Projeto e Análise de Algoritmos

Buscas em grafos: Busca em Profundidade

Atílio G. Luiz

Primeiro Semestre de 2024

Busca em profundidade

Busca em largura × busca em profundidade

Na busca em largura:

- Dado o conjunto não vazio Q de vértices alcançados pela origem s e ainda não finalizados, selecionamos o vértice que está há **mais tempo** em Q para continuar a busca.
- Logo, Q é uma fila.

Na busca em profundidade:

- Dado o conjunto não vazio Q de vértices alcançados pela origem s e ainda não finalizados, selecionamos o vértice que está há **menos tempo** em Q para continuar a busca.
- Logo, Q é uma pilha.

Busca em profundidade

Ideia do algoritmo

- começamos com o vértice de origem s
- para cada vizinho não visitado v do vértice atual
  1. adicionamos uma aresta (u, v) à árvore de busca
  2. visitamos recursivamente a partir de v

Floresta de busca

Visitando todos os vértices

- a árvore de busca contém só vértices alcançáveis de s
- algumas vezes queremos visitar todos os vértices
- repetimos o processo com os vértices não visitados
- obteremos uma floresta de busca

Representando uma floresta

- também utilizamos um vetor de pais π
- um vértice v com $\pi[v] = \text{NIL}$ é raiz de uma árvore de busca
- as arestas da floresta são

$$\{(\pi[v], v) : v \in V(G) \text{ e } \pi[v] \neq \text{NIL}\}$$

Cores dos vértices

De novo, vamos pintar o grafo durante a busca

1. $cor[v] = \text{branco}$ se não descobrimos v ainda
2. $cor[v] = \text{cinza}$ se já descobrimos, mas não finalizamos v
3. $cor[v] = \text{preto}$ se já descobrimos e já finalizamos v

Observações

- os vértices cinza têm suas chamadas recursivas ativas
- a pilha de chamadas induz um caminho na floresta

Tempo de descoberta e finalização

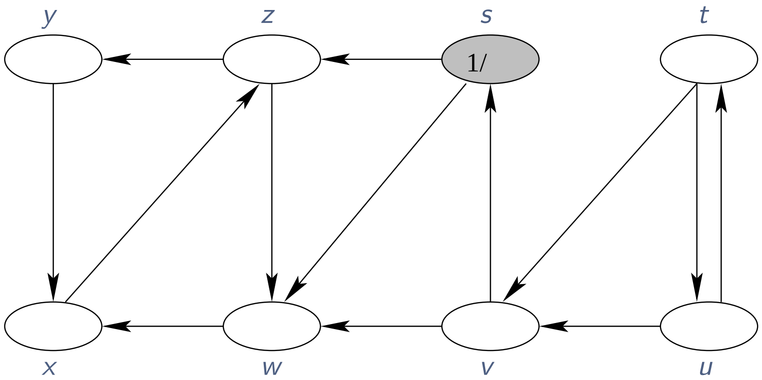
Vamos associar rótulos aos vértices

- $d[v]$ é instante de descoberta de v
- $f[v]$ é instante de finalização de v

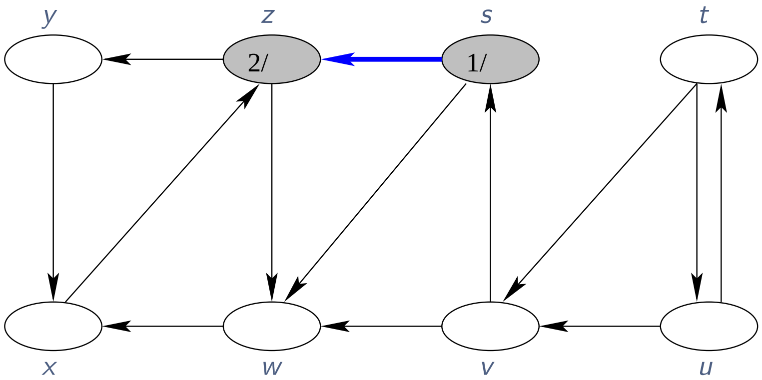
Observações

- os rótulos são inteiros distintos entre 1 e $2|V|$
- refletem os instantes em que v muda de cor

