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 $\times$ Não-simples.
  2. Dirigido $\times$ Não-dirigido.
  3. Com peso $\times$ Sem peso.
  4. Esparso $\times$ Denso.
  5. Cíclico $\times$ Acíclico.
  6. Incorporado $\times$ Topológico.
  7. Implícito $\times$ Explícito.
  8. Rotulado $\times$ Não-rotulado.

Sumário

Tipos de Grafos

Simples × 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

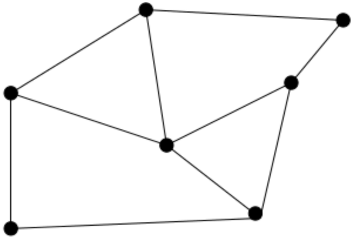


Figura: Simples × Não-simples.

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

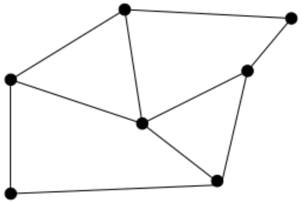
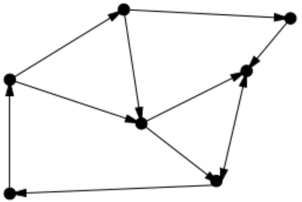
 

Figura: Dirigido × Não-dirigido.

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

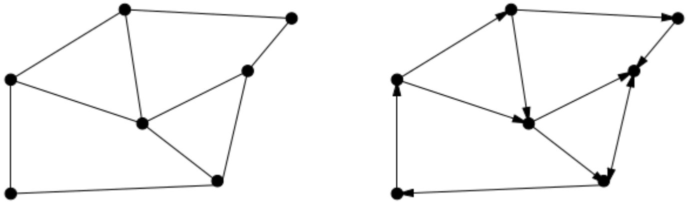


Figura: Dirigido × Não-dirigido.

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

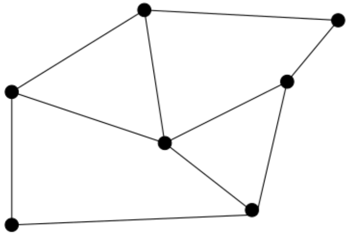
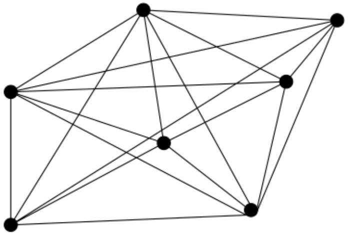



Figura: Esparso × Denso.

Tipos de Grafos

Cíclicos $\times$ Acíclicos

- Um grafo acíclico são grafos que não possuem ciclos.

Tipos de Grafos

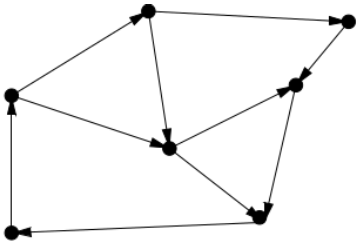
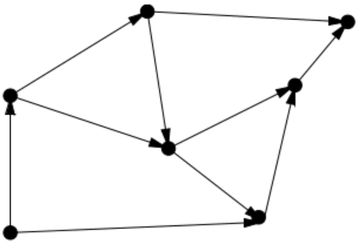
 

Figura: Cíclico $\times$ Acíclico.

Tipos de Grafos

Incorporado $\times$ 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

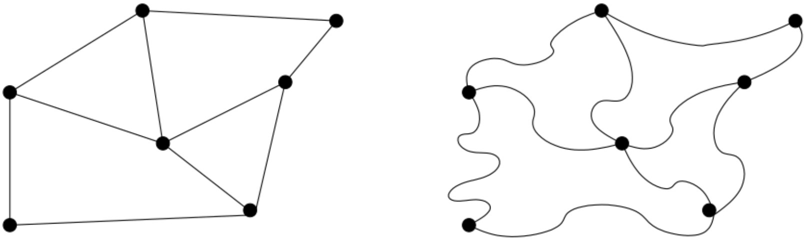


Figura: Incorporado × Topológico.

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

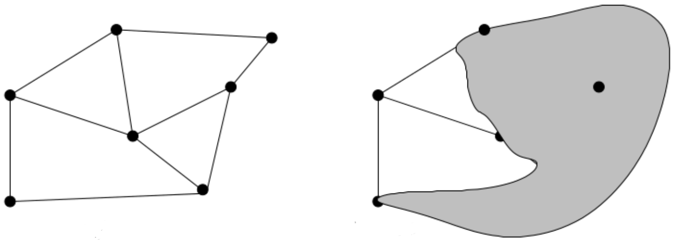


Figura: Implícito × Explícito.

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

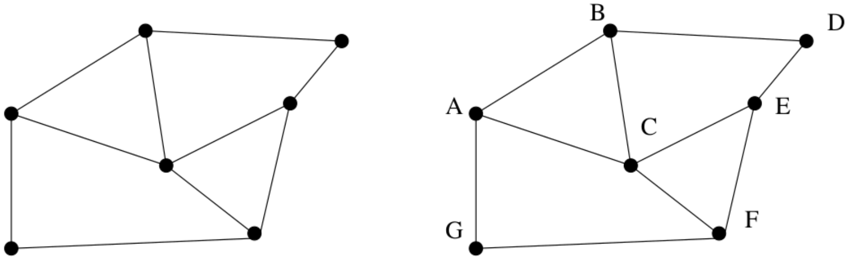


Figura: Rotulado × Não-rotulado.

Sumário

Aplicações

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

Aplicações

Figura: Navegação.

Aplicações

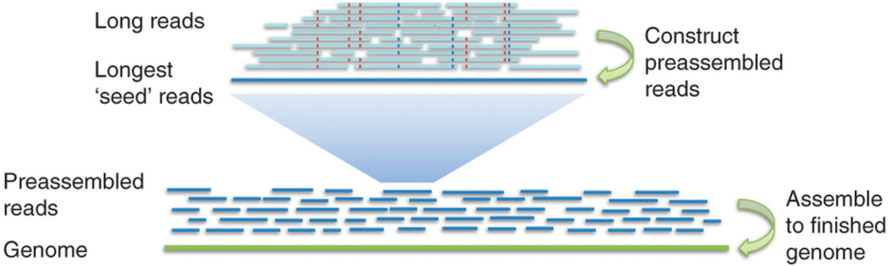


Figura: Montagem de Genomas.

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

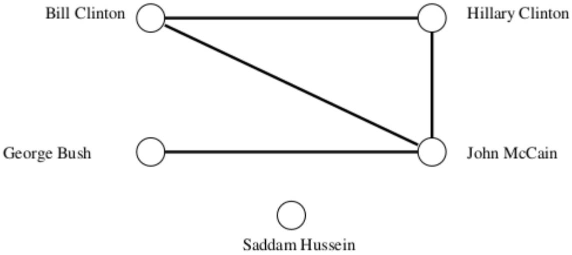


Figura: Grafo de relacionamento de pessoas.

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

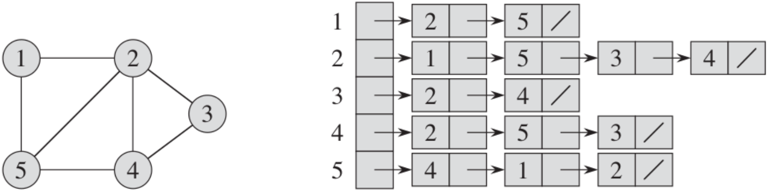


Figura: Lista de adjacências.

Listas de Adjacência

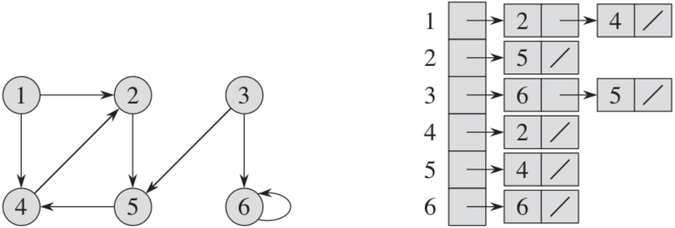


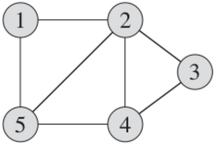
Figura: Lista de adjacências.

