Algoritmos em Grafos

Ana Paula Tomás

LEIC - Desenho e Análise de Algoritmos
Universidade do Porto

Março 2024

Tópicos a abordar

1. Grafos
2. Pesquisa em Largura e em Profundidade
3. Componentes fortemente conexas (SCC)
4. Aplicação de SCC à Resolução de 2-SAT

Grafos - modelo fundamental em computação

Os grafos representam um **modelo fundamental em computação** e noutras áreas (biologia, química, sociologia, economia, etc).

Recordar noções:

- nó/vértice
- ramo/aresta/arco
- origem e fim de um ramo
- extremos de um ramo
- grafos dirigidos e não dirigidos
- grafos com valores nos ramos
- **percurso** e circuito
- **graus** dos nós
- **caminho** e **ciclo**
- origem e fim de um percurso
- **adjacentes** de nó v
- **acessibilidade**
- **conectividade**
- **árvore**
- **DAG - grafo dirigido acíclico**

Exemplos de problemas em grafos

- **Existe caminho** de um nó s para um nó t?
- Qual é o **caminho mais curto** de s para t?
- Qual é o **caminho mais longo** de s para t? Num DAG? Em geral?
- Quais são os **nós acessíveis de** um nó s?
- Que nós acessíveis de s garantem que pode voltar a s, se os visitar?

Representações para grafos

Seja $G = (V, E)$ um grafo dirigido, com $V = \{v_1, v_2, \ldots, v_n\}$, para $n \geq 1$. Seja $|E| = m$. O grafo G pode ser representado por:

- **Matriz de adjacências:** matriz booleana M, de dimensão $n \times n$, com $M_{ij} = 1$ se $(v_i, v_j) \in E$ e $M_{ij} = 0$ se $(v_i, v_j) \notin E$.

Representações para grafos

Seja $G = (V, E)$ um grafo dirigido, com $V = \{v_1, v_2, \ldots, v_n\}$, para $n \geq 1$. Seja $|E| = m$. O grafo G pode ser representado por:

- **Matriz de adjacências:** matriz booleana M, de dimensão $n \times n$, com $M_{ij} = 1$ se $(v_i, v_j) \in E$ e $M_{ij} = 0$ se $(v_i, v_j) \notin E$.
  - Desvantagem: **memória** $\Theta(n^2)$. Inapropriado se o grafo for *esparso*, isto é, se $m \in O(n)$.
  - Vantagem: poder determinar em **tempo** $\Theta(1)$ se $(v_i, v_j) \in E$.

Representações para grafos

Seja $G = (V, E)$ um grafo dirigido, com $V = \{v_1, v_2, \ldots, v_n\}$, para $n \geq 1$. Seja $|E| = m$. O grafo G pode ser representado por:

- **Matriz de adjacências:** matriz booleana M, de dimensão $n \times n$, com $M_{ij} = 1$ se $(v_i, v_j) \in E$ e $M_{ij} = 0$ se $(v_i, v_j) \notin E$.
  - Desvantagem: **memória** $\Theta(n^2)$. Inapropriado se o grafo for *esparso*, isto é, se $m \in O(n)$.
  - Vantagem: poder determinar em **tempo** $\Theta(1)$ se $(v_i, v_j) \in E$.
- **Listas de adjacências:** a cada nó v associa a lista de seus adjacentes, i.e., a lista de nós w tais que $(v, w) \in E$.

Representações para grafos

Seja $G = (V, E)$ um grafo dirigido, com $V = \{v_1, v_2, \ldots, v_n\}$, para $n \geq 1$. Seja $|E| = m$. O grafo G pode ser representado por:

- **Matriz de adjacências:** matriz booleana M, de dimensão $n \times n$, com $M_{ij} = 1$ se $(v_i, v_j) \in E$ e $M_{ij} = 0$ se $(v_i, v_j) \notin E$.
  - Desvantagem: **memória** $\Theta(n^2)$. Inapropriado se o grafo for *esparso*, isto é, se $m \in O(n)$.
  - Vantagem: poder determinar em **tempo** $\Theta(1)$ se $(v_i, v_j) \in E$.

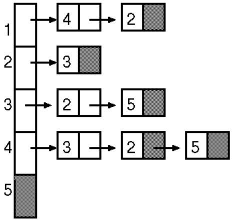
- **Listas de adjacências:** a cada nó v associa a lista de seus adjacentes, i.e., a lista de nós w tais que $(v, w) \in E$. $\Leftarrow$ assumiremos esta representação

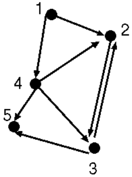
Representações para grafos

Seja $G = (V, E)$ um grafo dirigido, com $V = \{v_1, v_2, \dots, v_n\}$, para $n \geq 1$. Seja $|E| = m$. O grafo G pode ser representado por:

- **Matriz de adjacências:** matriz booleana M, de dimensão $n \times n$, com $M_{ij} = 1$ se $(v_i, v_j) \in E$ e $M_{ij} = 0$ se $(v_i, v_j) \notin E$.
  - Desvantagem: **memória** $\Theta(n^2)$. Inapropriado se o grafo for *esparso*, isto é, se $m \in O(n)$.
  - Vantagem: poder determinar em **tempo** $\Theta(1)$ se $(v_i, v_j) \in E$.

- **Listas de adjacências:** a cada nó v associa a lista de seus adjacentes, i.e., a lista de nós w tais que $(v, w) \in E$. $\Leftarrow$ assumiremos esta representação





- Vantagem: **memória** $\Theta(n + m)$. Em várias aplicações reais, os grafos são esparsos.
- Desvantagem: no pior caso, determinar se $(v_i, v_j) \in E$ requer **tempo** $\Theta(|Adjs(v_i)|)$.

Exemplo de implementação em Java

```java
import java.util.Scanner;
import java.util.LinkedList;

class Edge {
    private int enode;
    private int value;

    Edge(int endv, int v) {
        enode = endv;
        value = v;
    }
    public int endnode() {
        return enode;
    }
    public int value() {
        return value;
    }
    public void newvalue(int v) {
        value = v;
    }
}
```

Exemplo de implementação em Java (cont.)

```
class Node {
    //private int label;
    private LinkedList<Edge> neighbours;

    Node() {
        neighbours  = new LinkedList<Edge>();
    }

    public LinkedList<Edge> adjs() {
        return neighbours;
    }

}
```

Exemplo de implementação em Java (cont.)

