Algoritmos e Estruturas de Dados II Aula 07 – Grafos: Busca em Profundidade - Aplicações (continuação)

Prof. Luciano A. Digiampietri
digiampietri@usp.br
@digiampietri

Aplicações da Busca em Profundidade (continuação)

Grafos – Busca em Profundidade

Aplicações da Busca em Profundidade (continuação)

A Busca em Profundidade por ser utilizada (ou adaptada) para ajudar a resolver uma série de problemas:

- Encontrar soluções para labirintos
- Verificar se um grafo é acíclico
- **Realizar a ordenação topológica**
- Encontrar componentes conexos/conectados
- Encontrar componentes fortemente conexos/conectados
- ...

Ordenação Topológica

- Em teoria dos grafos, uma ordenação topológica de um **grafo direcionado acíclico** (DAG) é uma ordem linear de seus vértices de forma que **cada vértice aparece antes de todos os seus adjacentes**.

Ordenação Topológica

- Em teoria dos grafos, uma ordenação topológica de um **grafo direcionado acíclico** (DAG) é uma ordem linear de seus vértices de forma que **cada vértice aparece antes de todos os seus adjacentes**.

- Grafos **cíclicos não possuem ordenação topológica possível**.

Ordenação Topológica

- Em teoria dos grafos, uma ordenação topológica de um **grafo direcionado acíclico** (DAG) é uma ordem linear de seus vértices de forma que **cada vértice aparece antes de todos os seus adjacentes**.

- Grafos **cíclicos não possuem ordenação topológica possível**.

- **Todo DAG possui uma ou mais ordenações topológicas**.

Ordenação Topológica

- É possível pensar em **diferentes formas** de realizar esta ordenação.

Ordenação Topológica

- É possível pensar em **diferentes formas** de realizar esta ordenação.

- Por exemplo, começar pelos vértices com grau de entrada igual a zero, remover cada um destes vértices e suas respectivas arestas e repetir o processo até não sobrarem vértices no grafo.

Ordenação Topológica

- É possível pensar em **diferentes formas** de realizar esta ordenação.
- Por exemplo, começar pelos vértices com grau de entrada igual a zero, remover cada um destes vértices e suas respectivas arestas e repetir o processo até não sobrarem vértices no grafo.
- Ou o contrário: colocar no final da lista os vértices com grau de saída igual a zero, remover cada um destes vértices e suas respectivas arestas e repetir o processo até não sobrarem vértices no grafo.

Ordenação Topológica

- É possível pensar em diferentes formas de realizar esta ordenação.

Ordenação Topológica

- É possível pensar em diferentes formas de realizar esta ordenação.

- Porém, estes procedimentos podem ser computacionalmente caros: para cada vértice com grau igual a zero (de entrada/saída), remover suas arestas, computar o grau de saída/entrada dos demais e repetir o processo até ordenar todos os vértices ....

Ordenação Topológica

- Podemos utilizar a **Busca em Profundidade para nos ajudar**:

Ordenação Topológica

- Podemos utilizar a **Busca em Profundidade para nos ajudar**:

- Quando **terminamos o processamento de um vértice** (e ele fica PRETO na busca), significa que não há mais nenhum vértice a ser visitado a partir dele.

Ordenação Topológica

- Podemos utilizar a **Busca em Profundidade para nos ajudar**:

- Quando **terminamos o processamento de um vértice** (e ele fica PRETO na busca), significa que não há mais nenhum vértice a ser visitado a partir dele.

- O **primeiro nó a se tornar PRETO** poderá ser o **último da ordenação**. Pelo mesmo raciocínio, o segundo nó PRETO poderá ser o penúltimo e assim por diante.

Ordenação Topológica

Desta forma, uma ordenação topológica possível para nosso DAG será aquela formada pelos **vértices ordenados de forma decrescente pelo tempo de término** da Busca em Profundidade.

Ordenação Topológica

Desta forma, uma ordenação topológica possível para nosso DAG será aquela formada pelos **vértices ordenados de forma decrescente pelo tempo de término** da Busca em Profundidade.

Faremos algumas alterações para que a busca **crie e retorne uma lista ligada com a ordenação topológica**.

Ordenação Topológica

Desta forma, uma ordenação topológica possível para nosso DAG será aquela formada pelos **vértices ordenados de forma decrescente pelo tempo de término** da Busca em Profundidade.

Faremos algumas alterações para que a busca **crie e retorne uma lista ligada com a ordenação topológica**.

Cada vértice que se tornar PRETO **será inserido no início da lista**. Este tipo de inserção, começando pelo primeiro término, produzirá **uma lista ordenada de forma decrescente pelo tempo de término**.

Ordenação Topológica

```
ElemLista* DFSOrdTopologica(Grafo* g){



}
```

Ordenação Topológica

```
ElemLista* DFSOrdTopologica(Grafo* g){
  ElemLista *inicio;
  ElemLista *cabeca = (ElemLista*) malloc(sizeof(ElemLista));
  cabeca->prox = NULL;









}
```

Ordenação Topológica

```
ElemLista* DFSOrdTopologica(Grafo* g){
  ElemLista *inicio;
  ElemLista *cabeca = (ElemLista*) malloc(sizeof(ElemLista));
  cabeca->prox = NULL;
  int* visitado = (int*) malloc(sizeof(int)*g->numVertices);
  int x;
  for (x=0;x<g->numVertices;x++) visitado[x] = false;









}
```

Ordenação Topológica

```
ElemLista* DFSOrdTopologica(Grafo* g){
  ElemLista *inicio;
  ElemLista *cabeca = (ElemLista*) malloc(sizeof(ElemLista));
  cabeca->prox = NULL;
  int* visitado = (int*) malloc(sizeof(int)*g->numVertices);
  int x;
  for (x=0;x<g->numVertices;x++) visitado[x] = false;
  for (x=0;x<g->numVertices;x++)
    if(!visitado[x]) DFSOrdTopAux(g, x, visitado, cabeca);




}
```

Ordenação Topológica

```
ElemLista* DFSOrdTopologica(Grafo* g){
  ElemLista *inicio;
  ElemLista *cabeca = (ElemLista*) malloc(sizeof(ElemLista));
  cabeca->prox = NULL;
  int* visitado = (int*) malloc(sizeof(int)*g->numVertices);
  int x;
  for (x=0;x<g->numVertices;x++) visitado[x] = false;
  for (x=0;x<g->numVertices;x++)
    if(!visitado[x]) DFSOrdTopAux(g, x, visitado, cabeca);
  inicio = cabeca->prox;
  free(cabeca);
  free(visitado);
  return inicio;
}
```

Ordenação Topológica

```
ElemLista* DFSOrdTopologica(Grafo* g){
  ElemLista *inicio;
  ElemLista *cabeca = (ElemLista*) malloc(sizeof(ElemLista));
  cabeca->prox = NULL;
  int* visitado = (int*) malloc(sizeof(int)*g->numVertices);
  int x;
  for (x=0;x<g->numVertices;x++) visitado[x] = false;
  for (x=0;x<g->numVertices;x++)
    if(!visitado[x]) DFSOrdTopAux(g, x, visitado, cabeca);
  inicio = cabeca->prox;
  free(cabeca);
  free(visitado);
  return inicio;
}
```

$O(V + E)$

Ordenação Topológica

```
void DFSOrdTopAux(Grafo* g, int atual, bool* visitado,
                                       ElemLista* cabeca){









}
```

Ordenação Topológica

```
void DFSOrdTopAux(Grafo* g, int atual, bool* visitado,
                                       ElemLista* cabeca){
  visitado[atual] = true;
  ElemLista *novo, *end = g->A[atual];
  while (end){
    if (!visitado[end->vertice])
      DFSOrdTopAux(g, end->vertice, visitado, cabeca);
    end = end->prox;
  }



}
```

Ordenação Topológica

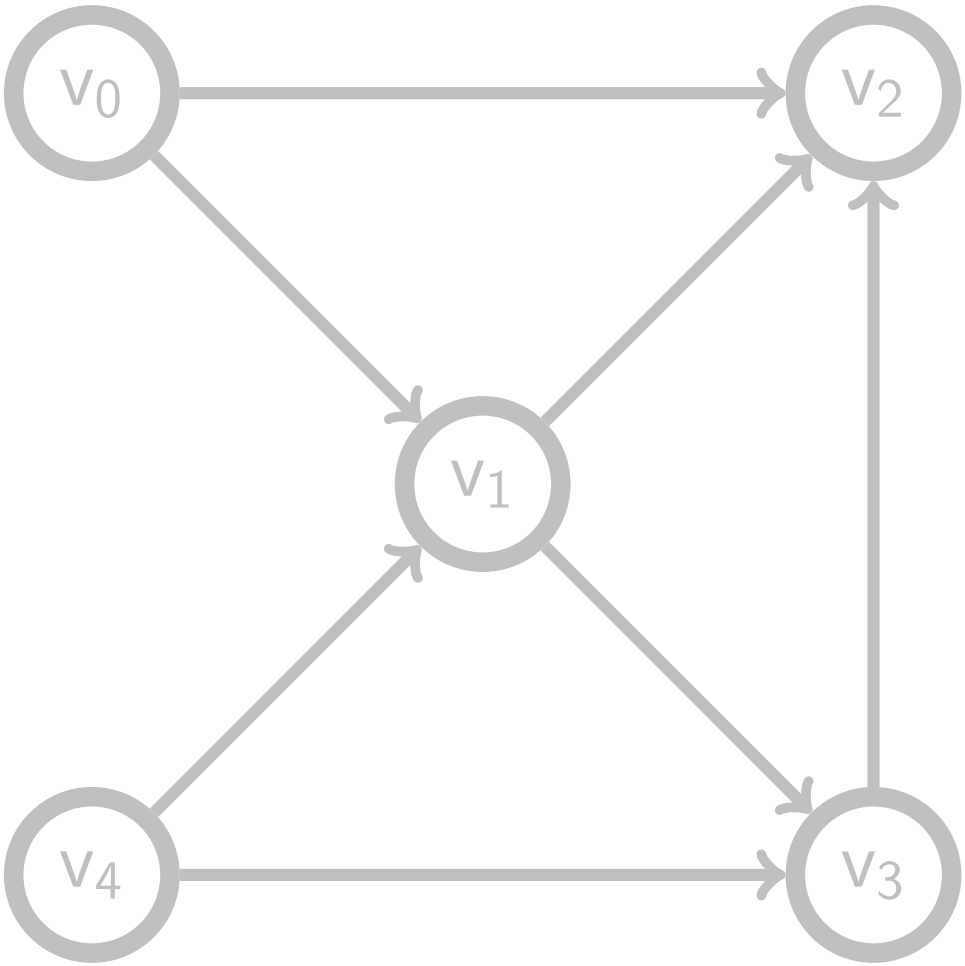
```
void DFSOrdTopAux(Grafo* g, int atual, bool* visitado,
                                       ElemLista* cabeca){
  visitado[atual] = true;
  ElemLista *novo, *end = g->A[atual];
  while (end){
    if (!visitado[end->vertice])
      DFSOrdTopAux(g, end->vertice, visitado, cabeca);
    end = end->prox;
  }
  novo = (ElemLista*) malloc(sizeof(ElemLista));
  novo->vertice = atual;
  novo->prox = cabeca->prox;
  cabeca->prox = novo;
}
```

Ordenação Topológica

```
void DFSOrdTopAux(Grafo* g, int atual, bool* visitado,
                                       ElemLista* cabeca){
  visitado[atual] = true;
  ElemLista *novo, *end = g->A[atual];
  while (end){
    if (!visitado[end->vertice])
      DFSOrdTopAux(g, end->vertice, visitado, cabeca);
    end = end->prox;
  }
  novo = (ElemLista*) malloc(sizeof(ElemLista));
  novo->vertice = atual;
  novo->prox = cabeca->prox;
  cabeca->prox = novo;
}
```

$O(V + E)$

Ordenação Topológica



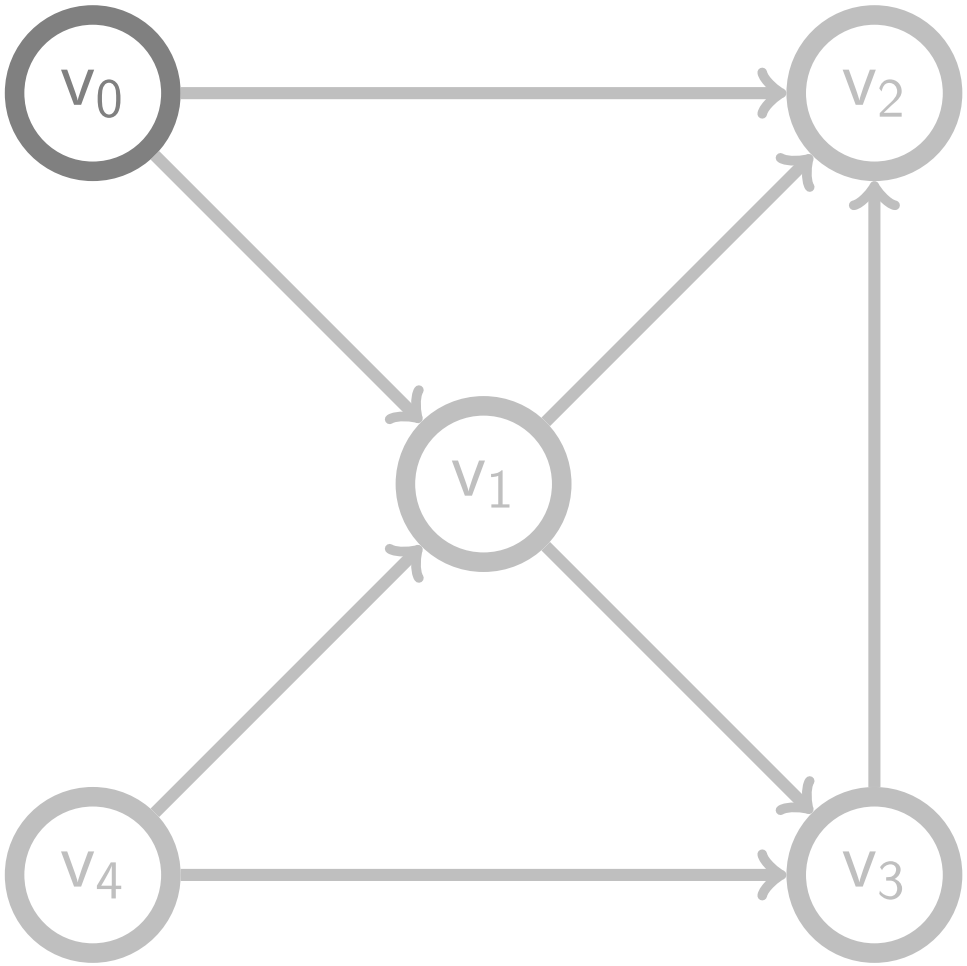
Resumo da Busca em Profundidade:

Nó	Anterior	Descoberta	Término	Cor:
0	-1	1	8	2
1	0	2	7	2
2	1	3	4	2
3	1	5	6	2
4	-1	9	10	2

Tempo: 0

Lista:

Ordenação Topológica



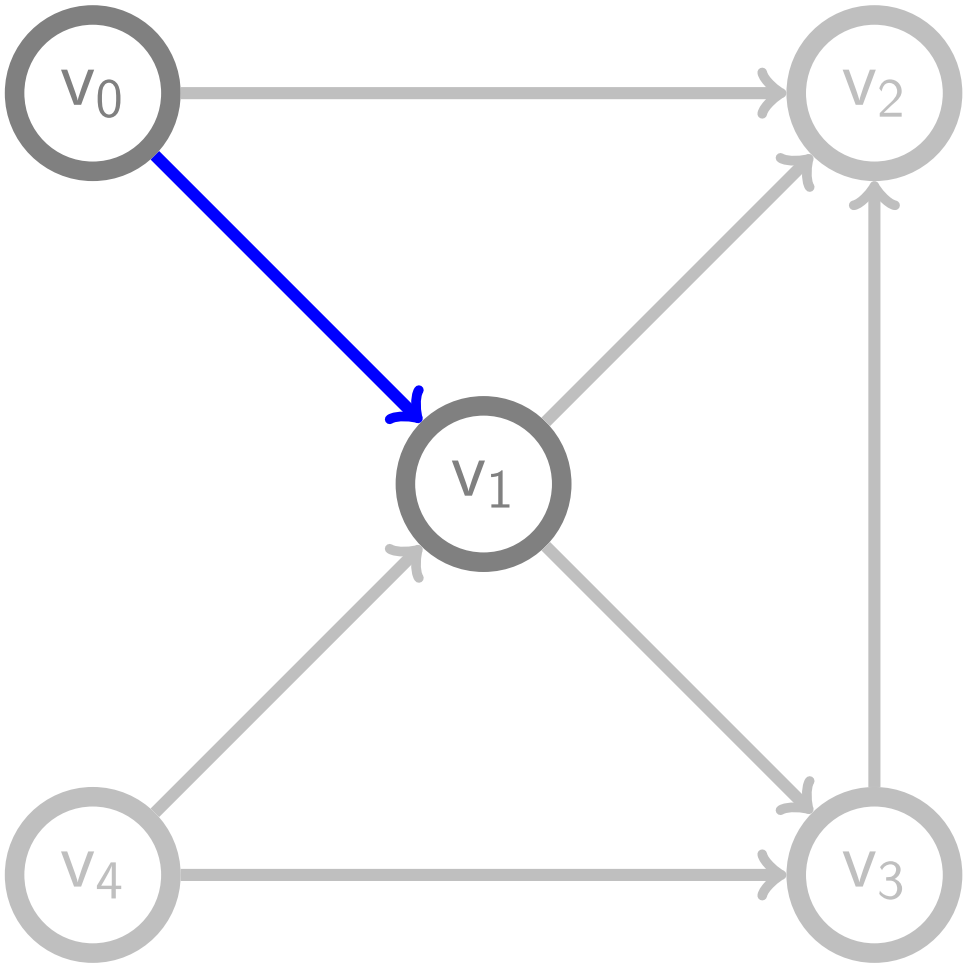
Resumo da Busca em Profundidade:

Nó	Anterior	Descoberta	Término	Cor:
0	-1	1	8	2
1	0	2	7	2
2	1	3	4	2
3	1	5	6	2
4	-1	9	10	2

Arestas de Árvore

Tempo: 1
Lista:

Ordenação Topológica



Resumo da Busca em Profundidade:

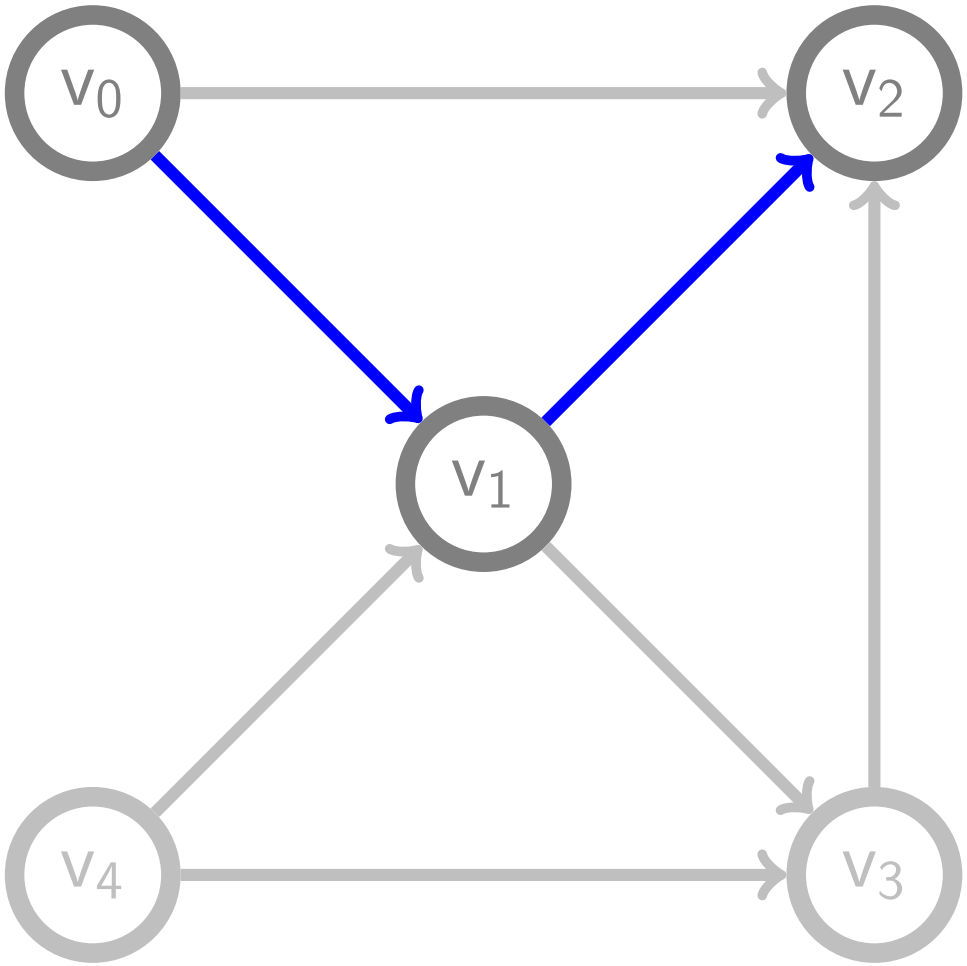
Nó	Anterior	Descoberta	Término	Cor:
0	-1	1	8	2
1	0	2	7	2
2	1	3	4	2
3	1	5	6	2
4	-1	9	10	2

Arestas de Árvore

Tempo: 2

Lista:

Ordenação Topológica



Resumo da Busca em Profundidade:

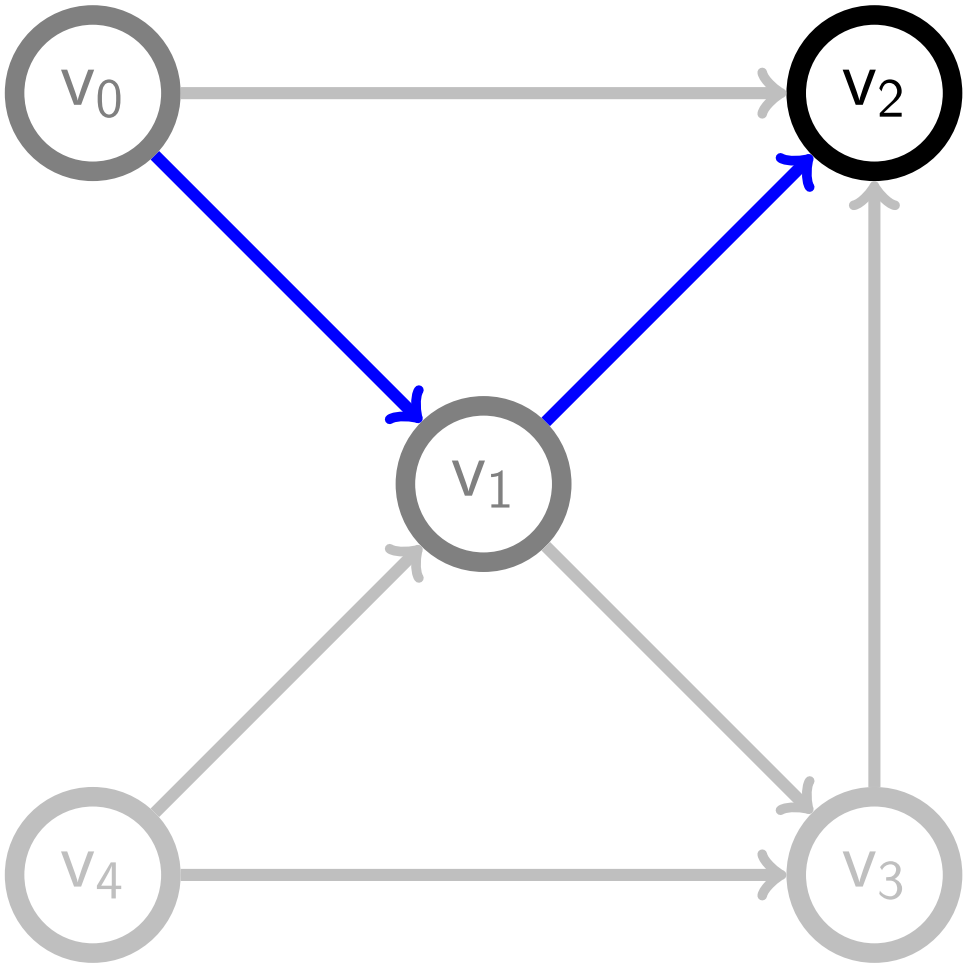
Nó	Anterior	Descoberta	Término	Cor:
0	-1	1	8	2
1	0	2	7	2
2	1	3	4	2
3	1	5	6	2
4	-1	9	10	2

Arestas de Árvore

Tempo: 3

Lista:

Ordenação Topológica



Resumo da Busca em Profundidade:

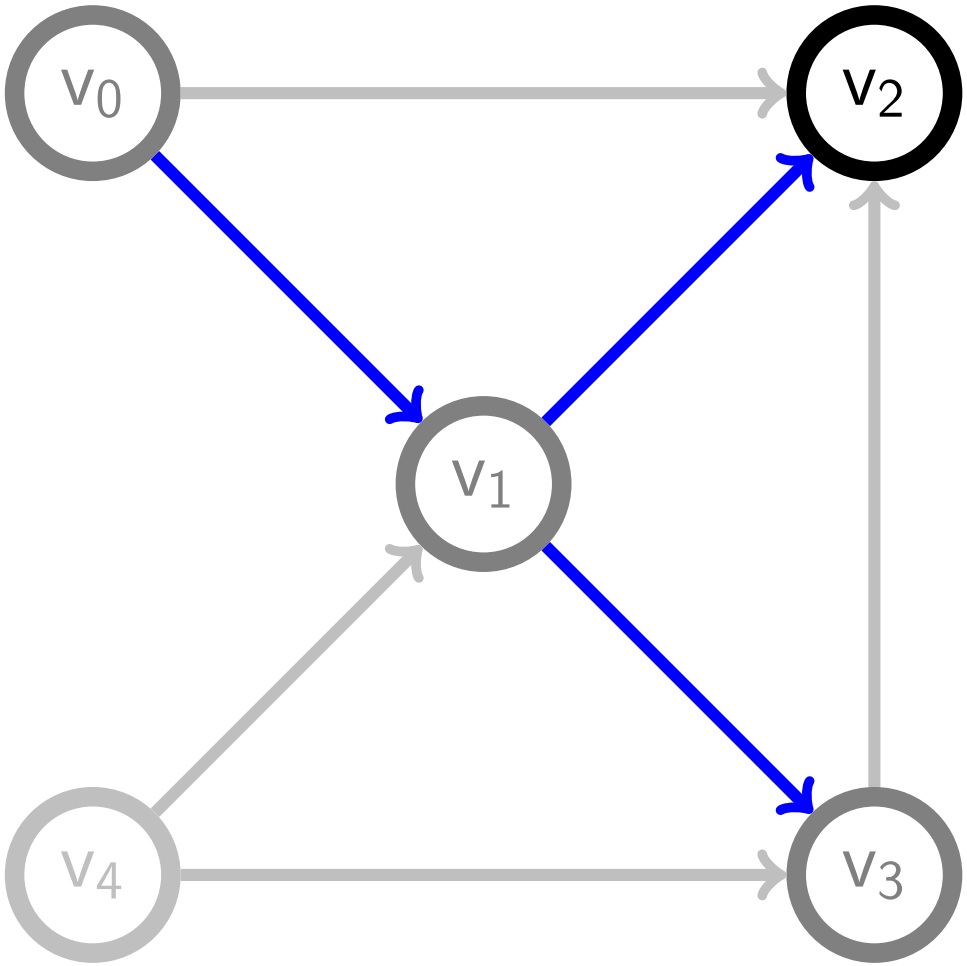
Nó	Anterior	Descoberta	Término	Cor:
0	-1	1	8	2
1	0	2	7	2
2	1	3	4	2
3	1	5	6	2
4	-1	9	10	2

Arestas de Árvore

Tempo: 4

Lista: v2

Ordenação Topológica



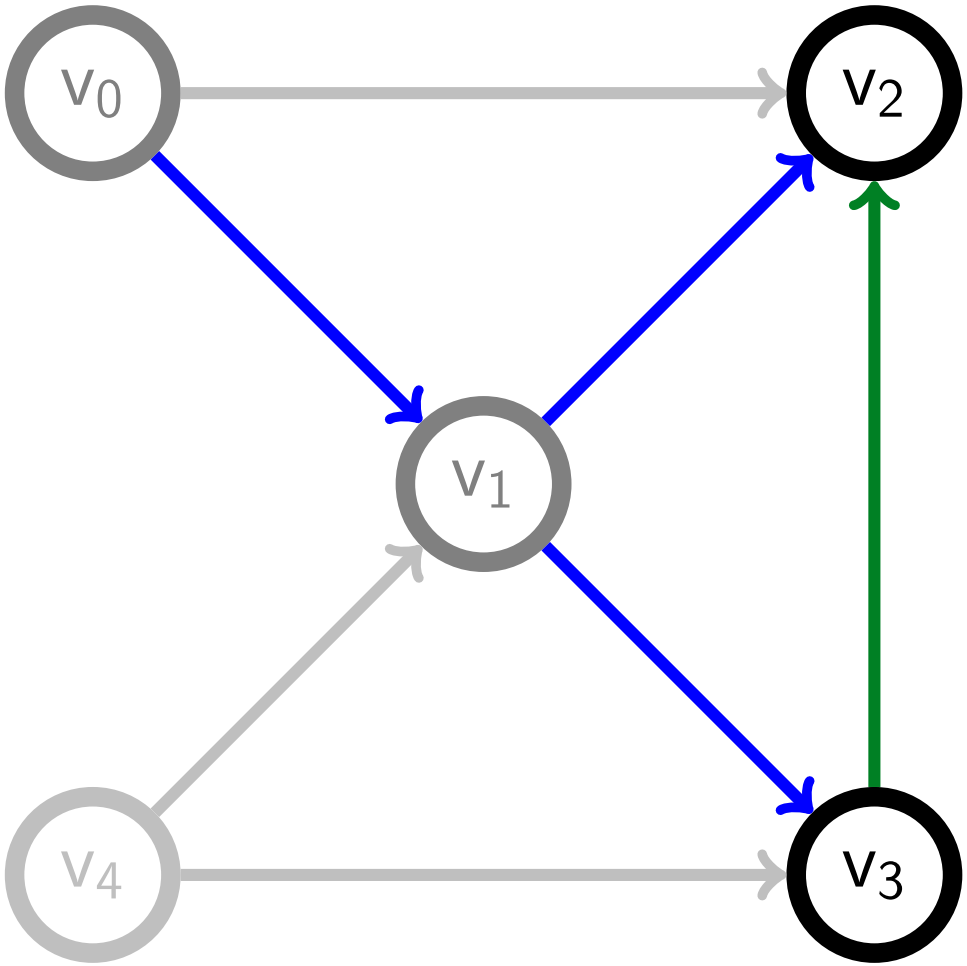
Resumo da Busca em Profundidade:

Nó	Anterior	Descoberta	Término	Cor:
0	-1	1	8	2
1	0	2	7	2
2	1	3	4	2
3	1	5	6	2
4	-1	9	10	2

Arestas de Árvore

Tempo: 5
Lista: v2

Ordenação Topológica



Resumo da Busca em Profundidade:

Nó	Anterior	Descoberta	Término	Cor:
0	-1	1	8	2
1	0	2	7	2
2	1	3	4	2
3	1	5	6	2
4	-1	9	10	2

