Algoritmos em Grafos

Ana Paula Tomás

LEIC - Desenho de Algoritmos
Universidade do Porto

Março 2022

Tópicos a abordar

Grafos - modelo fundamental em computação

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 $(v_i, v_j) \in E$.

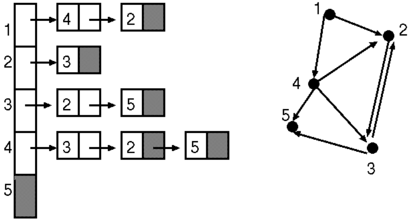
- **Listas de adjacências:** a cada nó v associa a lista de seus adjacentes, i.e., a lista de nós w tais que $(v, w) \in E$. $\Leftarrow$ assumiremos esta representação

Representações para grafos

Seja $G = (V, E)$ um grafo dirigido, com $V = \{v_1, v_2, \ldots, v_n\}$, 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



- 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** $\Theta(|Adj(v_i)|)$.

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 ∈ G.V fazer
    visitado[v] ← false;
    pai[v] ← NULL;
visitado[s] ← true;
Q ← MkEmptyQueue();
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;        // v precede w no caminho de s para w
até (QueueIsEmpty(Q) = 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 $pai[v] \neq \texttt{NULL}$.
- 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).

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.

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 $\text{BFS\_V}{\small\text{ISIT}}(s, G)$ 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 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

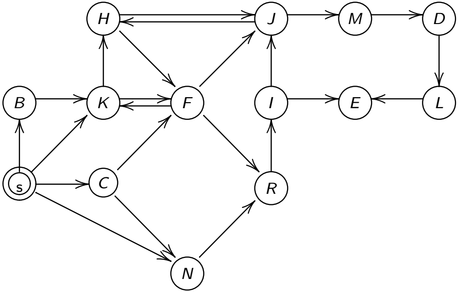
- $pai[\cdot]$ define uma árvore com raíz s, que se chama **árvore de pesquisa em largura a partir de s**. Os ramos são os pares $(pai[v], v)$ com $pai[v] \neq \texttt{NULL}$.
- O caminho de s até v na árvore é um **caminho mínimo de s até v** no grafo (aqui, *mínimo* significa que tem o menor número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

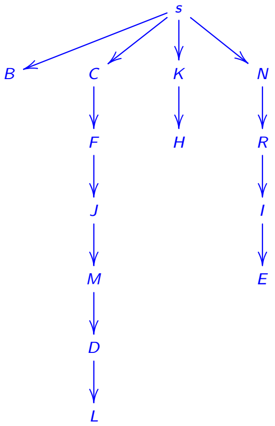


Ordem de saída dos nós da fila na pesquisa em BFS a partir de s:

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

(ordem crescente de distância a s)

Árvore de pesquisa em largura (BFS)



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 ∈ 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 $dist[w_k] \leq dist[w_1] + 1$.

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

BFS_Visit_Distancia(s, G)

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

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 $v \neq s$.

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

BFS(G)

Para cada $v \in G.V$ fazer
    visitado[v] $\leftarrow$ `false`;
    pai[v] $\leftarrow$ NULL;
$Q \leftarrow$ MkEmptyQueue();
Para cada $v \in G.V$ fazer
    Se *visitado*[v] $=$ `false` então
        BFS_Visit(v, G, Q);

BFS_Visit(s, G, Q)

visitado[s] $\leftarrow$ `true`;
Enqueue(s, Q);
Repita
    $v \leftarrow$ Dequeue(Q);
    Para cada $w \in G.Adjs[v]$ fazer
        Se *visitado*[w] $=$ `false` então
            Enqueue(w, Q);
            visitado[w] $\leftarrow$ `true`;
            pai[w] $\leftarrow v$;
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.

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

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

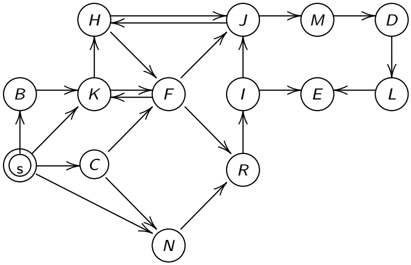
DFS(G) // **Depth-First Search**

- *stack* $\leftarrow$ MK_EMPTY_STACK();
- Para cada $v \in G.V$ fazer
  - *visitado*[v] $\leftarrow$ **false**;
- Para cada $v \in G.V$ fazer
  - Se *visitado*[v] $=$ **false** então
    - DFS_VISIT($v, G, visitado, stack$);
- return *stack*;

DFS_VISIT($v, G, visitado, stack$)

- *visitado*[v] $\leftarrow$ **true**;
- Para cada $w \in G.Adjs[v]$ 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**.



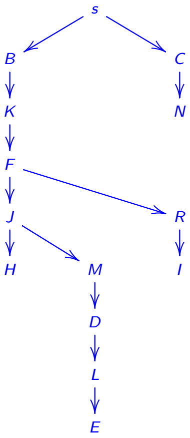
Ordem de descoberta:

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

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)

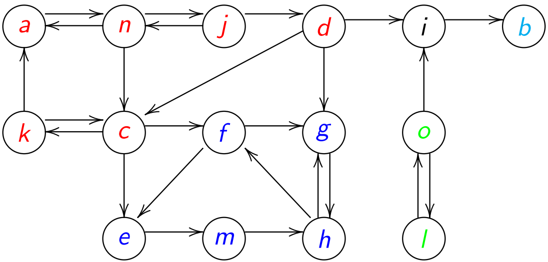


Componentes fortemente conexas

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;

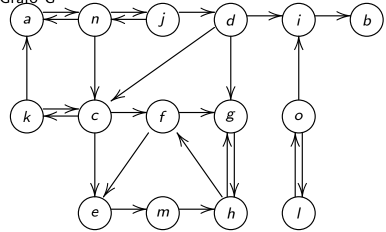
$G^T = (V, A^T)$ denota o **grafo transposto** de $G = (V, A)$, obtém-se de G se se trocar o sentido dos arcos, sendo, $A^T = \{(y, x) \mid (x, y) \in A\}$.