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.

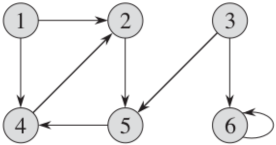
Matrizes de Adjacências



	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

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

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

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

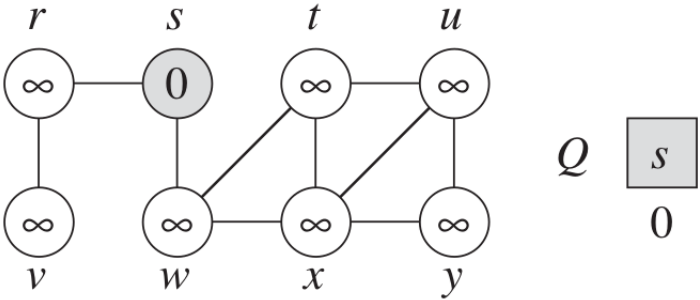


Figura: Como ficaria a busca em largura para este grafo?

Busca em Largura

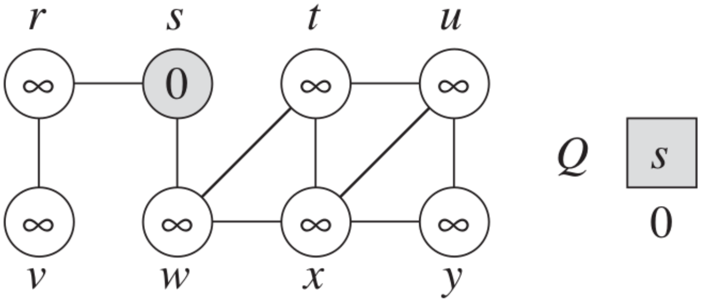


Figura: Busca em largura partindo do nó s.

Busca em Largura

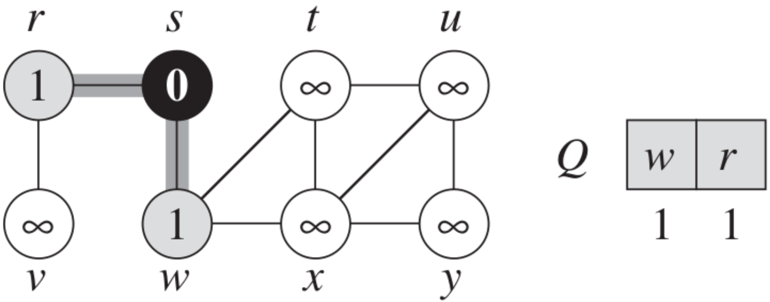


Figura: Busca em largura partindo do nó s.

Busca em Largura

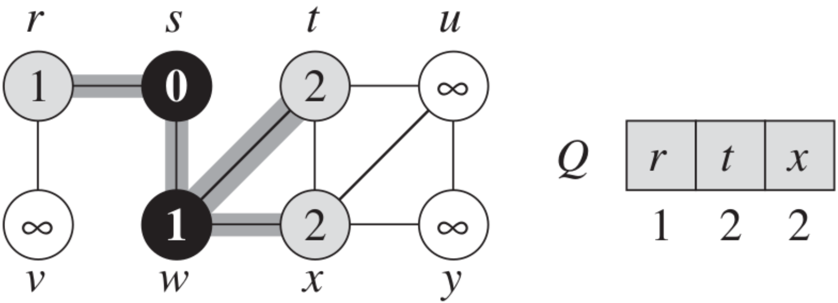


Figura: Busca em largura partindo do nó s.

Busca em Largura

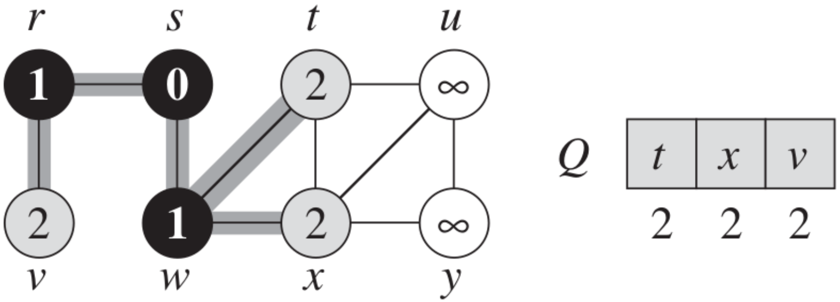


Figura: Busca em largura partindo do nó s.

Busca em Largura

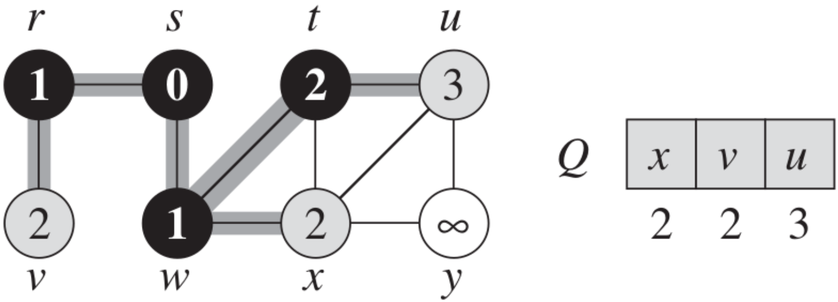


Figura: Busca em largura partindo do nó s.

Busca em Largura

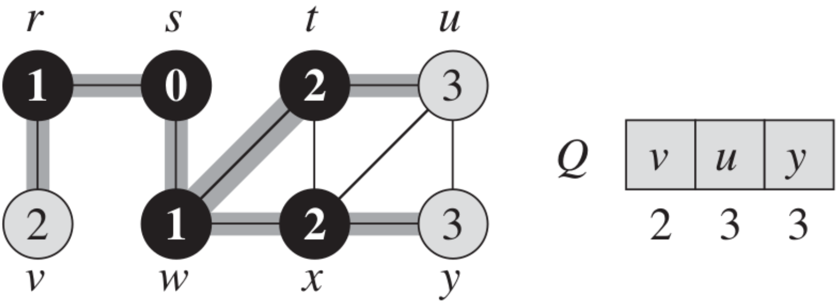


Figura: Busca em largura partindo do nó s.

Busca em Largura

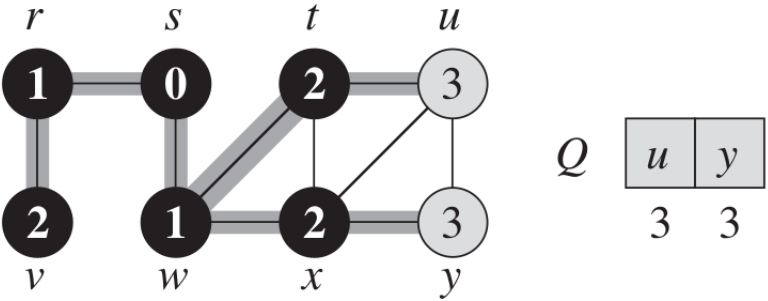


Figura: Busca em largura partindo do nó s.

Busca em Largura

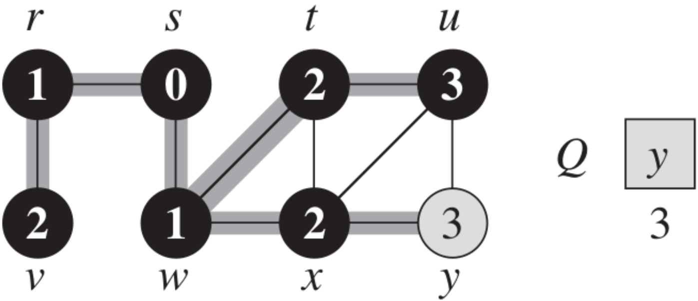


Figura: Busca em largura partindo do nó s.

