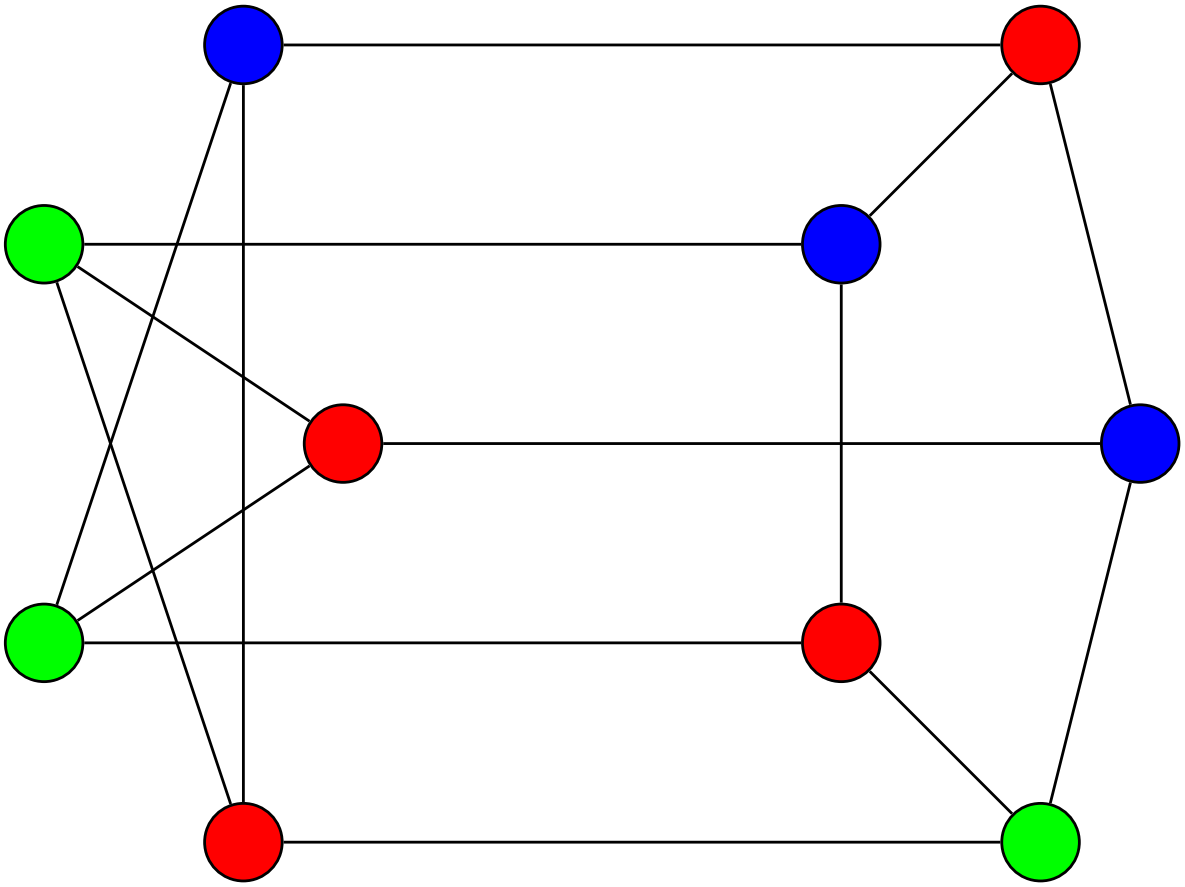
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC
Profa. Carla Negri Lintzmayer

Notas de aula da disciplina Teoria dos Grafos

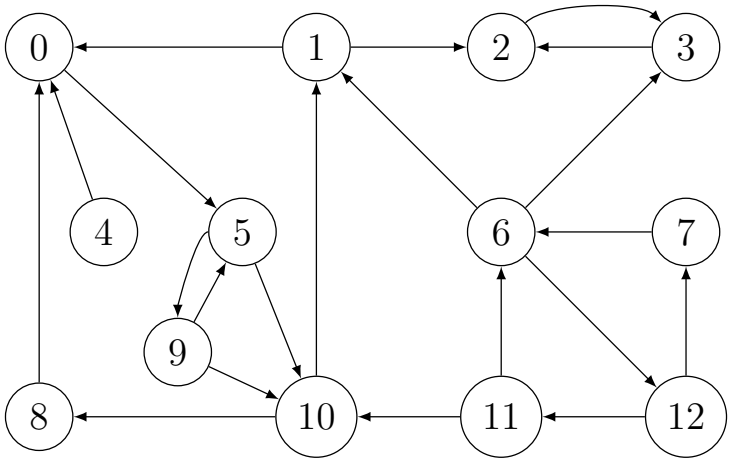


Esse material é um compilado dos seguintes:

- Sedgewick, R.. *Algorithms in C, part 5: graph algorithms*. 3rd ed. Addison-Wesley. 2002.
- Bondy, J. A.; Murty, U. S. R.. *Graph Theory*. Graduate Texts in Mathematics. Springer. New York. 2008.
- Lintzmayer, C. N.; Mota, G. O.. *Notas de aulas - Análise de algoritmos e estruturas de dados*. Em construção.