Complexidade temporal do algoritmo de Kosaraju-Sharir

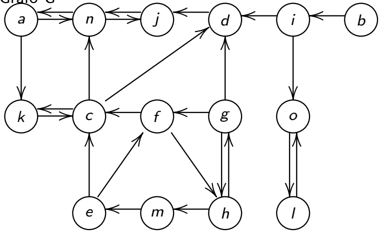
O algoritmo de Kosajaru-Sharir tem complexidade $\Theta(|V| + |A|)$, (ou seja, linear na estrutura do grafo), se o grafo for representado por listas de adjacências.

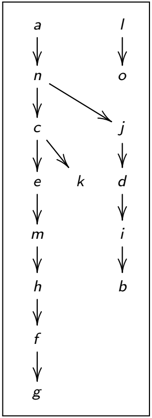
Exemplo

Grafo G



Grafo G^T





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

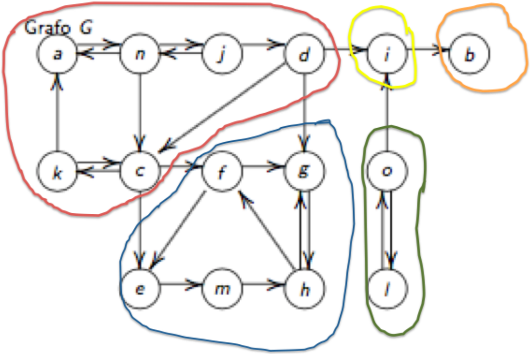
Componentes fortemente conexas

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

DAG componentes em G^T

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

DAG das componentes fortemente conexas de G



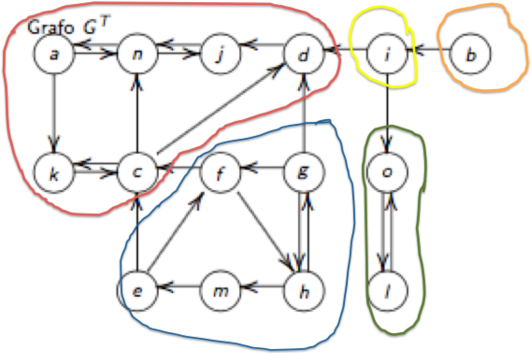
Componentes fortemente conexas de G

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

DAG componentes em G^T

$$
\begin{array}{ccccc}
C_2 & \rightarrow & C_3 & \rightarrow & C_4 \\
\downarrow & & \uparrow & & \\
C_5 & & C_1 & &
\end{array}
$$

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

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

DAG componentes em G^T

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

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

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

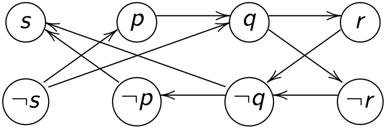
Dada uma instância de **2-SAT**, construir o **grafo de implicações** subjacente:

- 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.**



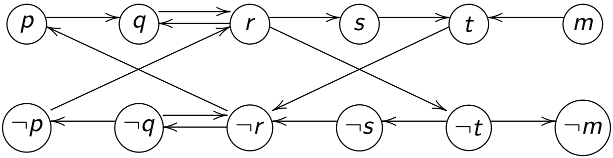
Cada componente fortemente conexa deste grafo só tem um nó.

A estudar mais à frente na uc de Desenho de Algoritmos:

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.

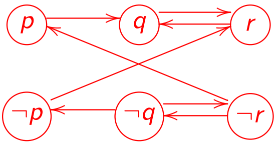
O grafo de implicações tem três componentes fortemente conexas, que são definidas por: $\{p, q, r, s, t, \neg p, \neg q, \neg r, \neg s, \neg t\}$, $\{m\}$ e $\{\neg m\}$.
Como, por exemplo, **p e $\neg p$ estão na mesma componente**, então F_4 não é satisfazível.
Notar que ter percurso no grafo de implicações 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?

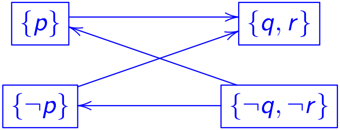
Seja F uma instância de 2-SAT satisfazível. Seja σ uma ordenação topológica do DAG das componentes fortemente conexas do grafo de implicações de F. Sejam $\mathcal{C}(x)$ e $\mathcal{C}(\neg x)$ as componentes de x e de $\neg x$. **Se $\sigma(\mathcal{C}(x)) < \sigma(\mathcal{C}(\neg x))$, atribuir a 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:

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

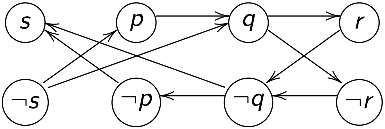
Como obter uma solução para instância de 2SAT?

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.

O DAG das componentes fortemente conexas (que têm todas apenas um nó) é semelhante ao grafo de implicações.



