Swiss Olympiad in Informatics 2018

Graph Theory and DFS

Dennis Komm

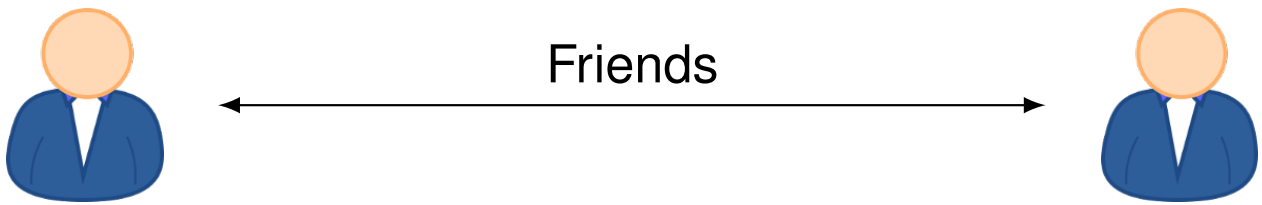
February 12, 2018 – Ftan

Searching in Networks

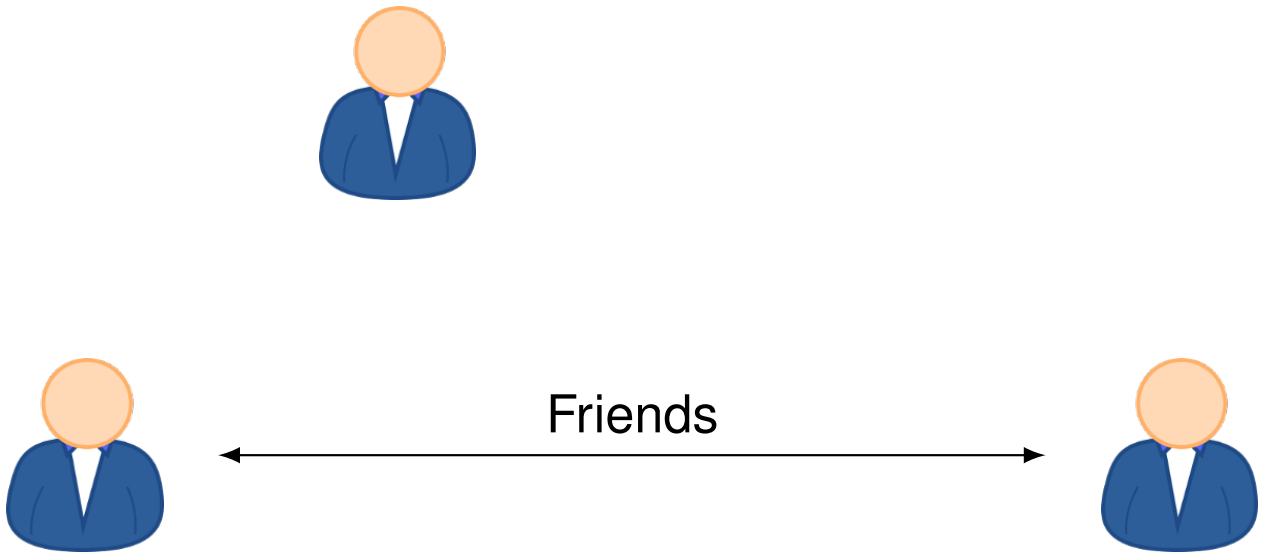
Social Network

Social Network

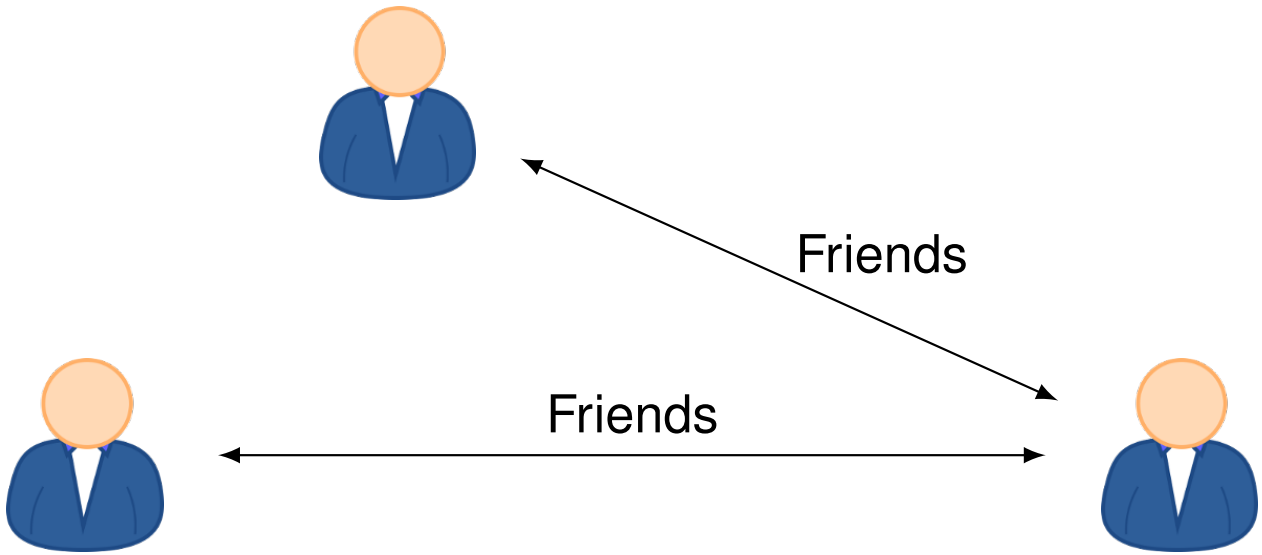
Social Network



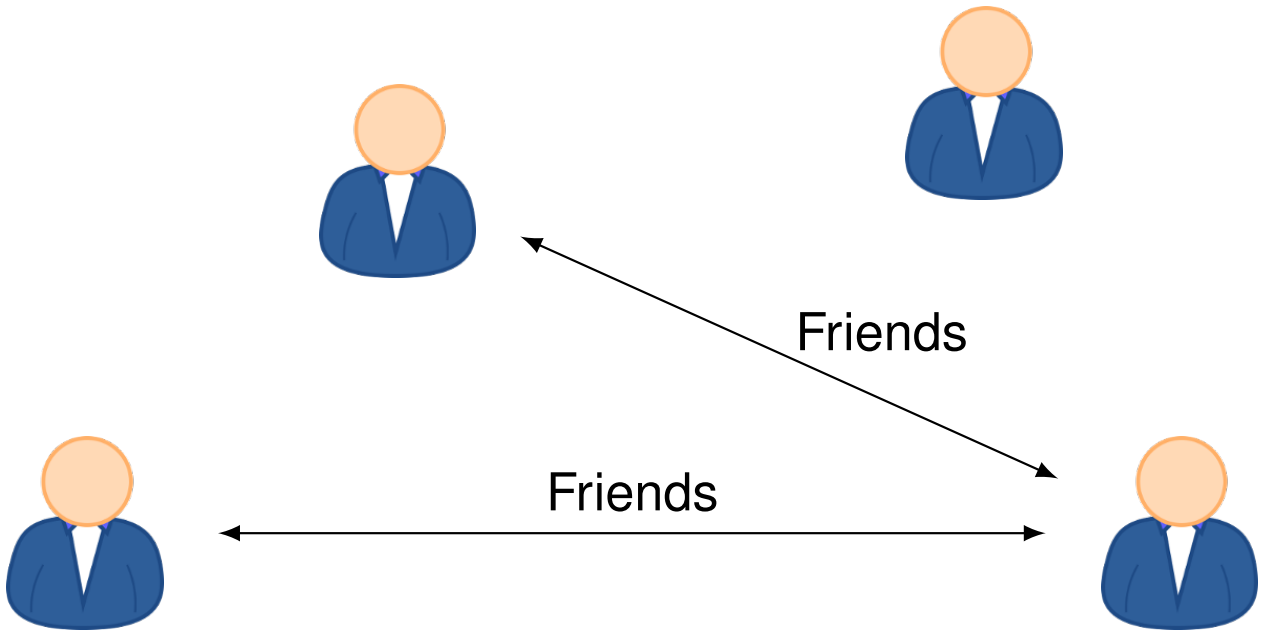
Social Network



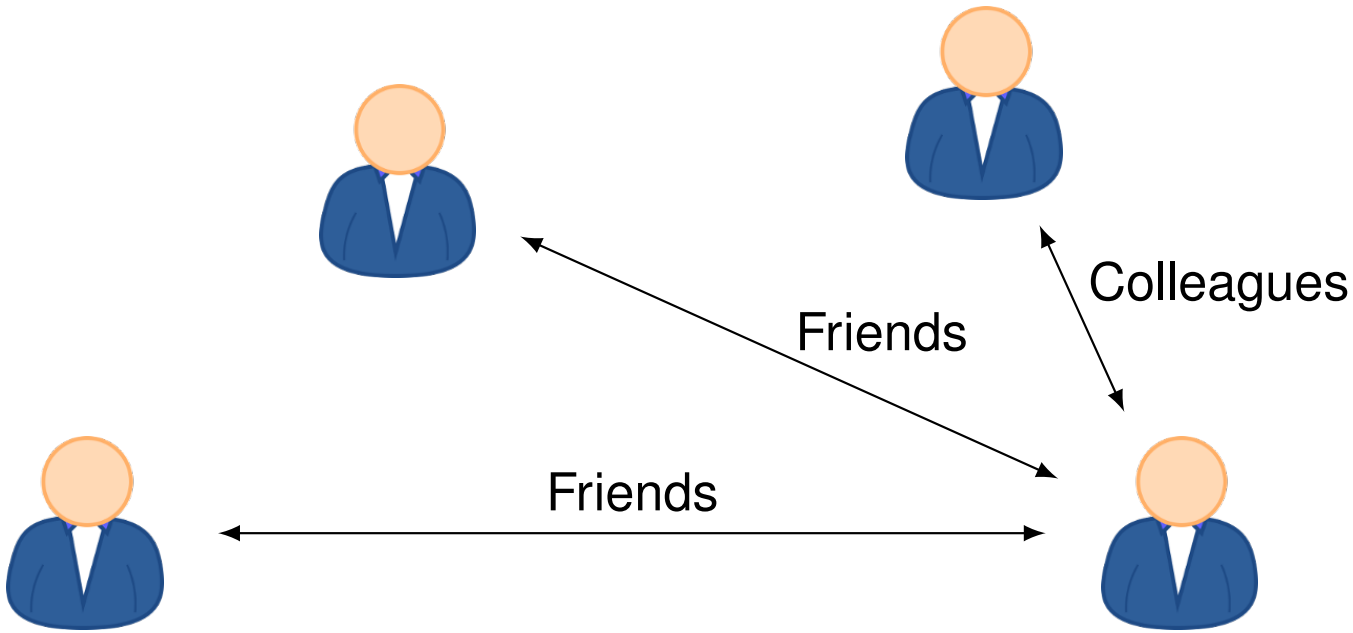
Social Network



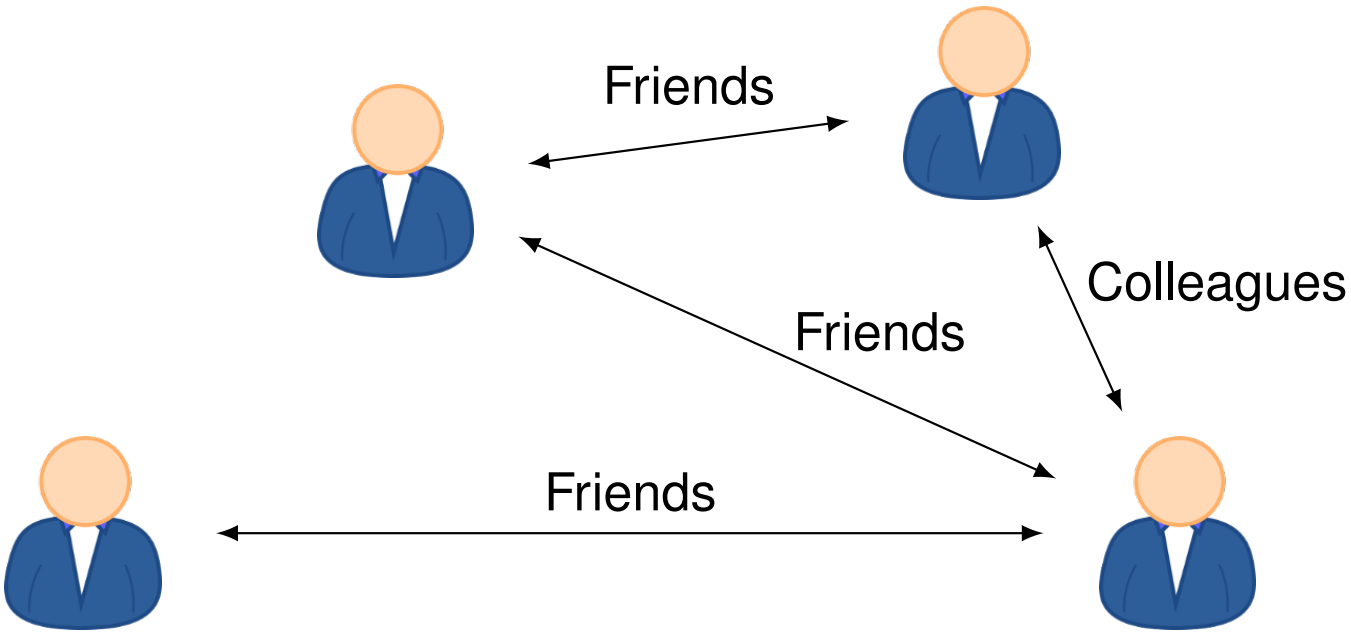
Social Network



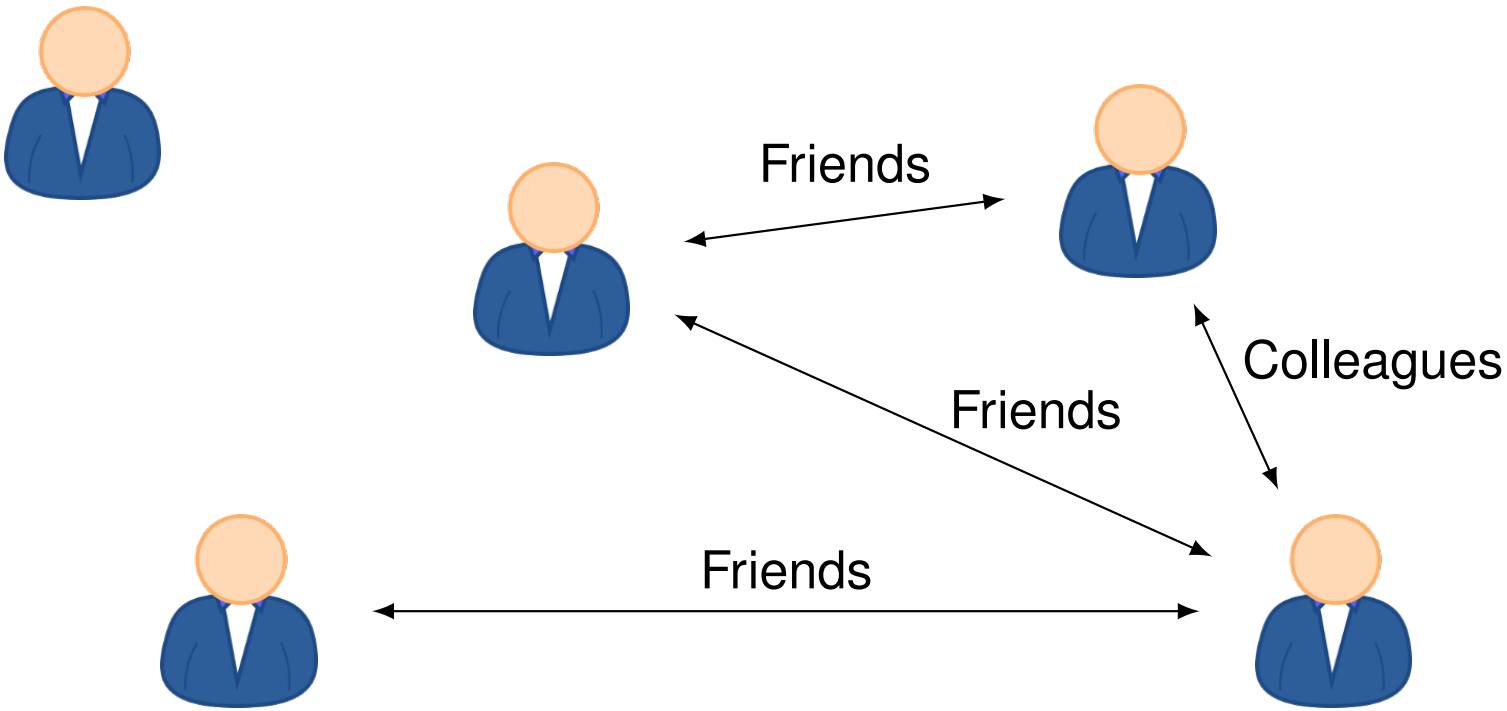
Social Network



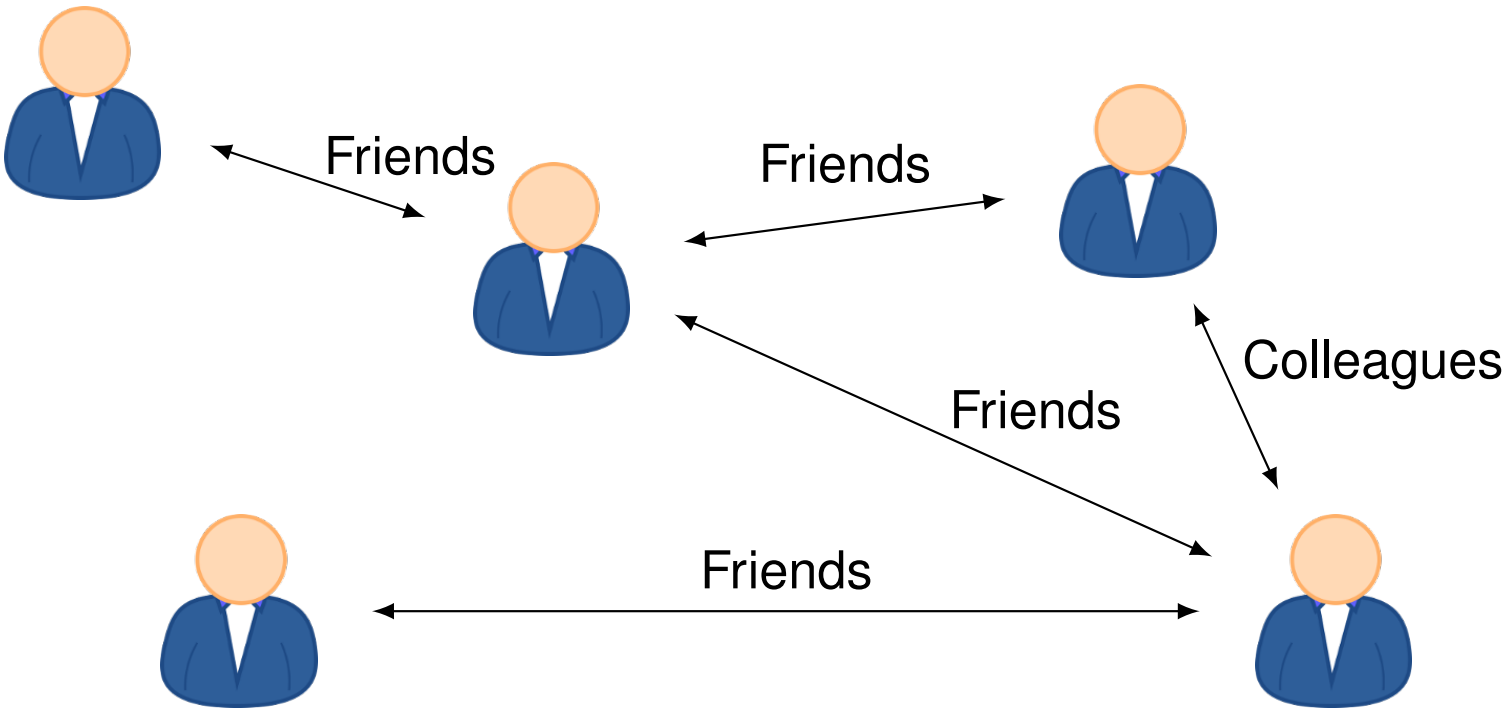
Social Network



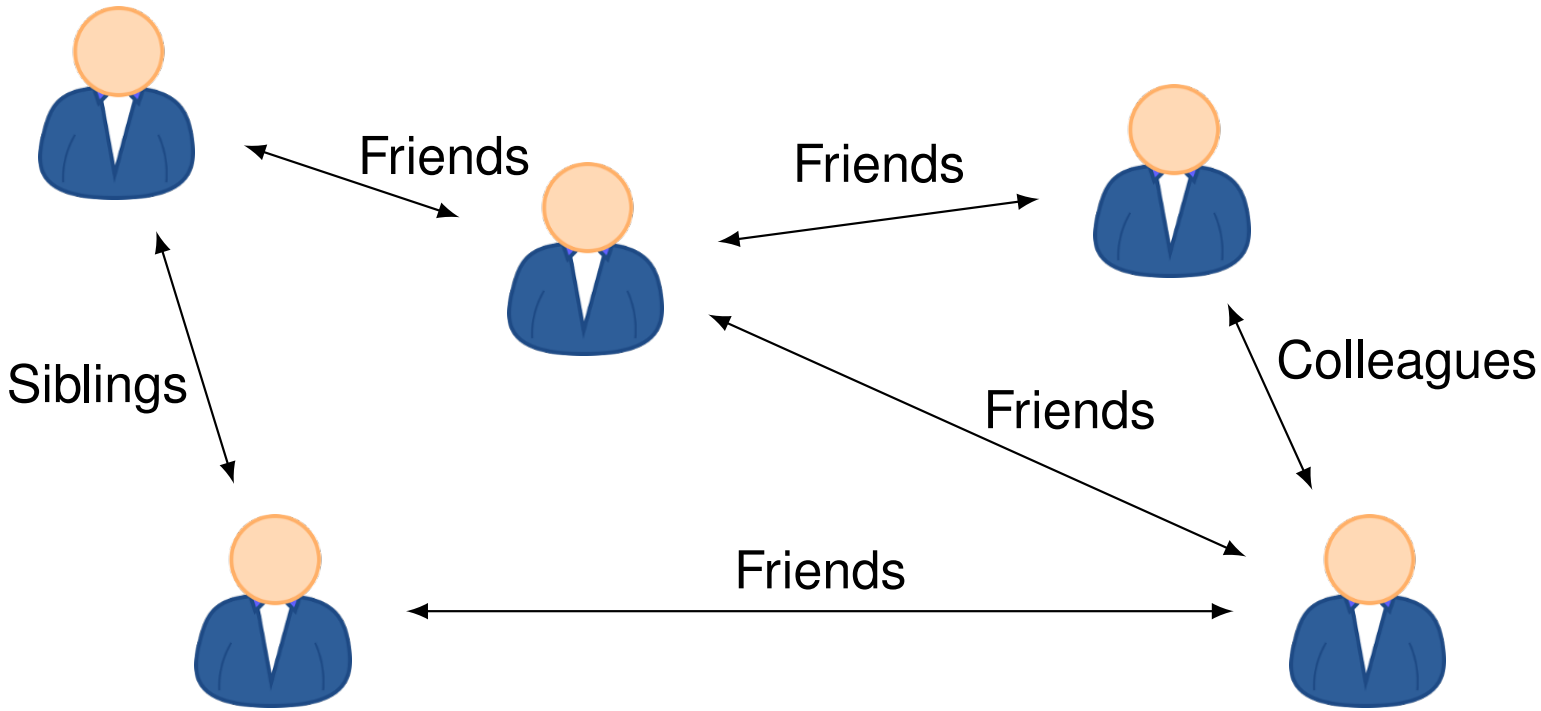
Social Network



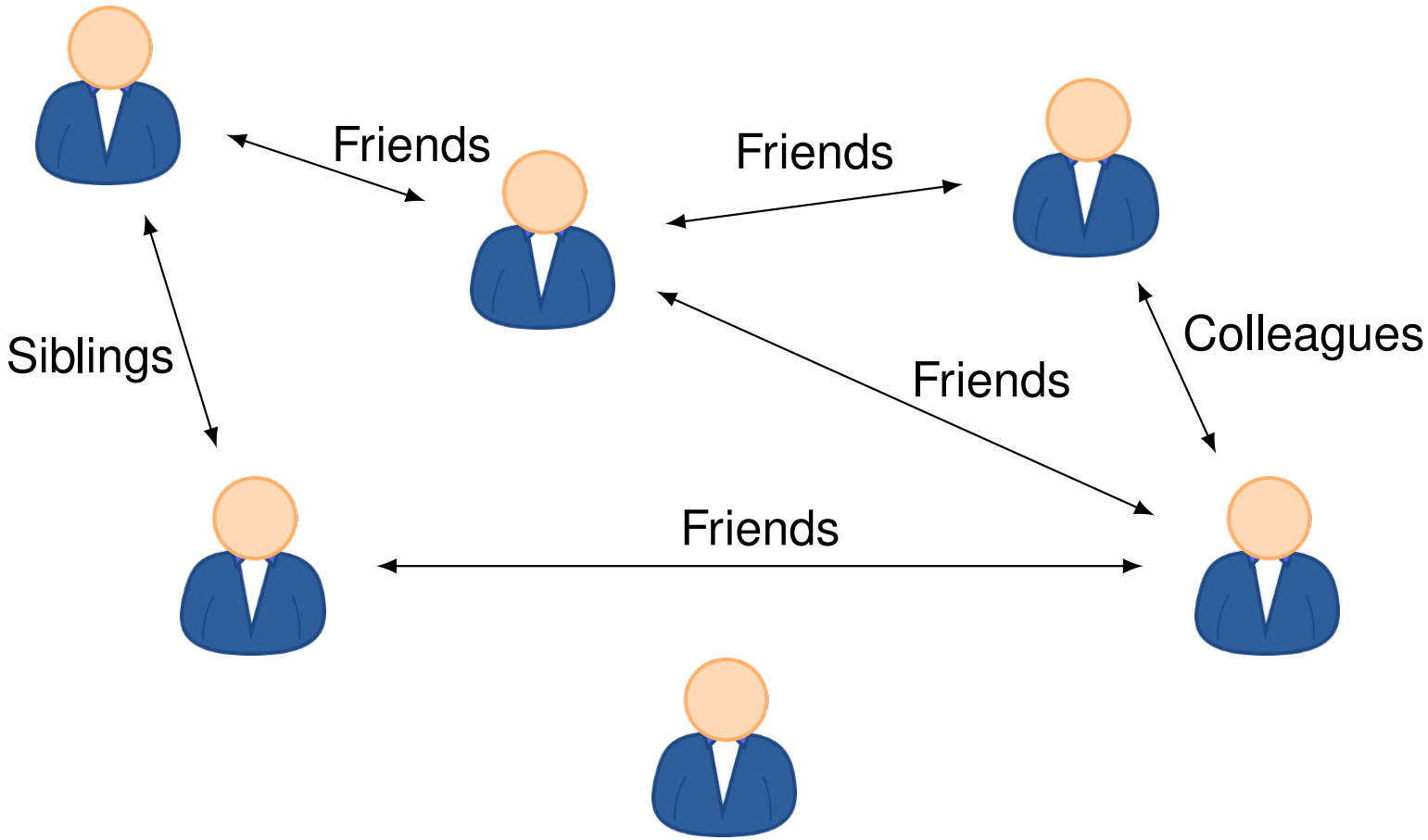
Social Network



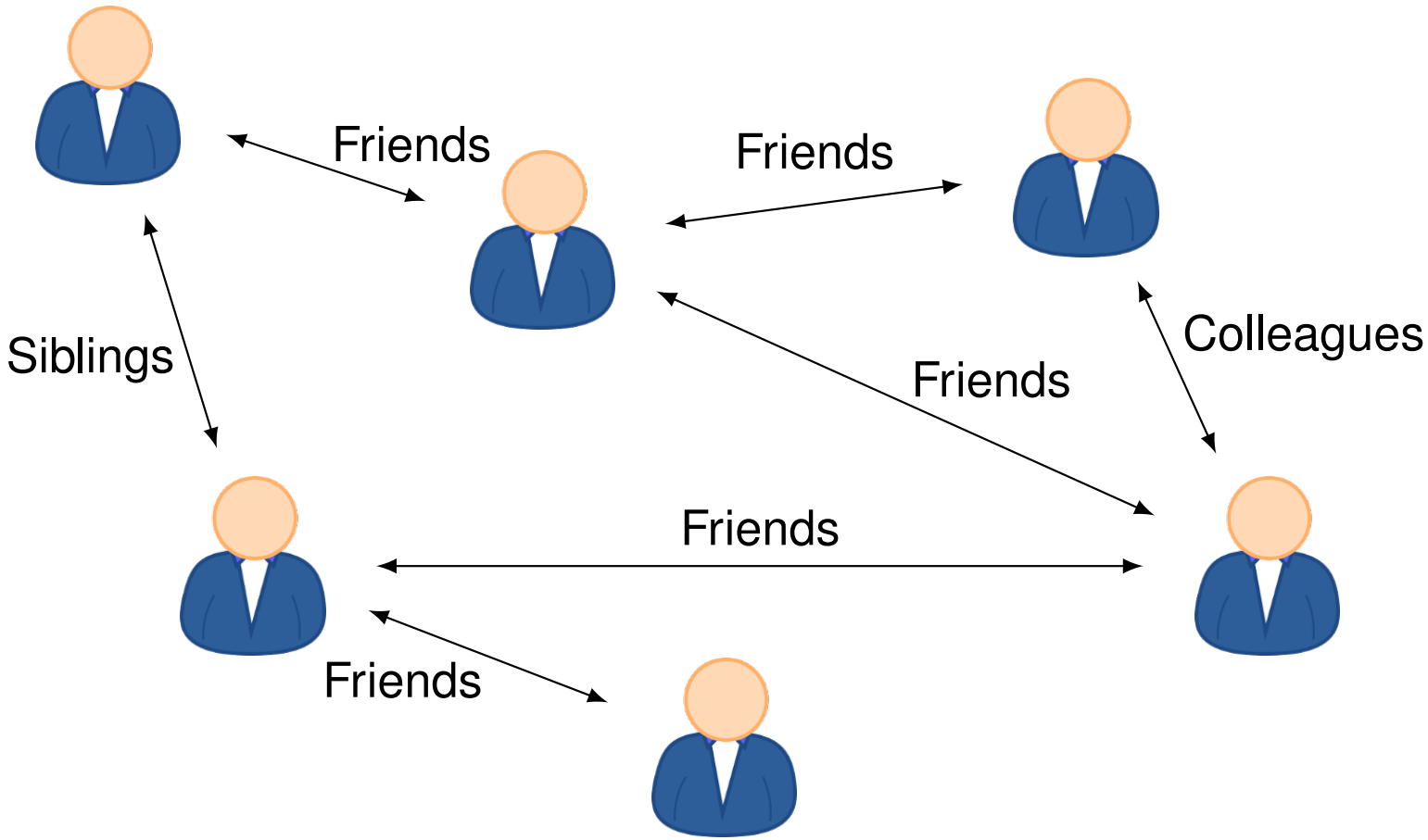
Social Network



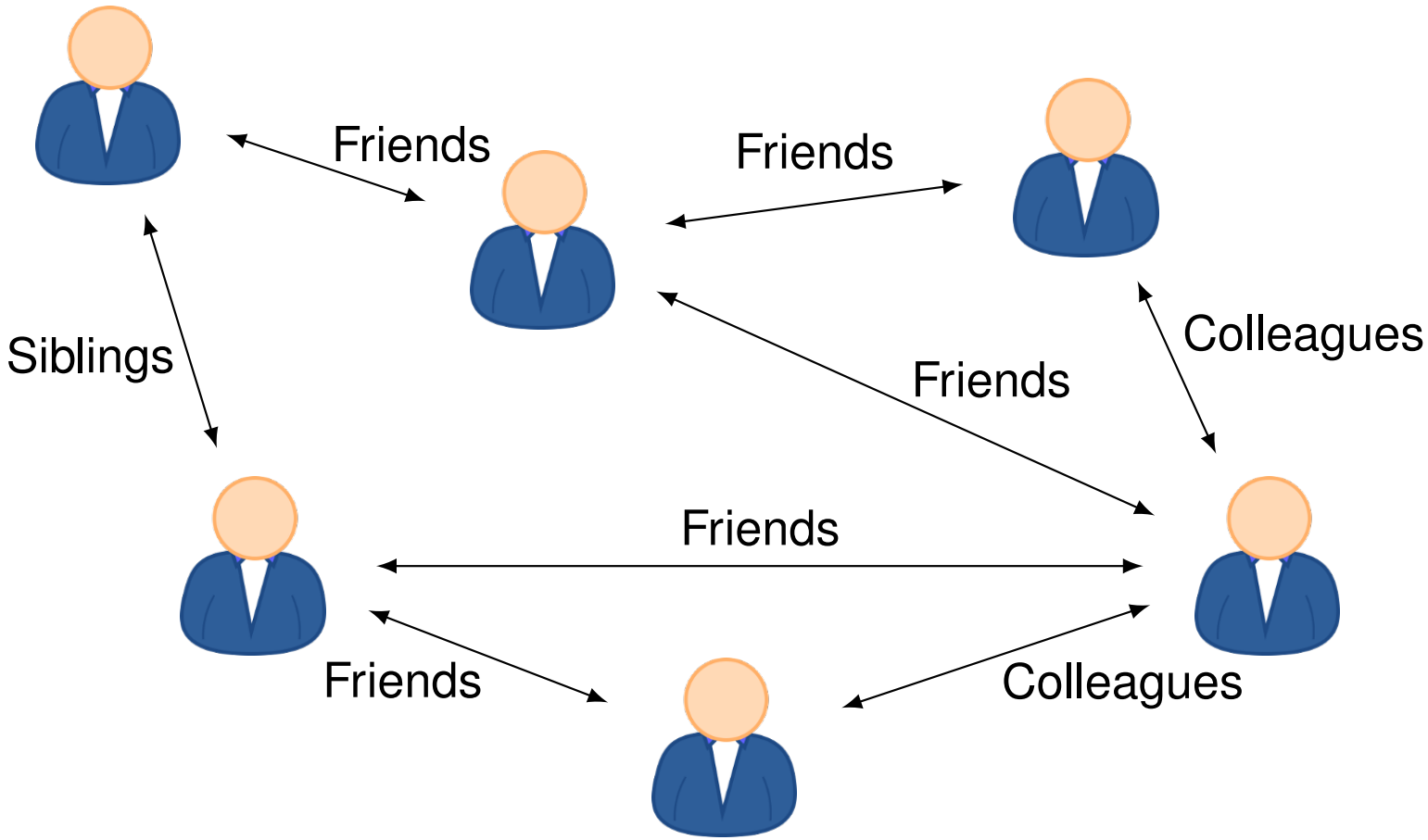
Social Network



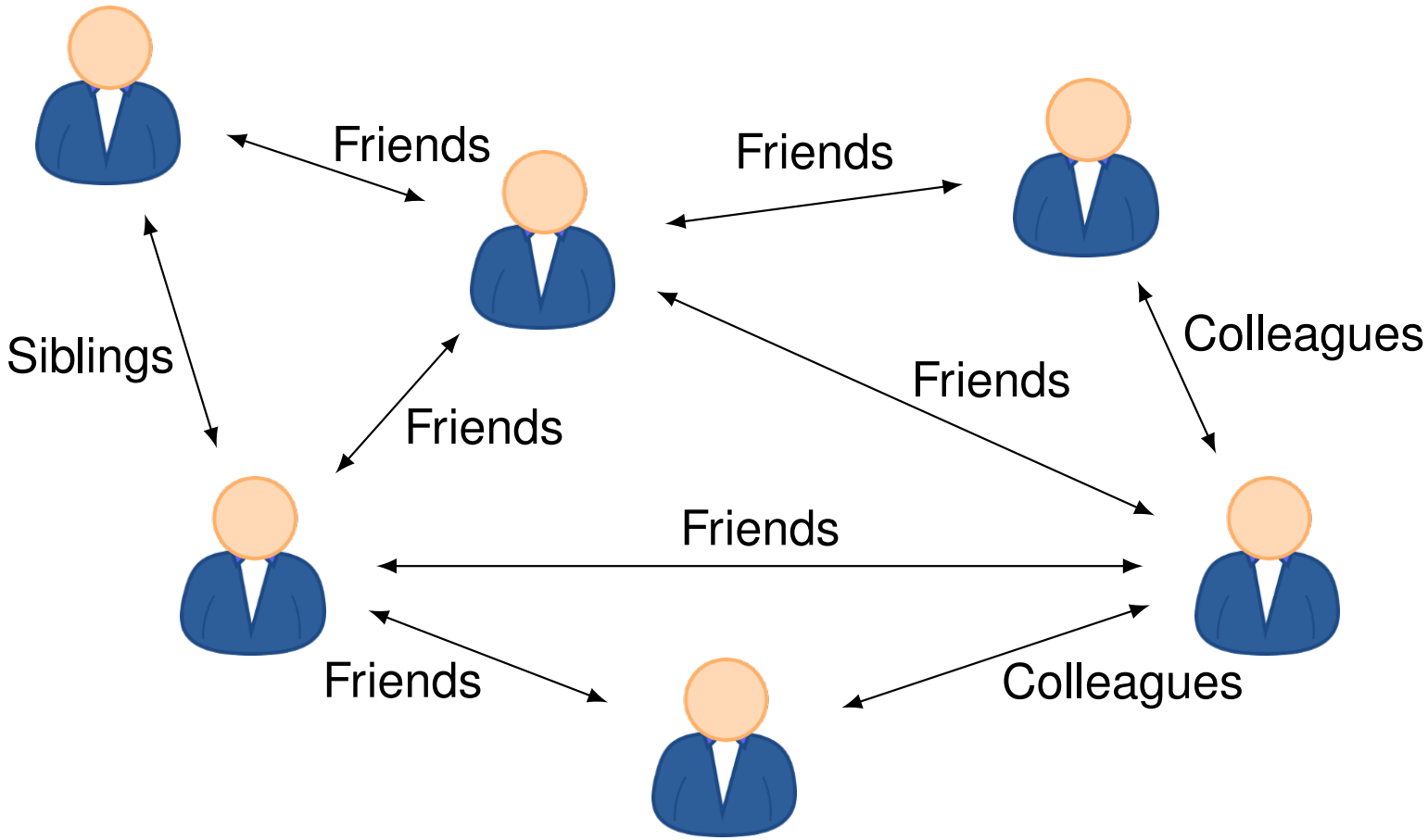
Social Network



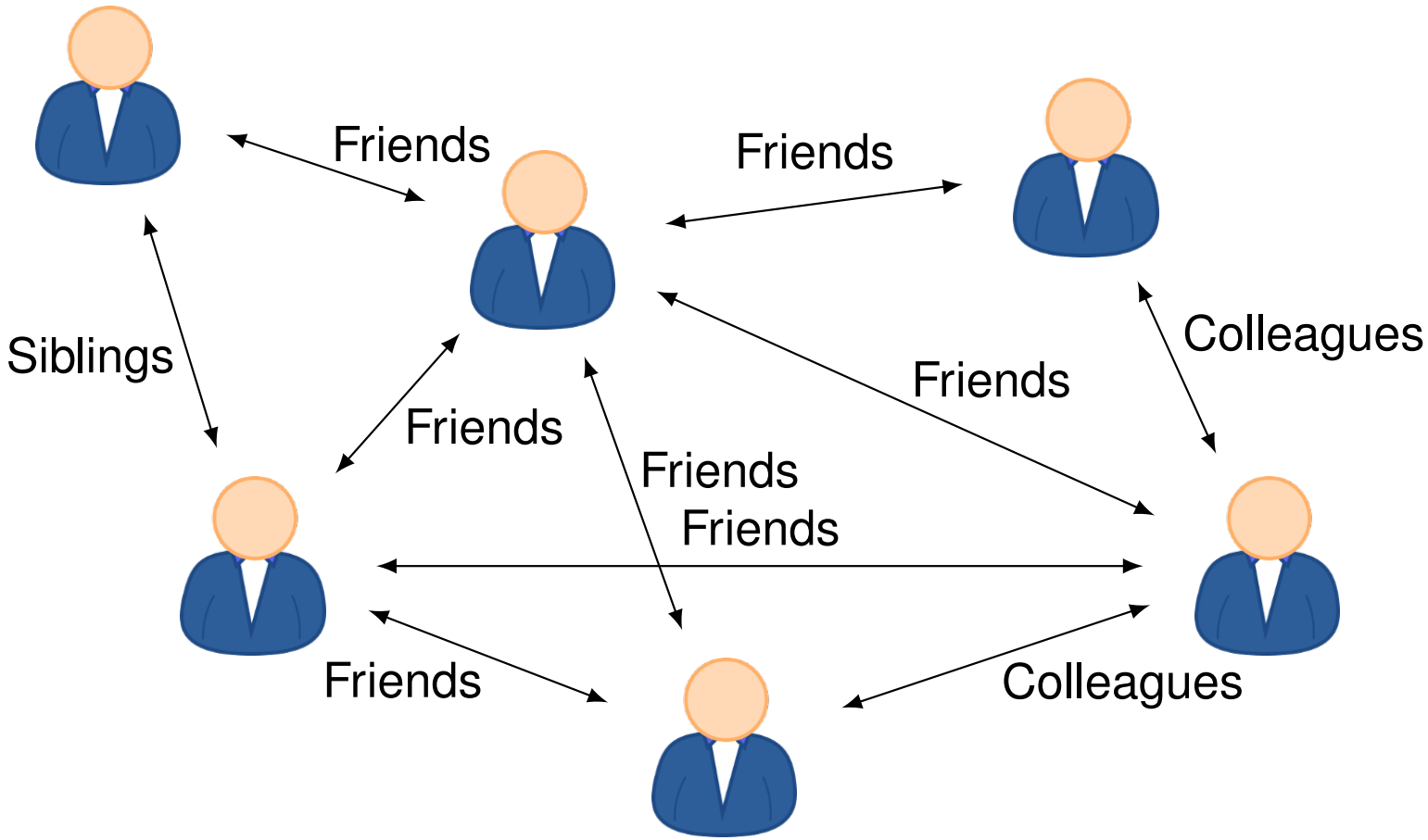
Social Network



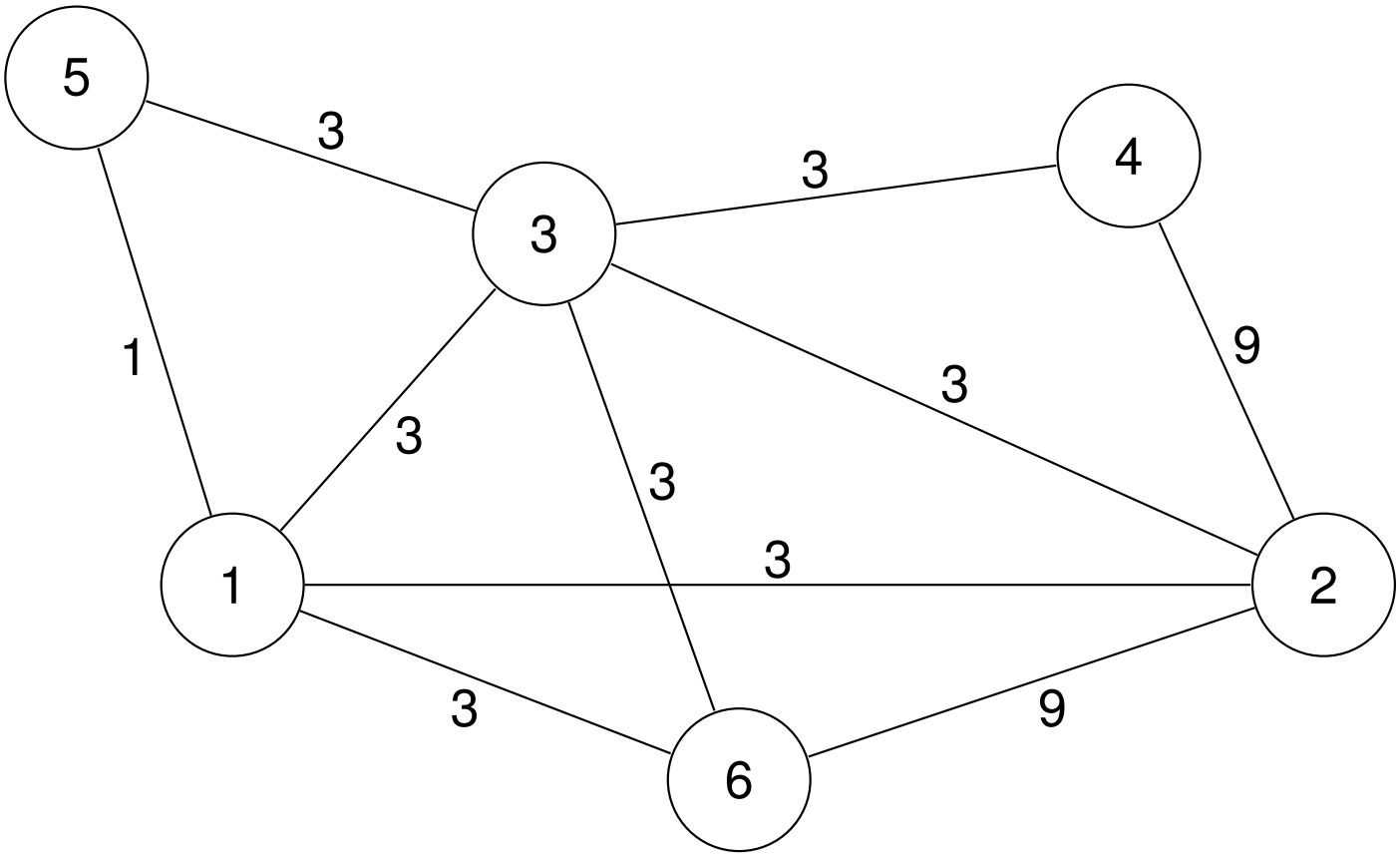
Social Network



Social Network



Abstract Modelling



Abstract Modelling

A **graph** G consists of

Abstract Modelling

A **graph** G consists of

1. a set V of vertices

Abstract Modelling

A **graph** G consists of

1. a set V of vertices
2. a set E of edges

Abstract Modelling

A **graph** G consists of

1. a set V of vertices
2. a set E of edges
3. (a weight function w)

