AULA 05 – GRAFOS BIPARTIDOS E INTRODUÇÃO À BUSCA EM PROFUNDIDADE

Prof. Daniel Kikuti

Universidade Estadual de Maringá

1 de abril de 2015

Sumário

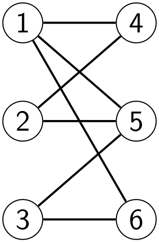
Grafo bipartido

Definição

Um grafo **bipartido** é um grafo não orientado $G = (V, E)$ em que V pode ser particionado em dois subconjuntos V_1 e V_2 tais que $(u, v) \in E$ implica que:

- $u \in V_1 \text{ e } v \in V_2$ ou
- $v \in V_1 \text{ e } u \in V_2$.

Exemplo



Grafo bipartido

Possíveis aplicações???

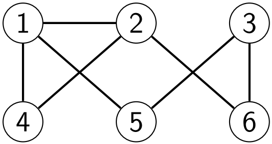
- Relacionamento entre homens e mulheres;
- Aptidão entre pessoas e tarefas;
- ...

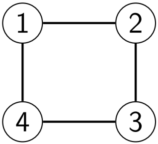
Problema de nosso interesse

Dado um grafo não-orientado $G = (V, E)$, determinar se ele é bipartido.

Grafo bipartido

Os grafos a seguir são bipartidos?





Bons vs. Maus

```
Bons-vs-Maus(G,s)
v.cor ← branco ∀ v ∈ V − {s}
s.cor ← azul
Insere(Q, s)
enquanto Q ≠ ∅ faça
    u ← Remove(Q)
    para cada v ∈ u.adj faça
        se v.cor = branco então
            se u.cor = azul então
                v.cor ← vermelho
            senão
                v.cor ← azul
            Insere(Q, v)
        senão se u.cor = v.cor então
            devolva IMPOSSÍVEL
devolva V;
```

Correção do algoritmo

O algoritmo efetua uma busca em largura, colorindo de azul todos os vértices nas camadas pares e de vermelho todos os vértices nas camadas ímpares.

Correção do algoritmo

O algoritmo efetua uma busca em largura, colorindo de azul todos os vértices nas camadas pares e de vermelho todos os vértices nas camadas ímpares.

Propriedade 1

Se um grafo é bipartido, então não pode conter um ciclo de comprimento ímpar.

Correção do algoritmo

O algoritmo efetua uma busca em largura, colorindo de azul todos os vértices nas camadas pares e de vermelho todos os vértices nas camadas ímpares.

Propriedade 1

Se um grafo é bipartido, então não pode conter um ciclo de comprimento ímpar.

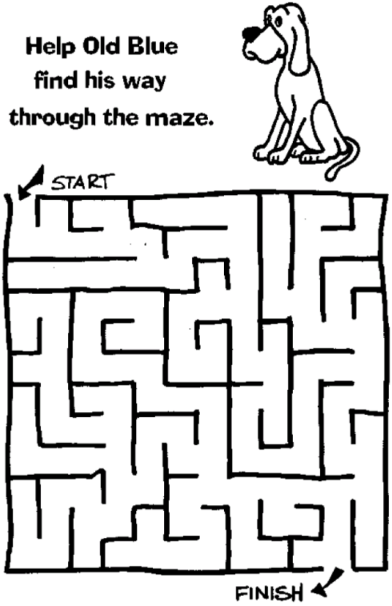
Propriedade 2

Dado um grafo não orientado G, duas coisas podem acontecer na execução do algoritmo:

(i) Não existe nenhuma aresta de G conectando dois vértices na mesma camada (G é um grafo bipartido).

(ii) Existe uma aresta de G conectando dois vértices na mesma camada (G possui um ciclo de comprimento ímpar, e portanto não é bipartido).

Introdução - Labirinto



Busca em grafos

Estratégias de busca

- Busca em largura (BFS - *Breadth First Search*)
- Busca em profundidade (DFS - *Depth First Search*)

Busca em grafos

Estratégias de busca

- Busca em largura (BFS - *Breadth First Search*)
- Busca em profundidade (DFS - *Depth First Search*)

Busca em profundidade

- Entrada: $G = (V, E)$ e um vértice s.

Busca em grafos

Estratégias de busca

- Busca em largura (BFS - *Breadth First Search*)
- Busca em profundidade (DFS - *Depth First Search*)

Busca em profundidade

- Entrada: $G = (V, E)$ e um vértice s.
- Percorre todos os vértices alcançáveis a partir de s, mas ao passo que um novo vértice é descoberto, utiliza-se este para descobrir outros novos (profundidade).

Busca em grafos

Estratégias de busca

- Busca em largura (BFS - *Breadth First Search*)
- Busca em profundidade (DFS - *Depth First Search*)

Busca em profundidade

- Entrada: $G = (V, E)$ e um vértice s.
- Percorre todos os vértices alcançáveis a partir de s, mas ao passo que um novo vértice é descoberto, utiliza-se este para descobrir outros novos (profundidade).
- Quando todas as arestas adjacentes a um vértice v tiverem sido exploradas a busca retrocede (*backtracking*) para explorar outras arestas que saem do vértice do qual v foi descoberto.