Duas ordens topológicas também:

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

Planeamento de tarefas

Exemplo 3: Scheduling

Um projeto é constituído por um conjunto de tarefas, sendo conhecida a **duração** de cada tarefa e as **restrições de precedência** entre tarefas. Não se pode dar início a uma tarefa sem que as que a precedem estejam concluídas. Pretendemos agendar as tarefas de modo a concluir o projeto o mais cedo possível. Cada tarefa requer um certo número de **trabalhadores**. Cada trabalhador só pode estar a realizar uma tarefa em cada instante. Assuma que:

- Caso 1: não há restrições quanto ao número de trabalhadores a contratar.
- Caso 2: há restrições quanto ao número de trabalhadores a contratar.

Admita que as habilitações necessárias são idênticas para todas as tarefas.

Caso 1: sem partilha de recursos

Caso 2: com partilha de recursos

Exemplo: Tarefas A, B, C, D e E; A precede C; B precede C e D; C precede E.

	A	B	C	D	E
duração	1	3	4	5	2
# trabalhadores	2	3	1	1	2

Exemplo 3 - Caso 1

Dada a descrição das tarefas do projeto, as suas durações d_i com $i \in$ *Tarefas*, e a relação de precedência $\mathcal{R}$, determinar a data de conclusão mais próxima para o projeto e uma data para início de cada tarefa.

Variáveis de decisão:

- z: data de conclusão do projeto
- x_i: data de início da tarefa i, para $i \in$ Tarefas

Modelo de otimização linear:

minimizar z

sujeito a

$$\begin{cases} x_i + d_i \leq x_j, & \text{para todo } (i,j) \in \mathcal{R} \\ x_i + d_i \leq z, & \text{para todo } i \in \text{Tarefas} \\ z \in \mathbb{R}_0^+, \;\; x_i \in \mathbb{R}_0^+, & \text{para todo } i \in \text{Tarefas} \end{cases}$$

Planeamento de tarefas

Problema de escalonamento de tarefas

Um projeto é constituído por um conjunto de tarefas, sendo conhecida a duração de cada tarefa e as restrições de precedência entre tarefas. Não se pode dar início a uma tarefa sem concluir as que a precedem. Pretende-se agendar as tarefas de modo a concluir o projeto o mais cedo possível.

- Cenário 1: Há apenas uma pessoa para realizar o projeto e não pode realizar várias tarefas ao mesmo tempo.

  Problema: Determinar um plano, i.e., uma **ordenação das tarefas compatível com as precedências** definidas.

- Cenário 2: As tarefas não partilham recursos. Podem ser realizadas várias tarefas simultaneamente, devendo satisfazer as precedências definidas.

  Problema: Determinar quando é que o projeto pode ficar concluído:

  (i) se todas as tarefas tiverem **duração unitária**;
  (ii) se as tarefas puderem ter **durações distintas**.

Redes Nó-Atividade e Arco-Atividade

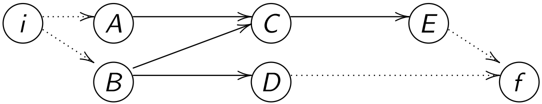
Exemplo: Tarefas A, B, C, D e E; A precede C, B precede C e D, e C precede E.

- **Nó-atividade:** os nós representam atividades e os ramos precedências.

Redes Nó-Atividade e Arco-Atividade

Exemplo: Tarefas A, B, C, D e E; A precede C, B precede C e D, e C precede E.

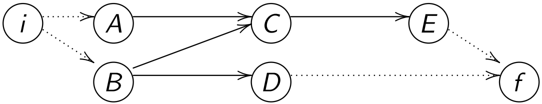
- **Nó-atividade:** os nós representam atividades e os ramos precedências.



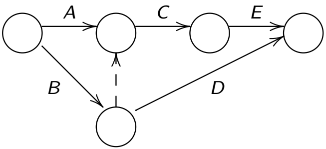
Redes Nó-Atividade e Arco-Atividade

Exemplo: Tarefas A, B, C, D e E; A precede C, B precede C e D, e C precede E.

- **Nó-atividade:** os nós representam atividades e os ramos precedências.



- **Arco-atividade:** os ramos representam atividades e os nós acontecimentos.



Para não criar precedências novas, foi necessário introduzir uma **atividade fictícia** (a tracejado no grafo). Define-se com **duração zero**

Planeamento de tarefas – Modelo Nó-Atividade

A **relação de precedência** é dada por um DAG (grafo dirigido acíclico). Os nós do grafo definem as tarefas e um ramo (x, y) representa o facto de a tarefa x preceder a tarefa y. Se tem os ramos (x, y) e (y, z), também x precede z mesmo que não tenha o ramo (x, z).

- Cenário 1: **Nenhum par de tarefas decorre simultaneamente.**
  Problema: corresponde à ordenação topológica dos nós de um DAG

- Cenário 2: **Algumas tarefas podem decorrer em simultâneo.**
  Calcular a duração mínima do projeto se:
  - (i) todas as tarefas têm **duração unitária**;
    Problema: Encontrar o caminho mais longo num DAG, sendo o comprimento é dado pelo *número de ramos* no caminho.
  - (ii) as tarefas podem ter **durações distintas**.
    Problema: Encontrar o caminho mais longo num DAG, sendo o comprimento dado pela *soma dos valores nos nós* do caminho.

Ordenação topológica dos nós de um 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.

TopSortDAG(G)

Para todo $v \in G.V$ fazer $GrauE[v] \leftarrow 0$;
Para todo $(v, w) \in G.A$ fazer $GrauE[w] \leftarrow GrauE[w] + 1$;
$S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$; /* S deve ser suportado por uma fila ou uma pilha. */
$i \leftarrow 0$;
Enquanto $(S \neq \emptyset)$ fazer
    $v \leftarrow$ um qualquer elemento de S; $S \leftarrow S \setminus \{v\}$;
    $\sigma[v] \leftarrow i$; $i \leftarrow i + 1$;
    Para todo $w \in G.Adjs[v]$ fazer
        $GrauE[w] \leftarrow GrauE[w] - 1$;
        Se $GrauE[w] = 0$ então $S \leftarrow S \cup \{w\}$;

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.