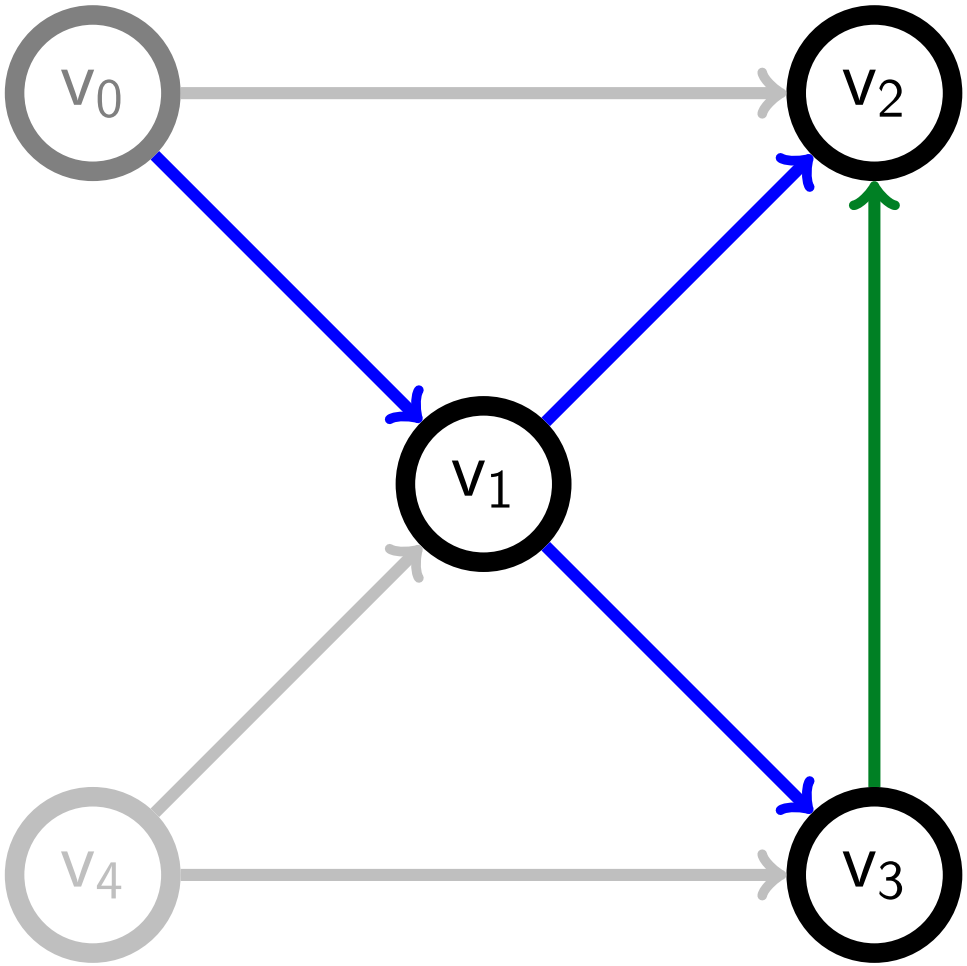
Arestas de Árvore

Arestas de Cruzamento

Tempo: 6

Lista: v3->v2

Ordenação Topológica



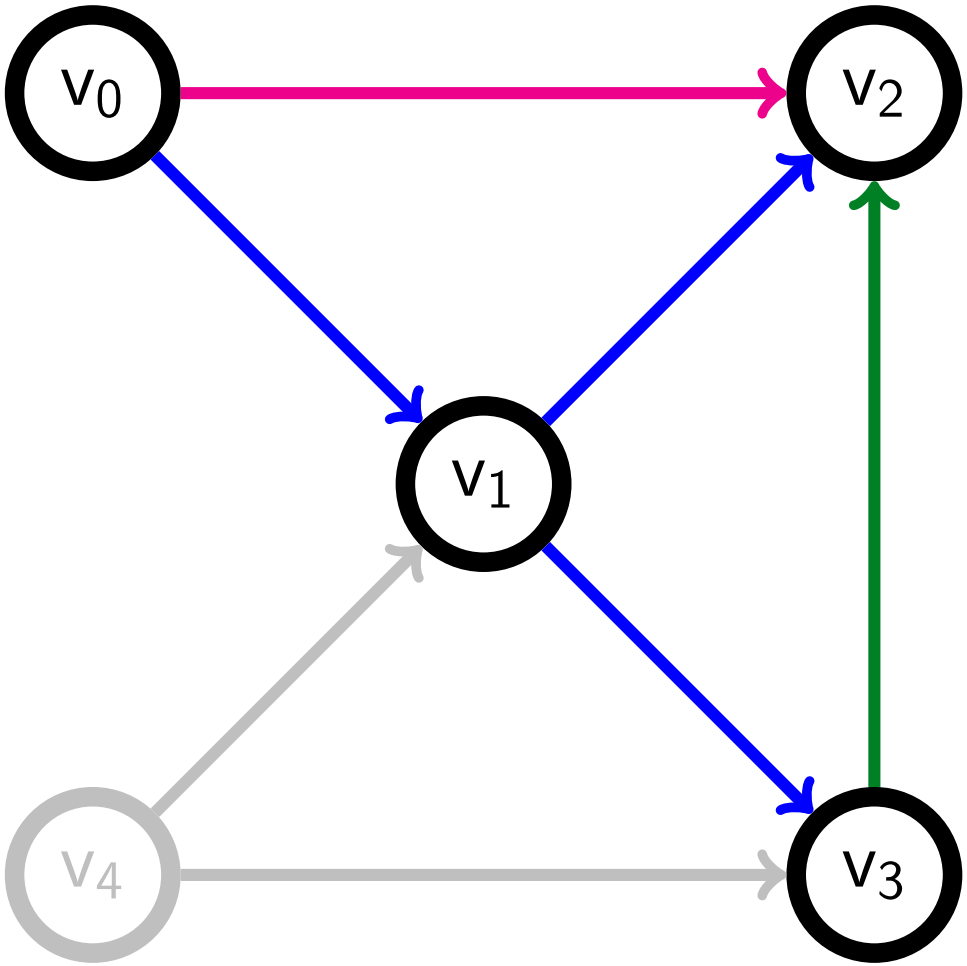

```
Resumo da Busca em Profundidade:
Nó Anterior Descoberta Término Cor:
 0        -1          1       8   2
 1         0          2       7   2
 2         1          3       4   2
 3         1          5       6   2
 4        -1          9      10   2
```

Arestas de Árvore

Arestas de Cruzamento

```
Tempo: 7
Lista: v1->v3->v2
```

Ordenação Topológica



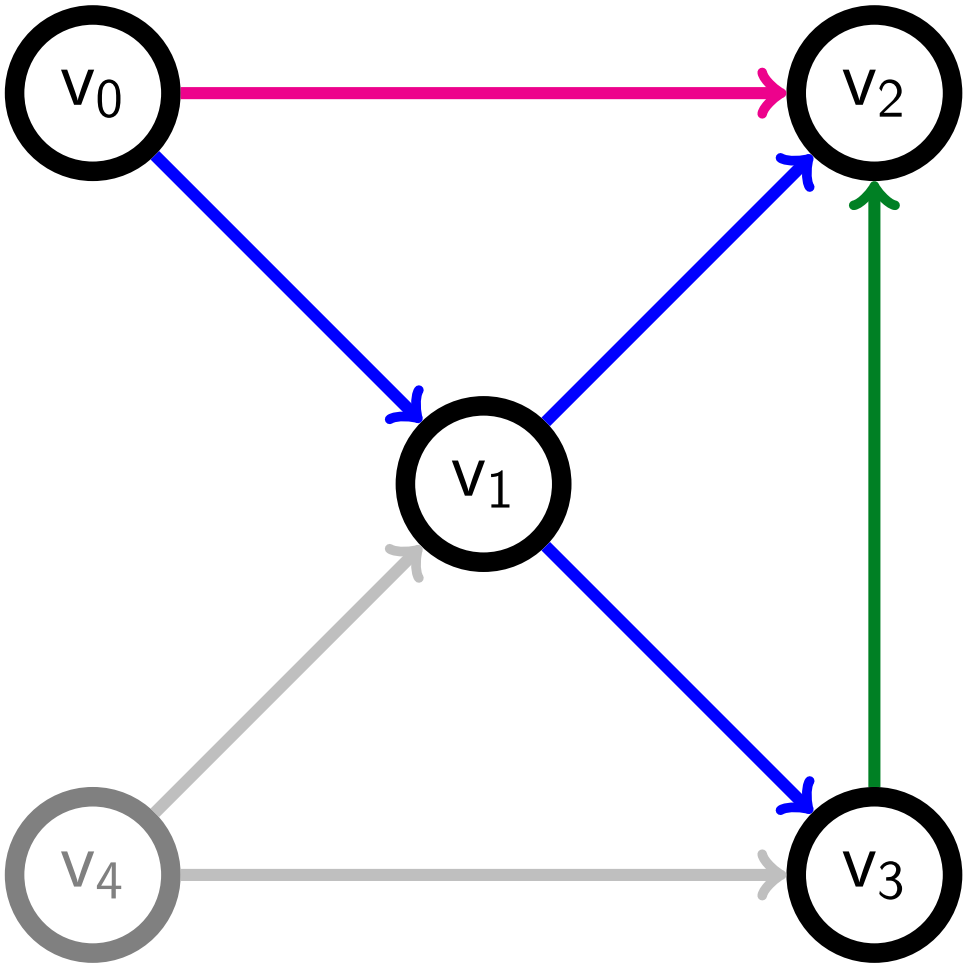
Resumo da Busca em Profundidade:

Nó	Anterior	Descoberta	Término	Cor:
0	-1	1	8	2
1	0	2	7	2
2	1	3	4	2
3	1	5	6	2
4	-1	9	10	2

Arestas de Árvore
Arestas de Cruzamento
Arestas de Avanço

Tempo: 8
Lista: v0->v1->v3->v2

Ordenação Topológica



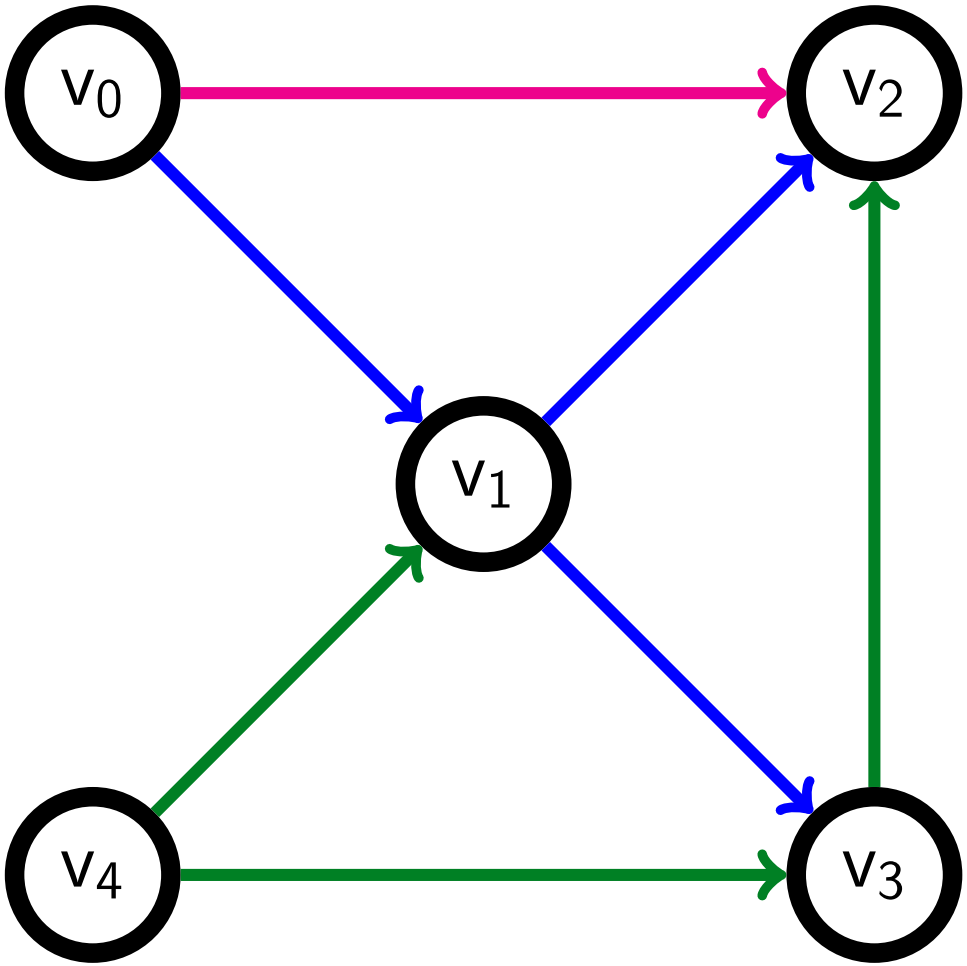
Resumo da Busca em Profundidade:

Nó	Anterior	Descoberta	Término	Cor:
0	-1	1	8	2
1	0	2	7	2
2	1	3	4	2
3	1	5	6	2
4	-1	9	10	2

Arestas de Árvore
Arestas de Cruzamento
Arestas de Avanço

Tempo: 9
Lista: v0->v1->v3->v2

Ordenação Topológica



Resumo da Busca em Profundidade:

Nó	Anterior	Descoberta	Término	Cor:
0	-1	1	8	2
1	0	2	7	2
2	1	3	4	2
3	1	5	6	2
4	-1	9	10	2

Arestas de Árvore
Arestas de Cruzamento
Arestas de Avanço

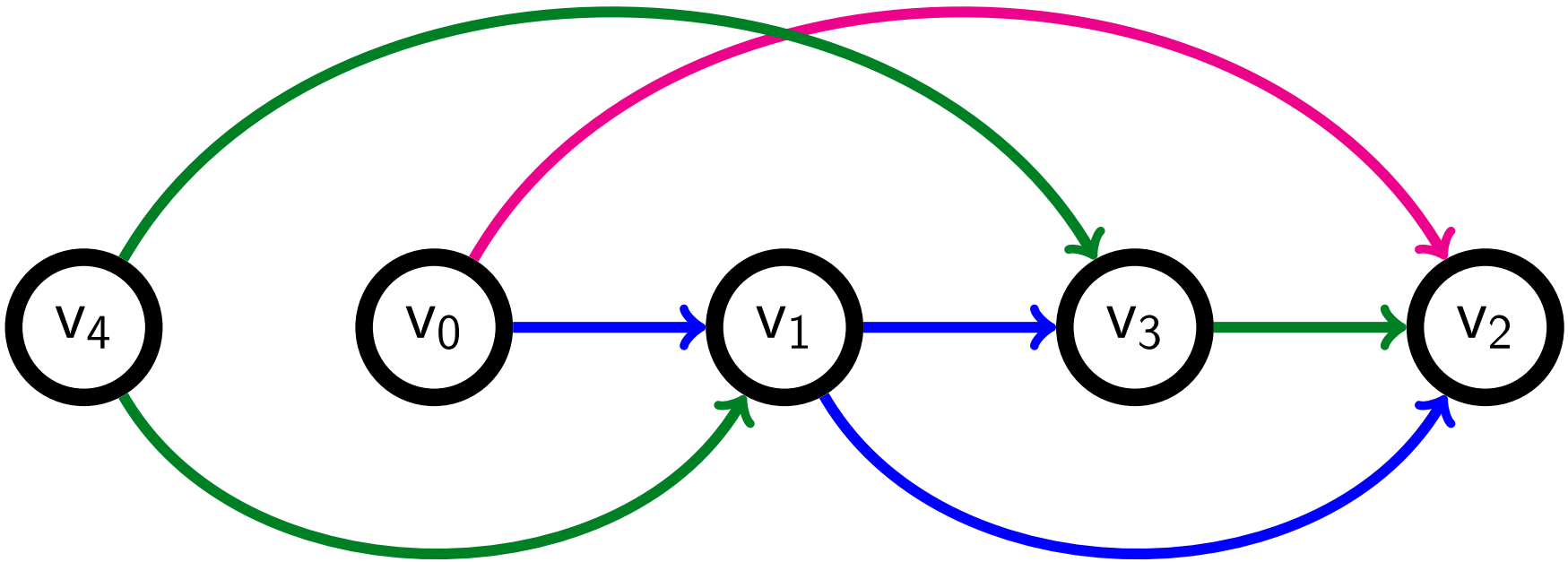
Tempo: 10
Lista: v4->v0->v1->v3->v2

Ordenação Topológica

```
Resumo da Busca em Profundidade:
Nó Anterior Descoberta Término Cor:
 0       -1          1       8   2
 1        0          2       7   2
 2        1          3       4   2
 3        1          5       6   2
 4       -1          9      10   2
```

Arestas de Árvore
Arestas de Cruzamento
Arestas de Avanço

Lista: v4->v0->v1->v3->v2



Grafos – Busca em Profundidade

Aplicações da Busca em Profundidade (continuação)

A Busca em Profundidade por ser utilizada (ou adaptada) para ajudar a resolver uma série de problemas:

- Encontrar soluções para labirintos
- Verificar se um grafo é acíclico
- Realizar a ordenação topológica
- **Encontrar componentes conexos/conectados**
- Encontrar componentes fortemente conexos/conectados
- ...

Encontrar componentes conexos em grafos não direcionados

Um grafo **não direcionado** é **conexo** (ou conectado) se cada par de seus vértices estiver conectado por um caminho.

Encontrar componentes conexos em grafos não direcionados

Um grafo **não direcionado** é **conexo** (ou conectado) se cada par de seus vértices estiver conectado por um caminho.

Um **componente conexo** de G é um **subgrafo** conexo maximal de G.