Grafo dirigido pesado, com valor inteiro em cada aresta e nós numerados de 1 a n.

```
class Graph {
    private Node verts[];
    private int nverts, nedges;

    public Graph(int n) {
        nverts = n;
        nedges = 0;
        verts  = new Node[n+1];
        for (int i = 0 ; i <= n ; i++)
            verts[i] = new Node();
    }

    public int num_vertices(){
        return nverts;
    }
```

Exemplo de implementação em Java (cont.)

```
    public int num_edges(){
        return nedges;
    }

    public LinkedList<Edge> adjs_no(int i) {
        return verts[i].adjs();
    }

    public void insert_new_edge(int i, int j, int value_ij){
        verts[i].adjs().addFirst(new Edge(j,value_ij));
        nedges++;
    }

    public Edge find_edge(int i, int j){
        for (Edge adj: adjs_no(i))
            if (adj.endnode() == j) return adj;
        return null;
    }
}
```

Existe caminho de um nó s para um nó t?

BFS e DFS são **estratégias genéricas de pesquisa exaustiva**, com ideias distintas:

- Pesquisa em largura (*Breadth-First Search*) – BFS:

  visita s, depois todos os vizinhos de s, a seguir *os vizinhos dos vizinhos* de s que ainda não tenham sido visitados, e sucessivamente.

- Pesquisa em profundidade (*Depth-First Search*) – DFS:

  visita s, depois visita um dos vizinhos de s, a seguir *um vizinho desse vizinho* de s que ainda não tenha sido visitado, e sucessivamente. Quando o próximo nó já não tem mais vizinhos por visitar, a pesquisa prossegue com a análise de outro vizinho do **pai desse nó**, que ainda não tenha sido visitado.

Estratégia: **pesquisa em largura** a partir do nó s em $G = (V, A)$

BFS_Visit(s, G) // **Breadth-First Search**

Para cada $v \in G.V$ fazer
  $visitado[v] \leftarrow \texttt{false}$;
  $pai[v] \leftarrow \texttt{NULL}$;
$visitado[s] \leftarrow \texttt{true}$;
$Q \leftarrow \textsc{MkEmptyQueue}()$;
$\textsc{Enqueue}(s, Q)$;
Repita
  $v \leftarrow \textsc{Dequeue}(Q)$;
  Para cada $w \in G.Adjs[v]$ fazer
    Se $visitado[w] = \texttt{false}$ então
      $\textsc{Enqueue}(w, Q)$;
      $visitado[w] \leftarrow \texttt{true}$;
      $pai[w] \leftarrow v$; // v precede w no caminho de s para w
até $(\textsc{QueueIsEmpty}(Q) = \texttt{true})$;

Visita o grafo $G = (V, A)$ em largura a partir de s

Propriedades

- $pai[\cdot]$ define uma árvore com raíz s, que se chama **árvore de pesquisa em largura a partir de s**. Os ramos são os pares $(pai[v], v)$ com $pai[v] \neq \texttt{NULL}$.

Visita o grafo $G = (V, A)$ em largura a partir de s

Propriedades

- $pai[\cdot]$ define uma árvore com raíz s, que se chama **árvore de pesquisa em largura a partir de s**. Os ramos são os pares $(pai[v], v)$ com $pai[v] \neq$ `NULL`.
- O caminho de s até v na árvore é um **caminho mínimo de s até v** no grafo (aqui, *mínimo* significa que tem o menor número de ramos possível).

Visita o grafo $G = (V, A)$ em largura a partir de s

Propriedades

- $pai[\cdot]$ define uma árvore com raíz s, que se chama **árvore de pesquisa em largura a partir de s**. Os ramos são os pares $(pai[v], v)$ com $pai[v] \neq \texttt{NULL}$.
- O caminho de s até v na árvore é um **caminho mínimo de s até v** no grafo (aqui, *mínimo* significa que tem o menor número de ramos possível). Os nós são visitados por ordem crescente de distância a s, nesse sentido.

Visita o grafo $G = (V, A)$ em largura a partir de s

Propriedades

- $pai[\cdot]$ define uma árvore com raíz s, que se chama **árvore de pesquisa em largura a partir de s**. Os ramos são os pares $(pai[v], v)$ com $pai[v] \neq \texttt{NULL}$.
- O caminho de s até v na árvore é um **caminho mínimo de s até v** no grafo (aqui, *mínimo* significa que tem o menor número de ramos possível). Os nós são visitados por ordem crescente de distância a s, nesse sentido.
- Se G for não dirigido, os vértices visitados na chamada $\mathrm{BFS\_V{\small ISIT}}(s, G)$ são os que definem a componente conexa a que s pertence.

Visita o grafo $G = (V, A)$ em largura a partir de s

Propriedades

- $pai[\cdot]$ define uma árvore com raíz s, que se chama **árvore de pesquisa em largura a partir de s**. Os ramos são os pares $(pai[v], v)$ com $pai[v] \neq \texttt{NULL}$.
- O caminho de s até v na árvore é um **caminho mínimo de s até v** no grafo (aqui, *mínimo* significa que tem o menor número de ramos possível). Os nós são visitados por ordem crescente de distância a s, nesse sentido.
- Se G for não dirigido, os vértices visitados na chamada $\text{BFS\_VISIT}(s, G)$ são os que definem a componente conexa a que s pertence.