TopSortDAG(G)

Para todo $v \in G.V$ fazer $GrauE[v] \leftarrow 0$;
Para todo $(v, w) \in G.A$ fazer $GrauE[w] \leftarrow GrauE[w] + 1$;
$S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$; /* S deve ser suportado por uma fila ou uma pilha. */
$i \leftarrow 0$;
Enquanto $(S \neq \emptyset)$ fazer
    $v \leftarrow$ um qualquer elemento de S; $S \leftarrow S \setminus \{v\}$;
    $\sigma[v] \leftarrow i$; $i \leftarrow i + 1$;
    Para todo $w \in G.Adjs[v]$ fazer
        $GrauE[w] \leftarrow GrauE[w] - 1$;
        Se $GrauE[w] = 0$ então $S \leftarrow S \cup \{w\}$;

A correção do algoritmo resulta de qualquer DAG ter algum nó com **grau de entrada zero** e de quando se retira um ramo a um DAG, obtém-se um DAG.

Observar também que: se G não for um DAG, o ciclo "Enquanto" termina sempre mas com $i < n$ em vez de $i = n$.

Ordenação topológica de um DAG por DFS (retorna stack)

TopSort_DFS(G)

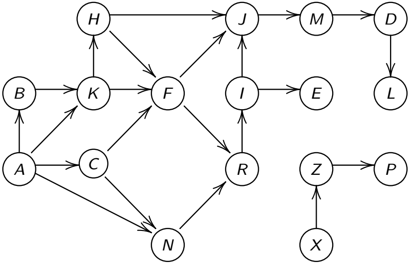
$S \leftarrow \{\}$; // definir stack vazia
Para cada $v \in G.V$ fazer $visitado[v] \leftarrow$ **false**;
Para cada $v \in G.V$ fazer
  Se $visitado[v] =$ **false** então $\textsc{TopSort\_DFSVisit}(v, G, visitado, S)$;
retorna S; // ordem topológica se efetuar POPs sucessivos de S

TopSort_DFSVisit($v, G, visitado, S$)

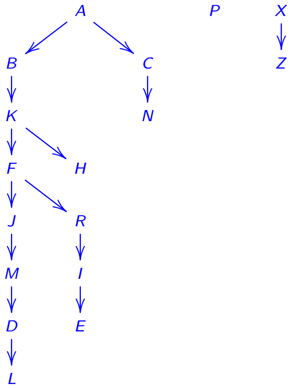
$visitado[v] \leftarrow$ **true**;
Para cada $w \in G.Adjs[v]$ fazer
  Se $visitado[w] =$ **false** então
    $\textsc{TopSort\_DFSVisit}(w, G, visitado, S)$;
$\textsc{Push}(v, S)$;

Visita em profundidade para determinar ordem topológica

Assumir que, se for necessário escolher adjacente, segue **ordem lexicográfica**.



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):

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

Caminho máximo num DAG (distâncias unitárias)

Problema: obter um **caminho máximo** num grafo dirigido acíclico $G = (V, A)$, sendo o comprimento dado pelo número de ramos do caminho.

MaxPathDAG(G)

Para todo $v \in G.V$ fazer $ES[v] \leftarrow 0$; $Pai[v] \leftarrow \texttt{NULL}$; $GrauE[v] \leftarrow 0$;
Para todo $(v, w) \in G.A$ fazer $GrauE[w] \leftarrow GrauE[w] + 1$;
$S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$; /* S deve ser suportado por uma fila ou uma pilha. */
$Max \leftarrow -1$; $v_f \leftarrow \texttt{NULL}$; // v_f **indica o vértice final no caminho mais longo obtido**
Enquanto $(S \neq \emptyset)$ fazer
    $v \leftarrow$ um qualquer elemento de S;
    $S \leftarrow S \setminus \{v\}$;
    Se $Max < ES[v]$ então $Max \leftarrow ES[v]$; $v_f \leftarrow v$; /* $ES[v]$ o número de ramos do caminho */
    Para todo $w \in G.Adjs[v]$ fazer
        Se $ES[w] < ES[v] + 1$ então
            $ES[w] \leftarrow ES[v] + 1$; $Pai[w] \leftarrow v$;
        $GrauE[w] \leftarrow GrauE[w] - 1$;
        Se $GrauE[w] = 0$ então $S \leftarrow S \cup \{w\}$;
$\text{ESCREVECAMINHO}(v_f, Pai)$; escrever($Max$);

Dado $G = (V, A, D)$ em que $D(v) \in \mathbb{R}_0^+$ é a duração da tarefa $v \in V$, determinar $ES[v]$, o instante mais próximo em que pode dar início a v **("earliest start")**.

