DCC206 – Algoritmos 1

Aula 03 – Aplicações de caminhamento em grafos – Parte 1

Professor Renato Vimieiro

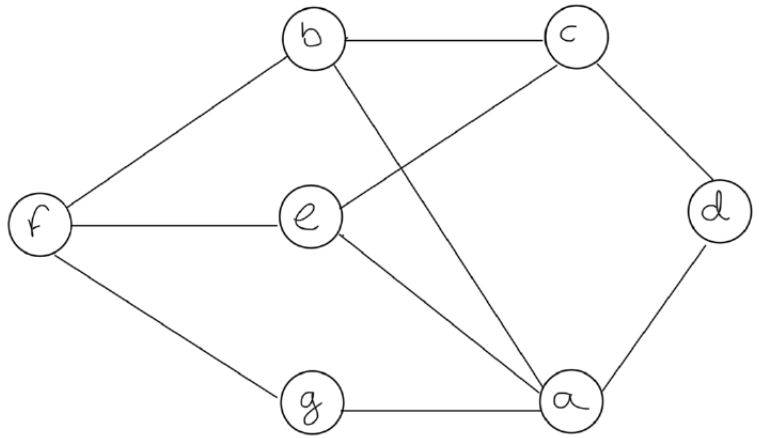
DCC/ICEx/UFMG

Introdução

- Tendo visto os algoritmos de caminhamento e algumas aplicações mais simples, nessa aula vamos explorar outras aplicações dos algoritmos
- Veremos como os algoritmos de caminhamento podem ser usados para detectar propriedades bastante úteis em grafos e auxiliar em outras tarefas de computação
- Veremos também como as buscas se comportam com grafos direcionados e como elas são usadas para verificar propriedades desses grafos

Verificar se um grafo é bipartido

- Como visto anteriormente, um grafo é bipartido se seu conjunto de vértices pode ser particionado em dois conjuntos disjuntos de forma que as arestas possuam terminais em duas partições distintas
- Quando as partições já são claramente identificadas no desenho do grafo, por exemplo, é fácil validar que se trata de um grafo bipartido
- Mas e quando eles não são apresentadas dessa forma?



Verificar se um grafo é bipartido

- Intuitivamente, podemos colorir as partições com duas cores (azul e vermelho) para identificá-las
- Se verificarmos que não existem arestas ligando vértices com a mesma cor, então o grafo é bipartido. Caso contrário, não é.
- O problema é que existem muitas possibilidades de coloração dos vértices para obter as partições.
- Talvez, aquela coloração testada não funcione, mas não necessariamente o grafo não é bipartido
- Precisamos encontrar uma propriedade que possibilite desenhar um algoritmo mais eficiente
- Vamos raciocinar de forma inversa: qual o grafo mais simples que não é bipartido?

Verificar se um grafo é bipartido

- **Lema 1**: Se um grafo é bipartido, então ele não contém um ciclo de tamanho ímpar
- **Prova**: A prova será pela contrapositiva. Seja C um ciclo de tamanho ímpar no grafo. Podemos arbitrariamente numerar os vértices de 1 a 2k+1. Se pensarmos na estratégia de colorir as partições, podemos atribuir a cor vermelho ao vértice 1 sem perda de generalidade. Percorremos o ciclo alternando as cores entre os vértices ímpares e pares. Quando chegarmos no vértice 2k+1, temos um problema, já que ele é conectado ao vértice 1, que também possui a cor vermelho. Dessa forma, não é possível particionar o conjunto de vértices em dois sem que haja arestas entre vértices de uma mesma cor. Portanto, o grafo não é bipartido.

