SCC

Componentes Fortemente Conexas (CFC / SCC)

1. Descrição

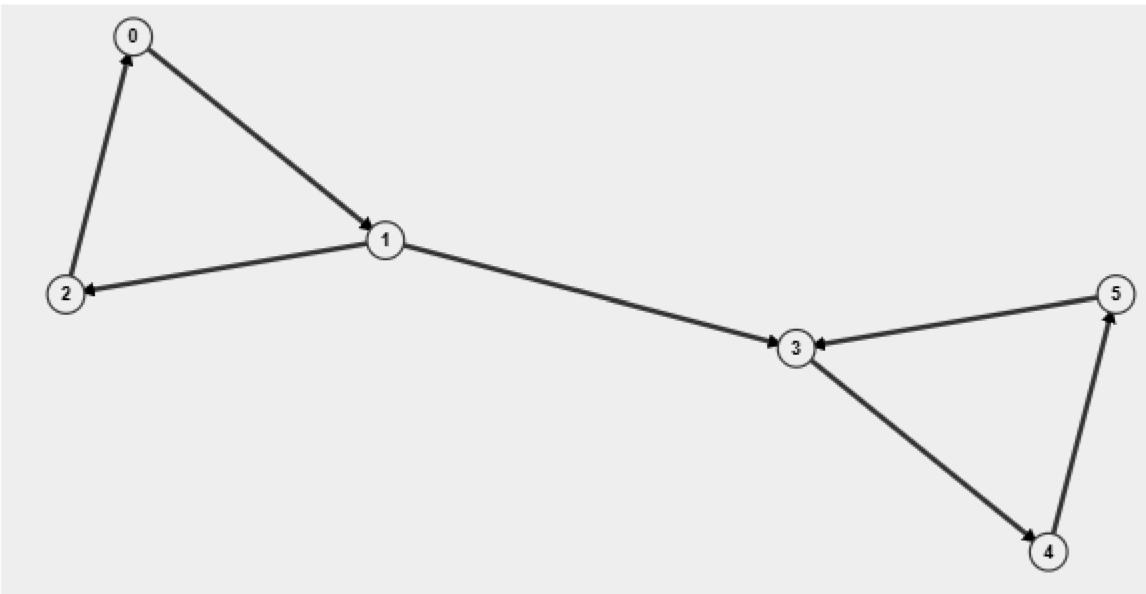
Uma **Componente Fortemente Conexa (CFC / SCC)** em um grafo direcionado (G=(V,E)) é um subconjunto máximo
de vértices (C ⊆ V) tal que, para quaisquer (u,v ∈ C), existe um caminho de (u) até (v) **e**
de (v) até (u). Em outras palavras: todos os vértices do componente se alcançam mutuamente.

CFCs são úteis para detectar grupos fechados em redes direcionadas, ciclos robustos e “influencers” que formam
núcleos recíprocos.

2. Grafo de exemplo

Considere o grafo direcionado com vértices {0, 1, 2, 3, 4, 5} e arestas:

- 0 → 1, 1 → 2, 2 → 0 (triângulo 0-1-2)
- 1 → 3
- 3 → 4, 4 → 5, 5 → 3 (triângulo 3-4-5)