Busca em Largura

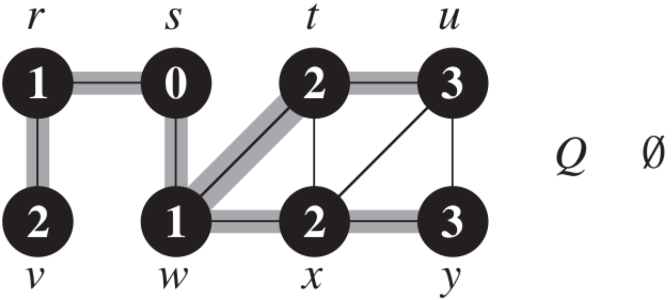


Figura: Busca em largura partindo do nó s.

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

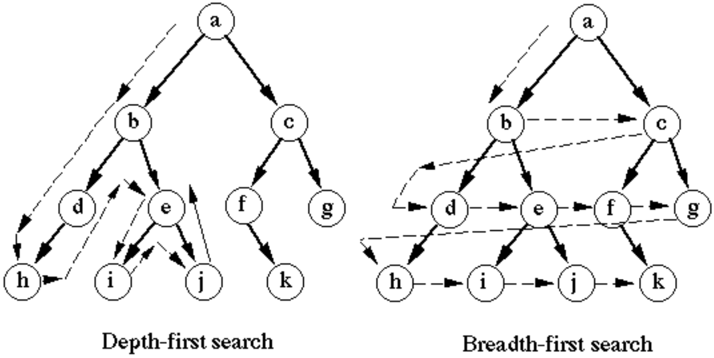


Figura: Busca em Largura vs Busca em Profundidade.

Busca em Profundidade

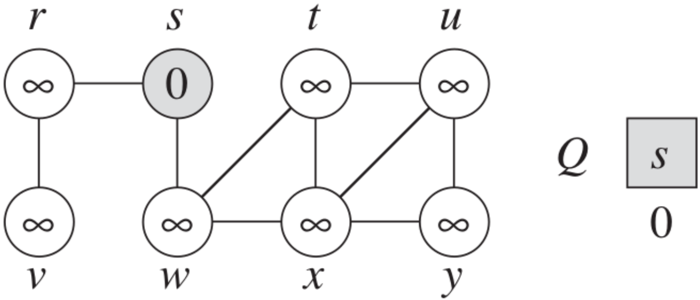


Figura: Como ficaria a busca em profundidade para este grafo?

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

Busca em Profundidade

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

Busca em Profundidade

Problemas

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

Sumário

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

Busca em Largura

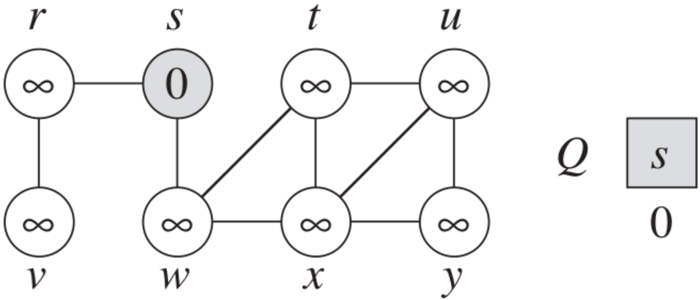


Figura: Busca em largura partindo do nó s.

Busca em Largura

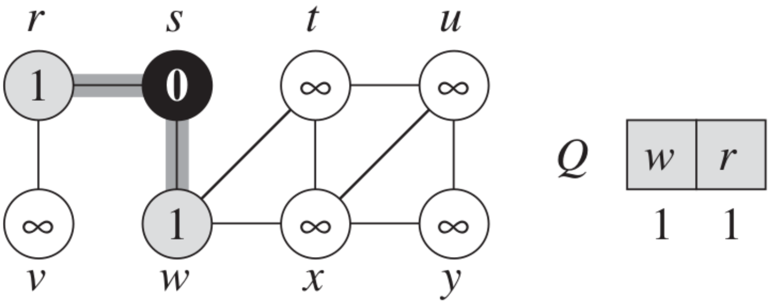


Figura: Busca em largura partindo do nó s.

Busca em Largura

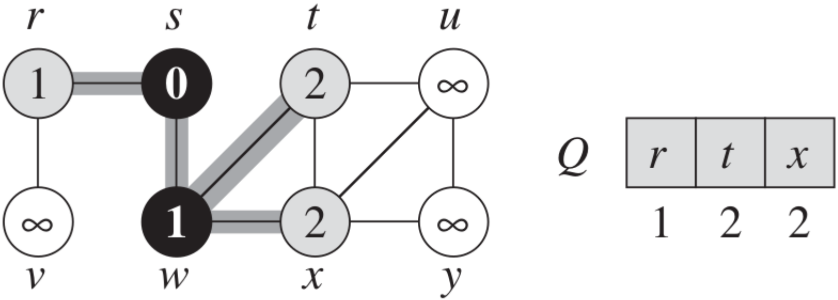


Figura: Busca em largura partindo do nó s.

Busca em Largura

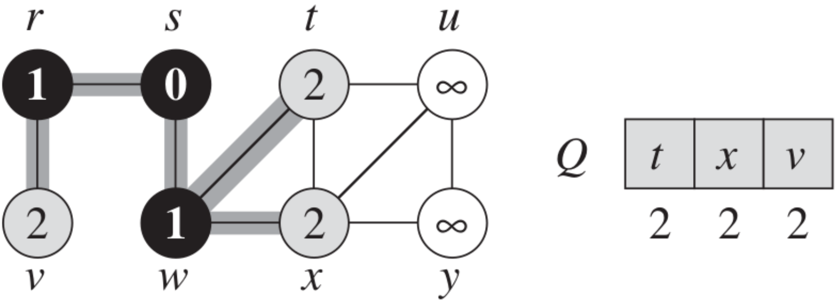


Figura: Busca em largura partindo do nó s.

Busca em Largura

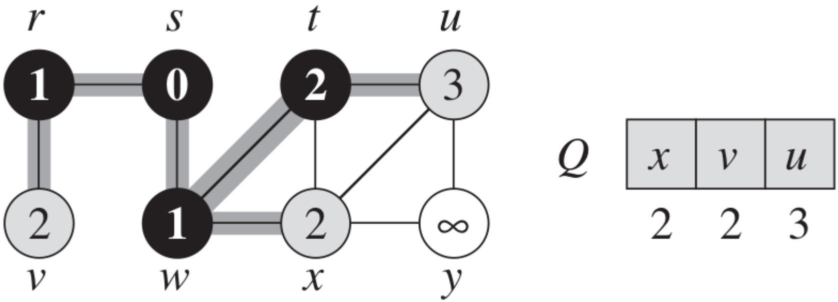


Figura: Busca em largura partindo do nó s.

Busca em Largura

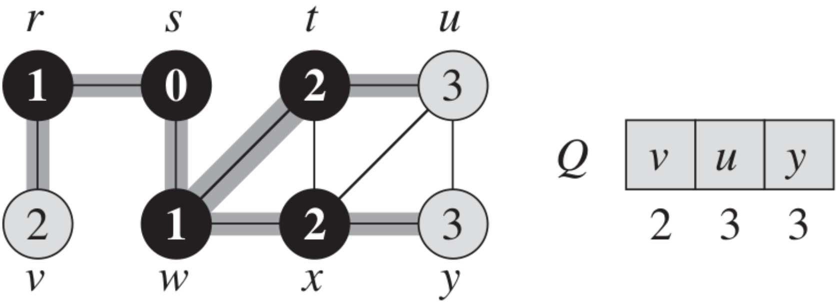


Figura: Busca em largura partindo do nó s.

Busca em Largura

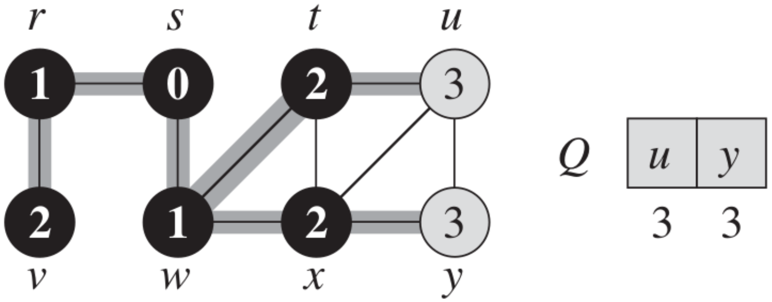


Figura: Busca em largura partindo do nó s.