Abstract Modelling

A **graph** G consists of

1. a set V of vertices
2. a set E of edges
3. (a weight function w)

- Vertices are called $v_0, v_1, v_2, \ldots$

Abstract Modelling

A **graph** G consists of

1. a set V of vertices
2. a set E of edges
3. (a weight function w)

- Vertices are called $v_0, v_1, v_2, \ldots$
- Graphs are either **weighted** or **unweighted**

Abstract Modelling

A **graph** G consists of

1. a set V of vertices
2. a set E of edges
3. (a weight function w)

- Vertices are called $v_0, v_1, v_2, \ldots$
- Graphs are either **weighted** or **unweighted**
- Graphs are either **undirected** or **directed**

Abstract Modelling

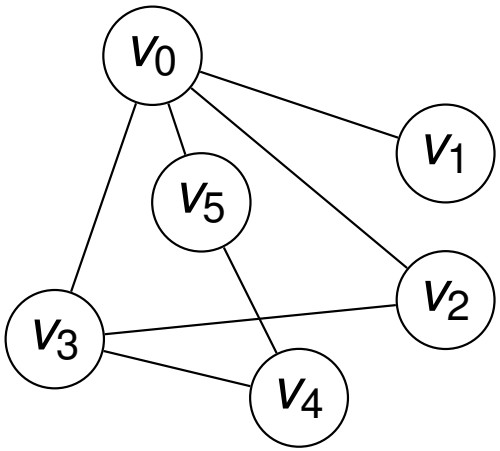
A **graph** G consists of

1. a set V of vertices
2. a set E of edges
3. (a weight function w)

- Vertices are called $v_0, v_1, v_2, \ldots$
- Graphs are either **weighted** or **unweighted**
- Graphs are either **undirected** or **directed**
- Graphs are either **connected** or **unconnected**

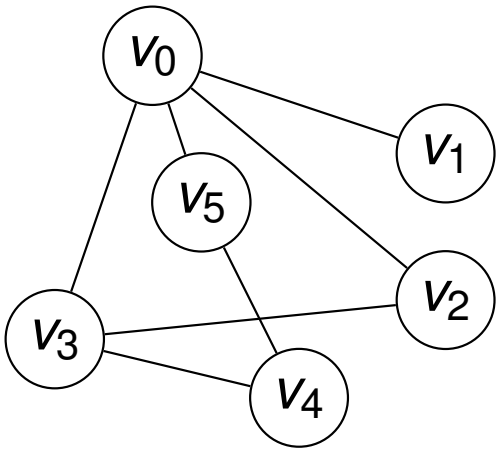
Abstract Modelling

Abstract Modelling

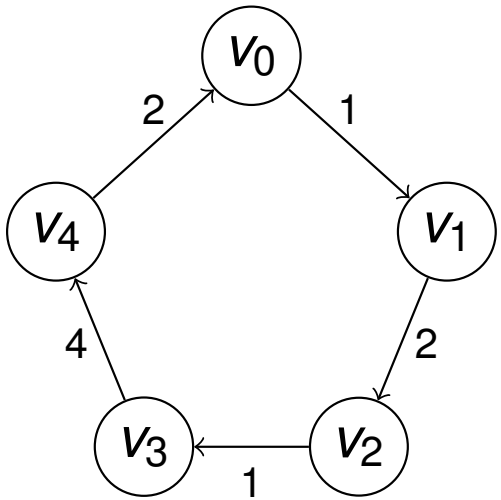


Undirected unweighted graph

Abstract Modelling

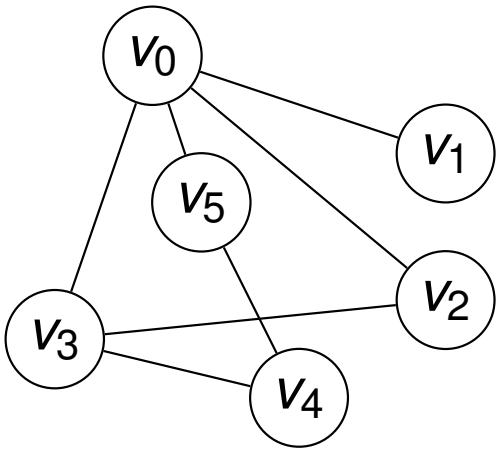


Undirected unweighted graph

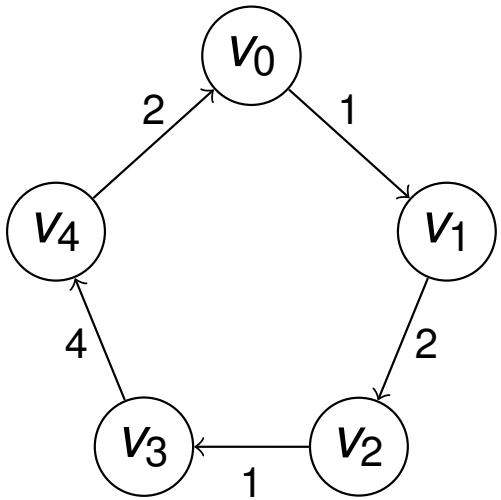


Undirected weighted graph

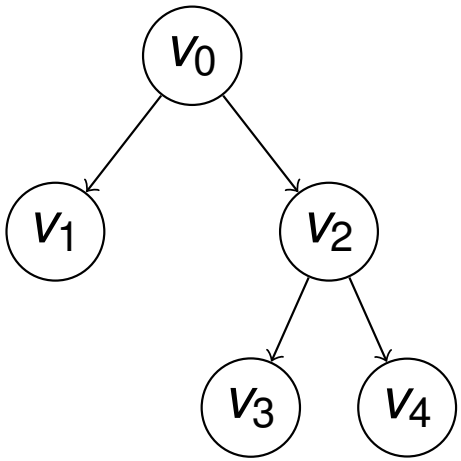
Abstract Modelling



Undirected unweighted graph

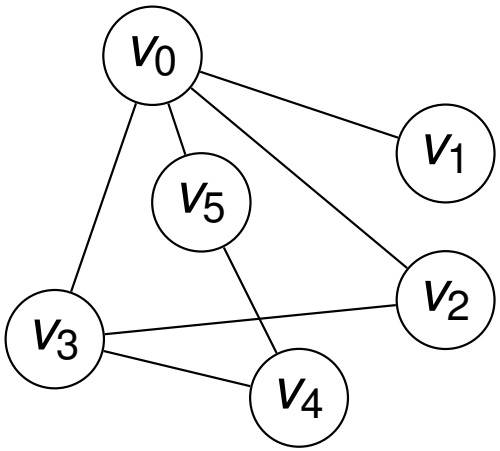


Undirected weighted graph

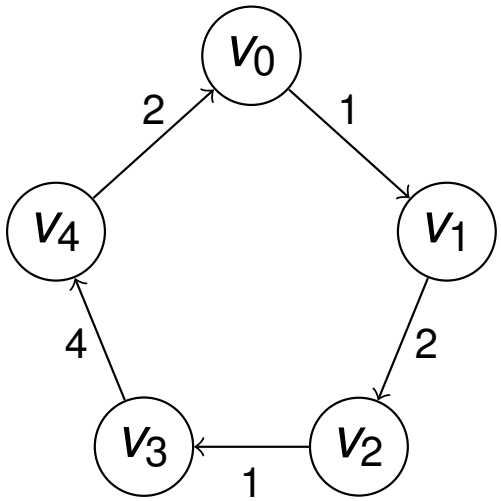


Directed unweighted graph

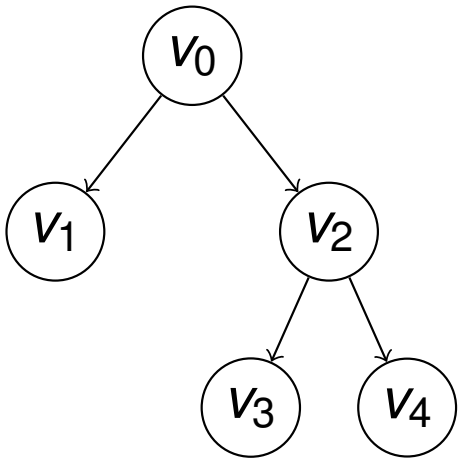
Abstract Modelling



Undirected unweighted graph



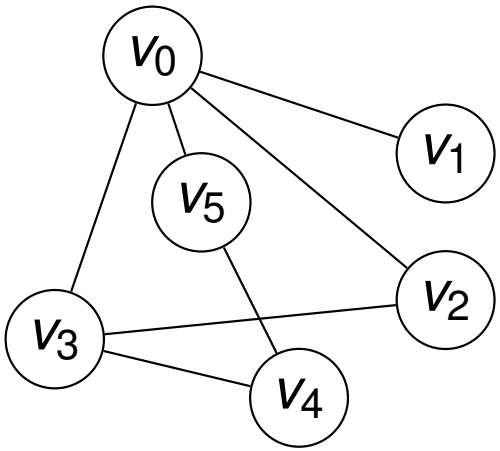
Undirected weighted graph



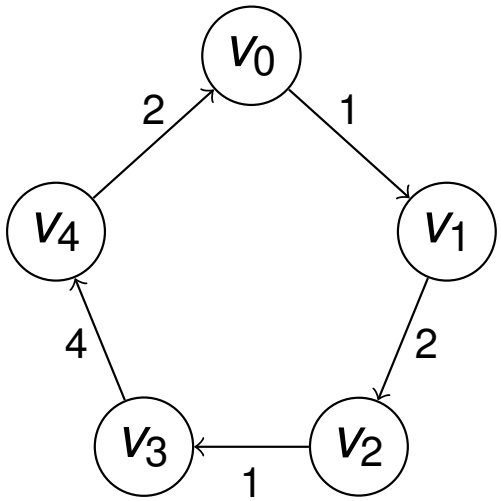
Directed unweighted graph

Which graphs are used depends on what we want to model

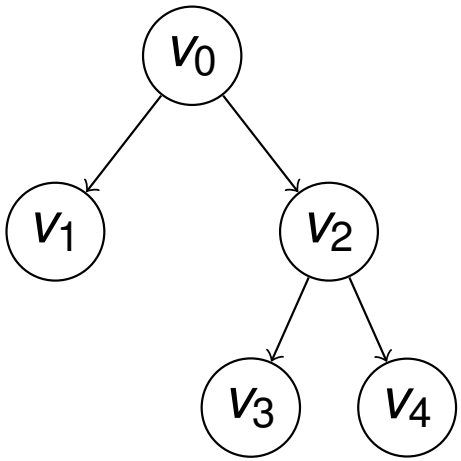
Abstract Modelling



Undirected unweighted graph



Undirected weighted graph



Directed unweighted graph

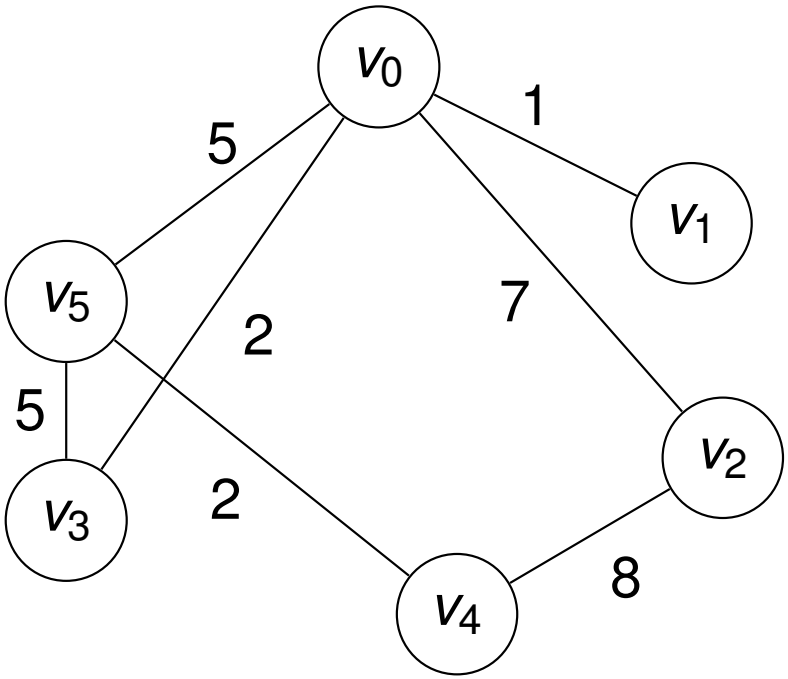
Which graphs are used depends on what we want to model

For simple arguments, we mostly assume connected graphs

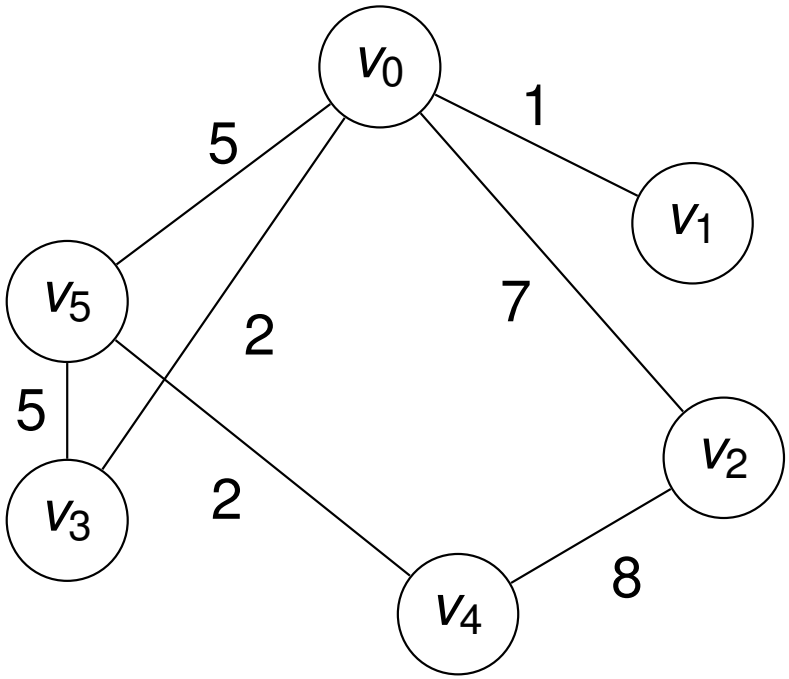
On the Computer

Graphs

Adj. Matrices: Undirected Weighted Graph

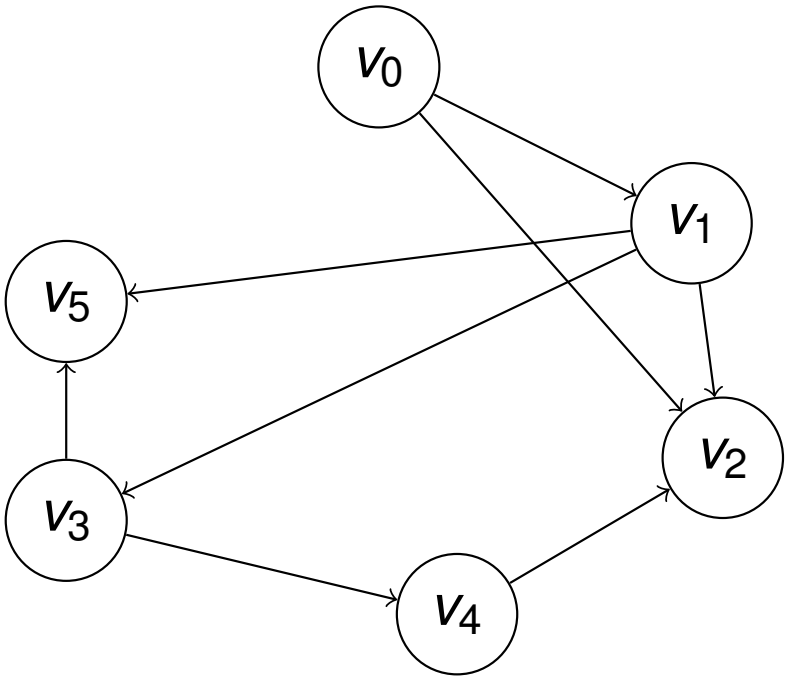


Adj. Matrices: Undirected Weighted Graph

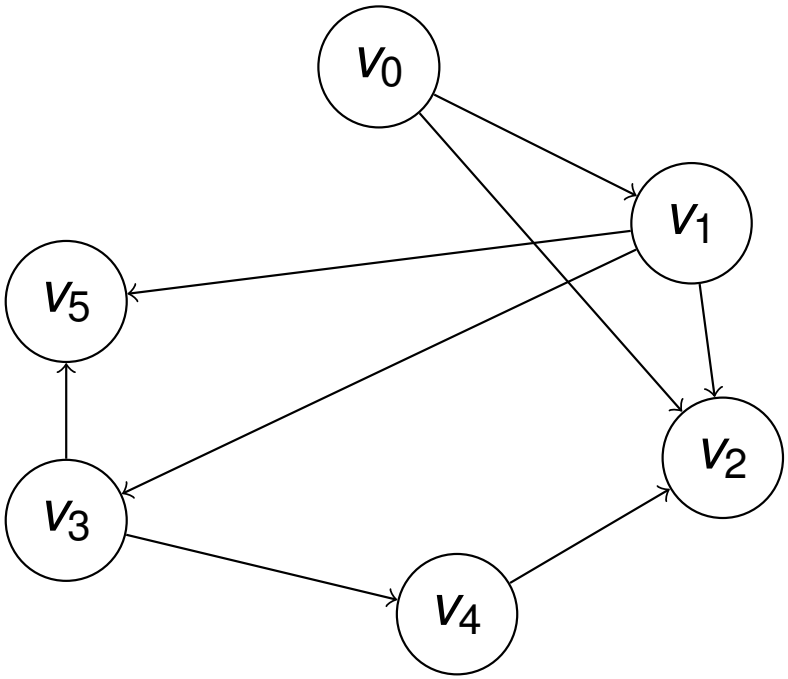


$$\begin{pmatrix} 0 & 1 & 7 & 2 & 0 & 5 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 8 & 0 \\ 2 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 8 & 0 & 0 & 2 \\ 5 & 0 & 0 & 5 & 2 & 0 \end{pmatrix}$$

Adj. Matrices: Directed Unweighted Graph

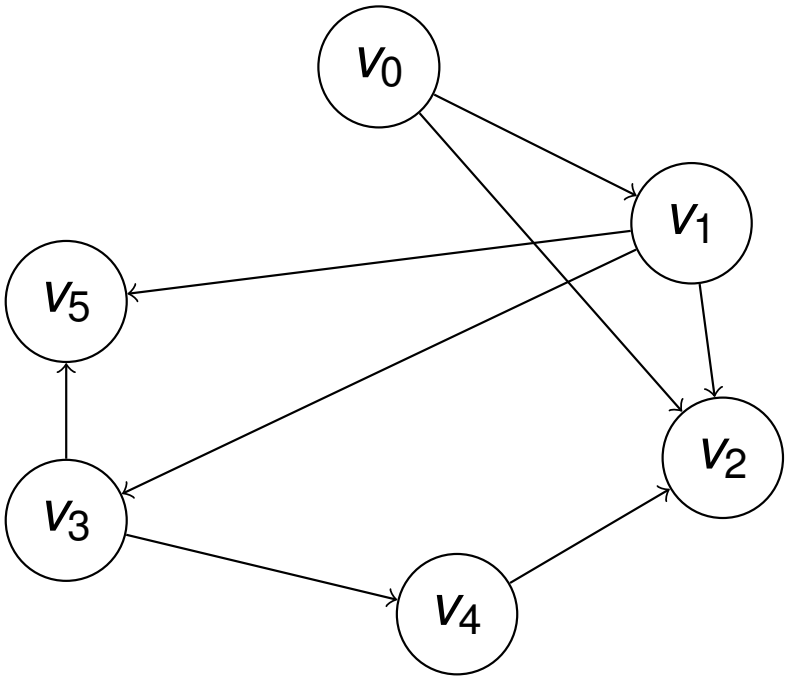


Adj. Matrices: Directed Unweighted Graph



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Adj. Lists: Directed Unweighted Graph



$((1,2),$
$(2,3,5),$
$(),$
$(4,5),$
$(2),$
$())$

Adj. Matrices and Lists in C++

Use 2-dimensional `vector` of `int`s

Adj. Matrices and Lists in C++

Use 2-dimensional `vector` of `ints`

Matrix: Weighted

```
vector<vector<int>>
G { {0, 1, 7, 2, 0, 5},
    {1, 0, 0, 0, 0, 0},
    {7, 0, 0, 0, 8, 0},
    {2, 0, 0, 0, 0, 5},
    {0, 0, 8, 0, 0, 2},
    {5, 0, 0, 5, 2, 0} };
```

Adj. Matrices and Lists in C++

Use 2-dimensional `vector` of `ints`

Matrix: Weighted

```
vector<vector<int>>
G { {0, 1, 7, 2, 0, 5},
    {1, 0, 0, 0, 0, 0},
    {7, 0, 0, 0, 8, 0},
    {2, 0, 0, 0, 0, 5},
    {0, 0, 8, 0, 0, 2},
    {5, 0, 0, 5, 2, 0} };
```

Matrix: Unweighted

```
vector<vector<int>>
G { {0, 1, 1, 0, 0, 0},
    {1, 0, 1, 1, 0, 1},
    {1, 1, 0, 0, 1, 0},
    {0, 1, 0, 0, 1, 1},
    {0, 0, 1, 1, 0, 0},
    {0, 1, 0, 1, 0, 0} };
```

Adj. Matrices and Lists in C++

Use 2-dimensional `vector` of `ints`

Matrix: Weighted

```
vector<vector<int>>
G { {0, 1, 7, 2, 0, 5},
    {1, 0, 0, 0, 0, 0},
    {7, 0, 0, 0, 8, 0},
    {2, 0, 0, 0, 0, 5},
    {0, 0, 8, 0, 0, 2},
    {5, 0, 0, 5, 2, 0} };
```

Matrix: Unweighted

```
vector<vector<int>>
G { {0, 1, 1, 0, 0, 0},
    {1, 0, 1, 1, 0, 1},
    {1, 1, 0, 0, 1, 0},
    {0, 1, 0, 0, 1, 1},
    {0, 0, 1, 1, 0, 0},
    {0, 1, 0, 1, 0, 0} };
```

List: Unweighted

```
vector<vector<int>> G
{{1,2},{0,2,3,5},{0,1,4},{1,4,5},{2,3},{1,3}};
```

Adj. Matrices and Lists in C++

Use 2-dimensional `vector` of `ints`

Matrix: Weighted

```
vector<vector<int>>
G { {0, 1, 7, 2, 0, 5},
    {1, 0, 0, 0, 0, 0},
    {7, 0, 0, 0, 8, 0},
    {2, 0, 0, 0, 0, 5},
    {0, 0, 8, 0, 0, 2},
    {5, 0, 0, 5, 2, 0} };
```

Matrix: Unweighted

```
vector<vector<int>>
G { {0, 1, 1, 0, 0, 0},
    {1, 0, 1, 1, 0, 1},
    {1, 1, 0, 0, 1, 0},
    {0, 1, 0, 0, 1, 1},
    {0, 0, 1, 1, 0, 0},
    {0, 1, 0, 1, 0, 0} };
```

List: Unweighted (We prefer those)

```
vector<vector<int>> G
{{1,2},{0,2,3,5},{0,1,4},{1,4,5},{2,3},{1,3}};
```

Stacks

For iterative DFS

Stacks

- Last-In First-Out

Stacks

- Last-In First-Out
- `stack<int>` creates stack of integers

Stacks

- Last-In First-Out
- `stack<int>` creates stack of integers
- `push(x)` “pushes” `x` into end

Stacks

- Last-In First-Out
- `stack<int>` creates stack of integers
- `push(x)` “pushes” `x` into end
- `top()` returns last element

Stacks

- Last-In First-Out
- `stack<int>` creates stack of integers
- `push(x)` “pushes” `x` into end
- `top()` returns last element
- `pop()` “pops” out last element

Stacks

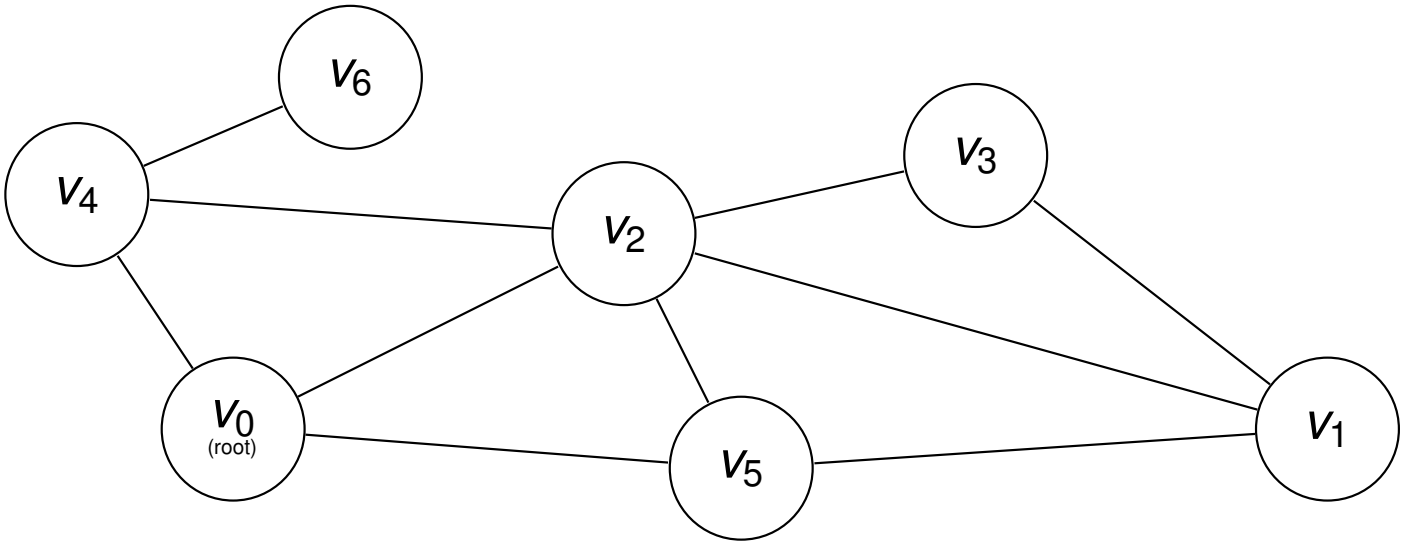
- Last-In First-Out
- `stack<int>` creates stack of integers
- `push(x)` “pushes” `x` into end
- `top()` returns last element
- `pop()` “pops” out last element

```
stack<int> numbers;
numbers.push(0);
numbers.push(4);
numbers.push(7);
cout << numbers.top() << "\n";
numbers.pop();
cout << numbers.top() << "\n";
```

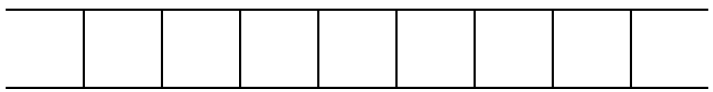
Depth-First Search

in undirected graphs

DFS with a Stack

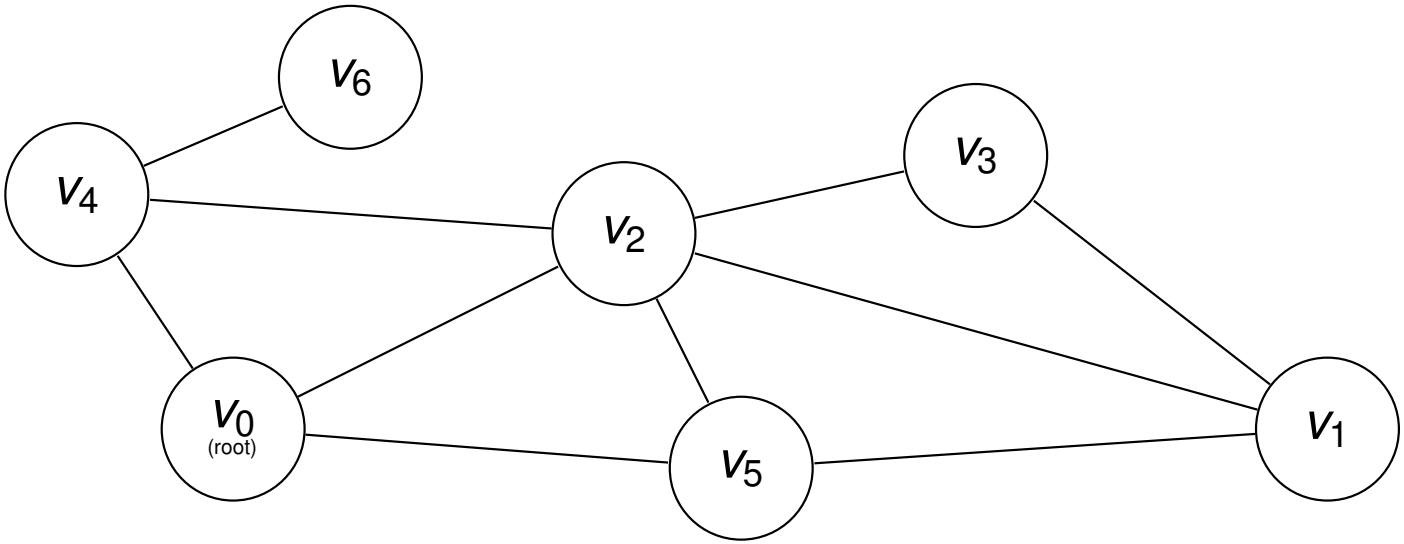


Stack:

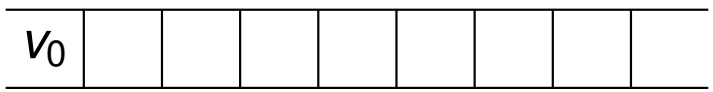


Output:

DFS with a Stack

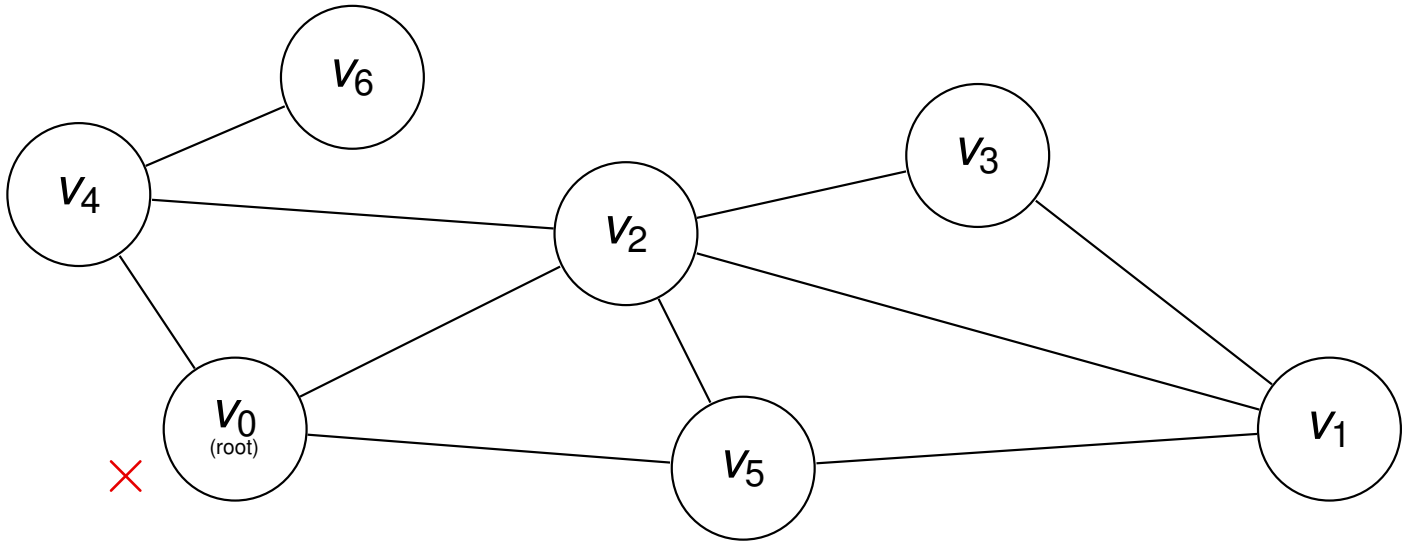


Stack:

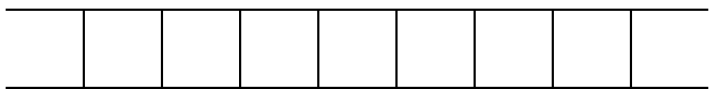


Output:

DFS with a Stack

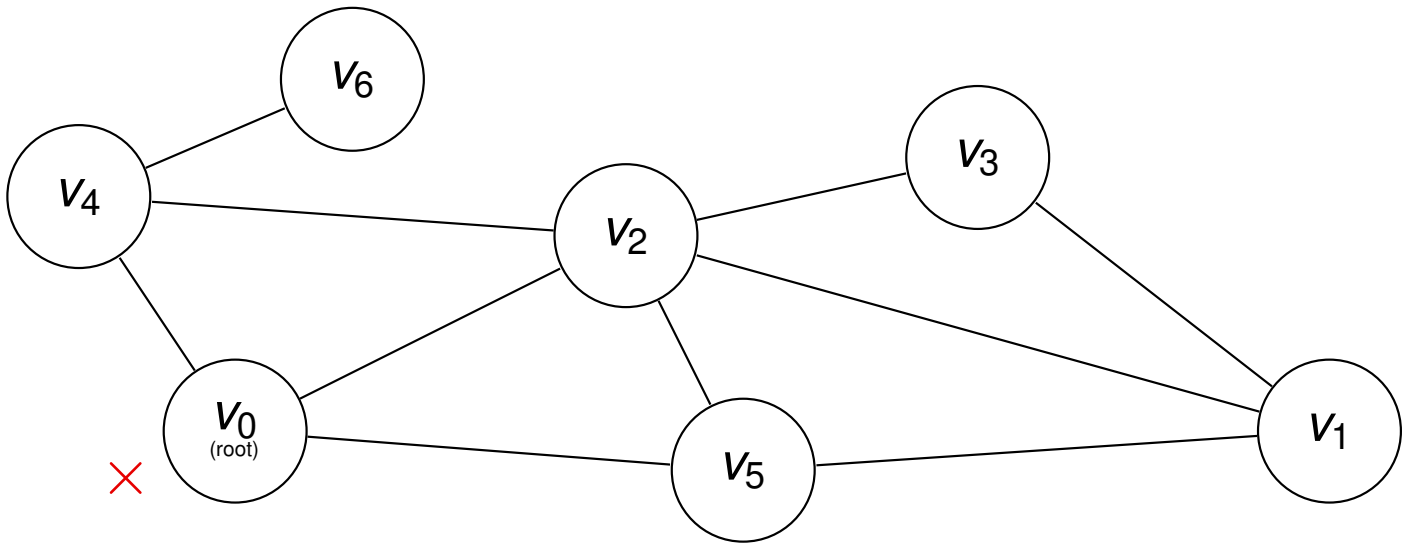


Stack:



Output: v_0

DFS with a Stack

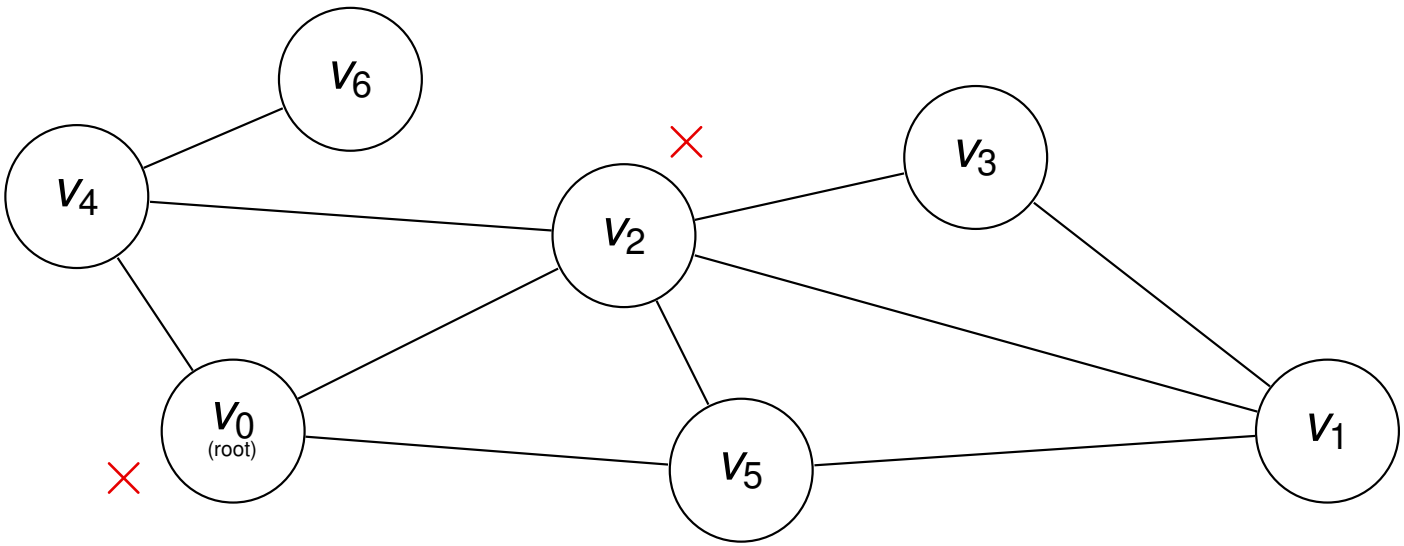


Stack:

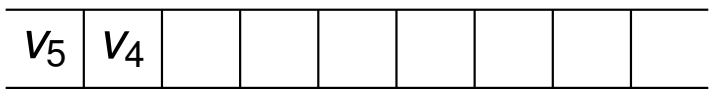
v_5	v_4	v_2						

Output: v_0

DFS with a Stack

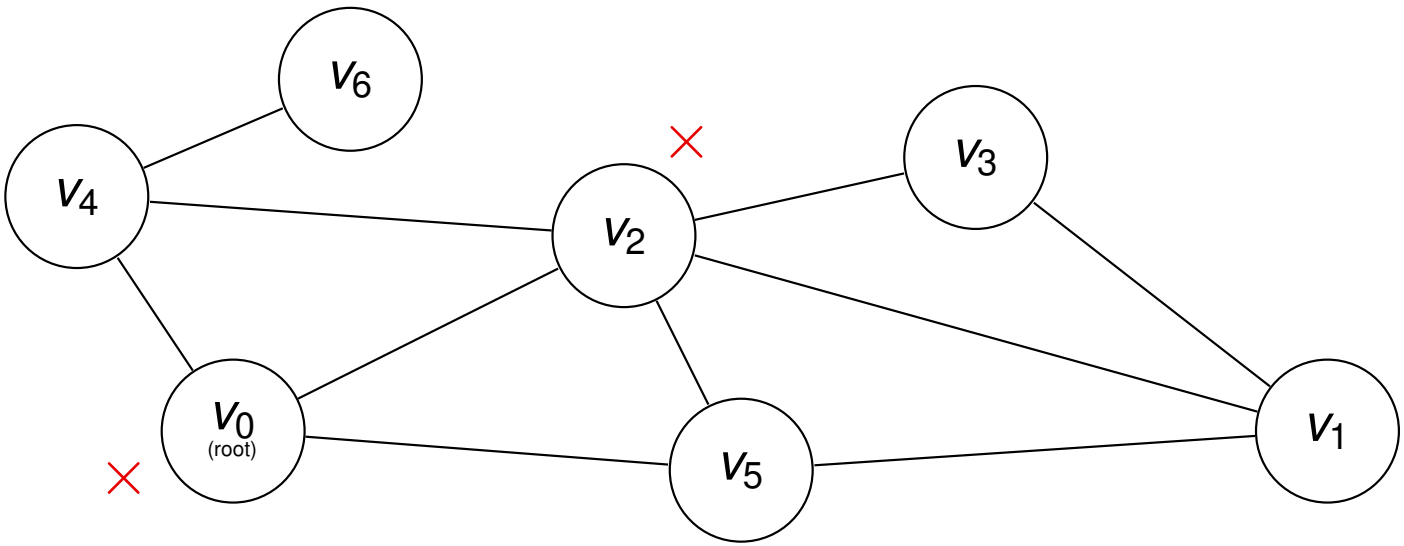


Stack:



Output: v_0 v_2

DFS with a Stack

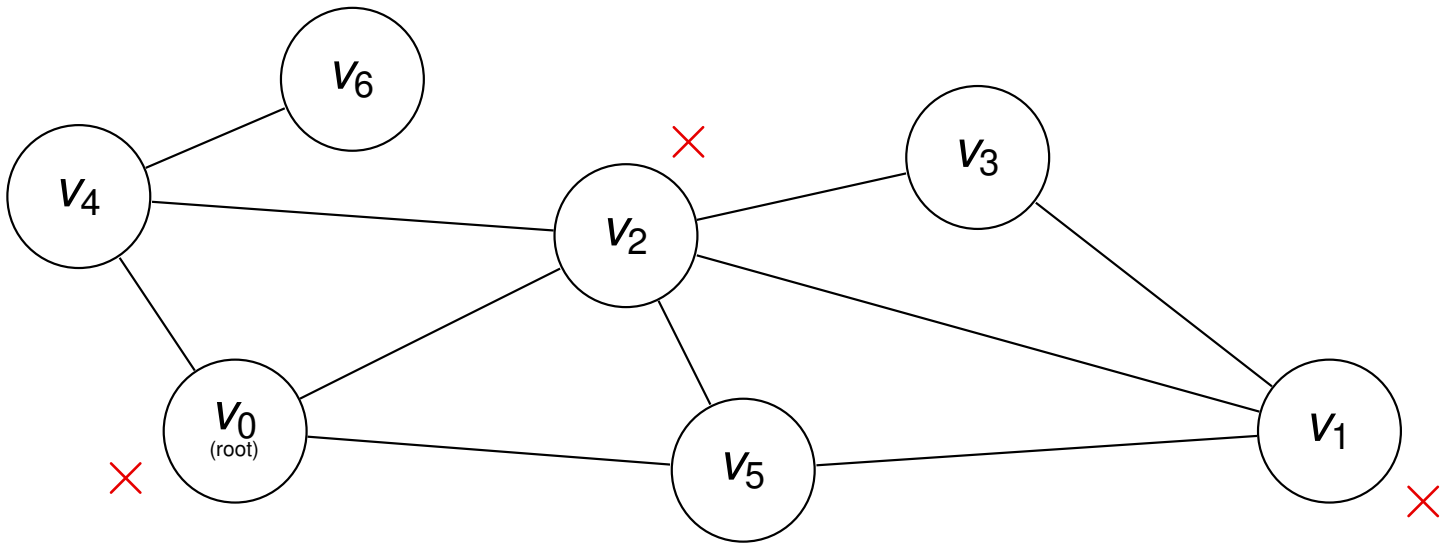


Stack:

v_5	v_4	v_5	v_4	v_3	v_1			

Output: v_0 v_2

DFS with a Stack

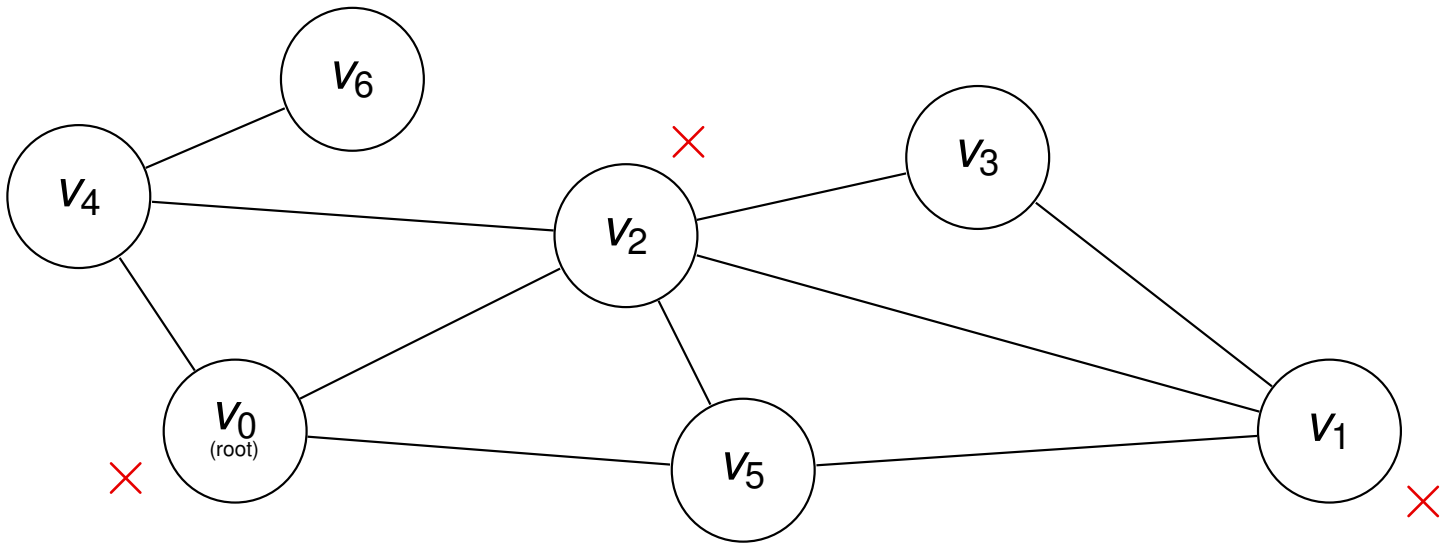


Stack:

v_5	v_4	v_5	v_4	v_3				

Output: v_0 v_2 v_1

DFS with a Stack

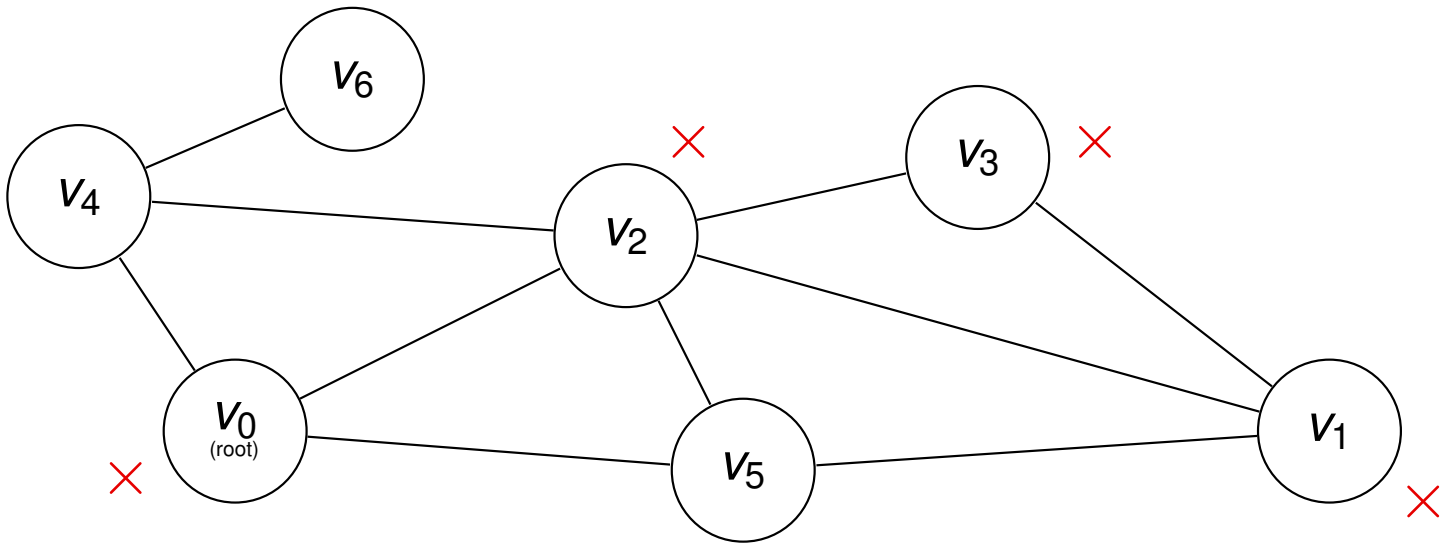


Stack:

v_5	v_4	v_5	v_4	v_3	v_5	v_3		

Output: v_0 v_2 v_1

DFS with a Stack

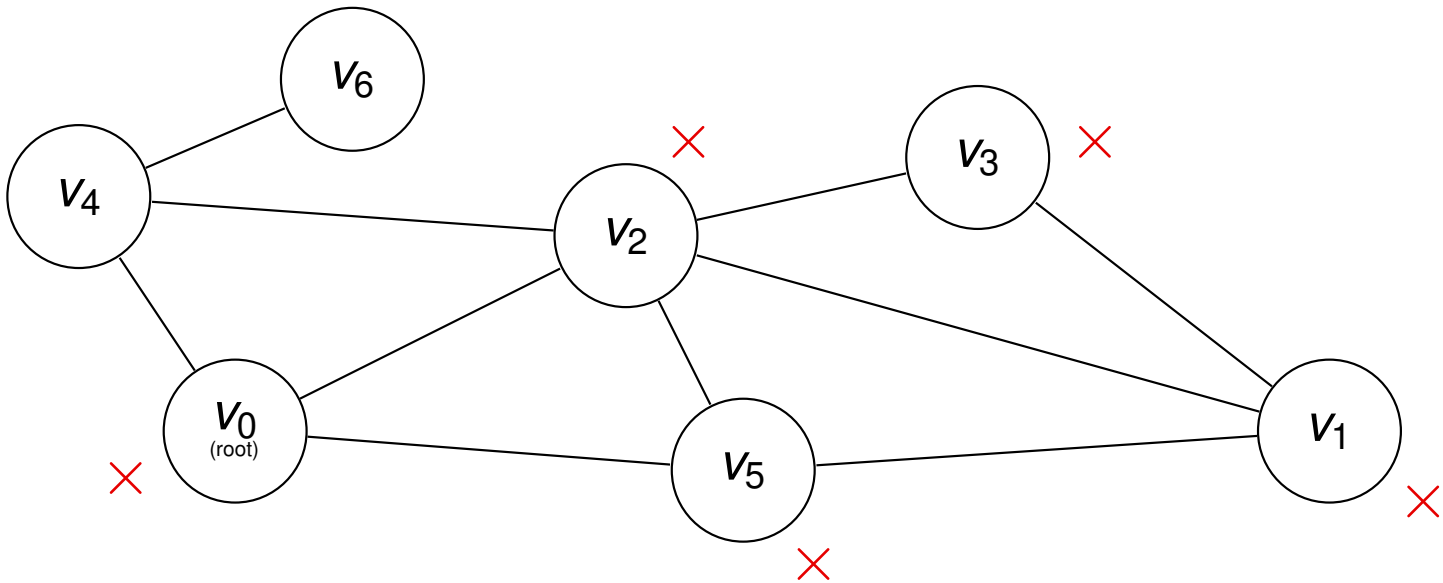


Stack:

v_5	v_4	v_5	v_4	v_3	v_5			

Output: v_0 v_2 v_1 v_3

DFS with a Stack

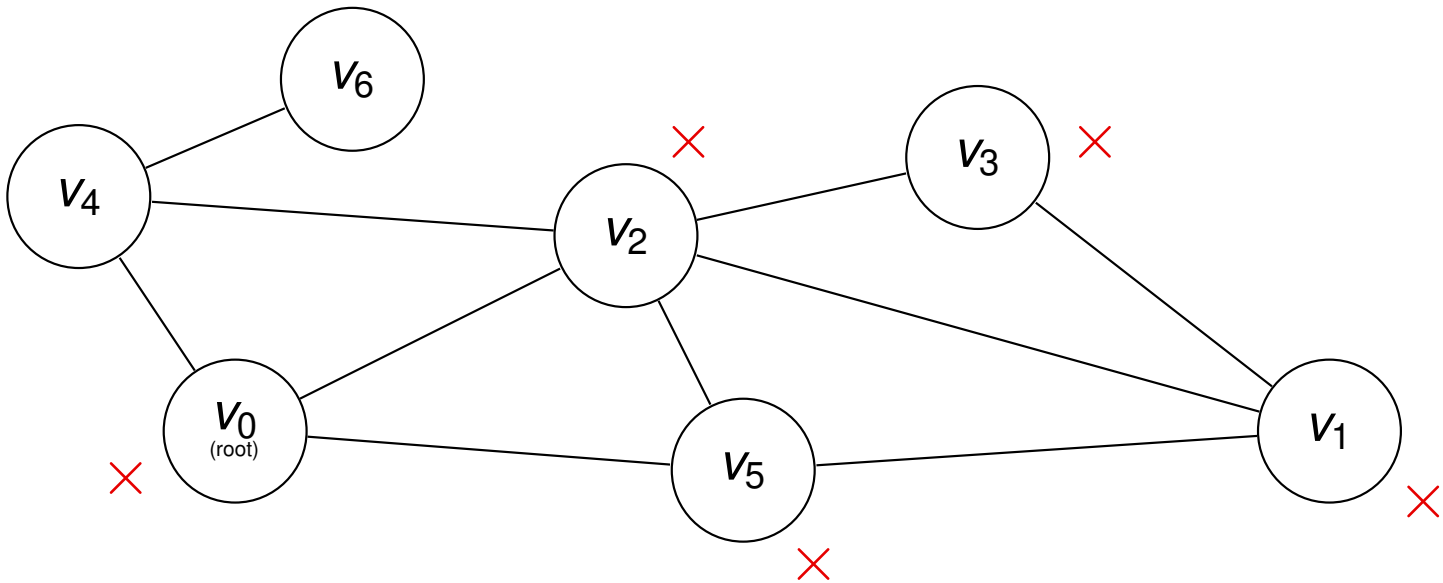


Stack:

v_5	v_4	v_5	v_4	v_3				

Output: v_0 v_2 v_1 v_3 v_5

DFS with a Stack

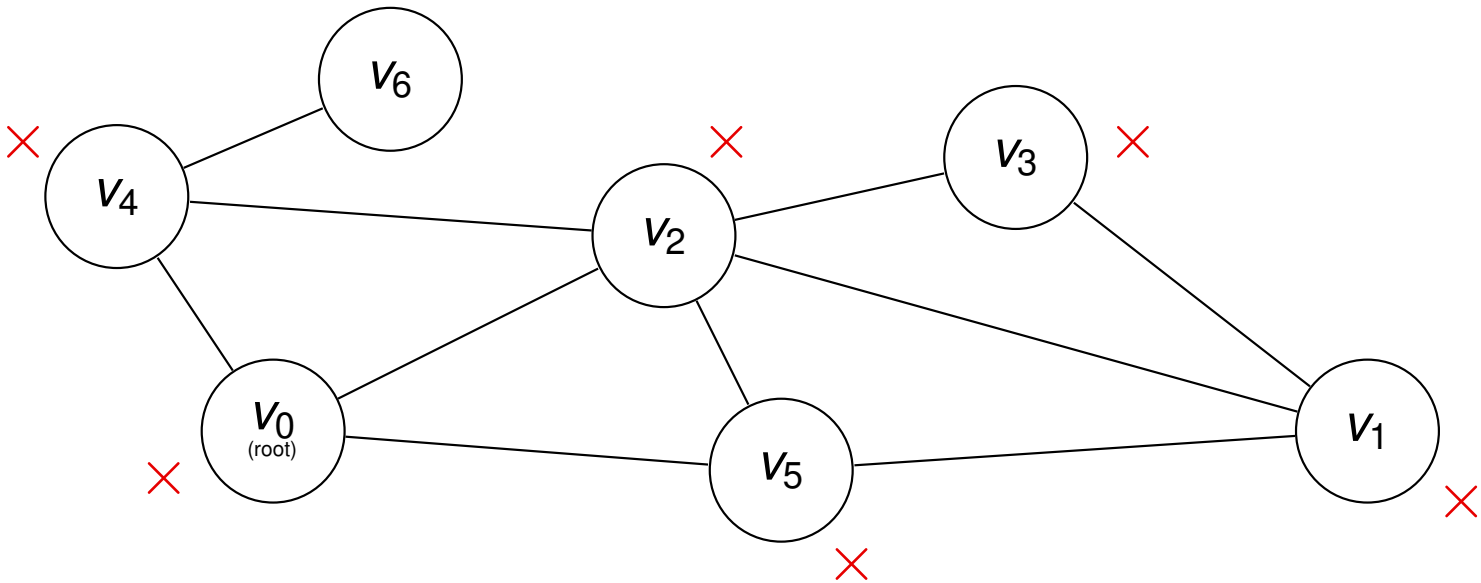


Stack:

v_5	v_4	v_5	v_4					

Output: v_0 v_2 v_1 v_3 v_5

DFS with a Stack

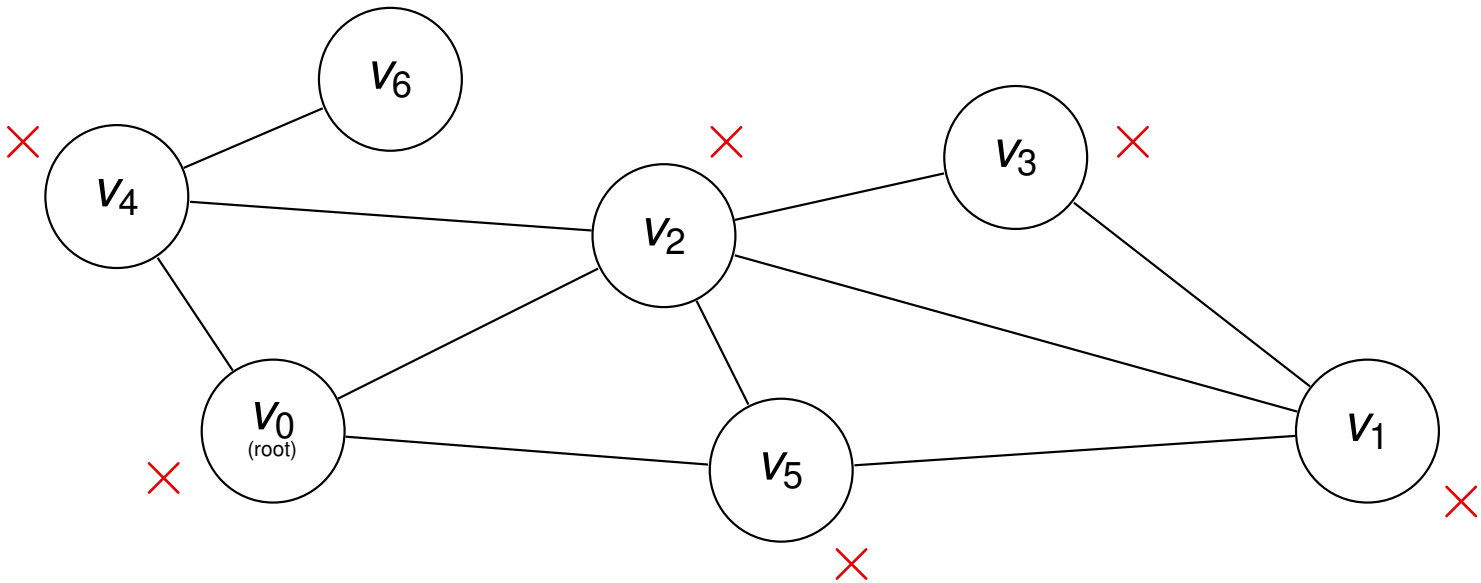


Stack:

v_5	v_4	v_5						

Output: v_0 v_2 v_1 v_3 v_5 v_4

DFS with a Stack

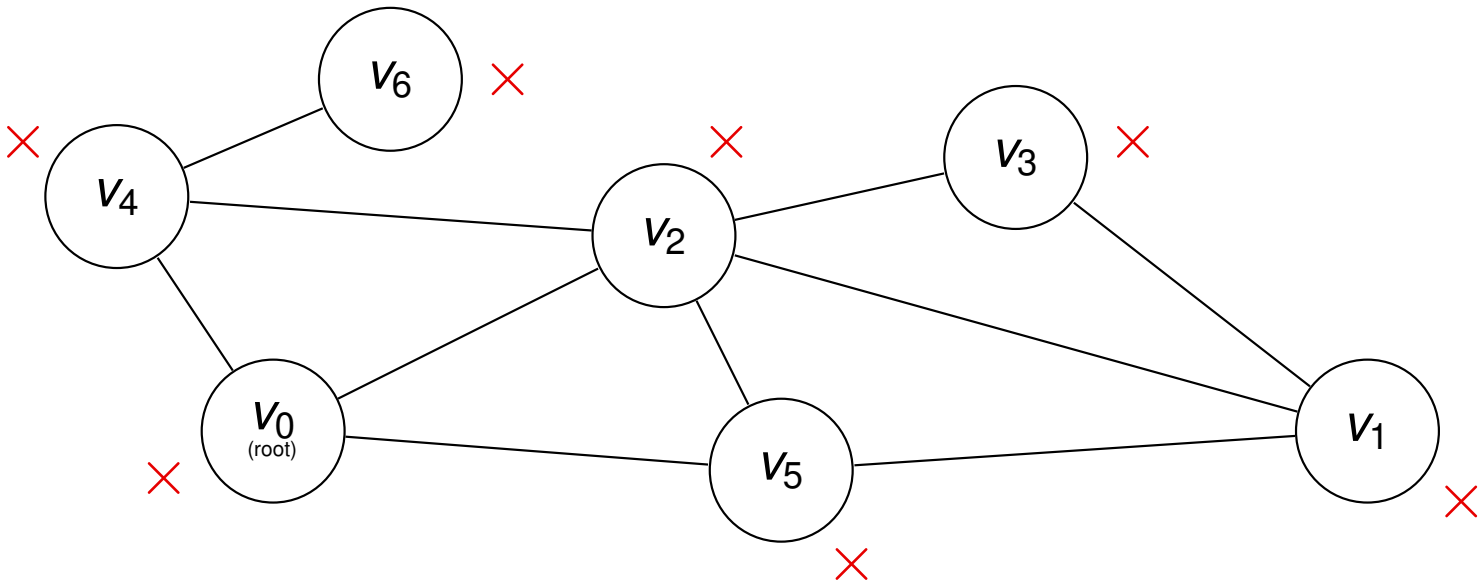


Stack:

v_5	v_4	v_5	v_6					

Output: v_0 v_2 v_1 v_3 v_5 v_4

DFS with a Stack

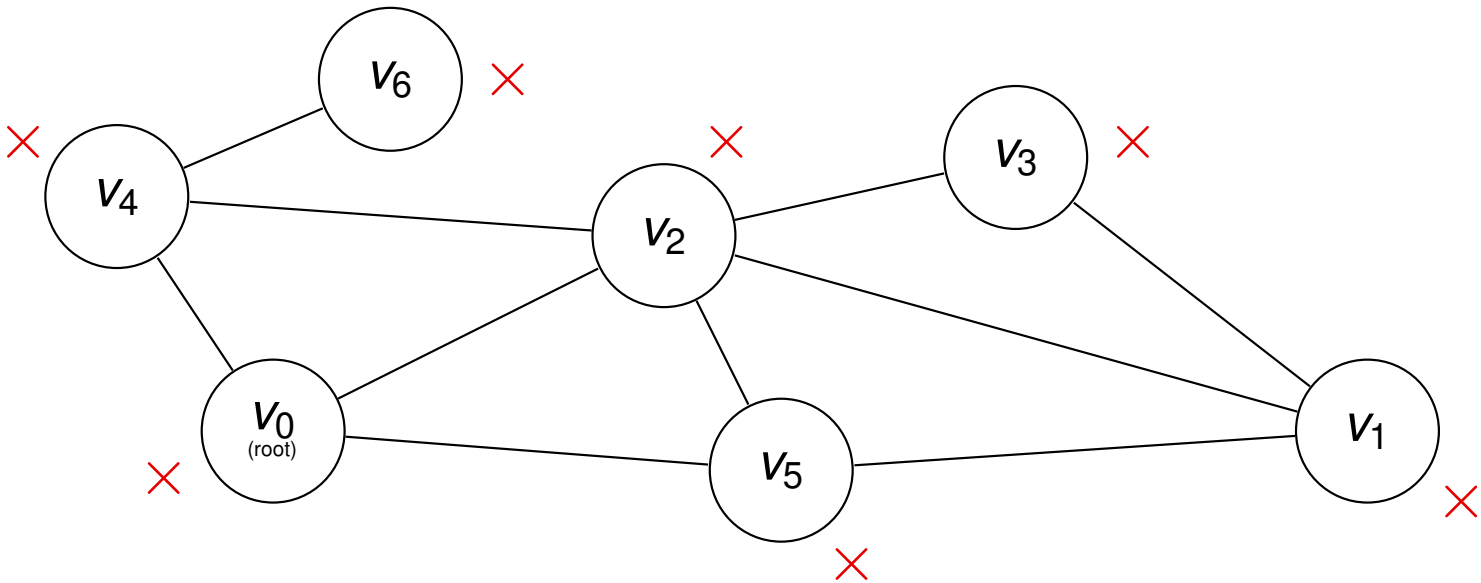


Stack:

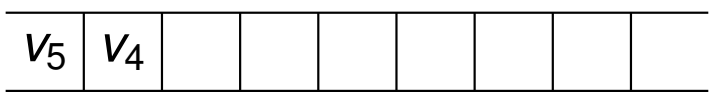
v_5	v_4	v_5						

Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

DFS with a Stack

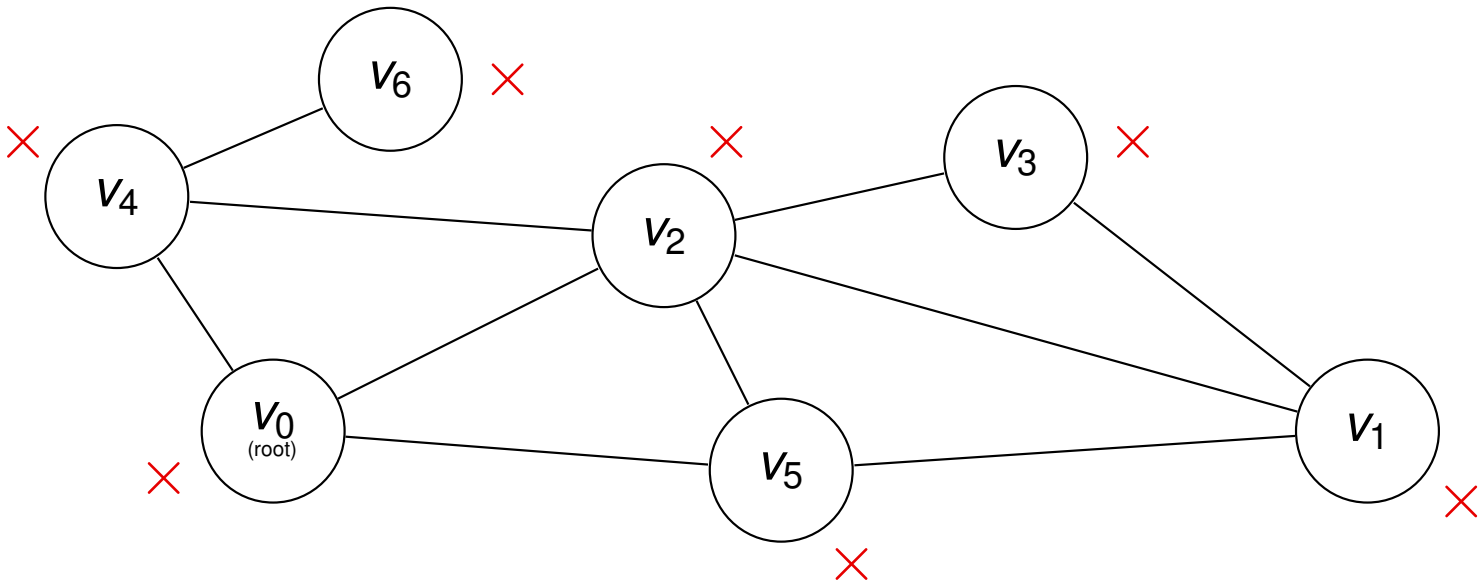


Stack:

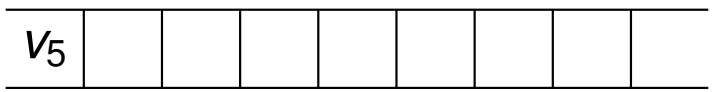


Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

DFS with a Stack

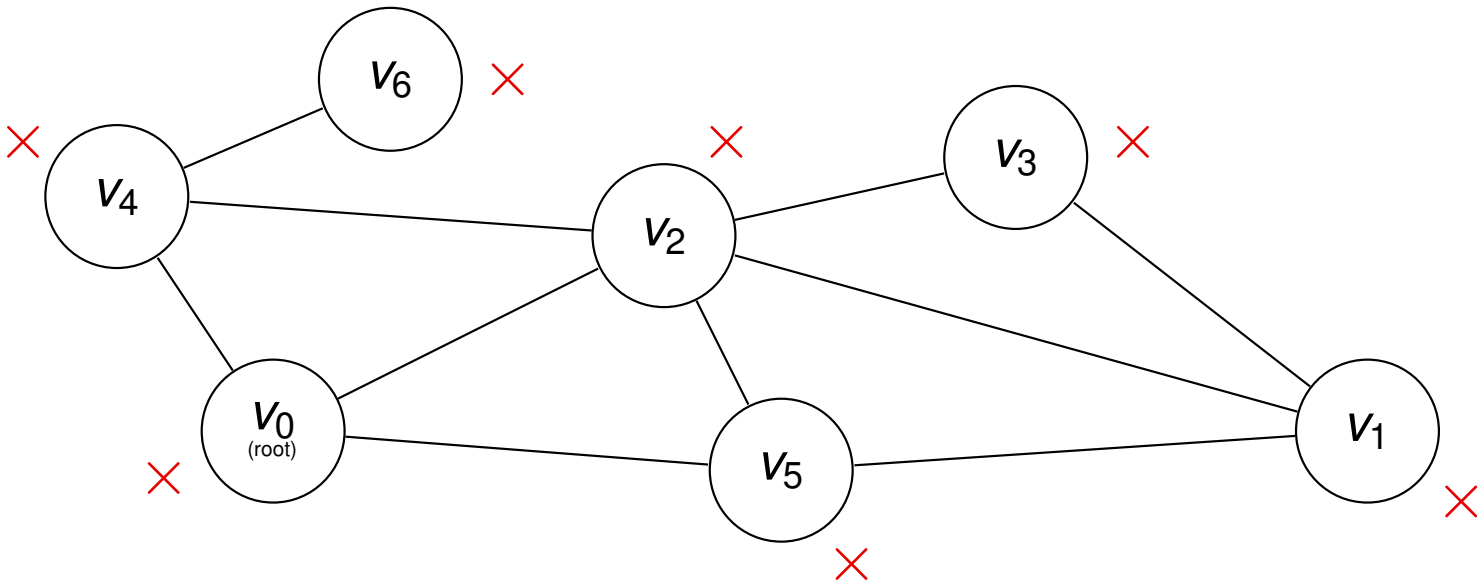


Stack:

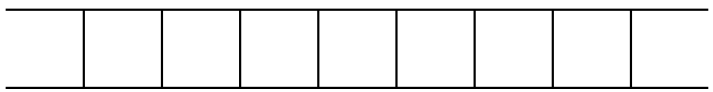


Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

DFS with a Stack



Stack:



Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

DFS with a Stack

```
void dfs() {
   stack<int> vertices;
   vertices.push(0);
   int current, v;
   while (vertices.size() > 0) {
      current = vertices.top();
      vertices.pop();
      if (!visited[current]) {
         visited[current] = true;
         cout << current << " ";
         for (int i=G[current].size()-1; i>=0; --i) {
            v = G[current][i];
            vertices.push(v);
         }
      }
   }
}
```

Depth-First Search

in undirected graphs recursively

Recursive DFS

- Again start at root and mark it as visited

Recursive DFS

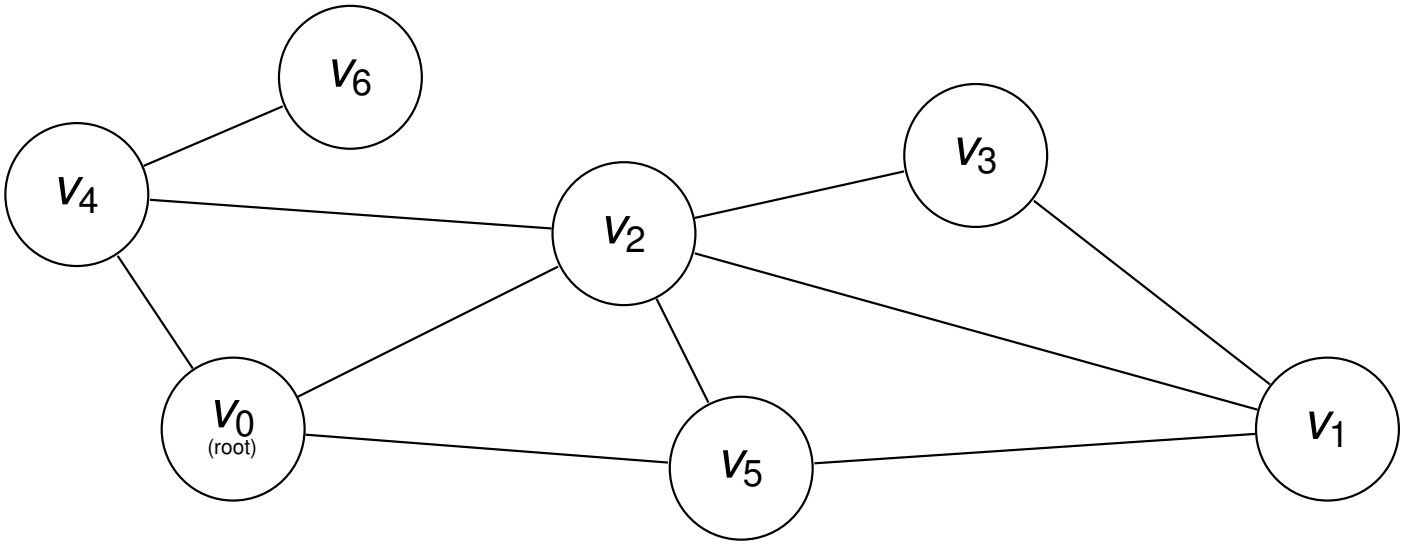
- Again start at root and mark it as visited
- Recursively call algorithm on non-visited neighbors

Recursive DFS

- Again start at root and mark it as visited
- Recursively call algorithm on non-visited neighbors

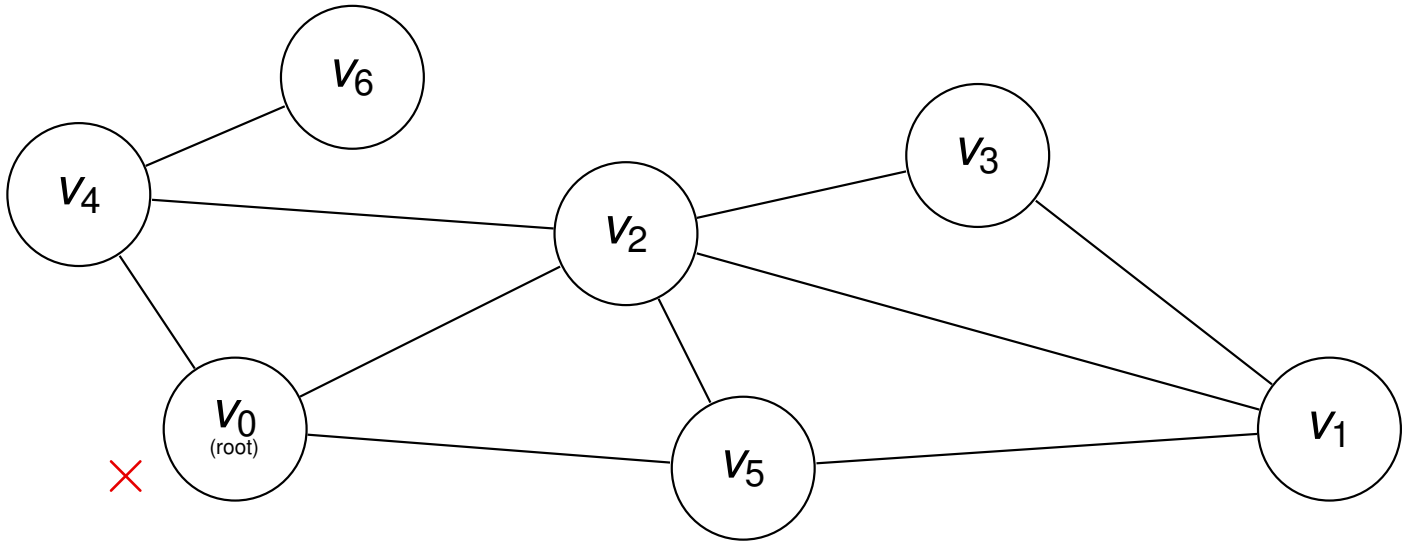
```
vector<vector<int>> G {{1,2},{2},{0}};
vector<int> visited(G.size(),false);

void dfs(int current) {
   if (!visited[current]) {
      visited[current] = true;
      cout << current << " ";
      for (int i=0; i<G[current].size(); ++i) {
         int v = G[current][i];
         dfs(v);
      }
   }
}
```

Recursive DFS



Output:

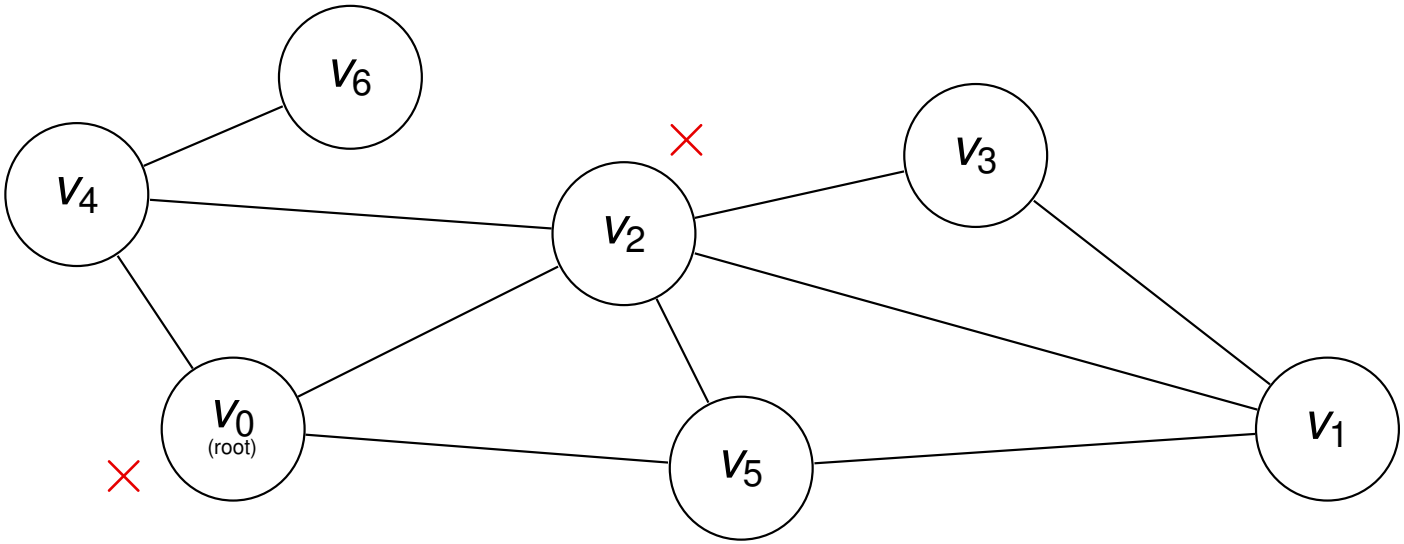
Recursive DFS




```
dfs(0)
```

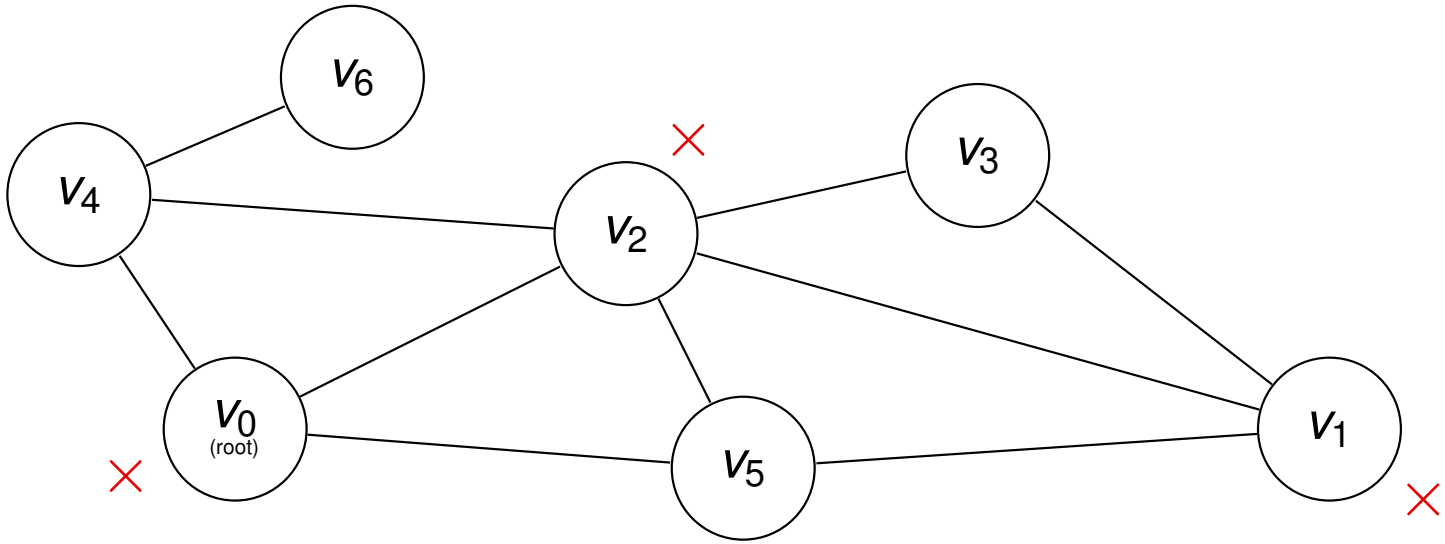
Output: v_0

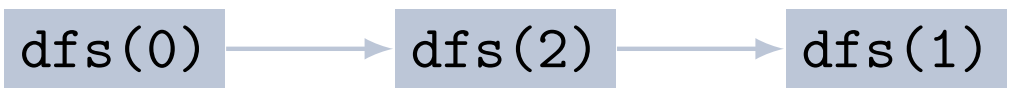
Recursive DFS



Output: v_0 v_2

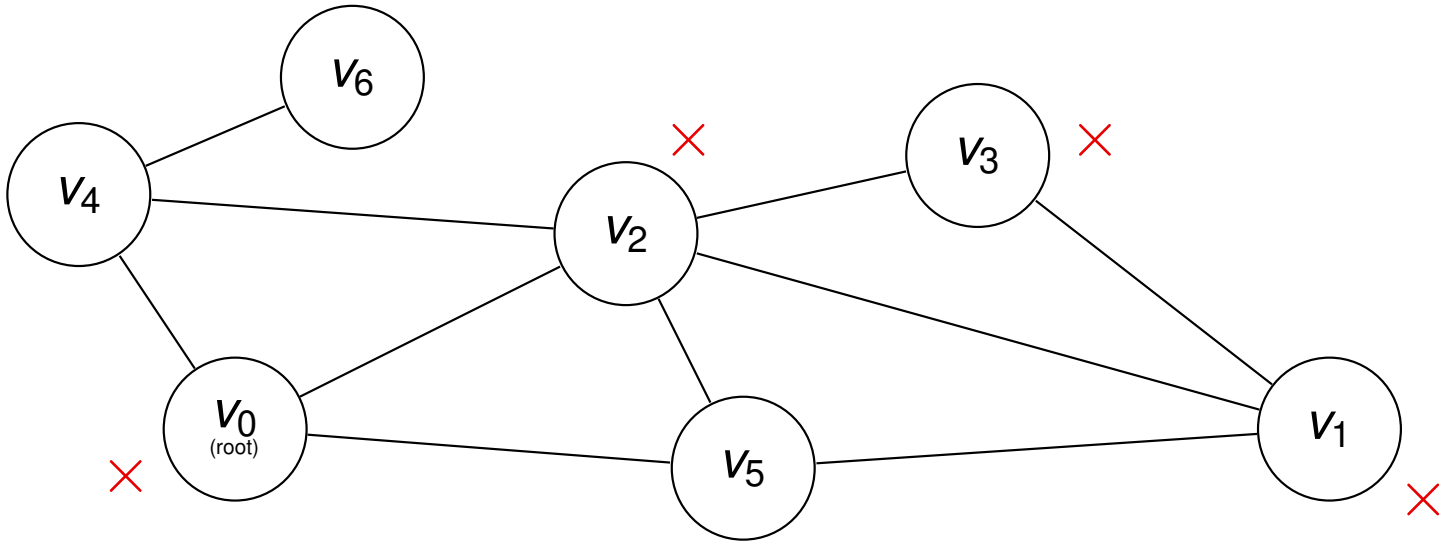
Recursive DFS





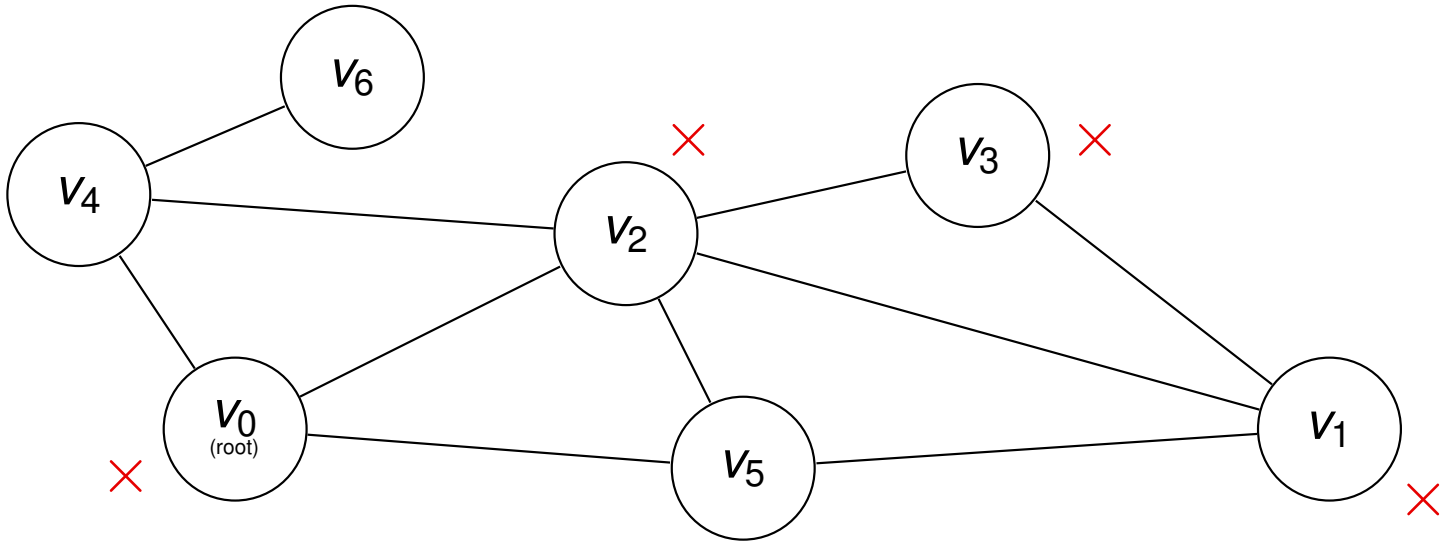
Output: v_0 v_2 v_1

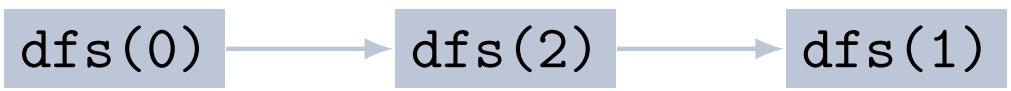
Recursive DFS



Output: v_0 v_2 v_1 v_3

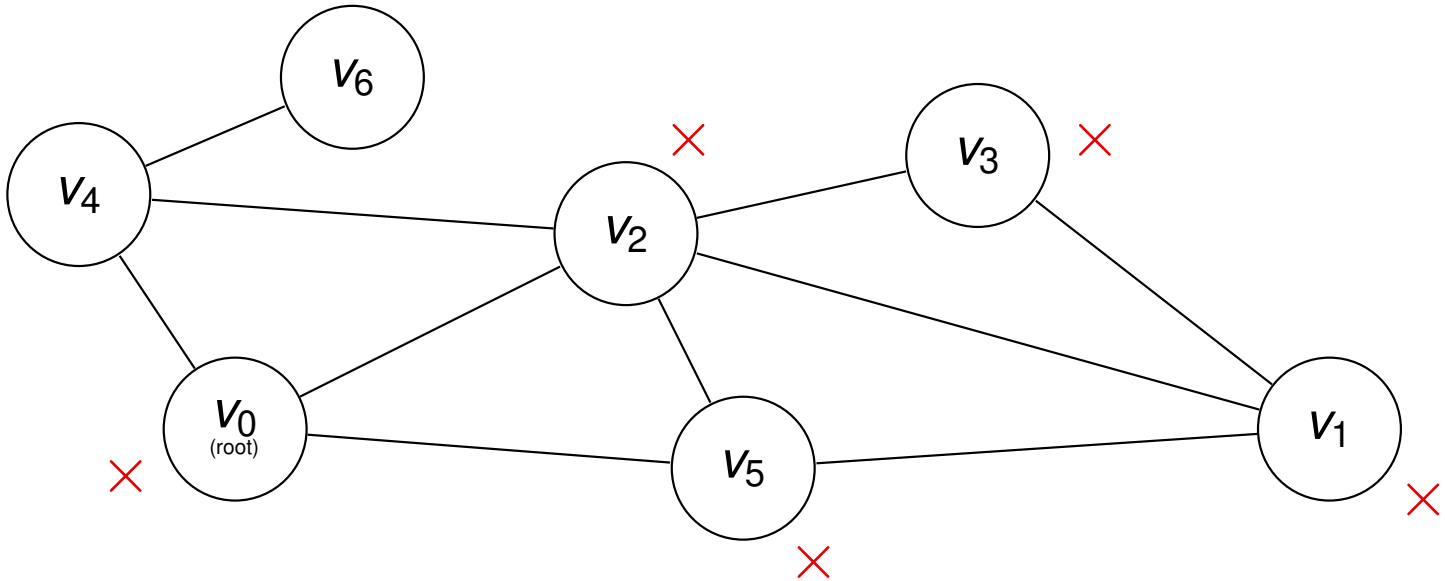
Recursive DFS





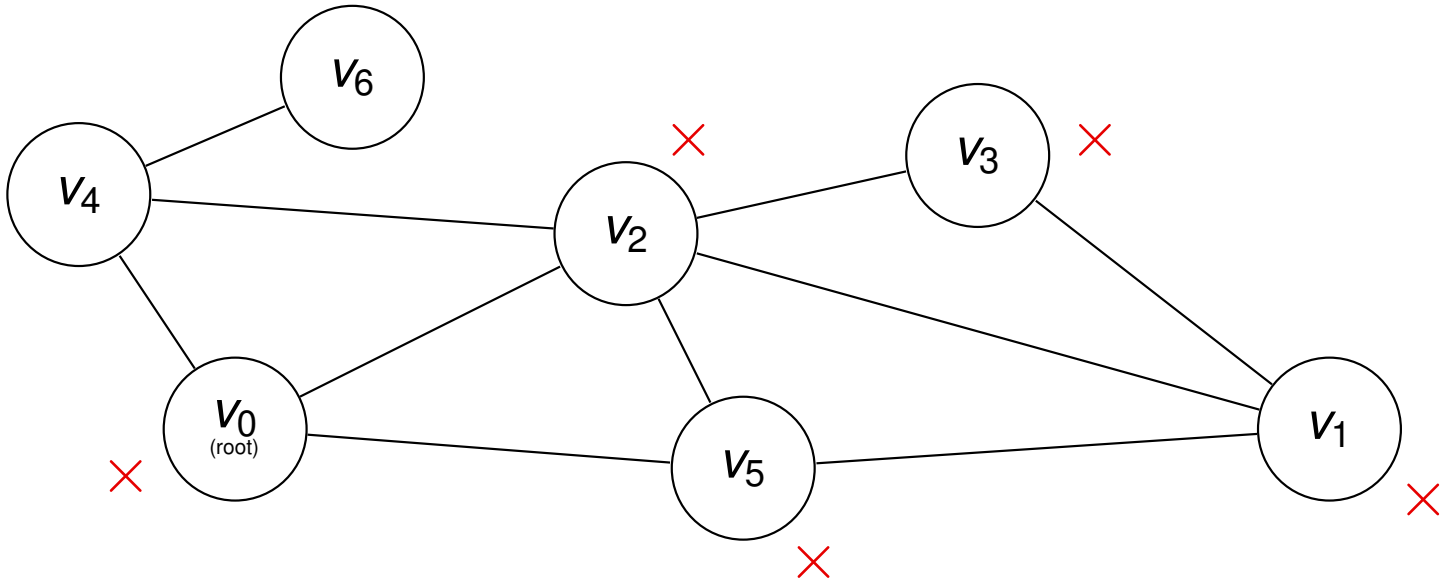
Output: v_0 v_2 v_1 v_3

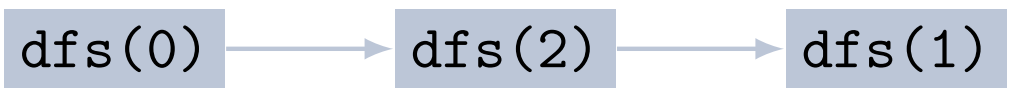
Recursive DFS



Output: v_0 v_2 v_1 v_3 v_5

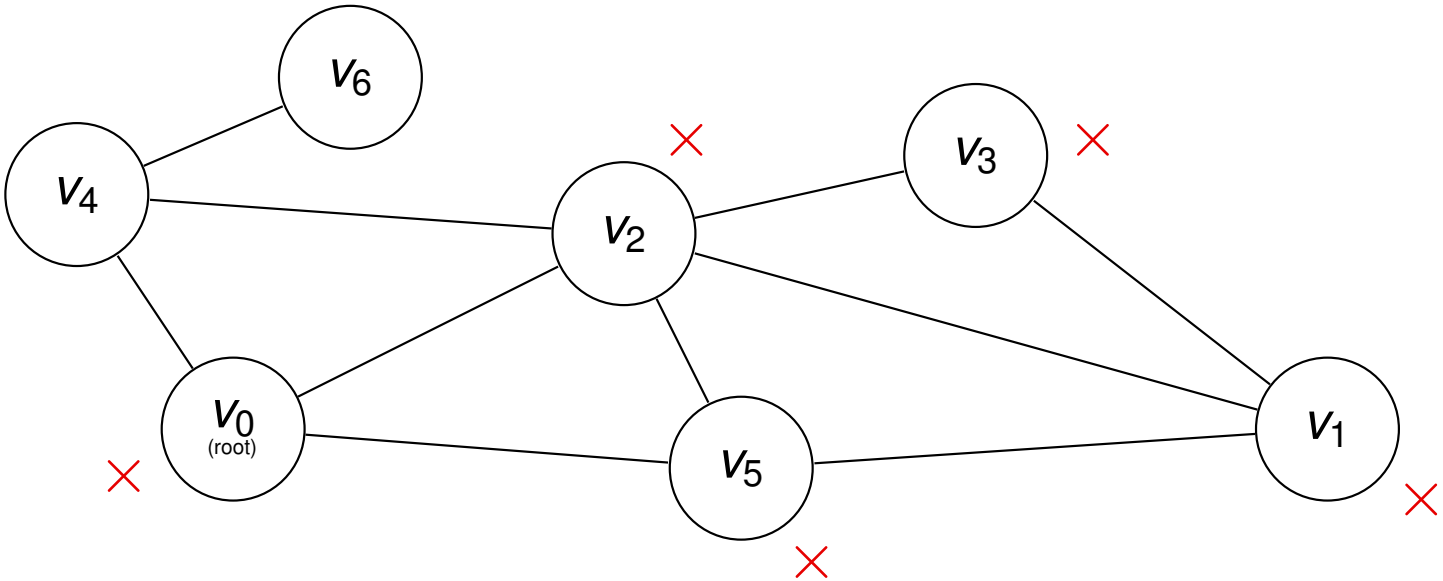
Recursive DFS





Output: v_0 v_2 v_1 v_3 v_5

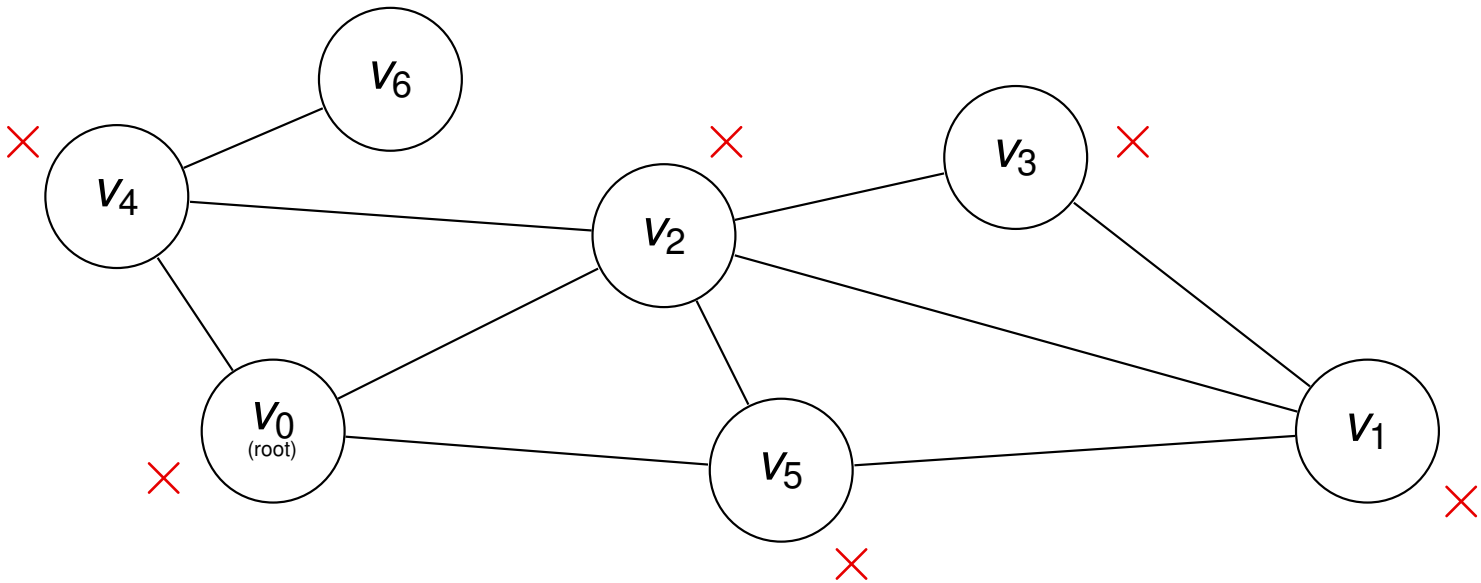
Recursive DFS

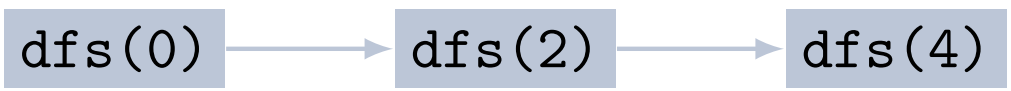



```
dfs(0) → dfs(2)
```

Output: v_0 v_2 v_1 v_3 v_5

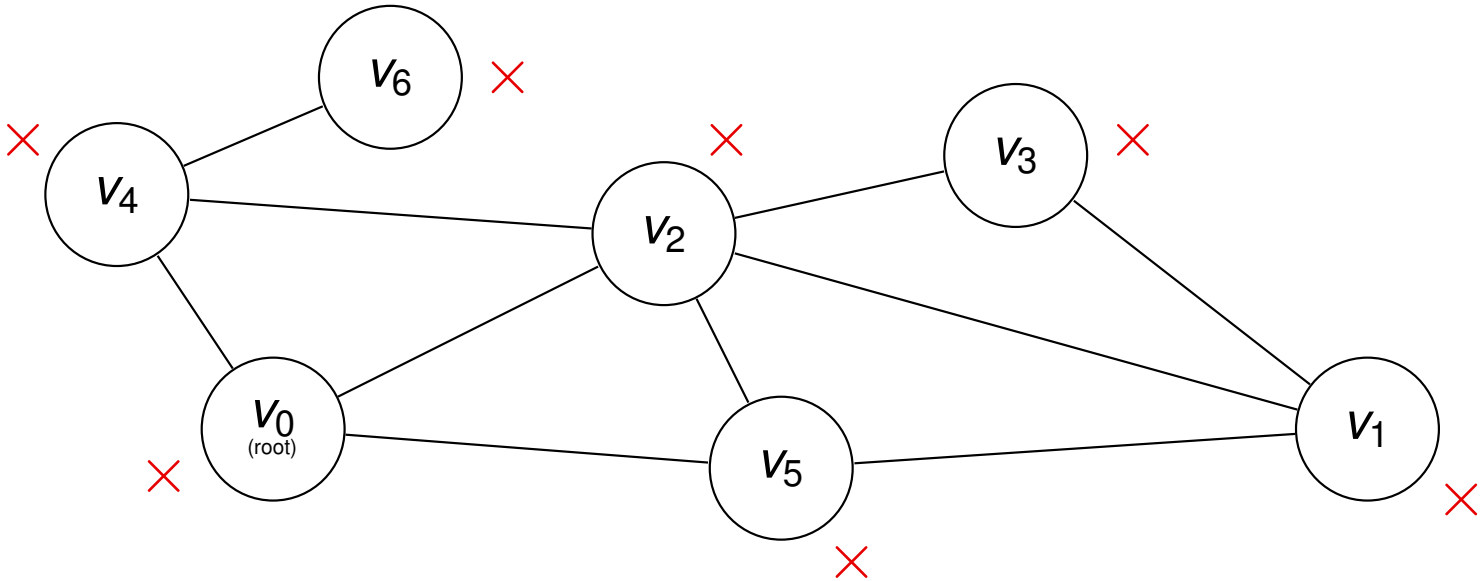
Recursive DFS





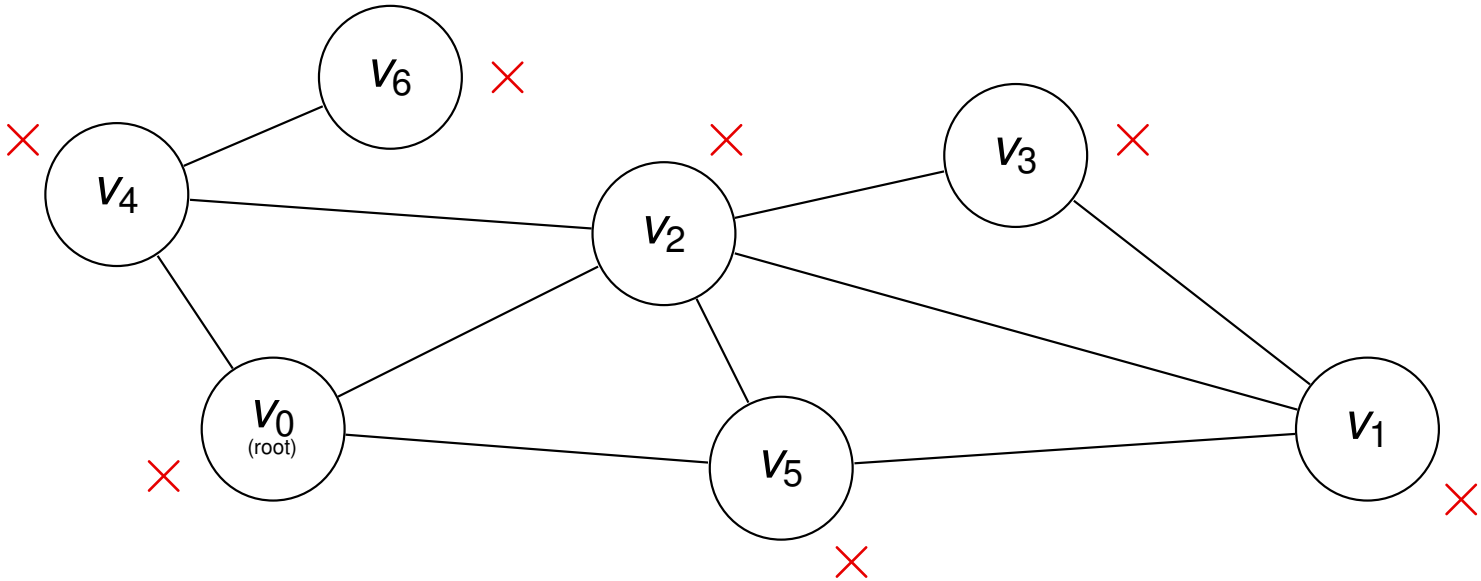
Output: v_0 v_2 v_1 v_3 v_5 v_4

Recursive DFS



Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

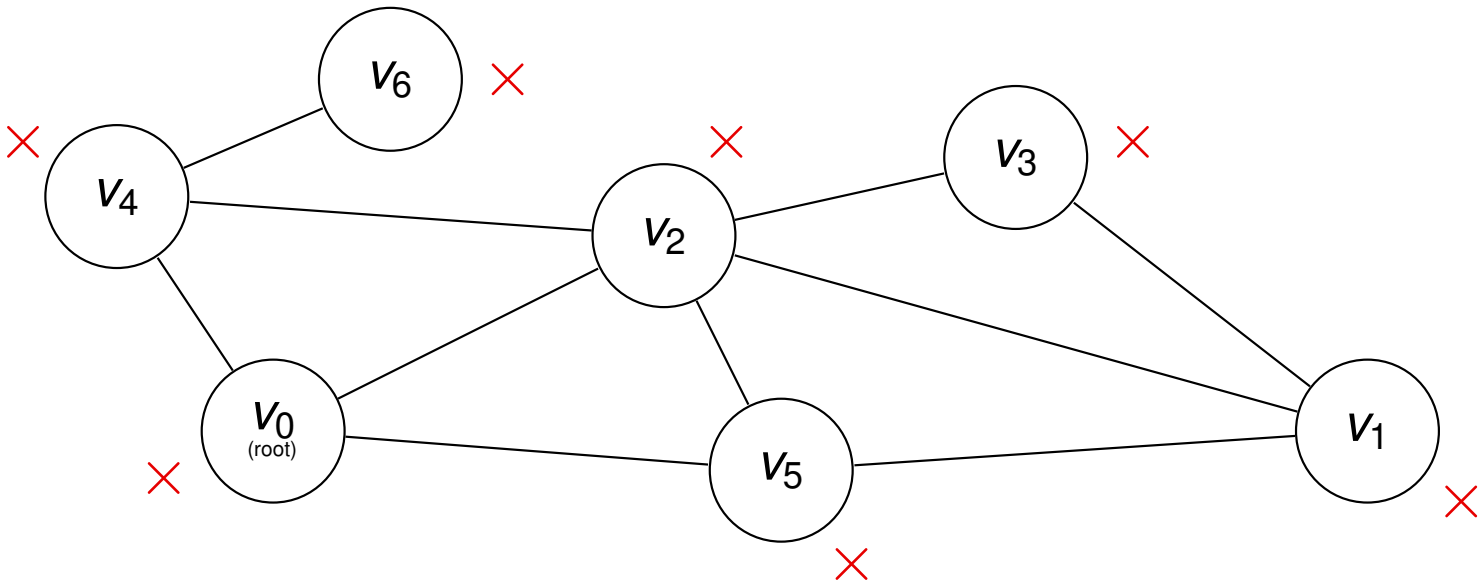
Recursive DFS




```
dfs(0) → dfs(2) → dfs(4)
```

Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

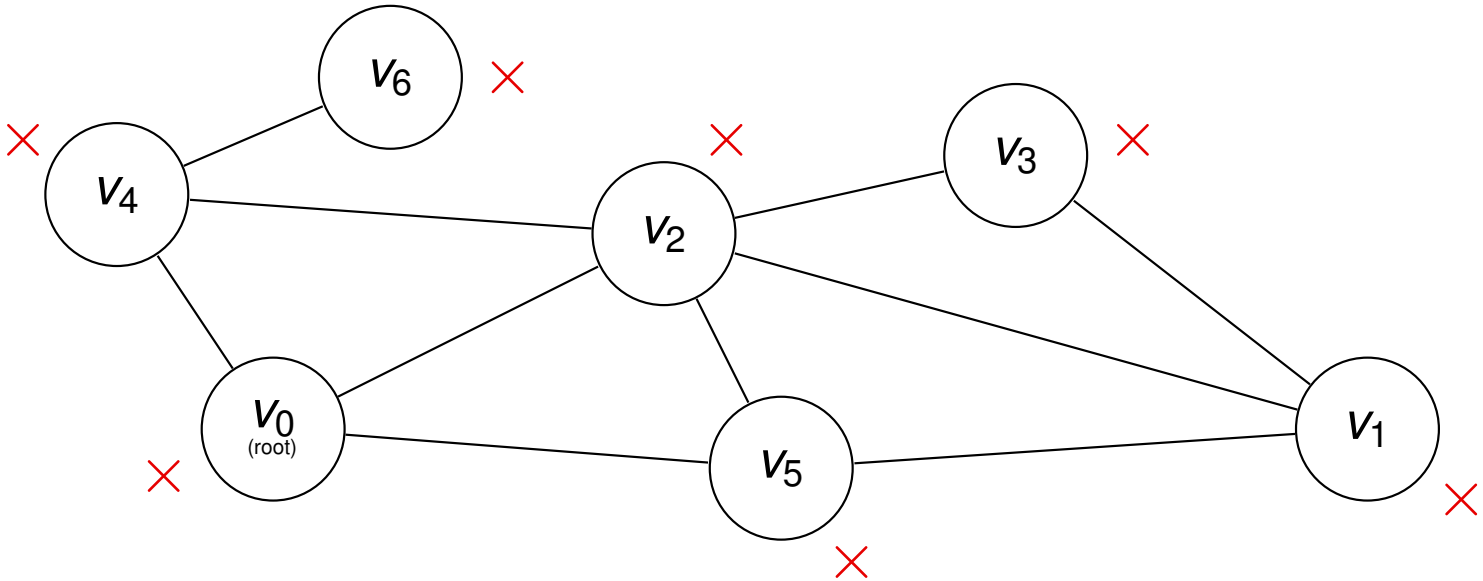
Recursive DFS




```
dfs(0) → dfs(2)
```

Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

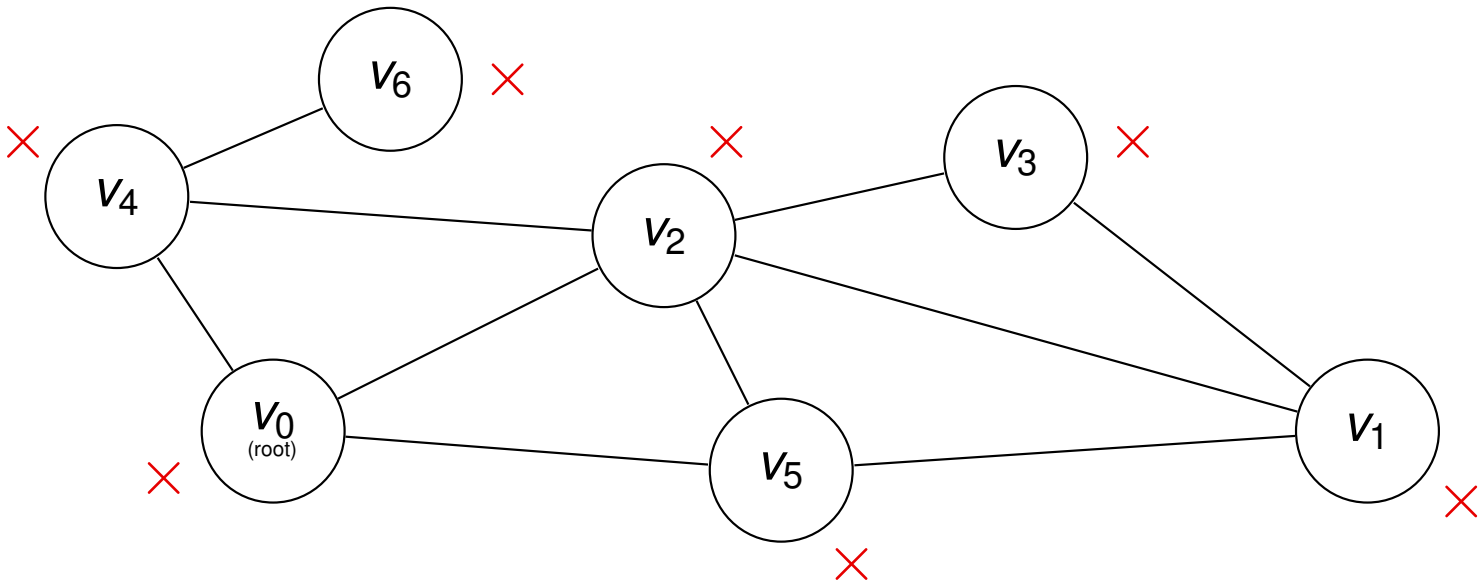
Recursive DFS




```
dfs(0)
```

Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

Recursive DFS

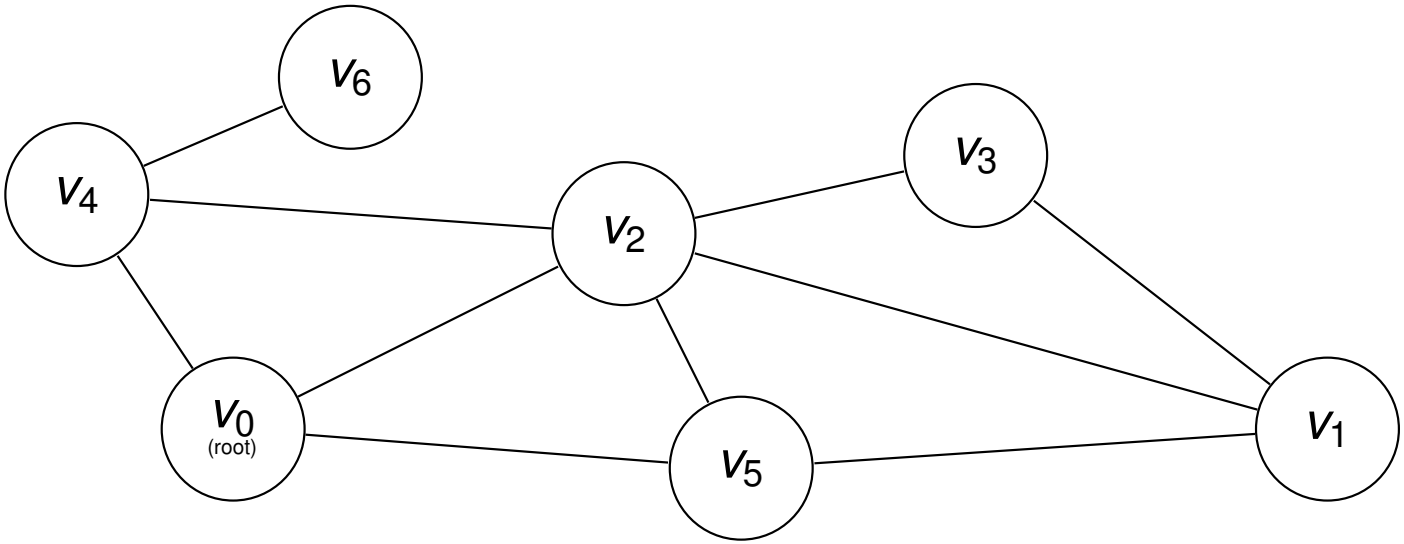


Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

DFS Tree

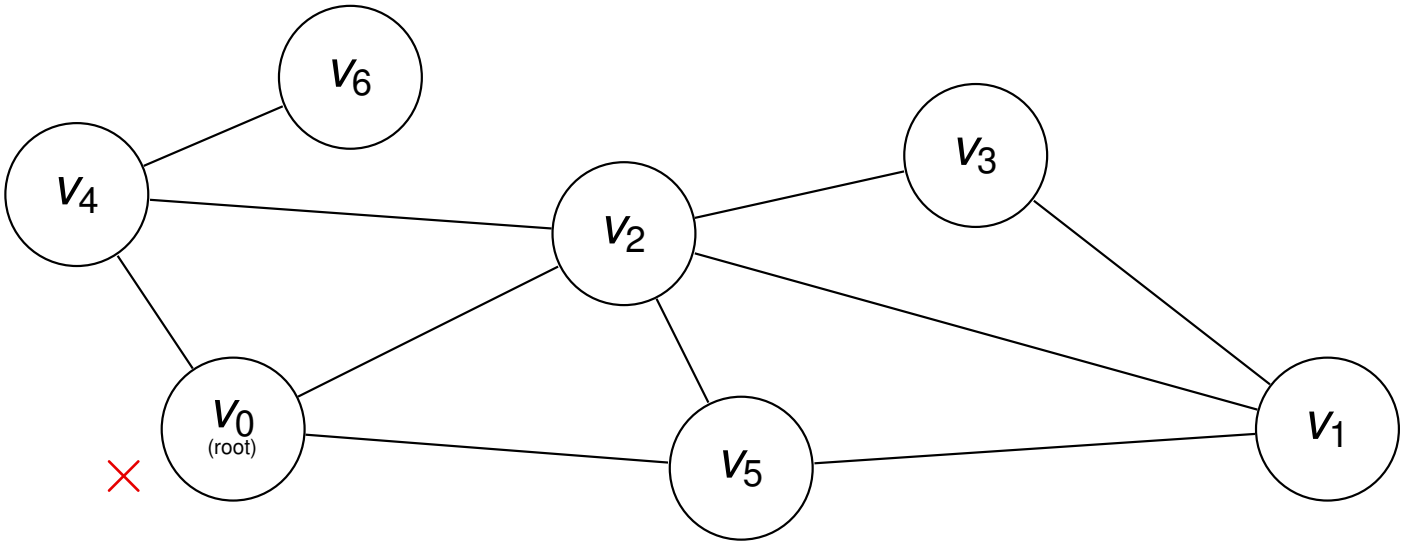
of an undirected graph

DFS Tree



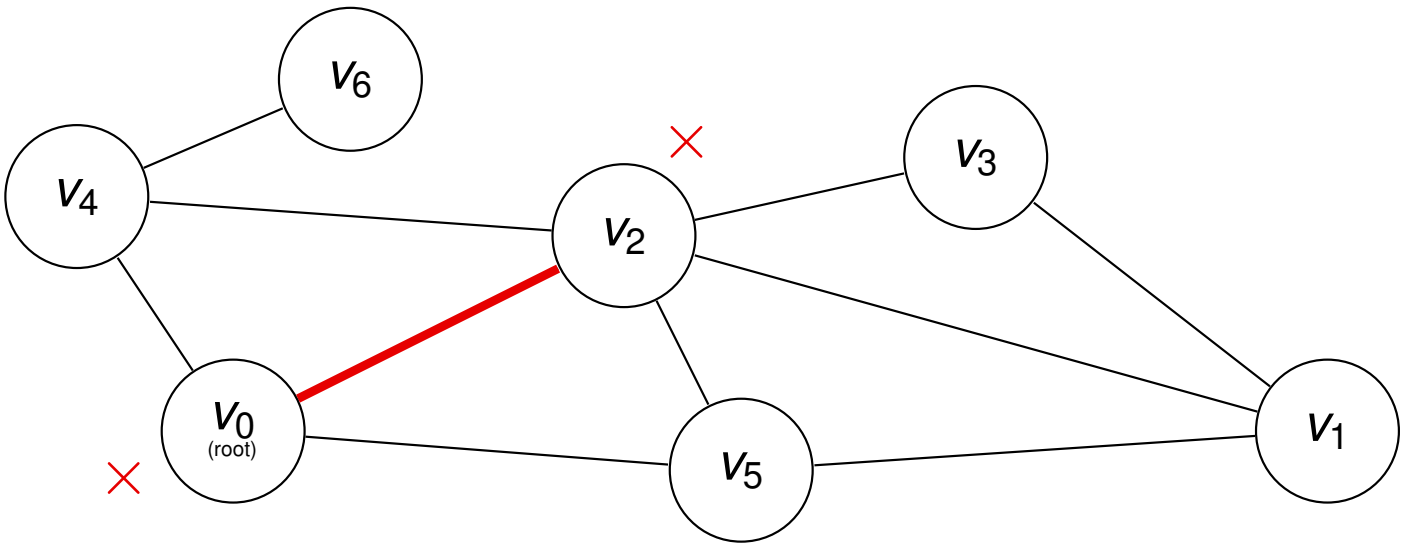
Output:

DFS Tree



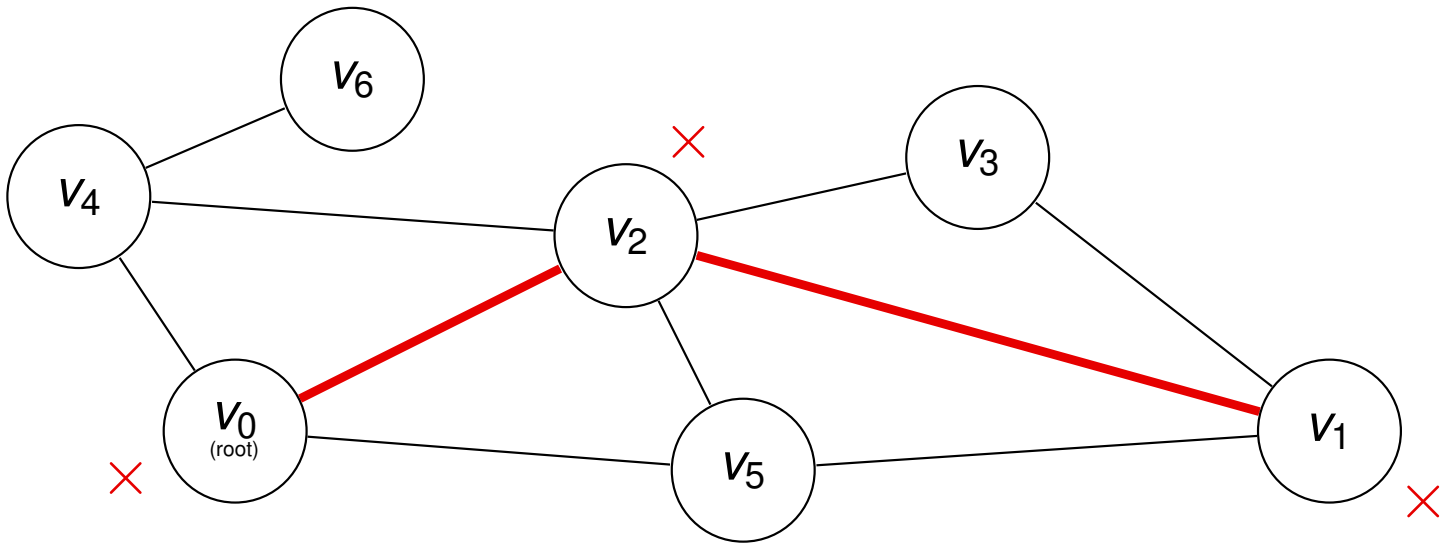
Output: v_0

DFS Tree



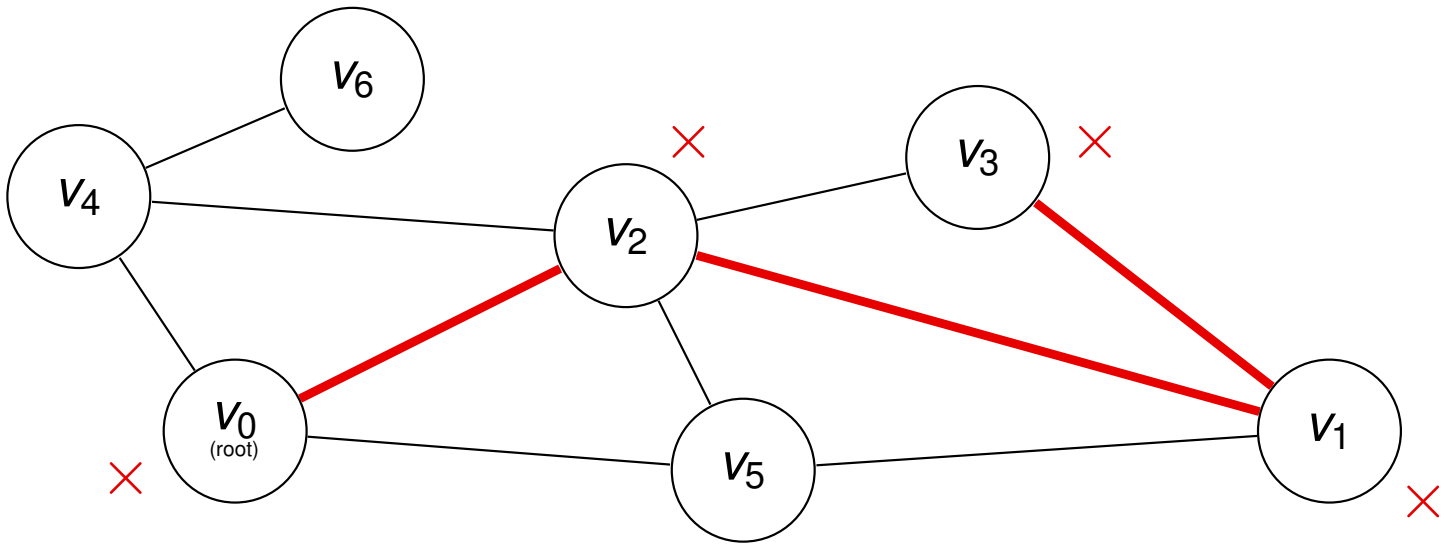
Output: v_0 v_2

DFS Tree



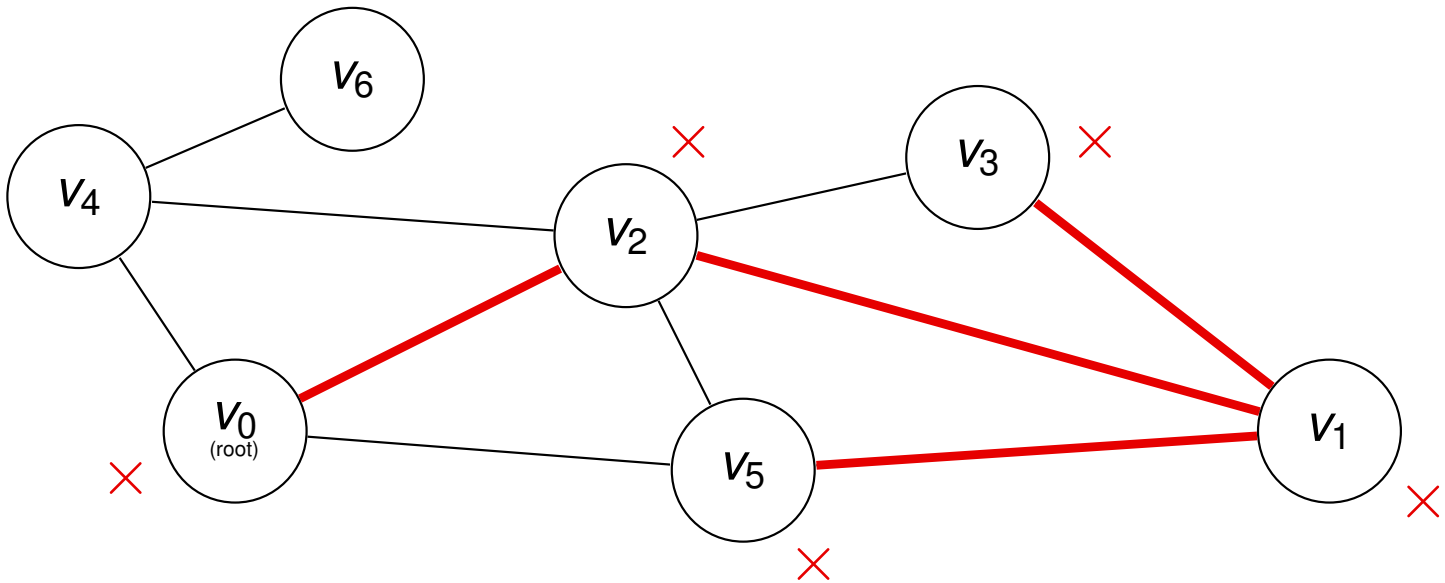
Output: v_0 v_2 v_1

DFS Tree



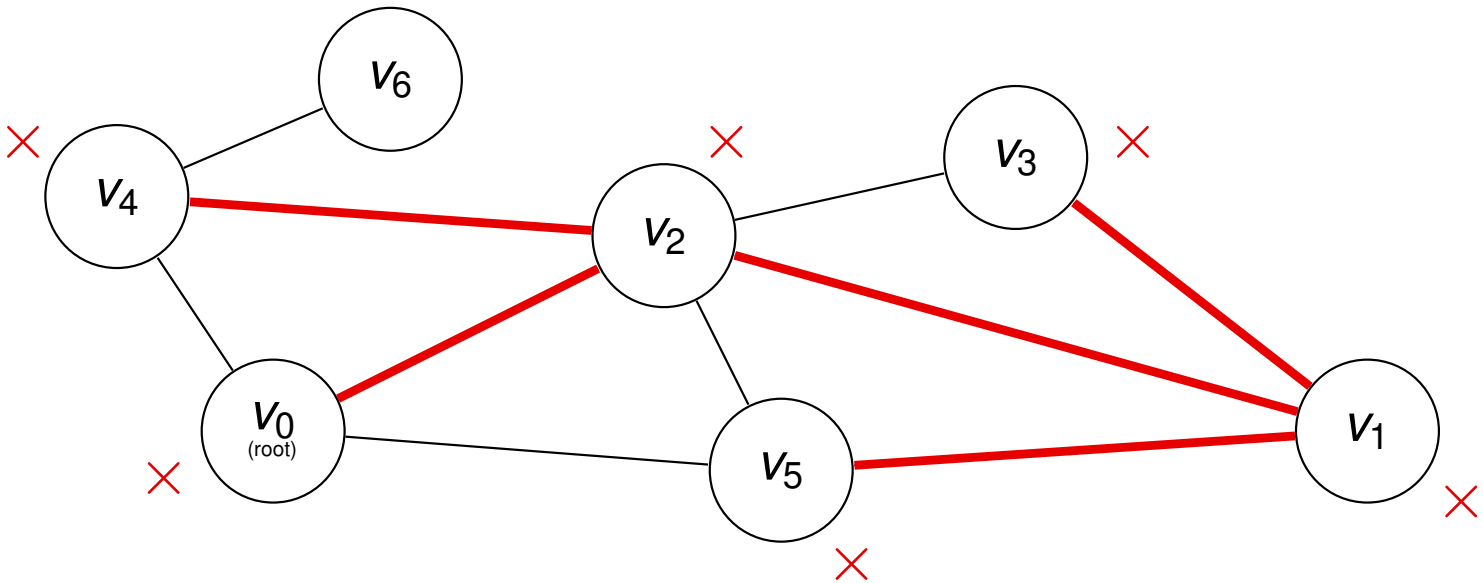
Output: v_0 v_2 v_1 v_3

DFS Tree



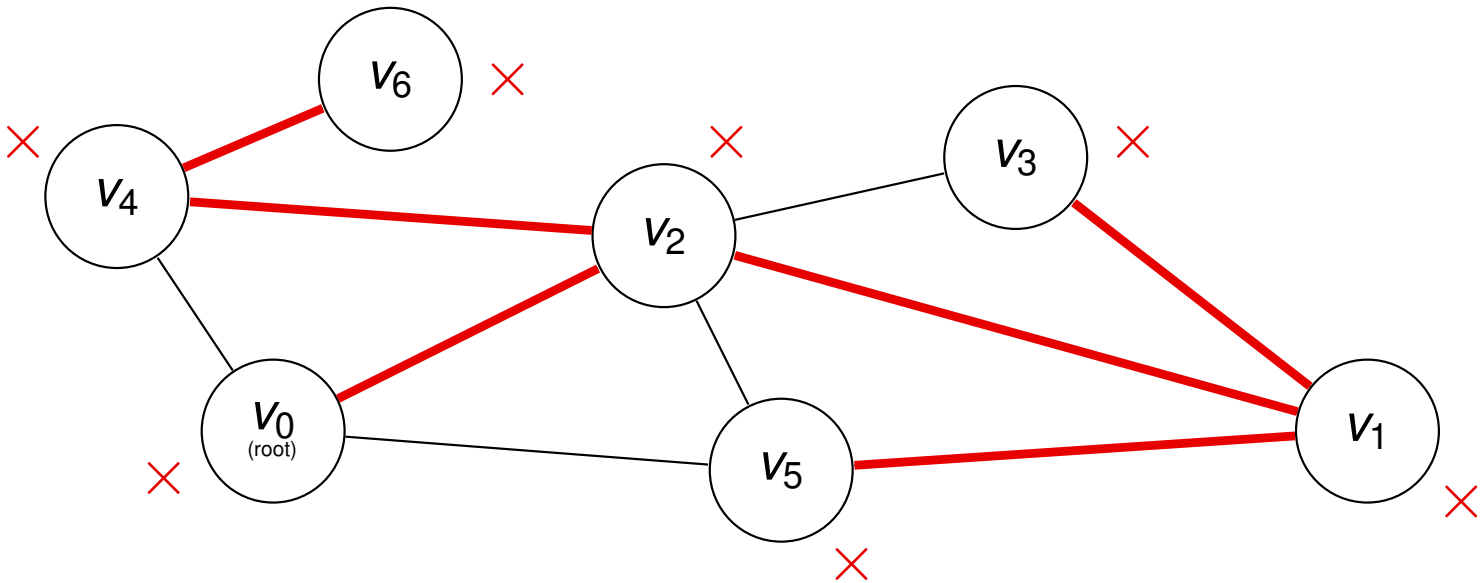
Output: v_0 v_2 v_1 v_3 v_5

DFS Tree



Output: v_0 v_2 v_1 v_3 v_5 v_4

DFS Tree

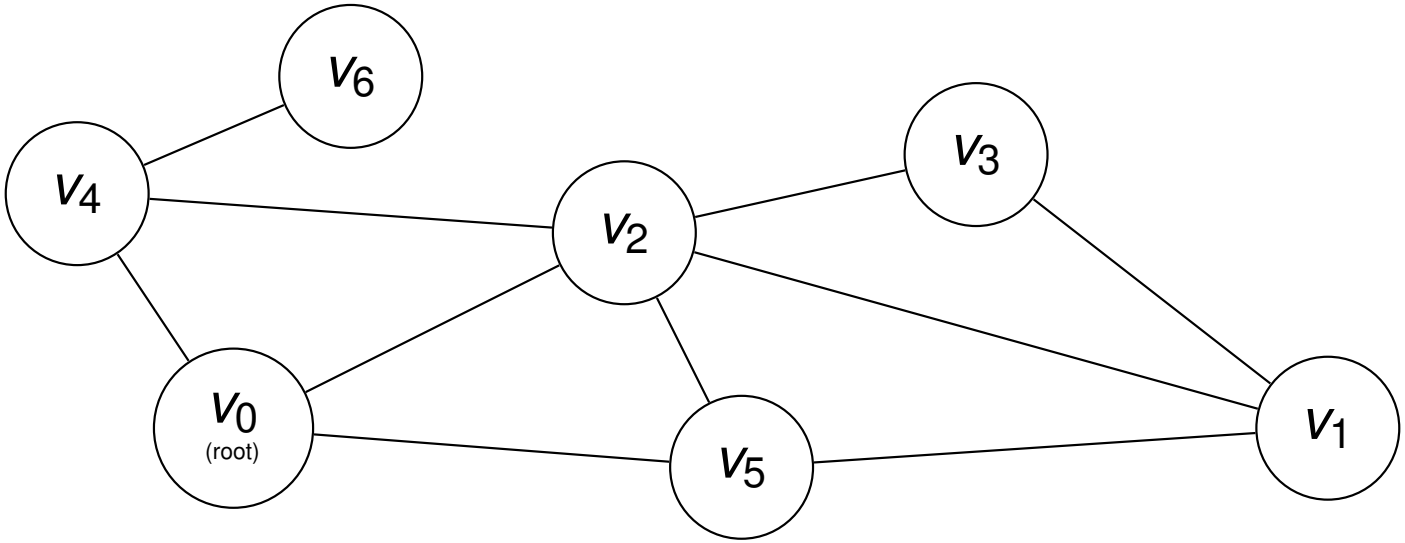


Output: v_0 v_2 v_1 v_3 v_5 v_4 v_6

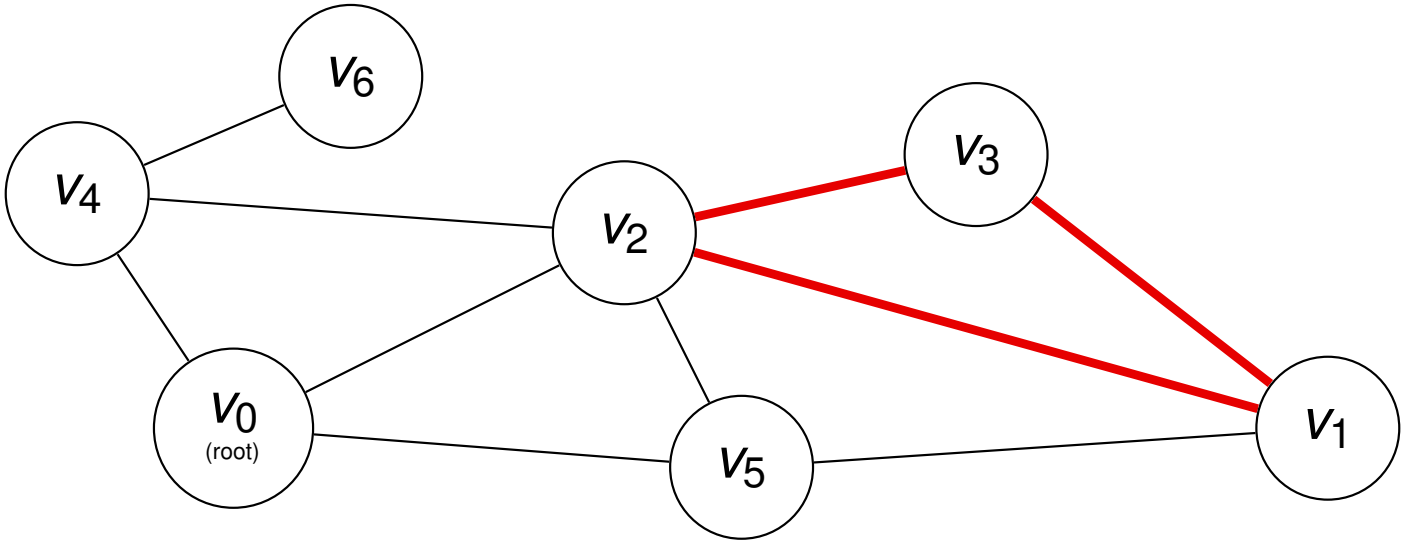
Finding Cycles

in undirected graphs

Finding Cycles

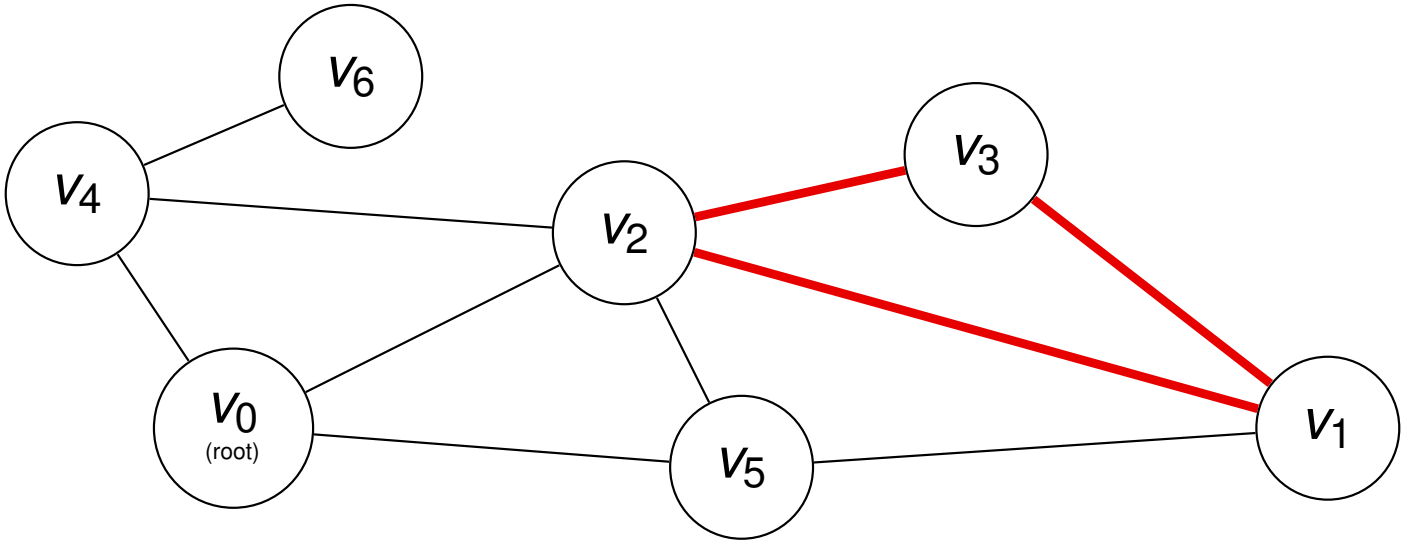


Finding Cycles



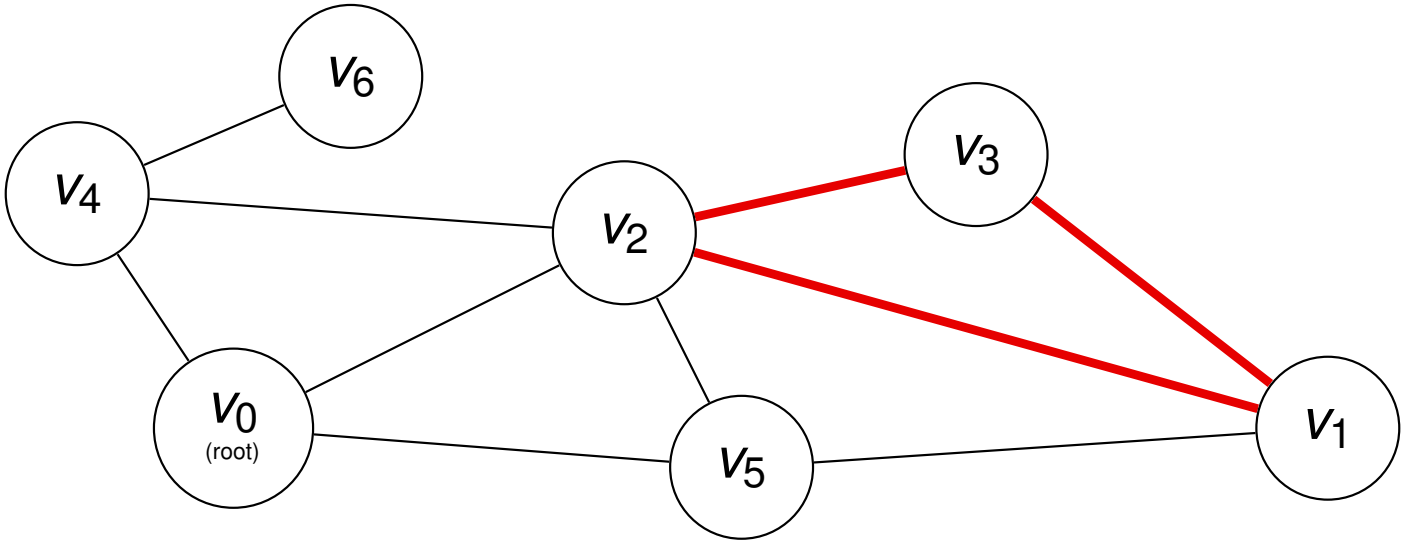
- DFS supplies all we need

Finding Cycles



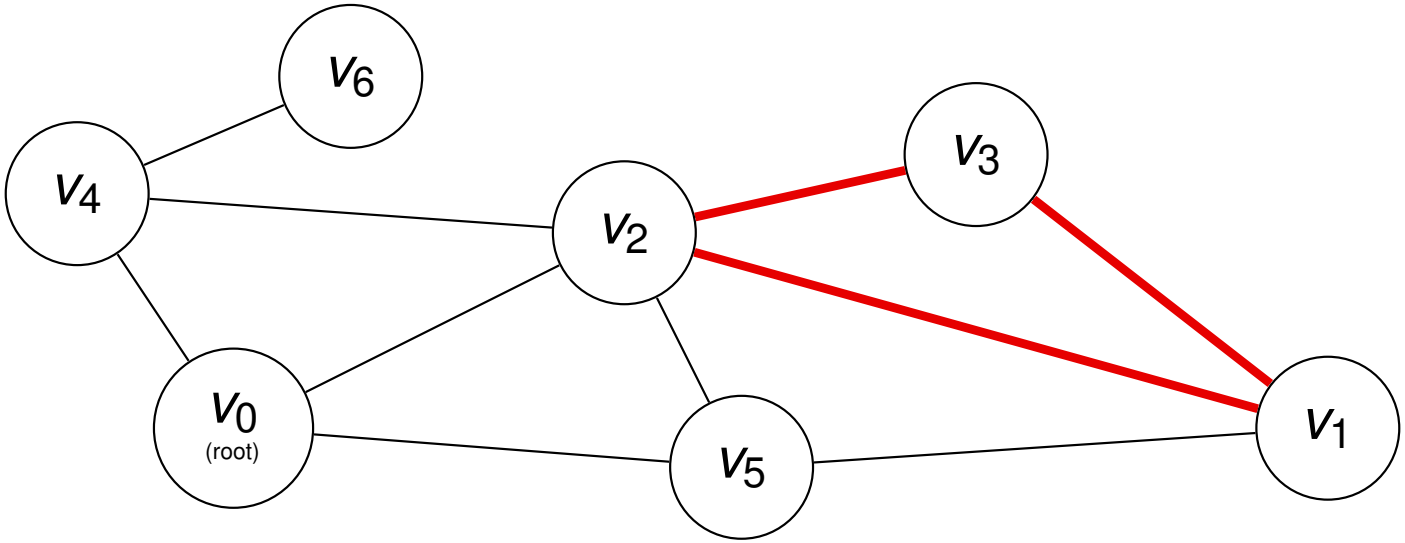
- DFS supplies all we need
- Traverse graph as before

Finding Cycles



- DFS supplies all we need
- Traverse graph as before
- Find edge connecting current vertex to one already visited?

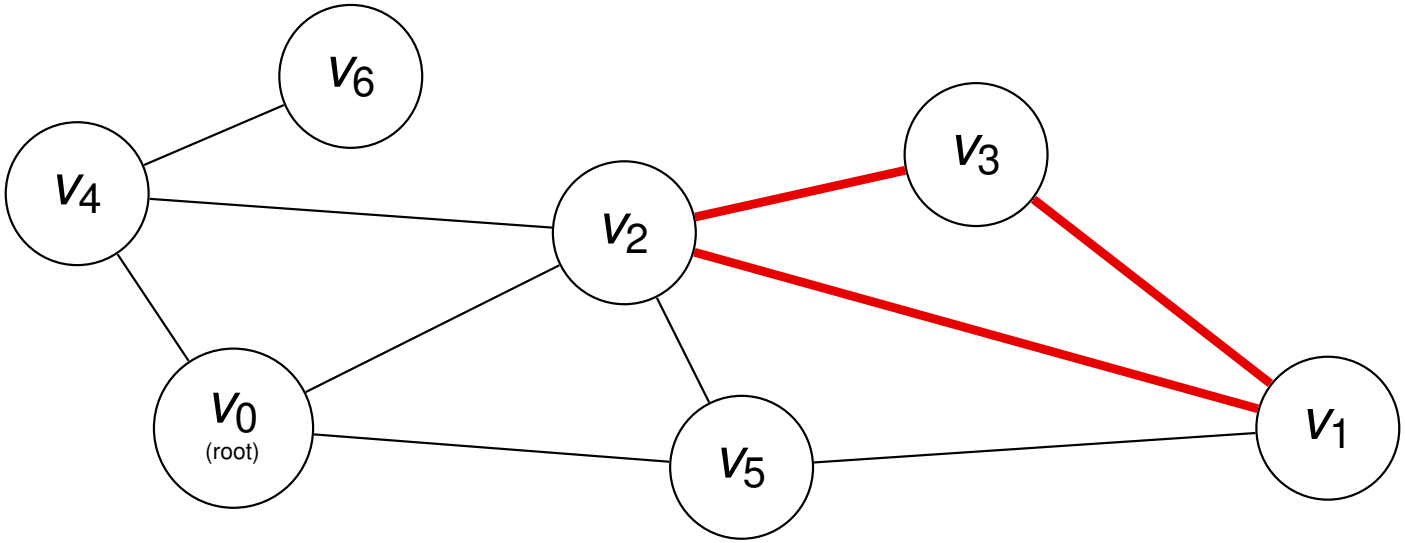
Finding Cycles



- DFS supplies all we need
- Traverse graph as before
- Find edge connecting current vertex to one already visited?

⇨ Back edge

Finding Cycles



- DFS supplies all we need
- Traverse graph as before
- Find edge connecting current vertex to one already visited?

⇨ Back edge

- Careful though: A single edge is not a cycle

Finding Cycles

```
vector<vector<int>>
G { {2,4,5},
    {2,3,5},
    {0,1,3,4,5},
    {1,2},
    {0,2,6},
    {0,1,2},
    {4} };
```

Finding Cycles

Extend DFS to keep track of parent vertex

Finding Cycles

Extend DFS to keep track of parent vertex

```
void cyclefind(int current, int parent) {
   if (!visited[current]) {
      cout << current << " ";
      visited[current] = true;
      for (int i=0; i<G[current].size(); ++i) {
         int v = G[current][i];
         if (v != parent) {
            if (visited[v]) {
               cout << "CYCLE";
            } else {
               cyclefind(v,current);
            }
         }
      }
   }
}
```

Other Applications

DFS in undirected graphs

Other Applications

Is graph connected?

Other Applications

Is graph connected?

- DFS from arbitrary vertex

Other Applications

Is graph connected?

- DFS from arbitrary vertex
- Are all vertices visited when done?

Other Applications

Is graph connected?

- DFS from arbitrary vertex
- Are all vertices visited when done?

Is vertex w reachable from vertex v?

Other Applications

Is graph connected?

- DFS from arbitrary vertex
- Are all vertices visited when done?

Is vertex w reachable from vertex v?

- DFS from v

Other Applications

Is graph connected?

- DFS from arbitrary vertex
- Are all vertices visited when done?

Is vertex w reachable from vertex v?

- DFS from v
- Is w visited when done?

Other Applications

Is graph connected?

- DFS from arbitrary vertex
- Are all vertices visited when done?

Is vertex w reachable from vertex v?

- DFS from v
- Is w visited when done?

Is connected graph 2-colorable?

Other Applications

Is graph connected?

- DFS from arbitrary vertex
- Are all vertices visited when done?

Is vertex w reachable from vertex v?

- DFS from v
- Is w visited when done?

Is connected graph 2-colorable?

- DFS from arbitrary vertex v

Other Applications

Is graph connected?

- DFS from arbitrary vertex
- Are all vertices visited when done?

Is vertex w reachable from vertex v?

- DFS from v
- Is w visited when done?

Is connected graph 2-colorable?

- DFS from arbitrary vertex v
- Neighbors of v get color different from that of v, and so on

Other Applications

Is graph connected?

- DFS from arbitrary vertex
- Are all vertices visited when done?

Is vertex w reachable from vertex v?

- DFS from v
- Is w visited when done?

Is connected graph 2-colorable?

- DFS from arbitrary vertex v
- Neighbors of v get color different from that of v, and so on
- Check for collisions via back edges

Finding Cycles

in directed graphs

Finding Cycles

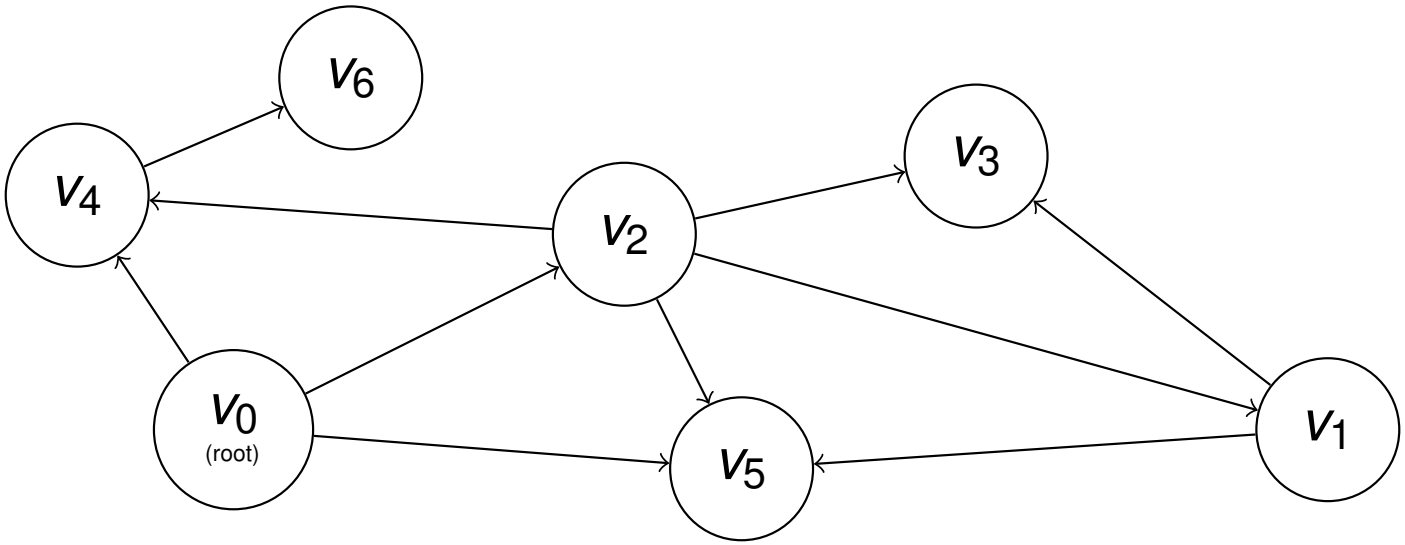
- Same DFS strategy as before also works for directed graphs

Finding Cycles

- Same DFS strategy as before also works for directed graphs
- But what about finding cycles?

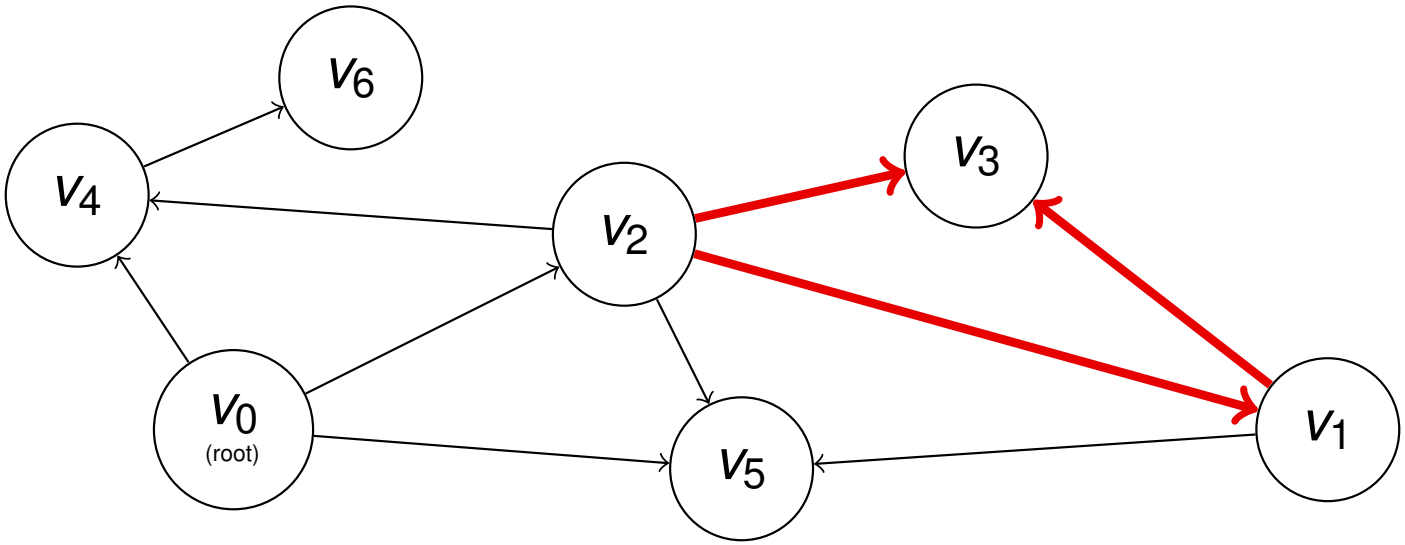
Finding Cycles

- Same DFS strategy as before also works for directed graphs
- But what about finding cycles?



Finding Cycles

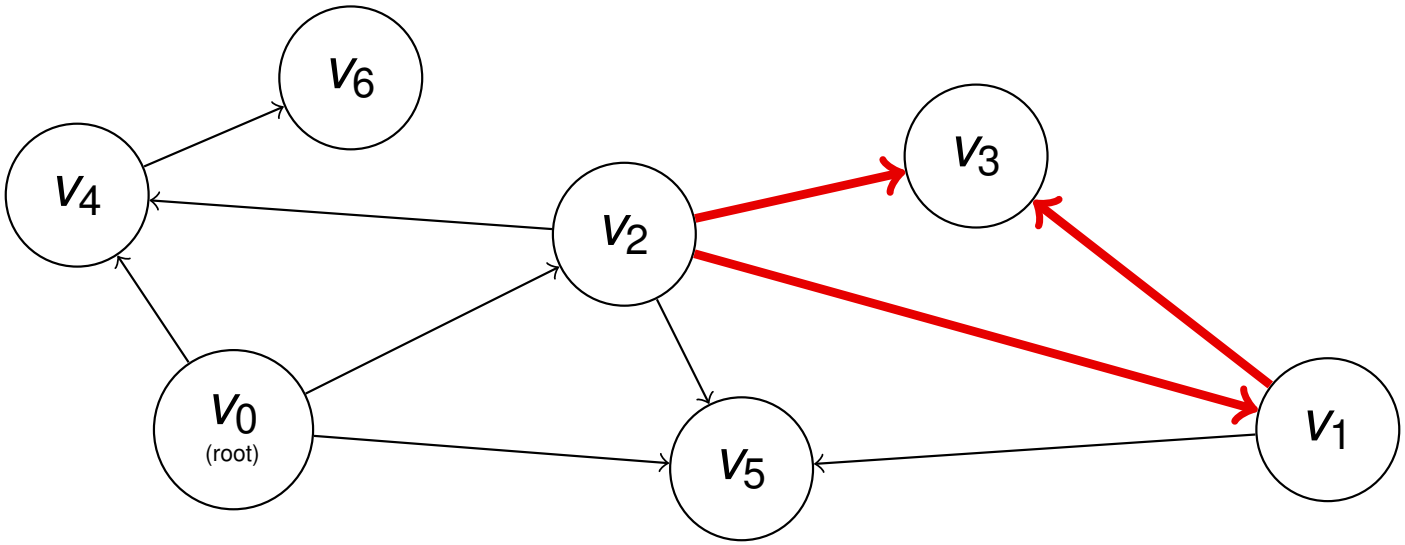
- Same DFS strategy as before also works for directed graphs
- But what about finding cycles?



- This is not a cycle

Finding Cycles

- Same DFS strategy as before also works for directed graphs
- But what about finding cycles?



- This is not a cycle
- But our algorithm would identify it as one

Finding Cycles

Instead of just marking whether a vertex is visited or not, keep track of whether it is on our current path

Finding Cycles

Instead of just marking whether a vertex is visited or not, keep track of whether it is on our current path

With this, a vertex has three states

Finding Cycles

Instead of just marking whether a vertex is visited or not, keep track of whether it is on our current path

With this, a vertex has three states

- unvisited

Finding Cycles

Instead of just marking whether a vertex is visited or not, keep track of whether it is on our current path

With this, a vertex has three states

- unvisited
- active

Finding Cycles

Instead of just marking whether a vertex is visited or not, keep track of whether it is on our current path

With this, a vertex has three states

- unvisited
- active
- visited

Finding Cycles

Instead of just marking whether a vertex is visited or not, keep track of whether it is on our current path

With this, a vertex has three states

- unvisited (marked white)
- active (marked gray)
- visited (marked black)

Finding Cycles

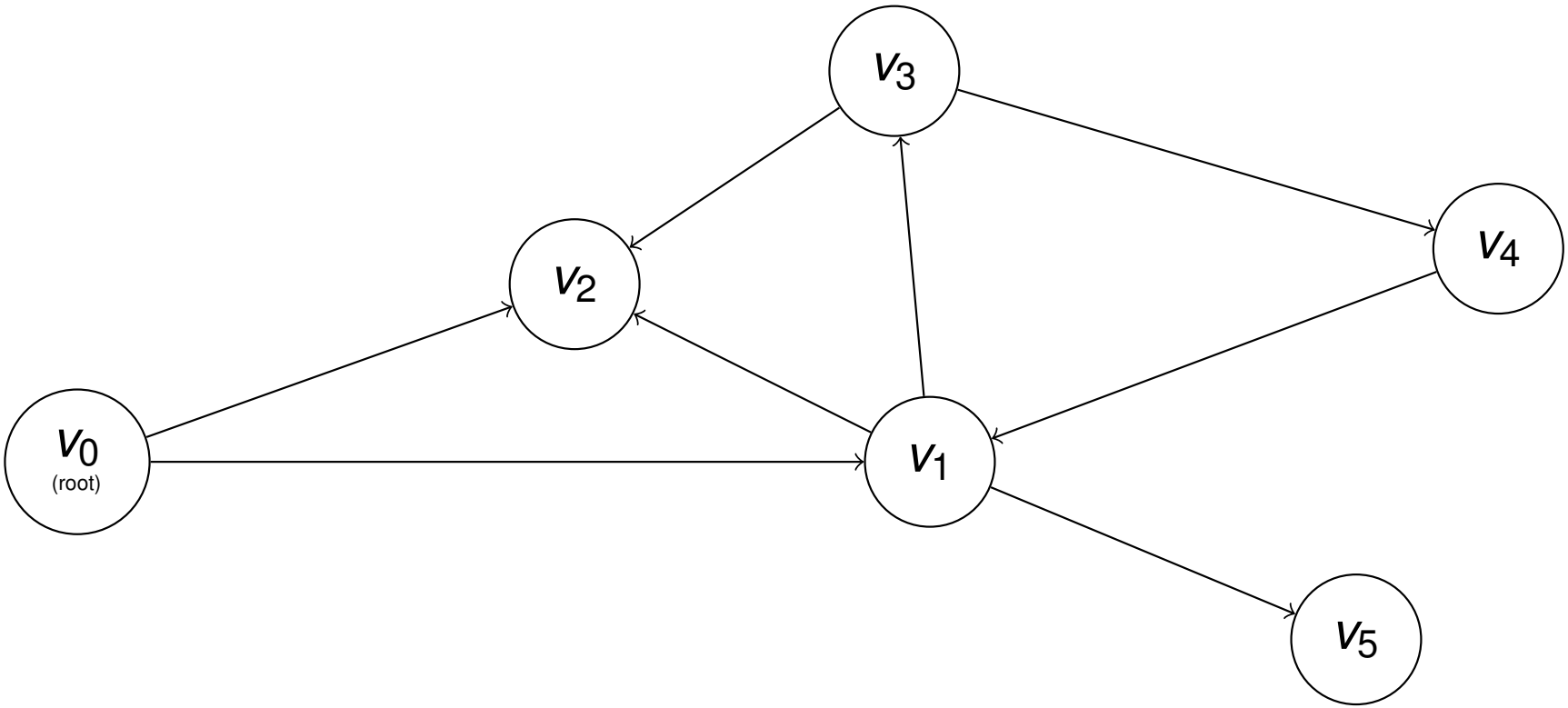
Instead of just marking whether a vertex is visited or not, keep track of whether it is on our current path

With this, a vertex has three states

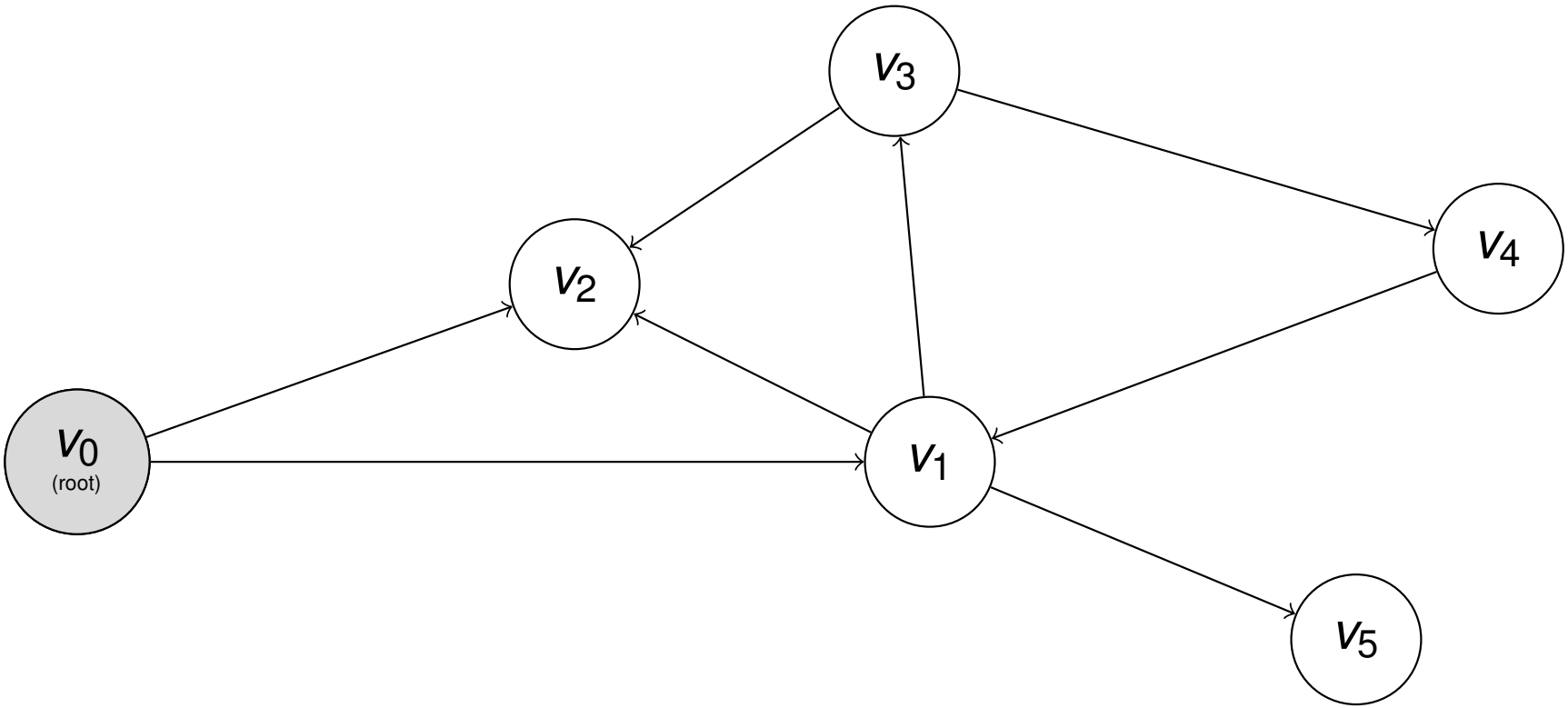
- unvisited (marked white)
- active (marked gray)
- visited (marked black)