Complexidade de BFS_Visit(s, G):

Sendo G dado por **listas de adjacências** e as operações MKEMPTYQUEUE, ENQUEUE, QUEUEISEMPTY e DEQUEUE suportadas em $O(1)$:

Visita o grafo $G = (V, A)$ em largura a partir de s

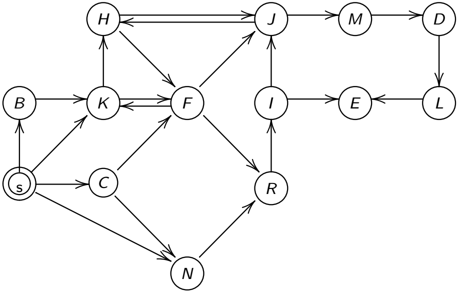
Propriedades

- $pai[\cdot]$ define uma árvore com raíz s, que se chama **árvore de pesquisa em largura a partir de s**. Os ramos são os pares $(pai[v], v)$ com $pai[v] \neq \texttt{NULL}$.
- O caminho de s até v na árvore é um **caminho mínimo de s até v** no grafo (aqui, *mínimo* significa que tem o menor número de ramos possível). Os nós são visitados por ordem crescente de distância a s, nesse sentido.
- Se G for não dirigido, os vértices visitados na chamada $\text{BFS\_VISIT}(s, G)$ são os que definem a componente conexa a que s pertence.

Complexidade de BFS_Visit(s, G):
Sendo G dado por **listas de adjacências** e as operações MKEMPTYQUEUE, ENQUEUE, QUEUEISEMPTY e DEQUEUE suportadas em $O(1)$:

- a complexidade temporal de $\text{BFS\_VISIT}(s, G)$ é $O(|V| + |A|)$;
- a complexidade espacial é $O(|V|)$, se a fila for suportada por um vetor (ou lista ligada com acesso ao primeiro e ao último elemento), além de $\Theta(|V| + |A|)$ para G.

Exemplo de aplicação de BFS a partir de nó s

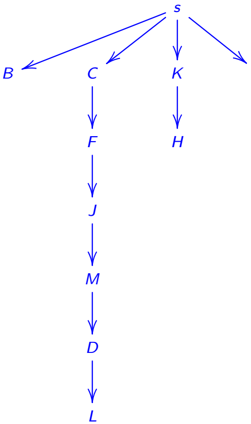


Ordem de saída dos nós da fila na pesquisa em BFS a partir de s:

$$s, B, C, K, N, F, H, R, J, I, M, E, D, L$$

(ordem crescente de distância a s)

Árvore de pesquisa em largura (BFS)



Observação: supusemos que os vizinhos estavam por ordem alfabética, o que, para s, dá B, C, K, N.

Distância mínima do nó s a cada nó (minimizar número de ramos)

BFS_Visit_Distancia(s, G)

```
Para cada v ∈ G.V fazer
    visitado[v] ← false;
    pai[v] ← NULL;
    dist[v] ← ∞;
visitado[s] ← true;
dist[s] ← 0;
Q ← MkEmptyQueue();
Enqueue(s, Q);
Repita
    v ← Dequeue(Q);
    Para cada w ∈ G.Adjs[v] fazer
        Se visitado[w] = false então
            dist[w] ← dist[v] + 1;
            Enqueue(w, Q);
            visitado[w] ← true;
            pai[w] ← v;
até (QueueIsEmpty(Q) = true );
```

Distância mínima do nó s a cada nó (minimizar número de ramos)

BFS_Visit_Distancia(s, G)

Para cada $v \in G.V$ fazer
  $visitado[v] \leftarrow \texttt{false}$;
  $pai[v] \leftarrow \text{NULL}$;
  $dist[v] \leftarrow \infty$;
$visitado[s] \leftarrow \texttt{true}$;
$dist[s] \leftarrow 0$;
$Q \leftarrow \text{MkEmptyQueue}()$;
$\text{Enqueue}(s, Q)$;
Repita
  $v \leftarrow \text{Dequeue}(Q)$;
  Para cada $w \in G.Adjs[v]$ fazer
    Se $visitado[w] = \texttt{false}$ então
      $dist[w] \leftarrow dist[v] + 1$;
      $\text{Enqueue}(w, Q)$;
      $visitado[w] \leftarrow \texttt{true}$;
      $pai[w] \leftarrow v$;
até ($\text{QueueIsEmpty}(Q) = \texttt{true}$);

Propriedades

Se $Q = w_1, w_2, \ldots, w_k$ então
$dist[w_1] \leq dist[w_2] \leq \ldots \leq dist[w_k]$
e $dist[w_k] \leq dist[w_1] + 1$.

Distância mínima do nó s a cada nó (minimizar número de ramos)

```
BFS_Visit_Distancia(s, G)
  Para cada v ∈ G.V fazer
        visitado[v] ← false;
        pai[v] ← NULL;
        dist[v] ← ∞;
  visitado[s] ← true;
  dist[s] ← 0;
  Q ← MkEmptyQueue();
  Enqueue(s, Q);
  Repita
        v ← Dequeue(Q);
        Para cada w ∈ G.Adjs[v] fazer
              Se visitado[w] = false então
                    dist[w] ← dist[v] + 1;
                    Enqueue(w, Q);
                    visitado[w] ← true;
                    pai[w] ← v;
  até (QueueIsEmpty(Q) = true );
```

Propriedades

Se $Q = w_1, w_2, \ldots, w_k$ então $dist[w_1] \leq dist[w_2] \leq \ldots \leq dist[w_k]$ e $dist[w_k] \leq dist[w_1] + 1$.

Se v é acessível de s então, no fim, $dist[v]$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Distância mínima do nó s a cada nó (minimizar número de ramos)

BFS_Visit_Distancia(s, G)

Para cada $v \in G.V$ fazer
    $visitado[v] \leftarrow \texttt{false}$;
    $pai[v] \leftarrow \text{NULL}$;
    $dist[v] \leftarrow \infty$;
$visitado[s] \leftarrow \texttt{true}$;
$dist[s] \leftarrow 0$;
$Q \leftarrow \text{MkEmptyQueue}()$;
$\text{Enqueue}(s, Q)$;
Repita
    $v \leftarrow \text{Dequeue}(Q)$;
    Para cada $w \in G.Adjs[v]$ fazer
        Se $visitado[w] = \texttt{false}$ então
            $dist[w] \leftarrow dist[v] + 1$;
            $\text{Enqueue}(w, Q)$;
            $visitado[w] \leftarrow \texttt{true}$;
            $pai[w] \leftarrow v$;
até ($\text{QueueIsEmpty}(Q) = \texttt{true}$);

Propriedades

Se $Q = w_1, w_2, \ldots, w_k$ então $dist[w_1] \leq dist[w_2] \leq \ldots \leq dist[w_k]$ e $dist[w_k] \leq dist[w_1] + 1$.