Busca em Largura

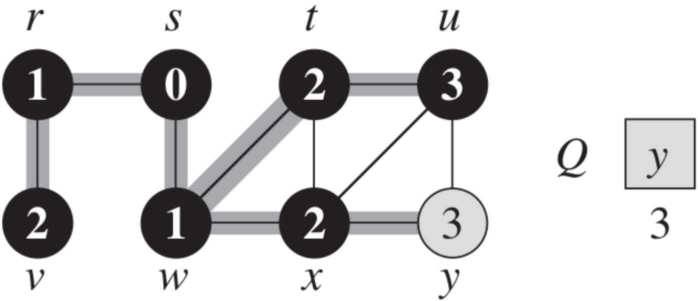


Figura: Busca em largura partindo do nó s.

Busca em Largura

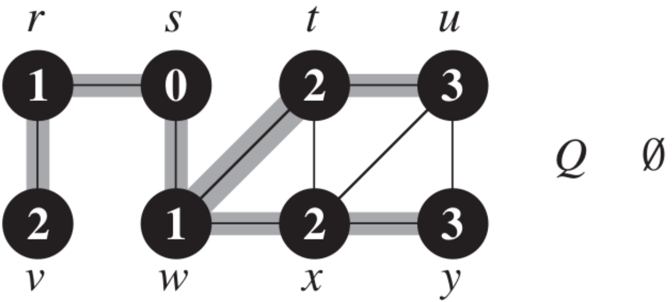


Figura: Busca em largura partindo do nó s.

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

Ordenação Topológica

Ordenação Topológica

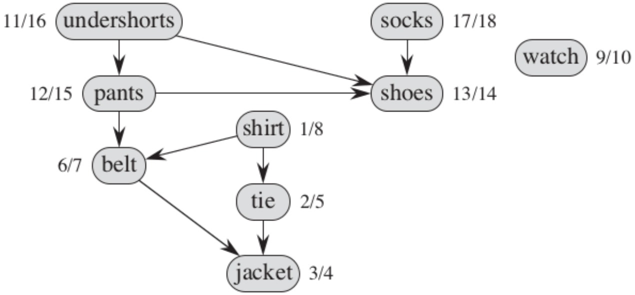


Figura: Ordenação topológica da sequência de vestimento.

Ordenação Topológica

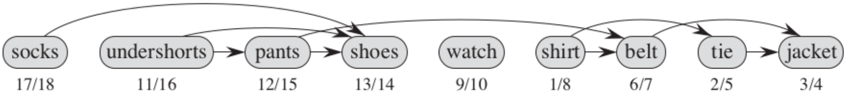


Figura: Ordenação topológica da sequência de vestimento.

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

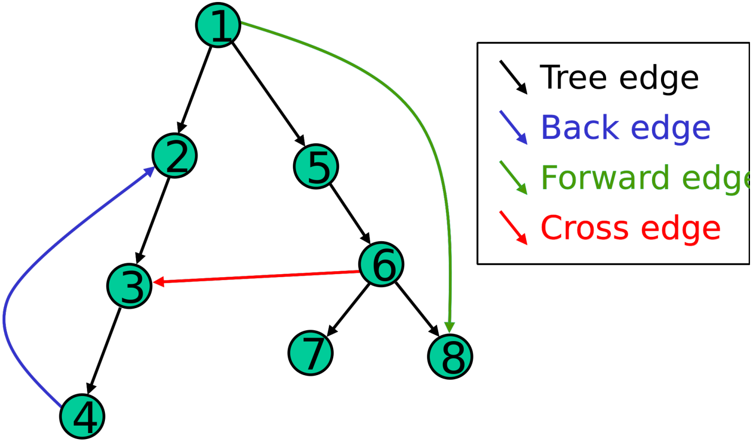


Figura: Tipos de aresta.

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

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

Detecção de Ciclos

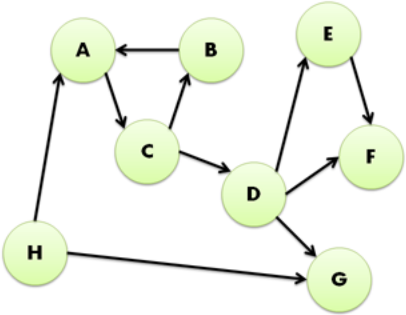


Figura: Detecção de Ciclos

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

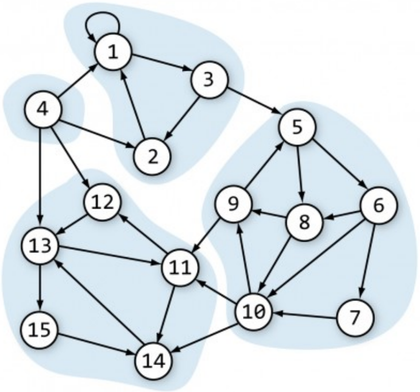


Figura: Componentes fortemente conexas.

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

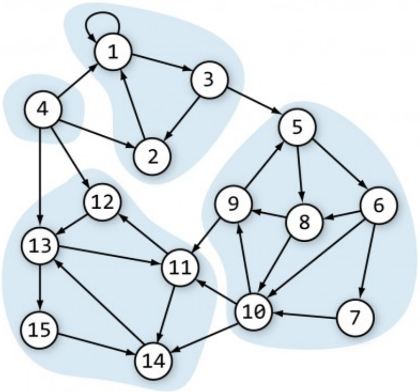


Figura: Componentes fortemente conexas.

Componentes Fortemente Conexas

Algorithm 9: FIND-STRONG-COMPONENTS(G)

Input: G

Output: Componentes Fortemente Conexas de G

```
list ← [ ]
S ← ∅
n ← 0
for all( v ∈ 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}
if( v.index = v.low )
    list.INSERT(CREATE-NEW-COMPONENT(S, v))
```

Componentes Fortemente Conexas

Algorithm 11: $\textsc{create-new-component}(\&S, v)$

Input: $\&S, v$

Output: Componentes Fortemente Conexa que inclui o nó v

$list' \leftarrow [\,]$

repeat

    $w = S.\textsc{pop}()$

    $list'.\textsc{insert}(w)$

    $w.color = BLACK$

until $w \neq v$

return $list'$

Componentes Fortemente Conexas

Complexidade

- O custo é de uma DFS pelo grafo.
- $O(|V| + |E|)$.

Problemas

Definição (Ponto de Articulação)

- Tome um grafo não-direcionado.
- Um ponto de articulação é um vértice cuja retirada desconecta o grafo.
- Um grafo que não possui pontos de articulação é 2-conexo.

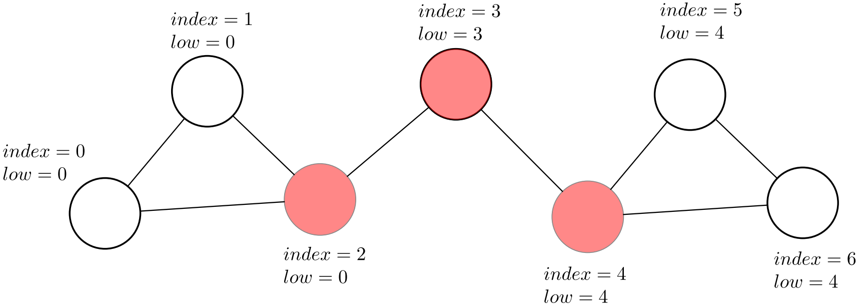
Pontos de Articulação

- Em muitas aplicações, pontos de articulação são críticos e precisam ser detectados.
- Usando uma busca em profundidade adaptada é possível dizer se um grafo tem ou não tem pontos de articulação.
- Utilizando o mesmo conceito de $index$ e low do problema anterior é possível detectar pontos de articulação.

