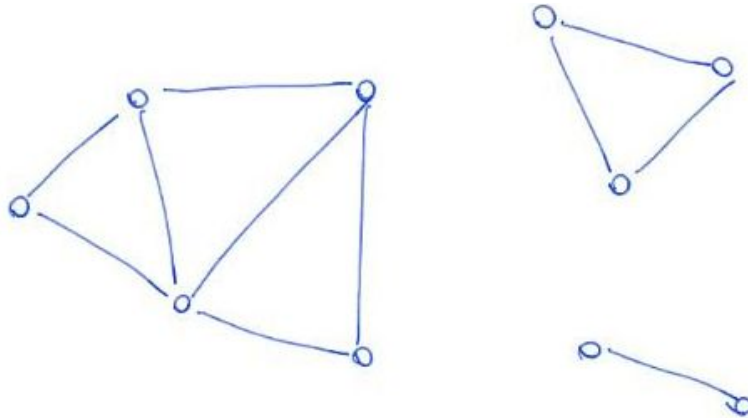


AED2 - Aula 24

Componentes fortemente conexos, algoritmo de Kosaraju

Em um grafo não orientado, um componente conexo

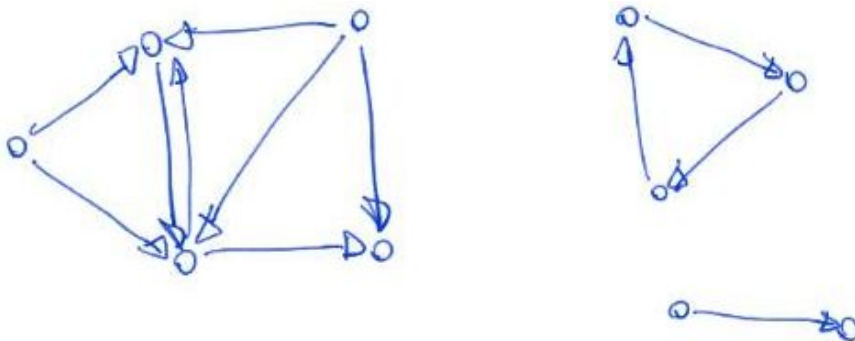
- é um conjunto de vértices maximal em que
 - entre qualquer par de vértices, existe um caminho.



- 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.



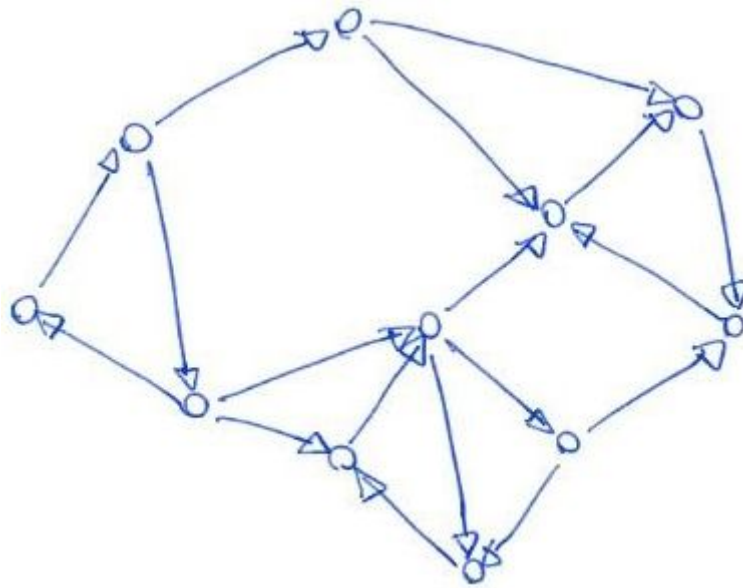
Podemos falar em componentes fracamente conexos, que correspondem