11 Aula 15: componentes fortemente conexas

- Em um grafo, se há uv-caminho, então há vu-caminho.
- Em um DAG, se há uv-caminho, então não há vu-caminho.
- Em um digrafo qualquer, tudo pode acontecer.
  - Porém dentro de uma componente fortemente conexa, se há uv-caminho, então há vu-caminho.
- Algoritmos clássicos famosos que detectam componentes fortemente conexas:
  - Tarjan: inventado em 1972, tempo $O(V + E)$
  - Kosajaru: inventado em 1980, tempo $O(V + E)$
  - Gabow: inventado em 1999, tempo $O(V + E)$



11.1 Algoritmo de Kosajaru

- Seja D um digrafo e sejam $D_1, \ldots, D_k$ todas as componentes fortemente conexas de D (cada D_i é um subdigrafo, portanto).
- Cada vértice de D pertence somente a uma componente e entre quaisquer duas componentes D_i e D_j existem arcos em apenas uma direção, pois caso contrário a união de D_i com D_j formaria uma componente maior contendo D_i e D_j.
- Então sempre existe pelo menos uma componente D_i que é um *ralo*: não existe arco saindo de D_i em direção a nenhuma outra componente.
  - Intuição: começar uma busca em uma tal componente, vai encontrar apenas os vértices dela.

- O algoritmo de Kosajaru tem essencialmente dois passos:

  1. Execute uma DFS sobre $\overleftarrow{D}$ e seja $S = (v_1, \ldots, v_n)$ uma ordenação dos n vértices de D de forma decrescente de *posordem*;
  2. Execute uma DFS sobre D usando S para percorrer os vértices.

```
1:  Função KOSAJARU(D)
2:      Para cada v ∈ V(D) faça
3:          visitado[v] ← 0
4:          pred[v] ← −1
5:          preordem[v] ← −1
6:          posordem[v] ← −1
7:      tempo ← 0
8:      Para cada v ∈ V(D) faça
9:          Se visitado[v] = 0 então
10:             DFS(←D, s)
11:     seja (v_1, ..., v_n) uma ordenação decrescente dos vértices por posordem
12:     Para cada v ∈ V(D) faça
13:         visitado[v] ← 0
14:         pred[v] ← −1
15:         comp[v] ← −1
16:     ncomp ← 0
17:     Para i = 1 até n faça
18:         Se visitado[v_i] = 0 então
19:             comp[v_i] = ncomp
20:             DFS2(D, v_i)              ▷ Alterar para atualizar comp
21:             ncomp ← ncomp + 1
```

```
1:  Função DFS2(D, s)
2:      visitado[s] ← 1
3:      Para cada v ∈ N+(s) faça
4:          Se visitado[v] = 0 então
5:              pred[v] ← s
6:              comp[v] ← comp[s]
7:              DFS2(D, v)
```

- Claramente, o algoritmo de Kosajaru leva tempo $O(V + E)$ pois é basicamente o tempo de realizar duas chamadas à DFS e construir $\overleftarrow{D}$.

Teorema 11.1. *Seja D um digrafo. Ao fim da execução de* KOSAJARU*(D) temos que, para quaisquer $u, v \in V(D)$, os vértices u e v estão na mesma componente fortemente conexa se e somente se $comp[u] = comp[v]$.*

Demonstração. Seja v_i um vértice arbitrário de D para o qual a linha 20 foi executada e seja D_i a componente fortemente conexa de D que contém v_i. Para provarmos o resultado do teorema, note que basta mostrarmos que após a chamada a DFS(D, v_i) (na linha 20), vale o seguinte:

$$v \in V(D) \text{ é visitado durante DFS}(D, v_i) \text{ se e somente se } v \in V(D_i)\,. \qquad (2)$$

De fato, se (2) é válida, então após a execução de DFS(D, v_i) teremos que os únicos vértices com $comp[v] = comp[v_i]$ são os vértices que estão em D_i. Assim, podemos refrasear o enunciado do teorema para "dois vértices estão na componente fortemente conexa D_i se e somente se eles são visitados durante uma chamada DFS(D, v_i)". Como o algoritmo KOSAJARU(D) só encerra sua execução quando todos os vértices são visitados, provar (2) é suficiente para concluir o resultado do teorema.

Para provarmos (2), vamos primeiro mostrar a seguinte afirmação.

