Téc. Avançadas de Programação
Pseudocódigos
Prof. Leandro M. Zatesko
1º semestre de 2014

Buscas em Grafos

1 Buscas simples

```
BuscaGenérica(G, s):
1  para todo u ∈ V(G), faça:
2     visitado[u] ← F;
3  visitado[s] ← V;
4  inicialize a estrutura de dados S com o vértice s;
5  enquanto S não estiver vazia, faça:
6     remova um vértice u de S;
7     para todo w ∈ N_G(u), faça:
8        se visitado[w] = F, então:
9           visitado[w] ← V;
10          insira o vértice w em S.
```

Algoritmo 1

Algumas observações:

- Se a estrutura S é uma fila, temos uma *Busca em Largura* (*Breadth-First Search*, *BFS*).
- Se é uma pilha, temos uma *Busca em Profundidade* (*Depth-First Search*, *DFS*).
- Em competições de programação, filas e pilhas podem ser implementadas simplesmente com um vetor estático com suficientes $|V(G)|$ posições, já que cada vértice entra em S no máximo uma vez.
- Se o grafo é representado por uma lista de adjacências, a complexidade de tempo é $O(|V(G)| + |E(G)|)$.
- Se é representado por uma matriz de adjacências, é $O((|V(G)|)^2)$.
- Para buscas em grafos dirigidos, basta trocarmos $N_G(u)$ na linha 7 por $N_G^+(u)$.
- É importante que o vértice seja marcado como visitado quando *entra* em S, não quando sai, para que não corramos o risco de um mesmo vértice entrar em S mais de uma vez, o que pode ocorrer se a busca for em largura.

- Se estamos buscando um vértice destino t específico, podemos abortar a busca ao encontrarmos t.

Pode ser útil guardarmos o *pai* de cada vértice na busca:

```
BuscaGenérica(G, s):
para todo u ∈ V(G), faça:
   visitado[u] ← F;
visitado[s] ← V;
pai[s] ← s;
inicialize a estrutura de dados S com o vértice s;
enquanto S não estiver vazia, faça:
   remova um vértice u de S;
   para todo w ∈ N_G(u), faça:
      se visitado[w] = F, então:
         visitado[w] ← V;
         pai[w] ← u;
         insira o vértice w em S.
```

Algoritmo 2

Para calcularmos a distância do vértice s para todos os vértices alcançáveis a partir de s (ou para algum destino específico t), não tendo as arestas pesos, podemos usar a BFS:

```
BFS(G, s):
para todo u ∈ V(G), faça:
   dist[u] ← ∞;
pai[s] ← s;
dist[s] ← 0;
inicialize a fila Q com o vértice s;
enquanto a fila Q não estiver vazia, faça:
   desenfileire um vértice u de Q;
   para todo w ∈ N_G(u), faça:
      se dist[w] = ∞, então:
         pai[w] ← u;
         dist[w] ← dist[u] + 1;
         enfileire o vértice w em Q.
```

Algoritmo 3

Como a DFS utiliza uma pilha como estrutura de dados, pode ser implementada recursivamente:

```
variável global: visitado[];

DFS(G, s):
para todo u ∈ V(G), faça:
    visitado[u] ← F;
visitado[s] ← V;
DFS'(G, s).

DFS'(G, u):
para todo w ∈ N_G(u), faça:
    se visitado[w] = F, então:
        visitado[w] ← V;
        DFS'(G, w).
```

Algoritmo 4

2 Problemas clássicos

1. Como decidir se um grafo é bipartido?
2. Como decidir se um grafo é cíclico?
3. Como decidir se um grafo é conexo?
4. Como contar o número de componentes conexas de um grafo?