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

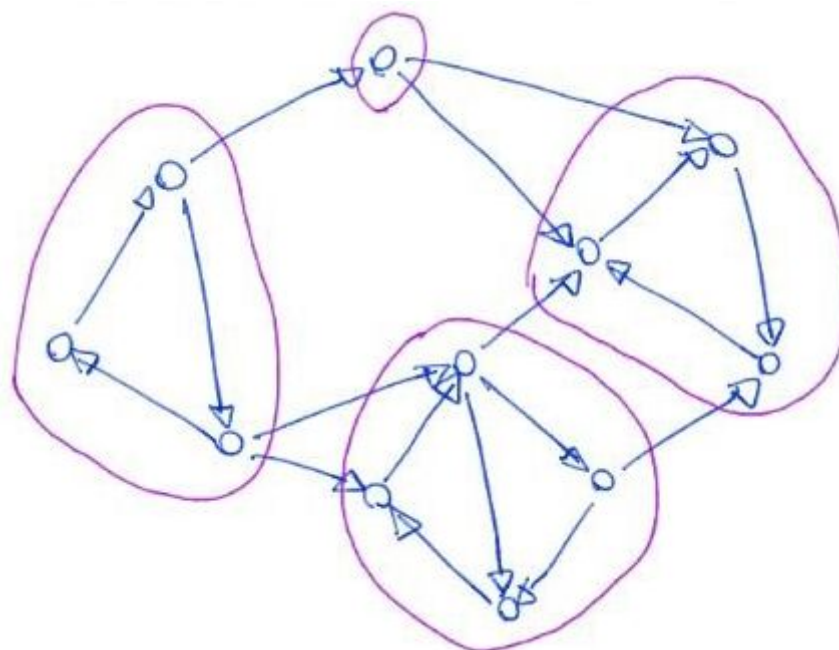
Também podemos falar de componente fortemente conexo,

- 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 .

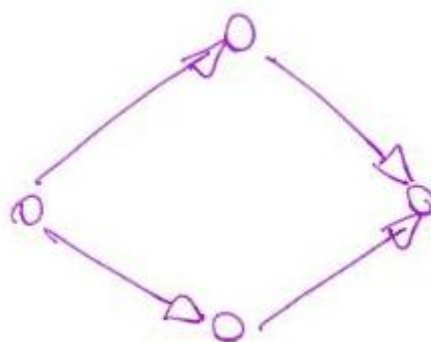
Como exemplo, considere o seguinte grafo dirigido



- 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.
- Por exemplo, 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 um 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 - corresponde ao momento em que
 - a busca termina de passar por esse vértice,
 - após explorar todos os vértices alcançáveis a partir dele.
- Vimos isso na primeira aula de busca em profundidade.

Vamos ver/lembrar como usar a busca em profundidade

- para registrar o tempo de término dos vértices.

```

LoopBuscaProfTempoTerm(grafo G=(V,E)) {
  marque todos os vértices em V como não visitados
  t = 0
  para cada v ∈ V
    se v não foi visitado
      buscaProfRecTT(G, v)
}