Pontos de Articulação

- Suponha um nó u e um vizinho v de u. Caso $v.low \geq u.index$ significa que v não consegue alcançar nenhum nó com $index$ menor que o de u, isto é, os ancestrais de u considerando a DFS.
- Isto implica que u é um ponto de articulação!
- Exceção: nó raiz da DFS. Ele é uma articulação caso tenha 2 ou mais vizinhos considerando a **árvore de DFS**.

Pontos de Articulação



Pontos de Articulação

Algorithm 12: ARTICULATION-POINT$(G, u, \&n)$

Input: G, u, n
Output: Marcação de pontos de articulação partindo do nó u

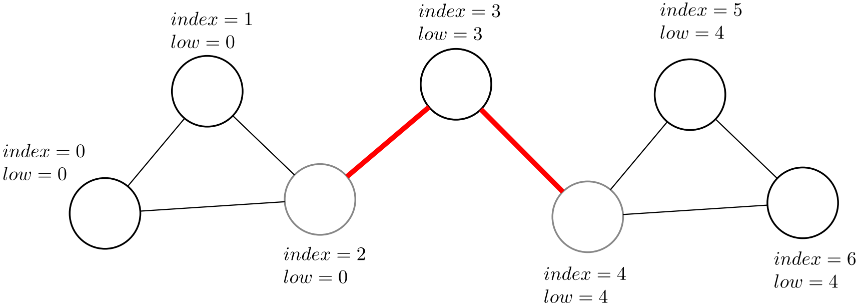
$u.index \leftarrow n$
$u.low \leftarrow n$
$n \leftarrow n + 1$
$u.color \leftarrow BLACK$
$children \leftarrow 0$
for all($(u, v) \in E$)
  if($v.color = WHITE$) `// v não visitado`
    $children++$
    $v.parent \leftarrow u$
    ARTICULATION-POINT(G, v, n, B)
    $u.low \leftarrow \min(u.low, v.low)$
    if($u.parent = \perp \wedge children > 1$) `// AP: u é a raiz da DFS e possui mais de um filho.`
      $u.ap \leftarrow true$
    if($u.parent \neq \perp \wedge v.low \geq u.index$)
      `// se o vizinho de u tem v.low ≥ index.u, então u é AP.`
      $u.ap \leftarrow true$
  else if($v \neq u.parent$)
    `// Detecção de back-edge que não forme um ciclo imediato.`
    $u.low \leftarrow min(u.low, v.index)$

Problemas

Definição (Pontes)

- Tome um grafo não-direcionado.
- O conceito de ponte é o análogo do conceito de ponto de articulação para arestas.
- Uma ponte é qualquer aresta cuja retirada desconecta o grafo.

Detecção de Pontes



Detecção de Pontes

- Para detectar pontes podemos usar o algoritmo anterior com uma pequena modificação.
- A partir do momento que temos um nó u e um vizinho v de u e $v.low > u.index$, isso significa que v não consegue alcançar o nó u através de outro caminho, logo, a aresta (u, v), se retirada, desconecta o grafo.

Detecção de Pontes

Algorithm 13: BRIDGE-DETECTION$(G, u, \&n, \&B)$

Input: G, u, n, B

Output: Marcação de pontos de articulação partindo do nó u

```
u.index ← n
u.low ← n
n ← n + 1
u.color ← BLACK
children ← 0
for all( (u, v) ∈ E )
    if( v.color = WHITE ) // v não visitado
        v.parent ← u
        BRIDGE-DETECTION(G, v, n)
        u.low ← min(u.low, v.low)
        if( v.low ≥ u.index )
            // se o vizinho de u tem v.low > index.u, então (u, v) é uma ponte.
            B.insert((u, v))
    else if( v ≠ u.parent )
        // Detecção de back-edge que não forme um ciclo imediato.
        u.low ← min(u.low, v.index)
```

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:

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

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

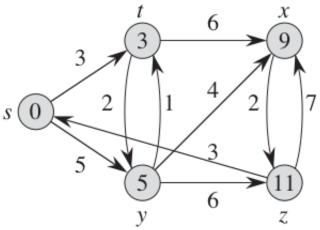


Figura: Grafo $G(V, E)$ com as respectivas distâncias de uma origem.

Menor Caminho

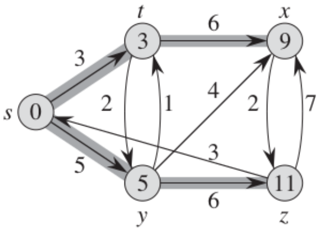


Figura: Menor rota até um destino.

Menor Caminho

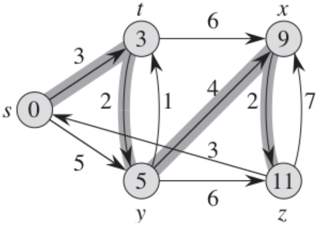


Figura: Menor rota até um destino.

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 14: 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 15: DIJKSTRA

Input: G, w, s

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

```
INITIALIZE-DJIKSTRA(G, s)
Q.INSERT(s) // Fila de prioridades
while ¬Q.EMPTY do
    u ← Q.EXTRACT-MIN()
    visited[u] ← true
    for all( (u, v) ∧ ¬visited[v] )
        if( v.d > u.d + w(u, v) )
            v.d ← u.d + w(u, v)
            v.π ← u
            Q.INSERT-UPDATE(v) // Insere ou atualiza o nó na
                fila
```

Algoritmo de Djikstra

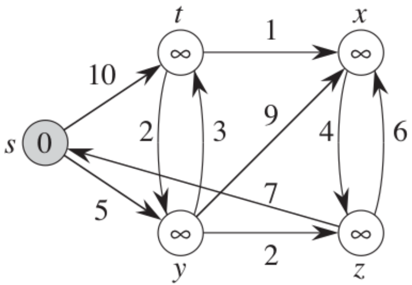


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

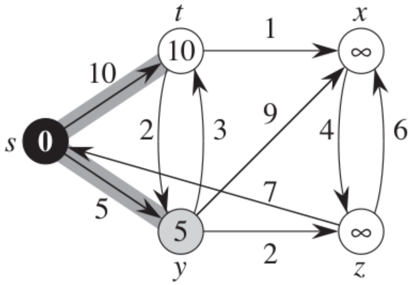


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

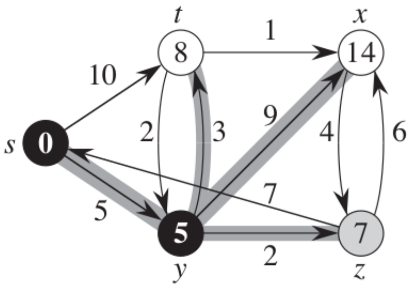


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

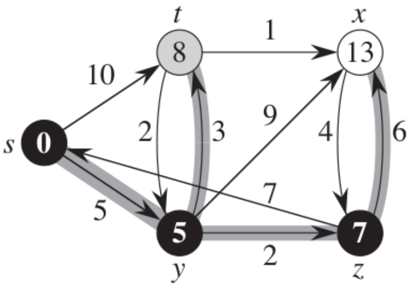


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

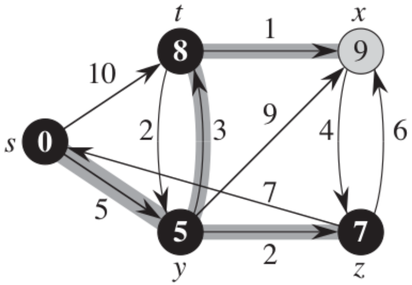


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

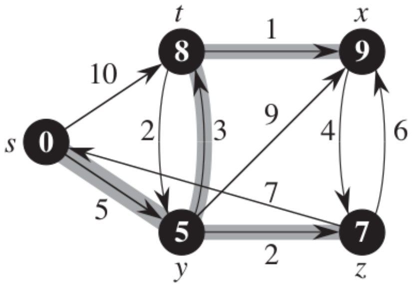


Figura: Algoritmo de Dijkstra.

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

Sumário

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.