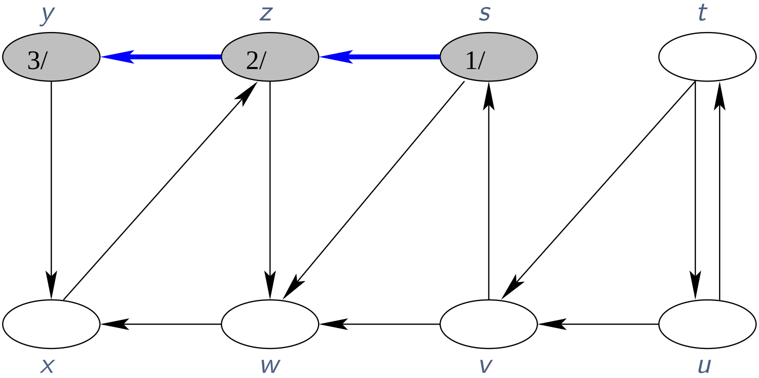
Exemplo de busca em profundidade



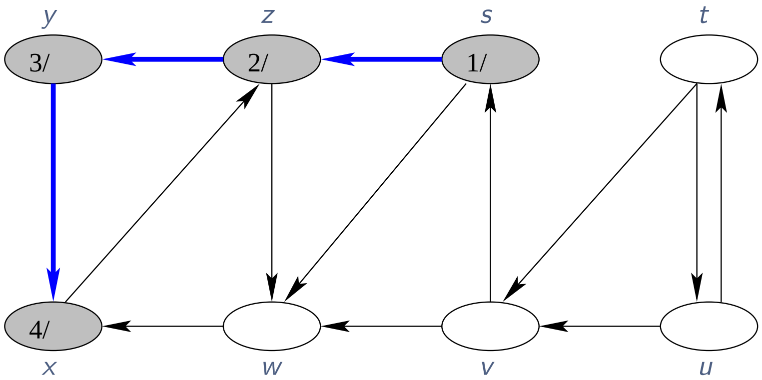
Exemplo de busca em profundidade



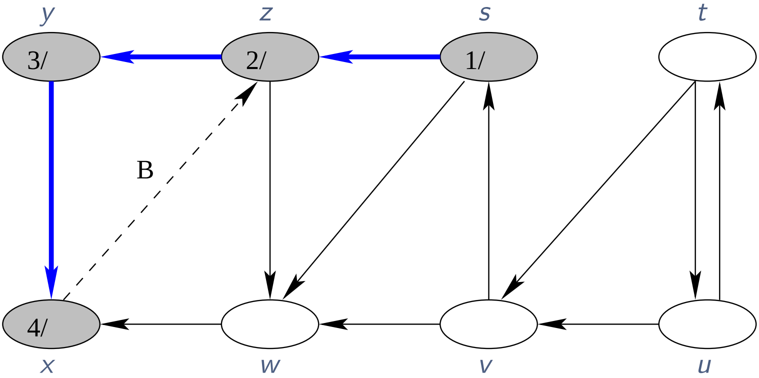
Exemplo de busca em profundidade



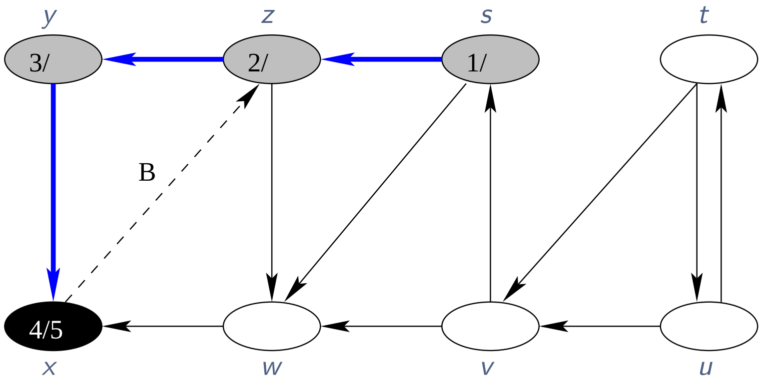
Exemplo de busca em profundidade



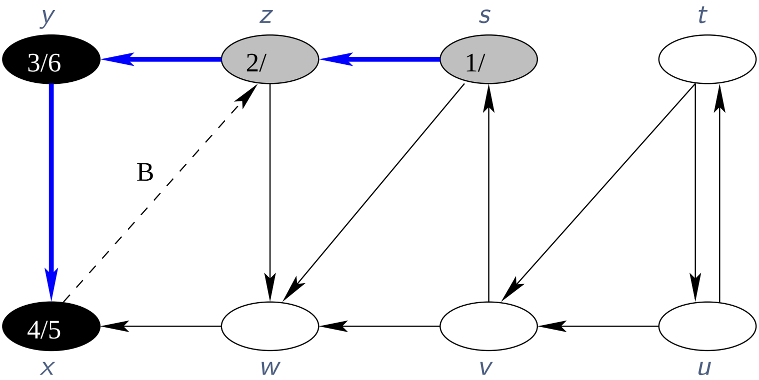
Exemplo de busca em profundidade



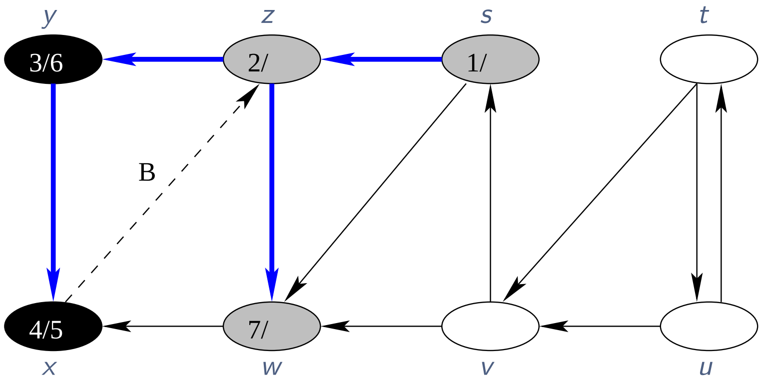
Exemplo de busca em profundidade



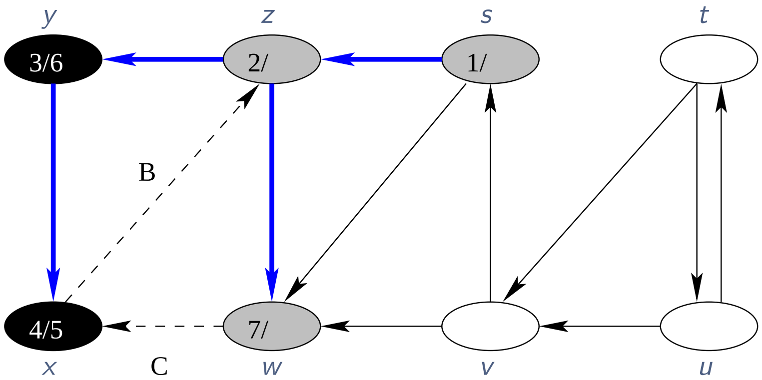
Exemplo de busca em profundidade



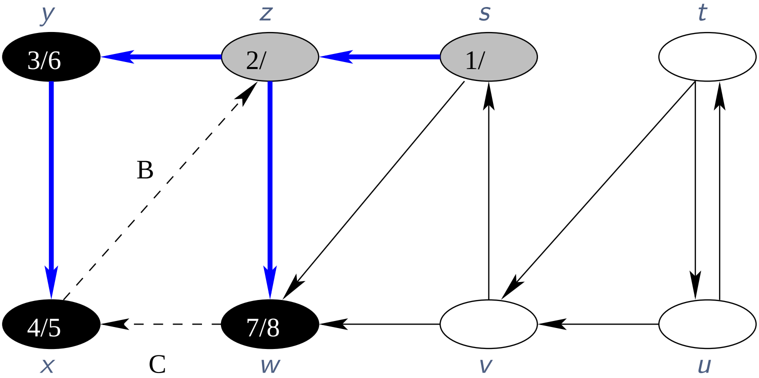
Exemplo de busca em profundidade



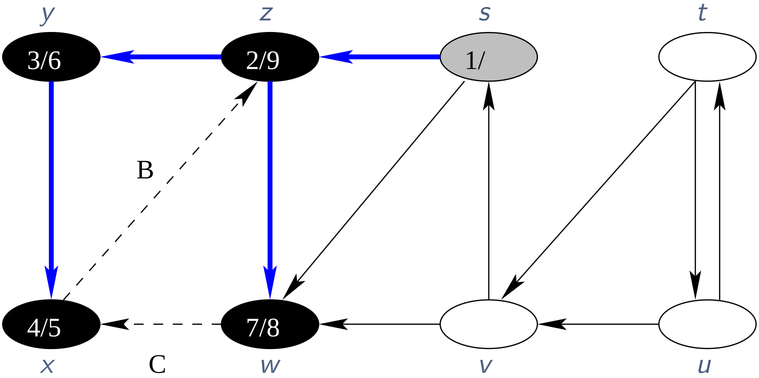
Exemplo de busca em profundidade



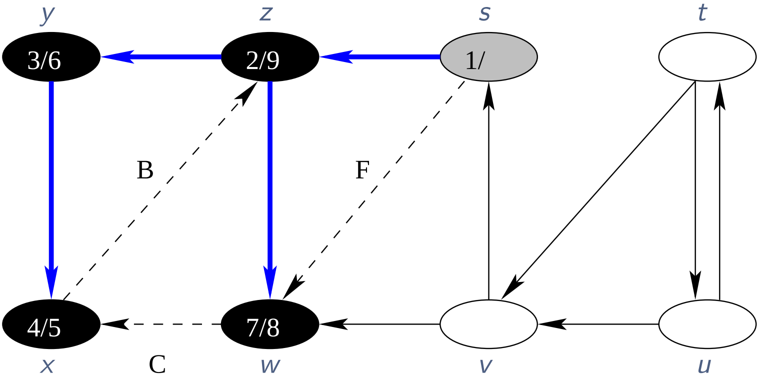
Exemplo de busca em profundidade



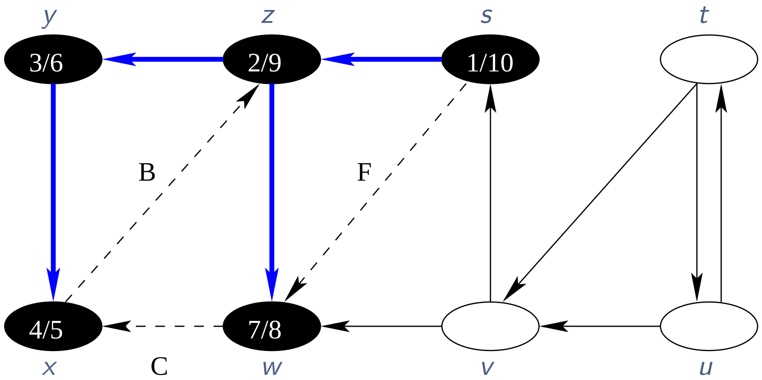
Exemplo de busca em profundidade



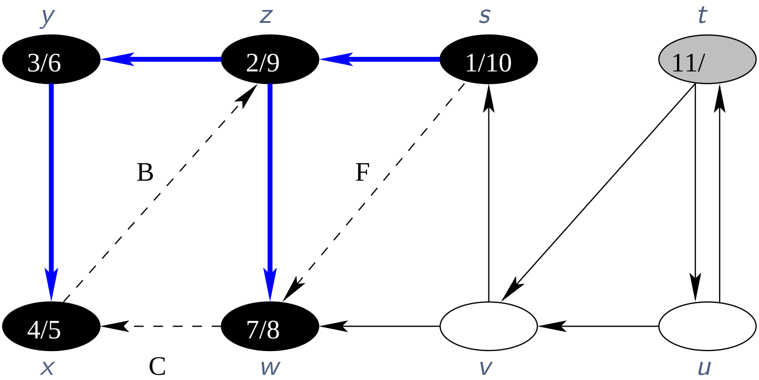
Exemplo de busca em profundidade



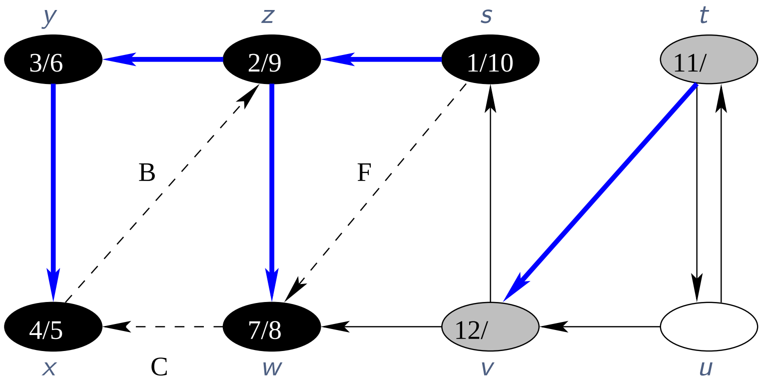
Exemplo de busca em profundidade



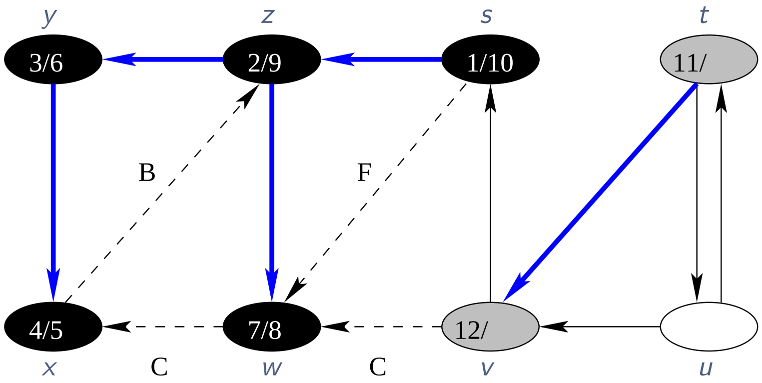
Exemplo de busca em profundidade



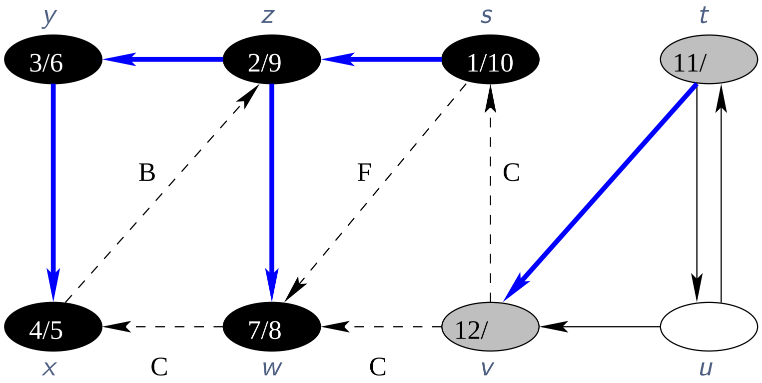
Exemplo de busca em profundidade



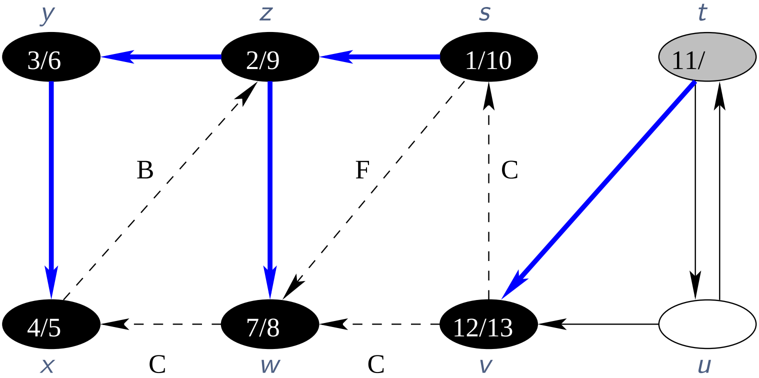
Exemplo de busca em profundidade



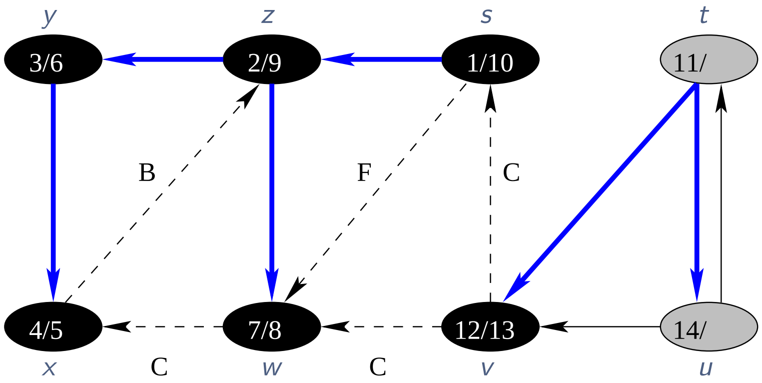
Exemplo de busca em profundidade



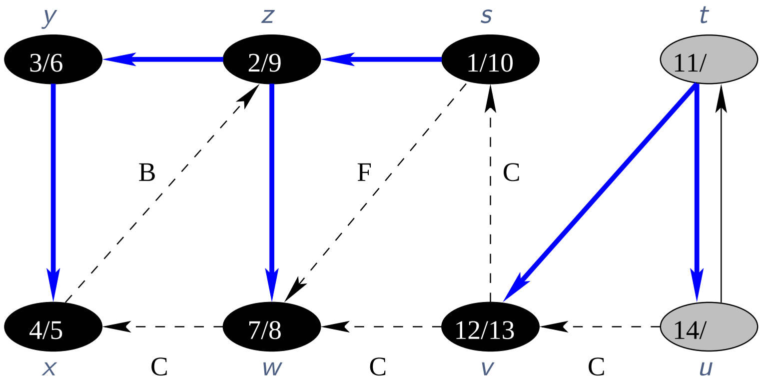
Exemplo de busca em profundidade



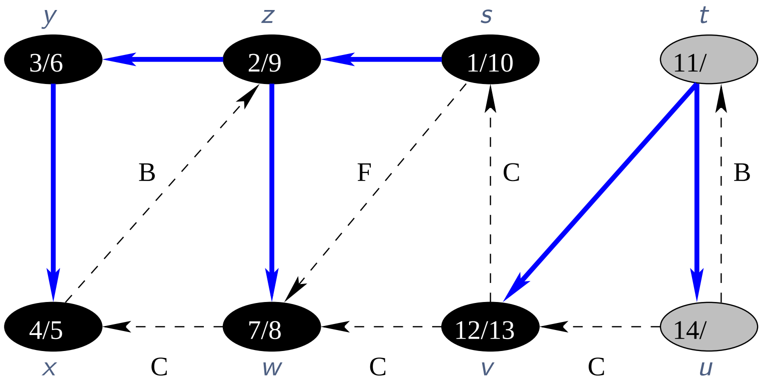
Exemplo de busca em profundidade



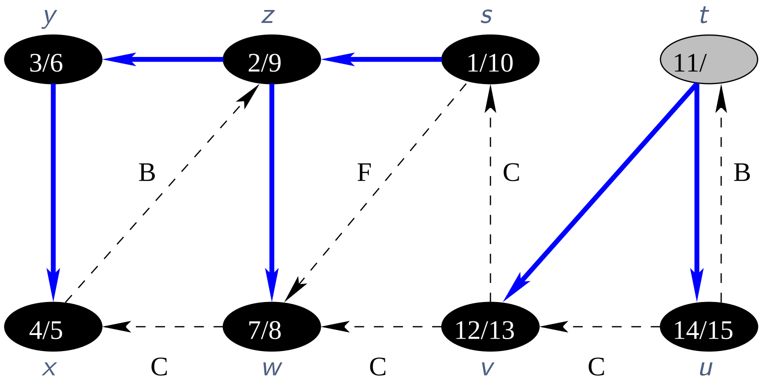
Exemplo de busca em profundidade



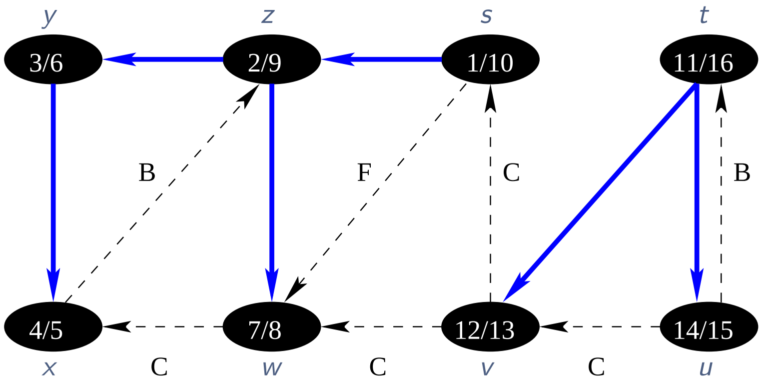
Exemplo de busca em profundidade



Exemplo de busca em profundidade



Exemplo de busca em profundidade



Rótulos versus cores

Observe que para todo vértice v

- v é branco antes do instante $d[v]$
- v é cinza entre os instantes $d[v]$ e $f[v]$
- v é preto após o instante $f[v]$

Algoritmo DFS

```
DFS(G)
    para cada u ∈ V(G) faça
        cor[u] = branco
        π[u] = NIL
    tempo = 0
    para cada u ∈ V(G) faça
        se cor[u] == branco então
            DFS-VISIT(u)
```

- representamos G com listas de adjacências
- a floresta de busca em profundidade é representada por π
- calcula os instantes $d[v]$ e $f[v]$

Algoritmo DFS-VISIT

DFS-visit(u)

```
tempo = tempo + 1
d[u] = tempo
cor[u] = cinza
para cada v ∈ Adj[u] faça
    se cor[v] == branco então
        π[v] = u
        DFS-VISIT(v)
tempo = tempo + 1
f[u] = tempo
cor[u] = preto
```

- constrói uma arvore de busca com origem u

Análise de complexidade

Analisamos o tempo do algoritmo principal DFS

- a inicialização consome tempo $O(V)$
- realizamos $|V|$ chamadas a DFS-VISIT

E o tempo da sub-rotina DFS-VISIT

- processamos cada vértice exatamente uma vez