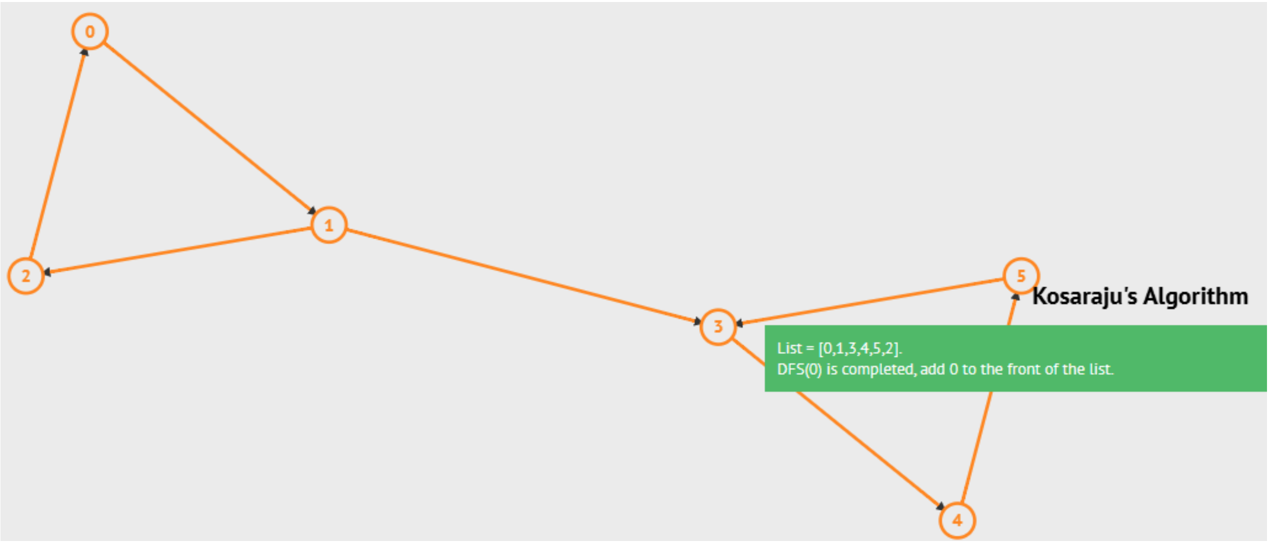


Neste grafo temos duas CFCs: {0, 1, 2} e {4, 5, 6}.