Busca em Profundidade

Visão geral

- Inicialmente a árvore de busca contém apenas o vértice s.
- Ao encontrar um vértice não explorado v adjacente a s, o vértice v e a aresta (s, v) são acrescentados a árvore.
- Processo é repetido para v e seus descendentes, retrocedendo quando necessário.
- Pode ser implementado usando uma pilha `Q` (LIFO), ou um algoritmo recursivo.

Busca em Profundidade

Cores dos vértices indicam:

- Branco = não descoberto
- Cinza = descoberto
- Preto = lista de adjacências completamente examinada

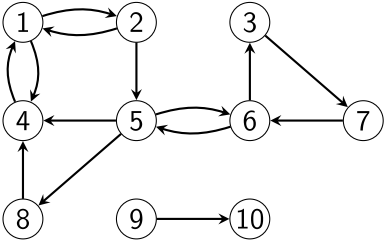
Busca em Profundidade

Cores dos vértices indicam:

- Branco = não descoberto
- Cinza = descoberto
- Preto = lista de adjacências completamente examinada

Exemplo

- Considere o grafo orientado a seguir:



O algoritmo

```
dfs(G)
para cada vértice u em G.V
  u.cor = branco
  u.pred = nil
tempo = 0
para cada vértice u em G.V
  se u.cor == branco
    dfs-visit(u)

dfs-visit(u)
tempo = tempo + 1
u.cor = cinza
u.d = tempo
para cada vértice v em u.adj
  se v.cor == branco
     v.pred = u
     dfs-visit(v)
u.cor = preto
tempo = tempo + 1
u.f = tempo
```

Análise do algoritmo

Consumo de tempo (análise agregada)

Análise do algoritmo

Consumo de tempo (análise agregada)

- Os loops nas linhas 1 a 3 e nas linhas 5 a 7 de `dfs` consomem tempo $\Theta(V)$, sem contar o tempo das chamadas a `dfs-visit`.

Análise do algoritmo

Consumo de tempo (análise agregada)

- Os loops nas linhas 1 a 3 e nas linhas 5 a 7 de `dfs` consomem tempo $\Theta(V)$, sem contar o tempo das chamadas a `dfs-visit`.
- O procedimento `dfs-visit` é chamado exatamente uma vez para cada vértice (somente para vértices brancos), e no início de `dfs-visit` o vértice é pintado de cinza.

Análise do algoritmo

Consumo de tempo (análise agregada)

- Os loops nas linhas 1 a 3 e nas linhas 5 a 7 de `dfs` consomem tempo $\Theta(V)$, sem contar o tempo das chamadas a `dfs-visit`.
- O procedimento `dfs-visit` é chamado exatamente uma vez para cada vértice (somente para vértices brancos), e no início de `dfs-visit` o vértice é pintado de cinza.
- Durante a execução de `dfs-visit(v)`, o loop nas linhas 4 a 7 é executado $|v.adj|$ vezes, como $\sum_{v \in V} |v.adj| = \Theta(E)$, o custo total da execução das linhas 4 a 7 de `dfs-visit` é $\Theta(E)$

Análise do algoritmo

Consumo de tempo (análise agregada)

- Os loops nas linhas 1 a 3 e nas linhas 5 a 7 de `dfs` consomem tempo $\Theta(V)$, sem contar o tempo das chamadas a `dfs-visit`.
- O procedimento `dfs-visit` é chamado exatamente uma vez para cada vértice (somente para vértices brancos), e no início de `dfs-visit` o vértice é pintado de cinza.
- Durante a execução de `dfs-visit(v)`, o loop nas linhas 4 a 7 é executado $|v.adj|$ vezes, como $\sum_{v \in V} |v.adj| = \Theta(E)$, o custo total da execução das linhas 4 a 7 de `dfs-visit` é $\Theta(E)$

Conclusão

A complexidade do algoritmo `dfs(G)` é $O(V + E)$.

Propriedades

- O subgrafo predecessor definido pela busca em profundidade é uma floresta. Um vértice v é descendente de u nesta floresta se e somente se v foi descoberto durante o tempo em que u era cinza.

Propriedades

- O subgrafo predecessor definido pela busca em profundidade é uma floresta. Um vértice v é descendente de u nesta floresta se e somente se v foi descoberto durante o tempo em que u era cinza.
- Tempo de descoberta e término apresentam estrutura parentizada. Se o tempo de descoberta representar um parênteses esquerdo e o tempo de término representar um parênteses direito, então o histórico de descobertas e términos formam uma expressão bem formada (parênteses aninhados adequadamente).

