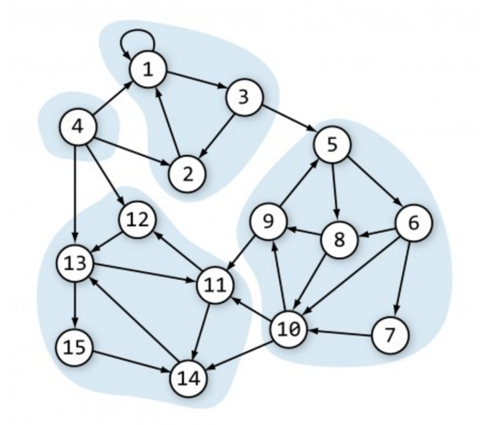


Figura: Componentes fortemente conexas.

Componentes Fortemente Conexas

Algorithm 9: FIND-STRONG-COMPONENTS(G)

Input: G

Output: Componentes Fortemente Conexas de G

$list \leftarrow [\,]$

$S \leftarrow \emptyset$

$n \leftarrow 0$

for all($v \in V$)

  if($v.color = WHITE$)

    STRONG-COMPONENTS$(G, v, list, S, n)$

return $list$

Componentes Fortemente Conexas

Algorithm 10: STRONG-COMPONENTS$(G, v, \&list, \&S, \&n)$

Input: $G.v, list, S$

Output: Componentes Fortemente Conexa que inclui o nó v

```
v.index = n
v.low = n
n ← n + 1
v.color = GREY
S.PUSH(v)
for all( (v, u) ∈ E )
    if( u.color = GREY )
        v.low = min{v.low, u.index}
    if( u.color = WHITE )
        STRONG-COMPONENTS(G, u, list, S)
        v.low = min{v.low, u.low}
v.color = BLACK
if( v.index = v.low )
    list.INSERT(CREATE-NEW-COMPONENT(S, v))
```

Componentes Fortemente Conexas

Algorithm 11: CREATE-NEW-COMPONENT$(\&S, v)$

Input: $\&S, v$

Output: Componentes Fortemente Conexa que inclui o nó v

```
list' ← [ ]
repeat
    w = S.POP()
    list'.INSERT(w)
until w ≠ v
return list'
```

Componentes Fortemente Conexas

Complexidade

- $O(|V| + |E|)$.

Sumário

Menor Caminho

- Detectar o menor caminho por dois vértices é fácil quando o grafo não possui peso.
  - Busca em largura.
- E no caso genérico? Se o grafo possuir pesos, como resolvemos este problema.
- Antes de definir o problema, examinaremos alguns conceitos.

Menor Caminho

Definição (Custo do Caminho)

- Suponha um grafo dirigido $G(V, E)$ e uma função de peso sobre as arestas $w : E \to \mathbb{R}$.
- Tome um caminho $p = (v_0, \ldots, v_k)$. O custo do caminho é definido como:

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

Menor Caminho

Definição (Custo do Menor Caminho)

- O custo do menor caminho dentre um vértice u e um vértice v é dado por:

$$\delta(u,v) = \begin{cases} \min\{w(p) | u \rightarrow^{p} v\}, & \text{se existe um caminho de } u \text{ a } v \\ \infty, & \textbf{caso contrário.} \end{cases}$$

Menor Caminho

- Agora podemos definir o problema!

Menor Caminho

- Entrada: um grafo dirigido $G(V, E)$, uma função de peso $w : E \to \mathbb{R}$ e um vértice de origem v.
- Saída: $\delta(v, w)$, o custo do menor caminho de v até os demais vértices w.

Menor Caminho

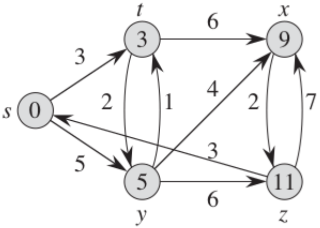


Figura: Grafo $G(V,E)$ com as respectivas distâncias de uma origem.

Menor Caminho

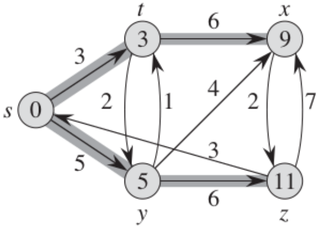


Figura: Menor rota até um destino.

Menor Caminho

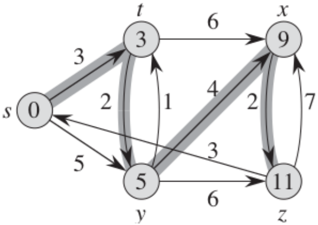


Figura: Menor rota até um destino.

Menor Caminho

- Um dos algoritmos que resolve esse problema é o algoritmo de Djikstra.