Afirmação 11.2. *Se $v \in V(D_i)$, então v é visitado na chamada DFS(D, v_i).*

Demonstração. Seja $v \in V(D_i)$. Como v está na mesma componente fortemente conexa de v_i, então existe um v_iv-caminho e um vv_i-caminho em D. Se v já tivesse sido visitado no momento em que DFS(D, v_i) começa, então como existe vv_i-caminho, certamente o vértice v_i seria visitado, de modo que a chamada a DFS(D, v_i) nunca seria executada, levando a um absurdo. Portanto, sabemos no início da execução de DFS(D, v_i), o vértice v ainda não foi visitado. Logo, como existe um v_iv-caminho, o vértice v é visitado durante essa chamada. ◊

Para completar a prova, resta mostrar a seguinte afirmação.

Afirmação 11.3. *Se v é visitado na chamada DFS(D, v_i), então $v \in V(D_i)$.*

Demonstração. Seja $v \in V(D)$ um vértice que foi visitado durante a chamada DFS(D, v_i). Então existe um v_iv-caminho em D, e resta mostrar que existe um vv_i-caminho em D.

Como o laço **para** da linha 17 visita os vértices por ordem decrescente de *posordem* e como v foi visitado durante a chamada DFS(D, v_i), isso significa

que $posordem[u] > posordem[v]$. Portanto, durante as chamadas de DFS sobre $\overleftarrow{D}$, vale que

a chamada DFS($\overleftarrow{D}$, v) termina antes do fim da execução de DFS($\overleftarrow{D}$, v_i). (3)

Considere o momento do *início* da execução de DFS($\overleftarrow{D}$, v_i). Como existe vv_i-caminho em $\overleftarrow{D}$ (pois existe v_iv-caminho em D), sabemos que DFS($\overleftarrow{D}$, v_i) não pode ter sido iniciada após o término da execução de DFS($\overleftarrow{D}$, v), pois nesse caso v_i teria sido visitado durante a execução de DFS($\overleftarrow{D}$, v) e, por conseguinte, DFS($\overleftarrow{D}$, v) terminaria depois do fim da execução de DFS($\overleftarrow{D}$, u), contrariando (3).

Assim, a chamada DFS($\overleftarrow{D}$, v_i) teve início antes de DFS($\overleftarrow{D}$, v) e, por (3), terminou depois do fim de DFS($\overleftarrow{D}$, v), o que significa que existe um v_iv-caminho em $\overleftarrow{D}$. Portanto, existe um vv_i-caminho em D. ◊

□

11.2 Algoritmo de Tarjan

- Seja D um digrafo e sejam $D_1, \ldots, D_k$ todas as componentes fortemente conexas de D (cada D_i é um subdigrafo, portanto).
- Observe que, em uma árvore de DFS, uma componente fortemente conexa vai aparecer como uma subárvore.
- Além disso, não vai haver aresta de retorno de uma D_i para uma D_j (pode haver arestas de cruzamento ou de descida).
- Intuição: basta encontrar a raiz dessas subárvores!
- Vamos manter qual é a menor ordem de um vértice que pode ser atingido por uma aresta de retorno com um extremo na subárvore enraizada em u em $low[u]$.
  - Assim, se $low[u] = preordem[u]$, então não há aresta de um descendente de u para um ancestral de u e, portanto, u é raiz de uma componente fortemente conexa.
- Vamos também manter uma pilha, que irá armazenar os vértices que vão sendo alcançados mas que ainda não foram atribuídos a uma componente.
- Claramente, o algoritmo de Tarjan leva tempo $O(V + E)$ pois é basicamente o tempo de uma chamada à DFS (note que leva tempo $O(V)$ para desempilhar todos os vértices da pilha).

```
1: Função TARJAN(D)
2:     Para cada v ∈ V(D) faça
3:         pred[v] ← preordem[v] ← comp[v] ← low[v] ← −1
4:         visitado[v] ← esta_na_pilha ← 0
5:     tempo ← 0
6:     seja P uma pilha vazia
7:     ncomp ← 0
8:     Para cada s ∈ V(D) faça
9:         Se visitado[s] = 0 então
10:            AUXTARJAN(D, s)
```

```
1: Função AUXTARJAN(D, s)
2:     visitado[s] ← 1
3:     tempo ← tempo + 1
4:     preordem[s] ← low[s] ← tempo
5:     EMPILHA(P, s)
6:     esta_na_pilha[s] ← 1
7:     Para v ∈ N⁺(s) faça
8:         Se visitado[v] = 0 então
9:             pred[v] = s
10:            AUXTARJAN(D, v)
11:            Se low[v] < low[s] então
12:                low[s] ← low[v]
13:        Senão Se esta_na_pilha[v] = 1 e low[v] < low[s] então
14:            low[s] ← low[v]
15:    Se low[s] = preordem[s] então
16:        faça
17:            u ← DESEMPILHA(P)
18:            esta_na_pilha[u] ← 0
19:            comp[u] ← ncomp
20:        Enquanto u ≠ s
21:        ncomp ← ncomp + 1
```