Se v é acessível de s então, no fim, $dist[v]$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Os percursos com $|G.V|$ ramos ou mais contêm ciclos (não são mínimos). Podemos substituir ∞ por $|G.V|$.

BFS visita vértices por ordem crescente de distância

Lema:

Se $Q = w_1, w_2, \ldots, w_k$ então $dist[w_1] \leq dist[w_2] \leq \ldots \leq dist[w_k]$ e $dist[w_k] \leq dist[w_1] + 1$. ($w_1$ está à cabeça da fila Q e w_k na cauda)

Prova: Podemos concluir, por indução matemática, que a propriedade é verdadeira em todas as iterações do bloco de "Repita" porque:

- **(i) Base:** No início da iteração 1, $Q = s$. Logo, condição é trivialmente verdade pois $0 = dist[s] \leq dist[s] + 1 = 1$.
- **(ii) Hereditariedade:** Suponhamos, como hipótese de indução, que condição é verdadeira no início da i-ésima iteração, com $i \geq$ fixo. Então, depois de executar o bloco, também é verdade, pois:

  – Se $Q = v, u_1, \ldots, u_t$ então, pela H.I., $dist[u_j] \leq dist[u_{j+1}]$, para $1 \leq j < t$ e $d[u_t] \leq dist[v] + 1$. Os adjacentes w de v que entrarem para Q, ficam no fim, e $dist[w] = dist[v] + 1 \leq dist[u_1] + 1$, pois, pela H.I., $dist[v] \leq dist[w_1]$.

  – Se $Q = v$, então Q ou fica vazia ou têm apenas os w's colocados por v, sendo $dist[w] = dist[v] + 1$ e, portanto, a condição seria trivialmente verdade. $\square$

BFS visita vértices por ordem crescente de distância

Proposição:

Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

BFS visita vértices por ordem crescente de distância

Proposição:

Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Prova (**por indução forte sobre a distância** d):

BFS visita vértices por ordem crescente de distância

Proposição:

Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Prova (**por indução forte sobre a distância** d):

- **Base:** Se $\delta(s, v) = \mathbf{1}$ então $(s, v) \in E$ e $dist[v] = 1$, pois s visita os adjacentes.

BFS visita vértices por ordem crescente de distância

Proposição:

Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Prova (**por indução forte sobre a distância** d):

- **Base:** Se $\delta(s, v) = \mathbf{1}$ então $(s, v) \in E$ e $dist[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $dist[u] = \delta(s, u)$, para todo o u tal que $dist[u] < d$.

BFS visita vértices por ordem crescente de distância

Proposição:

Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Prova (**por indução forte sobre a distância** d):

- **Base:** Se $\delta(s, v) = \mathbf{1}$ então $(s, v) \in E$ e $dist[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $dist[u] = \delta(s, u)$, para todo o u tal que $dist[u] < d$.

  Seja v tal que $dist[v] = d$.

BFS visita vértices por ordem crescente de distância

Proposição:

Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Prova (**por indução forte sobre a distância** d):

- **Base:** Se $\delta(s, v) = \mathbf{1}$ então $(s, v) \in E$ e $dist[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $dist[u] = \delta(s, u)$, para todo o u tal que $dist[u] < d$.

  Seja v tal que $dist[v] = d$. De todos os caminhos mínimos de s para v, tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo.

BFS visita vértices por ordem crescente de distância

Proposição:

Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Prova (**por indução forte sobre a distância d**):

- **Base:** Se $\delta(s, v) = \mathbf{1}$ então $(s, v) \in E$ e $dist[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $dist[u] = \delta(s, u)$, para todo o u tal que $dist[u] < d$.

  Seja v tal que $dist[v] = d$. De todos os caminhos mínimos de s para v, tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \to v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$.

Proposição:

Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Prova (**por indução forte sobre a distância** d):

- **Base:** Se $\delta(s, v) = \mathbf{1}$ então $(s, v) \in E$ e $dist[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $dist[u] = \delta(s, u)$, para todo o u tal que $dist[u] < d$.

  Seja v tal que $dist[v] = d$. De todos os caminhos mínimos de s para v, tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$. Quando v' sai da fila, v está por visitar. Se não, pelo **lema** e hipótese, $dist[v] \leq dist[v'] + 1 = \delta(s, v)$, o que é absurdo pela escolha de v', pois $s \rightsquigarrow pai[v] \rightarrow v$ seria um caminho mínimo também.

BFS visita vértices por ordem crescente de distância

Proposição:

Se v é acessível de s em $G = (V, E)$ então $dist[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v, para todo $v \neq s$.

Prova (**por indução forte sobre a distância** d):

- **Base:** Se $\delta(s, v) = \mathbf{1}$ então $(s, v) \in E$ e $dist[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $dist[u] = \delta(s, u)$, para todo o u tal que $dist[u] < d$.

  Seja v tal que $dist[v] = d$. De todos os caminhos mínimos de s para v, tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \to v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$. Quando v' sai da fila, v está por visitar. Se não, pelo **lema** e hipótese, $dist[v] \leq dist[v'] + 1 = \delta(s, v)$, o que é absurdo pela escolha de v', pois $s \rightsquigarrow pai[v] \to v$ seria um caminho mínimo também.