- cada chamada percorre sua lista de adjacências
- o tempo gasto percorrendo adjacências é $O(E)$

A complexidade da busca em profundidade é $O(V + E)$

Propriedades da Busca em Profundidade

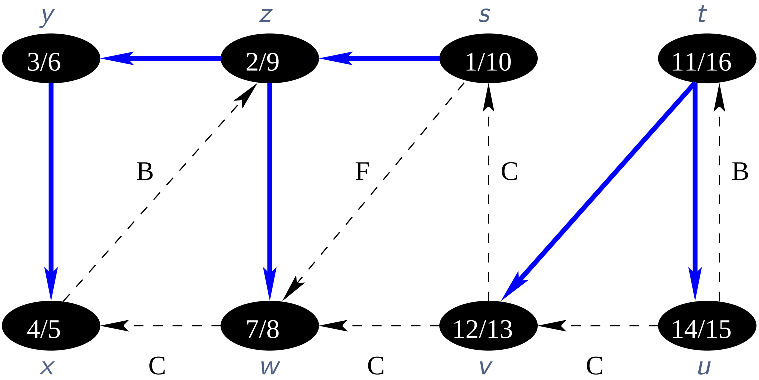
Estrutura de parênteses

Teorema dos parênteses

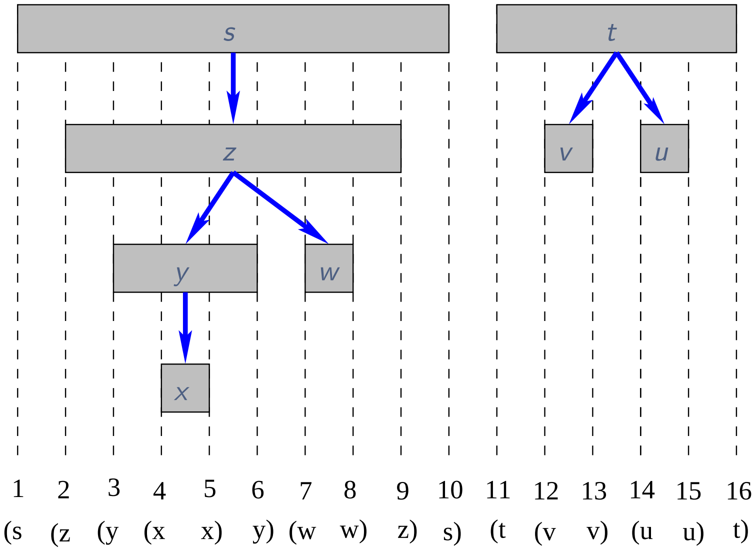
Se u e v são vértices de uma árvore de busca em profundidade, então ocorre exatamente um entre os três casos abaixo:

1. (a) os intervalos $[d[u], f[u]]$ e $[d[v], f[v]]$ são disjuntos
   (b) nesse caso u e v não são descendentes um do outro
2. (a) o intervalo $[d[u], f[u]]$ está contido em $[d[v], f[v]]$
   (b) nesse caso u é descendente de v
3. (a) o intervalo $[d[v], f[v]]$ está contido em $[d[u], f[u]]$
   (b) nesse caso v é descendente de u

Exemplo de floresta de busca



Exemplo de estrutura de parênteses



Demonstração do teorema dos parênteses

- Sem perda de generalizadade, podemos supor que $d[u] < d[v]$.
- analisamos dois casos:

Caso 1: suponha que $d[v] < f[u]$

- então v foi descoberto enquanto u era cinza, o que implica que v é um descendente de u
- e a chamada recursiva para v termina antes da de u, ou seja, $f[v] < f[u]$
- neste caso, $[d[v], f[v]]$ está contido em $[d[u], f[u]]$

Caso 2: suponha que $f[u] < d[v]$

- então u foi finalizado enquanto v era branco
- e a chamada de u termina antes que a de v comece
- portanto u e v não são descendentes um do outro
- neste caso, $[d[v], f[v]]$ e $[d[u], f[u]]$ são disjuntos. $\blacksquare$

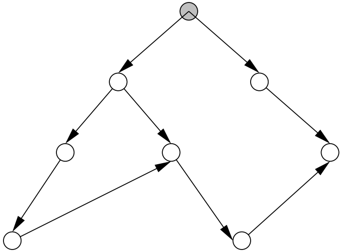
Vértices alcançáveis

Teorema do caminho branco

Considere dois vértices u e v. As seguintes afirmações são equivalentes:

(1) v é descendente de u na floresta de busca

(2) quando u foi descoberto, existia um caminho de u a v formado apenas por vértices brancos



Prova do teorema do caminho branco

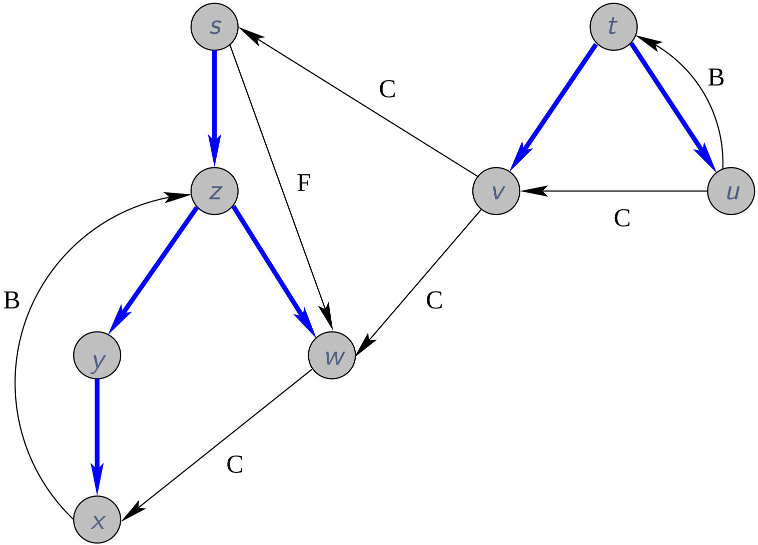
- $(1) \Rightarrow (2)$
  - suponha que v é um descendente próprio de u
  - pelo Teorema dos Parênteses (T.P.), $d[u] < d[v]$
  - portanto, v é branco no tempo $d[u]$
  - Como v pode ser qualquer descendente de u, todos os vértices no caminho de u a v na árvore de busca em profundidade eram brancos no tempo $d[u]$
- $(2) \Rightarrow (1)$
  - considere um caminho branco de u a v no instante $d[u]$
  - suponha que todo vértice no caminho virou descendente de u, com exceção do vértice v
  - seja w o vértice antecessor de v nesse caminho (w pode ser o próprio u)
  - como w é descendente de u, temos $f[w] \leq f[u]$ (T.P.)
  - como v é descoberto depois de u ser descoberto e antes de w ser finalizado, temos $d[u] < d[v] < f[w] \leq f[u]$
  - Pelo T.P. temos que $[d[v], f[v]]$ está inteiramente contido no intervalo $[d[u], f[u]]$, e v é descendente de u, contradição.