```

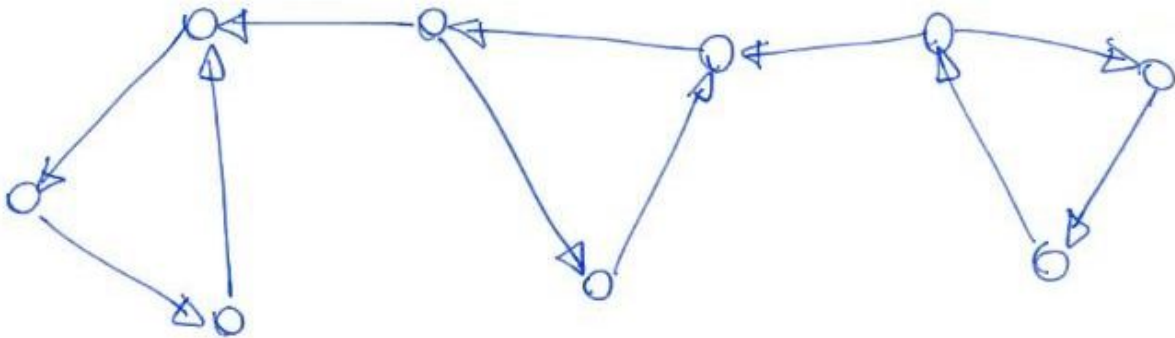
```

buscaProfRecTT(grafo G=(V,E), vértice v) {
  marque v como visitado
  para cada arco (v, w)
    se w não foi visitado
      buscaProfRecTT(G, w)

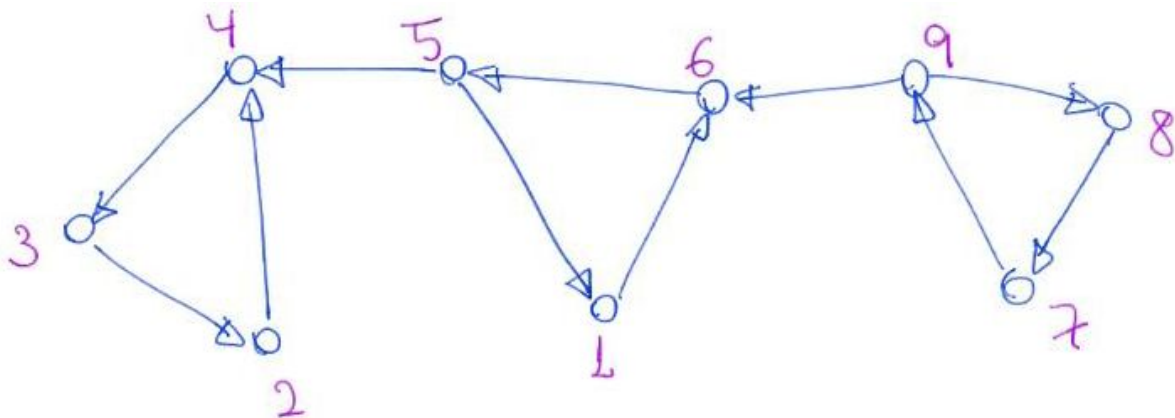
  t++
  tempoTerm(v) = t
}

```

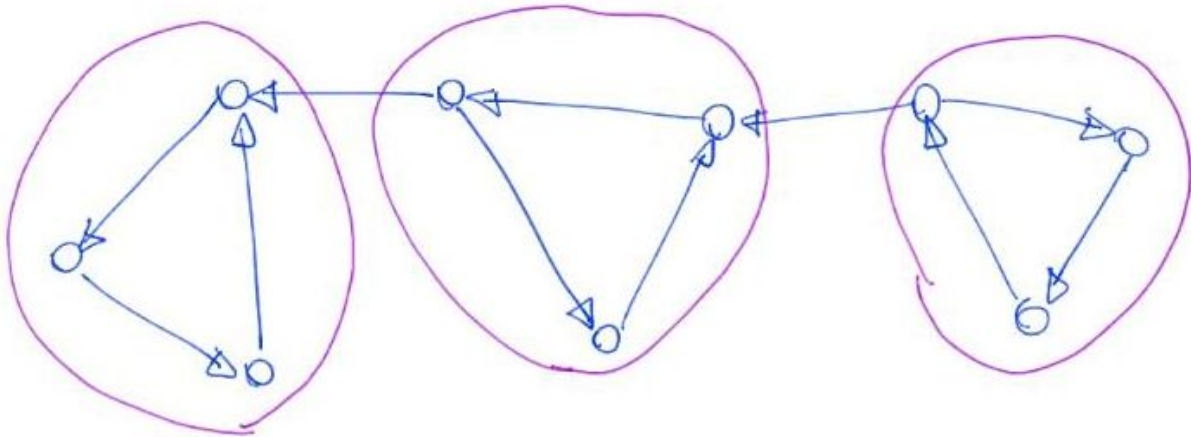
Vamos exemplificar o algoritmo anterior no seguinte grafo