MaxPathWeightedDAG(G)

Para todo $v \in G.V$ fazer $ES[v] \leftarrow 0$; $Pai[v] \leftarrow \texttt{Nenhum}$; $GrauE[v] \leftarrow 0$;
Para todo $(v, w) \in G.A$ fazer $GrauE[w] \leftarrow GrauE[w] + 1$;
$S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$; /* S deve ser suportado por uma fila ou uma pilha. */
$Max \leftarrow -1$; $v_f \leftarrow \texttt{NULL}$; /* $ES[v]$ o número de ramos do caminho */
Enquanto $(S \neq \emptyset)$ fazer
    $v \leftarrow$ um qualquer elemento de S; $S \leftarrow S \setminus \{v\}$;
    Se $Max < ES[v] + D[v]$ então $Max \leftarrow ES[v] + D[v]$; $v_f \leftarrow v$;
    Para todo $w \in G.Adjs[v]$ fazer
        Se $ES[w] < ES[v] + D[v]$ então
            $ES[w] \leftarrow ES[v] + D[v]$; $Pai[w] \leftarrow v$;
        $GrauE[w] \leftarrow GrauE[w] - 1$;
        Se $GrauE[w] = 0$ então $S \leftarrow S \cup \{w\}$;
EscreveCaminho(v_f, Pai); escrever(Max);

Modelo Nó-Atividade

Os nós do grafo denotam atividades e os arcos definem as precedências. As durações ficam associadas aos nós: d_i denota a duração da atividade i.

Objetivo: $\boxed{\text{Determinar a duração mínima do projeto}}$

Assumindo que se vai concluir o projeto o mais cedo possível, define-se:

ES_i *earliest start time*

data de início mais próxima para a tarefa i

$ES_i = \max_{(j,i)\in A}\{\ EF_j\ \}$

LF_i *latest finish time*

data de conclusão mais afastada para a tarefa i

menor das datas de início mais afastadas para as tarefas que seguem i

$LF_i = \min_{(i,j)\in A}\{\ LS_j\ \}$

EF_i *earliest finish time*

data de conclusão mais próxima para a tarefa i

$EF_i = ES_i + d_i$

LS_i *latest start time*

data de início mais afastada para a tarefa i

$LS_i = LF_i - d_i$

Modelo Arco-Atividade

Os nós do grafo representam acontecimentos (início ou fim de um conjunto de atividades) e os arcos representam as atividades: a atividade (i, j) tem i como acontecimento inicial e j como acontecimento final. Seja d_{ij} a sua duração.

Objetivo: Determinar a duração mínima do projeto.

- Sejam 1 e n os acontecimentos início e fim do projeto. Então,
  - $ES_{1j} = 0$, para as atividades com início no nó 1;
  - $LF_{jn} =$ duração mínima do projeto, para as atividades com fim no nó n.

Modelo Arco-Atividade

Os nós do grafo representam acontecimentos (início ou fim de um conjunto de atividades) e os arcos representam as atividades: a atividade (i,j) tem i como acontecimento inicial e j como acontecimento final. Seja d_{ij} a sua duração.

Objetivo: Determinar a duração mínima do projeto.

- Sejam 1 e n os acontecimentos início e fim do projeto. Então,
  - $ES_{1j} = 0$, para as atividades com início no nó 1;
  - $LF_{jn} =$ duração mínima do projeto, para as atividades com fim no nó n.
- **Pode começar (i,j) logo que todas as atividades que a precedem estiverem concluídas.**

$$ES_{ij} = \max\{ EF_{ki} \mid (k,i) \in A\} \qquad\qquad EF_{ij} = ES_{ij} + d_{ij}$$

- **Para não atrasar a realização do projeto, (i,j) tem de estar concluída quando alguma das atividades que a segue não puder ser mais adiada.**

$$LF_{ij} = \min\{ LS_{jk} \mid (j,k) \in A\} \qquad\qquad LS_{ij} = LF_{ij} - d_{ij}$$

Atividades Críticas e Folgas

Dois tipos de folgas

- Folga Total FT_{ij}: diferença entre a data de início mais afastada e a data de início mais próxima para a atividade (i, j).

$$FT_{ij} = LS_{ij} - ES_{ij} = LF_{ij} - EF_{ij} = LF_{ij} - ES_{ij} - d_{ij}$$

Atividades Críticas e Folgas

Dois tipos de folgas

- Folga Total FT_{ij}: diferença entre a data de início mais afastada e a data de início mais próxima para a atividade (i, j).

$$\boxed{FT_{ij} = LS_{ij} - ES_{ij}} = LF_{ij} - EF_{ij} = LF_{ij} - ES_{ij} - d_{ij}$$

- Folga Livre FL_{ij}: folga que não impede que as atividades que seguem (i, j) possam começar na sua data de início mais próxima.

$$FL_{ij} = min\{ES_{jk} \mid (j, k) \in A\} - EF_{ij}$$

Dois tipos de folgas

- Folga Total FT_{ij}: diferença entre a data de início mais afastada e a data de início mais próxima para a atividade (i, j).

$$FT_{ij} = LS_{ij} - ES_{ij} = LF_{ij} - EF_{ij} = LF_{ij} - ES_{ij} - d_{ij}$$

- Folga Livre FL_{ij}: folga que não impede que as atividades que seguem (i, j) possam começar na sua data de início mais próxima.

$$FL_{ij} = min\{ES_{jk} \mid (j, k) \in A\} - EF_{ij}$$

Propriedade: A folga livre é sempre menor ou igual que a folga total.

Atividades Críticas e Folgas

Dois tipos de folgas

- Folga Total FT_{ij}: diferença entre a data de início mais afastada e a data de início mais próxima para a atividade (i, j).

$$FT_{ij} = LS_{ij} - ES_{ij} = LF_{ij} - EF_{ij} = LF_{ij} - ES_{ij} - d_{ij}$$

- Folga Livre FL_{ij}: folga que não impede que as atividades que seguem (i, j) possam começar na sua data de início mais próxima.

$$FL_{ij} = min\{ES_{jk} \mid (j, k) \in A\} - EF_{ij}$$

Propriedade: A folga livre é sempre menor ou igual que a folga total.