Verificar se um grafo é bipartido

- Combinando o raciocínio da coloração com o lema, podemos especificar um algoritmo para verificar se um grafo é bipartido
- Podemos executar uma busca em largura no grafo, atribuindo cores alternadas a cada nível da árvore
  - Iniciamos colorindo a raiz com vermelho
  - Depois atribuímos azul ao nível 1
  - E assim sucessivamente
  - Ao finalizar, verificamos exaustivamente se alguma aresta possui terminais de uma mesma cor
- O custo desse algoritmo continua o mesmo da BFS, $O(|V|+|E|)$.
- Resta provar que ele é correto

Verificar se um grafo é bipartido

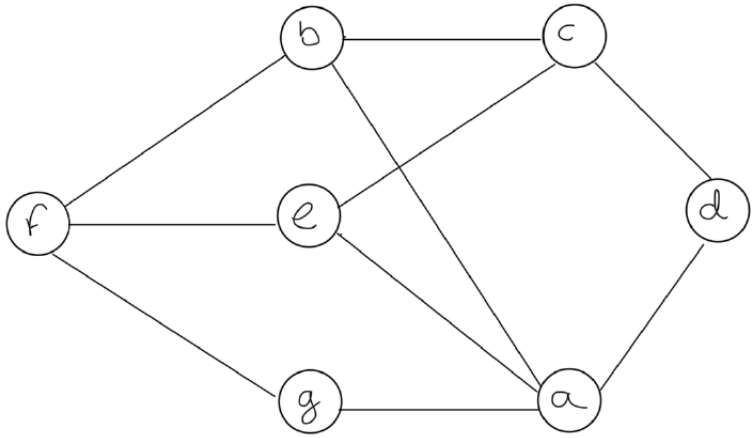
- **Lema 2**: Seja T a árvore de busca da BFS. Sejam x e y nós pertencentes aos níveis i e j, respectivamente. Se (x,y) é uma aresta de G, $i-j \leq 1$.
- **Prova** (intuição): Os vértices adjacentes a x são avaliados, no máximo, no nível seguinte a x. Ou seja, caso y não tenha sido avaliado antes, ele será avaliado logo após x ser explorado.

Verificar se um grafo é bipartido

- **Teorema 1**: Seja G um grafo conectado e s a raiz da BFS em G. Os dois itens a seguir são **mutuamente exclusivos**
  1. Não existe aresta conectando vértices de um mesmo nível. Nesse caso, os vértices de cada nível podem ser pintados com cores alternadas (vermelho ímpar e azul par). Ou seja, o **grafo é bipartido**.
  2. Existe aresta conectando vértices de um mesmo nível. Nesse caso, existe um ciclo de tamanho ímpar e o **grafo não é bipartido**.
- **Prova**:
  1. Suponha que não existam arestas conectando vértices do mesmo nível. Pelo lema 1, as arestas conectam vértices de níveis adjacentes. Assim, podemos colorir cada nível com uma cor sem que dois adjacentes tenham a mesma cor.
  2. Sejam x e y vértices adjacentes em um mesmo nível. Suponha que esse nível seja j. Seja z o ancestral em comum mais próximo de ambos (cujo nível seja o mais próximo de j). Suponha que ele esteja no nível i. Como x e y estão a uma distância (j-i) de z, podemos construir um circuito z~x-y~z cujo tamanho é (j-i)+1+(j-i)=2(j-i)+1. Portanto, o grafo não é bipartido.
- **Corolário**: *um grafo é bipartido sse não contém um ciclo de tamanho ímpar.*