Sumário

Algoritmo de Dijkstra

- O algoritmo de Dijkstra se parece muito com uma busca em largura.
- Só que em vez de pegar sempre o próximo vizinho, consideramos o nó com menor custo até o momento.
- Pode ser visto como um algoritmo guloso!

Algoritmo de Dijkstra

- O algoritmo de Dijkstra se baseia no "relaxamento" de distâncias até chegar na distância ótima.
- Se a distância atual de uma origem a um nó v é maior do que a distância atual da origem a um nó u mais $w(u, v)$, atualizamos a distância atual.

$$v.d > u.d + w(u, v)$$
$$v.d \leftarrow u.d + w(u, v)$$

Algoritmo de Dijkstra

Algorithm 12: INITIALIZE-DIJKSTRA

Input: G, s

for all($v \in V$)
    $v.d \leftarrow \infty$
    $v.\pi \leftarrow$ **NULL**
$s.d \leftarrow 0$

Algorittmo de Dijkstra

Algorithm 13: DIJKSTRA

Input: G, w, s

Output: $\delta(s, v), \forall v \in V$

INITIALIZE-DJIKSTRA(G, s)

Q.INSERT(s) `// Fila de prioridades`

while $\neg Q$.EMPTY **do**

  $u \leftarrow Q$.EXTRACT-MIN$()$

  $visited[u] \leftarrow true$

  for all($(u, v) \wedge \neg visited[v]$)

    if($v.d > u.d + w(u, v)$)

      $v.d \leftarrow u.d + w(u, v)$

      $v.\pi \leftarrow u$

      Q.INSERT-UPDATE(v) `// Insere ou atualiza o nó na fila`

Algoritmo de Djikstra

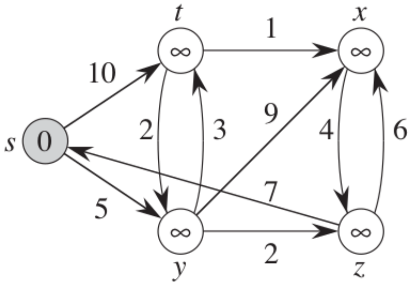


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

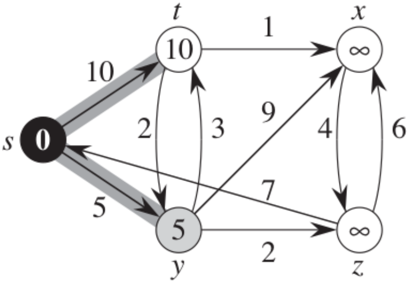


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

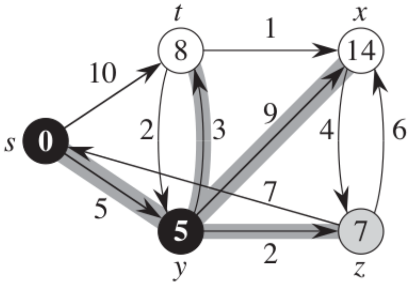


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

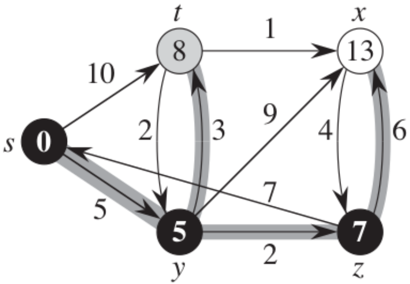


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

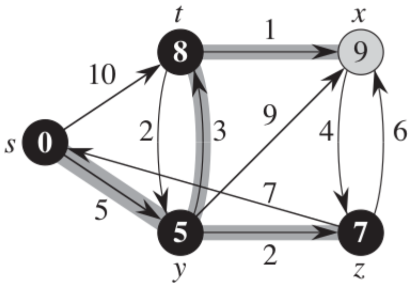


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

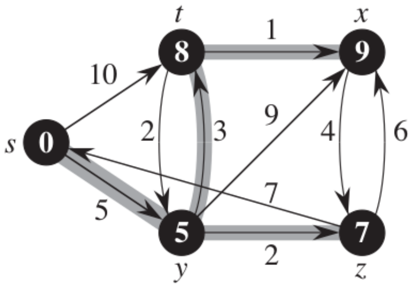


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

Complexidade

- Qual a complexidade do algoritmo?
- Depende da estrutura Q utilizada.
- **For** interno: $\Theta(|E|)$ vezes.
- **While** externo: $\Theta(|V|)$ vezes.
- Vai depender do custo das operações EXTRACT-MAX e INSERT-UPDATE para a estrutura de dados Q.

Algoritmo de Djikstra

Complexidade

- Usando vetores: $\Theta(|V|^2 + |E|)$.
- Usando heap: $\Theta(|V| \log |V| + |E| \log |V|)$.
- Usando heap de fibonacci: $\Theta(|V| \log |V| + |E|)$.
- O que você vai usar em grafos densos? E em grafos esparsos?