Classificação de arestas

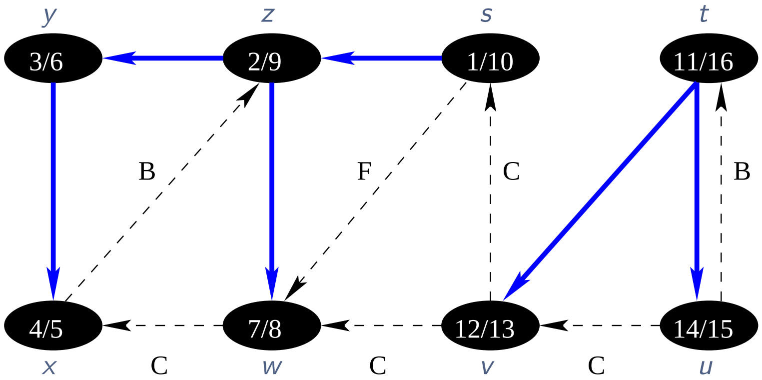
Dada a floresta de busca, podemos classificar arestas do grafo

- arestas de árvore (*tree edges*) são arestas da floresta de busca em profundidade
- arestas de retorno (*back edges*) ligam um vértice a um ancestral. Laços em grafos direcionados são arestas de retorno.
- arestas de avanço (*forward edges*) ligam um vértice a um descendente (não são arestas de árvore)
- arestas de cruzamento (*cross edges*) são todas as outras arestas do grafo

Classificação de arestas



Classificação de arestas



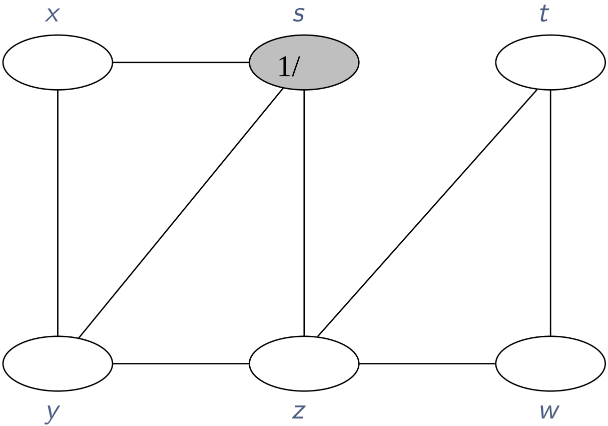
É fácil modificar o algoritmo $\mathrm{DFS}(G)$ para que ele também classifique as arestas de G. (Exercício)

DFS em grafos não direcionados

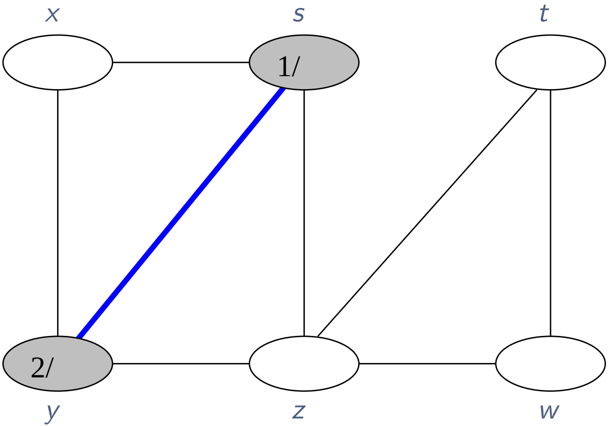
Classificando arestas de um grafo não direcionado:

- não pode haver aresta de avanço (por quê?)
- tampouco aresta de cruzamento (por quê?)
- daí cada aresta é aresta de árvore ou aresta de retorno