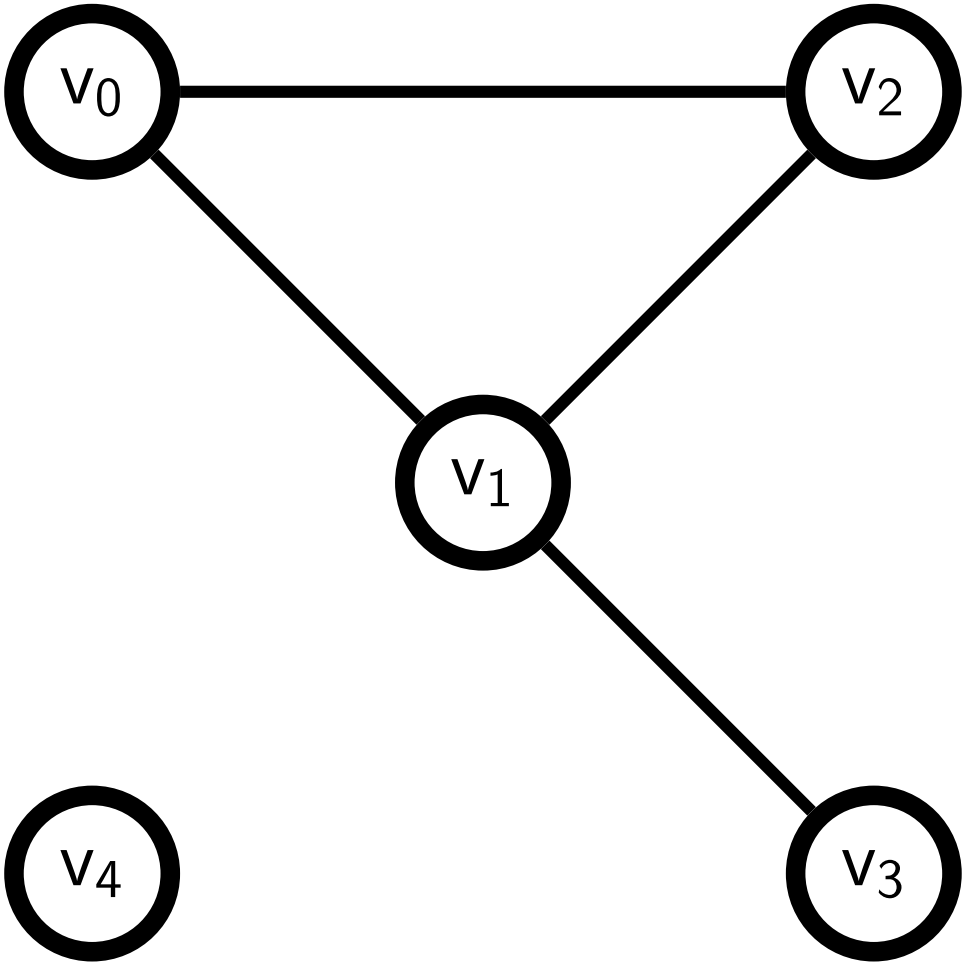
Encontrar componentes conexos em grafos não direcionados

Um grafo **não direcionado** é **conexo** (ou conectado) se cada par de seus vértices estiver conectado por um caminho.

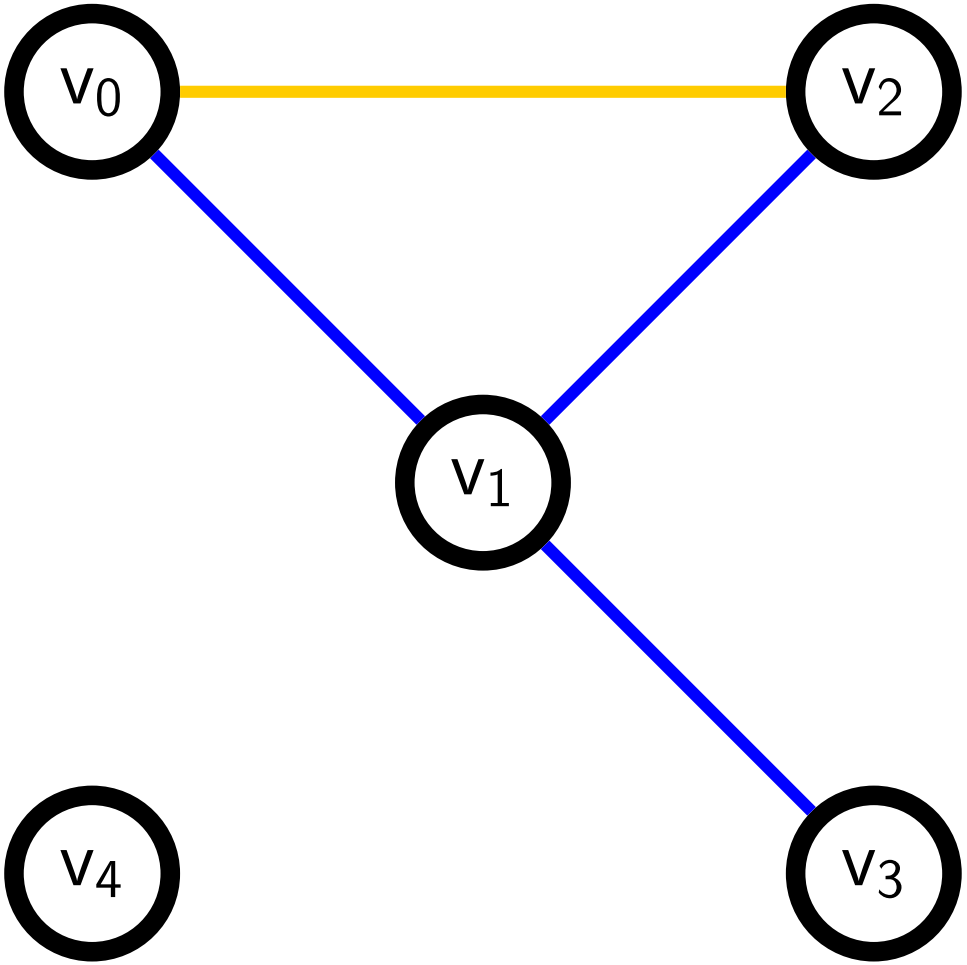
Um **componente conexo** de G é um **subgrafo** conexo maximal de G.

Cada vértice pertence a um **único componente** conexo.

Busca em Profundidade - Componentes



Busca em Profundidade - Componentes



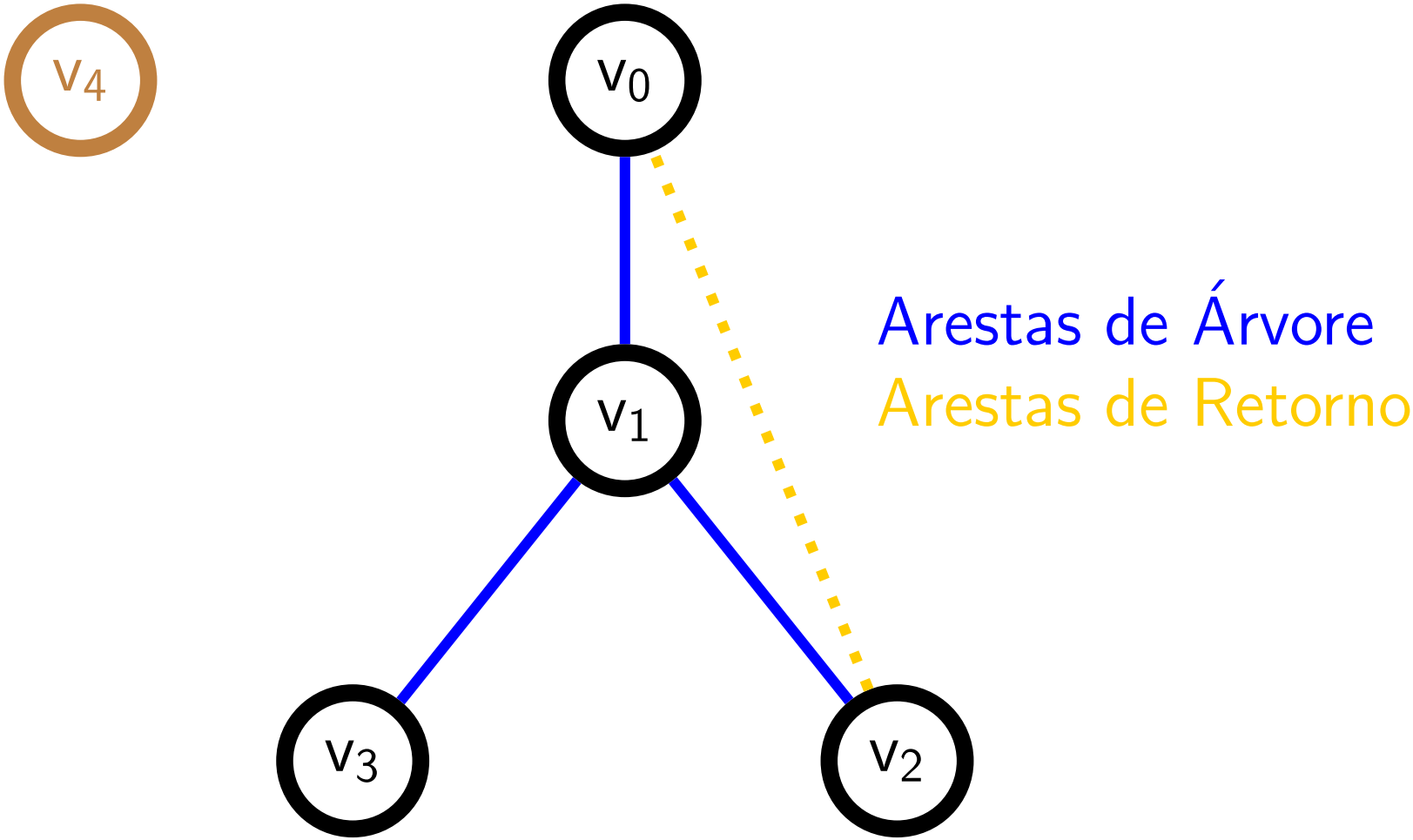
Resumo da Busca em Profundidade:

Nó	Anterior	Descoberta	Término	Cor:
0	-1	1	8	2
1	0	2	7	2
2	1	3	4	2
3	1	5	6	2
4	-1	9	10	2

Arestas de Árvore

Arestas de Retorno

Árvores de Busca em Profundidade



Cada árvore contém os vértices de um componente conexo.

Componentes Conexos - Grafo Não Direcionado

```
int* componentesConexos(Grafo* g){

}
```

Componentes Conexos - Grafo Não Direcionado

```
int* componentesConexos(Grafo* g){
  if (!g || g->numVertices<1) return NULL;
  int x, compAtual = 0;
  bool* visitado = malloc(sizeof(bool)*g->numVertices);
  int* componente = malloc(sizeof(int)*g->numVertices);
  for (x=0;x<g->numVertices;x++){
    visitado[x] = false;
    componente[x] = -1;
  }




}
```

Componentes Conexos - Grafo Não Direcionado

```
int* componentesConexos(Grafo* g){
  if (!g || g->numVertices<1) return NULL;
  int x, compAtual = 0;
  bool* visitado = malloc(sizeof(bool)*g->numVertices);
  int* componente = malloc(sizeof(int)*g->numVertices);
  for (x=0;x<g->numVertices;x++){
    visitado[x] = false;
    componente[x] = -1;
  }
  for (x=0;x<g->numVertices;x++)
    if (!visitado[x])
      visitaComponentes(g,x,visitado,
                        ++compAtual,componente);


}
```

Componentes Conexos - Grafo Não Direcionado

```
int* componentesConexos(Grafo* g){
  if (!g || g->numVertices<1) return NULL;
  int x, compAtual = 0;
  bool* visitado = malloc(sizeof(bool)*g->numVertices);
  int* componente = malloc(sizeof(int)*g->numVertices);
  for (x=0;x<g->numVertices;x++){
    visitado[x] = false;
    componente[x] = -1;
  }
  for (x=0;x<g->numVertices;x++)
    if (!visitado[x])
      visitaComponentes(g,x,visitado,
                        ++compAtual,componente);
  free(visitado);
  return componente;
}
```

Componentes Conexos - Grafo Não Direcionado

```
int* componentesConexos(Grafo* g){
  if (!g || g->numVertices<1) return NULL;
  int x, compAtual = 0;
  bool* visitado = malloc(sizeof(bool)*g->numVertices);
  int* componente = malloc(sizeof(int)*g->numVertices);
  for (x=0;x<g->numVertices;x++){
    visitado[x] = false;
    componente[x] = -1;
  }
  for (x=0;x<g->numVertices;x++)
    if (!visitado[x])
      visitaComponentes(g,x,visitado,
                        ++compAtual,componente);
  free(visitado);
  return componente;
}
```

$O(V + E)$

Componentes Conexos - Grafo Não Direcionado

```
void visitaComponentes(Grafo* g, int atual,
        bool* visitado, int compAtual, int* componente ){

}
```

Componentes Conexos - Grafo Não Direcionado

```
void visitaComponentes(Grafo* g, int atual,
        bool* visitado, int compAtual, int* componente ){
  visitado[atual] = true;
  componente[atual] = compAtual;
  printf("Nó: %3i, Comp.: %i)\n", atual, compAtual);




}
```

Componentes Conexos - Grafo Não Direcionado

```
void visitaComponentes(Grafo* g, int atual,
        bool* visitado, int compAtual, int* componente ){
  visitado[atual] = true;
  componente[atual] = compAtual;
  printf("Nó: %3i, Comp.: %i)\n", atual, compAtual);
  ElemLista* end = g->A[atual];
  while (end){




    end = end->prox;
  }
}
```

Componentes Conexos - Grafo Não Direcionado

```
void visitaComponentes(Grafo* g, int atual,
        bool* visitado, int compAtual, int* componente ){
  visitado[atual] = true;
  componente[atual] = compAtual;
  printf("Nó: %3i, Comp.: %i)\n", atual, compAtual);
  ElemLista* end = g->A[atual];
  while (end){
    if (!visitado[end->vertice])
      visitaComponentes(g, end->vertice, visitado,
                                    compAtual, componente );
    end = end->prox;
  }
}
```

Componentes Conexos - Grafo Não Direcionado

```
void visitaComponentes(Grafo* g, int atual,
        bool* visitado, int compAtual, int* componente ){
  visitado[atual] = true;
  componente[atual] = compAtual;
  printf("Nó: %3i, Comp.: %i)\n", atual, compAtual);
  ElemLista* end = g->A[atual];
  while (end){
    if (!visitado[end->vertice])
      visitaComponentes(g, end->vertice, visitado,
                                  compAtual, componente );
    end = end->prox;
  }
}
```

$O(V + E)$

Encontrar componentes fortemente conexos em grafos direcionados

Um grafo **direcionado** é **fortemente conexo** se cada par de seus vértices estiver conectado por um caminho de ida e também de volta.

Encontrar componentes fortemente conexos em grafos direcionados

Um grafo **direcionado** é **fortemente conexo** se cada par de seus vértices estiver conectado por um caminho de ida e também de volta.

Um **componente fortemente conexo** de G é um **subgrafo** fortemente conexo maximal de G.

Encontrar componentes fortemente conexos em grafos direcionados

Para nossa solução, precisaremos utilizar o **grafo transposto** de nosso grafo G.

Encontrar componentes fortemente conexos em grafos direcionados

Para nossa solução, precisaremos utilizar o **grafo transposto** de nosso grafo G.

O grafo transposto G^T de G é um grafo que possui os **mesmos vértices de G**, porém todas as **suas arestas têm sua direção invertida**.

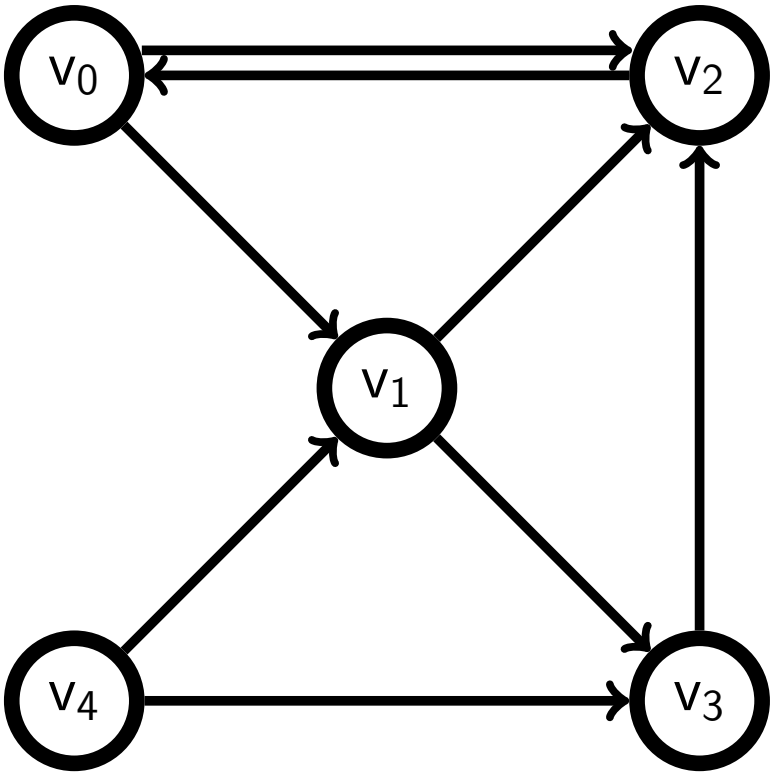
Encontrar componentes fortemente conexos em grafos direcionados

Para nossa solução, precisaremos utilizar o **grafo transposto** de nosso grafo G.