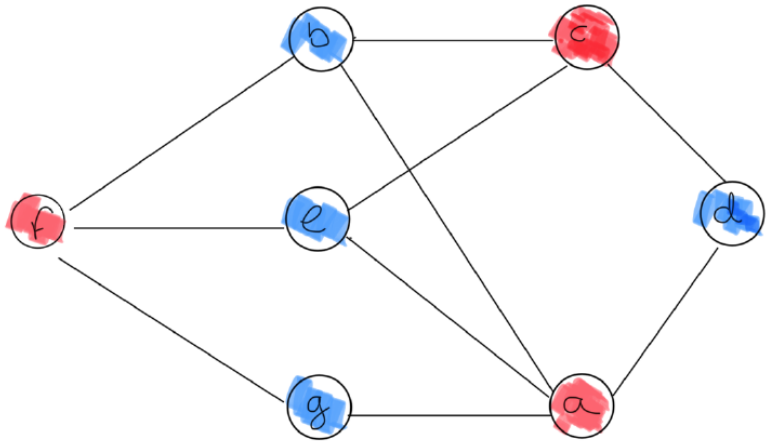
Verificar se um grafo é bipartido

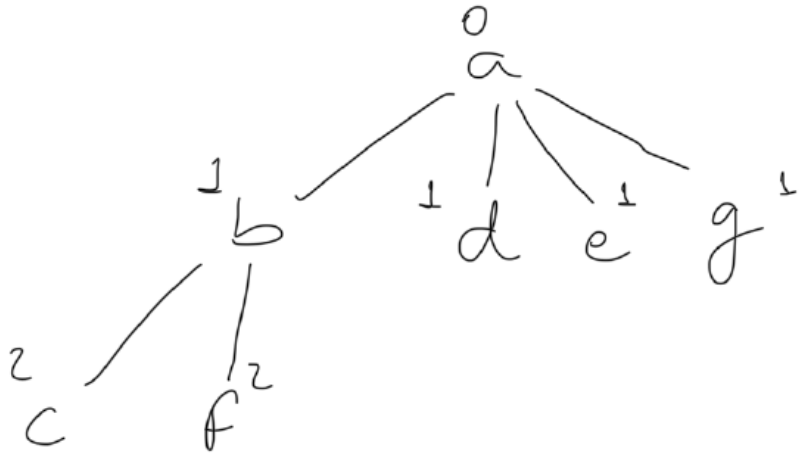
- Exemplo:



Verificar se um grafo é bipartido

- Exemplo:





Busca em grafos direcionados

- Os algoritmos de busca funcionam praticamente da mesma forma, exceto que as arestas agora possuem direção
- É importante lembrar que as buscas encontram os vértices alcançáveis a partir de uma origem.
- Pode haver caminho de s para t, mesmo que não exista de t para s
- A BFS continua encontrando o caminho mais curto (direcionado) de s para t
- Se quisermos encontrar os vértices que alcançam um vértice s, podemos computar o grafo reverso (mudando o sentido das arestas) e executar uma busca a partir de s nesse grafo
- Podemos usar caminhamentos para descobrir propriedades dos grafos

Classificação de arestas

- Como vimos no caso não direcionado, as arestas encontradas durante o caminhamento no grafo podem ser classificadas de diferentes formas
- Antes elas eram classificadas apenas em arestas de árvore e arestas de retorno
- Agora veremos que elas podem receber outros dois rótulos

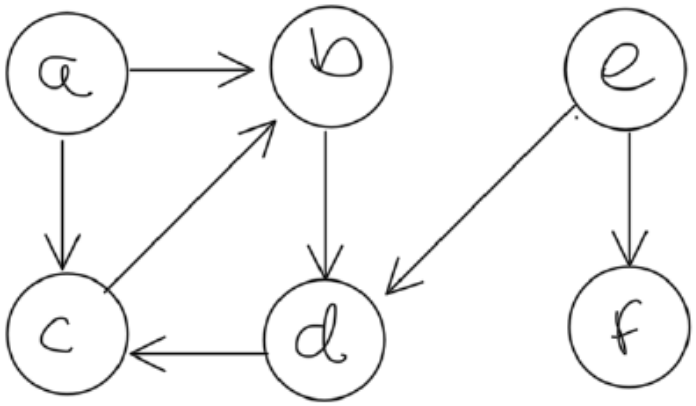
Classificação de arestas

- Antes de seguirmos, vamos fazer uma pequena alteração na DFS para registrar o momento em que um vértice é descoberto (tem sua exploração iniciada) e finalizado (todos os seus descendentes foram explorados e a busca retorna para seu pai)

```
dfs(g, v, antecessor, marcado)
    time += 1
    marcado[v] = true
    v.d = time
    para u ∈ g.adj(v)
        se !marcado[u]
            antecessor[u] = v
            dfs(g,u,antecessor,marcado)
    time += 1
    v.f = time
```