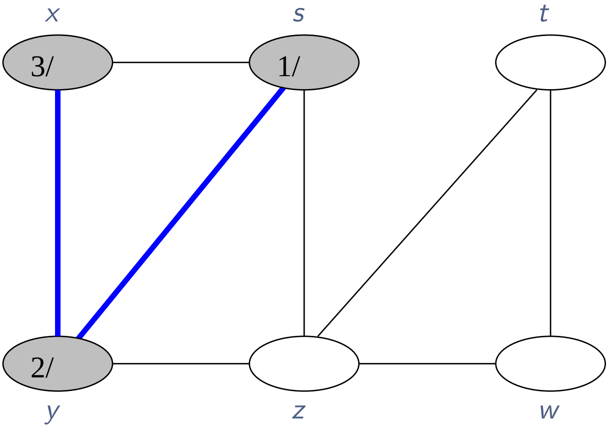
DFS em grafo não direcionado



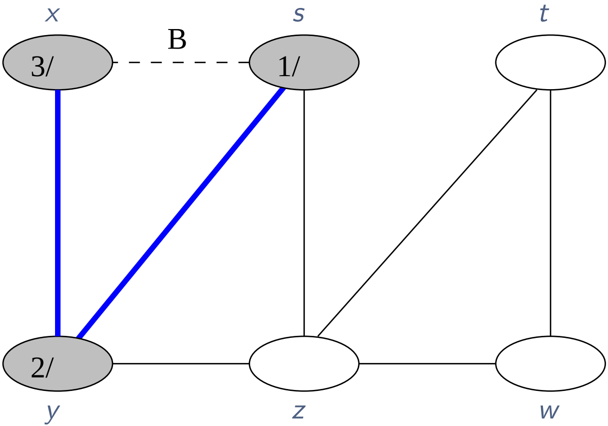
DFS em grafo não direcionado



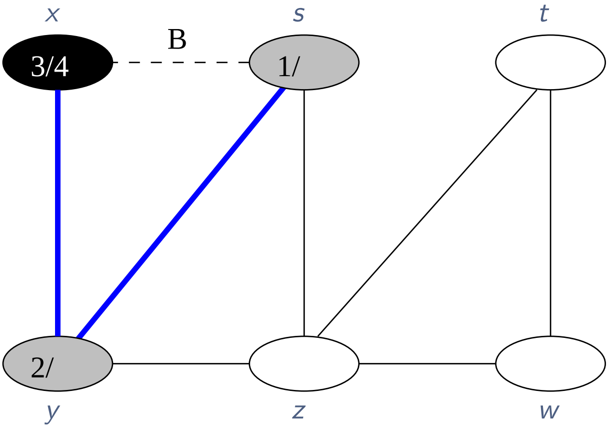
DFS em grafo não direcionado



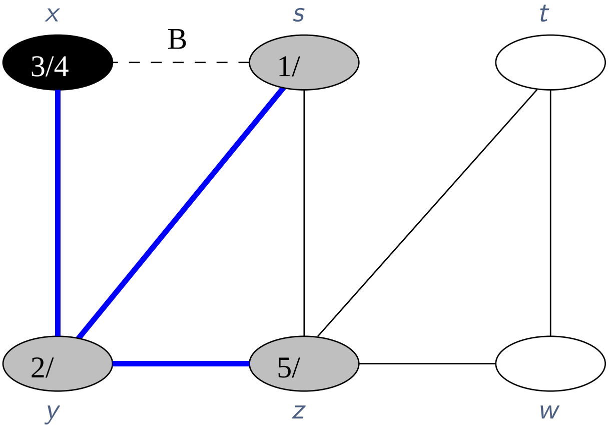
DFS em grafo não direcionado



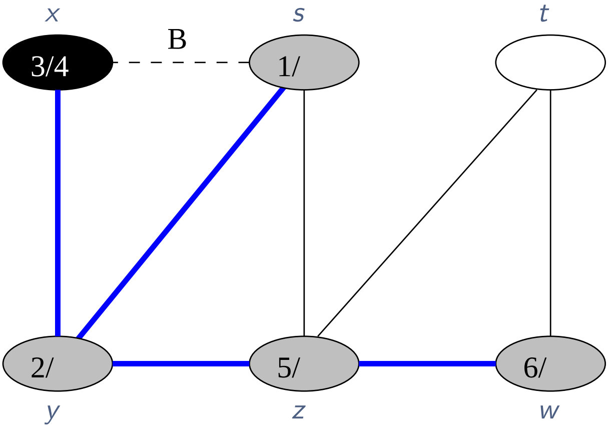
DFS em grafo não direcionado



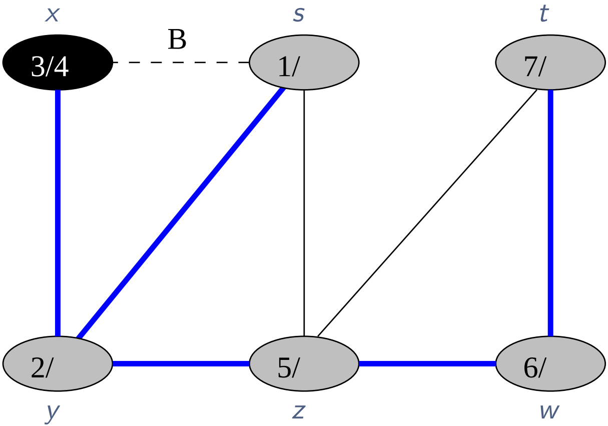
DFS em grafo não direcionado



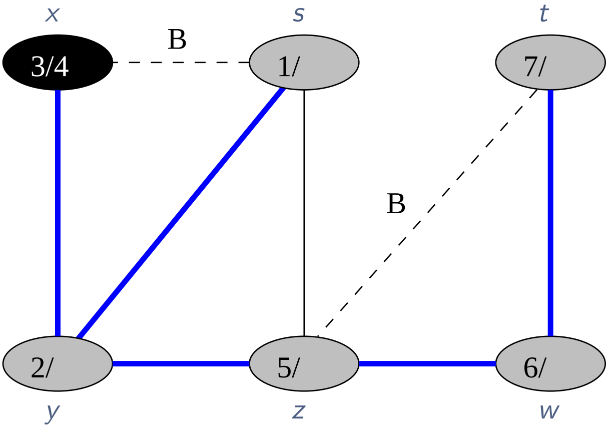
DFS em grafo não direcionado



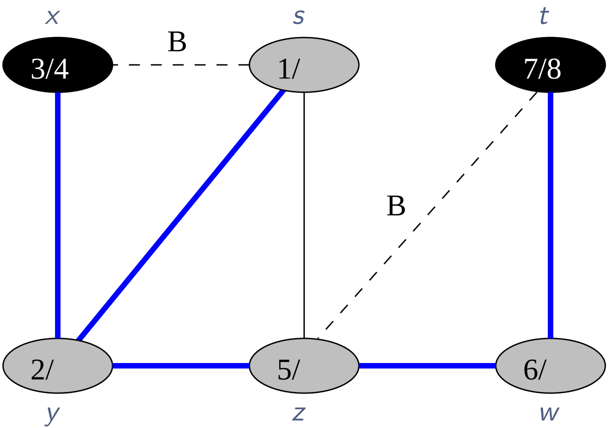
DFS em grafo não direcionado



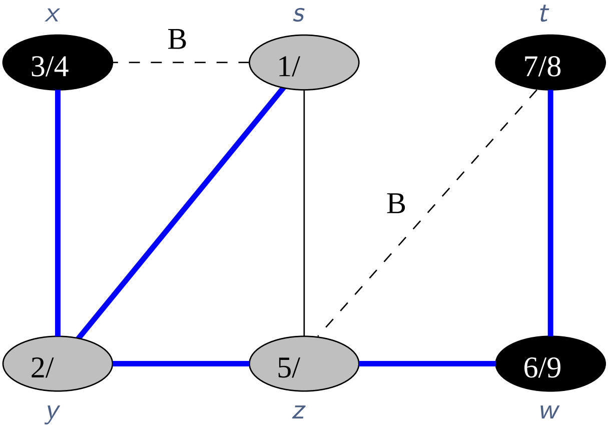
DFS em grafo não direcionado



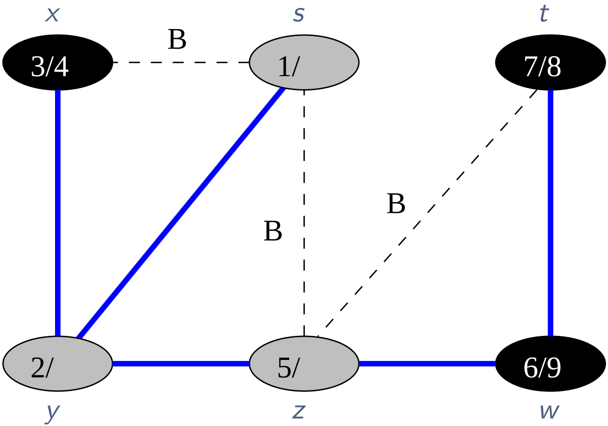
DFS em grafo não direcionado



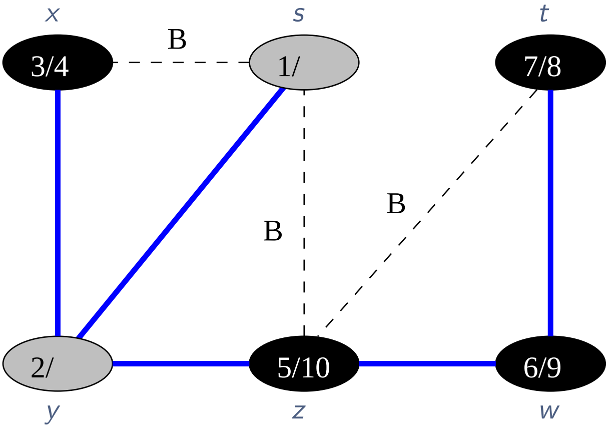
DFS em grafo não direcionado



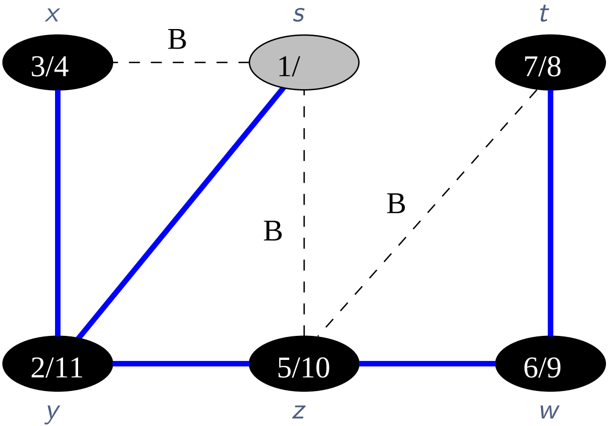
DFS em grafo não direcionado



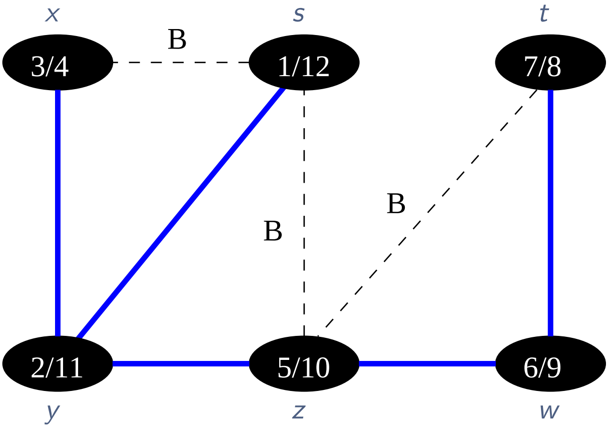
DFS em grafo não direcionado



DFS em grafo não direcionado

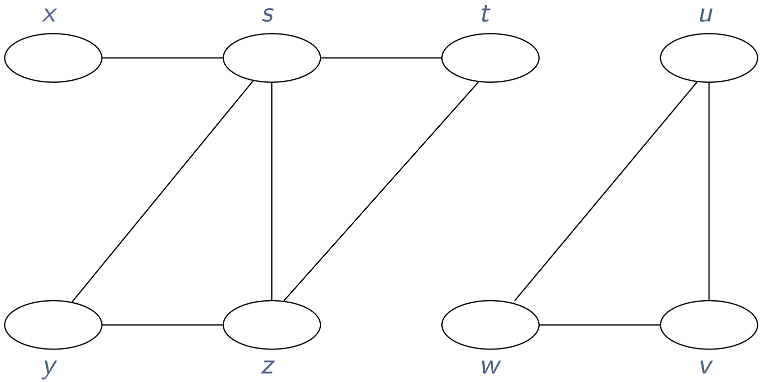


DFS em grafo não direcionado



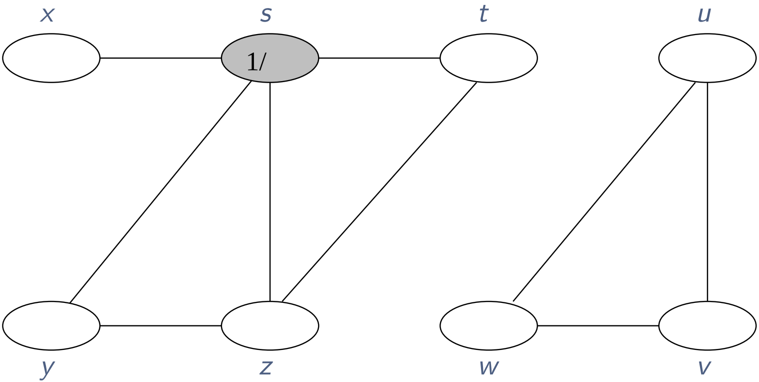
Componentes conexas

Componentes conexas

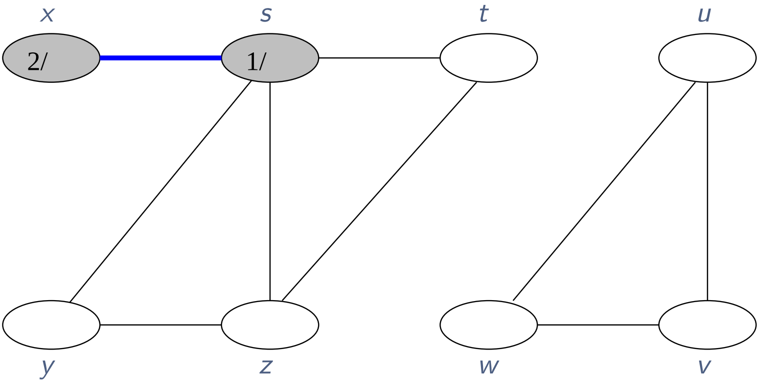


Problema: determinar as componentes conexas de um grafo não direcionado.