- Possíveis tempos de término são



- Os componentes fortemente conexos do grafo anterior são

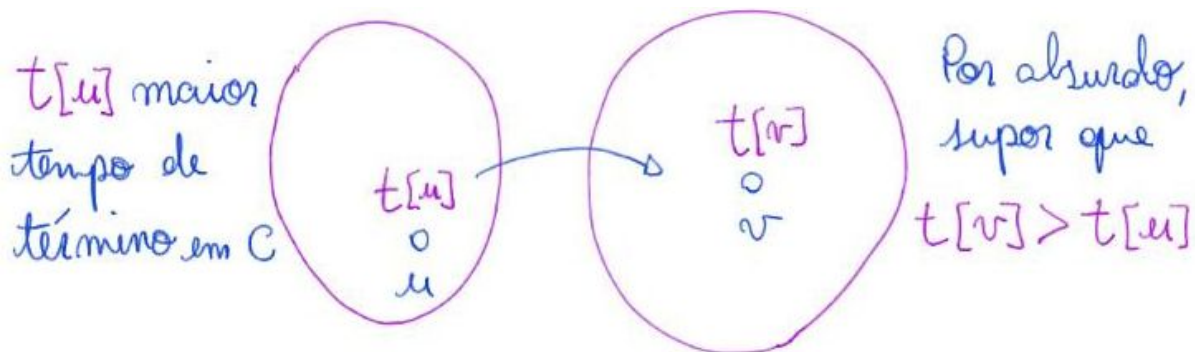


Do exemplo anterior, podemos inferir que,

- 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 uma outra componente C
 - que tem arcos incidindo na componente de v .
- Sendo u o primeiro vértice de C a ser visitado,
 - temos que u alcança v , mas v não alcança u .



- Dáí temos duas possibilidades.
 - Se u foi visitado antes de v ,
 - então v será visitado antes que u seja finalizado,
 - o que leva a tempo de u maior que tempo de v (absurdo),
 - já que o tempo de término só cresce.
 - Se v foi visitado antes de u ,
 - então v seria finalizado antes de u ser visitado,
 - pois não existe caminho de v para u .
 - Novamente, como o tempo de término só cresce,
 - tempo de u será maior que o tempo de v (absurdo).
- Como chegamos a uma contradição nos dois casos,
 - concluimos 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,
 - pois os primeiros caminhos seguidos pela busca em profundidade
 - podem terminar em vértices de qualquer componente.

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

```
void loopBuscaProfTempoTerm(Grafo G, int *tempoTermino) {
    int v, tempo, *visitado;
    visitado = malloc(G->n * sizeof(int));
    // inicializa todos como não visitados e sem tempo de término
    for (v = 0; v < G->n; v++) {
        visitado[v] = 0;
        tempoTermino[v] = -1;
    }
    tempo = 0;
    // inicia uma busca em profundidade a partir de cada vértice não
    visitado
    for (v = 0; v < G->n; v++)
        if (visitado[v] == 0)
            buscaProfTempoTermR(G, v, visitado, tempoTermino,
&tempo);
    free(visitado);
}

void buscaProfTempoTermR(Grafo G, int v, int *visitado,
    int *tempoTermino, int *ptempo) {
    int w;
    Noh *p;
    visitado[v] = 1; // marca v como visitado
    // para cada vizinho de v que ainda não foi visitado
    p = G->A[v];
    while (p != NULL) {
        w = p->rotulo;
        if (visitado[w] == 0)
```

```

        buscaProfTempoTermR(G, w, visitado, tempoTermino,
p tempo);
        p = p->prox;
    }
    // observe que o vetor é indexado pelos tempos e armazena
    // os vértices em ordem crescente de tempo de término
    tempoTermino[*p tempo] = v;
    (*p tempo)++;
}

```

- Um detalhe importante é que,
 - este algoritmo armazena 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.
- Uma curiosidade é que, o que o algoritmo faz
 - é equivalente a empilhar os vértices conforme eles são finalizados,
 - e na segunda passada ir desempilhando para visitá-los.