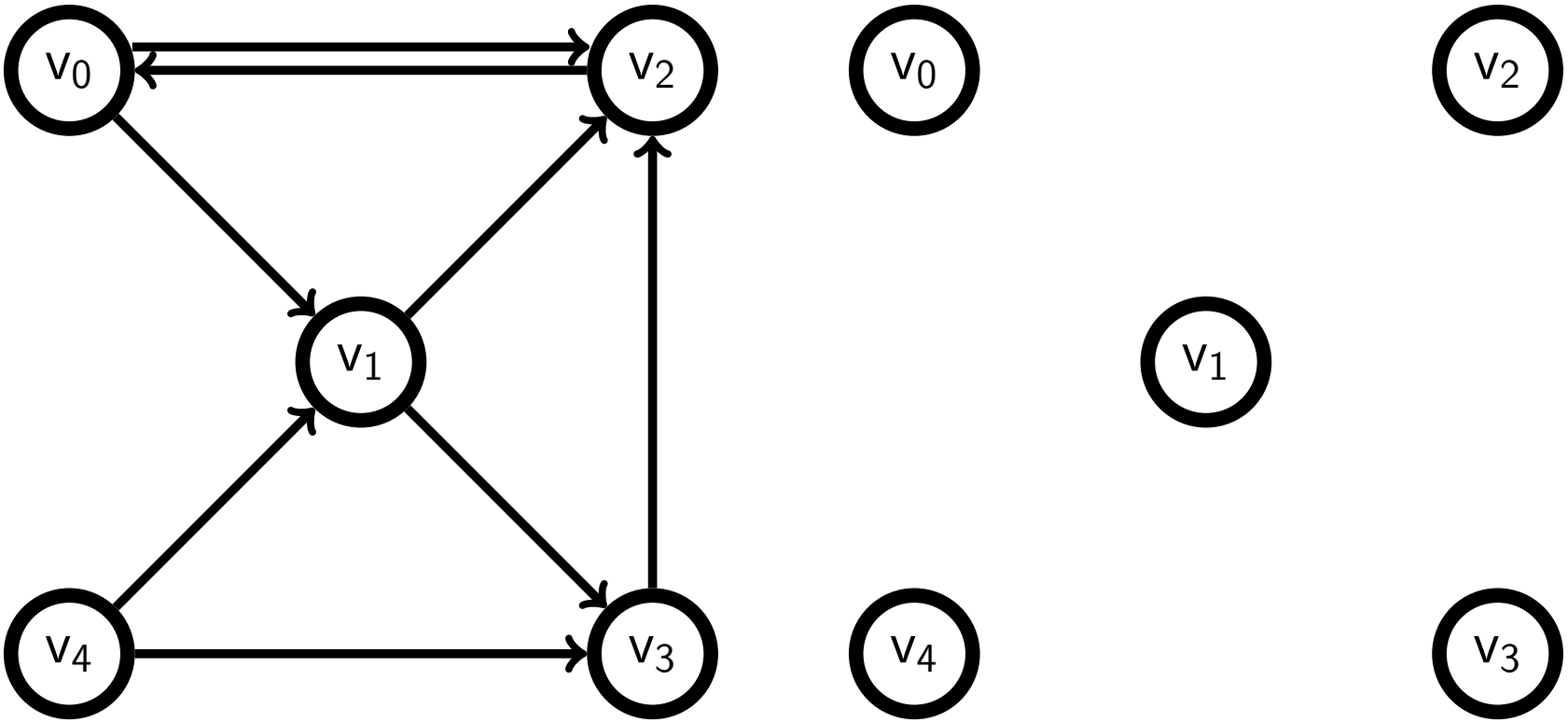
O grafo transposto G^T de G é um grafo que possui os **mesmos vértices de G**, porém todas as **suas arestas têm sua direção invertida**.

G e G^T possuem os **mesmos componentes fortemente conexos**.

Grafo Transposto

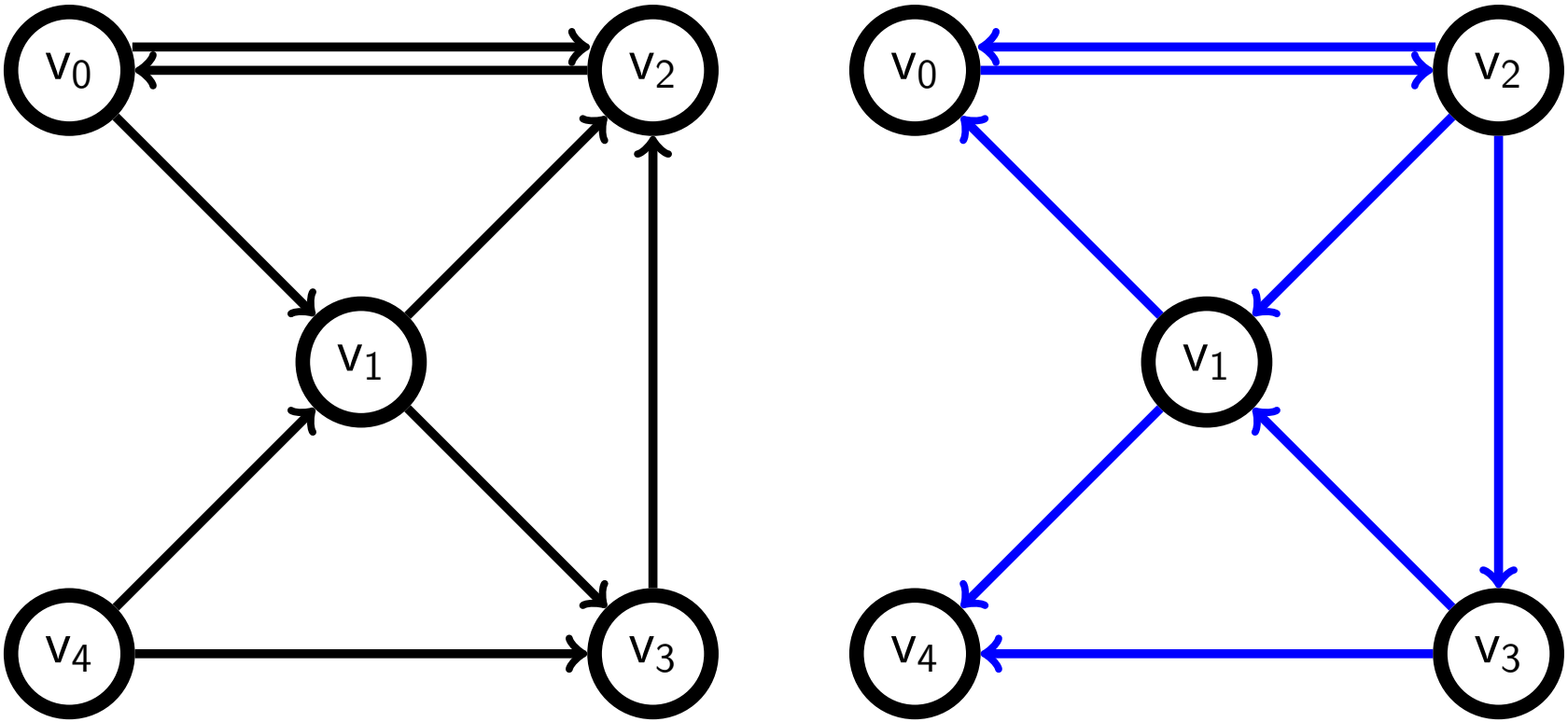


Grafo Transposto



Mesmos vértices.

Grafo Transposto



Mesmos vértices.

Arestas com direções invertidas.

Grafo Transposto

```
Grafo* transpoeGrafo(Grafo* g){
  if (!g) return NULL;
  Grafo* gt = (Grafo*)malloc(sizeof(Grafo));
  inicializaGrafo(gt,g->numVertices);
  ElemLista* atual;
  int x;
  for (x=0;x<g->numVertices;x++){
    atual = g->A[x];
    while (atual){
      insereAresta(gt, atual->vertice, x, atual->peso);
      atual = atual->prox;
    }
  }
  return gt;
}
```

$$O(V + E * O(insereAresta))$$

Grafo Transposto

```
ElemLista* novoElemento2(int vertice, Peso peso,
                                        ElemLista* prox) {
  ElemLista* novo=(ElemLista*)malloc(sizeof(ElemLista));
  novo->vertice = vertice;
  novo->peso = peso;
  novo->prox = prox;
  return novo;
}
```

O(1)

Grafo Transposto

```
Grafo* transpoeGrafo2(Grafo* g){
  if (!g) return NULL;
  Grafo* gt = (Grafo*)malloc(sizeof(Grafo));
  inicializaGrafo(gt,g->numVertices);
  ElemLista* atual;
  int x;
  for (x=g->numVertices-1;x>=0;x--){
    atual = g->A[x];
    while (atual){
      gt->A[atual->vertice] = novoElemento2(x,
                  atual->peso,gt->A[atual->vertice]);
      atual = atual->prox;
    }
  }
  return gt;
}
```

$O(V + E)$

Encontrar componentes fortemente conexos em grafos direcionados

G e G^T possuem os **mesmos componentes fortemente conexos**.

Encontrar componentes fortemente conexos em grafos direcionados

G e G^T possuem os **mesmos componentes fortemente conexos**.

Mas como encontrar esses componentes?

Encontrar componentes fortemente conexos em grafos direcionados

1. **Chamamos a Busca em Profundidade em G** para calcular os tempos de término de cada vértice[1];

[1] Tínhamos utilizado esses tempos para a ordenação topológica, mas aqui os grafos podem ser cíclicos.

Encontrar componentes fortemente conexos em grafos direcionados

1. **Chamamos a Busca em Profundidade em G** para calcular os tempos de término de cada vértice[1];
2. **Obtemos** G^T;

[1] Tínhamos utilizado esses tempos para a ordenação topológica, mas aqui os grafos podem ser cíclicos.

Encontrar componentes fortemente conexos em grafos direcionados

1. **Chamamos a Busca em Profundidade em G** para calcular os tempos de término de cada vértice[1];
2. **Obtemos** G^T;
3. Chamamos a **Busca em Profundidade em** G^T, mas visitando os vértices, recursivamente, a partir daqueles com maior tempo de término.

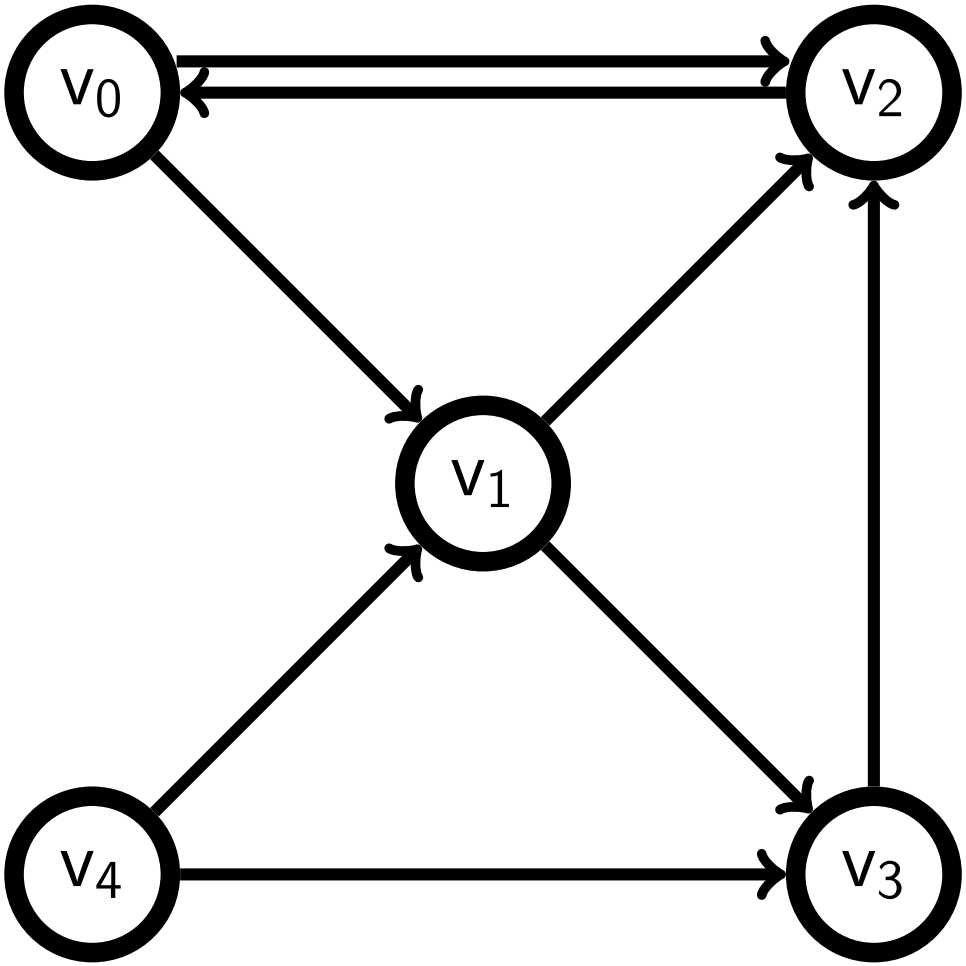
[1] Tínhamos utilizado esses tempos para a ordenação topológica, mas aqui os grafos podem ser cíclicos.

Encontrar componentes fortemente conexos em grafos direcionados

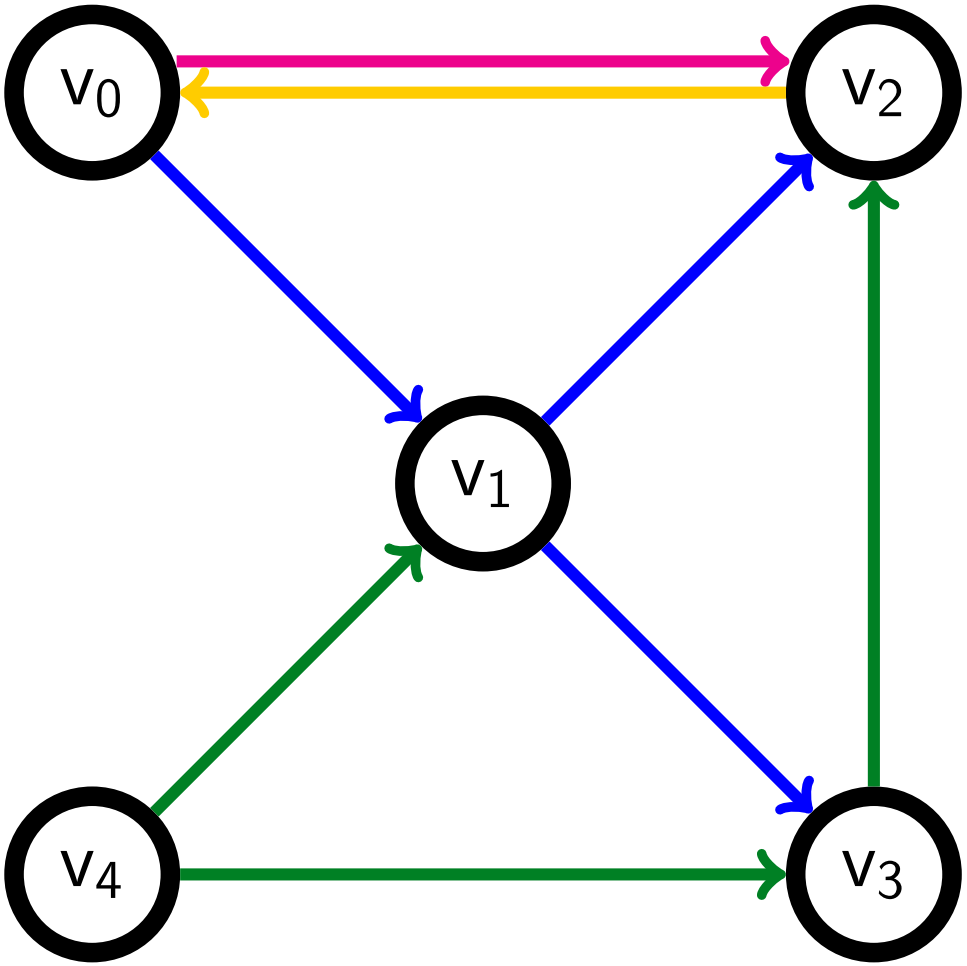
1. **Chamamos a Busca em Profundidade em G** para calcular os tempos de término de cada vértice[1];
2. **Obtemos** G^T;
3. Chamamos a **Busca em Profundidade em** G^T, mas visitando os vértices, recursivamente, a partir daqueles com maior tempo de término.
4. Cada uma das **árvores de busca obtidas na busca em** G^T terá os nós de um componente fortemente conexo.