This allows us to detect cycles in directed graphs

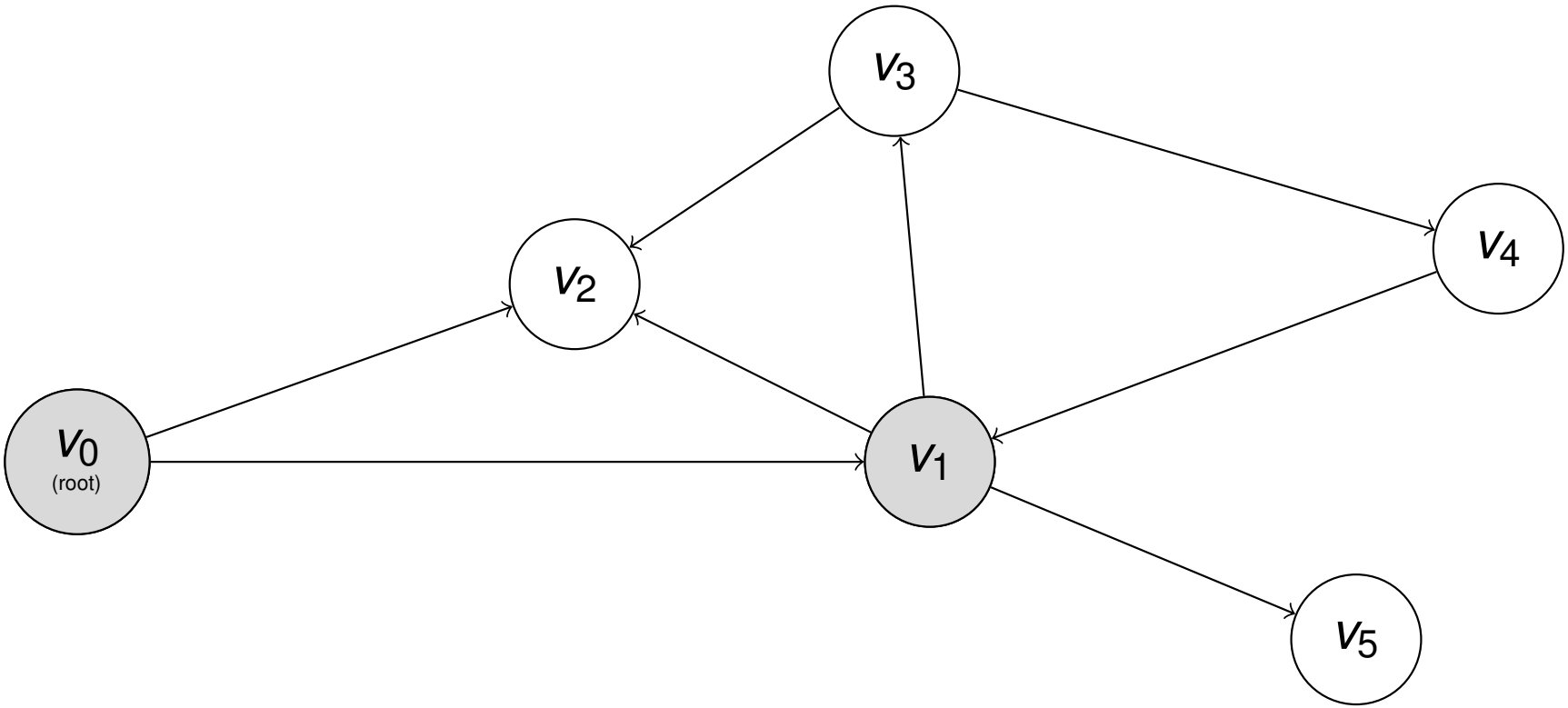
Finding Cycles



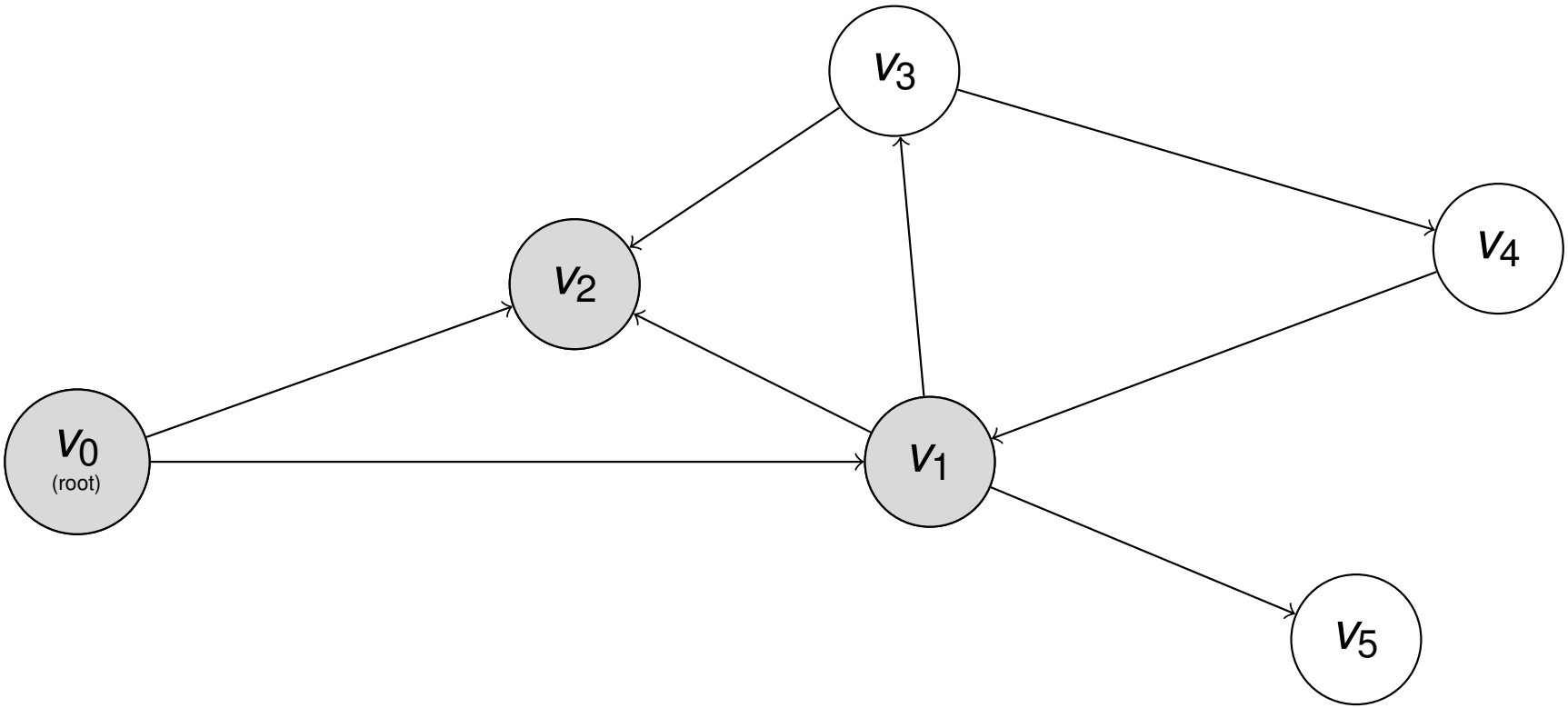
Finding Cycles



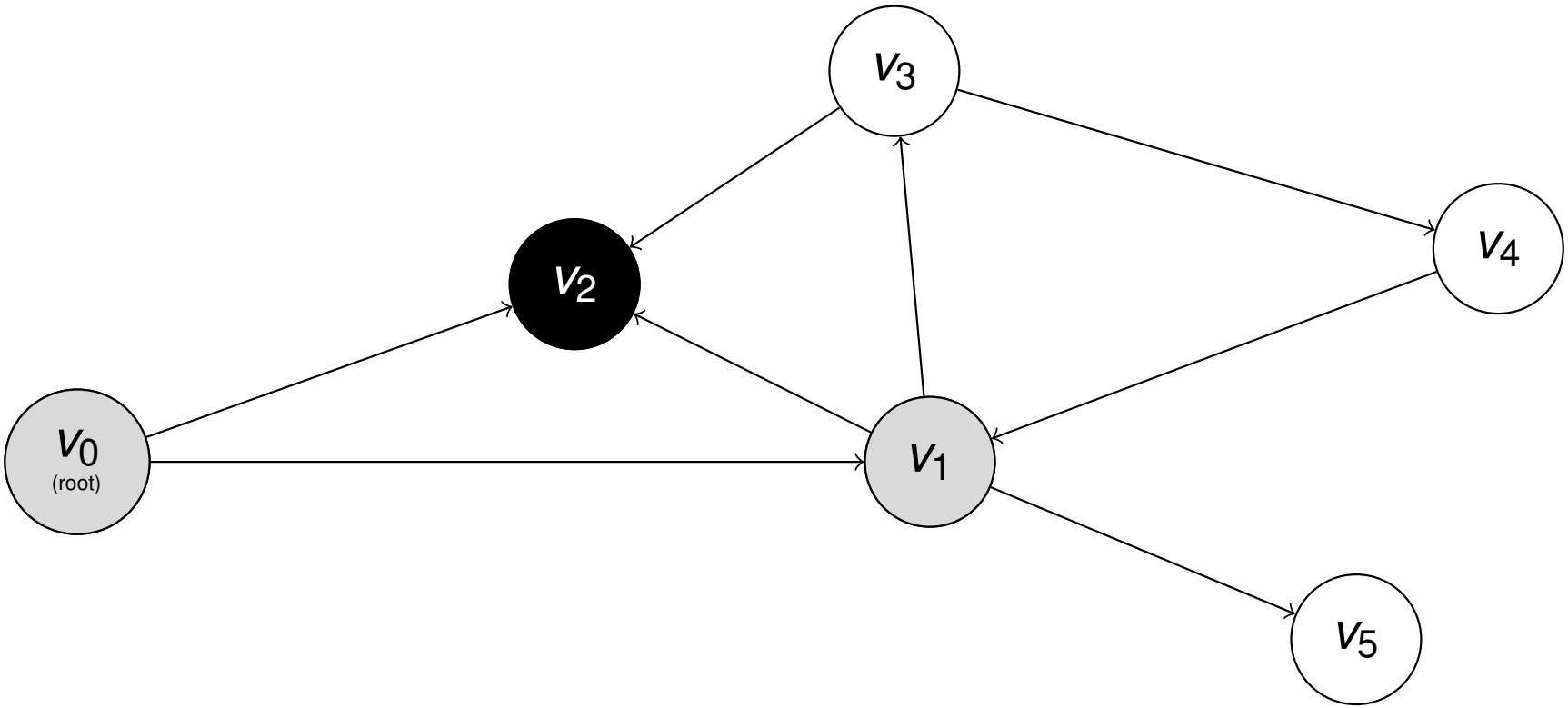
Finding Cycles



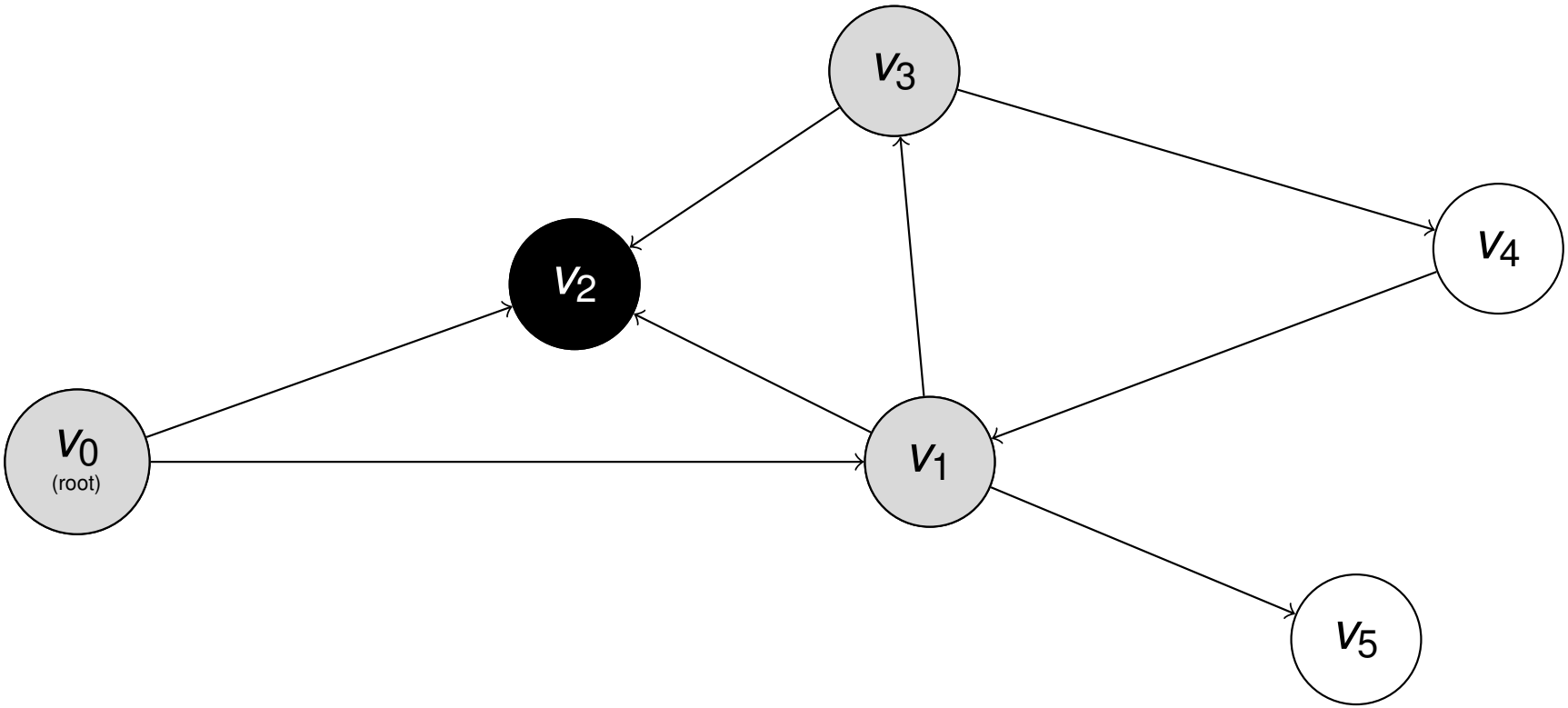
Finding Cycles



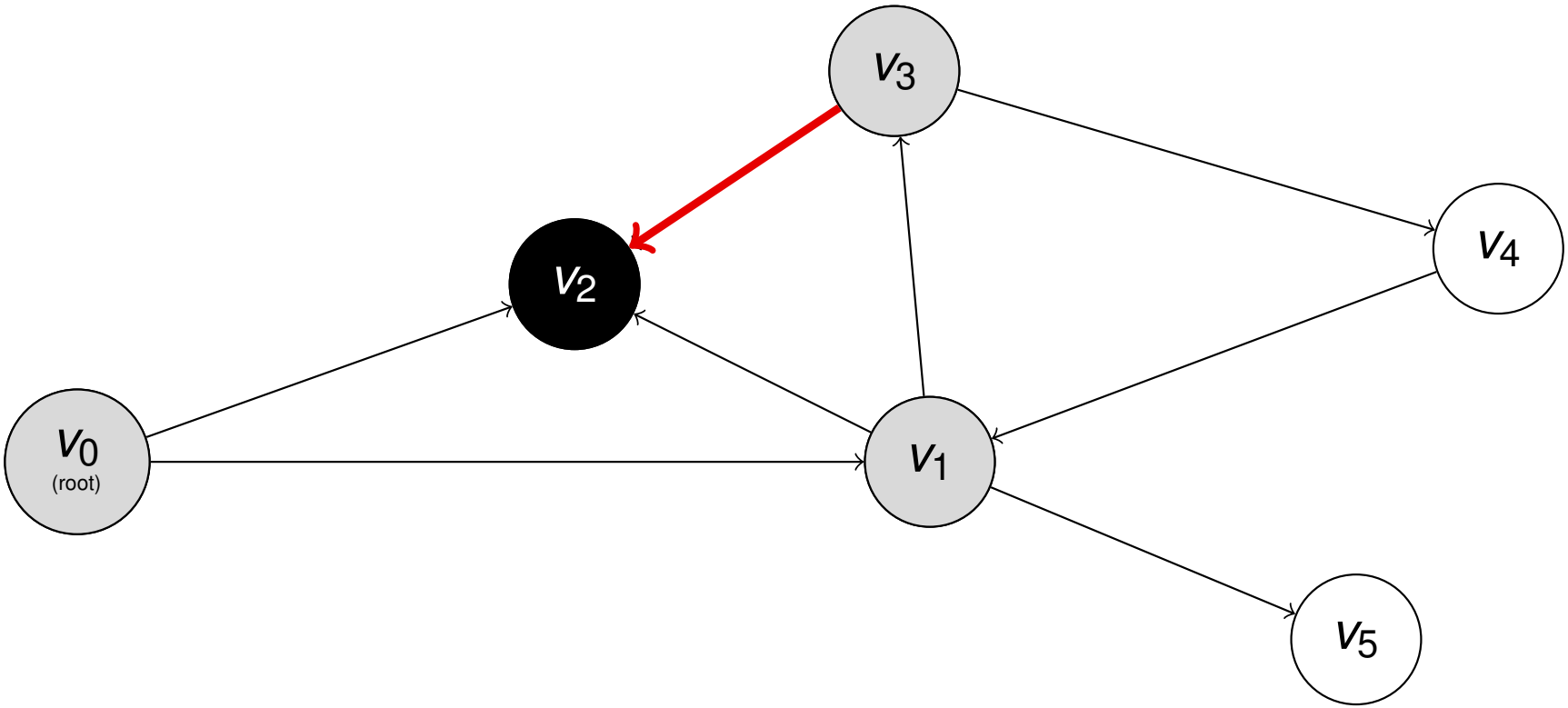
Finding Cycles



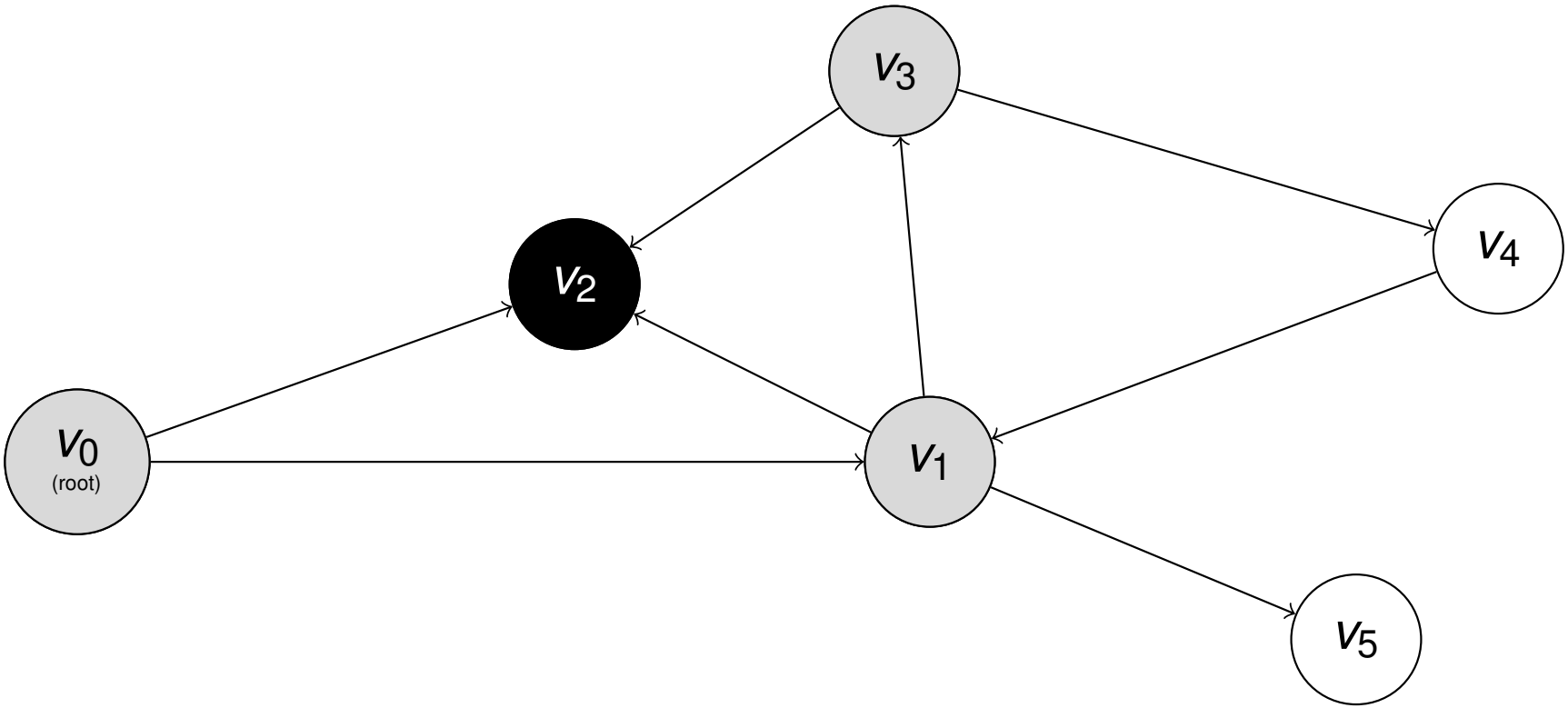
Finding Cycles



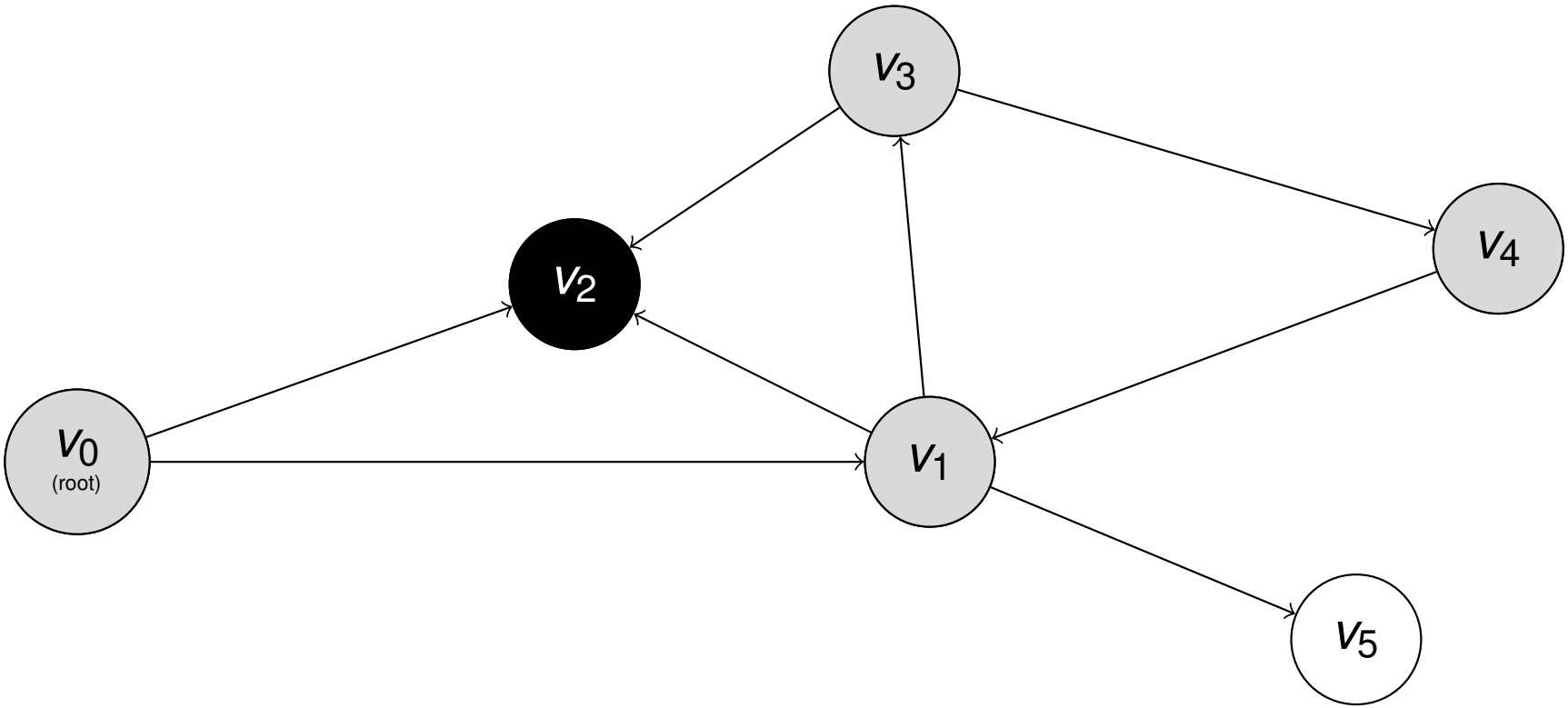
Finding Cycles



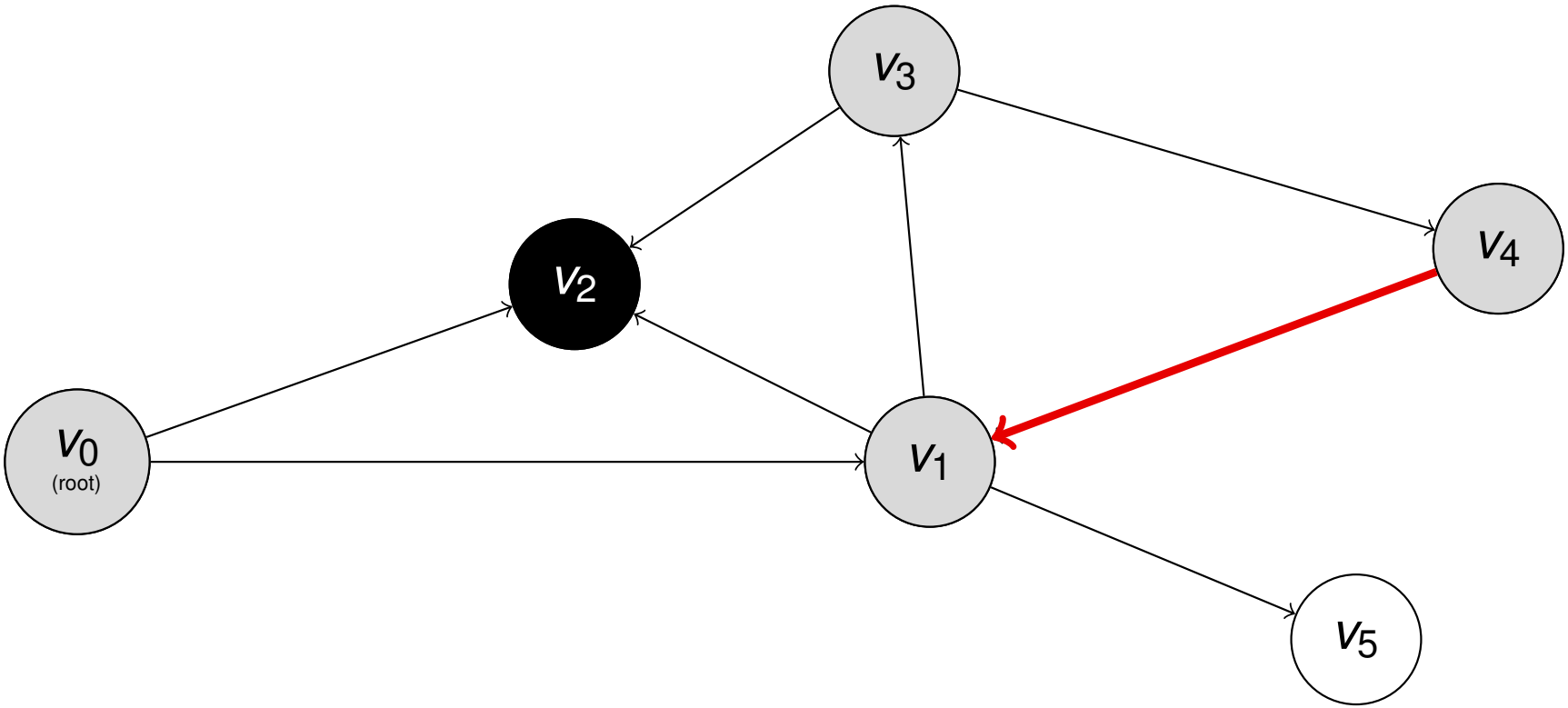
Finding Cycles



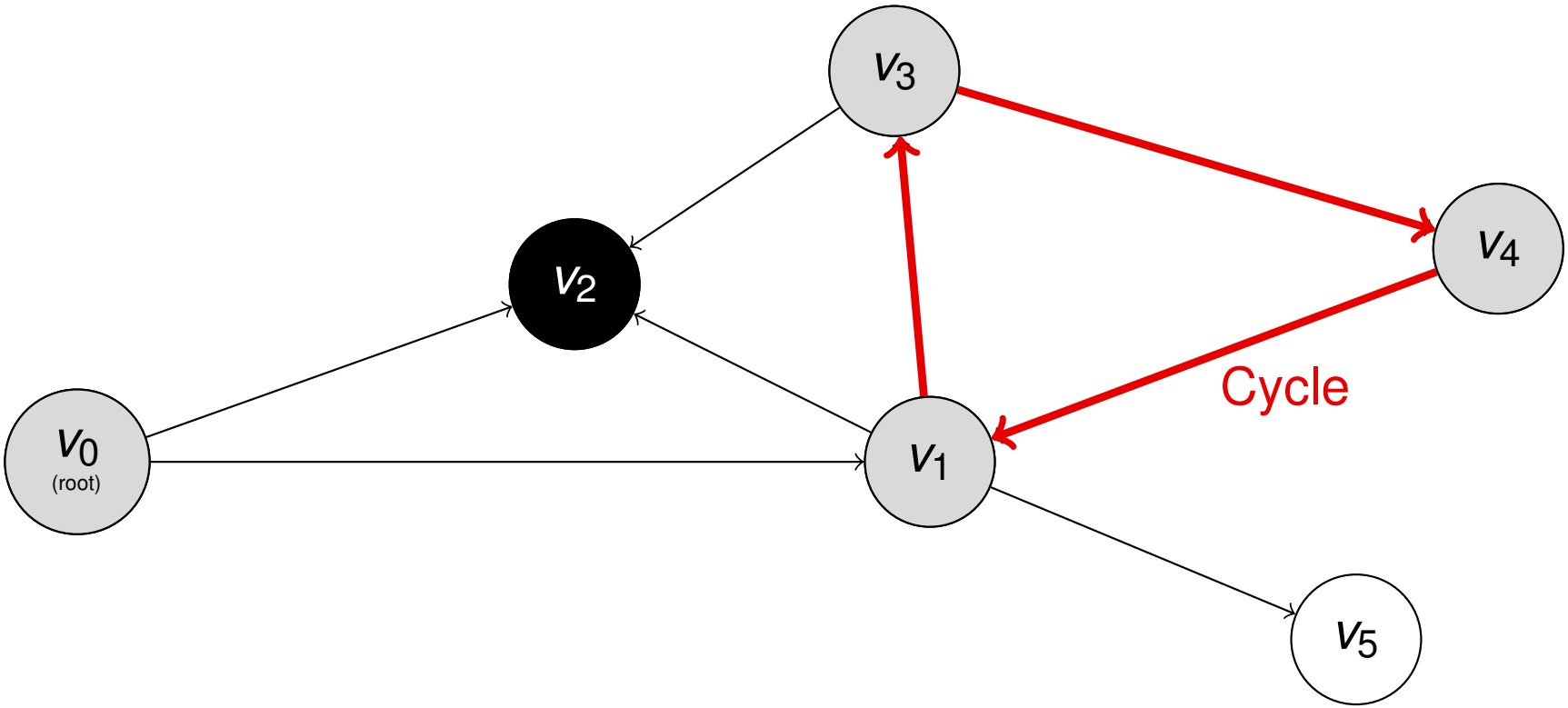
Finding Cycles



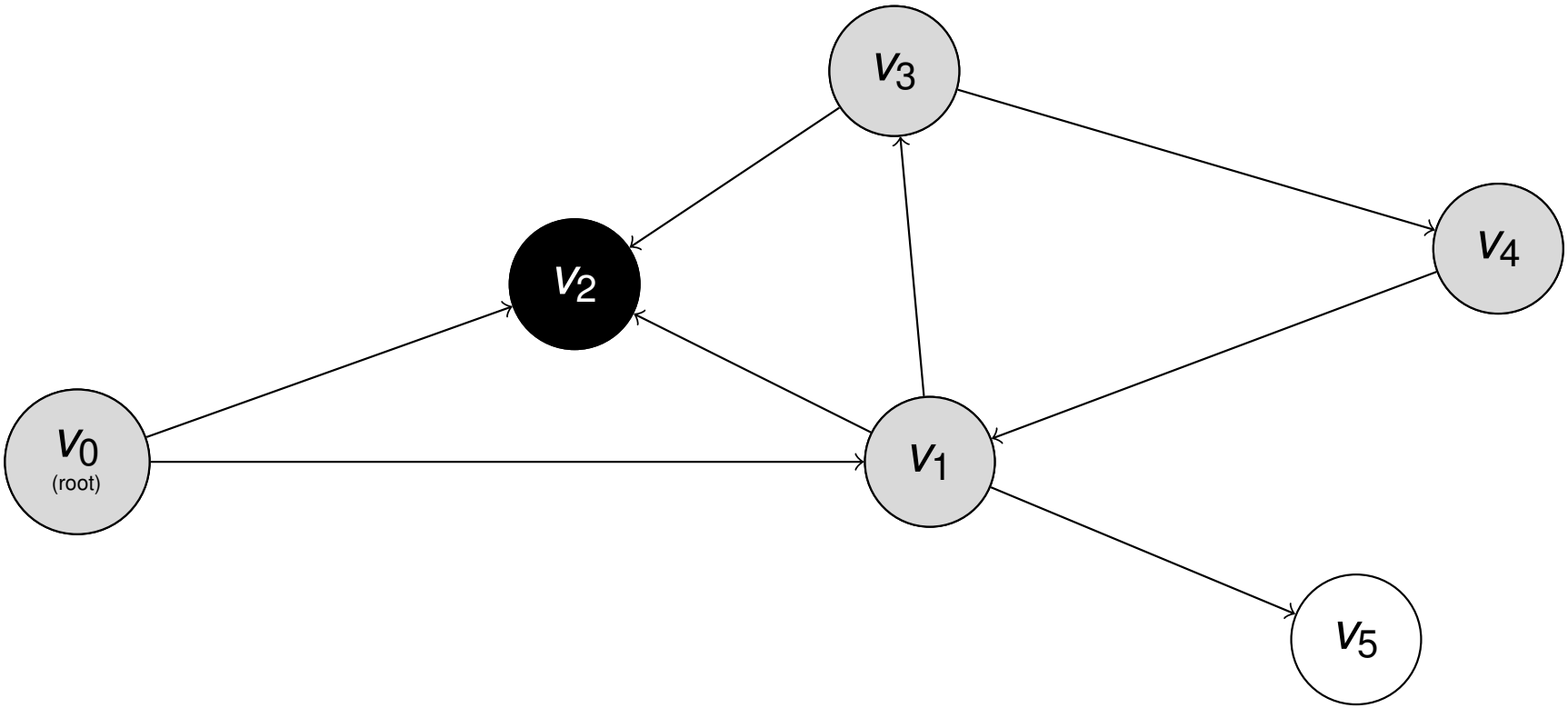
Finding Cycles



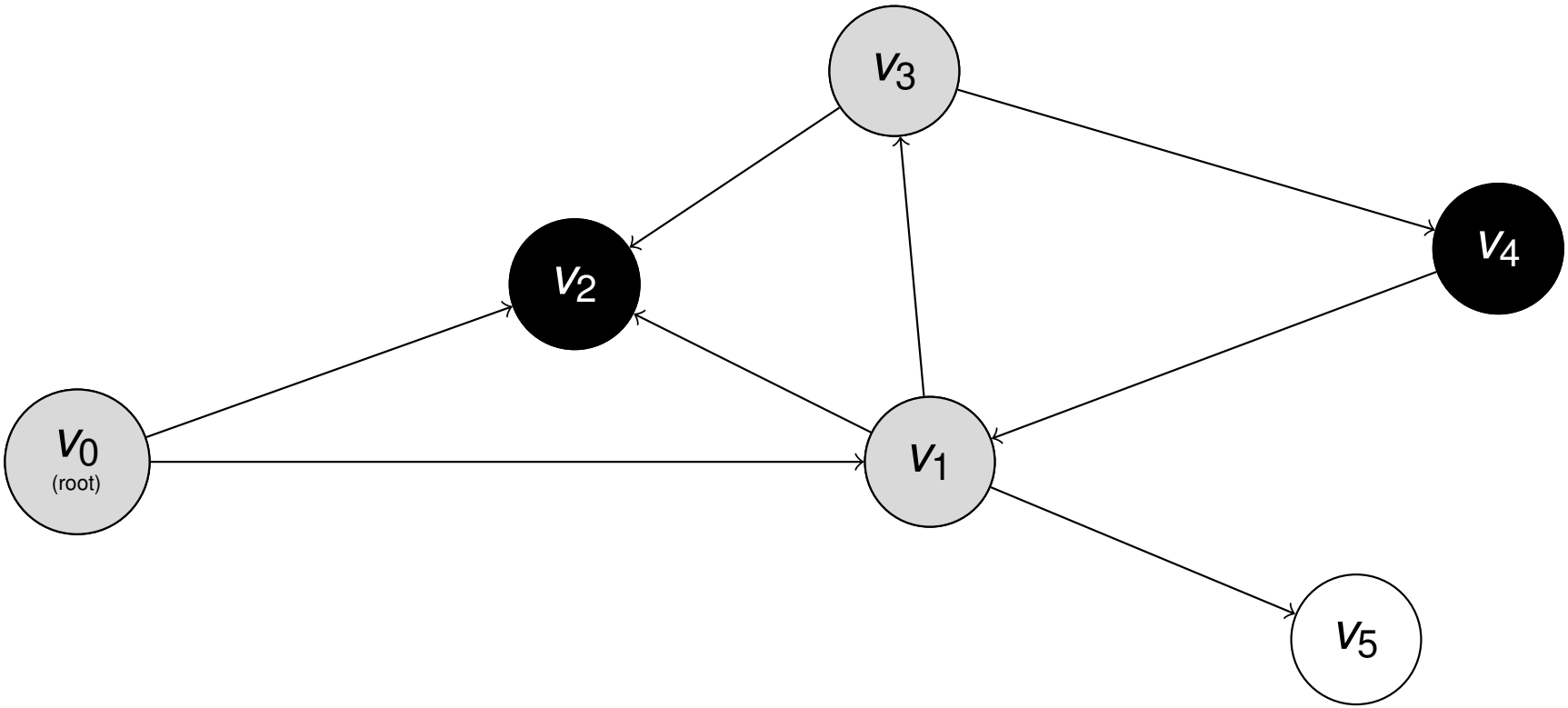
Finding Cycles



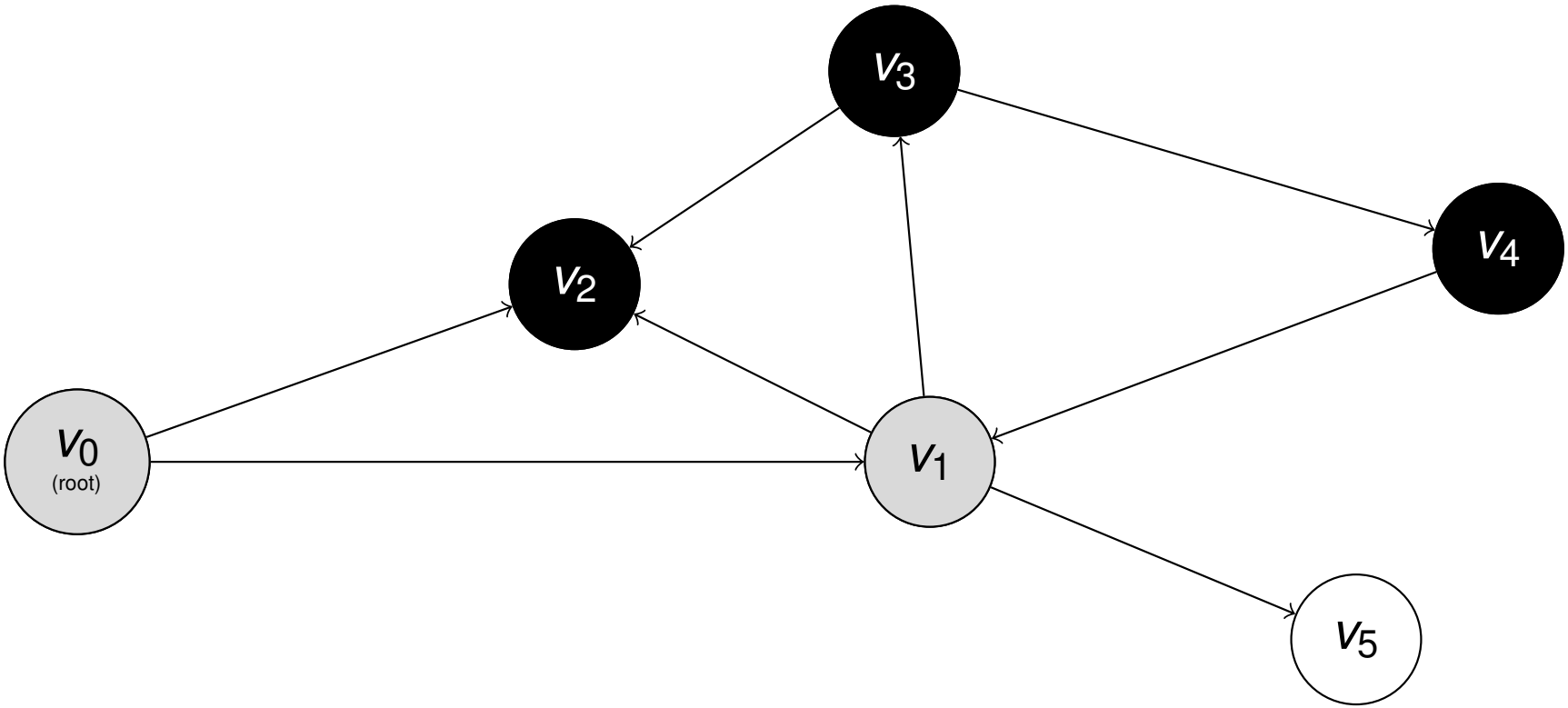
Finding Cycles



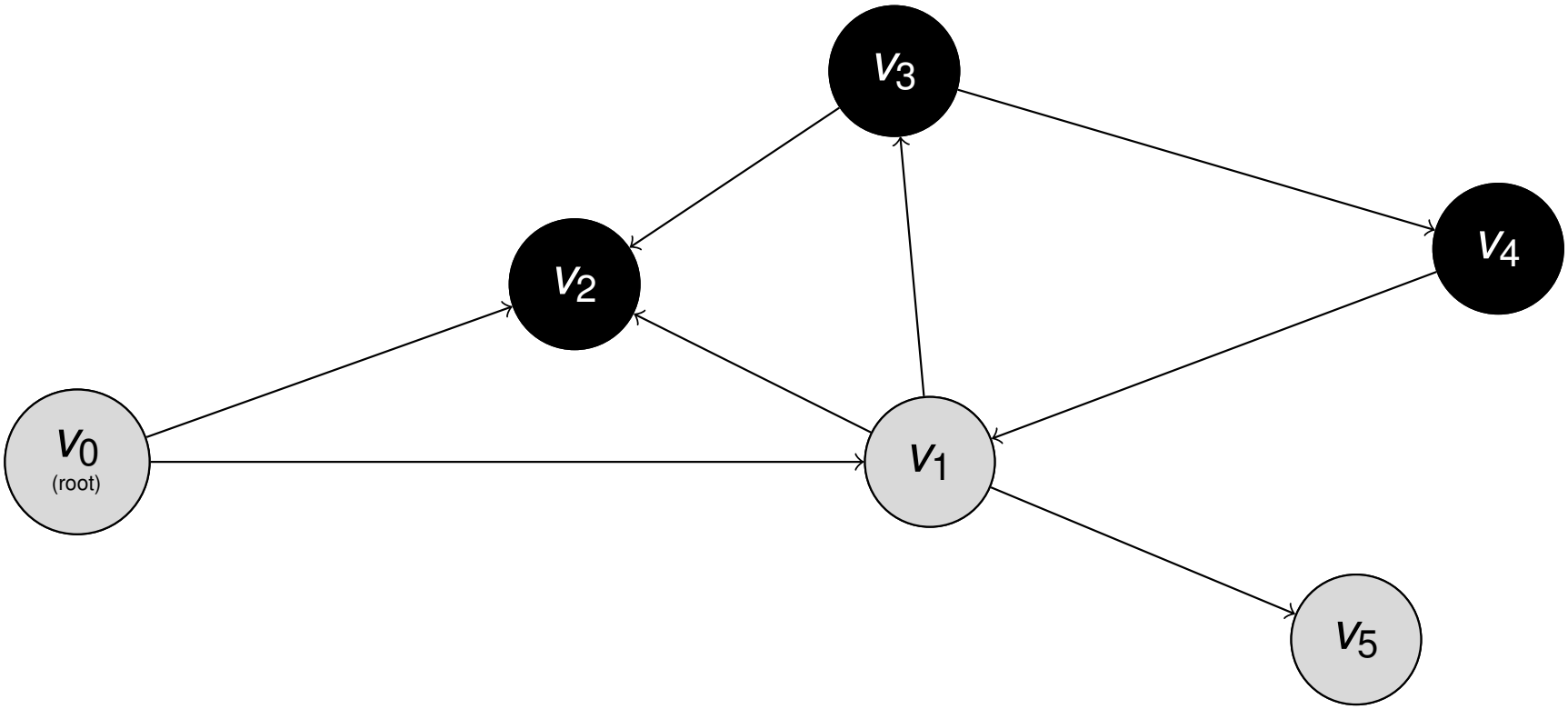
Finding Cycles



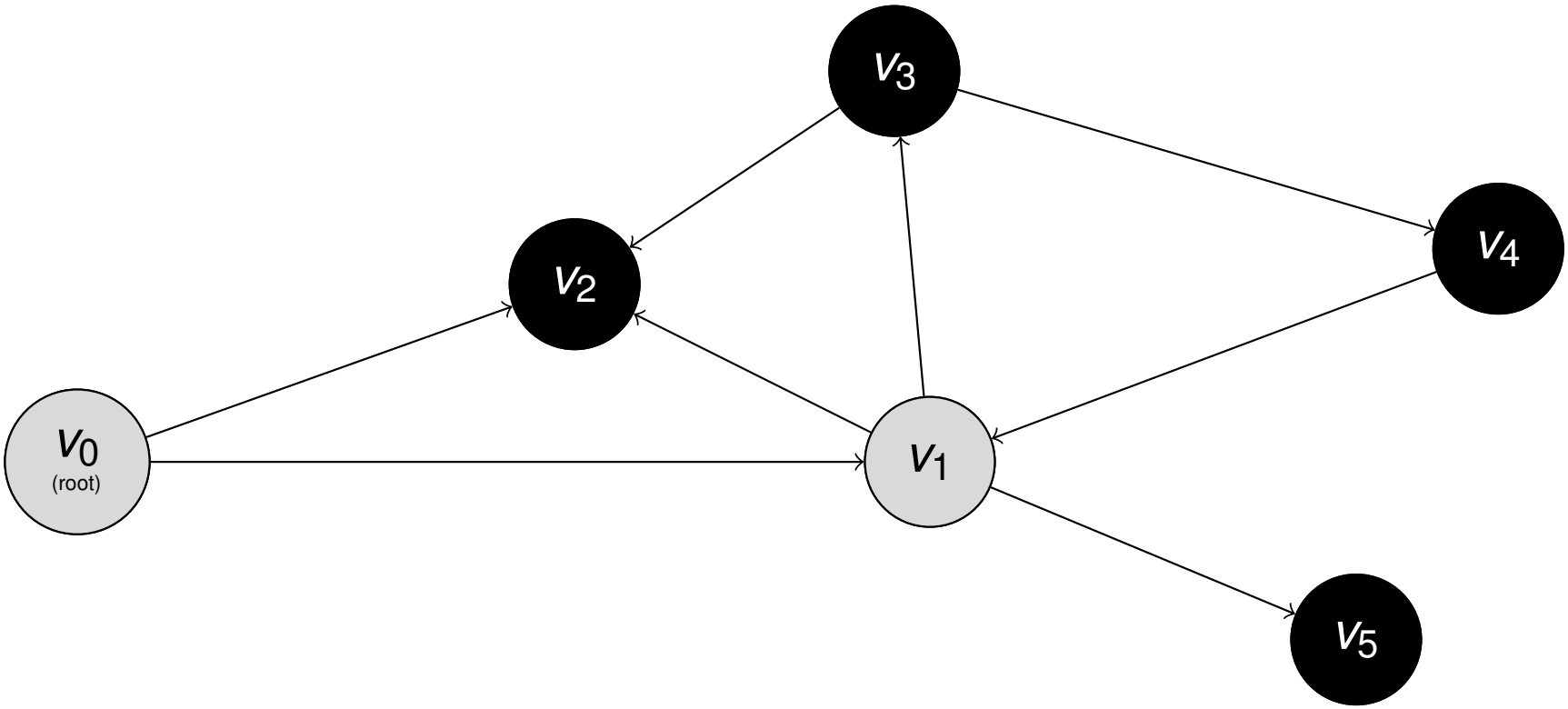
Finding Cycles



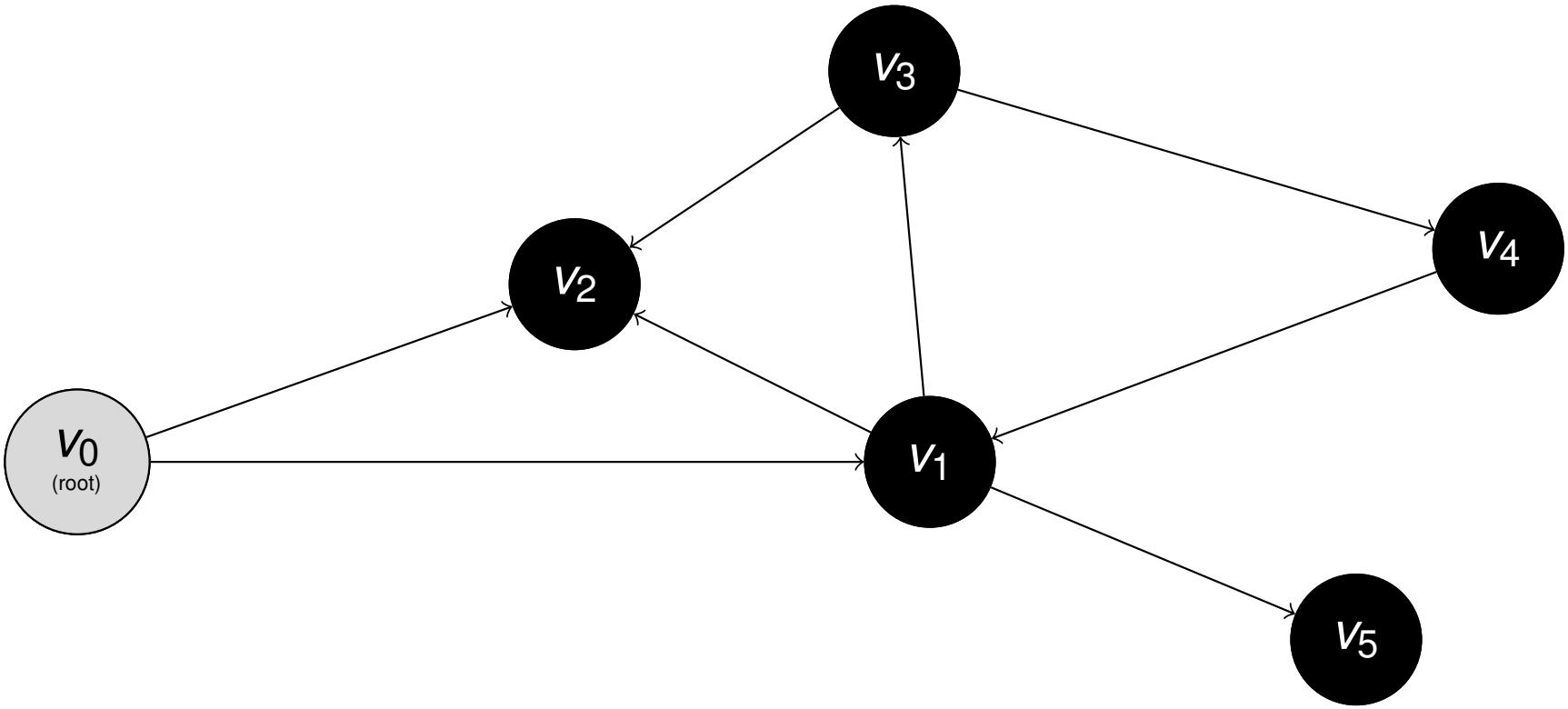
Finding Cycles



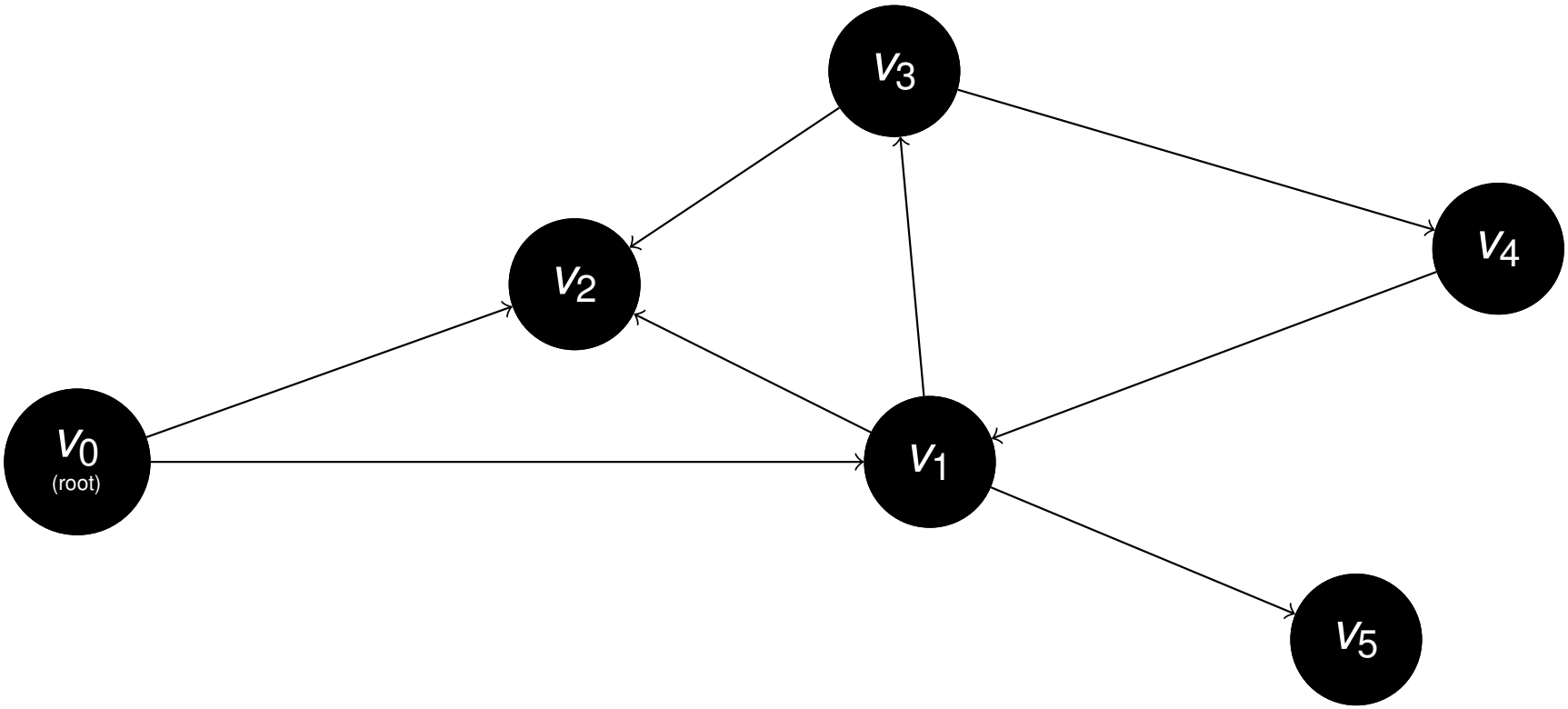
Finding Cycles



Finding Cycles



Finding Cycles



Finding Cycles

Edge from v to w can be classified at time it is considered

Finding Cycles

Edge from v to w can be classified at time it is considered

- **Tree edges.** edges that belong to DFS tree (w has color white)

Finding Cycles

Edge from v to w can be classified at time it is considered

- **Tree edges.** edges that belong to DFS tree (w has color white)
- **Forward edges.** v is ancestor of w (w has color black)

Finding Cycles

Edge from v to w can be classified at time it is considered

- **Tree edges.** edges that belong to DFS tree (w has color white)
- **Forward edges.** v is ancestor of w (w has color black)
- **Back edges.** w is ancestor of v (w has color gray)

Finding Cycles

Edge from v to w can be classified at time it is considered

- **Tree edges.** edges that belong to DFS tree (w has color white)
- **Forward edges.** v is ancestor of w (w has color black)
- **Back edges.** w is ancestor of v (w has color gray)
- **Cross edges.** Neither ancestor of other (w has color black)

Finding Cycles

Edge from v to w can be classified at time it is considered

- **Tree edges.** edges that belong to DFS tree (w has color white)
- **Forward edges.** v is ancestor of w (w has color black)
- **Back edges.** w is ancestor of v (w has color gray)
- **Cross edges.** Neither ancestor of other (w has color black)

Graph has cycle if and only if there is a back edge

Finding Cycles

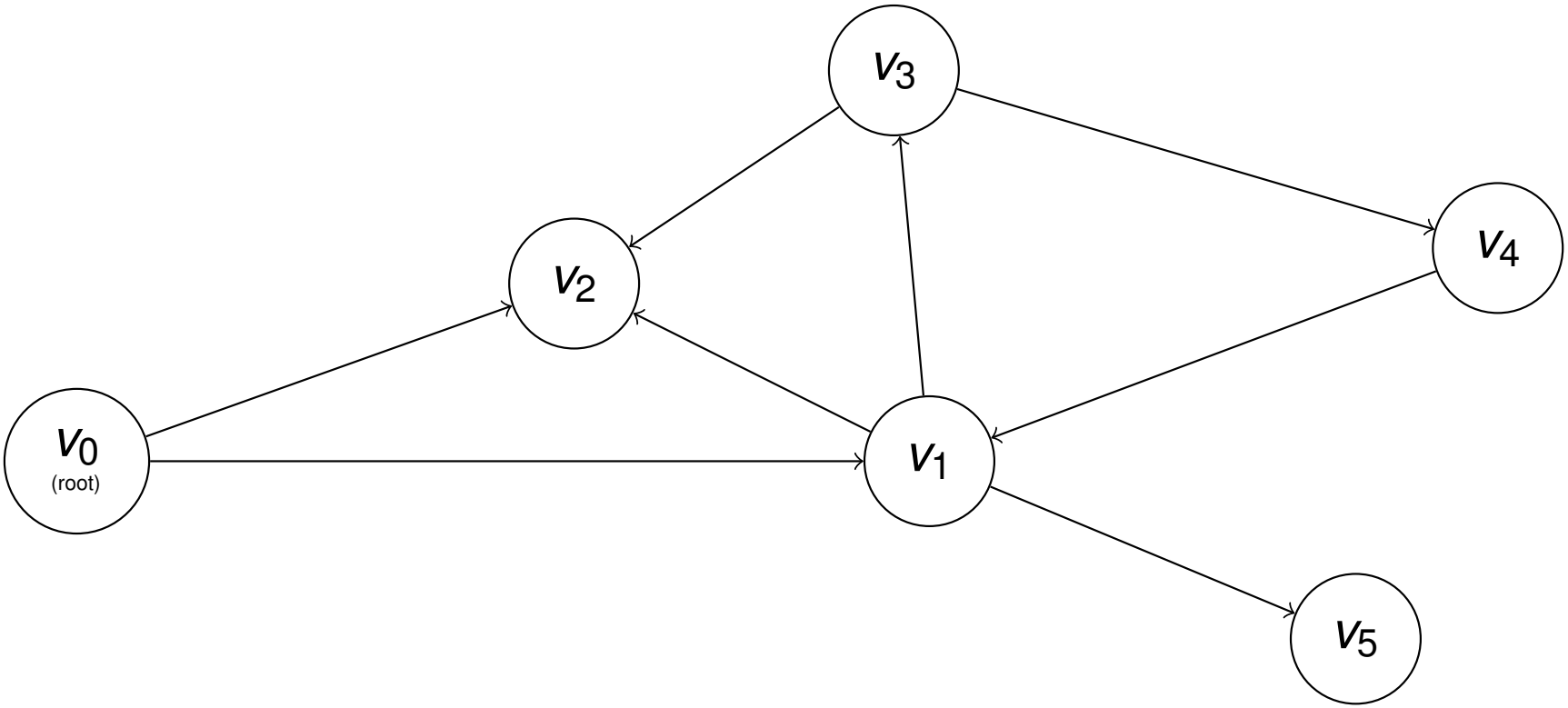
Edge from v to w can be classified at time it is considered

- **Tree edges.** edges that belong to DFS tree (w has color white)
- **Forward edges.** v is ancestor of w (w has color black)
- **Back edges.** w is ancestor of v (w has color gray)
- **Cross edges.** Neither ancestor of other (w has color black)

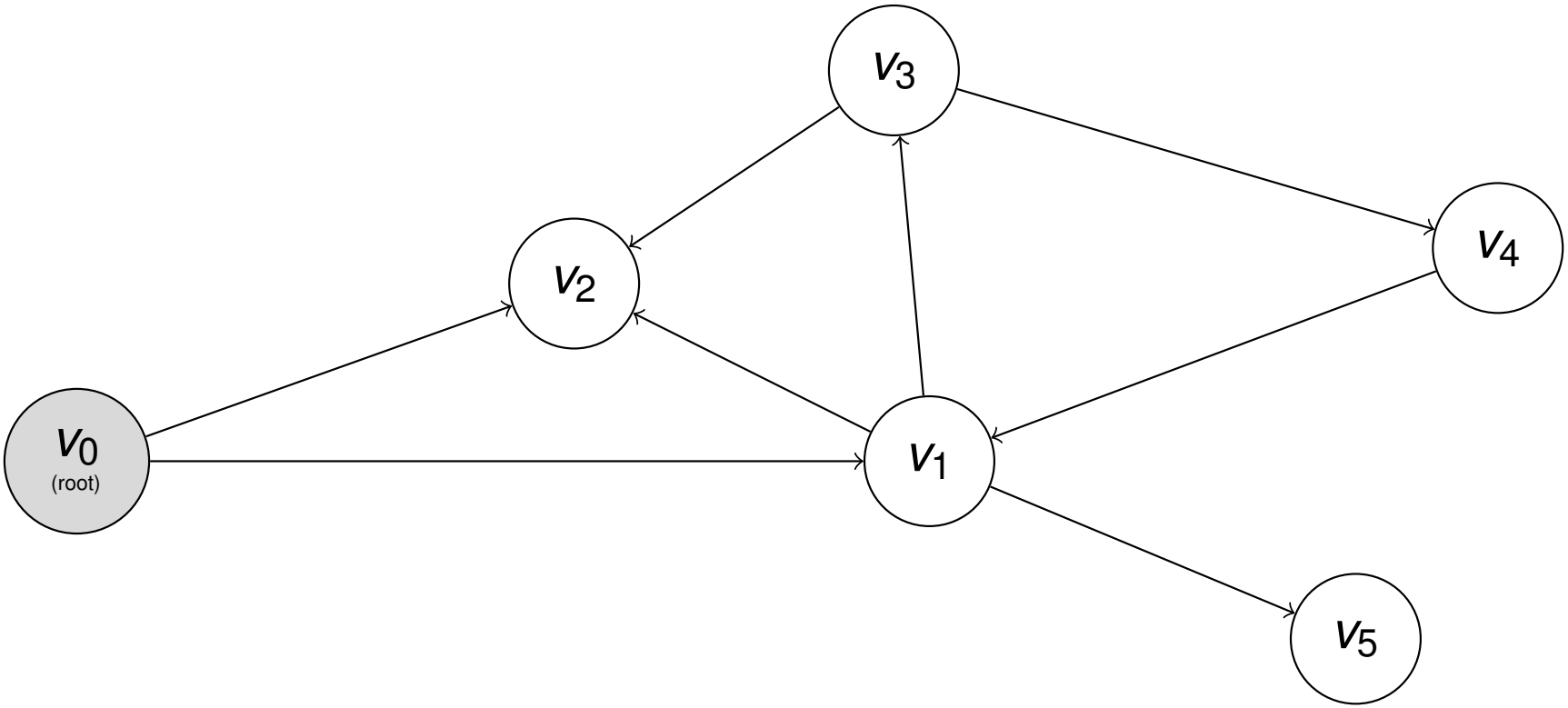
Graph has cycle if and only if there is a back edge

