AED2 - Aula 25
Componentes fortemente conexos, algoritmo de Kosaraju

Em um grafo não orientado, um componente conexo

- é um conjunto de vértices maximal em que
    - entre qualquer par de vértices, existe um caminho.
- Numa intuição física, se imaginarmos o grafo construído com linhas,
    - um componente conexo é um objeto que não pode ser separado,
        - sem romper as “linhas” que unem os vértices.

Num grafo orientado (ou dirigido), por conta da orientação dos arcos,

- ao considerarmos um par de vértices qualquer a e b,
    - é possível que haja caminho de a para b, mas não de b para a.
- Por isso, o conceito de componente conexo ganha uma certa nuance.

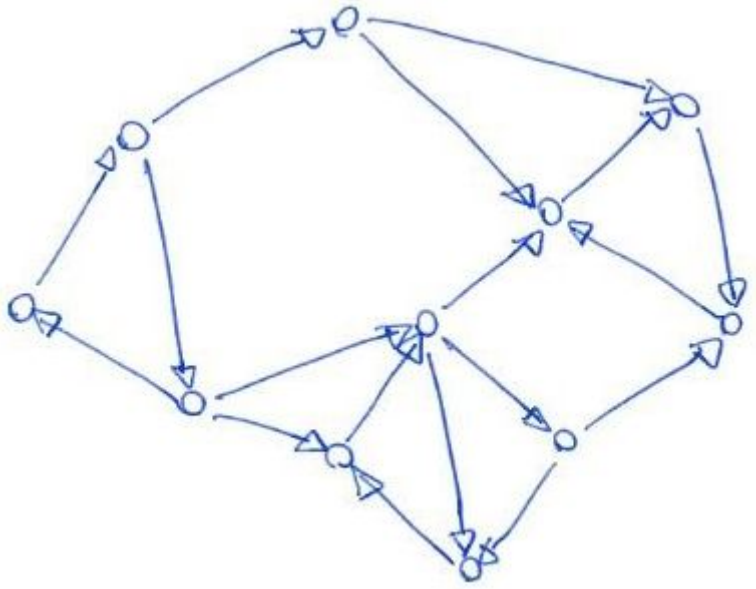
Podemos falar em componentes fracamente conexos, que correspondem

- aos componentes conexos que encontramos se
    - desconsideramos a orientação dos arcos e
        - tratarmos eles como arestas de um grafo não-orientado.

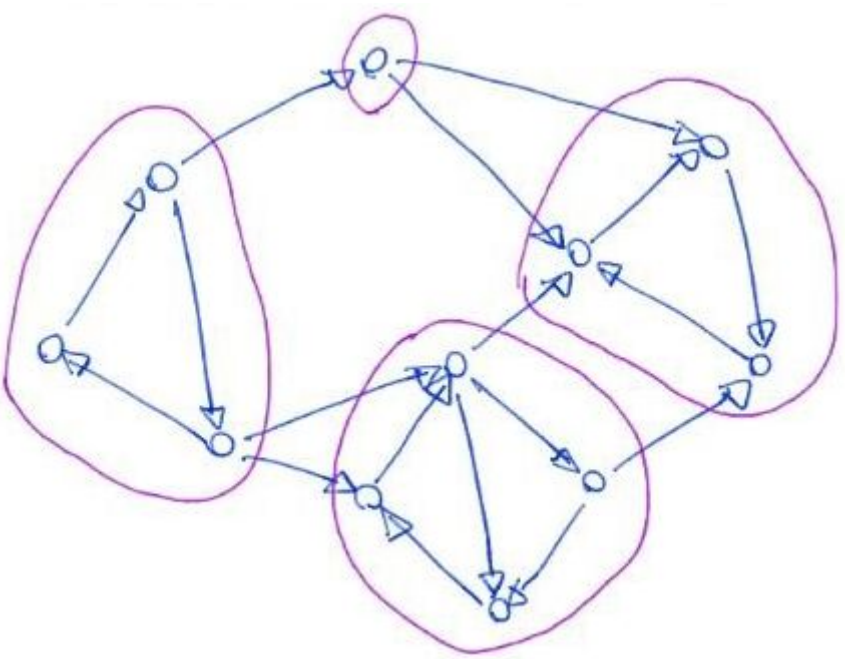
Também podemos falar de componente fortemente conexo,

- que é um subconjunto S maximal de vértices
    - tal que para quaisquer dois vértices u e v em S
        - existe caminho de u pra v e também caminho de v para u.