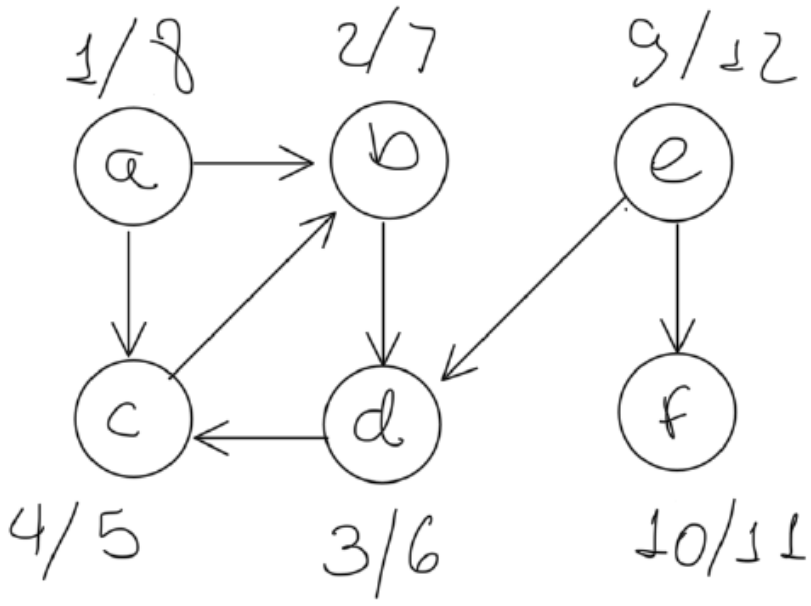
Classificação de arestas

- Exemplo



Classificação de arestas

- Exemplo

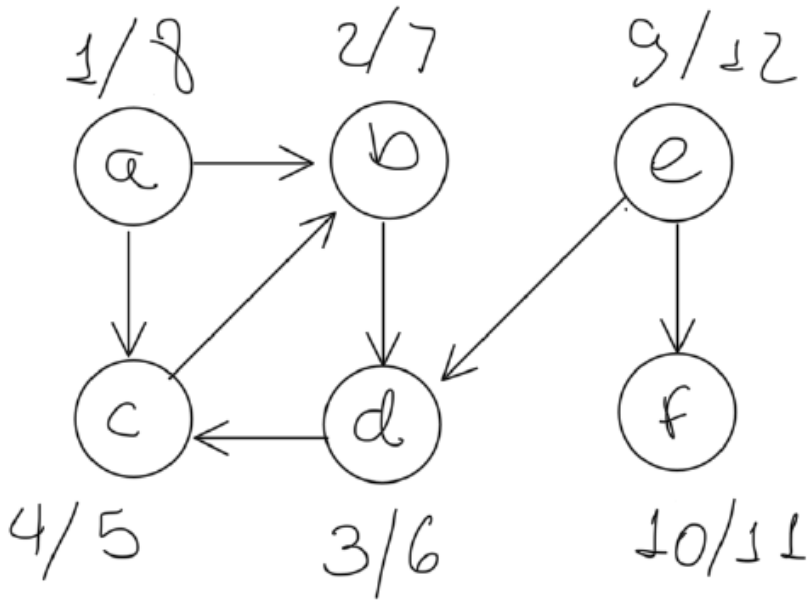


Classificação de arestas

- Voltando à classificação das arestas, consideramos quatro possibilidades
  1. As arestas conectando um vértice em exploração a um ainda não explorado continua sendo uma **aresta de árvore**
  2. Arestas ligando um vértice em exploração a um antecessor é uma **aresta de retorno**
  3. Arestas ligando um vértice em exploração a um finalizado (descendente) são chamadas de **arestas de avanço**
  4. Arestas ligando um vértice em exploração a um finalizado não descendente são chamadas de **arestas de cruzamento**