Algoritmo de Dijkstra

Correção do Algoritmo de Dijkstra

- Por que o algoritmo de Dijkstra funciona?

Algoritmo de Dijkstra

Limitações

- Apesar de ser um algoritmo clássico, o algoritmo de Dijkstra apresenta alguns problemas.
- Se a função w atribuir um custo negativo às arestas, o algoritmo de Dijkstra não apresentará o comportamento esperado.
- Problema: ciclos negativos!
- O que era um problema fácil, passa a ser um problema difícil.

Sumário

Motivação

- Suponha que tenhamos uma infraestrutura de rede montada.
- Várias máquinas estão conectadas à outras através de diversos roteadores.
- Ao mesmo tempo, a economia de energia se tornou uma situação crítica nos dias de hoje.
- Como você faria para continuar permitindo a comunicação de quaisquer computadores com menor custo possível?
- Quais roteadores você desativaria?
- Qual a estrutura obtida?

Motivação

- O problema da árvore espalhada mínima visa resolver este tipo de problemas.
- Queremos um subgrafo acíclico e conexo de menor custo (árvore de menor custo).
- Existem algoritmos bem conhecidos para resolução deste problema, tais como:
  - Algoritmo de Prim.
- No entanto, vamos examinar algumas definições antes de atacar o problema.

Árvore Espalhada Mínima

Menor custo

- Obviamente, o subgrafo gerado de menor custo tem que ser uma árvore.
- O custo desta árvore é dado pelo somatório de suas arestas:

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

Árvore Espalhada Mínima

Menor custo

- Qual a estratégia para chegar no menor custo possível?

Árvore Espalhada Mínima

Menor custo

- Qual a estratégia para chegar no menor custo possível?
- Inicialmente, todo vértice é uma componente conexa.

Árvore Espalhada Mínima

Menor custo

- Qual a estratégia para chegar no menor custo possível?
- Inicialmente, todo vértice é uma componente conexa.
- Adicione arestas de menor custo possível de modo que conecte componentes conexas.

Árvore Espalhada Mínima

Menor custo

- Qual a estratégia para chegar no menor custo possível?
- Inicialmente, todo vértice é uma componente conexa.
- Adicione arestas de menor custo possível de modo que conecte componentes conexas.
- Jamais adicione uma aresta de custo maior que conecte as mesmas componentes.

Árvore Espalhada Mínima

Menor custo

- Qual a estratégia para chegar no menor custo possível?
- Inicialmente, todo vértice é uma componente conexa.
- Adicione arestas de menor custo possível de modo que conecte componentes conexas.
- Jamais adicione uma aresta de custo maior que conecte as mesmas componentes.
- Jamais forme um ciclo!

Árvore Espalhada Mínima

Algorithm 14: GENERIC-MST

$T \leftarrow \emptyset$
while T *não for uma árvore* **do**
    Encontre uma aresta (u, v) que é segura para T
    Adicione a aresta à T

Àrvore Espalhada Mínima

- Como escolher uma aresta segura?

Sumário

4 Árvore Espalhada Minima
- Prim

Algoritmo de Prim

- O algoritmo de Prim de certa forma se parece muito com o algoritmo de Djikstra.
- Começamos de um nó arbitrário como o único nó de nossa árvore.
- Escolhemos sempre as arestas de menor custo para adicionarmos à árvore a partir dos nós previamente inseridos na árvore.
- Da mesma forma que no algoritmo de Djikstra, precisamos de uma estrutura de dados eficiente.

Algoritmo de Dijkstra

Algorithm 15: INITIALIZE-PRIM

Input: G, s

for all($v \in V$)
    $v.d \leftarrow \infty$
    $v.\pi \leftarrow$ **NULL**
$s.d \leftarrow 0$

Algorittmo de PRIM

Algorithm 16: PRIM

Input: G, w, s

Output: $MST(G)$

```
INITIALIZE-PRIM(G, s)
Q.INSERT(s)
T ← s
last ← s
while ¬Q.EMPTY do
    u ← Q.EXTRACT-MIN()
    visited[u] ← true
    T ← T ∪ (last, u)
    last ← u
    for all( (u, v) ∧ ¬visited[v] )
        if( v.d > w(u, v) )
            v.d ← w(u, v)
            v.π ← u
            Q.INSERT-UPDATE(v)
return T
```