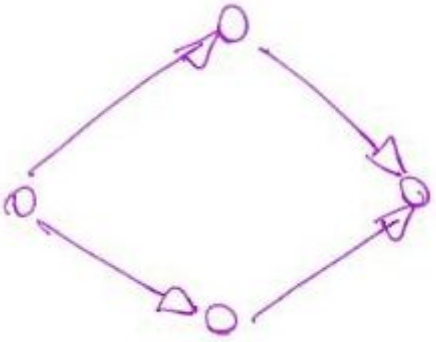
Como exemplo, considere o seguinte grafo dirigido



- Seus componentes fortemente conexos são



- Podemos contrair cada componente em um único vértice, obtendo



- Note que, o grafo resultante é um DAG. Será coincidência?
    - Não, pois se houvesse algum ciclo no grafo resultante,
        - isso colapsaria os vários componentes do ciclo
            - em apenas um componente
                - (e num único vértice no grafo contraído).

Para desenvolver nossa intuição sobre o problema

- e sobre como resolvê-lo,
    - podemos realizar buscas no grafo anterior.
- Dependendo a partir de qual vértice começamos uma busca,
    - nós encontramos exatamente um componente fortemente conexo.
        - Isso acontece se começarmos pelos vértices mais à direita.
- No entanto, se começarmos de outros vértices podemos acabar
    - encontrando vários componentes misturados, o que não nos ajuda.
        - Isso acontece quando começamos pelos vértices à esquerda.
- De modo geral, quando começamos a busca
    - a partir de uma componente sorvedouro,

        - encontramos um componente fortemente conexo corretamente.
- Um componente fortemente conexo é sorvedouro se
    - não tem arcos indo dele para outros componentes fortemente conexos.
    - Note que, tal componente corresponderá a um vértice sorvedouro
        - no grafo contraído.

Como saber a partir de quais vértices começar a busca?

- Ou seja, como localizar uma componente sorvedouro?

Para descobrir isso vamos usar alguns conceitos:

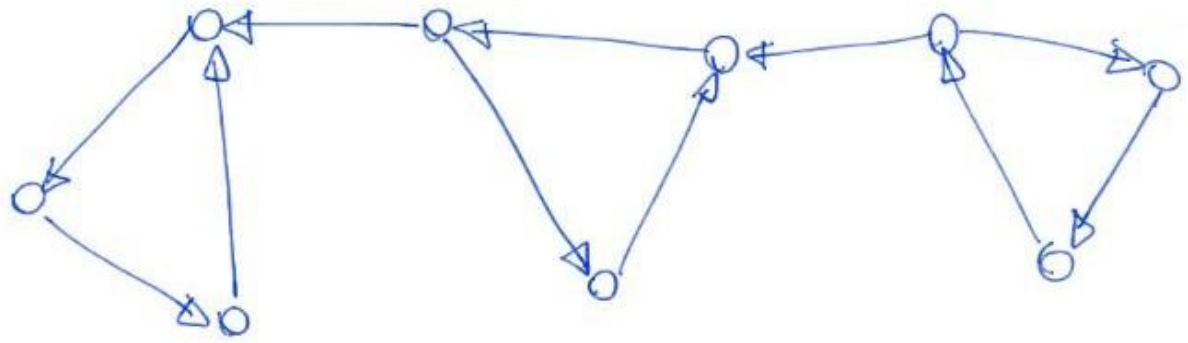
- Componente fonte - um componente fortemente conexo é fonte
    - se não tem arcos vindo de outros componentes para ele.
- Tempo de término de um vértice, que corresponde ao momento em que
    - a busca termina de passar por esse vértice.
    - Ele se assemelha com o rótulo que usamos na ordenação topológica,
        - mas não é decrescente.

Vamos ver/lembrar como usar a busca em profundidade para registrar o tempo de término dos vértices.