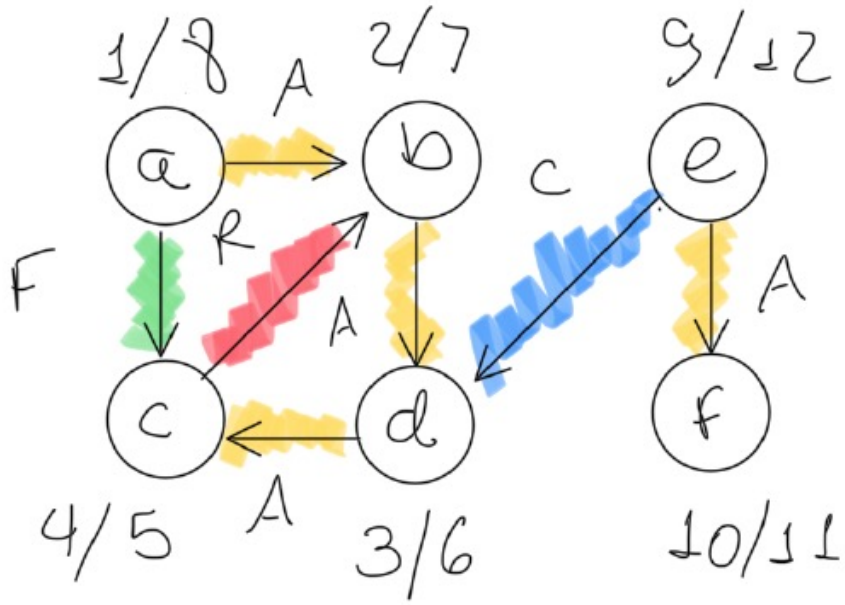
Classificação de arestas

- Exemplo



Classificação de arestas

- Exemplo:



Ordenação topológica

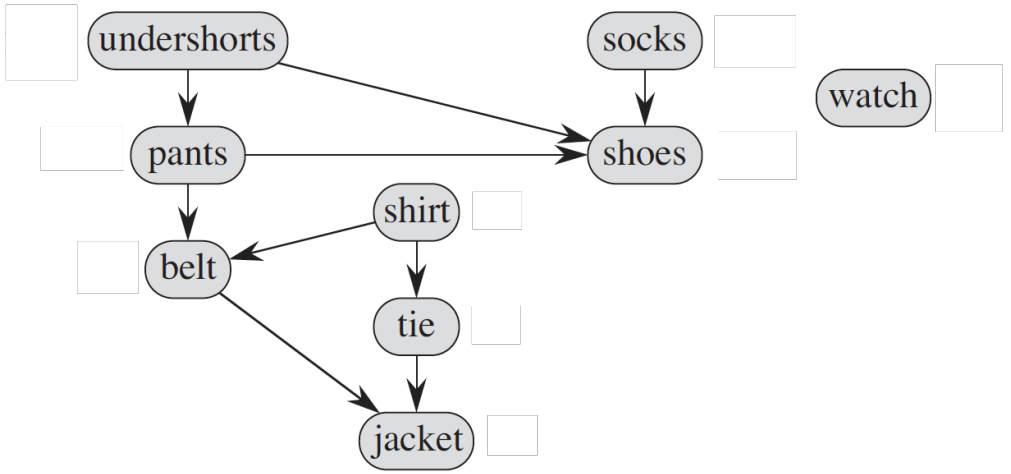
- Uma estrutura bastante útil em computação são os DAGs (directed acyclic graphs)
- Essas estruturas surgem frequentemente na modelagem de relações causais ou de dependência entre objetos
- Elas podem ser usadas, por exemplo, na modelagem de um pipeline de processamento de dados, indicando como certas tarefas dependem do resultado de outras para serem executadas
- Dessa forma, é natural que se busque uma ordenação dos vértices que respeite essa relação de dependência

Ordenação topológica

- A ordenação topológica de um DAG é uma ordenação dos vértices de forma que, para toda aresta (u,v), u < v.
- Ou seja, é uma ordenação horizontal dos vértices de forma que as arestas são sempre direcionadas da esquerda para direita
- Naturalmente, um grafo direcionado com ciclos não pode ser ordenado topologicamente, já que haveria ao menos uma aresta da direita para esquerda

Ordenação topológica

- Para ilustrar o conceito, considere a relação de precedência entre peças de vestuário mostrada no grafo ao lado
- Em que ordem as peças podem ser vestidas?
- Entre algumas, não importa a ordem exata. Por exemplo, não há uma ordem exata entre vestir a cueca ou a camisa. Porém, há uma ordem clara entre vestir a calça e a cueca
- Nosso objetivo é desenhar um algoritmo que retorne uma ordem viável