3. Algoritmo 1 — Kosaraju (duas passagens de DFS)

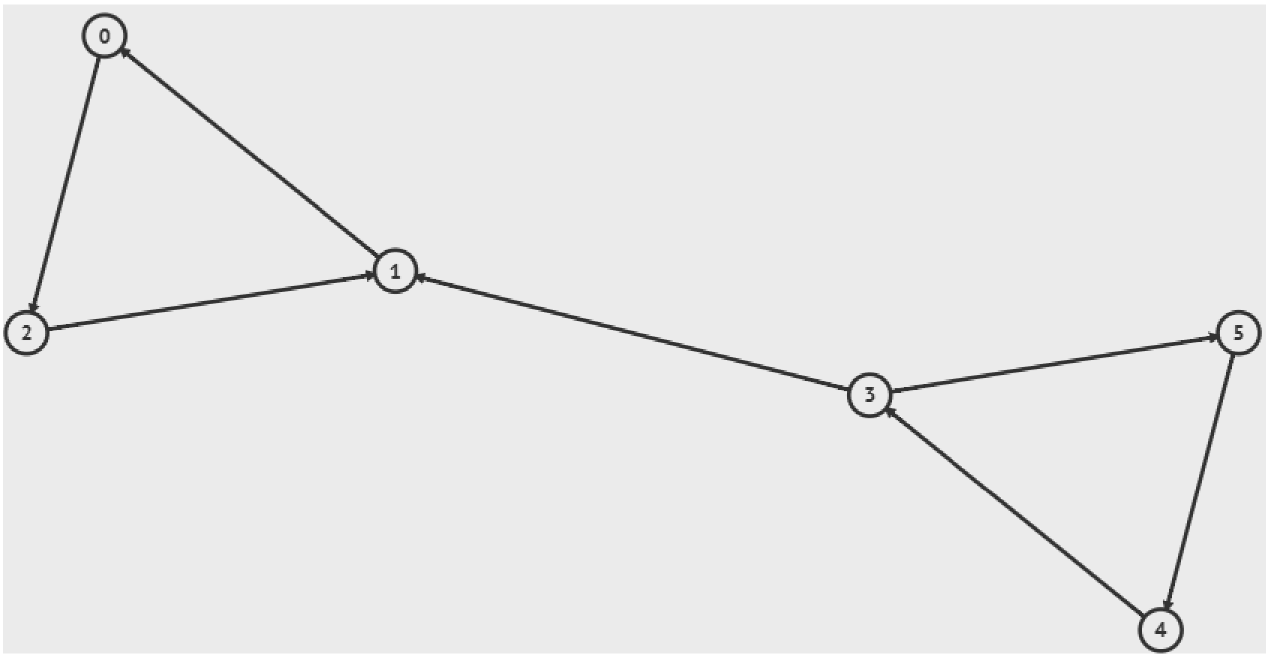
Ideia

1. Execute uma DFS em G e armazene os tempos de finalização (`f[u]`) de cada vértice.



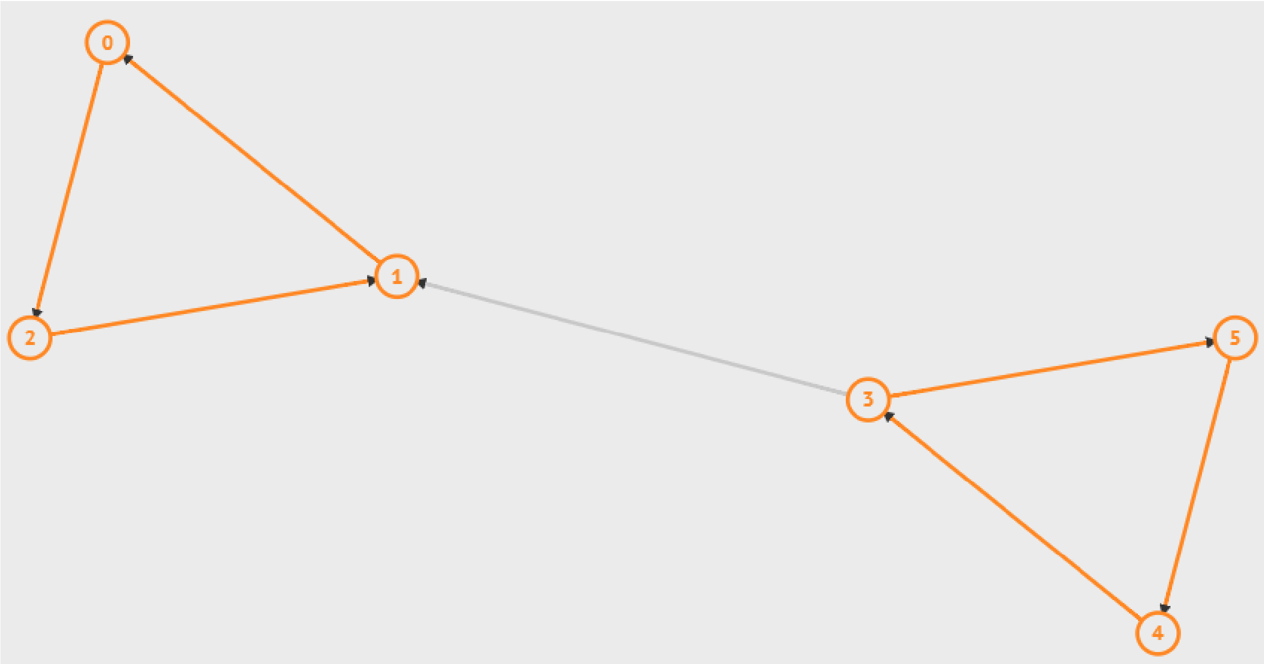
[0 → (1 / 12), 1 → (2 / 11), 2→ (3 / 4) 3 → (5 / 10), 4 → (6 / 9), 5 → (7 / 8)]

2. Construa o grafo transposto (G^T) (inverta todas as arestas).



3. Ordene os vértices em ordem decrescente de `f[u]` e, nessa ordem, execute DFS em (G^T).
   Ordem dos vértices: [0, 1, 3, 4, 5, 2]

4. Cada árvore gerada na segunda fase é uma CFC.



Execução passo a passo (exemplo)

1. DFS em G (suponha ordem alfabética de exploração): pode produzir tempos de término, por exemplo:
    a. f[A]=6, f[B]=5, f[C]=4, f[D]=3, f[E]=2, f[F]=1 (valores ilustrativos: A tem maior f)
2. Constrói-se (G^T): arestas invertidas.
3. Ordena-se por f decrescente: [A, B, C, D, E, F].
4. Rodar DFS em (G^T) na ordem acima:
    a. Primeiro DFS a partir de A visita A,B,C → componente {A,B,C}.
    b. Em seguida, começando em D visita D,E,F → componente {D,E,F}.