[1] Tínhamos utilizado esses tempos para a ordenação topológica, mas aqui os grafos podem ser cíclicos.

Componentes Fortemente Conexos - DFS(G)



Componentes Fortemente Conexos - DFS(G)



Resumo da Busca em Profundidade:

Nó	Anterior	Descoberta	Término	Cor:
0	-1	1	8	2
1	0	2	7	2
2	1	3	4	2
3	1	5	6	2
4	-1	9	10	2

Arestas de Árvore

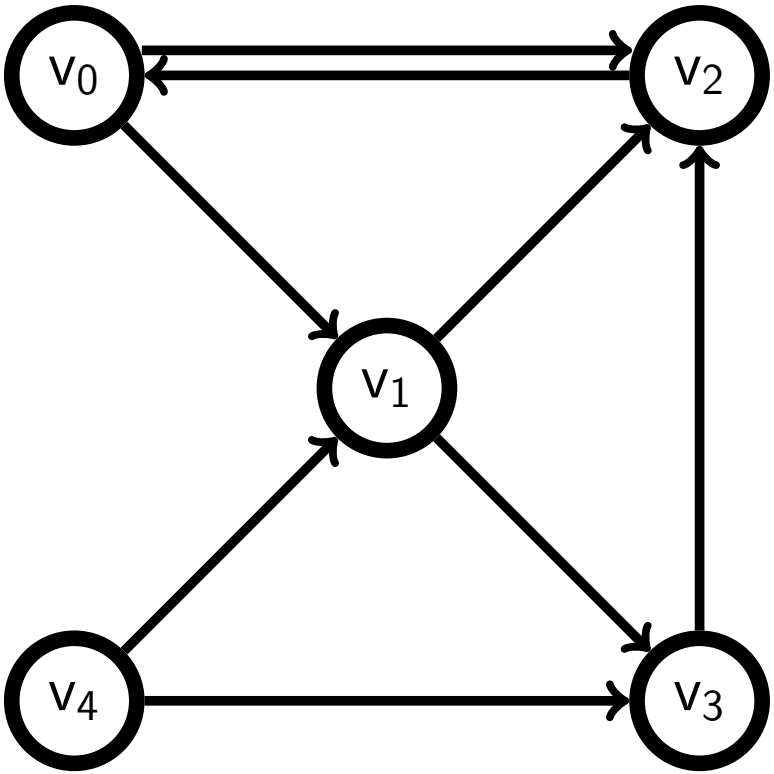
Arestas de Retorno

Arestas de Cruzamento

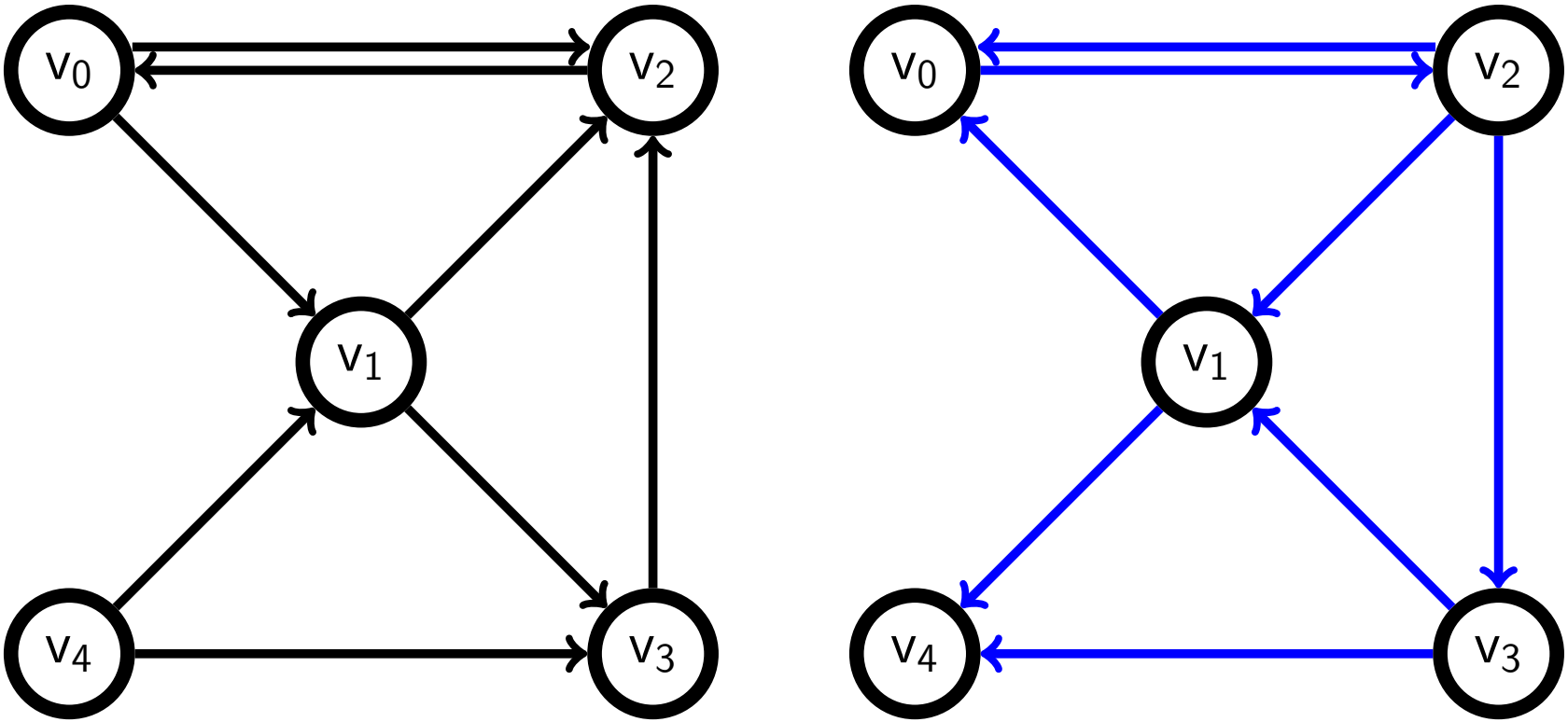
Arestas de Avanço

Lista: v4->v0->v1->v3->v2

Componentes Fortemente Conexos - Transpor



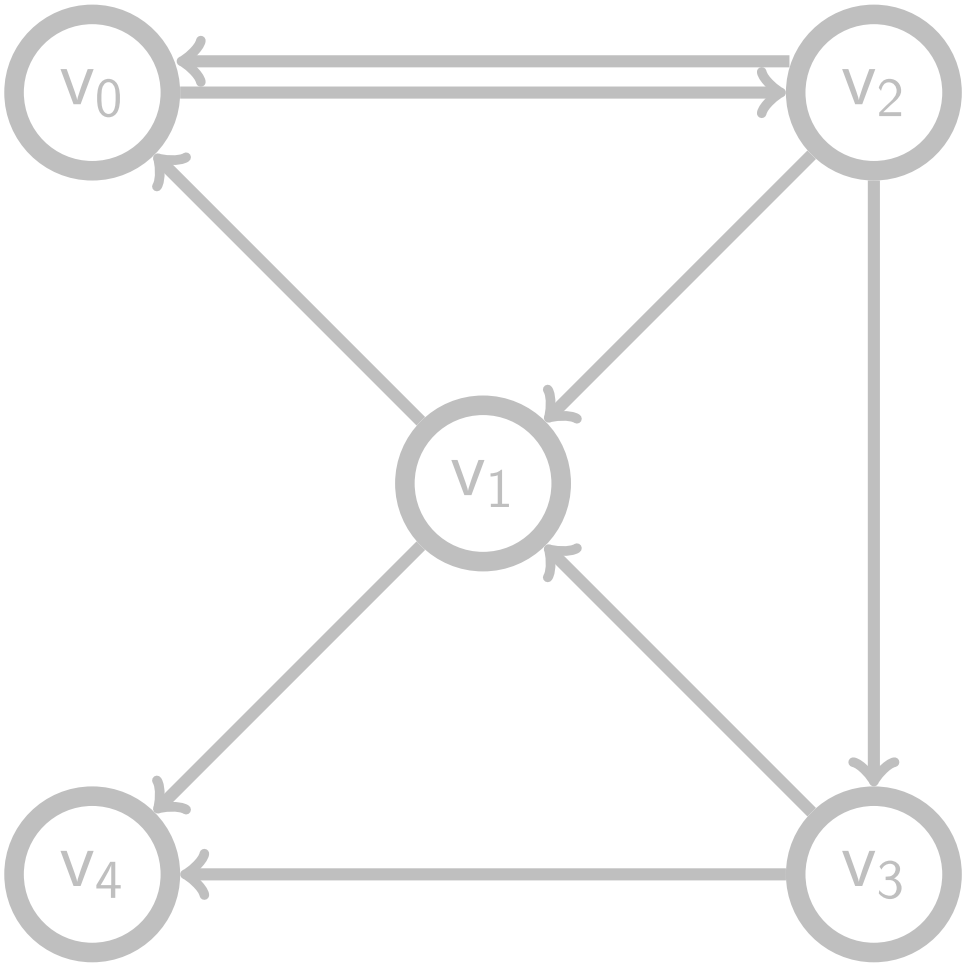
Componentes Fortemente Conexos - Transpor



Mesmos vértices.

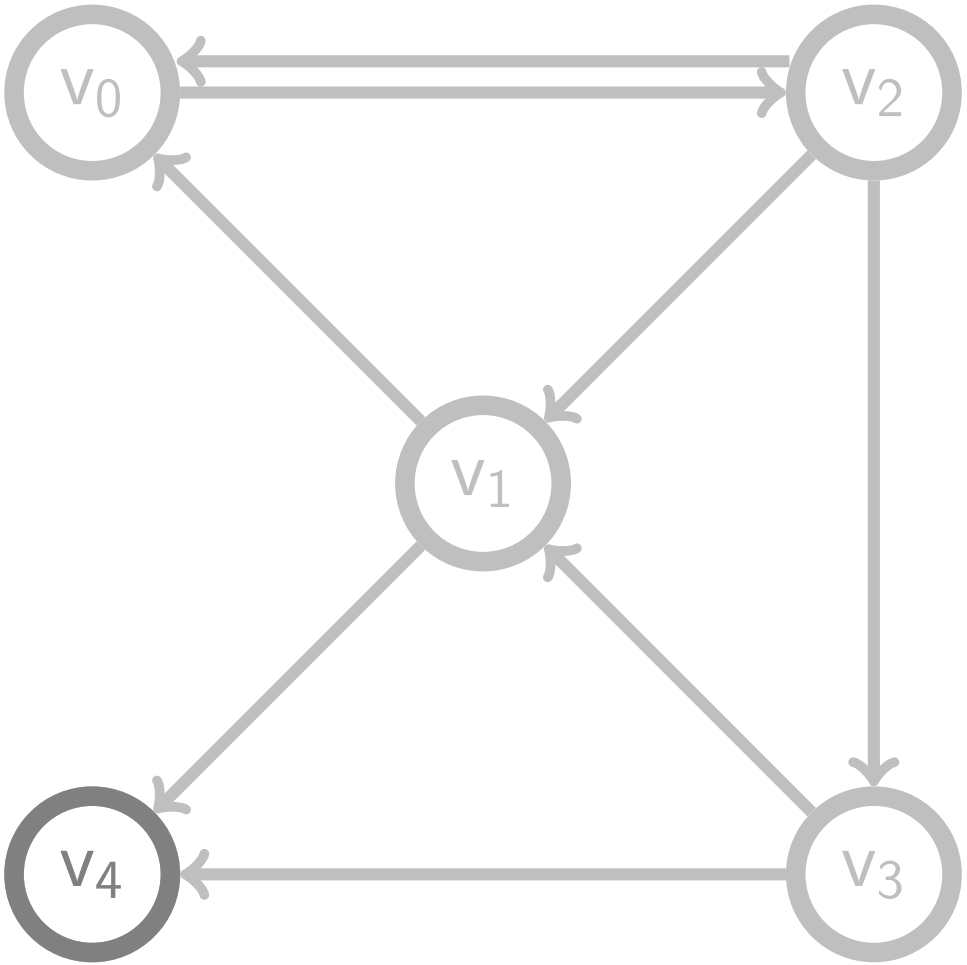
Arestas com direções invertidas.

Componentes Fortemente Conexos - DFS(G^T)*




```
Ordem: tTérmino
 (decrescente)
v4->v0->v1->v3->v2
```

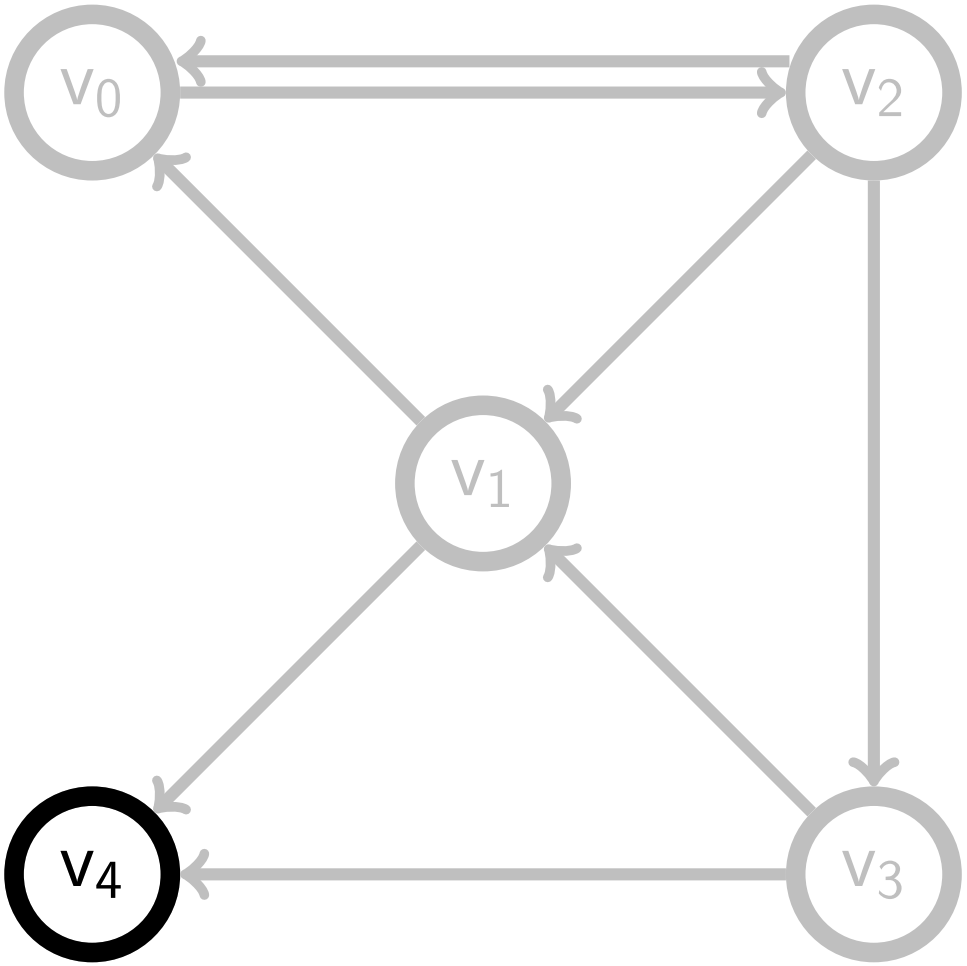
Componentes Fortemente Conexos - DFS(G^T)*




```
Ordem: tTérmino
 (decrescente)
v4->v0->v1->v3->v2
```