Undirected graph has only tree and back edges

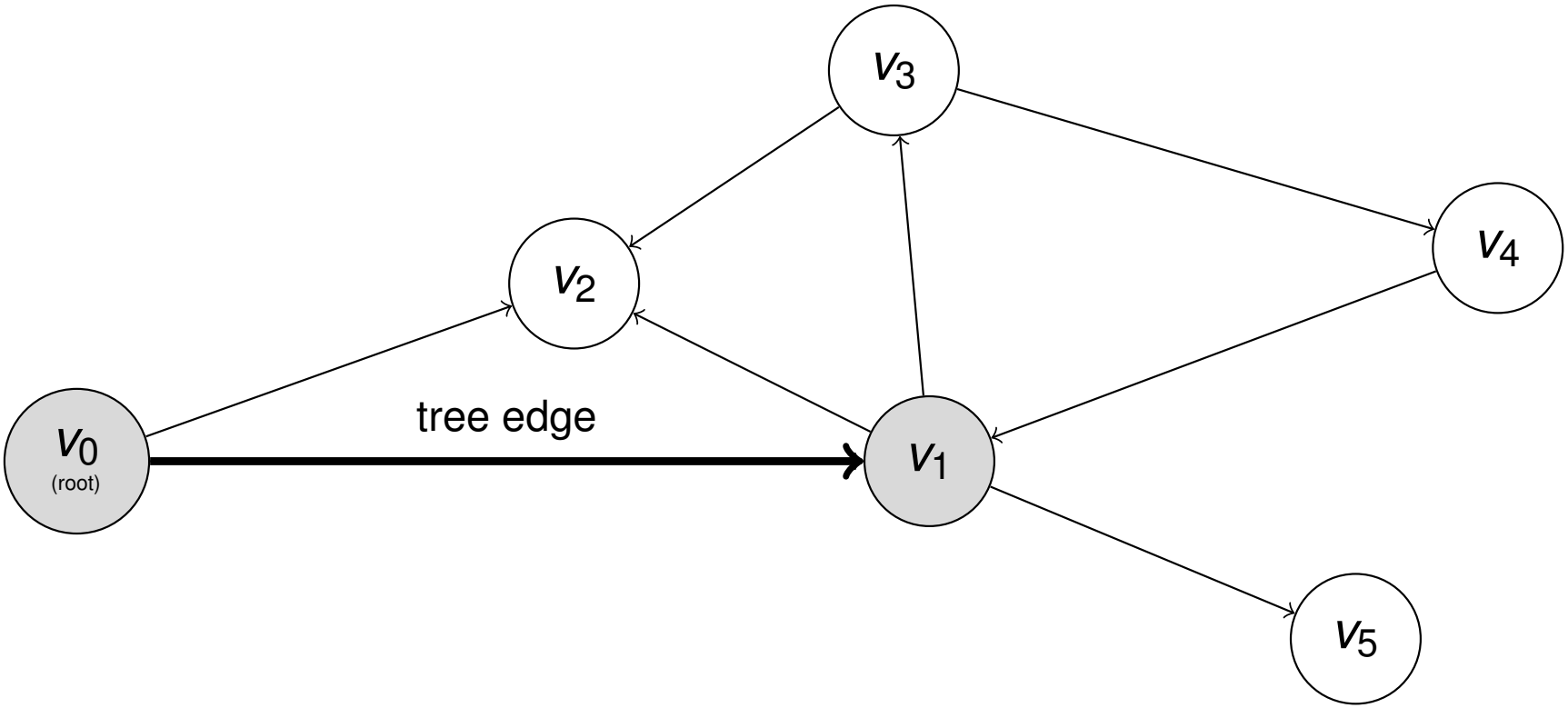
Finding Cycles



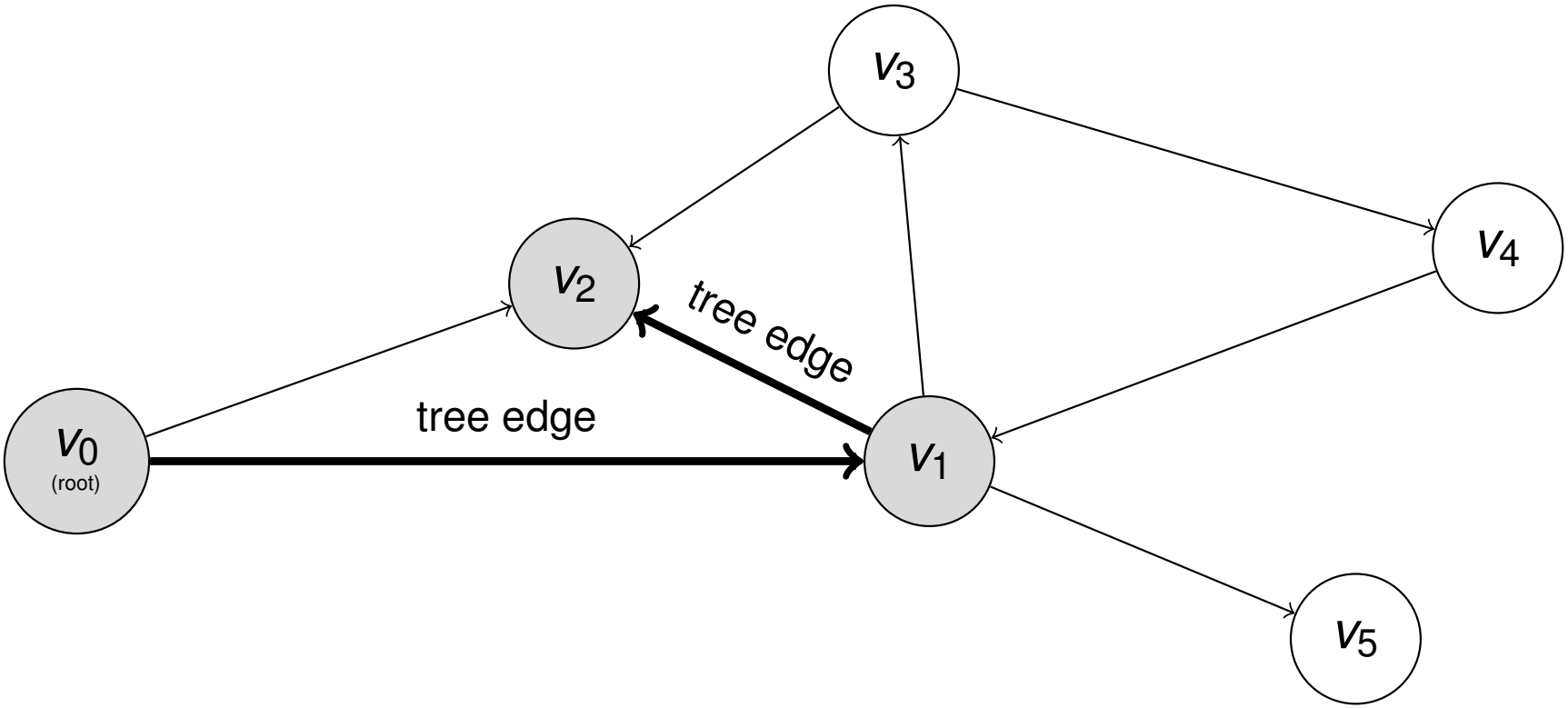
Finding Cycles



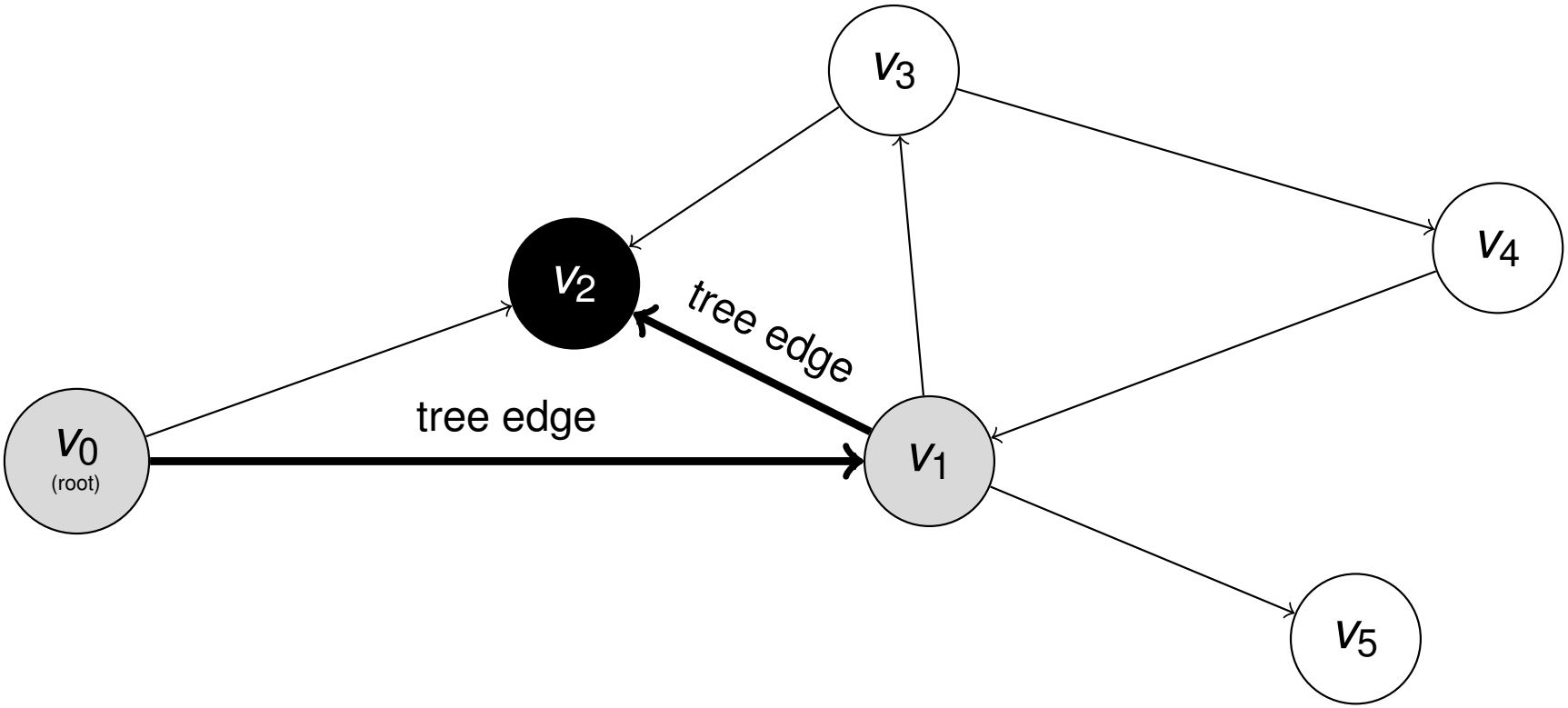
Finding Cycles



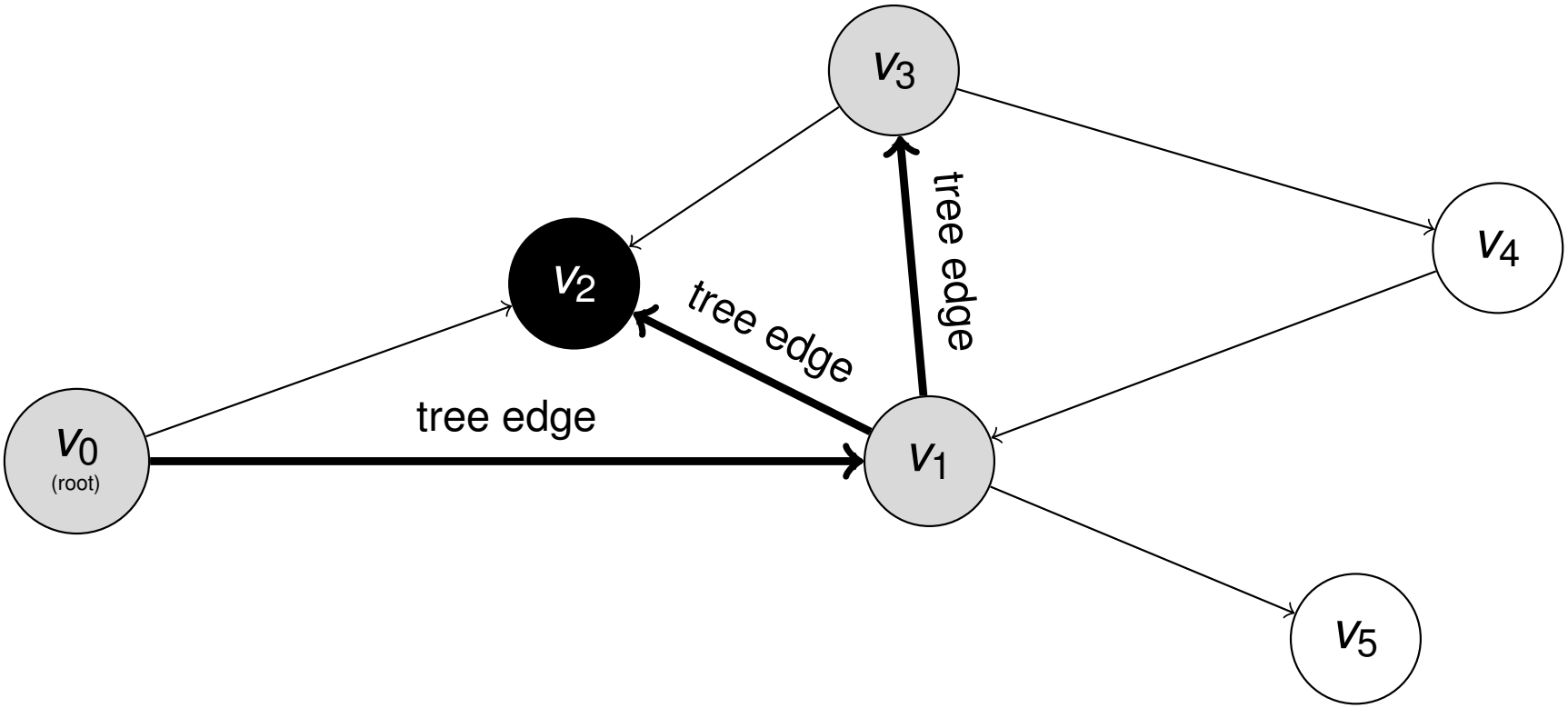
Finding Cycles



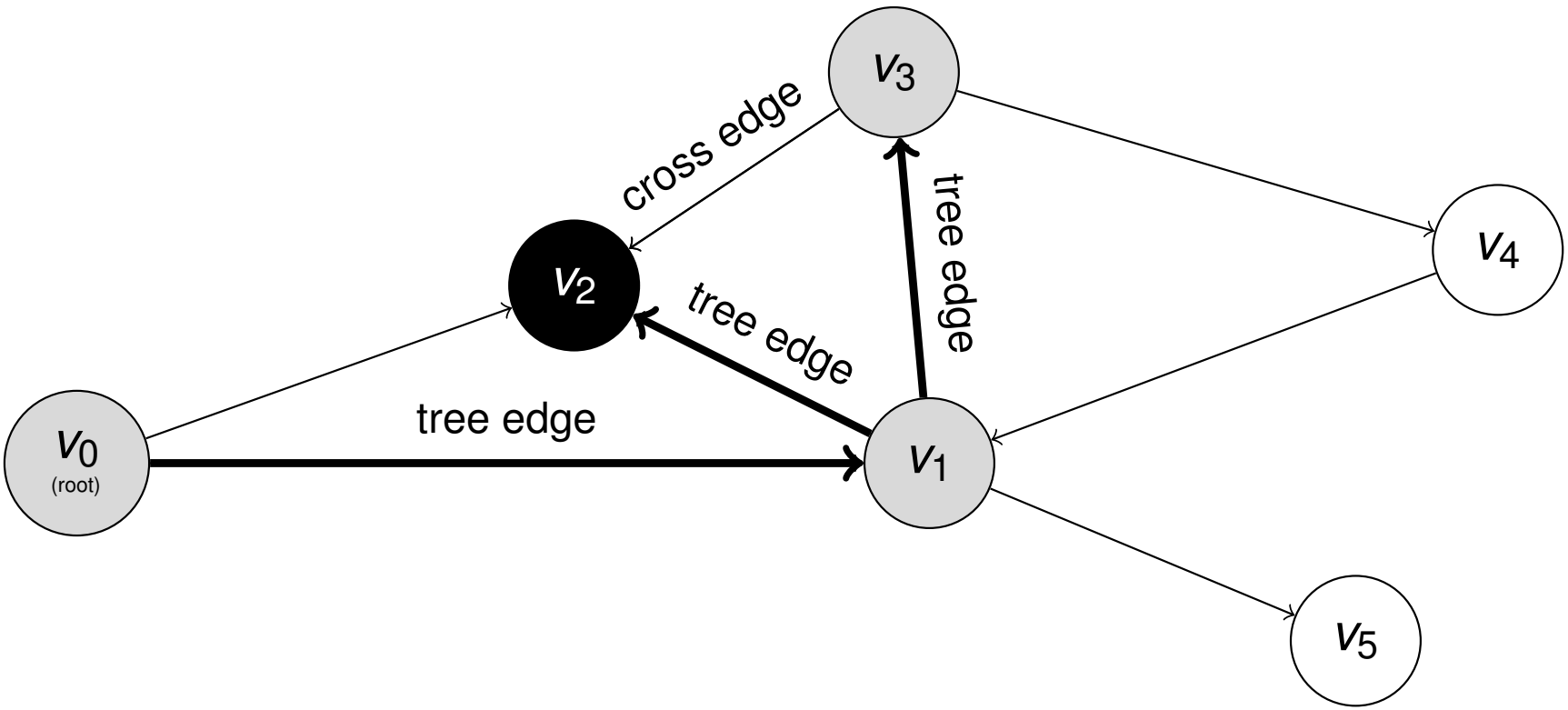
Finding Cycles



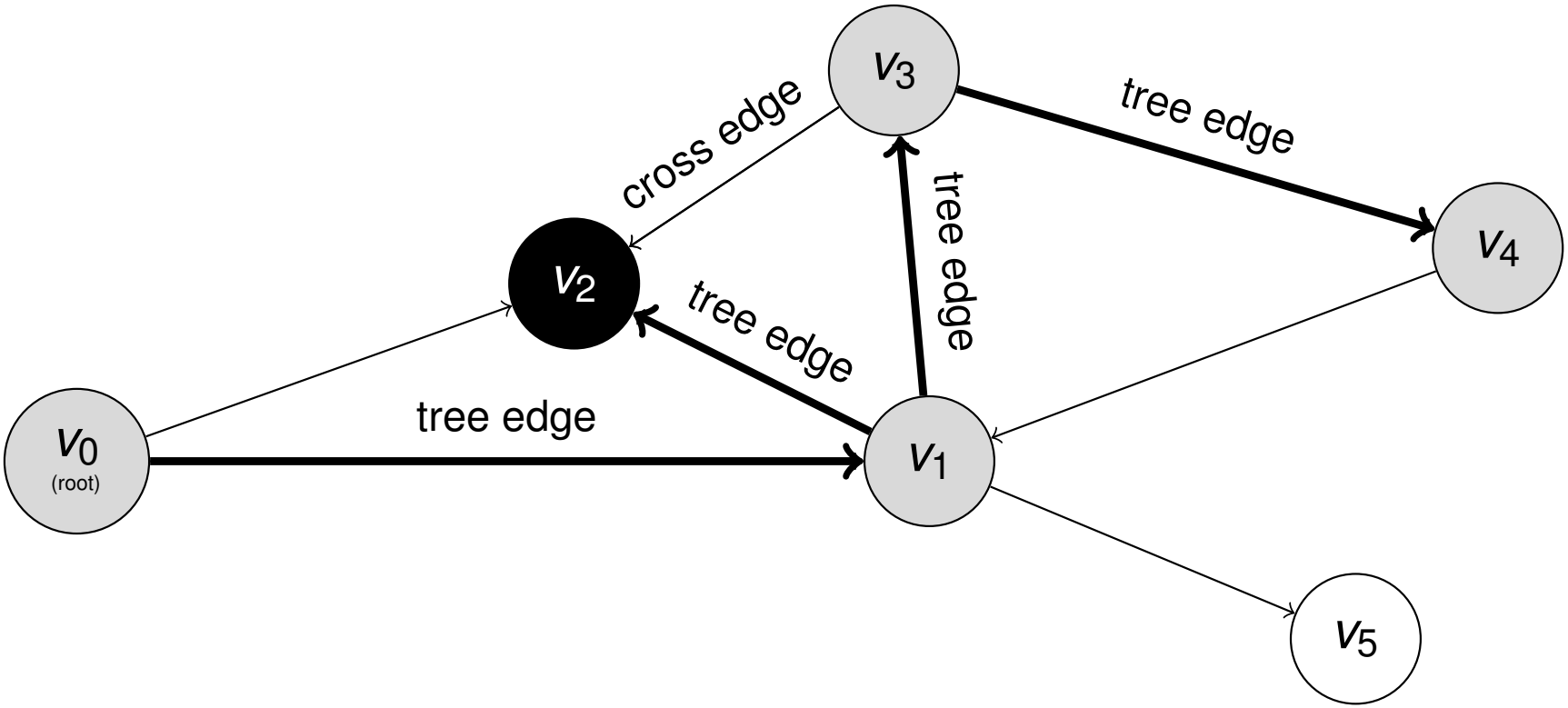
Finding Cycles



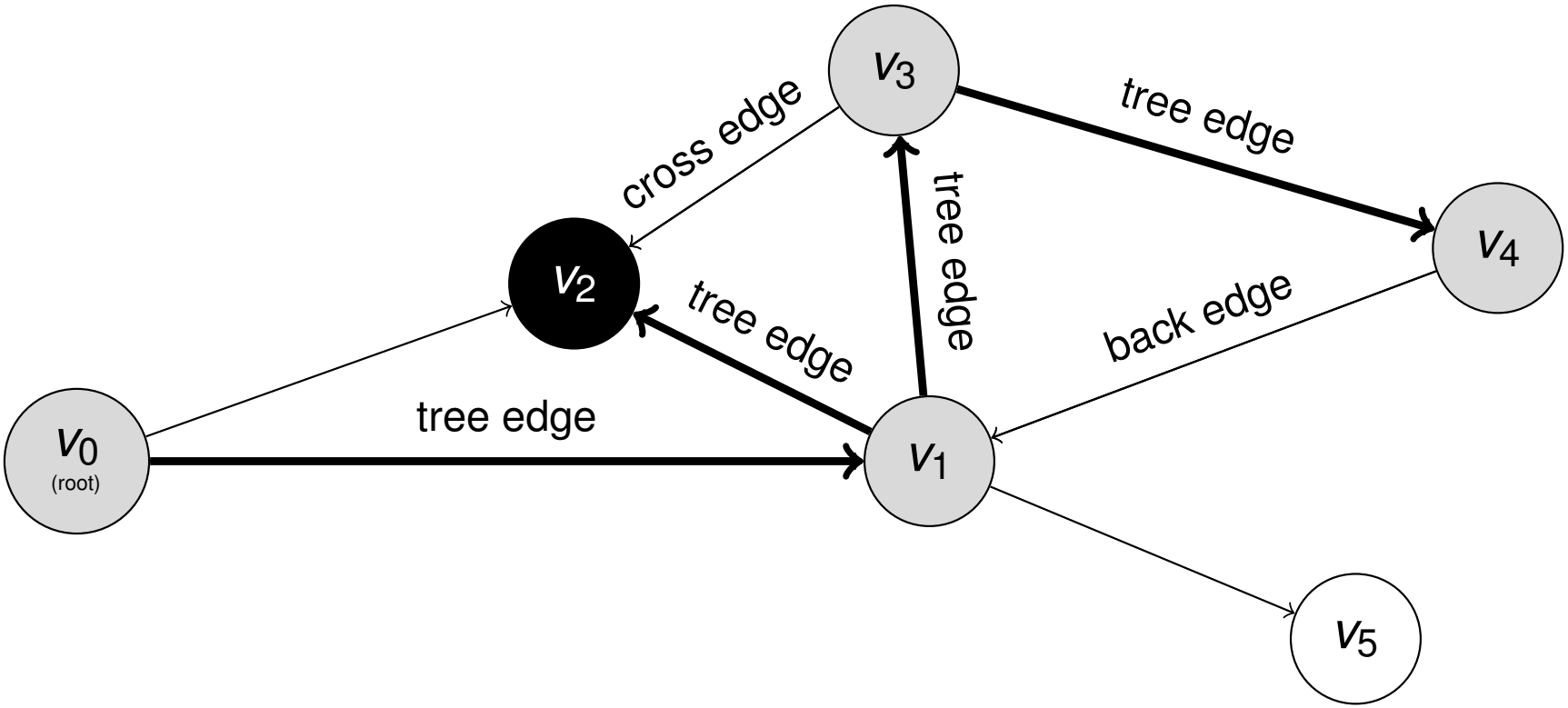
Finding Cycles



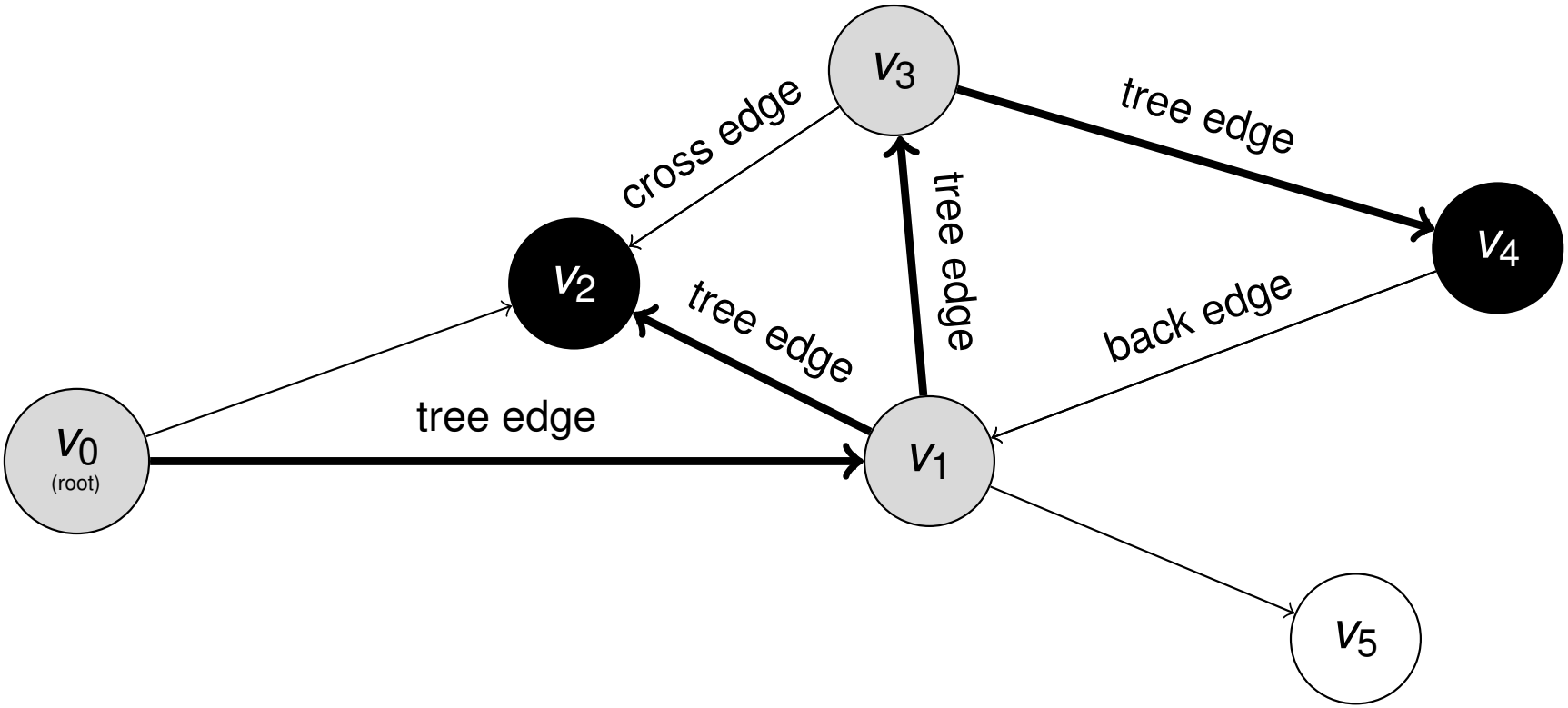
Finding Cycles



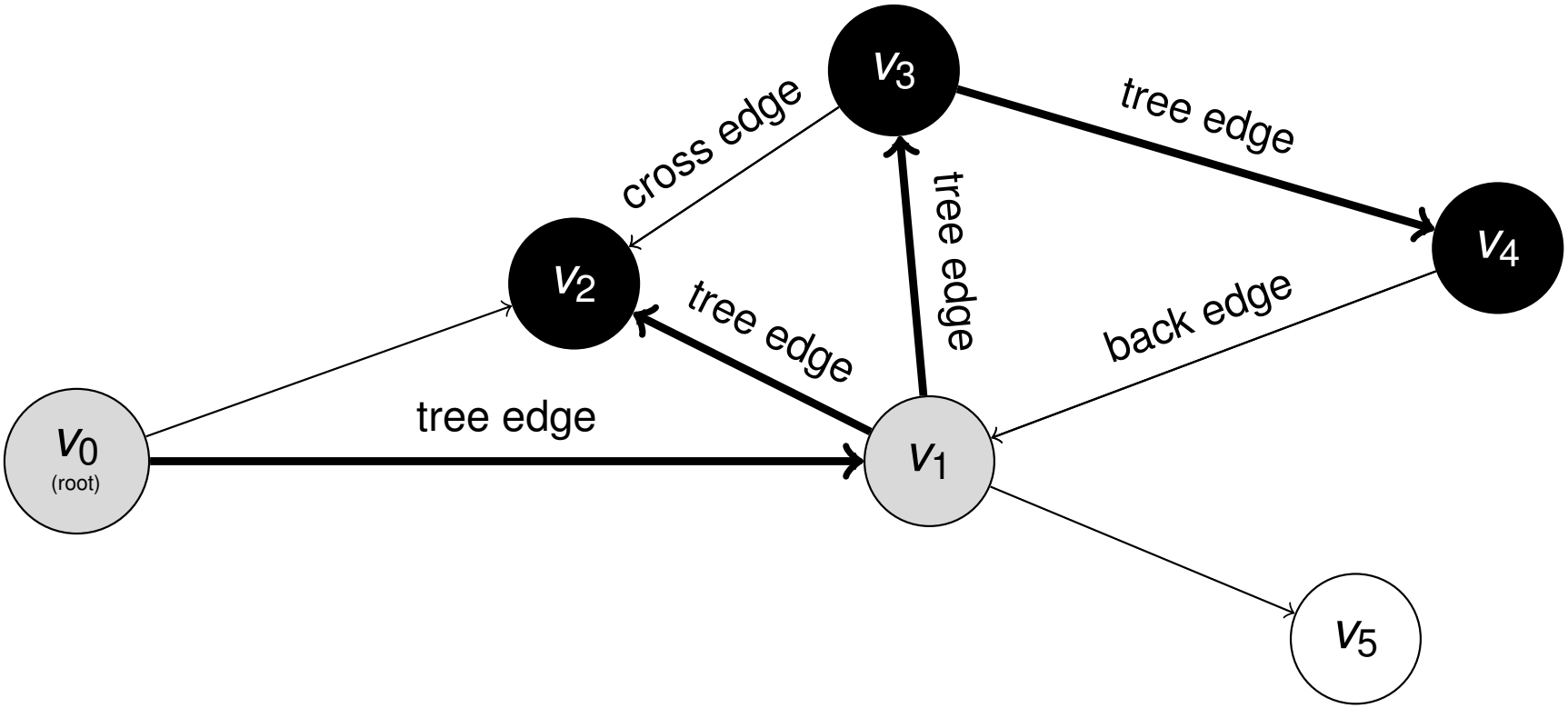
Finding Cycles



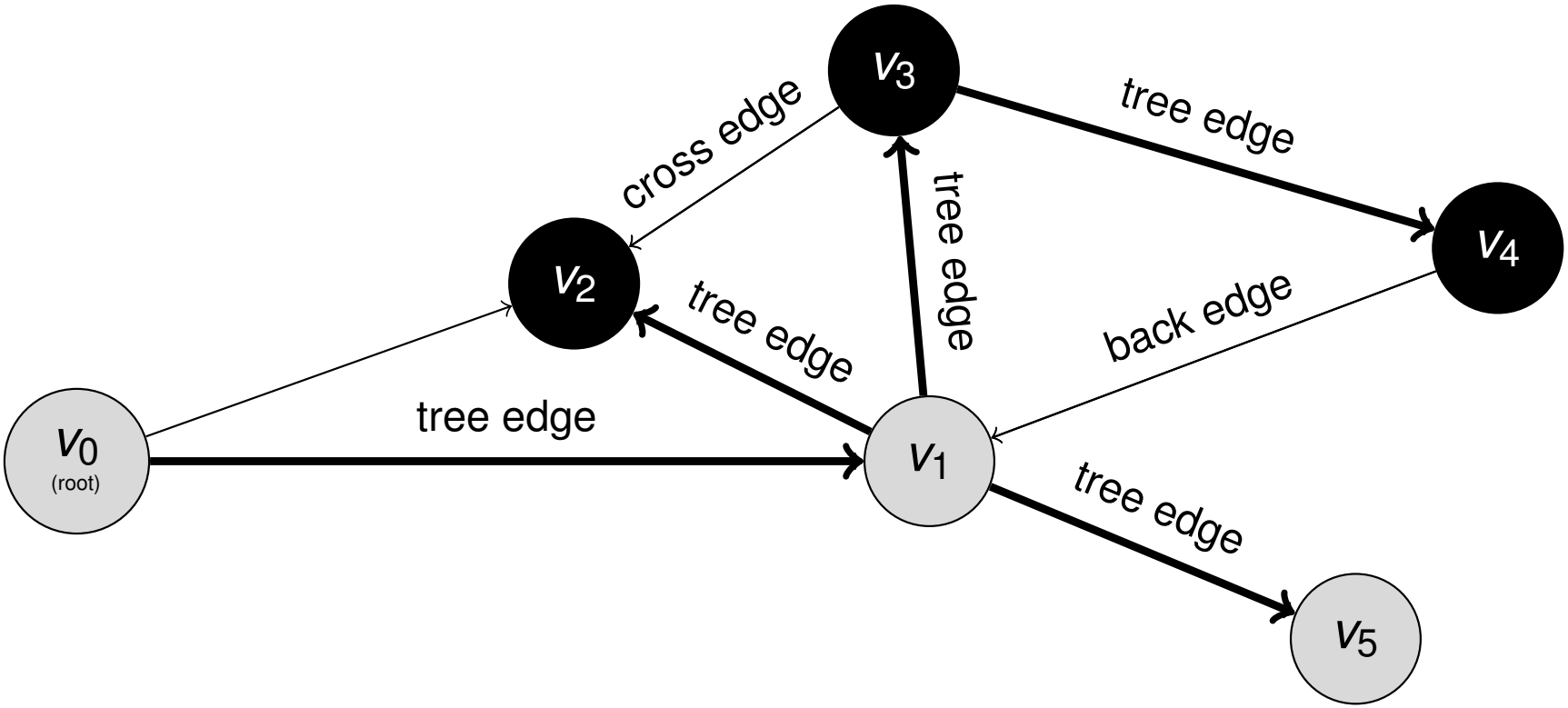