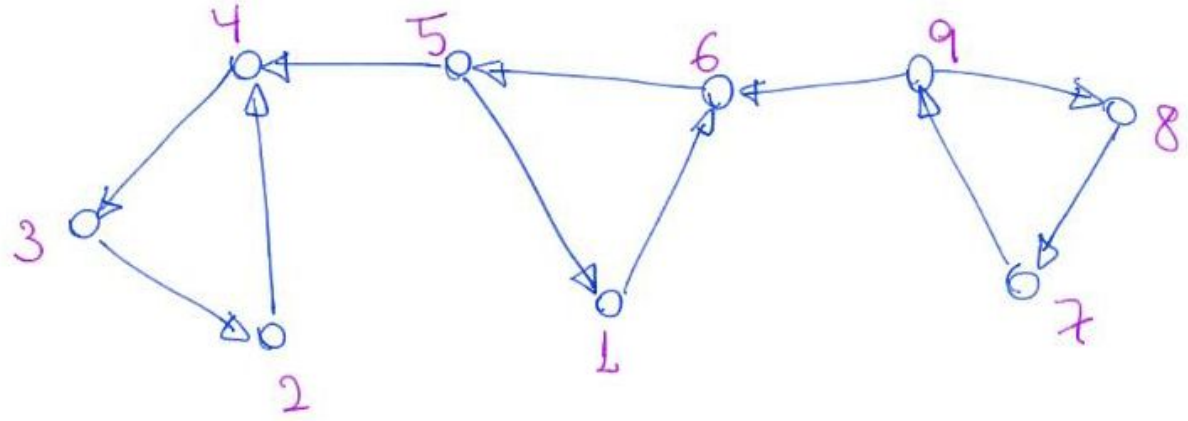
```
LoopBuscaProfT(grafo G=(V,E)) {
        marque todos os vértices em V como não visitados
        t = 0
        para cada v \in V
                se v não foi visitado
                        buscaProfRecT(G, v)
}

buscaProfRecT(grafo G=(V,E), vértice v) {
        marque v como visitado
        para cada arco (v, w)
                se w não foi visitado
                        buscaProfRec(G, w)
        t++
        defina f(v) = t
}
```

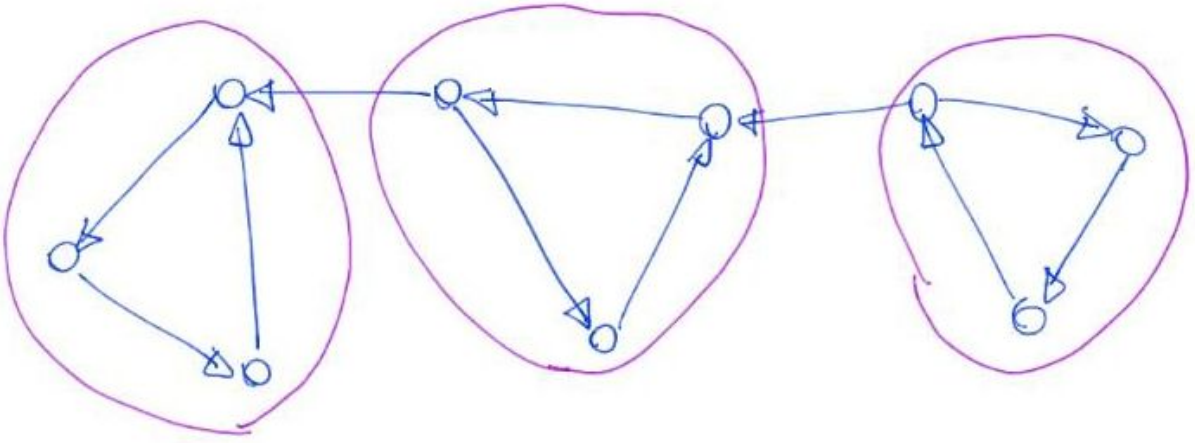
Vamos exemplificar o algoritmo anterior no seguinte grafo



- Possíveis tempos de término são



- Os componentes fortemente conexos do grafo anterior são



Do exemplo anterior, podemos inferir que,

- o vértice v com maior tempo de término
    - está em uma componente fonte.