  Logo, v' visita v e, portanto, $dist[v] = dist[v'] + 1$. Como $dist[v'] = d - 1 < d$, tem-se, pela H.I., $dist[v'] = \delta(s, v')$. Logo, $dist[v] = \delta(s, v') + 1 = \delta(s, v)$. $\square$

Visitar o grafo $G = (V, A)$ em largura

BFS(G)

```
Para cada v ∈ G.V fazer
    visitado[v] ← false;
    pai[v] ← NULL;
Q ← MkEmptyQueue();
Para cada v ∈ G.V fazer
    Se visitado[v] = false então
        BFS_Visit(v, G, Q);
```

BFS_Visit(s, G, Q)

```
visitado[s] ← true;
Enqueue(s, Q);
Repita
    v ← Dequeue(Q);
    Para cada w ∈ G.Adjs[v] fazer
        Se visitado[w] = false então
            Enqueue(w, Q);
            visitado[w] ← true;
            pai[w] ← v;
até (QueueIsEmpty(Q) = true );
```

- Neste código, assume-se que $pai[\cdot]$ e $visitado[\cdot]$ são globais.
- $pai[v]$ identifica o primeiro nó que descobriu v durante a procura.
- o *array* $pai[.]$ define uma **floresta** de árvores pesquisa em largura.

Obter as componentes conexas de um grafo não dirigido

Componente conexa

Uma componente conexa de um **grafo não dirigido** $G = (V, E)$ é um subgrafo $\mathcal{C} = (V_\mathcal{C}, E_\mathcal{C})$ tal que $V_\mathcal{C}$ é um conjunto máximo de nós acessíveis uns dos outros (*máximo* significa aqui que não podemos acrescentar mais nós).

Por definição, u é acessível de v se $u = v$ ou existe um *percurso* de v para u.

- Após a aplicação de **BFS**(G), o *array* $pai[.]$ define uma **floresta** de árvores pesquisa em largura.

Obter as componentes conexas de um grafo não dirigido

Componente conexa

Uma componente conexa de um **grafo não dirigido** $G = (V, E)$ é um subgrafo $\mathcal{C} = (V_\mathcal{C}, E_\mathcal{C})$ tal que $V_\mathcal{C}$ é um conjunto máximo de nós acessíveis uns dos outros (*máximo* significa aqui que não podemos acrescentar mais nós).

Por definição, u é acessível de v se $u = v$ ou existe um *percurso* de v para u.

- Após a aplicação de **BFS**(G), o *array* $pai[.]$ define uma **floresta** de árvores pesquisa em largura.
- Usando $pai[.]$ e análise para trás a partir de v, obtemos a raíz da árvore a que v pertence, que é v se $pai[v] = \texttt{NULL}$. Para grafos **não dirigidos**, os nós dessa árvore definem a componente conexa a que v pertence.
- Se G for **conexo**, a floresta só tem uma árvore (com todos os nós de G).

Obter as componentes conexas de um grafo não dirigido

Proposição:

Se G for um grafo não dirigido, cada árvore da floresta obtida por $\mathrm{BFS}(G)$ identifica uma componente conexa do grafo.

Ideia da prova:

- G pode ser representado por um **grafo dirigido simétrico** G', que designamos por adjunto de G.
- Os nós que constituem a árvore a que w pertence não dependem do nó raíz (a estrutura da árvore pode ser diferente mas os nós são os mesmos).
- Na chamada de $\mathrm{BFS}(v, G, Q)$ no segundo ciclo de $\mathrm{BFS}(G)$, serão visitados todos os nós acessíveis de v em G.
- Como o grafo adjunto de G é simétrico, se algum w acessível de v tivesse sido visitado numa chamada anterior, então também v teria de ter sido marcado como visitado por algum dos descendentes de w.

Que informação se pode extrair do *array pai*?

Exercício 1:

Após a aplicação de $\mathrm{BFS}(G)$ a um dado **grafo não dirigido** G, com nós numerados de 1 a 18, o conteúdo das posições 1 a 18 do *array pai* é

0	0	1	0	1	14	3	0	5	7	4	10	8	11	4	12	12	16

sendo G representado como um grafo **dirigido simétrico**.

1. Os vértices 9 e 17 estão na mesma componente conexa de G?
2. Existe um caminho entre os vértices 6 e 10 em G?
3. Quantas são as componentes conexas de G? Quais são os seus vértices?
4. Indicar um caminho de 18 para 9 no grafo G.
5. Quantos ramos tem o menor caminho entre 1 e 18 em G?
6. Se existir uma aresta entre u e v em G então $pai[u] = v$ ou $pai[v] = u$?

Que informação se pode extrair do *array pai* ?

Exercício 2:

Seja $G = (V, E)$ um não grafo dirigido, com $V = \{1, 2, \ldots, n\}$. Suponha que se aplicou $\mathrm{BFS}(G)$. Seja *pai* o *array* obtido.

Usando pseudocódigo, defina as funções seguintes:

1. $\textsc{SameComponent}(u, v, pai, n)$ que retorna **true** sse u e v estão na mesma componente conexa de G.
2. $\textsc{SimplePath}(u, v, pai, n)$, com $u \neq v$, retorna uma lista de nós que define um caminho de u para v no grafo, se existir, ou **null**, caso contrário. **Caminho** é um percurso que não contém ciclos (i.e., sem nós repetidos).
3. $\textsc{ConnectedComponents}(pai, n, comps)$, altera o conteúdo do *array comps* de modo que $comps[k]$ seja o índice do nó que é a raíz da árvore definida por *pai* para a componente conexa a que k pertence. O algoritmo deve ter complexidade $\Theta(n)$.

Pesquisa em profundidade (DFS) de $G = (V, E)$

Estratégia: pesquisa em profundidade $\Theta(n + m)$, com $n = |V|$ e $m = |E|$

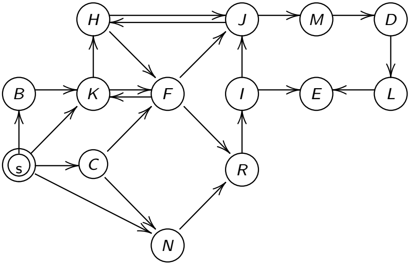
```
DFS(G)          // Depth-First Search
    stack ← Mk_Empty_Stack();
    Para cada v ∈ G.V fazer
        visitado[v] ← false;

    Para cada v ∈ G.V fazer
        Se visitado[v] = false então
            DFS_Visit(v, G, visitado, stack);
    retorna stack;

DFS_Visit(v, G, visitado, stack)
    visitado[v] ← true;
    Para cada w ∈ G.Adjs[v] fazer
        Se visitado[w] = false então
            DFS_Visit(w, G, visitado, stack);
    Push(v,stack);
```

Produz *stack* com os nós ordenados por **tempo de finalização decrescente**.