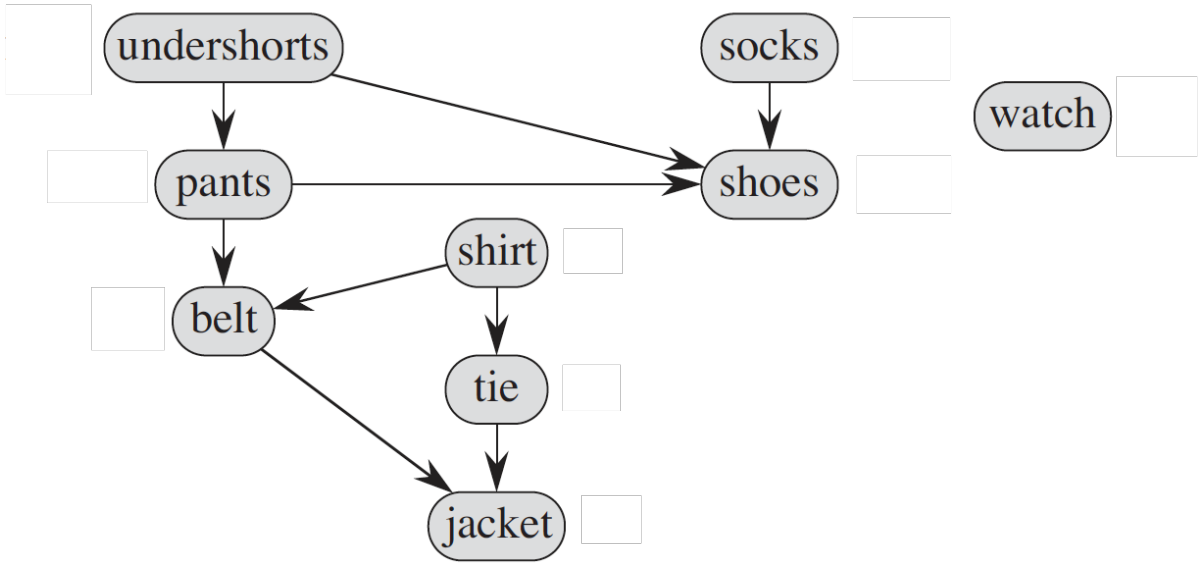


Ordenação topológica

- Intuitivamente, a solução é bem simples. Basta encontrarmos vértices sem arestas de saída e posicioná-los por último na nossa ordenação
- Assim, podemos executar uma DFS no grafo
- Sempre que terminarmos de explorar um vértice (não há mais arestas de saída), o inserimos no início de uma lista encadeada
- Terminada a busca, retornamos a lista com os vértices em ordem