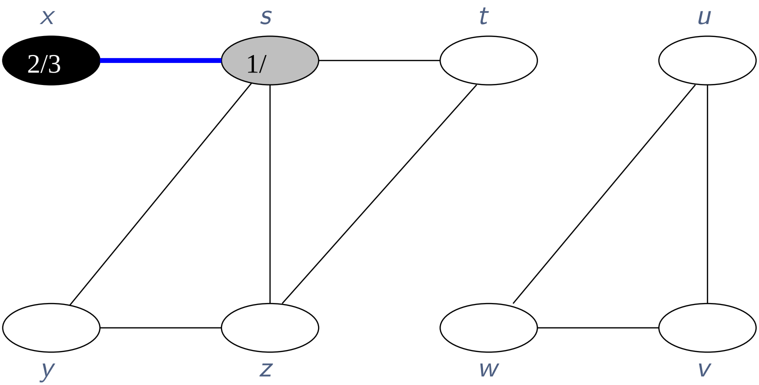
Executando DFS



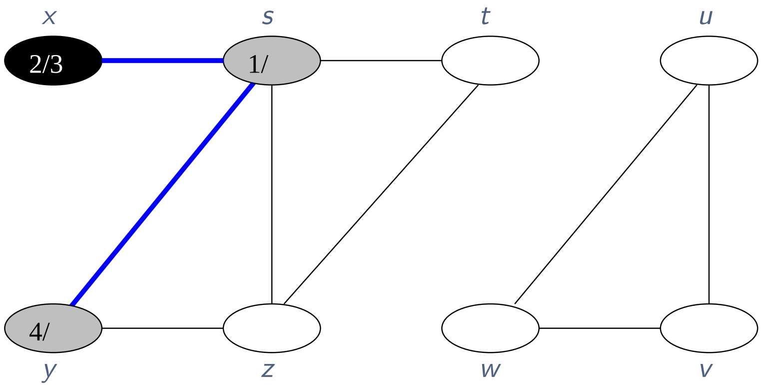
Executando DFS



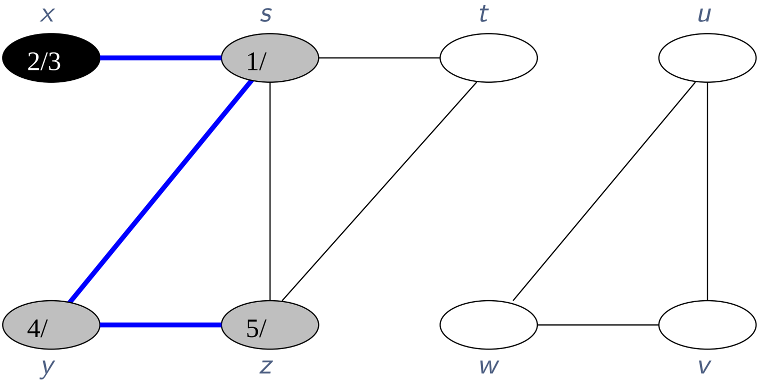
Executando DFS



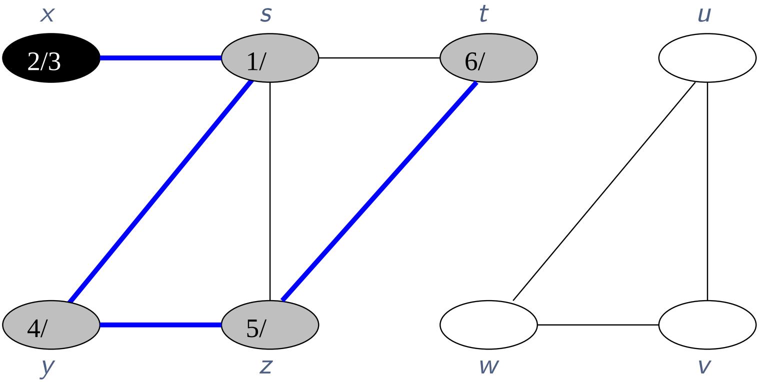
Executando DFS



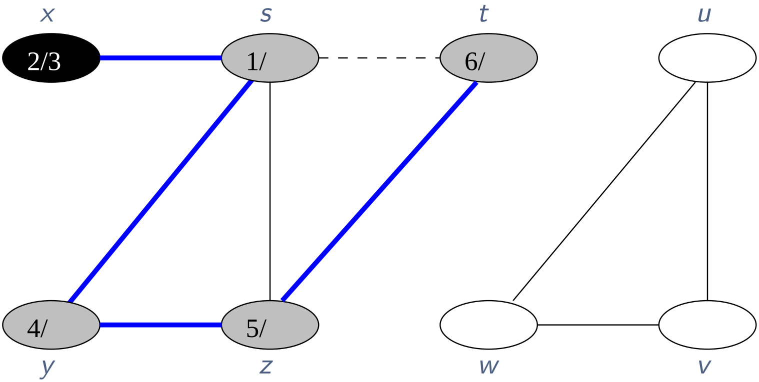
Executando DFS



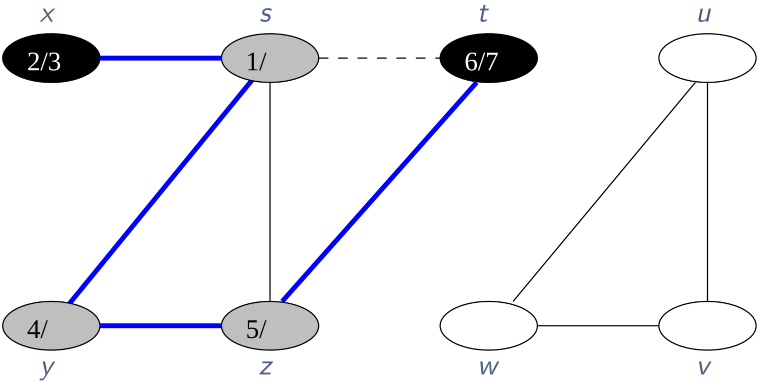
Executando DFS



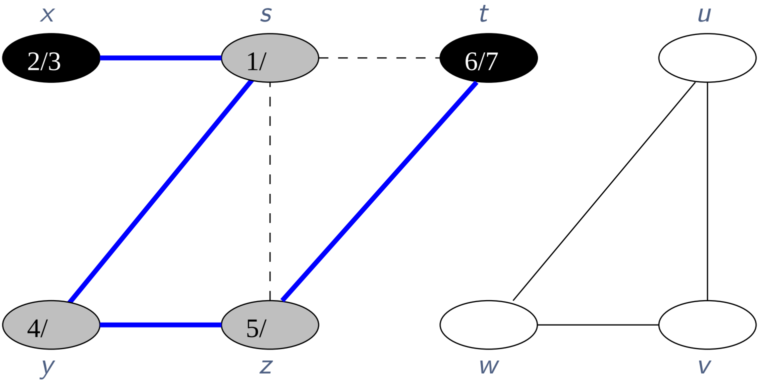
Executando DFS



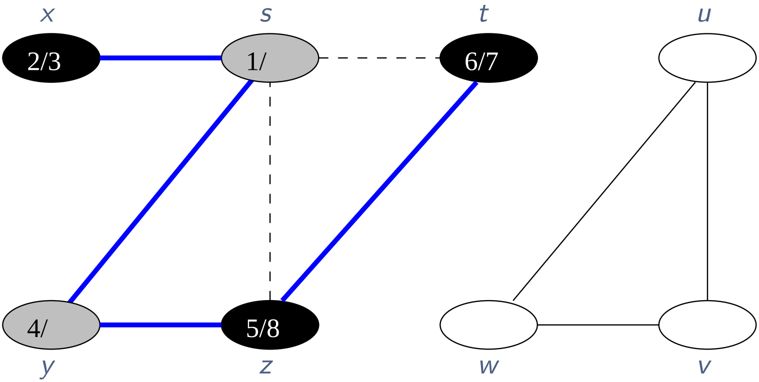
Executando DFS



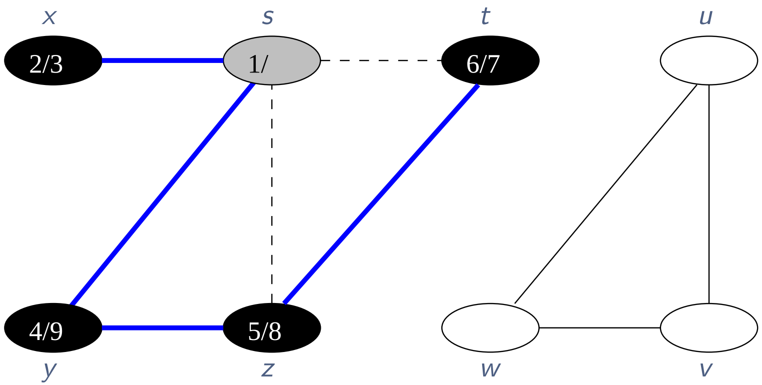
Executando DFS



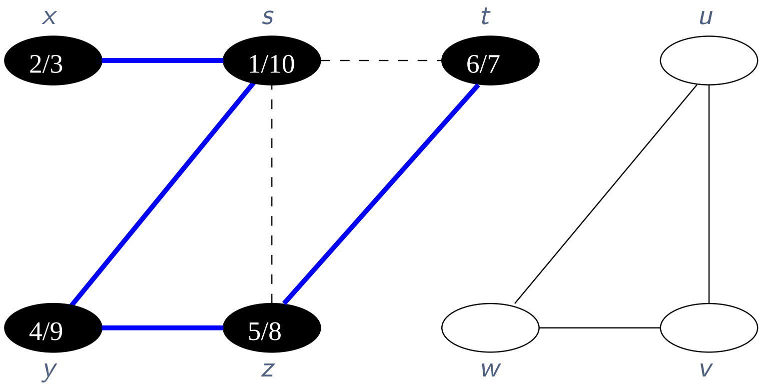
Executando DFS



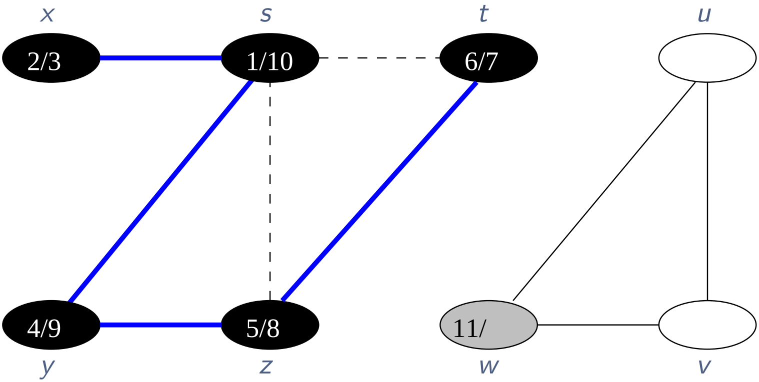
Executando DFS



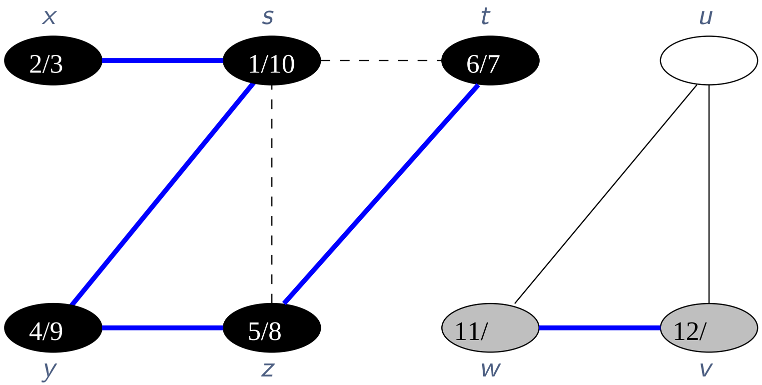
Executando DFS



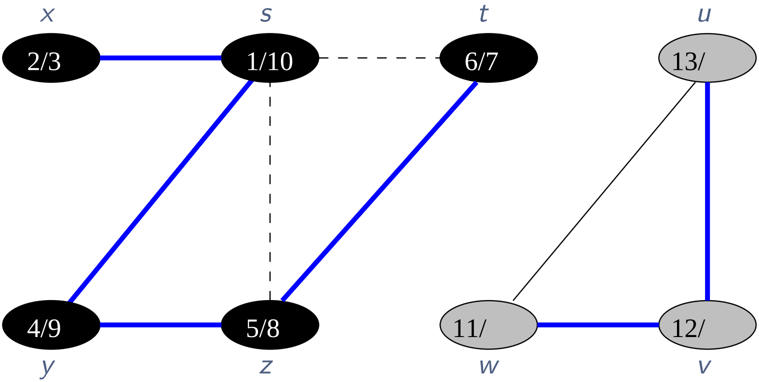
Executando DFS



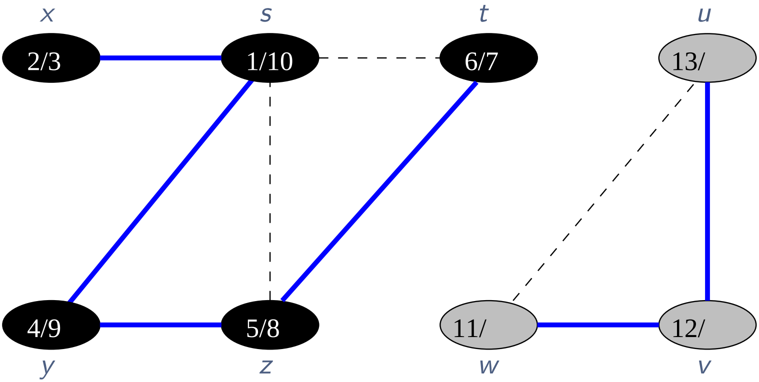
Executando DFS



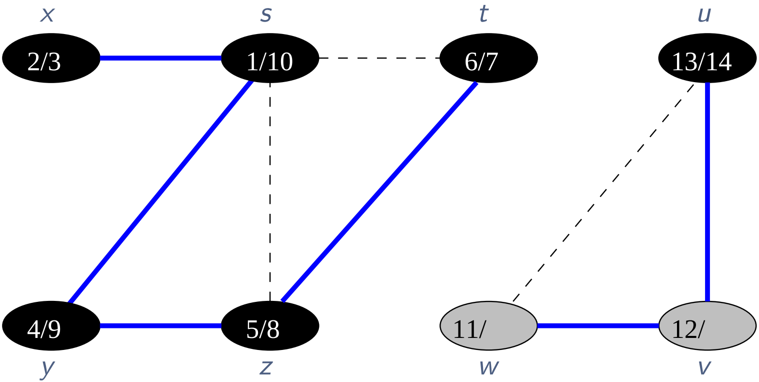
Executando DFS



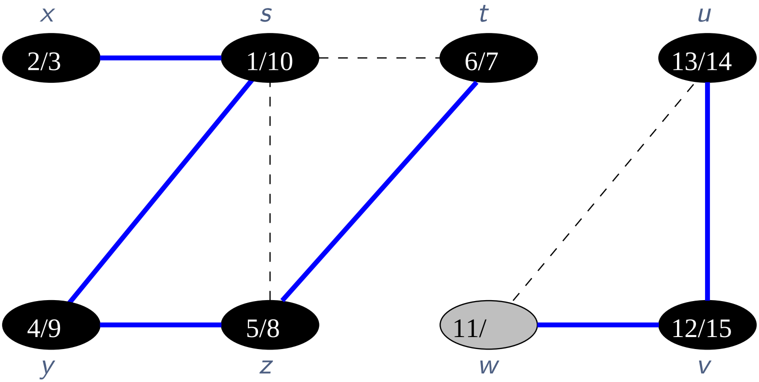
Executando DFS



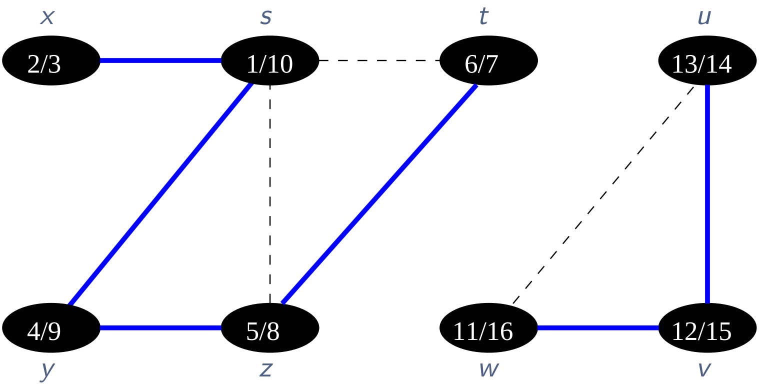
Executando DFS



Executando DFS



Executando DFS



Componentes conexas

Contando o número de componentes

- cada componente corresponde a uma árvore de busca
- é o número de chamadas a DFS-VISIT a partir de DFS