Pseudocódigo (Kosaraju)

```
KOSARAJU-SCC(G)
    // 1ª fase: calcular tempos de término
    for each u ∈ V[G]:
        cor[u] ← branco
    tempo ← 0
    L ← empty list
    for each u ∈ V[G]:
        if cor[u] = branco:
            KDFS(G, u, L)

    // 2ª fase: criar grafo transposto G^T
    Gt ← TRANSPOSE(G)
```

```
    // 3ª fase: processar vértices por ordem decrescente de término
    for each u in L in order of decreasing f (i.e., reverse(L)):
        if cor_t[u] = branco:
            componentes ← []
            KDFS-Collect(Gt, u, componentes)
            output componentes as one SCC

procedure KDFS(G, u, L)
    cor[u] ← cinza
    tempo ← tempo + 1
    d[u] ← tempo
    for each v ∈ Adj[u]:
        if cor[v] = branco:
            KDFS(G, v, L)
    cor[u] ← preto
    tempo ← tempo + 1
    f[u] ← tempo
    append u to L

procedure KDFS-Collect(G, u, componentes)
    cor_t[u] ← cinza
    add u to componentes
    for each v ∈ Adj_t[u]:
        if cor_t[v] = branco:
            KDFS-Collect(G, v, componentes)
    cor_t[u] ← preto
```

Complexidade

- Construir (G^T): (O(V+E)) (lista de adjacência)
- Duas DFS: (O(V+E)) cada
- Total: (O(V+E)).

4. Algoritmo 2 — Tarjan (uma passagem com stack)

Ideia

Tarjan encontra todas as CFCs em **uma única DFS** usando uma pilha e os valores `index[u]` e `lowlink[u]`:

- `index[u]`: ordem de descoberta (incremental).
- `lowlink[u]`: menor `index` alcançável a partir de `u` seguindo arestas e possivelmente subárvores.
  Quando `lowlink[u] == index[u]`, `u` é raiz de uma CFC; desempilha-se até `u`.

Pseudocódigo (Tarjan)

```
TARJAN-SCC(G)
    index ← 0
    S ← empty stack
    for each v ∈ V[G]:
        index[v] ← UNDEFINED
    for each v ∈ V[G]:
        if index[v] = UNDEFINED:
            TARJAN-DFS(v)

procedure TARJAN-DFS(v)
    index[v] ← index
    lowlink[v] ← index
    index ← index + 1
    push v onto S
    onStack[v] ← true

    for each w ∈ Adj[v]:
        if index[w] = UNDEFINED then
            TARJAN-DFS(w)
            lowlink[v] ← min(lowlink[v], lowlink[w])
        else if onStack[w] then
            lowlink[v] ← min(lowlink[v], index[w])

    // If v is a root node, pop the stack and generate an SCC
    if lowlink[v] = index[v] then
        start a new SCC
        repeat
            w ← pop S
            onStack[w] ← false
            add w to current SCC
        until w = v
        output current SCC
```

Complexidade

- Uma única DFS, cada aresta e vértice é processado (O(1)) vezes → (O(V+E)).
- Implementação popular e eficiente em prática.

5. Observações finais

- **Kosaraju** é simples de entender e implementar (usa transposição do grafo).

- **Tarjan** é mais elegante (uma única passagem) e frequentemente preferido quando se quer evitar criar explicitamente (G^T).
- Ambos rodam em tempo linear (O(V+E)) para representações por listas de adjacência.