3 Algoritmo de Dijkstra

Para grafos com pesos nas arestas, a distância de um vértice de origem s para qualquer vértice alcançável a partir de s pode ser calculada através do *Algoritmo de Dijkstra*.

```
Dijkstra(G, s):
para todo u ∈ V(G), faça:
    dist[u] ← ∞;
dist[s] ← 0;
inicialize a estrutura de dados H com V(G) ref. dist[];
enquanto H não estiver vazia, faça:
    extraia o vértice u de H com dist[] mínimo;
    para todo w ∈ N_G(u), faça:
        se dist[w] > dist[u] + ρ(u, w), então:
            dist[w] ← dist[u] + ρ(u, w);
            decresça a chave w em H ref. dist[w].
```

Algoritmo 5

Algumas observações:

- Se H for um simples vetor ordenado, a complexidade é $O(|V(G)||E(G)|)$.
- Se H for uma *heap* binária, a complexidade é $O((|V(G)|+|E(G)|)\log|V(G)|)$.
- Para grafos dirigidos, basta trocar $N_G(u)$ por $N_G^+(u)$.
- Pode ser interessante armazenar o *pai* de cada vértice.
- Se em algum momento extraímos u com $\boldsymbol{dist}[u]=\infty$, significa que o grafo não é conexo, e podemos parar a busca por aí.

O Algoritmo 5 considera que G é conexo. Se G não necessariamente é conexo, faz-se necessária uma ligeira alteração, apresentada no Algoritmo 6. Também exibimos no Algoritmo 7 uma versão em que se armazena o *pai* de cada vértice e em que a busca é abortada ao termos $\operatorname{dist}(s,t)$ computado para um dado t.

```
Dijkstra(G, s):
para todo u ∈ V(G), faça:
   dist[u] ← ∞;
dist[s] ← 0;
inicialize a estrutura de dados H com |V(G)| ref. dist[];
enquanto H não estiver vazia, faça:
   extraia o vértice u de H com dist[] mínimo;
   se dist[u] = ∞, pare;
   para todo w ∈ N_G(u), faça:
      se dist[w] > dist[u] + ρ(u,w), então:
         dist[w] ← dist[u] + ρ(u,w);
         decresça a chave w em H ref. dist[w].
```

Algoritmo 6

```
Dijkstra(G, s, t):
para todo u ∈ V(G), faça:
   dist[u] ← ∞;
dist[s] ← 0;
pai[s] ← s;
inicialize a estrutura de dados H com |V(G)| ref. dist[];
enquanto H não estiver vazia, faça:
   extraia o vértice u de H com dist[] mínimo;
   se dist[u] = ∞ ou u = t, pare;
   para todo w ∈ N_G(u), faça:
      se dist[w] > dist[u] + ρ(u,w), então:
         dist[w] ← dist[u] + ρ(u,w);
         pai[w] ← u;
         decresça a chave w em H ref. dist[w].
```

Algoritmo 7

Se conhecemos uma função heurística h que, para todo $u \in V(G)$, estima, mas nunca superestima, $\mathrm{dist}(u,t)$, ou seja, se $h(u) \leqslant \mathrm{dist}(u,t)$ para todo $u \in V(G)$, então, podemos *podar* a busca, evitando que o Algoritmo de Dijkstra desça a ramos na árvore de busca irrelevantes para a computação de $\mathrm{dist}(s,t)$. Esta variante é conhecida como *Algoritmo A**. Por exemplo, se os vértices são pontos numa malha cartesiana e se $\rho(x,y)$ é a distância euclidiana entre x e y, uma boa heurística para $h(u)$ é a distância euclidiana entre u e t, pois com certeza não excede a soma mínima dos pesos das arestas de um caminho de u a t.

$\mathrm{A}^*(G,s,t)$:
para todo $u \in V(G)$, **faça**:
  $\boldsymbol{dist}[u] \leftarrow \infty$;
  $\boldsymbol{g}[u] \leftarrow \infty$;
$\boldsymbol{dist}[s] \leftarrow 0$;
$\boldsymbol{g}[s] \leftarrow h(s)$;
$\boldsymbol{pai}[s] \leftarrow s$;
inicialize a estrutura de dados H com $|V(G)|$ ref. $\boldsymbol{g}[]$;
enquanto H não estiver vazia, **faça**:
  extraia o vértice u de H com $\boldsymbol{g}[]$ mínimo;
  se $\boldsymbol{dist}[u] = \infty$ ou $u = t$, **pare**;
  para todo $w \in N_G(u)$, **faça**:
    se $\boldsymbol{dist}[w] > \boldsymbol{dist}[u] + \rho(u,w)$, **então**:
      $\boldsymbol{dist}[w] \leftarrow \boldsymbol{dist}[u] + \rho(u,w)$;
      $\boldsymbol{g}[w] \leftarrow \boldsymbol{dist}[w] + h(w)$;
      $\boldsymbol{pai}[w] \leftarrow u$;
      decresça a chave w em H ref. $\boldsymbol{g}[w]$.