De fato, isso é sempre verdade. Provando por contradição,

- suponha que embora v tenha o maior tempo de término,
    - ele não está em um componente fonte.
- Neste caso, deve existir um vértice u que está em outra componente
    - e que tem arcos incidindo na componente de v,
        - i.e., u alcança v, mas v não alcança u.

- Daí temos duas possibilidades,
    - se u foi visitado antes de v,
        - então v será visitado antes que u seja finalizado,
            - o que levaria a tempo de u maior que tempo de v (absurdo).
    - já, se v foi visitado antes de u,
        - então v seria finalizado antes de u ser visitado.
        - Novamente, tempo de u seria maior que o tempo de v. (absurdo)
- Como chegamos a uma contradição nos dois casos,
    - concluímos a demonstração.

Uma observação importante, alguns exemplos podem nos levar a crer

- que o menor tempo de término estará nos componentes sorvedouros.
- No entanto, não existe garantia de que isso ocorra.

Código do loop da busca em profundidade para marcar tempos de término:

```
void loopBuscaProfTempoTermino(Graph G, int *tempoTermino)
{
   int i, t, *visitado;
   visitado = malloc(G->n * sizeof(int));
   /* inicializa todos como não visitados e sem ordem topologica */
   for (i = 0; i < G->n; i++)
   {
       visitado[i] = 0;
       tempoTermino[i] = -1;
   }
   t = 0;
   for (i = 0; i < G->n; i++)
       if (visitado[i] == 0)
           buscaProfTempoTerminoR(G, i, visitado, tempoTermino, &t);
   free(visitado);
}

void buscaProfTempoTerminoR(Graph G, int v, int *visitado, int *tempoTermino, int *pt)
{
   int w;
   link p;
   visitado[v] = 1;
   /* para cada vizinho de v que ainda não foi visitado */
   p = G->A[v];
   while (p != NULL)
   {
       w = p->index;
       if (visitado[w] == 0)
           buscaProfTempoTerminoR(G, w, visitado, tempoTermino, pt);
```

```
        p = p->next;
    }
    tempoTermino[*pt] = v; // observe que o vetor é indexado pelos tempos e armazenas os
vértices em ordem crescente de tempo de termino
    (*pt)++;
}
```

- Um detalhe importante é que, este algoritmo
    - armazenar os tempos de término usando
        - um vetor indexado por tempo de término,
            - cujos conteúdos são os rótulos dos vértices.
- Isso é mais eficiente que armazenar um vetor indexado por vértices,
    - cujos conteúdos são tempos de término, pois
        - quando o próximo algoritmo for usar tais tempos,
            - o vetor não precisa ser ordenado.

Voltando ao nosso problema de detectar componentes fortemente conexos,

- nosso interesse era encontrar vértices
    - que estão em componentes sorvedouros.
- Isso porque fazer uma busca a partir de um vértice de um sorvedouro,
    - encontra todos os vértices de uma componente fortemente conexa,
        - e nenhum a mais.

No entanto, acabamos de analisar uma forma

- de encontrar vértices de componentes fontes.
- Por isso, para resolver nosso problema
    - vamos começar invertendo a orientação dos arcos.
- Só então vamos realizar o loop da busca em profundidade,
    - para registrar os tempos de término.
- Isso porque, uma componente fonte no grafo invertido
    - é uma componente sorvedouro no grafo original.
- Note que, inverter os arcos não altera os conjuntos de vértices,
    - que pertencem a cada componente fortemente conexas.

Algoritmo de Duas Passadas de Kosaraju

1. Computa Grev invertendo todos os arcos de G.
2. Executa LoopBuscaProfT(Grev) para computar os tempos de término
    a. que permitirão localizar vértices de componentes sorvedouros.
3. Executa LoopBuscaProfL(G) começando cada busca em
    a. ordem decrescente de tempo de término e
    b. marcando os vértices visitados em cada busca com um rótulo distinto.