Arestas de Árvore
Arestas de Retorno
Arestas de Cruzamento
Arestas de Avanço

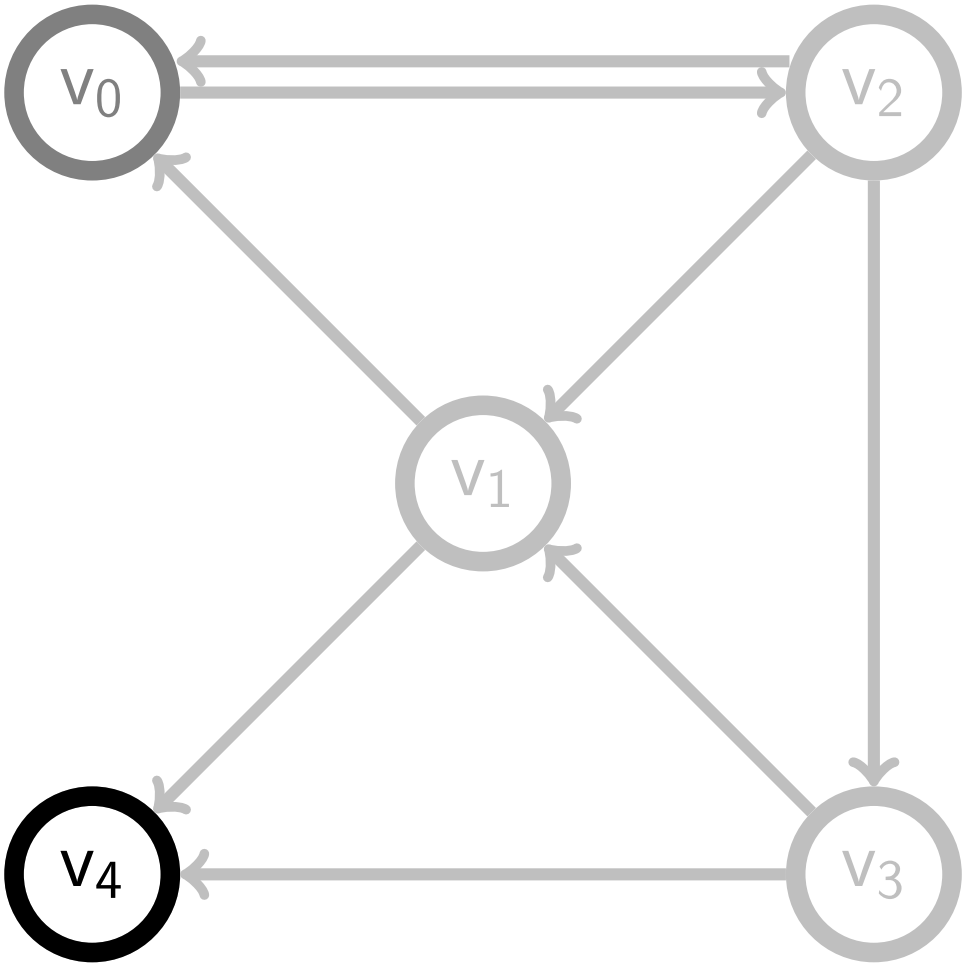
Componentes Fortemente Conexos - DFS(G^T)*



Ordem: tTérmino
(decrescente)
v4->v0->v1->v3->v2

Arestas de Árvore
Arestas de Retorno
Arestas de Cruzamento
Arestas de Avanço

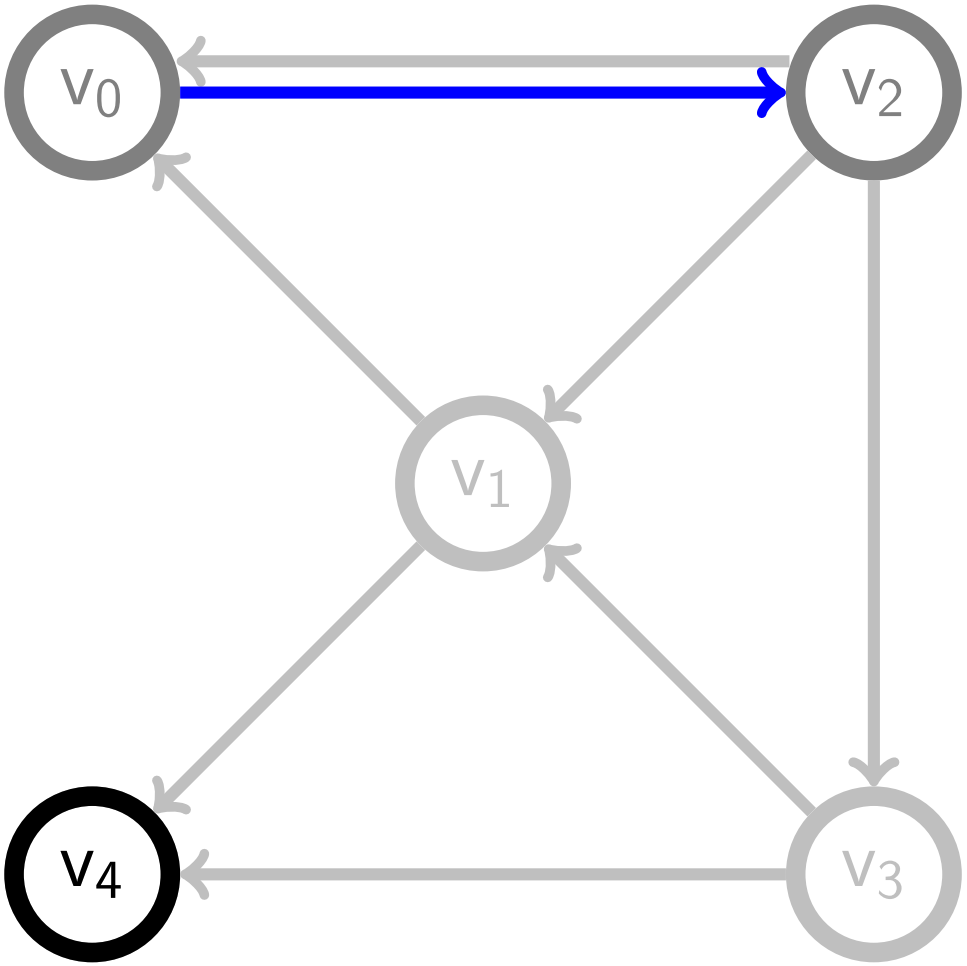
Componentes Fortemente Conexos - DFS(G^T)*



Ordem: tTérmino
(decrescente)
v4->v0->v1->v3->v2

Arestas de Árvore
Arestas de Retorno
Arestas de Cruzamento
Arestas de Avanço

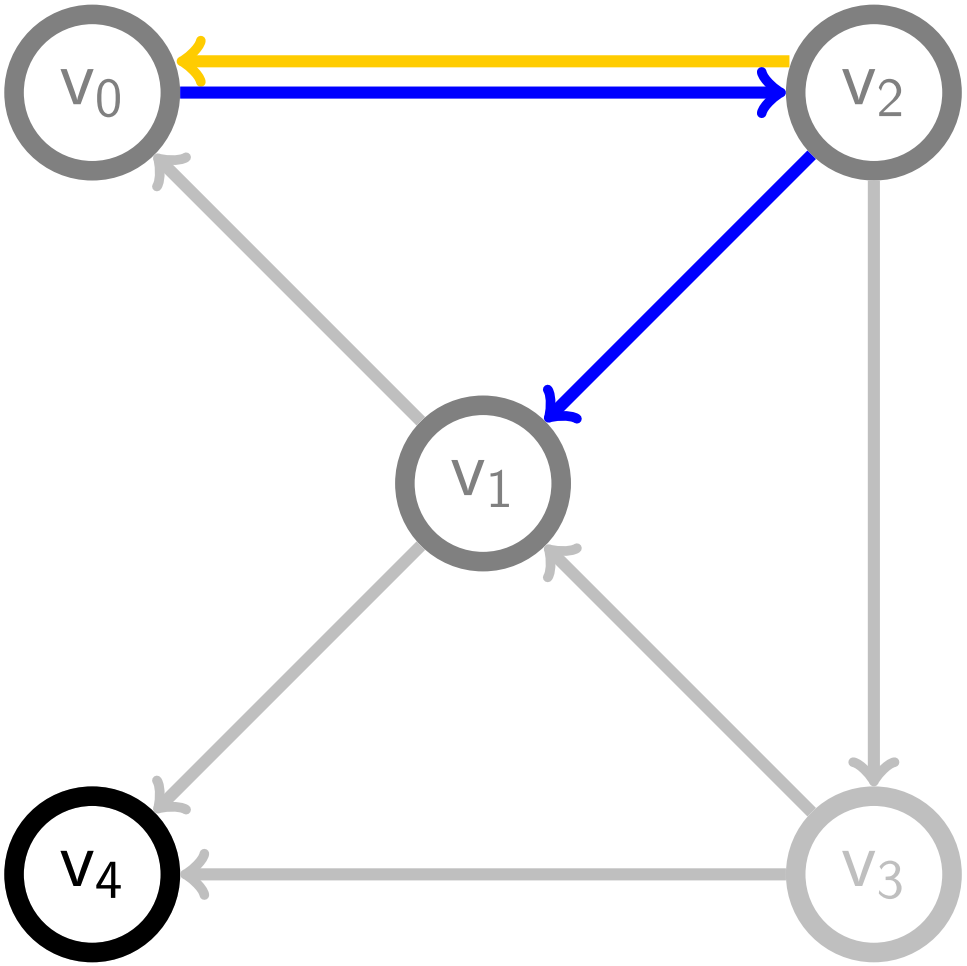
Componentes Fortemente Conexos - DFS(G^T)*



Ordem: tTérmino
(decrescente)
v4->v0->v1->v3->v2

Arestas de Árvore
Arestas de Retorno
Arestas de Cruzamento
Arestas de Avanço

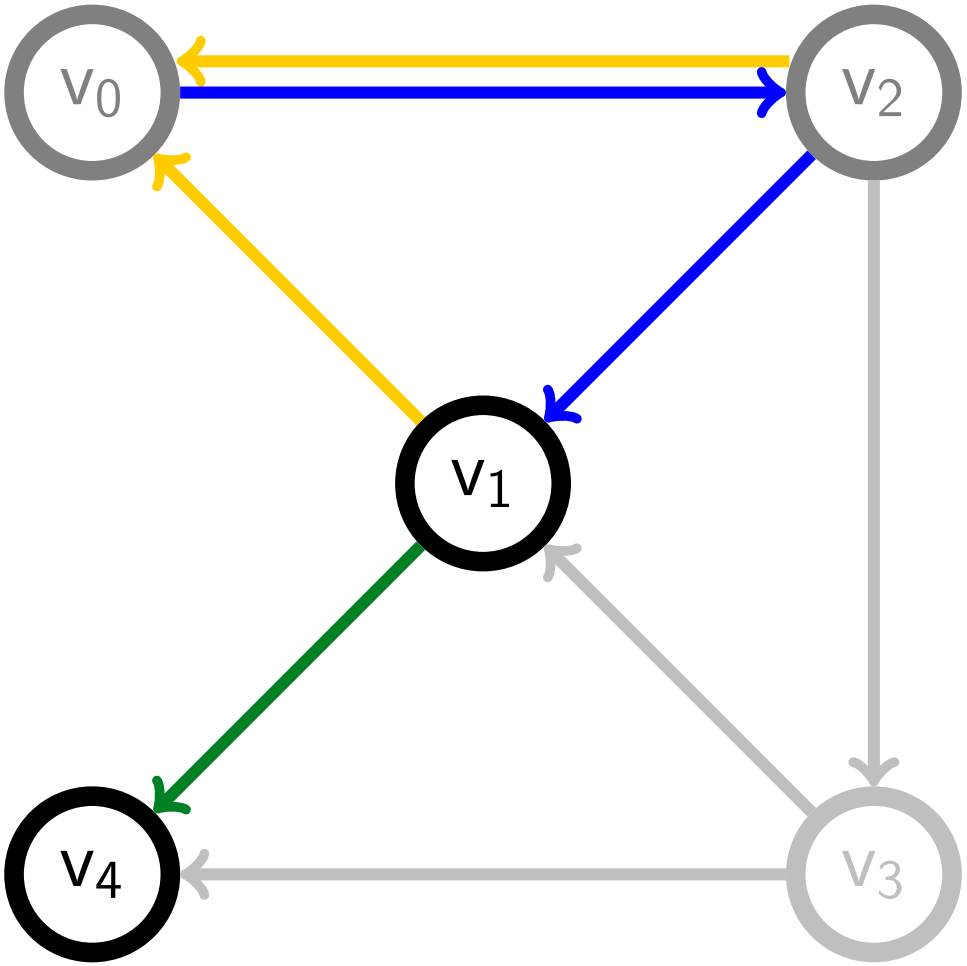
Componentes Fortemente Conexos - DFS(G^T)*



Ordem: tTérmino
(decrescente)
v4->v0->v1->v3->v2

Arestas de Árvore
Arestas de Retorno
Arestas de Cruzamento
Arestas de Avanço

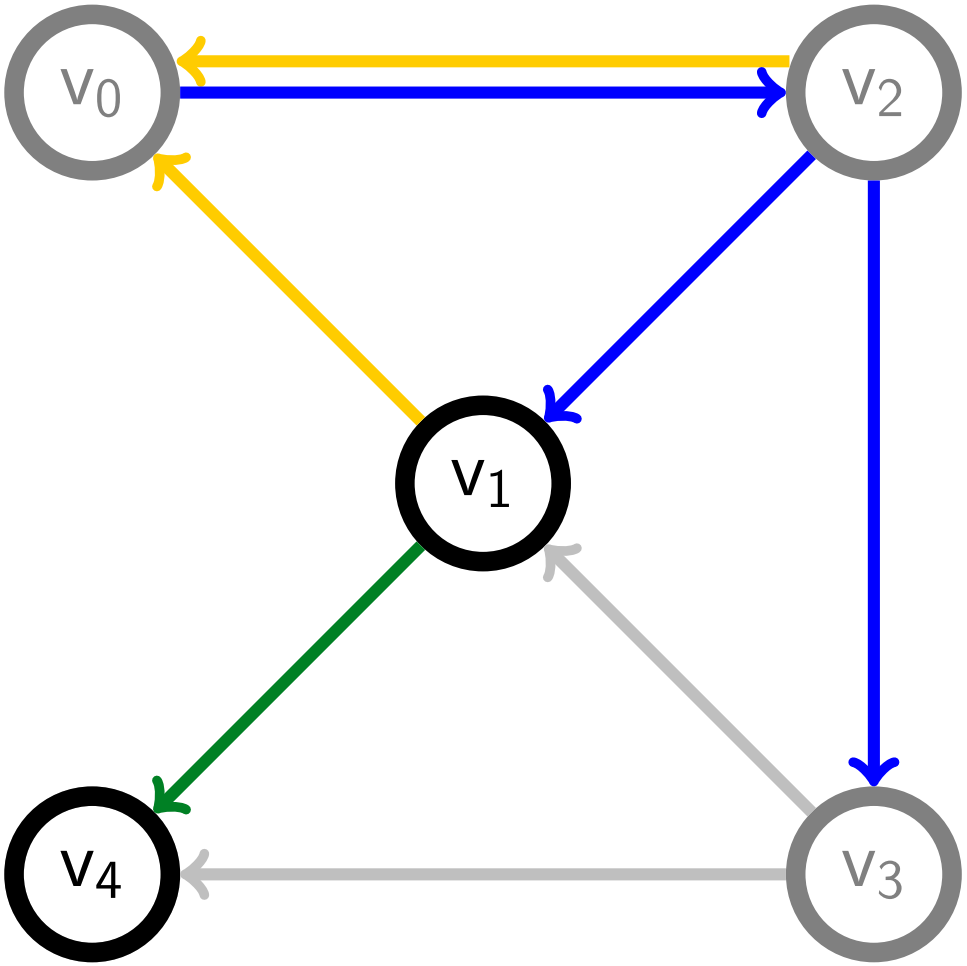
Componentes Fortemente Conexos - DFS(G^T)*