(i, j) é uma atividade crítica sse $ES_{ij} = LS_{ij}$, ou seja, tem folga total nula.

Observação: Noções idênticas para o modelo nó-atividade, com adaptações.

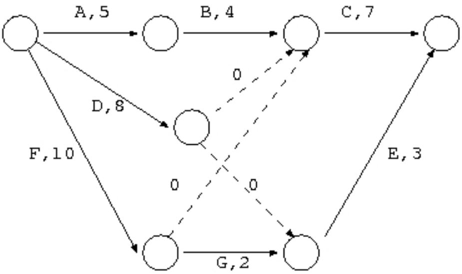
Exemplo 1

Qual é a duração mínima do projeto seguinte?

atividade	*duração (em dias)*	*precede*
A	5	B
B	4	C
C	7	-
D	8	C, E
E	3	-
F	10	C, G
G	2	E

Resposta: 17 dias

Modelo arco-atividade



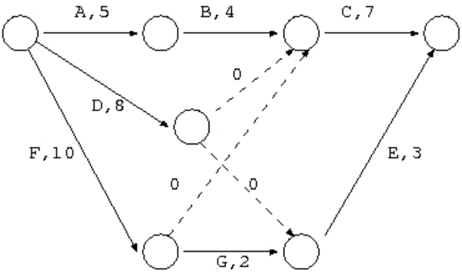
Exemplo 1

Qual é a duração mínima do projeto seguinte?

atividade	*duração (em dias)*	*precede*
A	5	B
B	4	C
C	7	-
D	8	C, E
E	3	-
F	10	C, G
G	2	E

Resposta: 17 dias

Modelo arco-atividade



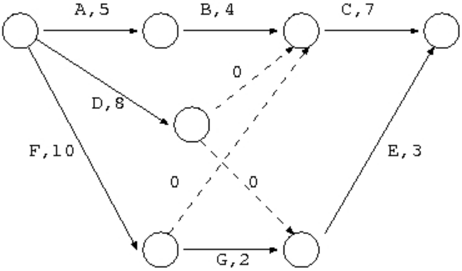
Exemplo 1

Qual é a duração mínima do projeto seguinte?

atividade	*duração (em dias)*	*precede*
A	5	B
B	4	C
C	7	-
D	8	C, E
E	3	-
F	10	C, G
G	2	E

Resposta: 17 dias

Modelo arco-atividade



A **duração mínima** do projeto é igual ao *comprimento do caminho máximo*. No modelo arco-atividade, o *comprimento de um caminho* é a soma dos valores nos seus arcos. No modelo nó-atividade, é a soma dos valores nos seus nós. Em ambos os casos, esses valores representam as durações das atividades.

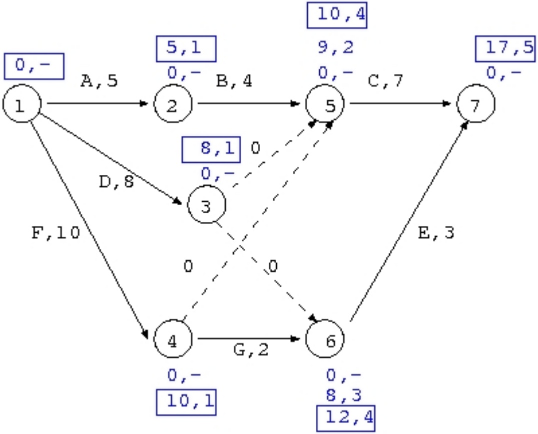
Método do Caminho Crítico (arco-atividade) – *earliest start*

Dado $G = (V, A, D)$, em que $D((i,j)) \in \mathbb{R}_0^+$ é a duração da tarefa $(i,j) \in A$, determinar a duração mínima do projeto e a data de início mais próxima para cada (i,j).

Para todo $v \in G.V$ fazer $ES[v] \leftarrow 0$; $Prec[v] \leftarrow \texttt{Nenhum}$; $GrauE[v] \leftarrow 0$;
Para todo $(v,w) \in G.A$ fazer $GrauE[w] \leftarrow GrauE[w] + 1$;
$S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$; /* S deve ser suportado por uma fila ou uma pilha. */
$DurMin \leftarrow -1$; $v_f \leftarrow \texttt{Nenhum}$;
Enquanto ($S \neq \emptyset$) fazer
    $v \leftarrow$ um qualquer elemento de S; $S \leftarrow S \setminus \{v\}$;
    Se $DurMin < ES[v]$ então $DurMin \leftarrow ES[v]$; $v_f \leftarrow v$;
    Para todo $w \in G.Adjs[v]$ fazer
        Se $ES[w] < ES[v] + D[(v,w)]$ então
            $ES[w] \leftarrow ES[v] + D[(v,w)]$; $Prec[w] \leftarrow v$;
        $GrauE[w] \leftarrow GrauE[w] - 1$;
        Se $GrauE[w] = 0$ então $S \leftarrow S \cup \{w\}$;
$\textsc{escreveCaminho}(v_f, Prec)$; escrever($DurMin$);

O valor $ES[v]$ calculado pelo algoritmo é a data de início mais próxima **para as tarefas com início no nó** v, pelo que $ES_{ij} = ES[i]$, para cada $(i,j) \in A$.

Exemplo



$\boxed{\text{ES[v],Prec[v]}}$ indica o comprimento do caminho mais longo desde a origem até v e o nó que antecede v no caminho encontrado. As outras anotações nos nós são os valores em passos intermédios. $ES[v]$ é a data mais próxima para as tarefas com início em v.