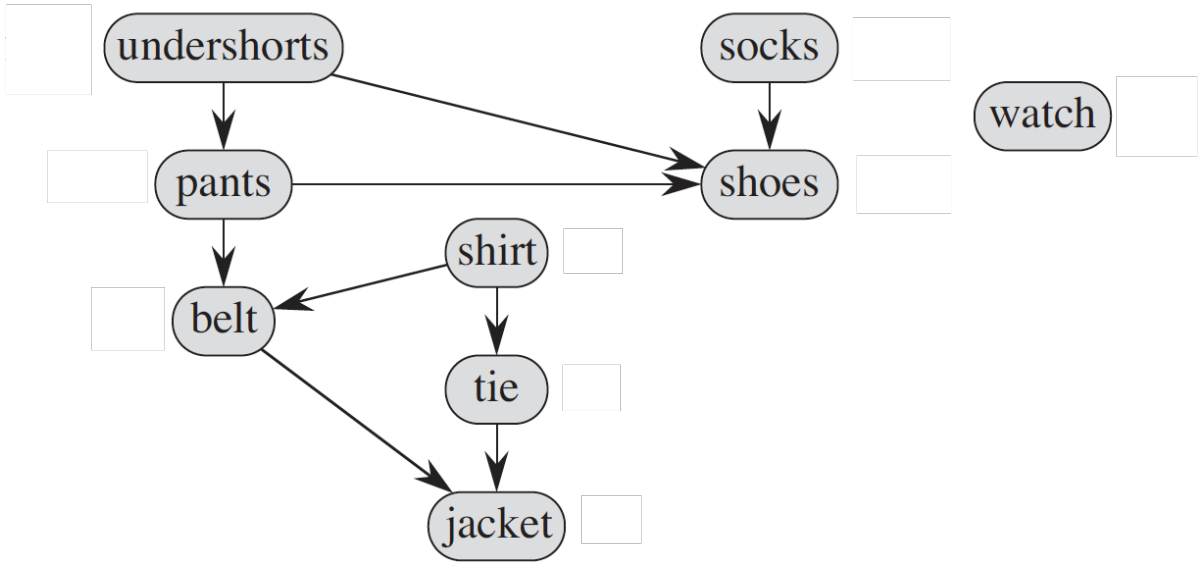
Ordenação topológica

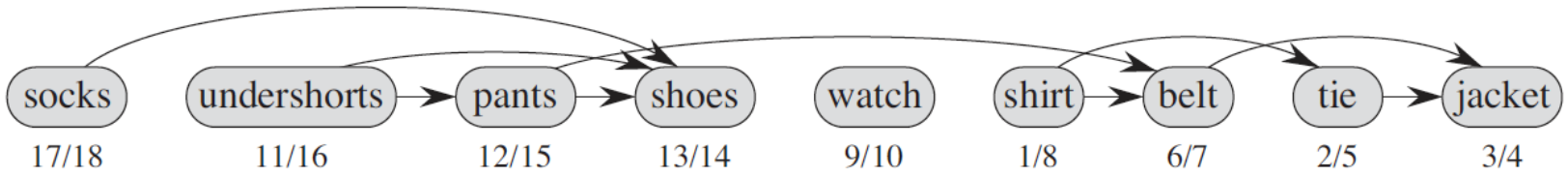
- Exemplo:



Ordenação topológica

- Exemplo:





Ordenação topológica

- O custo do algoritmo é idêntico ao da busca em profundidade
  - A única alteração é a inserção dos vértices na lista encadeada que tem custo O(1)
- Precisamos demonstrar a corretude do algoritmo

Ordenação topológica

- **Lema 3**: um grafo direcionado é acíclico se, e somente se, a DFS não encontra nenhuma aresta de retorno
- **Prova**:
- (⇒) Suponha que a busca encontre uma aresta de retorno. Seja (u,v) tal aresta. Logo, v é um antecessor de u, ou, u é um descendente de v. Assim, podemos construir um ciclo percorrendo o caminho de v para u e retornando pela aresta (u,v).
- (⇐) Suponha que G tenha um ciclo C. Seja $v \in C$ o primeiro vértice do ciclo a ser descoberto pela DFS. Seja (u,v) a aresta que fecha o ciclo. Como v é o primeiro a ser descoberto, os demais ainda não foram explorados. Logo, eles serão descendentes de v na busca, incluindo o vértice u. Portanto, a aresta (u,v) é uma aresta de retorno.

Ordenação topológica

- **Teorema**: Se G é um DAG, então o algoritmo encontra uma ordenação topológica do grafo.
- **Prova**: Suponha que uma DFS tenha sido executada no grafo e que os tempos de descoberta e finalização tenham sido registrados para cada vértice. É suficiente mostrar que, para toda aresta (u,v), v.f < u.f (o que mostra que v foi inserido na lista antes de u, e a aresta é da esquerda para direita na ordenação). Seja (u,v) uma aresta arbitrária. Quando ela for explorada pela busca, v não pode ainda estar sendo explorado, caso contrário, u seria seu descendente e (u,v) uma aresta de retorno, o que contraria a hipótese de que G é um DAG. Se v não tiver sido explorado, então v é descendente de u e, portanto, v.f < u.f. Se v tiver sido finalizado, obviamente v.f < u.f, já que ele ainda está sendo explorado. Como a aresta é arbitrária, para toda aresta (u,v), v foi inserido antes de u na lista.

Leitura

- Seções 3.4 (Kleinberg e Tardos)
- Seções 22.2 a 22.4 (CLRS)