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 e implementar um algoritmo

- para encontrar vértices de componentes fontes.
 - Como isso nos ajuda a resolver nosso problema?

Para resolvê-lo, 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, um componente fonte no grafo invertido
 - é um componente sorvedouro no grafo original.
- Note que, inverter os arcos não altera os conjuntos de vértices
 - que pertencem a cada componente fortemente conexo. Por que?

Algoritmo de Duas Passadas de Kosaraju

1. Computa G_{rev} invertendo todos os arcos de G .
2. Roda $LoopBuscaProfTempoTerm(G_{rev})$ para computar os tempos de término,
 - a. que permitirão localizar vértices de componentes sorvedouros.
3. Executa $LoopBuscaProfIdentComp(G)$, 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(Grafo G, int *comp) {
    int u, v, *tempoTermino;
    Noh *p;
    Grafo Grev;
    Grev = inicializaGrafo(G->n);
    // reverte os arcos do grafo G
    for (u = 0; u < G->n; u++) {
        p = G->A[u];
        while (p != NULL) {
            v = p->rotulo;
            insereArcoGrafo(Grev, v, u);
            p = p->prox;
        }
    }
    tempoTermino = malloc(G->n * sizeof(int));
    loopBuscaProfTempoTerm(Grev, tempoTermino);
    Grev = liberaGrafo(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 de armazenar os vértices
 - em ordem decrescente de tempo de término.

Faltou detalharmos os pseudocódigos do passo três do algoritmo:

```
LoopBuscaProfIdentComp(grafo G=(V,E)) {
    marque todos os vértices em V como não visitados
    r = 0
    // 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
            r++
            buscaProfReclentComp(G, v)
}
```

```
buscaProfReclentComp(grafo G=(V,E), vértice v) {
    marque v como visitado
    comp(v) = r
    para cada arco (v, w)
        se w não foi visitado
            buscaProfReclentComp(G, w)
}
```

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

```
void loopBuscaProfIdentComp(Grafo G, int *tempoTermino, int *comp) {
    int v, i, num_comp;
    // inicializa todos como não pertencentes
    for (v = 0; v < G->n; v++)
        comp[v] = -1;
    num_comp = 0;
    // inicia buscas a partir de vértices não visitados
    // seguindo a ordem decrescente dos tempos de término
    for (i = G->n - 1; i >= 0; i--) {
        v = tempoTermino[i];
        if (comp[v] == -1) {
            num_comp++;
            buscaProfIdentCompR(G, v, comp, num_comp);
        }
    }
}
```

```

void buscaProfIdentCompR(Grafo G, int v, int *comp, int num_comp) {
    int w;
    Noh *p;
    comp[v] = num_comp; // coloca v no componente atual
    // para cada vizinho de v que ainda não foi visitado
    p = G->A[v];
    while (p != NULL) {
        w = p->rotulo;
        if (comp[w] == -1)
            buscaProfIdentCompR(G, w, comp, num_comp);
        p = p->prox;
    }
}