Código da função principal do algoritmo de Kosaraju:

```
void identCompForteConexo(Graph G, int *comp)
```

```
{
   int i, j, *tempoTermino;
   link p;
   Graph Grev;
   Grev = graphInit(G->n);
   // reverte os arcos do grafo G
   for (i = 0; i < G->n; i++)
   {
       p = G->A[i];
       while (p != NULL)
       {
           j = p->index;
           graphInsertArc(Grev, j, i);
           p = p->next;
       }
   }
   tempoTermino = malloc(G->n * sizeof(int));
   loopBuscaProfTempoTermino(Grev, tempoTermino);
   Grev = graphFree(Grev);
   loopBuscaProfIdentComp(G, tempoTermino, comp);
   free(tempoTermino);
}
```

Eficiência:

- Este algoritmo executa em tempo O(n + m). Vale destacar que,
    - para tanto é necessário representar o grafo com listas de adjacência,
        - e tomar o cuidado armazenar os vértices
            - em ordem decrescente de tempo de término.

Faltou detalharmos os pseudocódigos do passo três do algoritmo:

```
LoopBuscaProfL(grafo G=(V,E)) {
        marque todos os vértices em V como não visitados
        // suponha que os vértices estão nomeados de acordo com seus tempos de término calculados anteriormente
        para v = n até 1
                se v não foi visitado
                        s = v
                        buscaProfRecL(G, v)
}

buscaProfRecL(grafo G=(V,E), vértice v) {
        marque v como visitado
        defina lider(v) = s
        para cada arco (v, w)
```

se w não foi visitado
buscaProfRecL(G, w)
}

Código do loop da busca em profundidade para identificar as componentes:

```
void loopBuscaProfIdentComp(Graph G, int *tempoTermino, int *comp)
{
   int i, k, v;
   /* inicializa todos como não pertencentes */
   for (i = 0; i < G->n; i++)
       comp[i] = -1;
   k = 0;
   for (i = G->n - 1; i >= 0; i--) // em ordem decrescente de tempo de término
   {
       v = tempoTermino[i];
       if (comp[v] == -1)
       {
           k++;
           buscaProfIdentCompR(G, v, comp, k);
       }
   }
}

void buscaProfIdentCompR(Graph G, int v, int *comp, int k)
{
   int w;
   link p;
   comp[v] = k;
   /* para cada vizinho de v que ainda não foi visitado */
   p = G->A[v];
   while (p != NULL)
   {
       w = p->index;
       if (comp[w] == -1)
           buscaProfIdentCompR(G, w, comp, k);
       p = p->next;
   }
}
```

Agora vamos mostrar que o algoritmo está correto, ou seja,

- que a busca a partir de um vértice v com maior tempo de término
    - realmente revela uma componente fortemente conexa.

Observe que, existe um caminho a partir de v

- até qualquer vértice w que a busca encontrou (pela propriedade da busca).

Então, só precisamos mostrar que existe um caminho de um w qualquer até v, pois

- pela transitividade (decorrente da concatenação de caminhos)
    - isso implica que existe caminho nos dois sentidos
        - entre qualquer par de vértices localizado na busca,
- o que implica que temos uma componente fortemente conexa.

Assim, tome um vértice w qualquer que foi alcançado a partir de v,

- vamos mostrar que existe um caminho a partir de w até v em G.

Sabemos que em Grev existe um caminho de w até v,

- pois em G o vértice v alcança w.

Além disso, o tempo de término de v

- é maior que o tempo de término de w nas buscas realizadas em Grev.

Vamos analisar as possibilidades para que isso ocorra:

- Se w fosse visitado antes que v no passo 1 do algoritmo,
    - o tempo de término de w seria maior, i.e., t[w] > t[v],
        - já que existe o caminho de w até v em Grev.

- Portanto, v foi visitado antes que w.
    - Mas, se não houver um caminho de v até w em Grev,
        - então mesmo nesse caso v será finalizado
            - antes de w ser visitado.

    - Novamente w ficará com tempo de término maior que v, t[w] > t[v].
- Logo, a única alternativa sobrando é que
    - v deve ter sido visitado antes que w
        - e deve existir um caminho de v até w em Grev.

$$\exists\, v \rightsquigarrow w \text{ em } G_{rev}$$

- Mas isso implica num caminho de w até v em G,

$$\exists\, w \rightsquigarrow v \text{ em } G$$

    - que é o que queríamos demonstrar.