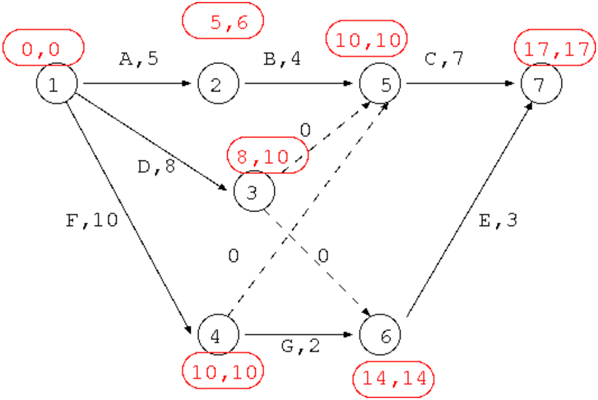
Os nós estão numerados segundo a ordem de visita pelo algoritmo (que é uma ordem topológica).

Obter a data de conclusão mais afastada para as tarefas com fim no nó v, para todo v, por **análise para trás** (*backward analysis*), fixando $LF[v]$ inicialmente como $DurMin$ (a duração mínima do projeto).

Para todo $v \in G.V$ fazer $LF[v] \leftarrow DurMin$; $GrauS[v] \leftarrow 0$;
Para todo $(v, w) \in G.A$ fazer
    $GrauS[v] \leftarrow GrauS[v] + 1$;   // grau de saída
$G^T \leftarrow$ grafo transposto de G;
$S \leftarrow \{v \mid GrauS[v] = 0\}$;
Enquanto $(S \neq \emptyset)$ fazer
    $v \leftarrow$ um qualquer elemento de S;
    $S \leftarrow S \setminus \{v\}$;
    Para todo $w \in G^T.Adjs[v]$ fazer
        Se $LF[w] > LF[v] - D[(w, v)]$ então   // (w, v) em $G.A$
            $LF[w] \leftarrow LF[v] - D[(w, v)]$;
        $GrauS[w] \leftarrow GrauS[w] - 1$;
        Se $GrauS[w] = 0$ então $S \leftarrow S \cup \{w\}$;

$LF_{ij} = LF[j]$, para todo (i, j).

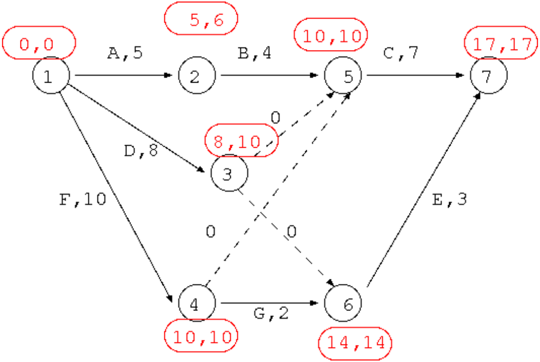
Exemplo (cont.)



As anotações representam $\boxed{ES[v], LF[v]}$, sendo $LF[v]$ a data de conclusão mais afastada para as tarefas que têm fim no nó v e $ES[v]$ a data de início mais próxima para as tarefas com início em v.

Exemplo (cont.)

As anotações representam $\boxed{ES[v], LF[v]}$.



	ES	LF	LS	FT	FL
A	0	6	1	1	0
B	5	10	6	1	1
C	10	17	10	0	0
D	0	10	2	2	2
E	12	17	14	2	2
F	0	10	0	0	0
G	10	14	12	2	0

$ES_{ij} = ES[i] \qquad LF_{ij} = LF[j]$

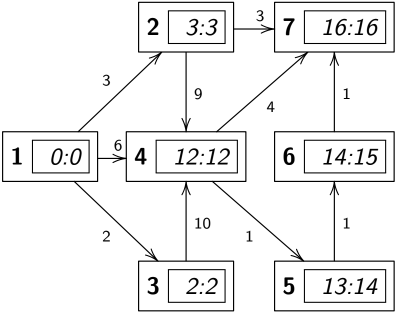
$LS_{ij} = LF_{ij} - d_{ij}$

$FT_{ij} = LS_{ij} - ES_{ij}$

$FL_{ij} = ES[j] - EF_{ij} = ES[j] - (ES_{ij} + d_{ij})$

Atividades críticas: C e F.

Exemplo 2



	ES	*LS*	*FT*	*FL*	*EF*	*LF*
(1, 2)	0	0	0	0	3	3
(1, 3)	0	0	0	0	2	2
(1, 4)	0	6	6	6	6	12
(2, 4)	3	3	0	0	12	12
(2, 7)	3	13	10	10	6	16
(3, 4)	2	2	0	0	12	12
(4, 5)	12	13	1	0	13	14
(4, 7)	12	12	0	0	16	16
(5, 6)	13	14	1	0	14	15
(6, 7)	14	15	1	1	15	16

Nós com anotação $\boxed{ES[v], LF[v]}$, sendo $ES[v]$ a data de início mais próxima para as tarefas que têm início no nó v e $LF[v]$ a data de conclusão mais afastada para as tarefas que têm fim em v.

Dois caminhos críticos: $((1,2),(2,4),(4,7))$ e $((1,3),(3,4),(4,7))$.

Propriedade: As atividades críticas são as que ocorrem em caminhos críticos.

Método do Caminho Crítico

Propriedades que suportam a correção dos métodos

- O grafo da relação de precedência é um DAG (grafo dirigido acíclico).
- A duração mínima do projeto é igual ao comprimento do caminho máximo no DAG.
- Qualquer DAG tem sempre algum vértice com grau de entrada zero.
- Se retirar alguma aresta a um DAG, obtém um DAG.
- O grafo transposto de um DAG é um DAG.