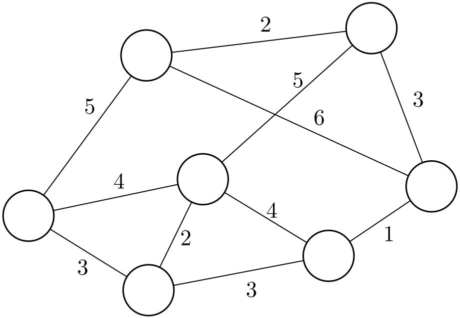
Árvore Espalhada Mínima

Menor custo

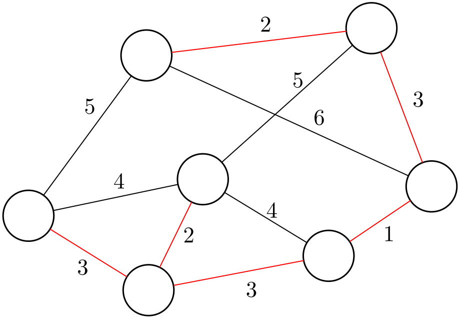
- Obviamente, o subgrafo gerado de menor custo tem que ser uma árvore (ou uma floresta).
- O custo desta árvore/floresta é dado pelo somatório de suas arestas:

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

Árvore Espalhada Mínima



Árvore Espalhada Mínima



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 16: 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

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 17: INITIALIZE-PRIM

Input: G, s

```
for all( v ∈ V )
    v.d ← ∞
    v.π ← NULL
s.d ← 0
```

Algoritmo de Prim

Algorithm 18: PRIM

Input: G, w, s
Output: $MST(G)$

INITIALIZE-PRIM(G, s)
Q.INSERT(s)
$T \leftarrow s$
$last \leftarrow s$
while $\neg Q$.EMPTY **do**
    $u \leftarrow Q$.EXTRACT-MIN$()$
    $visited[u] \leftarrow true$
    $T \leftarrow T \cup (last, u)$
    $last \leftarrow u$
    for all($(u, v) \wedge \neg visited[v]$)
        if($v.d > w(u, v)$)
            $v.d \leftarrow w(u, v)$
            $v.\pi \leftarrow u$
            Q.INSERT-UPDATE(v)
return T

Algoritmo de Prim

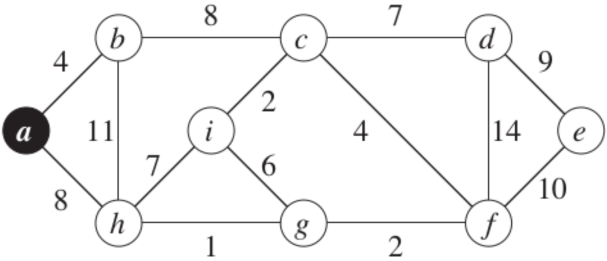


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

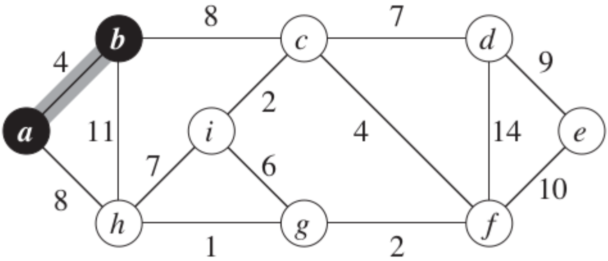


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

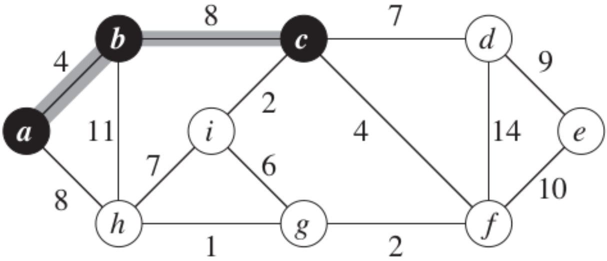


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

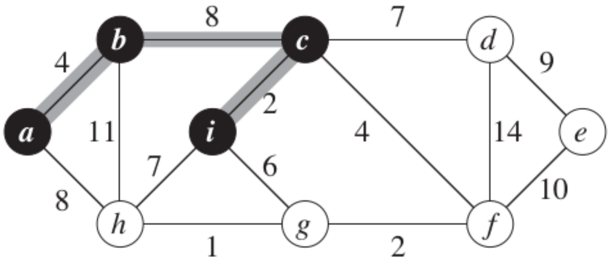


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

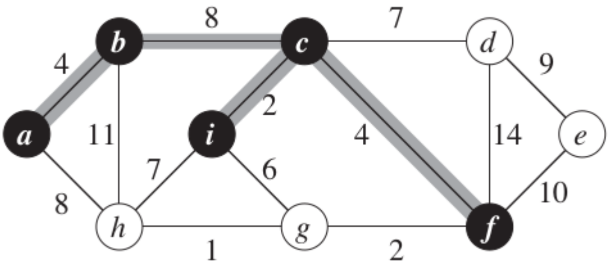


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

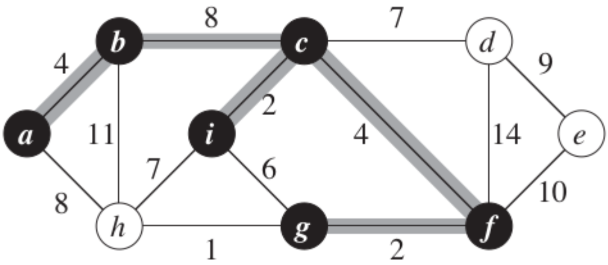


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

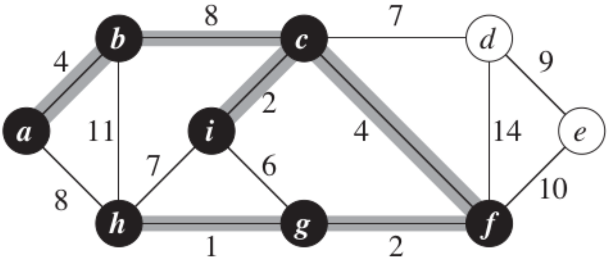


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

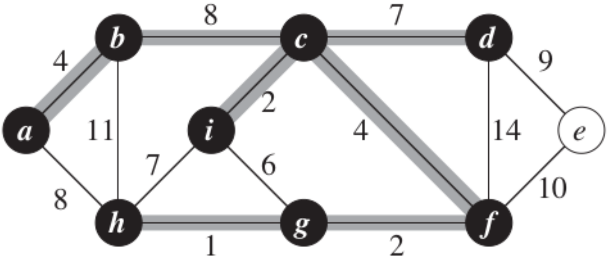


Figura: Execução do Algoritmo de Prim

Algoritmo de Prim

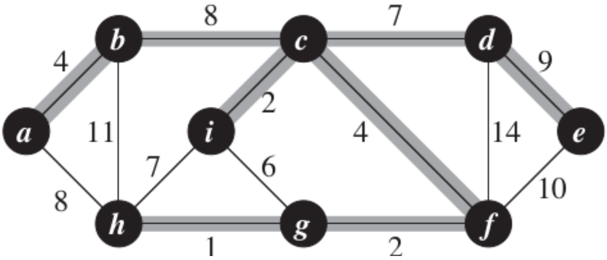


Figura: Execução do Algoritmo de Prim

Sumário

Algoritmo de Kruskal

- O algoritmo de Kruskal tem uma filosofia bem simples.
- Ordena-se as arestas em ordem crescente de custo.
- Para cada aresta (u, v) em ordem crescente, acrescente a aresta (u, v) bem como os vértices u e v na árvore espalhada mínima somente se u e v não estão na mesma componente conexa.
- Isso evita a formação de ciclos.
- Seja t_s o custo da ordenação, t_e o custo de varrer as arestas e t_{uf} o custo de verificar se dois vértices estão em uma mesma componente conexa.
- O custo total do algoritmo é $\Theta(t_s + t_e \cdot t_{uf})$.

Algoritmo de Kruskal

- Assumindo que:
  - O tempo de ordenação leve $t_s \in \Theta(|E| \lg |E|) = \Theta(|E| \lg |V|)$.
- O custo total do algoritmo é $\Theta(|E| \lg |V| + |E| \cdot t_{uf})$
- Caso uma estrutura eficiente que possibilite checar se dois vértices estejam na mesma componente conexa seja utilizada, conseguimos um algoritmo $\Theta(|E| \lg |V|)$.