Exemplo: Visita em profundidade a partir de s



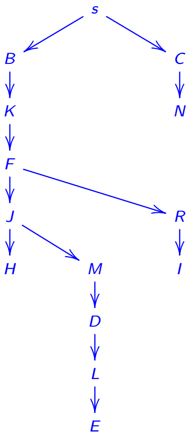
Ordem de descoberta:

$s, B, K, F, J, H, M, D, L, E, R, I, C, N$

Ordem na Stack (topo é s):

$H, E, L, D, M, J, I, R, F, K, B, N, C, s$

Árvore de pesquisa em profundidade (DFS)



Aplicação de DFS para obter ordenação topológica de DAG

Ordenação topológica

Ordenação topológica de um **DAG** $G = (V, A)$ é uma função bijetiva σ de V em $\{0 \ldots, |V| - 1\}$ tal que $\sigma(v) < \sigma(w)$, para todo $(v, w) \in A$. Ou seja, uma ordenação dos nós que é compatível com a relação de precedência definida por G.

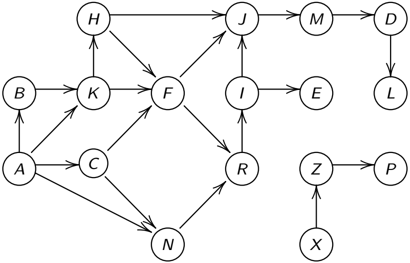
Exemplo de problema de aplicação de ordem topológica:

É necessário definir a ordem de execução de um conjunto de tarefas por uma máquina. São conhecidas algumas restrições de **restrições de precedência** entre tarefas. Não se pode iniciar uma tarefa sem concluir as que a precedem, segundo essa relação de precedência. A máquina só pode estar a realizar uma tarefa em cada instante. Por que ordem devem ser executadas?

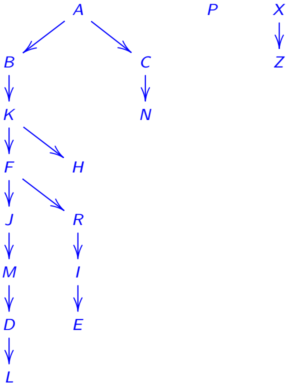
Voltaremos mais à frente a tratar problemas de calendarização de tarefas...

Exemplo: ordenação topológica de DAG por DFS

Visita em profundidade com desempate por ordem alfabética, se necessário.



Floresta (pesquisa DFS)



Ordem de descoberta:

$A, B, K, F, J, M, D, L, R, I, E, H, C, N, P, X, Z$

Ordem na stack (topo é X):

$L, D, M, J, E, I, R, F, H, K, B, N, C, A, P, Z, X$

Ordem topológica (por **pop**'s da stack): $X, Z, P, A, C, N, B, K, H, F, R, I, E, J, M, D, L$

Detetar ciclos num grafo por visita em profundidade

Versão de DFS com tempos de finalização e cores

DFS(G)

$instante \leftarrow 0$;
Para cada $v \in G.V$ fazer $cor[v] \leftarrow$ **branco**; $pai[v] \leftarrow$ NULL;
Para cada $v \in G.V$ fazer
  Se $cor[v] =$ **branco** então DFS_VISIT(v, G);

DFS_VISIT(v, G) // os vértices por visitar estão a **branco**

$instante \leftarrow instante + 1$;
$t\_inicial[v] \leftarrow instante$;
$cor[v] \leftarrow$ **cinzento**; // as cores são úteis para detetar ciclos no grafo G
Para cada $w \in G.Adjs[v]$ fazer
  Se $cor[w] =$ **branco** então // se fosse **cinzento**, G teria um **ciclo**
    $pai[w] \leftarrow v$;
    DFS_VISIT(w, G);
$cor[v] \leftarrow$ **preto**; // visita de v terminou
$instante \leftarrow instante + 1$;
$t\_final[v] \leftarrow instante$; // **tempo de finalização para** v

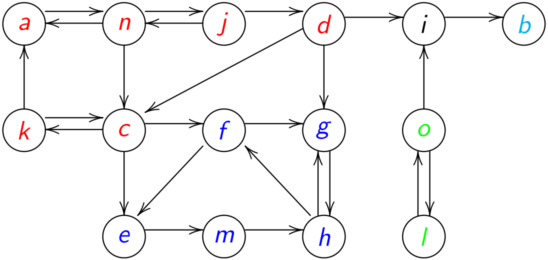
Assume que as variáveis $cor[\cdot]$, $pai[\cdot]$, $t\_inicial[\cdot]$, $t\_final[\cdot]$ e $instante$ são globais.

Componentes fortemente conexas

Que nós acessíveis de s garantem que pode voltar a s, se os visitar?

Componente fortemente conexa

Uma **componente fortemente conexa** (SCC) de um grafo (**dirigido**) $G(V, E)$ é um subgrafo $\mathcal{C} = (V_\mathcal{C}, E_\mathcal{C})$ tal que $V_\mathcal{C}$ é um conjunto *máximo* de nós *acessíveis uns dos outros* em G. Se u e v pertencerem à mesma componente, u é acessível de v e v é acessível de u.



Componentes dadas por:

$\{l, o\}$
$\{a, n, j, k, c, d\}$
$\{i\}$
$\{b\}$
$\{e, f, h, g, m\}$

Algoritmo de Kosaraju-Sharir

Usar $DFS(G)$ para ter pilha S com os nós por ordem decrescente de tempo final
Para $v \in G.V$ fazer $cor[v] \leftarrow$ **branco**;
Enquanto $(S \neq \{\ \})$ fazer
    $v \leftarrow POP(S)$;
    Se $cor[v] =$ **branco** então $\text{DFS\_VISIT}(v, G^T)$ e indica os nós visitados;

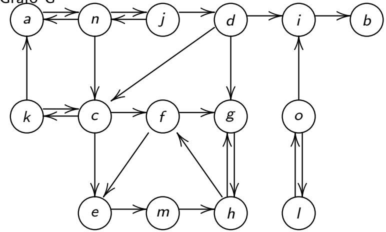
$G^T = (V, A^T)$ denota o **grafo transposto** de $G = (V, A)$, obtém-se de G se se trocar o sentido dos arcos, sendo, $A^T = \{(y, x) \mid (x, y) \in A\}$.