Algoritmo 8

4 Ordenação Topológica em DAGs

Uma *ordenação topológica* num grafo dirigido acíclico (*directed acyclic graph*, *DAG*) é uma permutação $\pi\colon [1..|V(G)|] \to |V(G)|$ tal que, para todo $i \in [1..|V(G)|]$ e todo $j < i$, não existe caminho de $\pi(j)$ a $\pi(i)$ que passe por $\pi(k)$ para algum $k > i$. Podemos encontrar uma ordenação topológica através do *Algoritmo de Tarjan*, que simplesmente executa uma DFS para cada *origem* s de G. Um vértice s de um grafo dirigido é dito uma *origem* se $d_G^-(s) = 0$. Como o grafo é acíclico, é certo haver ao menos uma origem se $V(G) \neq \emptyset$.

A complexidade do Algoritmo de Tarjan é $O(|V(G)| + |E(G)|)$. Para alguns problemas o Algoritmo de Tarjan pode não ser de grande ajuda, sendo mais apropriado um algoritmo mais antigo (e ligeiramente mais complicado), conhecido como *Algoritmo de Khan*, que realiza a busca não em profundidade, mas em largura.

A complexidade do Algoritmo de Khan também é linear, mas, para isso, alguns cuidados com a implementação devem ser tomados:

variáveis globais: i, ***visitado***[], ***pi***[];

<u>TARJAN(G)</u>:
$i \leftarrow |V(G)|$;
para todo $u \in V(G)$, **faça**:
  visitado$[u] \leftarrow$ **F**;
para todo $s \in V(G)$ que satisfaz $d^-_G(s) = 0$, **faça**:
  visitado$[s] \leftarrow$ **V**;
  TARJAN'(G, s);
  $\pi[i] \leftarrow s$; $i \leftarrow i - 1$.

<u>TARJAN'(G, u)</u>:
para todo $w \in N^+_G(u)$, **faça**:
  se ***visitado***$[w] =$ **F**, **então**:
    visitado$[w] \leftarrow$ **V**;
    TARJAN'(G, w);
    $\pi[i] \leftarrow w$; $i \leftarrow i - 1$.

Algoritmo 9

<u>KHAN(G)</u>:
1 $i \leftarrow 1$; inicialize a fila Q vazia;
2 **para** todo $s \in V(G)$ que satisfaz $d^-_G(s) = 0$, **faça**:
3   $\pi[i] \leftarrow s$; $i \leftarrow i + 1$;
4   enfileire o vértice s em Q;
5 **enquanto** a fila Q não estiver vazia, **faça**:
6   desenfileire um vértice u de Q;
7   **para todo** $w \in N^+_G(u)$, **faça**:
8   remova a aresta (u, w) de G;
9   **se** $d^-_G(w) = 0$, **faça**:
10       $\pi[i] \leftarrow w$; $i \leftarrow i + 1$;
11   enfileire o vértice w em Q.

Algoritmo 10

1. Percorra a lista das adjacências de u na linha 7 de trás para frente, assim, remover a aresta (u, w) na linha 8 pode ser feito em tempo $O(1)$. Do contrário, tomaria tempo $O(d^+_G(u))$.

2. Guarde o grau de chegada de todo vértice num vetor. Assim, quando precisar verificar se $d^-_G(w) = 0$, basta fazer uma consulta de tempo $O(1)$ ao vetor. Não se esqueça de, sempre que remover uma aresta (u, w), decrementar $d^-_G(w)$.