Algoritmo de Kruskal

Estruturas de Dados Para Conjuntos Disjuntos

- Suponha $\mathcal{S} = (\{s_0\}, \{s_1\}, \ldots, \{s_n\})$ conjuntos *singleton*.
- Deseja-se implementar duas operações básicas:
  - $\textsc{union}(A, B)$, une dois conjuntos A e B.
  - $\textsc{find}(x)$, diz em qual conjunto encontra-se o elemento x.
- É interessante observar que, após um número qualquer de sucessivas aplicações de UNION, o que se tem são conjuntos disjuntos, isto é, conjuntos cuja interseção é vazia.

Algoritmo de Kruskal

Estruturas de Dados Para Conjuntos Disjuntos

- Para simplificar, para cada conjunto, vamos eleger um representante.
- Assim, dois elementos estão na mesma coleção se e somente se eles possuem o mesmo representante.
- Inicialmente, cada singleton tem como representante ele mesmo.
- Desta forma, a operação de $\text{FIND}(x)$ pode simplesmente retornar o representante do conjunto no qual x está incluído.
- Assim, dados dois elementos, x e y, conseguimos saber se eles estão no mesmo conjunto verificando se $\text{FIND}(x) = \text{FIND}(y)$.

Algoritmo de Kruskal

Estruturas de Dados Para Conjuntos Disjuntos

- Podemos modelar isso computacionalmente através de uma floresta.
- Cada *singleton* x inicialmente faz parte de uma árvore e possui dois valores:
  - $x.parent$: diz quem é o pai de x (inicialmente é ele mesmo).
  - $x.rank$ diz o rank de x, que é uma cota superior da altura da árvore com raiz em x.
- Para achar o representante do conjunto que x está incluso, basta seguir os ponteiros de $parent$ até chegar na raiz.
- Logo, já sabemos implementar FIND.

Algoritmo de Kruskal

Estruturas de Dados Para Conjuntos Disjuntos

Algorithm 19: $\textsc{Find}(x)$

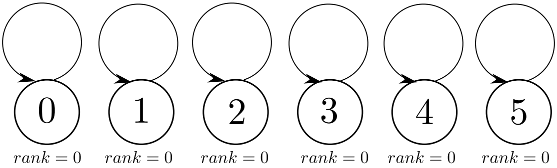
Input: x

Output: representante do conjunto de x

```
if( x.parent ≠ x )
    return FIND(x.parent)
return x
```

Algoritmo de Kruskal

Estruturas de Dados Para Conjuntos Disjuntos



Algoritmo de Kruskal

Estruturas de Dados Para Conjuntos Disjuntos

- Para implementar a operação de $\text{UNION}(x, y)$, achamos o representante dos conjuntos que x e y se encontram, chame-os de x' e y'. Se $x' \neq y'$, significa que x e y estão em conjuntos diferentes. Logo só é necessário unir as duas árvores fazendo com que y' vire filho de x' ou x' vire filho de y'.
- Qual opção escolher?
- Escolhemos a árvore de maior $rank$ para abrigar a de menor $rank$.

Algoritmo de Kruskal

Estruturas de Dados Para Conjuntos Disjuntos

Algorithm 20: $\text{UNION}(x, y)$

Input: x, y

$x' \leftarrow \text{FIND}(x)$
$y' \leftarrow \text{FIND}(y)$
if($x' \neq y'$)
  if($x'.rank > y'.rank$)
    $y'.parent \leftarrow x'$
  else
    $x'.parent \leftarrow y'$
    if($x'.rank = y'.rank$)
      $y'.rank + +$

Estruturas de Dados para Conjuntos Disjuntos

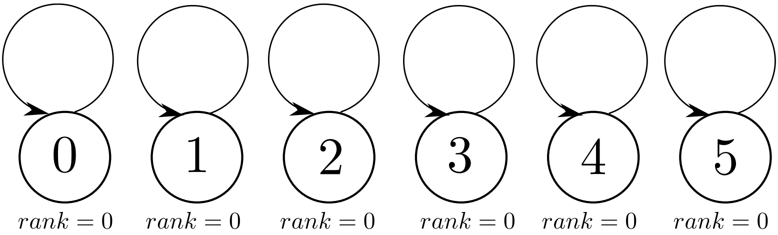


Figura: Estado inicial.