Vamos modificar DFS

- identificamos cada componente por um número
- denotaremos por $comp[v]$ a componente de v

Algoritmo DFS modificado

```
DFS(G)
    para cada u ∈ V[G] faça
        cor[u] = branco
    ℓ = 0
    para cada u ∈ V[G] faça
        se cor[u] = branco então
            ℓ = ℓ + 1
            DFS-VISIT(u)
```

- ℓ é o número de chamadas a DFS-VISIT a partir de DFS e é uma variável global

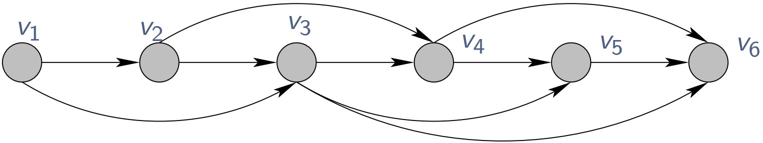
Algoritmo DFS-VISIT modificado

```
DFS-visit(u)
    cor[u] = cinza
    para cada v ∈ Adj[u] faça
        se cor[v] = branco então
            DFS-VISIT(v)
    cor[u] = preto
    comp[u] = ℓ
```

Ordenação Topológica

Ordenação Topológica

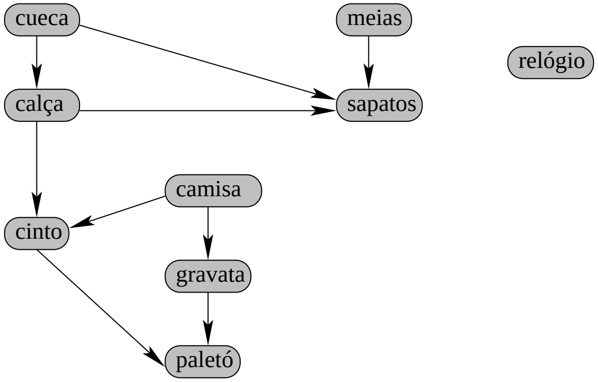
Uma ordenação topológica de um grafo direcionado é uma sequência linear $v_1, v_2, \ldots, v_n$ dos vértices de G tal que se (v_i, v_j) é uma aresta de G, então v_i aparace antes de v_j na sequência.



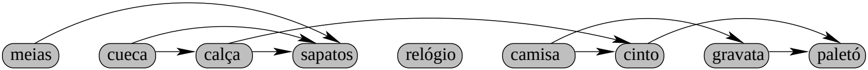
Exemplo de aplicação

Representando dependências

- um grafo pode representar precedências entre tarefas
- queremos um ordem que respeita as precedências



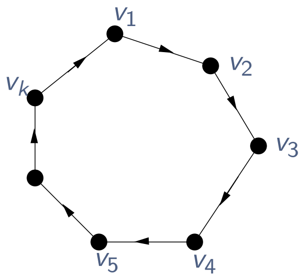
Exemplo de ordenação topológica



Condições de existência

Todo grafo direcionado possui ordenação topológica?

- não, um ciclo direcionado não possui
- nem outro grafo que contém um ciclo



Um grafo direcionado é acíclico se não contiver um ciclo direcionado.

Condições de existência

Teorema

Um grafo direcionado é acíclico se e somente se possui uma ordenação topológica.

Demonstração

- se G tem uma ordenação topológica, então ele é acíclico
- em seguida, vamos mostrar a recíproca

Um lema auxiliar

- Uma fonte é um vértice com grau de entrada zero.
- Um sorvedouro é um vértice com grau de saída zero.

Lema

Todo grafo direcionado acíclico G com pelo menos um vértice possui uma fonte e um sorvedouro.

Demonstração

- tome um caminho mais longo no grafo P que vai de s até t
- observe que s é uma fonte e t é um sorvedouro

Prova do teorema

Agora podemos terminar a demonstração. Queremos provar que se G é direcionado e acíclico, então G possui ordenação topológica.

- considere um grafo direcionado acíclico $G = (V, E)$
- afirmamos que G possui uma ordenação topológica
- vamos mostrar por indução em $|V|$
- se $|V| = 1$, então a afirmação é clara

Considere um grafo com pelo menos dois vértices

- pelo lema anterior, G possui uma fonte v_1
- pela hipótese de indução, o grafo $G - v_1$ possui uma ordenação topológica $v_2, \dots, v_n$
- logo $v_1, v_2, \dots, v_n$ é uma ordenação topológica de G

Encontrando uma ordenação topológica

A demonstração anterior é construtiva

- é baseada em exibir uma ordenação topológica
- ela sugere um algoritmo recursivo

Algoritmo para ordenação topológica

1. encontre uma fonte v_1 de G
2. recursivamente, obtenha ordenação $v_2, \ldots, v_n$ de $G - v_1$
3. devolva $v_1, v_2, \ldots, v_n$

- pode-se implementar esse algoritmo em tempo $O(V + E)$ (exercício)

Algoritmo baseado em DFS

Considere um grafo direcionado acíclico

- como não há ciclo, não existe aresta de retorno
- considere o instante em que v fica preto
- nesse instante todos seus vizinhos são pretos
- isso sugere considerar os vértices na ordem de término

Ideia para o algoritmo

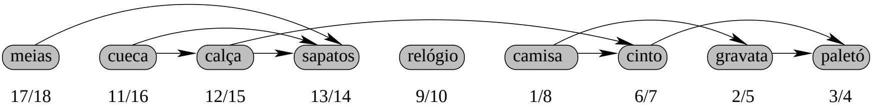
- o primeiro vértice a ficar preto não tem arestas saindo
- o segundo só pode ter arestas para o primeiro
- o terceiro só pode ter arestas para os dois primeiros
- etc.

Algoritmo TOPOLOGICAL-SORT

Topological-Sort(G)
1 execute $\mathrm{DFS}(G)$ e calcule $f[v]$ para cada vértice v
2 quando um vértice finalizar, insira-o no início de uma lista
3 devolva a lista resultante

- inserir cada um dos $|V|$ vértices leva tempo $O(1)$
- além disso, executamos DFS uma vez
- portanto, a complexidade de tempo é $O(V + E)$

Exemplo


meias
cueca
calça
sapatos
relógio
camisa
cinto
gravata
paletó
17/18
11/16
12/15
13/14
9/10
1/8
6/7
2/5
3/4

Correção

Teorema

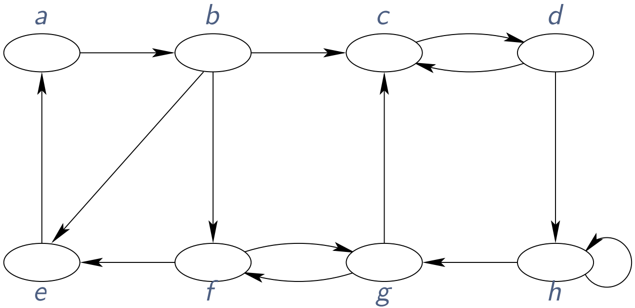
$\textsc{Topological-Sort}(G)$ devolve ordenação topológica de um grafo direcionado acíclico G.

Demonstração

- a lista devolvida está em ordem decrescente de $f[v]$
- considere uma aresta arbitrária (u, v)
- basta mostrar que $f[u] > f[v]$
- considere o instante em que (u, v) foi examinada
- como (u, v) não é aresta de retorno, v não pode ser cinza
  1. se v for branco, ele será descendente de u e $f[u] > f[v]$
  2. se v for preto, então ele já foi finalizado e $f[u] > f[v]$
- em qualquer caso, obtemos o que desejávamos. ∎

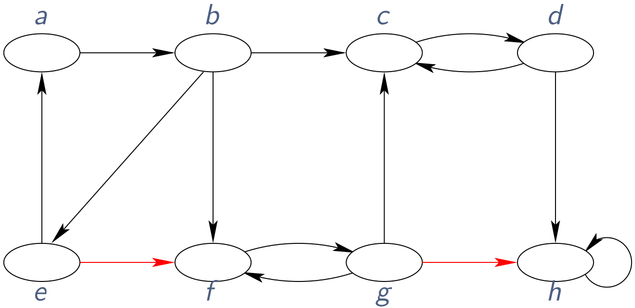
Componentes fortemente conexas

Grafo fortemente conexo



Um grafo direcionado $G = (V, E)$ é fortemente conexo se, para todo par de vértices u, v de G, existe um caminho direcionado de u a v e existe um caminho direcionado de v a u em G.

Grafo fortemente conexo



Nem todo grafo direcionado é fortemente conexo

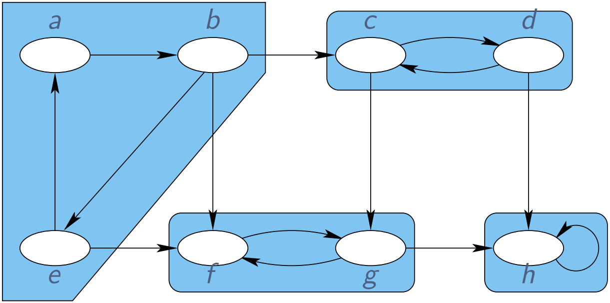
Componente fortemente conexa

Uma componente fortemente conexa de um grafo direcionado $G = (V, E)$ é um subconjunto de vértices $C \subseteq V$ tal que

(1) o subgrafo induzido por C é fortemente conexo e

(2) C é maximal com respeito à propriedade (1).

Componente fortemente conexa

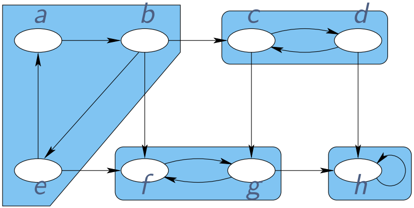


- podemos particionar um grafo direcionado em componentes fortemente conexas
- como encontrar as componentes fortemente conexas?

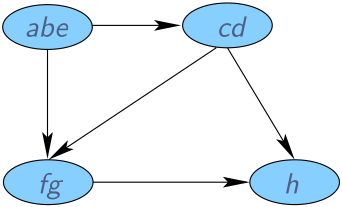
Grafo componente

O grafo componente de um grafo direcionado $G = (V, E)$ é um grafo direcionado em que

- cada vértice é uma componente fortemente conexa
- existe aresta (C, D) se houver $(u, v) \in E$ com $u \in C$ e $v \in D$



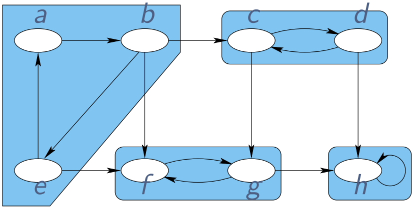
G



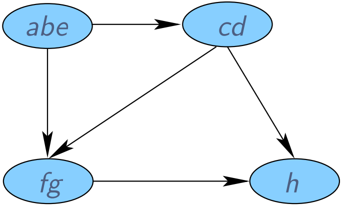
G^{CFC}

- denotamos o grafo componente por G^{CFC}
- note que G^{CFC} é acíclico (por quê?)