Propriedades

- O subgrafo predecessor definido pela busca em profundidade é uma floresta. Um vértice v é descendente de u nesta floresta se e somente se v foi descoberto durante o tempo em que u era cinza.
- Tempo de descoberta e término apresentam estrutura parentizada. Se o tempo de descoberta representar um parênteses esquerdo e o tempo de término representar um parênteses direito, então o histórico de descobertas e términos formam uma expressão bem formada (parênteses aninhados adequadamente).
- DFS pode ser usado para classificar arestas de um grafo.

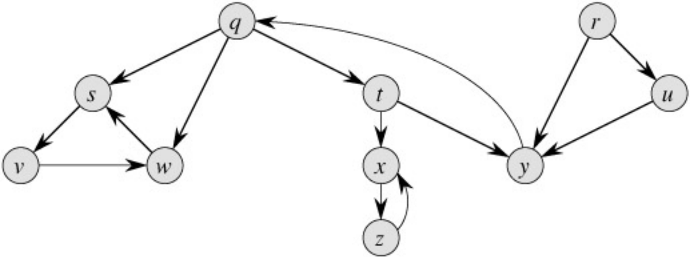
Classificação de arestas

Podemos definir quadro tipos de arestas em termos da floresta gerada pela busca em profundidade:

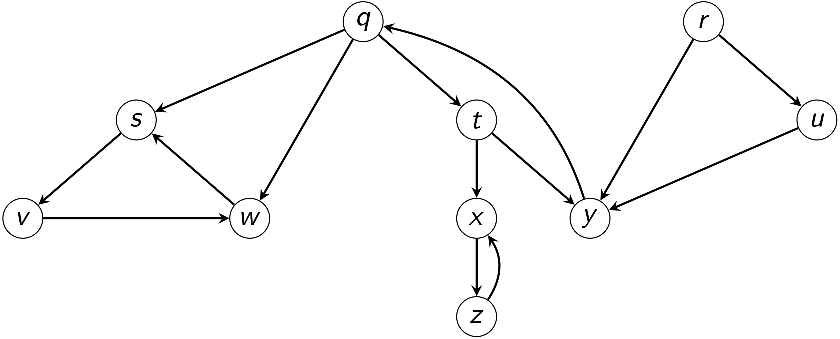
- **Arestas de árvore** são as arestas pertencentes à floresta (subgrafo de predecessores). Uma aresta (u, v) é uma aresta da árvore se v foi descoberto primeiro pela exploração da aresta (u, v).
- **Arestas de retorno** são as arestas (u, v) que conectam um vértice u a um ancestral v na árvore de busca em profundidade.
- **Arestas para frente (avanço)** são as arestas (u, v) que não são arestas da árvore e conectam o vértice u a um descendente v na árvore de busca em profundidade.
- **Arestas cruzadas (cruzamento)** são todas as outras arestas

Exercício 1

[Cormen 22.3-2] Mostre como a busca em profundidade funciona sobre o grafo da figura abaixo. Suponha que o loop for das linhas 5 a 7 do procedimento `DFS` considera os vértices em ordem alfabética, e suponha que cada lista de adjacência esteja em ordem alfabética. Mostre os tempos de descoberta e término para cada vértice, e mostre também a classificação de cada aresta.



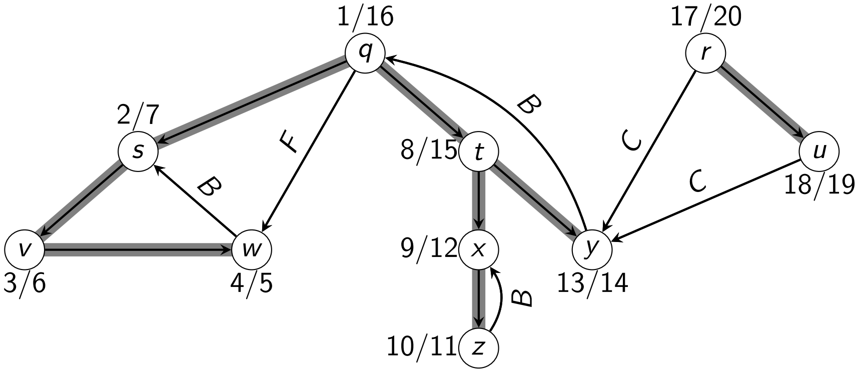
Resolução do Exercício 1



Legenda?

- B - Aresta de retorno (back edges)
- C - Aresta de cruzamento (cross edges)
- F - Aresta de avanço (forward edges)

Resolução do Exercício 1



Legenda?

- B - Aresta de retorno (back edges)
- C - Aresta de cruzamento (cross edges)
- F - Aresta de avanço (forward edges)

Exercício 2

[Cormen 22.3-12] Mostre que uma busca em profundidade de um grafo não orientado G pode ser usada para identificar os componentes conexos de G, e que a floresta da busca em profundidade contém tantas árvores quantos componentes conexos existem em G. Mais precisamente, mostre como modificar a busca em profundidade de modo que cada vértice v receba a atribuição de uma etiqueta inteira $v.cc$ entre 1 e k, onde k é o número de componentes conexos de G, de tal forma que $u.cc = v.cc$ se e somente se u e v estiverem no mesmo componente conexo.