Nas referências sobre escalonamento sem partilha de recursos, o método descrito designa-se por **Método do Caminho Crítico** – *Critical path method* (CPM).

Escalonamento com partilha de recursos (extra aulas)

"Constraint-based scheduling is one of the most successful application areas of CP. One of the key factors of this success lies in the fact that a combination was found of the best of two fields of research that pay attention to scheduling – namely, operations research (OR) and artificial intelligence (AI)." Cap. 2, Handbook of Constraint Programming, Elsevier, 2006

https://doi.org/10.1016/S1574-6526(06)80026-X

Para saber mais...

- Course on Constraint Programming and Scheduling, by H.Rudová
  https://www.fi.muni.cz/~hanka/konstanz09/

Focaremos aqui o problema de minimização do número de trabalhadores.

Restrição Cumulative para Scheduling

Um projeto é constituído por tarefas. Cada uma requer um certo número de pessoas e tem uma duração. São dadas as precedências. As habilitações necessárias são idênticas para todas as tarefas. Determinar um calendário para as tarefas que **minimize o prazo de execução do projeto** e, **adicionalmente**, **minimize o número de trabalhadores a contratar**.

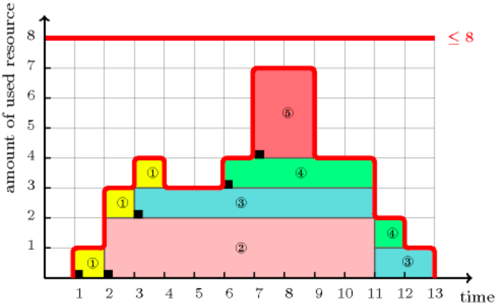


Ilustração de *Global Constraint Catalog* para a restrição global `cumulative`.

https://sofdem.github.io/gccat/gccat/Ccumulative.html

$$\forall t \quad \sum_{i:\ D_i \le t \le D_i + d_i} c_i \le C$$

O número total de pessoas necessárias para as tarefas que estão a decorrer em cada instante não excede o número das existentes.

Restrição Cumulative para Scheduling

Por ter diversas aplicações, a restrição `cumulative` está disponível como **restrição global** nos sistemas de programação por restrições, e tem associados algoritmos específicos de propagação de consistência.

O módulo `library(ic_cumulative)` do sistema ECLiPSe CLP contém `cumulative(+StartTimes, +Durations, +Resources, ++ResourceLimit)`, sendo

- `StartTimes` as datas de início para as tarefas (variáveis `fd` ou inteiros)
- `Durations` as suas durações (variáveis `fd` ou inteiros)
- `Resources` os recursos que usam (variáveis `fd` ou inteiros)
- ResourceLimit limite máximo de recurso disponível (inteiro)

Esta restrição impõe que, em cada instante, a soma total dos recursos usados não excede ResourceLimit.

Aplicação à minimização de trabalhadores

Para um caso de teste, definido por predicado `tarefa(J,SegJ,DurJ,TrabJ)`

```
tarefa(1,[2,3],5,4).%a   tarefa(5,[],3,2). %f    tarefa(3,[],2,1).  %b
tarefa(4,[6],2,2). %c    tarefa(6,[],10,1). %e   tarefa(2,[],12,1). %d
```

os resultados foram

```
Ntrabs minimo para as tarefas criticas : 4
Tarefas criticas e suas datas de inicio
tarefa(1) : inicio(0)
tarefa(2) : inicio(5)
Ntrabs se as tarefas comecam na data mais proxima : 8
Found a solution with cost 4
Trabalhadores : 4
tarefa(1) : inicio(0)
tarefa(4) : inicio(5)
tarefa(5) : inicio(7)
tarefa(6) : inicio(7)
tarefa(3) : inicio(5)
tarefa(2) : inicio(5)
Datas = [0, 5, 7, 7, 5, 5]  Concl = 17  Ntrabs = 4
```

Incerteza: *"E se as durações exatas não são conhecidas?"*

Para informação... não será tratado na uc de Desenho de Algoritmos

Nem sempre as durações exatas são conhecidas. O método PERT (Program Evaluation and Review Technique) assume que as durações das tarefas são variáveis aleatórias, i.e., seguem distribuições de probabilidade.

- É necessário estimar as durações das atividades (análise estatística).
- Dadas as durações *otimista* (a, " tudo corre bem"), *pessimista* (b, " tudo corre mal") e *mais frequente* (m) para cada tarefa, assume que a duração d segue uma **distribuição β** (semelhante à distribuição normal, sendo nula a probabilidade da duração da atividade ser $> b$ e $< a$) com:

valor esperado ($P(d \leq \mu_d) = 0.5$)	μ_d	$=$	$\frac{a+4m+b}{6}$
desvio padrão	σ_d	$=$	$\frac{b-a}{6}$

- Assume que as durações das tarefas são variáveis aleatórias independentes e que a **soma das durações das tarefas T num caminho crítico segue distribuição normal**, com média μ (dada pela soma das médias) e desvio padrão σ (soma dos desvios padrão): $\frac{T-\mu}{\sigma} \sim Normal(0,1)$.
- $\frac{T-\mu}{\sigma} \sim Normal(0,1)$ pode ser usado para: estimar a probabilidade de a duração do projeto ser menor ou igual ao valor calculado pelo CPM; indicar uma duração T que possa ser garantida com uma certa probabilidade, ...