Complexidade temporal do algoritmo de Kosaraju-Sharir

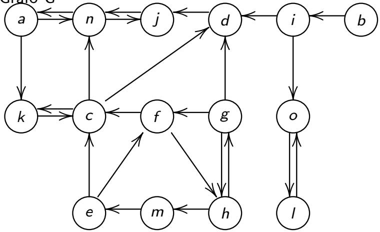
O algoritmo de Kosajaru-Sharir tem complexidade $\Theta(|V| + |A|)$, (ou seja, linear na estrutura do grafo), se o grafo for representado por listas de adjacências.

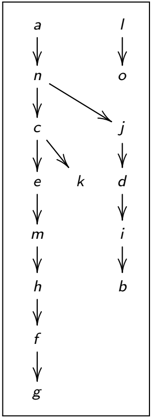
Exemplo

Grafo G



Grafo G^T





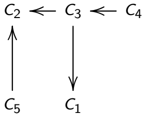
Pilha

l
o
a
n
j
d
i
b
c
k
e
m
h
f
g

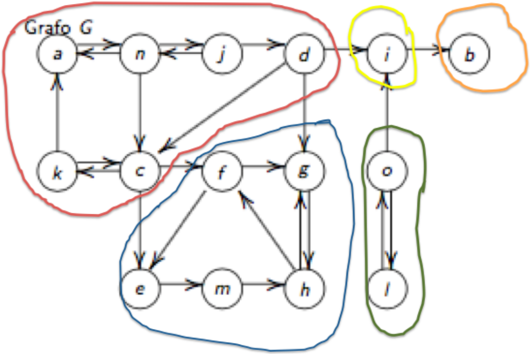
Componentes fortemente conexas

$$
\begin{aligned}
C_1 &= \{l, o\}\\
C_2 &= \{a, n, j, k, c, d\}\\
C_3 &= \{i\}\\
C_4 &= \{b\}\\
C_5 &= \{e, f, h, g, m\}
\end{aligned}
$$

DAG componentes em G^T



DAG das componentes fortemente conexas de G



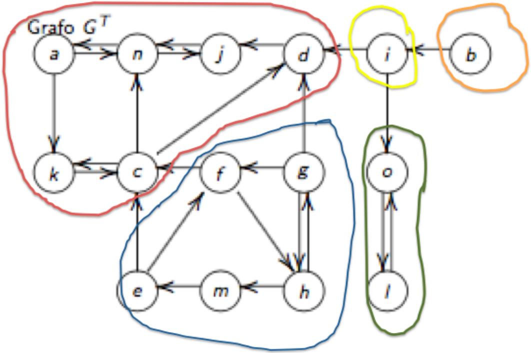
Componentes fortemente conexas de G

$$
\begin{aligned}
C_1 &= \{l, o\}\\
C_2 &= \{a, n, j, k, c, d\}\\
C_3 &= \{i\}\\
C_4 &= \{b\}\\
C_5 &= \{e, f, h, g, m\}
\end{aligned}
$$

DAG componentes em G^T

$$
\begin{tikzcd}
C_2 \arrow[r] \arrow[d] & C_3 \arrow[r] & C_4 \\
C_5 & C_1 \arrow[u] &
\end{tikzcd}
$$

DAG das componentes fortemente conexas de G^T



Pilha

l
o
a
n
j
d
i
b
c
k
e
m
h
f
g

Componentes fortemente conexas de G^T

$$C_1 = \{l, o\}$$
$$C_2 = \{a, n, j, k, c, d\}$$
$$C_3 = \{i\}$$
$$C_4 = \{b\}$$
$$C_5 = \{e, f, h, g, m\}$$

A pilha de **DFS**(G) induz uma visita do DAG das componentes de G^T por ordem inversa da topológica.

DAG componentes em G^T

$$\begin{tikzcd} C_2 & C_3 \arrow[l] \arrow[d] & C_4 \arrow[l] \\ C_5 \arrow[u] & C_1 & \end{tikzcd}$$

Ordem topológica: ordenação dos nós do DAG compatível com a relação de precedência que define.

Prova de Correção do Algoritmo Kosaraju-Sharir

G_{scc} Grafo das componentes fortemente conexas de G

Os nós correspondem às componentes fortemente conexas de G e os ramos são os pares $(\mathcal{C}, \mathcal{C}')$ tais que $\mathcal{C} \neq \mathcal{C}'$ e existem ramos em G de nós de $\mathcal{C}$ para nós de $\mathcal{C}'$.

Justificação da correção do algoritmo de Kosaraju-Sharir:

1. G_{scc} é um grafo dirigido acíclico (DAG).
2. As componentes fortemente conexas de G e G^T têm os mesmos nós.
3. Uma ordenação topológica de G_{scc} corresponde a uma ordenação topológica por ordem inversa (da cronológica) para o DAG das componentes de G^T.
4. Quando visita G^T pela ordem dada por S, as componentes $\mathcal{C}_w$ acessíveis da componente $\mathcal{C}_v$, com $w \neq v$, já estão visitadas quando inicia a visita de v.

Se G_{scc} não fosse acíclico, quaisquer dois nós x e y que estivessem em componentes C_x e C_y (distintas) envolvidas num ciclo seriam acessíveis um do outro em G. Isso é absurdo, pois contradiz a noção de componente fortemente conexa, por qualquer percurso de x para y em G ser um percurso de y para x em G^T (e vice-versa).

As ordens topológicas são inversas pois o DAG de componentes de G^T é o transposto de G_{scc}.

Aplicação de SCC à Resolução de 2-SAT

O problema SAT

Dada uma fórmula F da lógica proposicional em forma conjuntiva normal (CNF), decidir se é satisfazível. Em k-SAT, cada claúsula de F tem k literais.

