DCC206 – Algoritmos 1

Aula 02 – Caminhamento em grafos

Professor Renato Vimieiro

DCC/ICEx/UFMG

Introdução

- Os algoritmos para caminhamento em grafos a serem estudados hoje têm aplicações em várias situações
- A busca em largura e em profundidade são amplamente usadas em Inteligência Artificial, Otimização Combinatória, na verificação de propriedades de grafos …
- Eles são usados não só para o caminhamento explícito em grafos, mas também na construção de algoritmos que executam buscas em espaços de soluções:
  - Exemplos em Mineração de Dados: Apriori (busca em largura); FPGrowth (busca em profundidade)

Introdução

- Em grafos, a busca em largura e em profundidade podem ser usadas para:
  - Determinar se há, e qual o caminho mais curto entre dois vértices u e v
  - Determinar se há ciclos no grafo
  - Identificar os componentes conexos do grafo
  - Determinar a ordem de execução de tarefas onde existe interdependência
  - …

Introdução

- Para motivar a compreensão dos algoritmos, vamos considerar uma possível aplicação: busca de caminhos em labirintos
- Suponha que queremos encontrar a solução para um labirinto como o da figura abaixo
- Como modelar esse problema com grafos? E, intuitivamente, como resolvê-lo?

Busca em profundidade (Depth-first search)

- Intuitivamente, queremos seguir em um caminho até chegarmos em um beco sem saída. Nesse momento, regressamos até a última interseção e seguimos em outra direção
- Essa é a intuição por trás da busca em profundidade
- Levitin diz que a busca em profundidade é o caminhamento dos bravos
  - Os caminhos que os afastam cada vez mais de casa são sempre os escolhidos

Busca em profundidade

- A ideia central é: ao se deparar com um caminho ainda não percorrido, segui-lo
- No contexto de grafos, essa ideia se traduz em:
  - Inicia-se a busca em um vértice origem
  - Como não queremos retornar a vértices já visitados (exceto quando queremos explorar outras direções), marcamos os vértices quando os visitamos
  - Ao chegar em um vértice, visitamos um vértice adjacente ainda não visitado
  - Se não houver mais vértices adjacentes, retornamos a um antecessor para explorar outra direção
    - Se já tivermos explorado todos os adjacentes, então podemos marcar o vértice como finalizado

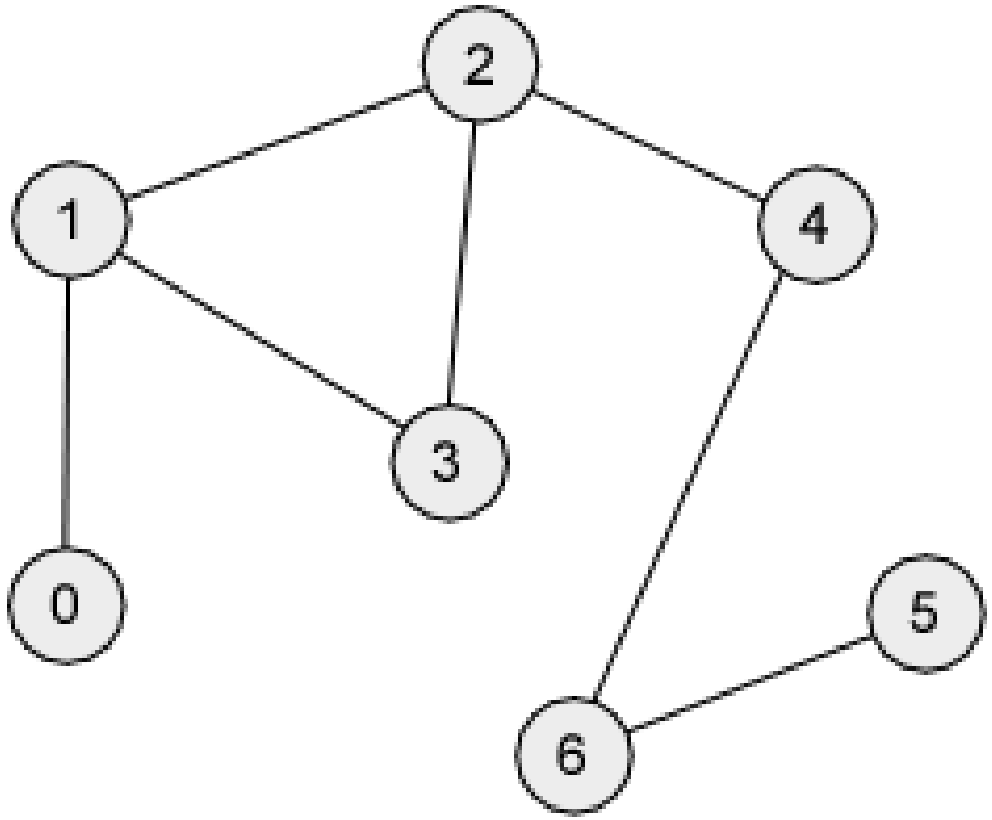
Busca em profundidade

```
buscaProfundidade(g)
    marcado = [false]*g.V
    antecessor = [-1]*g.V
    para v ∈ g.V
        se !marcado[v]
            dfs(g,v,antecessor,marcado)
```