```

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

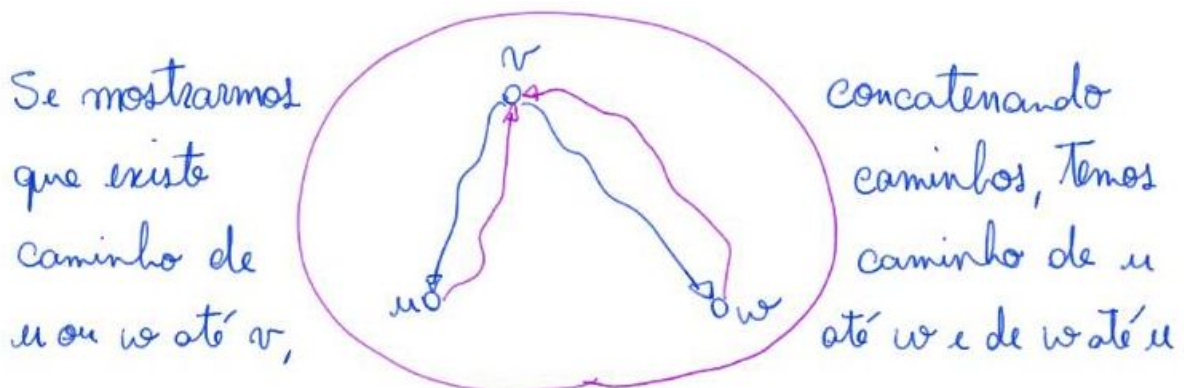
- que a busca a partir de um vértice v com maior tempo de término
 - realmente revela um componente fortemente conexo.

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 vértice w qualquer até v .



- Isso porque, 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 ,

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

$\exists w \rightsquigarrow v$ em G ?

Sabemos que em G o vértice v alcança w ,

$\exists v \rightsquigarrow w$ em G

- portanto em Grev existe um caminho de w até v .

$\exists w \rightsquigarrow v$ em Grev

Além disso, o tempo de término de v é maior que o tempo de término de w

- nas buscas realizadas em Grev no passo 1 do algoritmo.

$t[v] > t[w]$

Vamos analisar as possibilidades para que isso ocorra,

- i.e., para que $t[v]$ seja maior que $t[w]$.

Primeiro, vamos determinar qual vértice entre v e w

- foi visitado antes no passo 1 do algoritmo,
 - em que as buscas ocorrem em Grev.

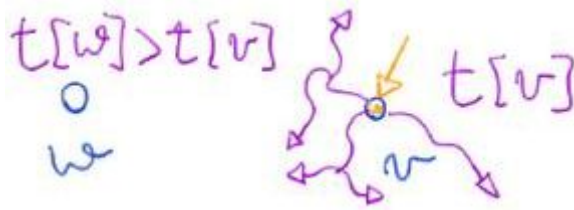
Caso 1: Suponha que w foi visitado antes que v em Grev.



- Neste caso, o tempo de término de w seria maior que de v ,
 - i.e., $t[w] > t[v]$, já que existe o caminho de w até v em Grev.
- Portanto, o caso 1 não pode ter ocorrido.

Caso 2: Consideramos que v foi visitado antes que w .

- Caso 2.1: Supomos que não há caminho de v até w em Grev.



- Neste caso, v será finalizado antes de w ser visitado e
 - quando w for finalizado receberá tempo de término maior que v ,
 - i.e., $t[w] > t[v]$.
 - Portanto, o caso 2.1 também não pode ter ocorrido.
- Caso 2.2: Assim, só nos resta concluir que há caminho de v até w em G_{rev} ,

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

- para que seja possível $t[v] > t[w]$.

Note que, isso implica na existência de um caminho de w até v em G ,

$\exists w \rightsquigarrow v$ em G

- que é o que queríamos demonstrar.

Quiz1: O que acontece se no passo 2 do algoritmo

- trocarmos a busca em profundidade que identifica os componentes
 - por uma busca genérica?
- Dica: alguma propriedade específica da busca em profundidade é usada?

Quiz2: O que acontece se fizermos o passo 1 do algoritmo no grafo original,

- e o passo 2 no grafo invertido?
 - O algoritmo ainda funciona?
- Dica: neste caso os tempos de término
 - vão identificar vértices de componentes fonte,
- mas o que serão esses componentes durante a busca em G_{rev} ?