Ordem: tTérmino
(decrescente)
v4->v0->v1->v3->v2

Arestas de Árvore
Arestas de Retorno
Arestas de Cruzamento
Arestas de Avanço

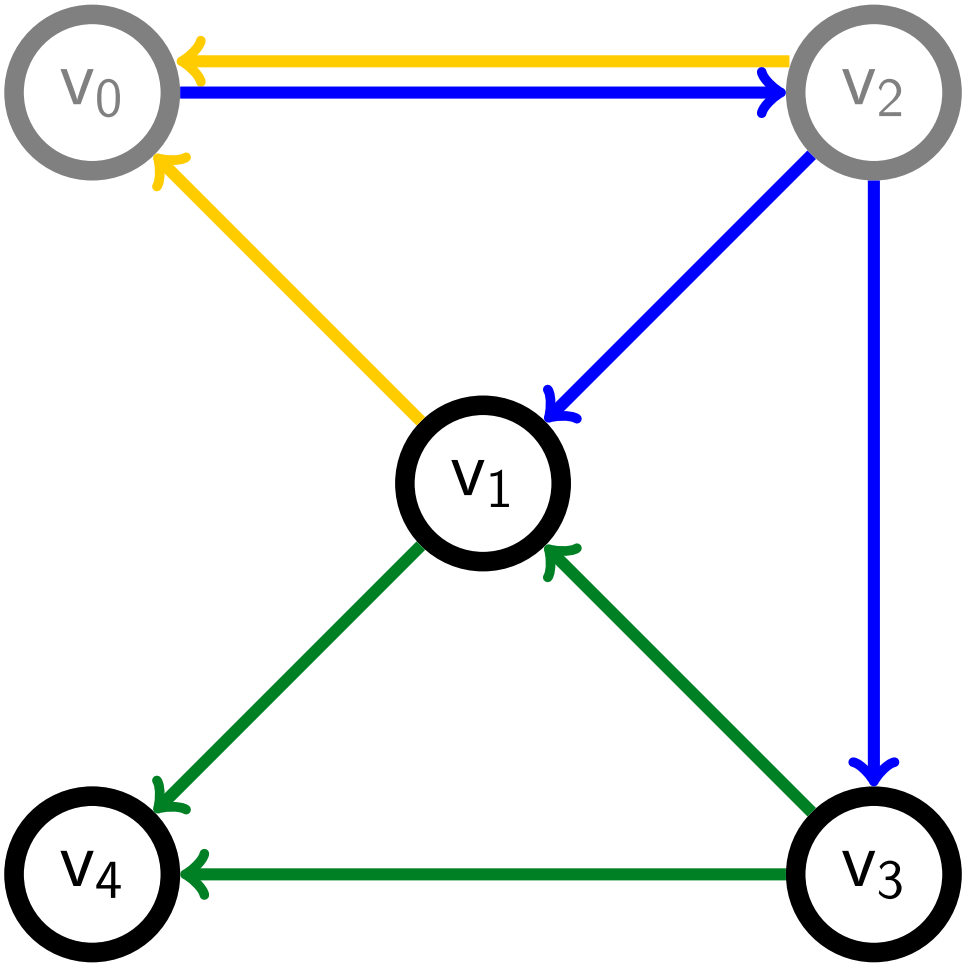
Componentes Fortemente Conexos - DFS(G^T)*



Ordem: tTérmino
(decrescente)
v4->v0->v1->v3->v2

Arestas de Árvore
Arestas de Retorno
Arestas de Cruzamento
Arestas de Avanço

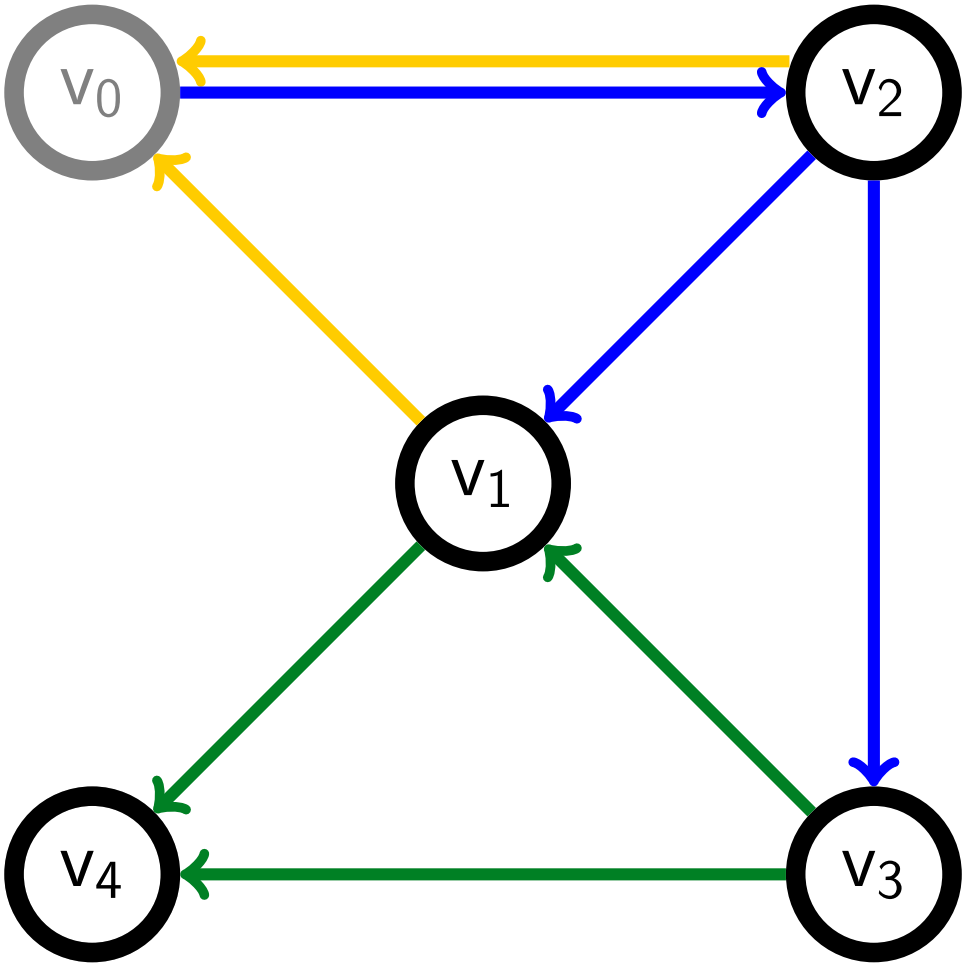
Componentes Fortemente Conexos - DFS(G^T)*



Ordem: tTérmino
(decrescente)
v4->v0->v1->v3->v2

Arestas de Árvore
Arestas de Retorno
Arestas de Cruzamento
Arestas de Avanço

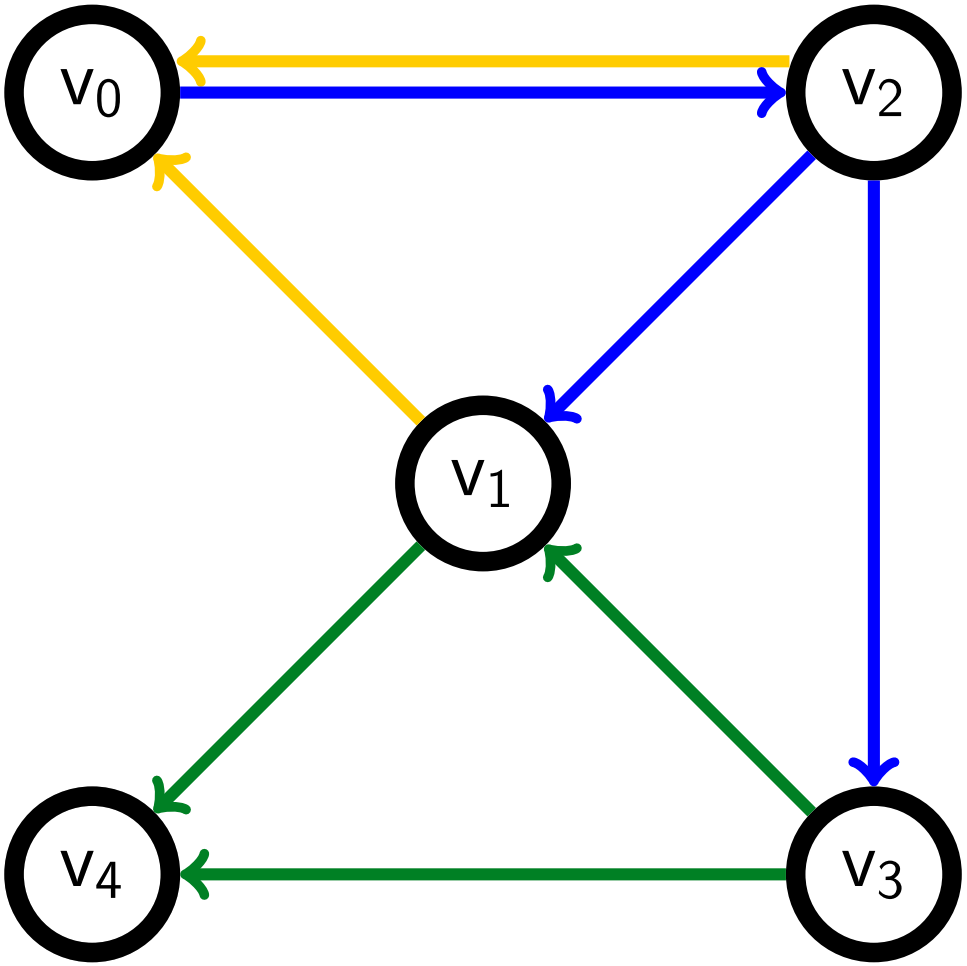
Componentes Fortemente Conexos - DFS(G^T)*



Ordem: tTérmino
(decrescente)
v4->v0->v1->v3->v2

Arestas de Árvore
Arestas de Retorno
Arestas de Cruzamento
Arestas de Avanço

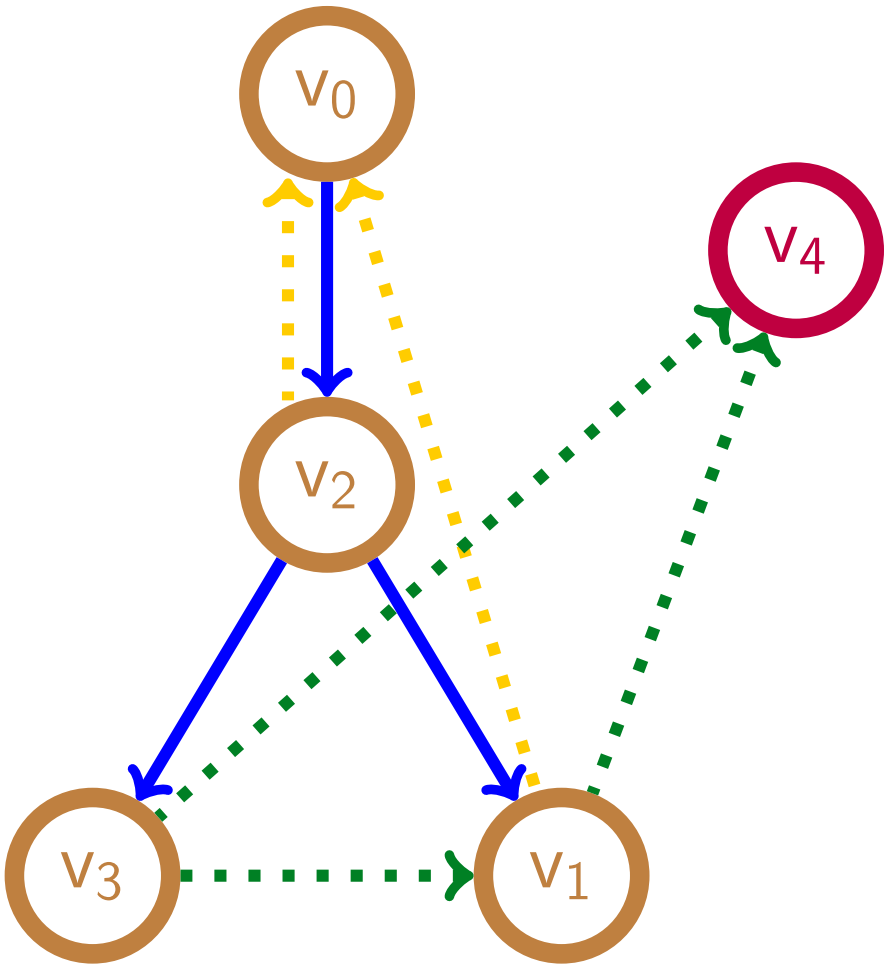
Componentes Fortemente Conexos - DFS(G^T)*



Ordem: tTérmino
(decrescente)
v4->v0->v1->v3->v2

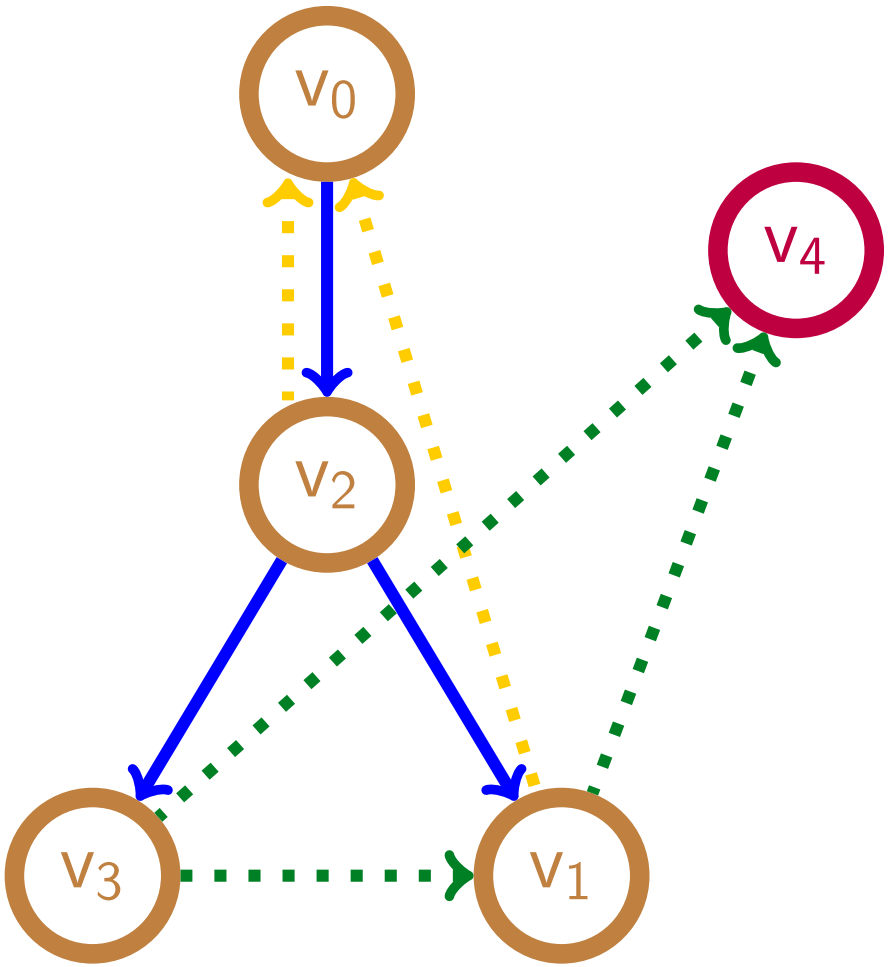
Arestas de Árvore
Arestas de Retorno
Arestas de Cruzamento
Arestas de Avanço

Componentes Fortemente Conexos - Árvores (G^T)



Duas árvores produzidas: dois componentes fortemente conexos.

Componentes Fortemente Conexos - Árvores (G^T)



Duas árvores produzidas: dois componentes fortemente conexos.

Arestas de Árvore
Arestas de Retorno
Arestas de Cruzamento
Arestas de Avanço

Componentes Fortemente Conexos

```
int* componentesFortementeConexos(Grafo* g){

  ElemLista *atual, *ordenacao = DFSOrdTopologica(g);

  int compAtual = 0;
  int* componente = malloc(sizeof(int)*g->numVertices);
  bool* visitado = malloc(sizeof(bool)*g->numVertices);
  int x;
  for (x=0;x<g->numVertices;x++){
    visitado[x] = false;
    componente[x] = -1;
  }
  ...
```

Componentes Fortemente Conexos

```
  ...
  Grafo* gt = transpoeGrafo2(g);
  atual = ordenacao;
  while(atual){
    if (!visitado[atual->vertice])
      visitaComponentes(gt, atual->vertice,
                        visitado, ++compAtual, componente);
    atual = atual->prox;
  }
  liberaGrafo(gt);
  free(gt);
  free(visitado);
  return componente;
}
```

$O(V + E)$

Algoritmos e Estruturas de Dados II Aula 07 – Grafos: Busca em Profundidade - Aplicações (continuação)

Prof. Luciano A. Digiampietri
digiampietri@usp.br
@digiampietri