Algoritmo de Prim

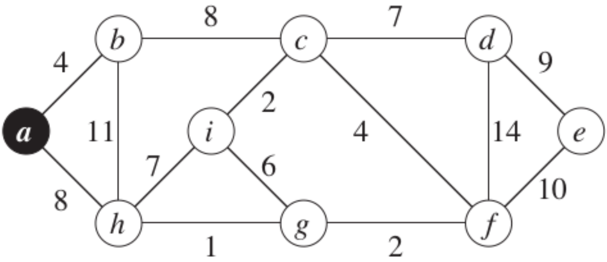


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

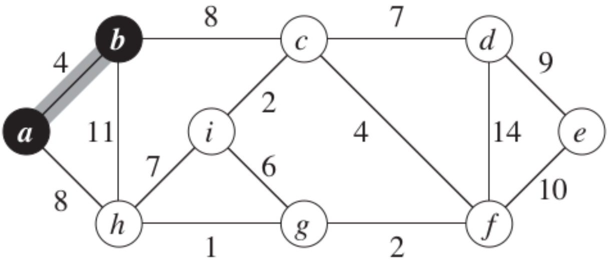


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

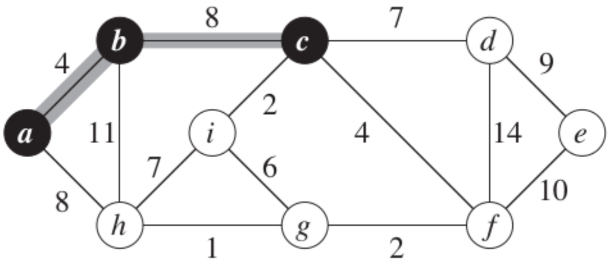


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

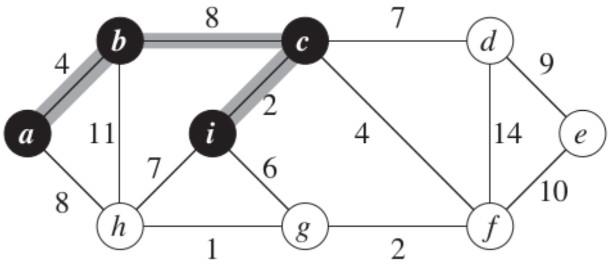


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

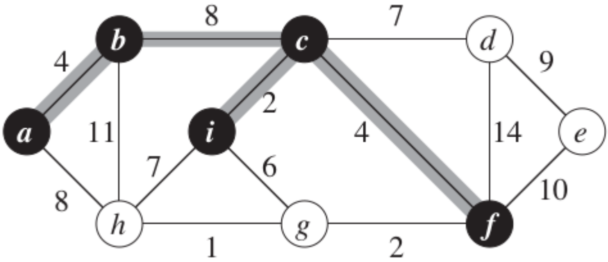


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

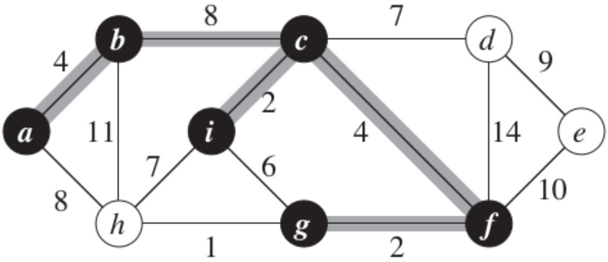


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

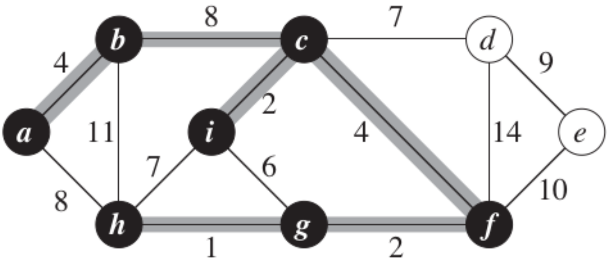


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

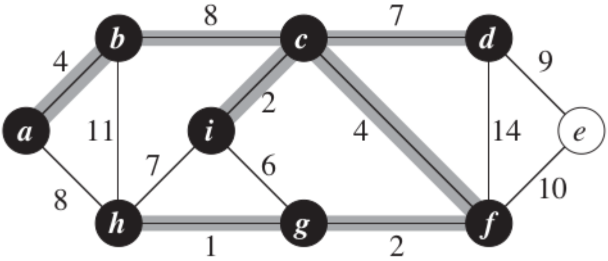


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

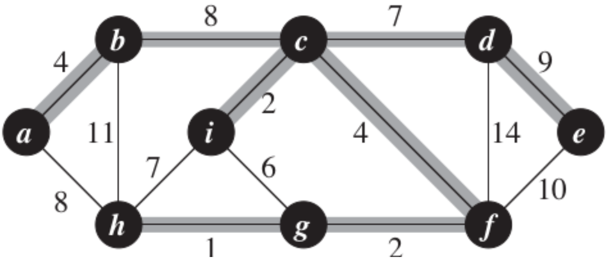


Figura: Execução do Algoritmo de Prim