Exemplos:

- $F_1 = (p \vee q) \wedge (r \vee \neg p \vee q) \wedge (\neg r \vee \neg q)$ é satisfazível?
- $F_2 = (p \vee q) \wedge (\neg p \vee \neg q) \wedge (\neg r \vee p) \wedge (r \vee q) \vee (r \vee \neg p)$ é satisfazível?
- $F_3 = (\neg p \vee q) \wedge (\neg q \vee r) \wedge (s \vee p) \wedge (\neg r \vee \neg q) \wedge (s \vee q)$ é satisfazível?

F_2 e F_3 são exemplos de instâncias de 2-SAT.

CNF: a fórmula F é uma conjunção de claúsulas. **Claúsula** é uma disjunção de literais. **Literal** é uma variável proposicional ou a negação de uma variável.

Aplicação de SCC à Resolução de 2-SAT

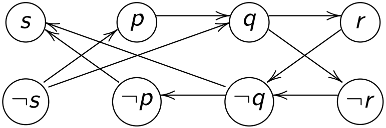
Dada uma instância de **2-SAT**, construir o **grafo de implicações** subjacente:

- os nós são definidos pelas variáveis proposicionais e as suas negações;
- para cada claúsula $u \vee v$, terá os ramos $(\neg u, v)$ e $(\neg v, u)$, com $\neg\neg x = x$.

Teorema (2-SAT resolve-se polinomialmente)

Uma instância F de 2-SAT é satisfazível sse nenhuma componente fortemente conexa do seu grafo de implicações contém uma variável x e a sua negação $\neg x$.

$F_3 = (\neg p \vee q) \wedge (\neg q \vee r) \wedge (s \vee p) \wedge (\neg r \vee \neg q) \vee (s \vee q)$ é satisfazível? **Sim.**



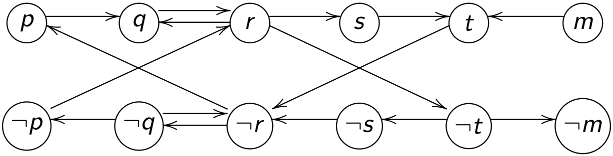
Cada componente fortemente conexa deste grafo só tem um nó.

A estudar mais à frente na uc de DAA:

k-SAT, para $k \geq 3$ é um problema NP-completo. A menos que P=NP, não pode ser resolvido polinomialmente.

Aplicação de SCC à Resolução de 2-SAT

$$F_4 = (\neg p \vee q) \wedge (\neg q \vee r) \wedge (s \vee \neg r) \wedge (t \vee \neg s) \vee (\neg r \vee q) \vee (\neg m \vee t) \wedge (\neg r \vee \neg t) \wedge (p \vee r)$$



F_4 não é satisfazível.

O grafo de implicações tem três componentes fortemente conexas, que são definidas por: $\{p, q, r, s, t, \neg p, \neg q, \neg r, \neg s, \neg t\}$, $\{m\}$ e $\{\neg m\}$.

Como, por exemplo, **p e $\neg p$ estão na mesma componente**, então F_4 não é satisfazível.

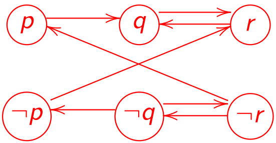
Notar que ter percurso no grafo de implicações do nó p para o nó $\neg p$ e do nó $\neg p$ para o nó p significa que $(p \Rightarrow \neg p) \wedge (\neg p \Rightarrow p)$, o que não é satisfazivel!

Como obter uma solução para instância de 2SAT?

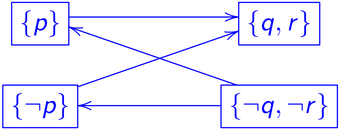
Seja F uma instância de 2-SAT satisfazível. Seja σ uma ordenação topológica do DAG das componentes fortemente conexas do grafo de implicações de F. Sejam $\mathcal{C}(x)$ e $\mathcal{C}(\neg x)$ as componentes de x e de $\neg x$. **Se $\sigma(\mathcal{C}(x)) < \sigma(\mathcal{C}(\neg x))$, atribuir a x o valor 0 (Falso). Senão, atribuir a x o valor 1 (Verdade).** O valor de F para esta valoração será 1.

Exemplo: $F_5 = (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r \vee q) \vee (p \vee r)$ é satisfazível.

Grafo de implicações para F_5:



DAG de componentes



O DAG das componentes fortemente conexas admite duas ordens topológicas:

$$\{\neg q, \neg r\}, \{\neg p\}, \{p\}, \{q, r\} \quad \rightsquigarrow \quad q = r = 1, p = 1$$
$$\{\neg q, \neg r\}, \{p\}, \{\neg p\}, \{q, r\} \quad \rightsquigarrow \quad q = r = 1, p = 0$$

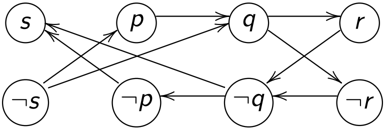
Como obter uma solução para instância de 2SAT?

Recordar. . .

Se $\sigma(\mathcal{C}(x)) < \sigma(\mathcal{C}(\neg x))$**, atribuir a** x **o valor** 0 **(Falso). Senão, atribuir a** x **o valor** 1 **(Verdade).** O valor de F para esta valoração será 1.

Exemplo: $F_3 = (\neg p \vee q) \wedge (\neg q \vee r) \wedge (s \vee p) \wedge (\neg r \vee \neg q) \vee (s \vee q)$ é satisfazível.

Nesta instância, o DAG das componentes fortemente conexas é semelhante ao grafo de implicações. Todas têm todas apenas um nó.



Duas ordens topológicas também:

$$\begin{array}{lcl}
\neg s, p, q, r, \neg r, \neg q, \neg p, s & \rightsquigarrow & s = 1, p = 0, q = 0, r = 0 \\
\neg s, p, q, \neg r, r, \neg q, \neg p, s & \rightsquigarrow & s = 1, p = 0, q = 0, r = 1
\end{array}$$