Busca profundidade

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

Busca profundidade: exemplo



Busca em profundidade

- A busca deve ser iniciada em cada vértice para garantir que todos os componentes conexos são explorados
- Assim, o custo do algoritmo inicial (nível mais alto anterior) é O(|V|)
- A parte recursiva explora todos os vértices adjacentes de cada um visitado
- A forma como o grafo é representado impacta no custo do algoritmo
  - Se representarmos o grafo como matriz de adjacências, gastamos tempo O(|V|) para visitar os adjacentes e custo total O(|V|^2)
  - Se usarmos listas de adjacências, gastamos Θ(d(v)) e custo total O(|V|+|E|)
- Em ambos os casos, o tempo é proporcional ao tamanho da entrada

Busca em profundidade

- Embora a definição recursiva seja a mais direta, podemos desenhar uma versão iterativa explicitando a pilha de execução

```
dfs(g, v, antecessor, marcado)
   empilhar(v)
   enquanto pilha não estiver vazia
      v = desempilhar()
      se !marcado[v]
         marcado[v] = true
         para u ∈ g.adj(v)
            se antecessor[u] == -1
                  antecessor[u] = v
            empilhar(u)
```

Verificar se um grafo é acíclico

- A busca em profundidade pode ser usada para verificar se um grafo é acíclico
- Toda aresta conectando um vértice marcado a um não marcado é chamada de **aresta de árvore**
- Toda aresta conectando um vértice marcado a outro marcado (um antecessor na busca) é chamada de **aresta de retorno**
- Arestas de retorno sempre formam ciclos no grafo

Determinar componentes conexos

- A busca em profundidade pode ser usada para determinar a quantidade e formação dos componentes conexos do grafo
- Todos os vértices num mesmo componente conexo são visitados durante uma chamada recursiva da função *dfs* a partir de um vértice de origem
- Usar um vetor adicional para armazenar o identificador do componente conexo
- Incrementar o contador de componentes a cada iteração do loop na função buscaProfundidade