Grafo componente



G



G^{CFC}

Considere uma busca em profundidade sobre G

- seja u o último vértice finalizado
- então u deve pertencer a uma fonte de G^{CFC} (por quê?)
- o algoritmo que veremos adiante, visita as componentes fortemente conexas de G^{CFC} em ordem topológica
  - para isso, ele usa o grafo transposto

Grafo transposto

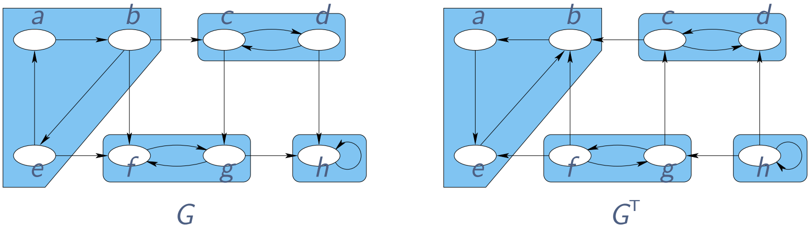
O grafo transposto de um grafo direcionado $G = (V, E)$ é um grafo direcionado que

- tem o mesmo conjunto de vértices
- tem uma aresta (u, v) se houver aresta (v, u) em G

Observações

- denotamos o grafo transposto por G^T
- ele é obtido invertendo-se as arestas de G
- podemos calcular G^T em tempo $O(V + E)$

Grafo transposto



G^{T}

Como encontrar uma componente fortemente conexa?

- note que G e G^{T} têm as mesmas componentes
- mas componentes fontes para G são sorvedouros para G^{T}
- suponha que temos um vértice u de uma fonte em G^{CFC}
- no transposto de G^{CFC}, os vértices alcançáveis de u formam uma componente!

Algoritmo

Componentes-Fortemente-Conexas(G)

1 execute $\mathrm{DFS}(G)$ e calcule $f[v]$ para cada $v \in V$

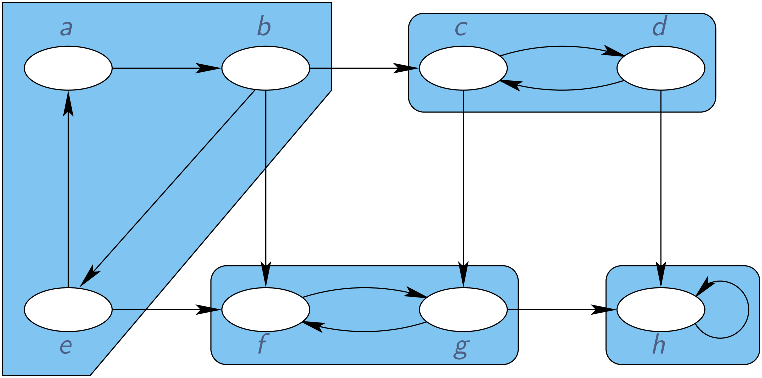
2 compute G^{T}

3 execute $\mathrm{DFS}(G^{\mathsf{T}})$, mas no laço principal de DFS, considere os vértices em ordem decrescente de $f[v]$

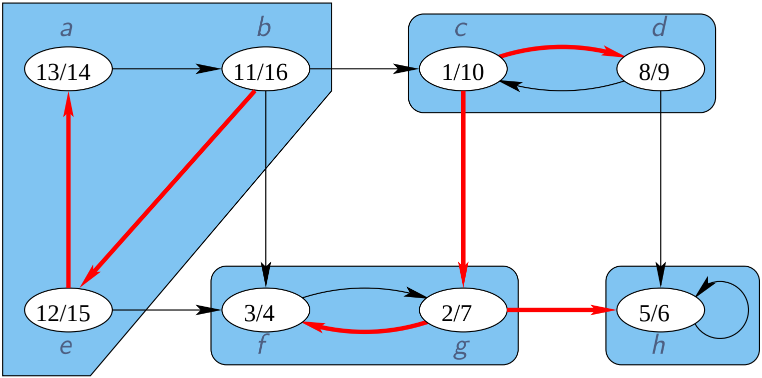
4 devolva os conjuntos de vértices de cada árvore da floresta de busca encontrada

- a complexidade de tempo é $O(V + E)$

Exemplo

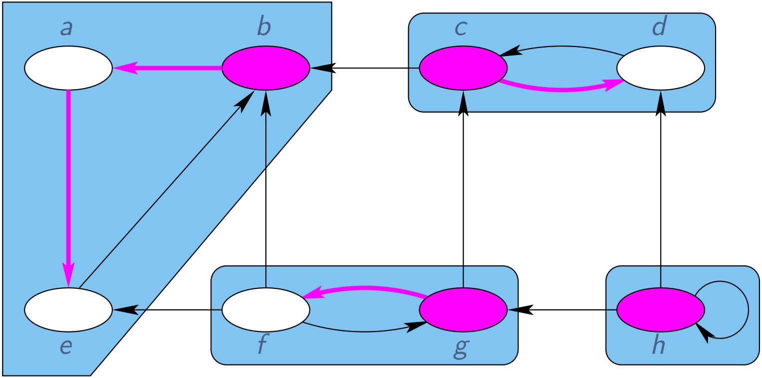


Exemplo



1 execute $\mathrm{DFS}(G)$ e calcule $f[v]$ para cada $v \in V$

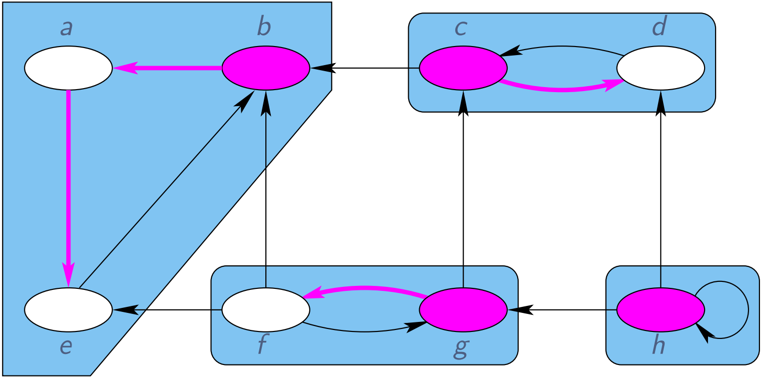
Exemplo



2 execute $\mathrm{DFS}(G^{\mathsf{T}})$ considerando os vértices em ordem decrescente de $f[v]$

3 devolva os conjuntos de vértices de cada árvore da floresta de busca encontrada

Exemplo



2 execute $\mathrm{DFS}(G^{\mathsf{T}})$ considerando os vértices em ordem decrescente de $f[v]$

3 devolva os conjuntos de vértices de cada árvore da floresta de busca encontrada

Correção

Componentes-Fortemente-Conexas(G)

1 execute $\mathrm{DFS}(G)$ e calcule $f[v]$ para cada $v \in V$

2 compute G^{T}

3 execute $\mathrm{DFS}(G^{\mathsf{T}})$, mas no laço principal de DFS, considere os vértices em ordem decrescente de $f[v]$

4 devolva os conjuntos de vértices de cada árvore da floresta de busca encontrada

Teorema

O algoritmo Componentes-Fortemente-Conexas determina as componentes fortemente conexas de G em tempo $O(V + E)$.

- antes da demonstração, precisamos de uma preparação

Lema auxiliar

Lema 1

Sejam C e D duas componentes fortemente conexas e considere vértices $u, v \in C$ e $u', v' \in D$.

- Se existe algum caminho $u \rightsquigarrow u'$,
- então não existe um caminho $v' \rightsquigarrow v$.

- segue da maximalidade de C e D
- o lema significa que G^{CFC} é acíclico $\blacksquare$

Definições auxiliares

Vamos adotar alguma convenção

- vamos considerar uma execução do algoritmo
- d e f referem-se à busca em profundidade da linha 1

Para cada subconjunto U de vértices, defina

$$d(U) = \min_{u \in U} \{d[u]\} \quad \text{e} \quad f(U) = \max_{u \in U} \{f[u]\}$$

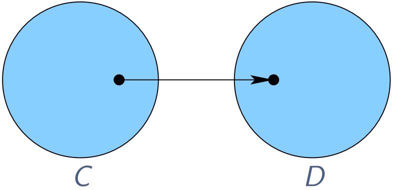
Em outras palavras

- $d(U)$ é o primeiro instante em que um vértice de U é descoberto
- $f(U)$ é o último instante em que um vértice de U é finalizado

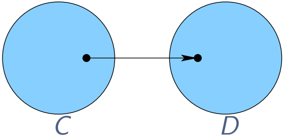
Outro lema auxiliar

Lema 2

Sejam C e D duas componentes fortemente conexas. Se existe aresta (u, v) tal que $u \in C$ e $v \in D$, então $f(C) > f(D)$.



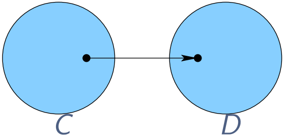
Prova do lema



Demonstração

- primeiro suponha que $d(C) < d(D)$
- isso é, suponha que descobrimos C antes de D
- seja x um vértice de C tal que $d[x] = d(C)$
- assim, x é o primeiro vértice de C a ser descoberto
- no instante $d[x]$, existia um caminho branco de x a cada um dos vértices em $C \cup D$
- então todos os vértices de $C \cup D$ são descendentes de x
- e portanto $f(D) < f[x] \leq f(C)$

Prova do lema (cont)



Continuando

- agora suponha que $d(C) > d(D)$
- assim, o primeiro vértice a ser descoberto está em D
- logo, cada um dos vértices de D é finalizado antes de qualquer vértice de C ser descoberto
- portanto $f(C) > f(D)$. ■

Corolário

Corolário

Seja G um grafo direcionado e X e Y duas componentes fortemente conexas de G. Se G^{T} tem aresta (u, v) tal que $u \in X$ e $v \in Y$, então $f(X) < f(Y)$.

- segue do fato de que G e G^{T} têm as mesmas componentes

Prova do teorema

Teorema

O algoritmo COMPONENTES-FORTEMENTE-CONEXAS determina as componentes fortemente conexas de G em tempo $O(V+E)$.

Demonstração

- vamos provar que as k primeiras árvores produzidas na linha 3 correspondem a componentes fortemente conexas
- a prova é por indução em k
- quando $k = 0$, a afirmação é trivial, então tome $k \geq 1$
- suponha que as primeiras $k-1$ primeiras árvores produzidas correspondem a componentes

Prova do teorema (cont)

Considere a k-ésima árvore produzida pelo algoritmo

- seja u a raiz dessa árvore de busca
- e seja C a componente fortemente conexa que contém u
- vamos mostrar que a árvore produzida contém todos os vértices de C e somente os vértices de C
- isso completará a indução e a prova do teorema

Prova do teorema (cont)

A árvore contém todos vértices de C

- considere o instante em que u é descoberto
- por indução nenhum vértice de C foi finalizado
- então nesse instante $d[u]$ os vértices de C são brancos
- assim, todos os vértices de C tornam-se descendentes de u na árvore de busca de G^{T}

A árvore contém somente vértices de C

- suponha que existe aresta (u, v) que sai de C
- seja D a componente fortemente conexa que contém v
- pelo corolário do Lema 2, temos $f(C) < f(D)$
- então descobrimos vértices de D antes de u na segunda DFS
- por indução, todos vértices de D já foram finalizados
- portanto, a árvore só contém vértices de C