5 Articulações e pontes

Uma articulação num grafo simples G é um vértice x cuja remoção incrementa o número de componentes conexas em G. Para encontrar as articulações de um grafo, o

Algoritmo de Hopcroft–Tarjan usa uma DFS.

```
variáveis globais: i, ordem[], ciclo[], pai[], art[], r, f_r;

Hopcroft–Tarjan(G):
i ← 1;
para todo u ∈ V(G), faça:
   ordem[u] ← 0; ciclo[u] ← 0; art[u] ← F;
para todo s ∈ V(G), faça:
   se ordem[s] = 0, então:
      r ← s; f_r ← 0; pai[r] ← r;
      Hopcroft–Tarjan'(G, r);
      se f_r > 1, então, art[r] ← V.

Hopcroft–Tarjan'(G, u):
ordem[u] ← i; ciclo[u] ← i; i ← i + 1;
para todo w ∈ N_G(u), faça:
   se ordem[w] = 0, então:
      pai[w] ← u;
      se u = r, então, f_r ← f_r + 1;
      Hopcroft–Tarjan'(G, w);
      se ciclo[w] ⩾ ordem[u], então:
         art[u] ← V;
      ciclo[u] ← min{ciclo[u], ciclo[w]};
   senão, se w ≠ pai[u], então:
      ciclo[u] ← min{ciclo[u], ciclo[w]}.
```

Algoritmo 11

O Algoritmo de Hopcroft–Tarjan também pode encontrar as pontes de G, como indica o Algoritmo 12.

```
variáveis globais: i, ordem[], ciclo[], pai[], art[], ponte[], r, f_r;

Hopcroft–Tarjan(G):
i ← 1;
para todo u ∈ V(G), faça:
   ordem[u] ← 0; ciclo[u] ← 0; art[u] ← F;
para todo e ∈ E(G), faça:
   ponte[e] ← F;
para todo s ∈ V(G), faça:
   se ordem[s] = 0, então:
      r ← s; f_r ← 0; pai[r] ← r;
      Hopcroft–Tarjan'(G, r);
      se f_r > 1, então, art[r] ← V.

Hopcroft–Tarjan'(G, u):
ordem[u] ← i; ciclo[u] ← i; i ← i + 1;
para todo w ∈ N_G(u), faça:
   se ordem[w] = 0, então:
      pai[w] ← u;
      se u = r, então, f_r ← f_r + 1;
      Hopcroft–Tarjan'(G, w);
      se ciclo[w] ⩾ ordem[u], então:
         art[u] ← V;
      se ciclo[w] > ordem[u], então:
         ponte[{u, w}] ← V;
      ciclo[u] ← min{ciclo[u], ciclo[w]};
   senão, se w ≠ pai[u], então:
      ciclo[u] ← min{ciclo[u], ciclo[w]}.
```

Algoritmo 12

6 Componentes fortemente conexas em grafos dirigidos

Uma componente fortemente conexa de um grafo dirigido G é um subgrafo induzido H de G em que, para todo x e todo y em $V(H)$, existe caminho em H tanto de x para y quanto de y para x. O outro *Algoritmo de Tarjan* encontra as componentes fortemente conexas de um grafo dirigido G conexo em complexidade $O(|V(G)| + |E(G)|)$.

variáveis globais: i, $componentes$, $\boldsymbol{ordem}[]$, $\boldsymbol{ciclo}[]$, $\boldsymbol{componente}[]$;

<u>TARJAN(G)</u>:
$i \leftarrow 1$; $componentes \leftarrow 0$;
para todo $u \in V(G)$, **faça**:
  $\boldsymbol{ordem}[u] \leftarrow 0$; $\boldsymbol{ciclo}[u] \leftarrow 0$;
inicialize uma pilha vazia S;
para todo $s \in V(G)$, **faça**:
  se $\boldsymbol{ordem}[s] = 0$, **então**:
    TARJAN'(G, s).

<u>TARJAN'(G, u)</u>:
$\boldsymbol{ordem}[u] \leftarrow i$; $\boldsymbol{ciclo}[u] \leftarrow i$; $i \leftarrow i + 1$;
empilhe u em S;
para todo $w \in N_G^+(u)$, **faça**:
  se $\boldsymbol{ordem}[w] = 0$, **então**:
    TARJAN'(G, w);
  se w está na pilha S, **então**:
    $\boldsymbol{ciclo}[u] \leftarrow \min\{\boldsymbol{ciclo}[u], \boldsymbol{ciclo}[w]\}$;
se $\boldsymbol{ordem}[u] = \boldsymbol{ciclo}[u]$, **então**:
  $componentes \leftarrow componentes + 1$;
  faça:
    desempilhe um w de S;
    $\boldsymbol{componente}[w] \leftarrow componentes$;
  até que $w = u$.

Algoritmo 13