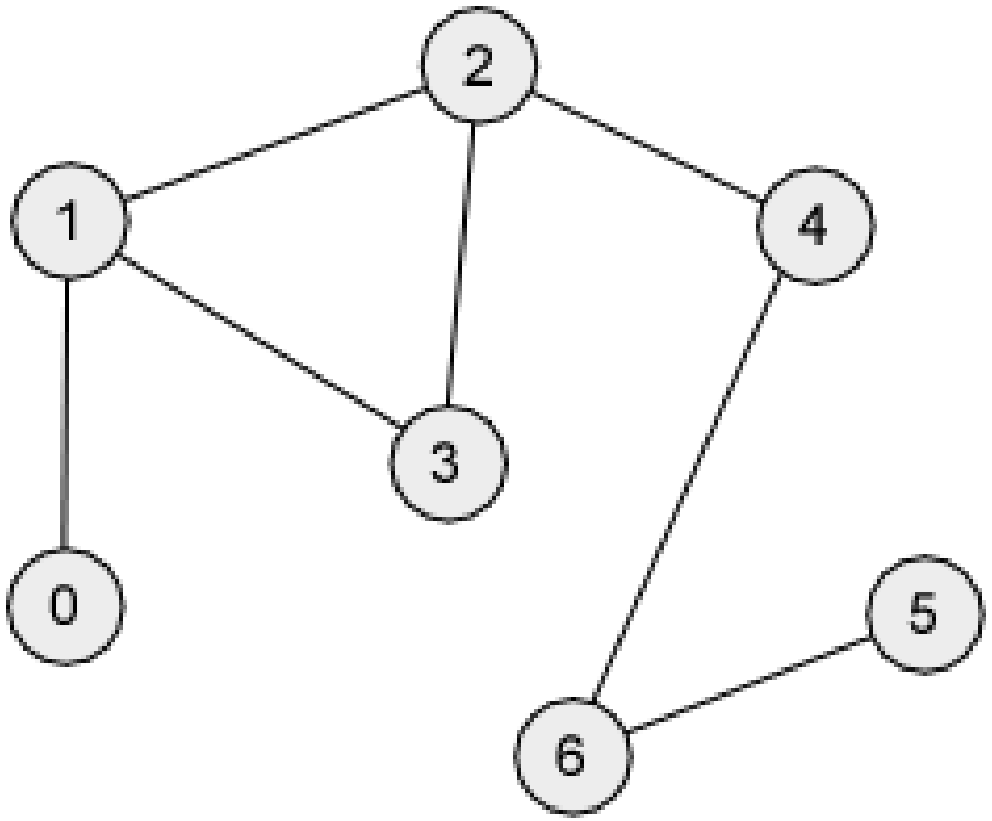
Busca em largura (breadth-first search)

- Uma segunda forma de caminhar em grafos é em largura
- Levitin chamou essa busca de o caminhamento dos cautelosos
  - Os caminhos mais próximos são visitados antes de se afastar
- A ideia é avaliar todos os vizinhos imediatos para depois explorar os vizinhos dos vizinhos
- Em grafos significa:
  - Iniciar busca na origem
  - Adicionar vértices adjacentes ainda não marcados à lista de vértices a explorar
  - Retirar vértice da lista e marcá-lo
  - Repetir enquanto a lista não estiver vazia

Busca em largura

```
bfs(g, v, antecessor, marcado)
    inserir_fim(v)
    enquanto fila não estiver vazia
        v = remover_primeiro()
        se !marcado[v]
            marcado[v] = true
            para u ∈ g.adj(v)
                se antecessor[u] == -1
                    antecessor[u] = v
                inserir_fim(u)
```

Busca em largura: exemplo



Busca em largura

- É perceptível que o algoritmo é bastante similar à busca em profundidade
- De fato, a alteração se dá apenas na estrutura de dados usada para armazenar os vértices a serem explorados
  - Fila (BFS) x Pilha (DFS)
- Como os custos dessas estruturas são idênticos, o custo do algoritmo é o mesmo da busca em profundidade
  - O(|V| + |E|) para representações usando listas de adjacências

Caminho mais curto a partir de uma origem

- Um efeito colateral da busca em largura é que ela sempre retorna o caminho mais curto desde a origem a qualquer outro vértice no mesmo componente conexo
- O caminho mais curto entre a origem e um vértice qualquer pode ser obtido através do vetor de antecessores
- Iniciando de v, percorre-se seus antecessores até a origem
- Se em algum ponto um antecessor diferente da origem não tiver antecessor, não existe caminho da origem ao vértice

Caminho mais curto a partir de uma origem

```
caminho(origem, v, antecessor)
    se origem == v
       imprimir v
    senão se antecessor[v] == -1
          imprimir "não há caminho entre os vértices”
    senão
          caminho(origem, antecessor[v], antecessor)
```

Leitura

- Seções 22.2 e 22.3 (CLRS)
- Seções 3.5 (Levitin)
- Seções 3.2 e 3.3 (Kleinberg e Tardos)