Estruturas de Dados para Conjuntos Disjuntos

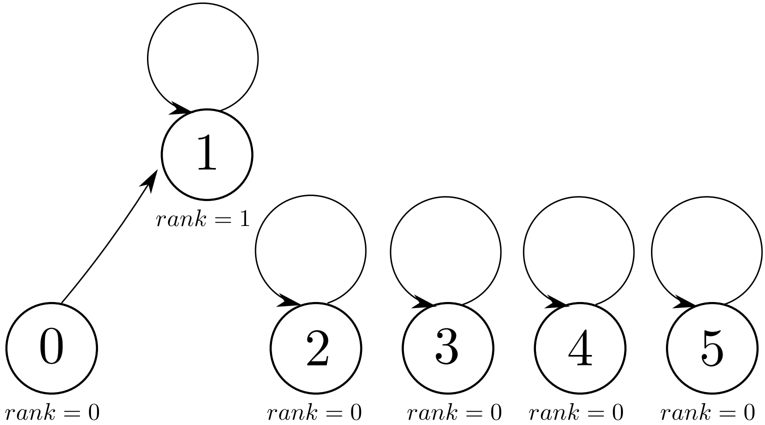


Figura: $\textsc{union}(0, 1)$

Estruturas de Dados para Conjuntos Disjuntos

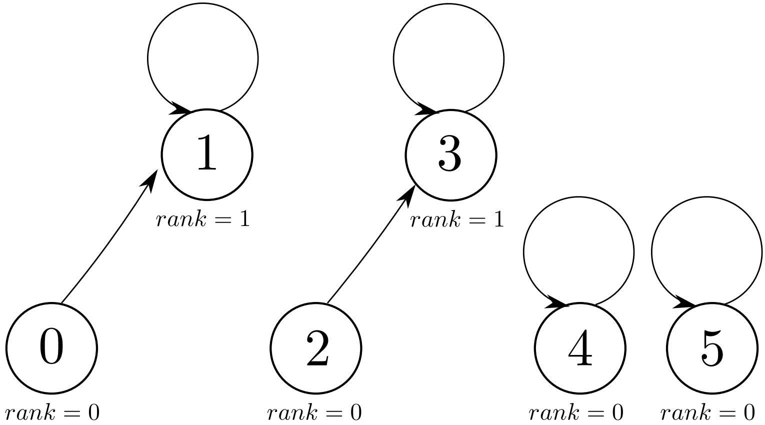


Figura: UNION$(2, 3)$

Estruturas de Dados para Conjuntos Disjuntos

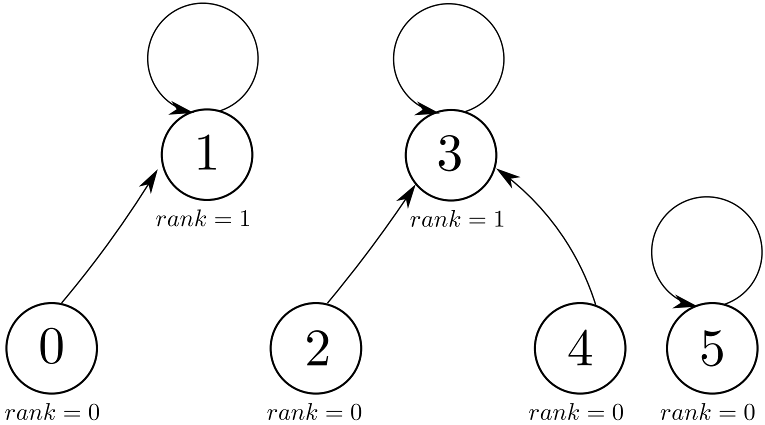


Figura: $\text{UNION}(2, 4)$

Estruturas de Dados para Conjuntos Disjuntos

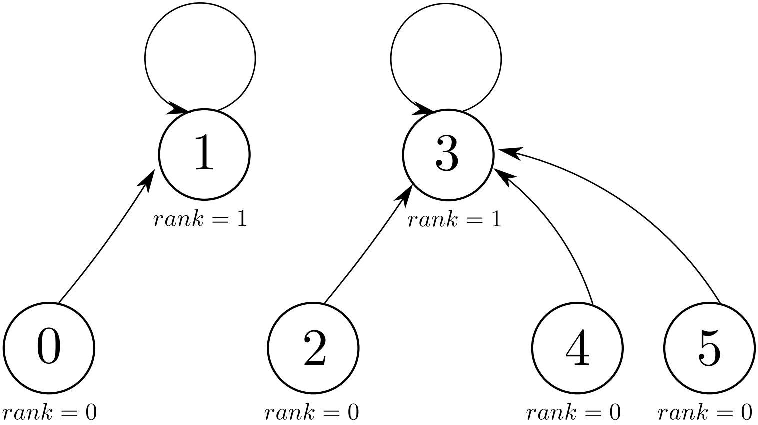


Figura: $\text{UNION}(2, 5)$

Estruturas de Dados para Conjuntos Disjuntos

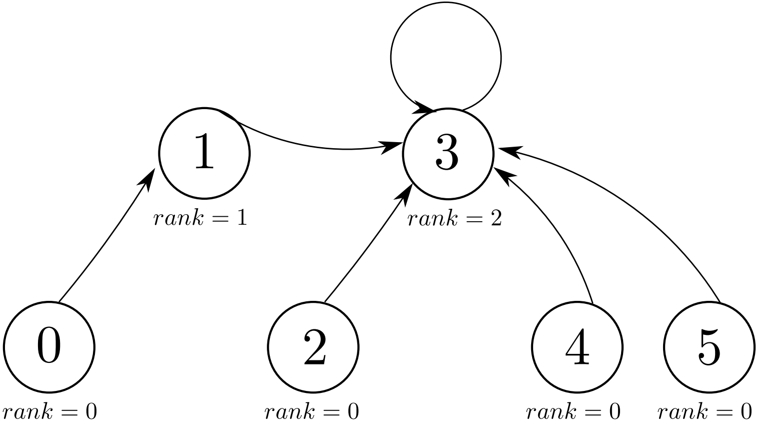


Figura: $\textsc{union}(0, 3)$

Algoritmo de Kruskal

Estruturas de Dados Para Conjuntos Disjuntos

- Da forma como estão implementados UNION e FIND, a árvore final pode ter altura $\Theta(n)$, o que ocasionaria um tempo linear para as consultas.
- Contudo é possível melhorar esse tempo significativamente ao utilizar uma técnica chamada de **path-compression**.
- Ao realizar a consulta de FIND(x), atualizamos x e os demais nós no caminho até a raiz para apontarem imediatamente para o representante do conjunto, isto é, a raiz.

Algoritmo de Kruskal

Estruturas de Dados Para Conjuntos Disjuntos

- Dadas M operações de UNION ou FIND complexidade final amortizada utilizando a técnica de **path-compression** é:

$$\Theta(M\alpha(n)) \subsetneq \Theta(M \lg^* n) \subsetneq \Theta(M \lg n)$$

em que α é a função de Ackermann inversa.

Algoritmo de Kruskal

Estruturas de Dados Para Conjuntos Disjuntos

Algorithm 21: $\textsc{Find}(x)$ com path-compression

Input: x

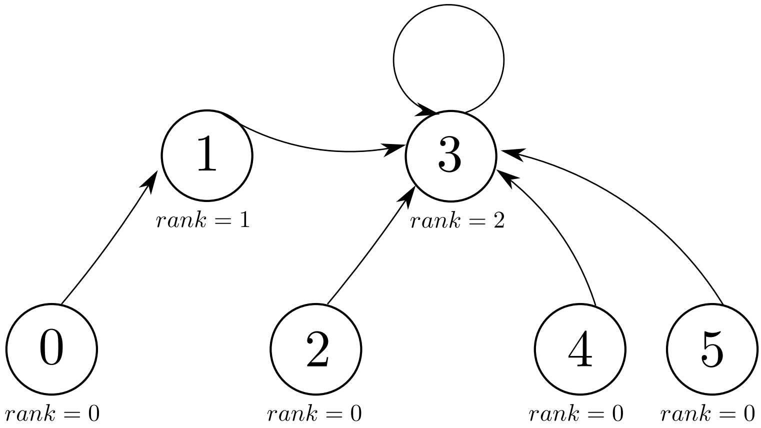
Output: representante do conjunto de x

if($x.parent \neq x$)

  $x.parent \leftarrow \textsc{Find}(x.parent)$

return x

Estruturas de Dados para Conjuntos Disjuntos



Estruturas de Dados para Conjuntos Disjuntos

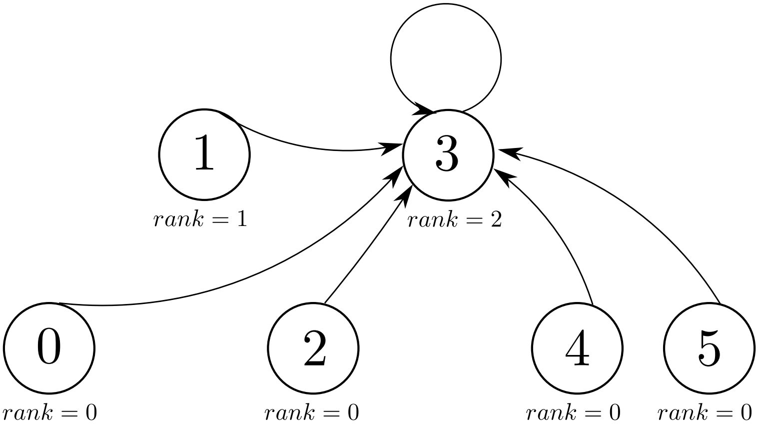


Figura: $\textsc{find}(0)$

Algoritmo de Kruskal

Estruturas de Dados Para Conjuntos Disjuntos

- Todas as operações que sucedem uma compressão de caminho se beneficiarão do tamanho reduzido da altura.
- Note que, após a compressão de caminho, o $rank$ da raiz não corresponde mais a altura da árvore, então $rank$ deve ser sempre interpretado como uma cota superior para a altura.

Algoritmo de Kruskal

- Suponha agora a existência de uma estrutura de dados UF que provê as operações de $\textsc{union}(x, y)$, $\textsc{find}(x)$ como discutido anteriormente e $\textsc{sameSet}(x, y)$. Esta última operação responde se x e y estão no mesmo conjunto e para respondê-la é necessário apenas verificar se $\textsc{find}(x) = \textsc{find}(y)$.
- O algoritmo de Kruskal inicializa os vértices, numerados e 0 a $|V| - 1$ em n singletons.
- Esta inicialização pode ser feita facilmente por um método $\textsc{makeSet}$.

Algoritmo de Kruskal

- Em seguida, as arestas são ordenadas em ordem crescente do peso.
- Por fim, para cada aresta (u, v) na ordem dada, pegamos ela para a árvore espalhada desde que $UF.\text{SAMESET}(u, v)$ seja falso, isto é, desde que u e v estejam em componentes conexas distintas. Em seguida, u e v são colocadas na mesma componente conexa através de $UF.\text{UNION}(u, v)$.

Algoritmo de Kruskal

Algorithm 22: KRUSKAL(G)

Input: G

Output: T, a árvore espalhada mínima

$UF.\text{MAKESET}(n)$
$T \leftarrow (\emptyset, \emptyset)$
SORT($G.E$) `// ordena-se as arestas pelo custo`
for all($(u, v) \in E$)
    if($\neg UF.\text{SAMESET}(u, v)$)
        $T.V \leftarrow T.V \cup \{u, v\}$
        $T.E \leftarrow T.E \cup (u, v)$
        $UF.\text{UNION}(u, v)$
return T