Finding Cycles



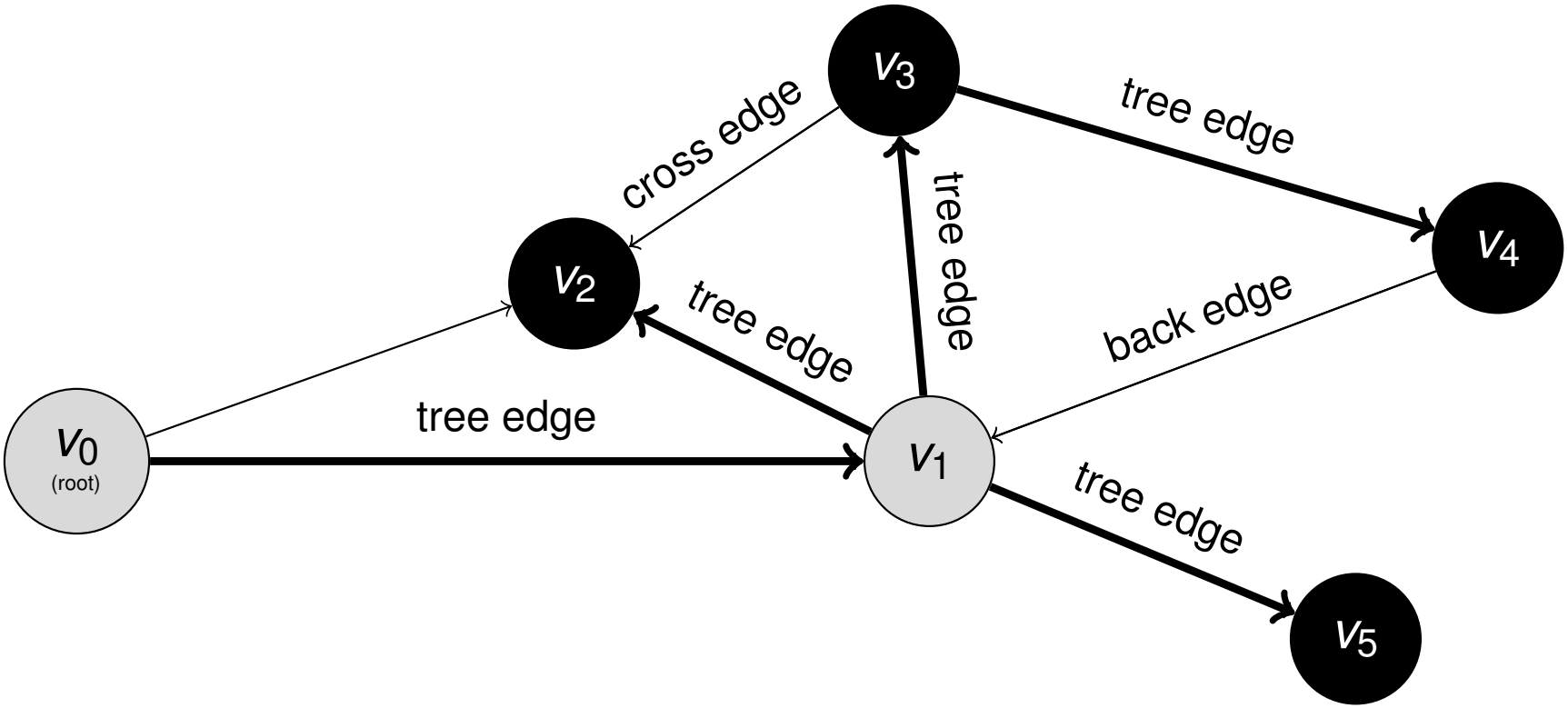
Finding Cycles



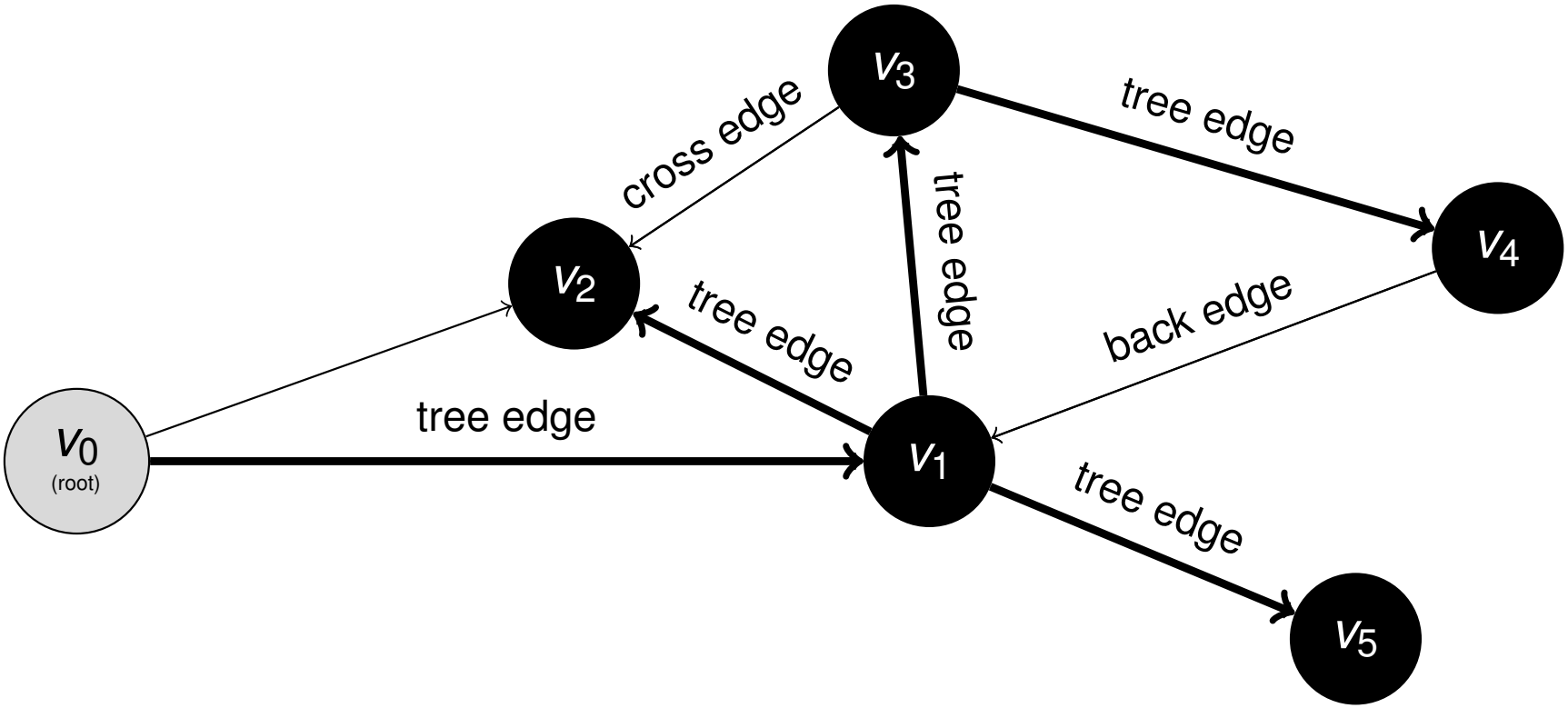
Finding Cycles



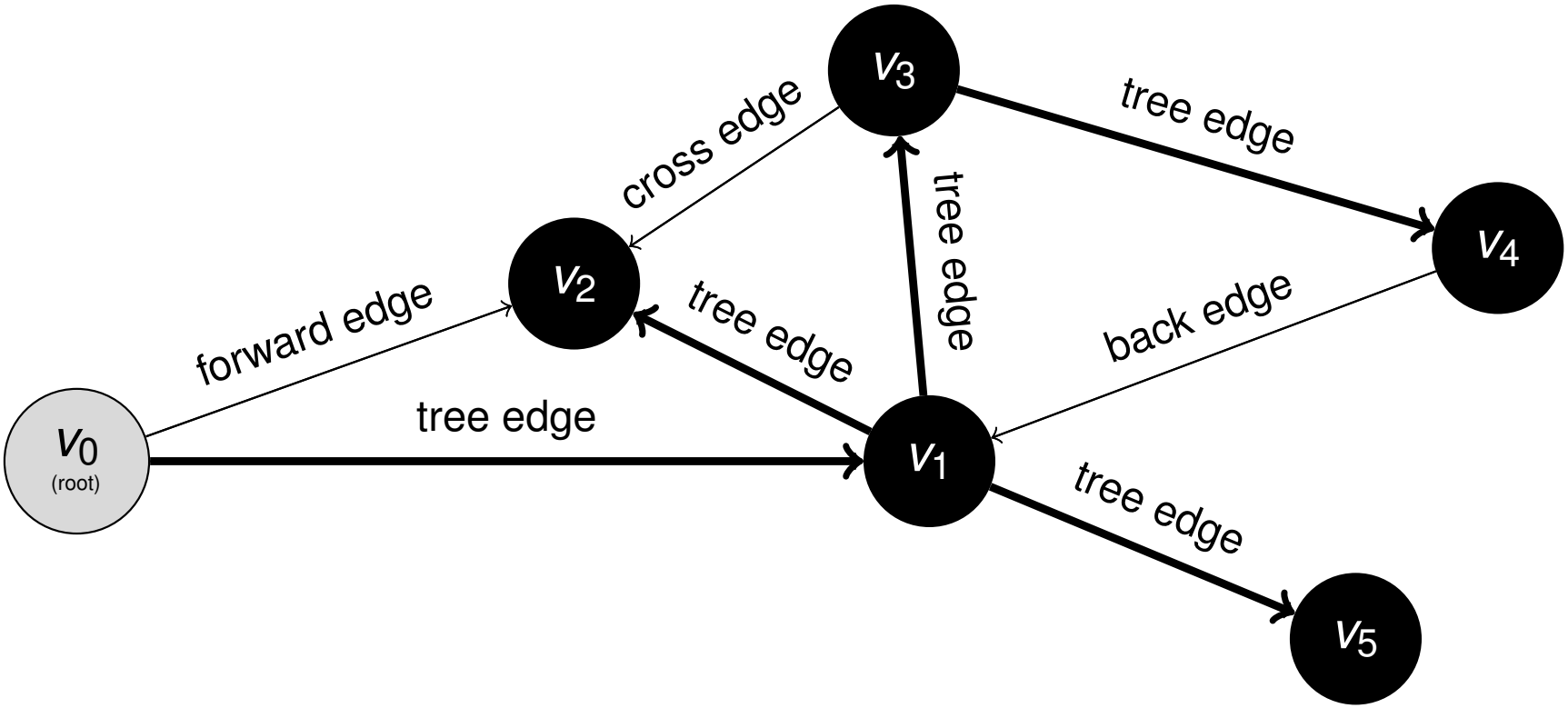
Finding Cycles



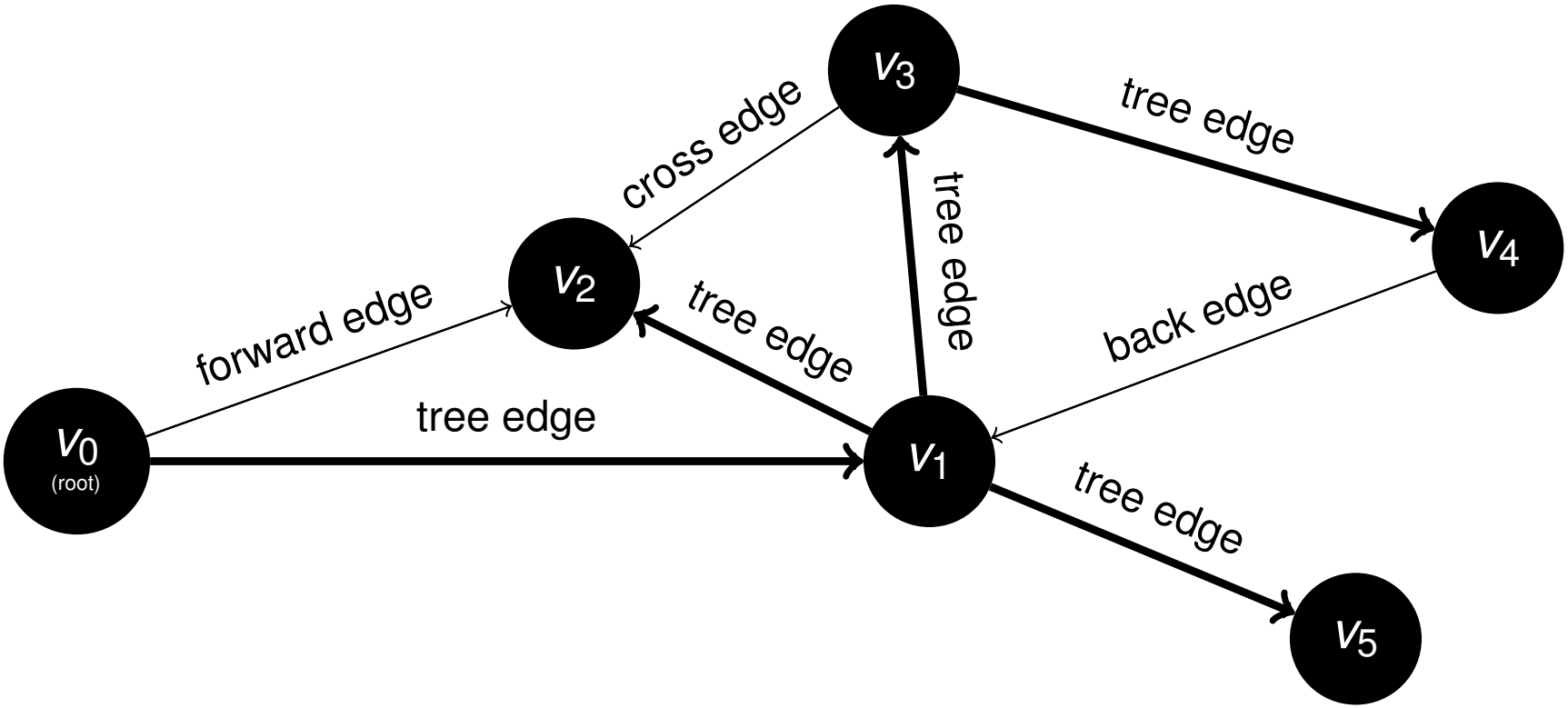
Finding Cycles



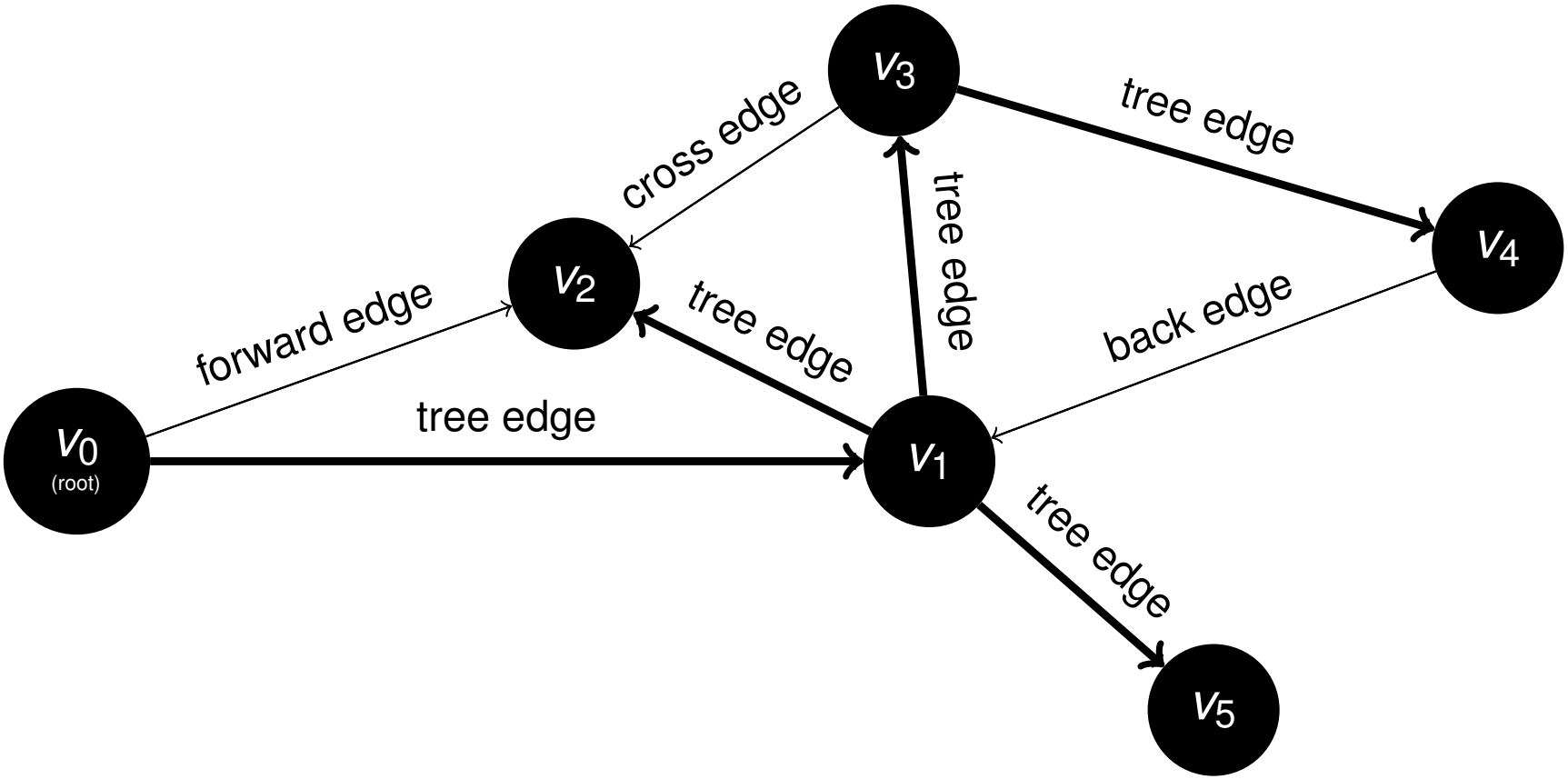
Finding Cycles



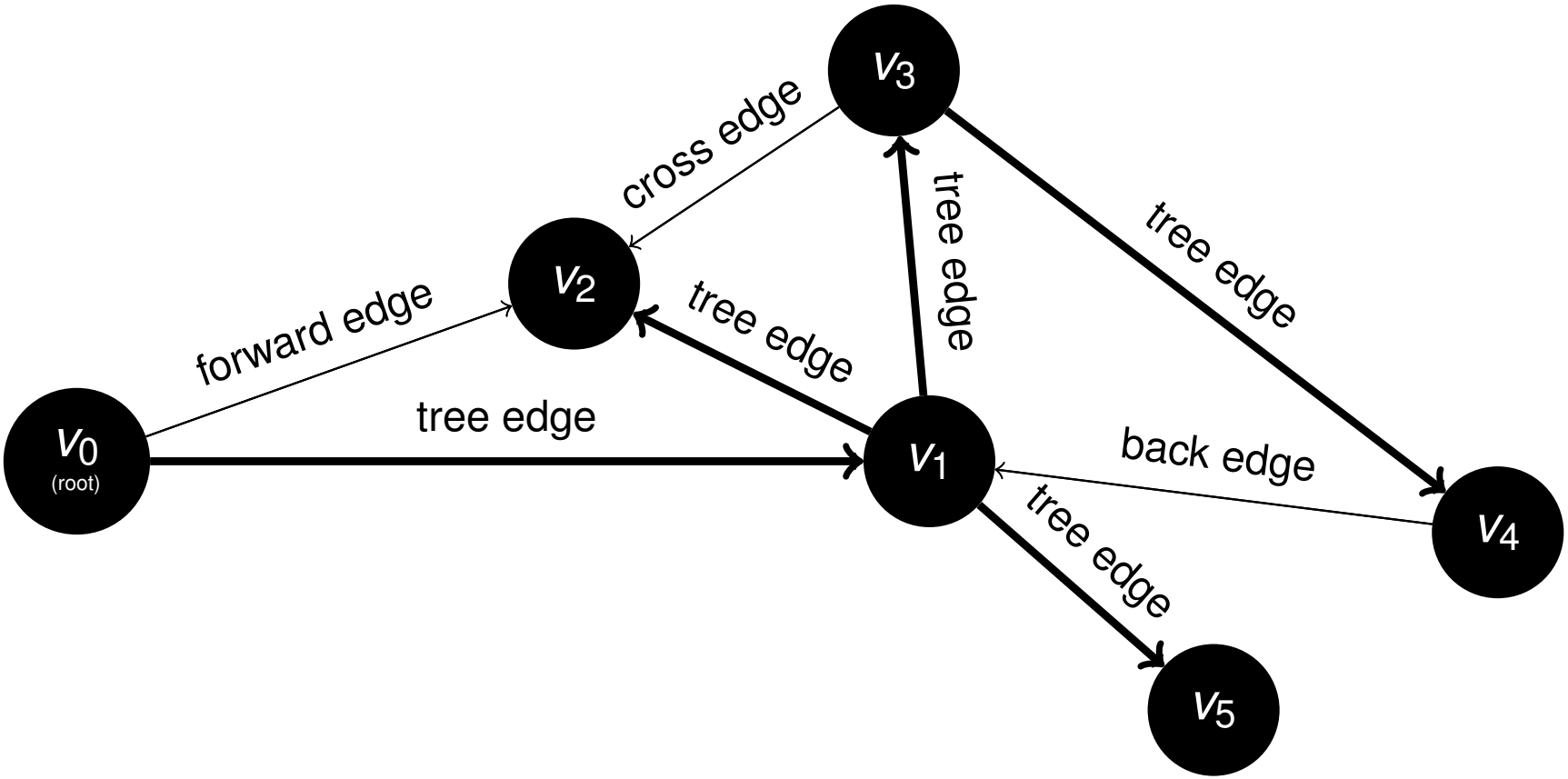
Finding Cycles



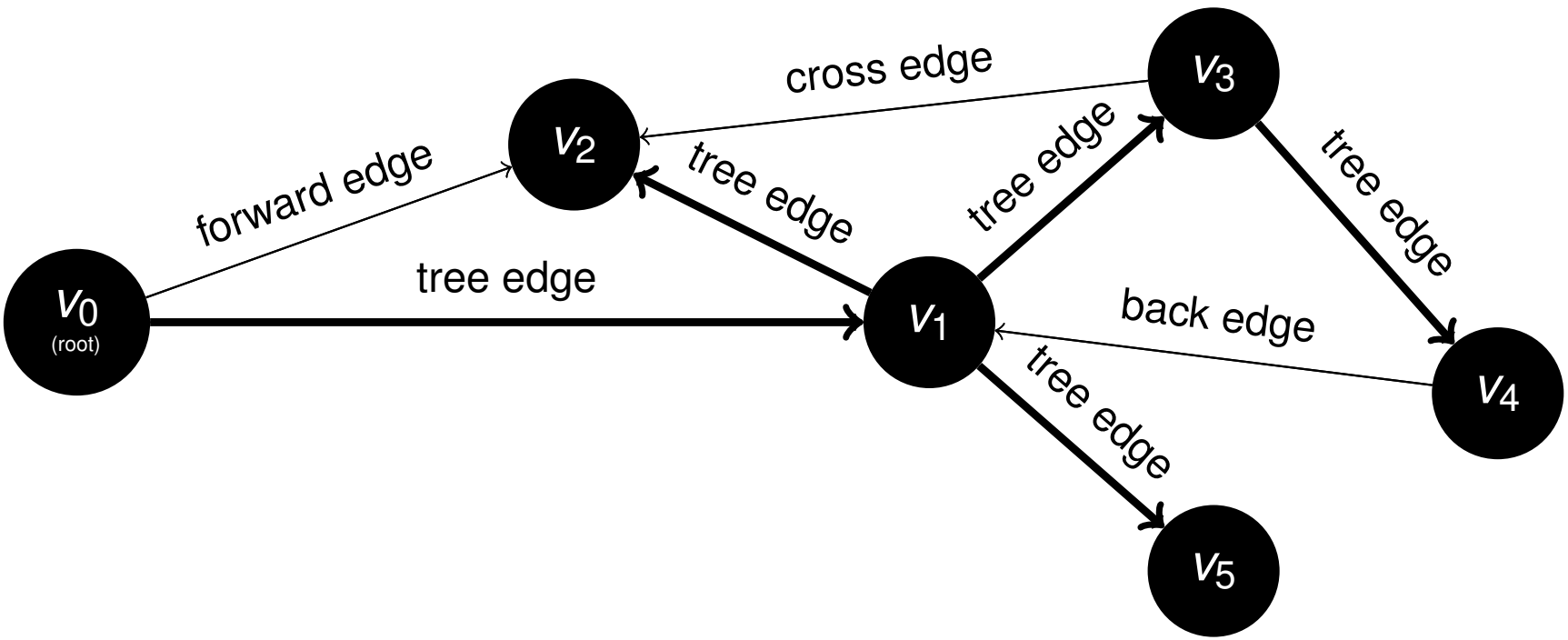
Finding Cycles



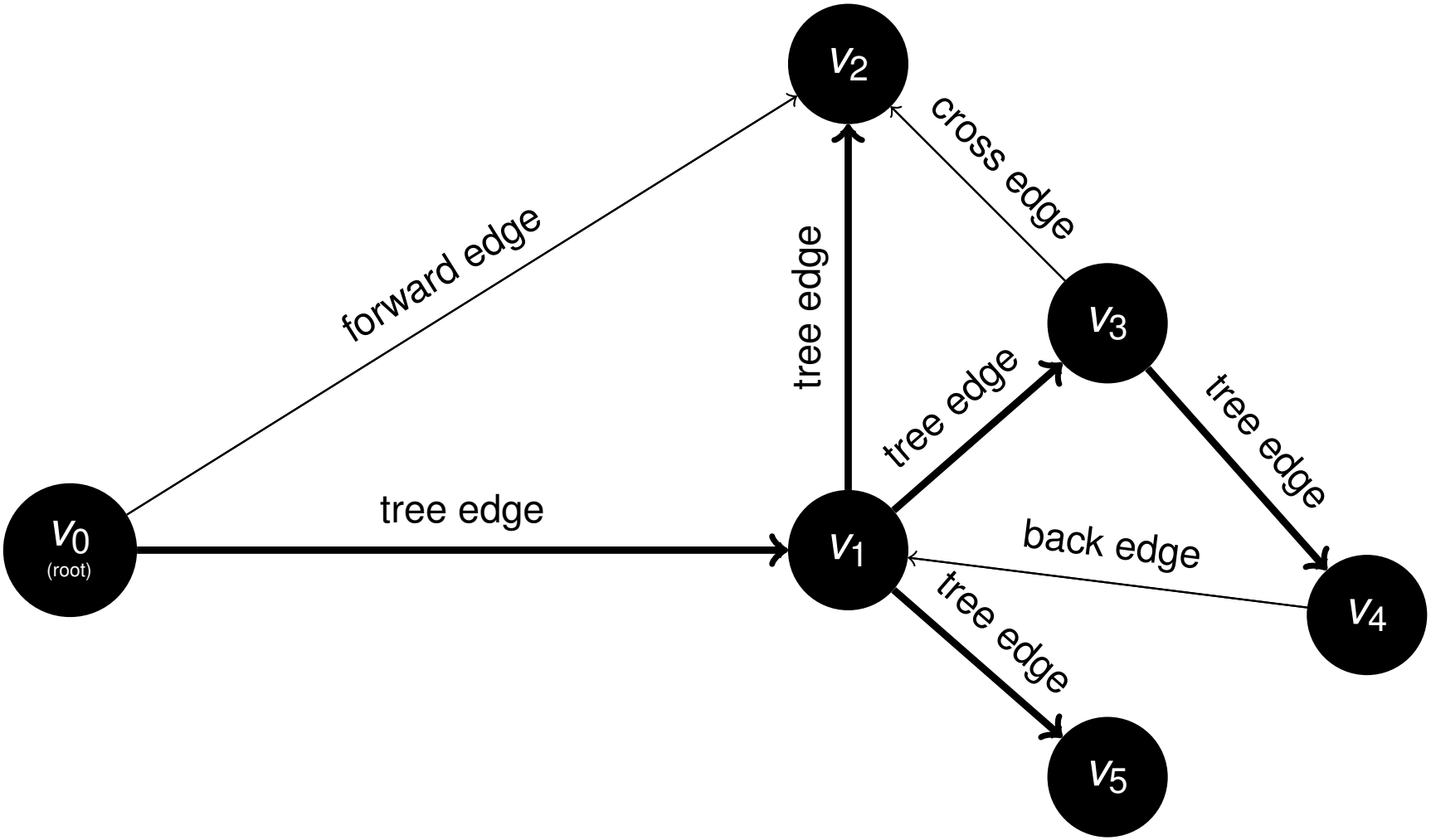
Finding Cycles



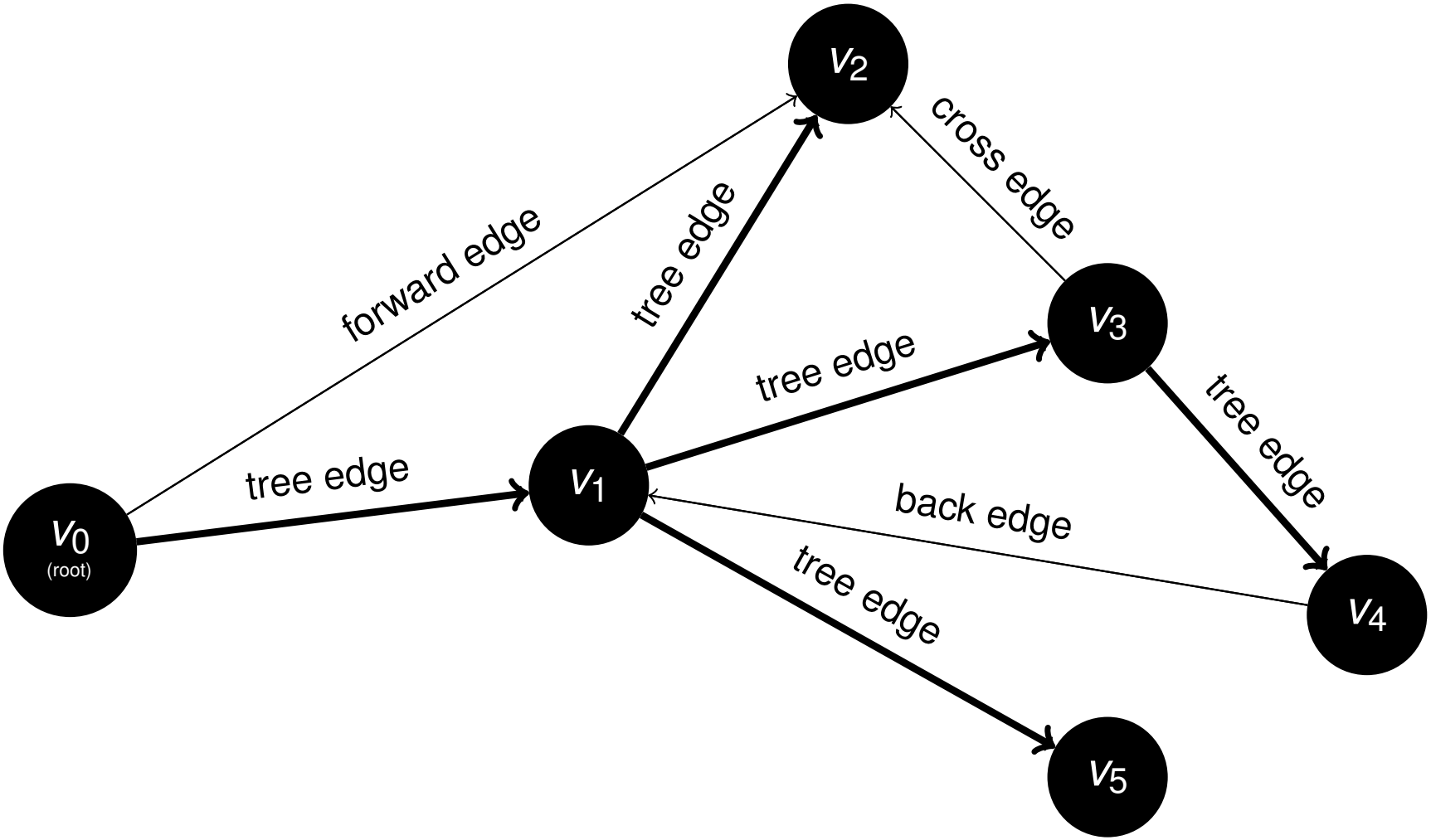
Finding Cycles



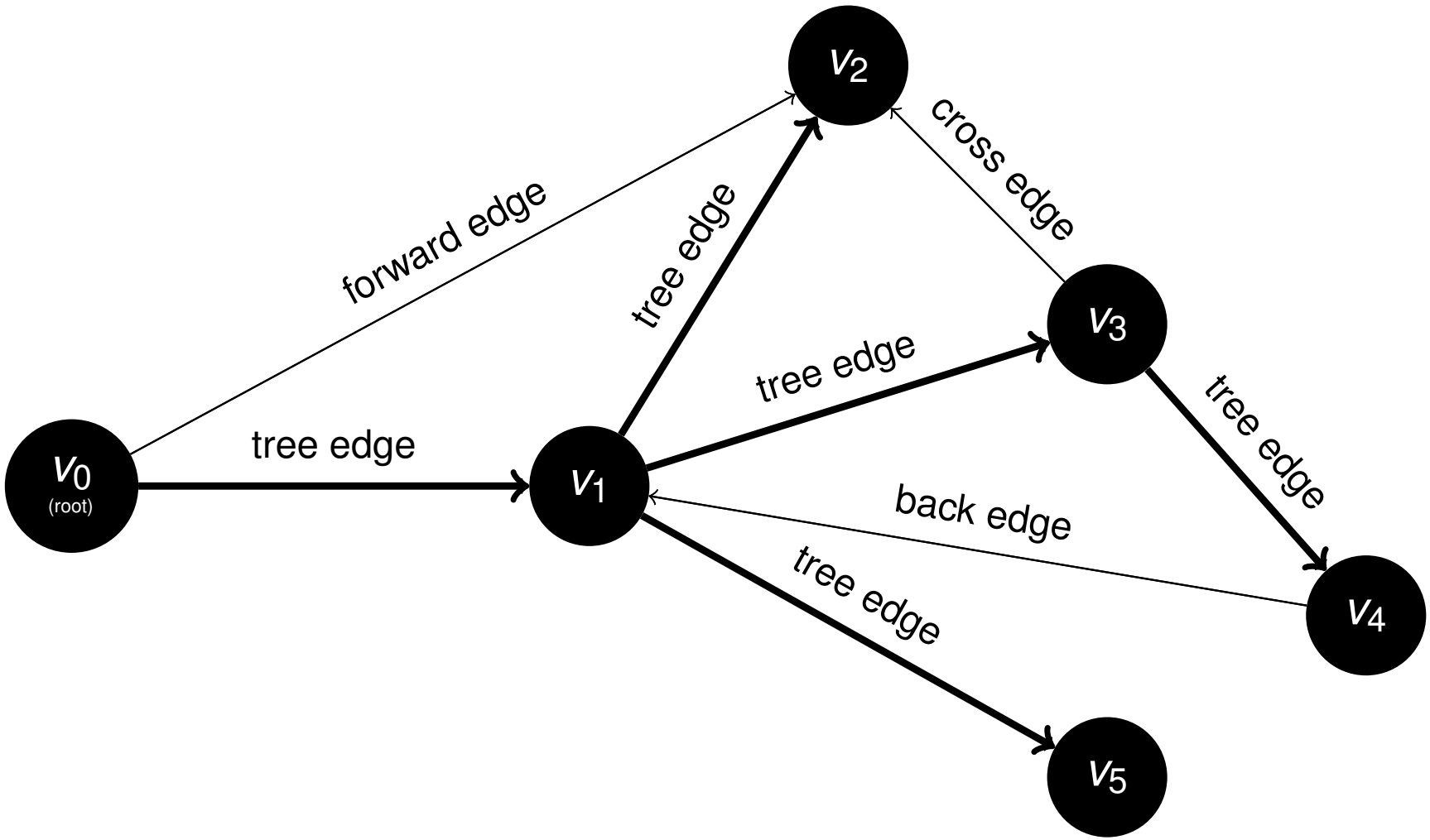
Finding Cycles



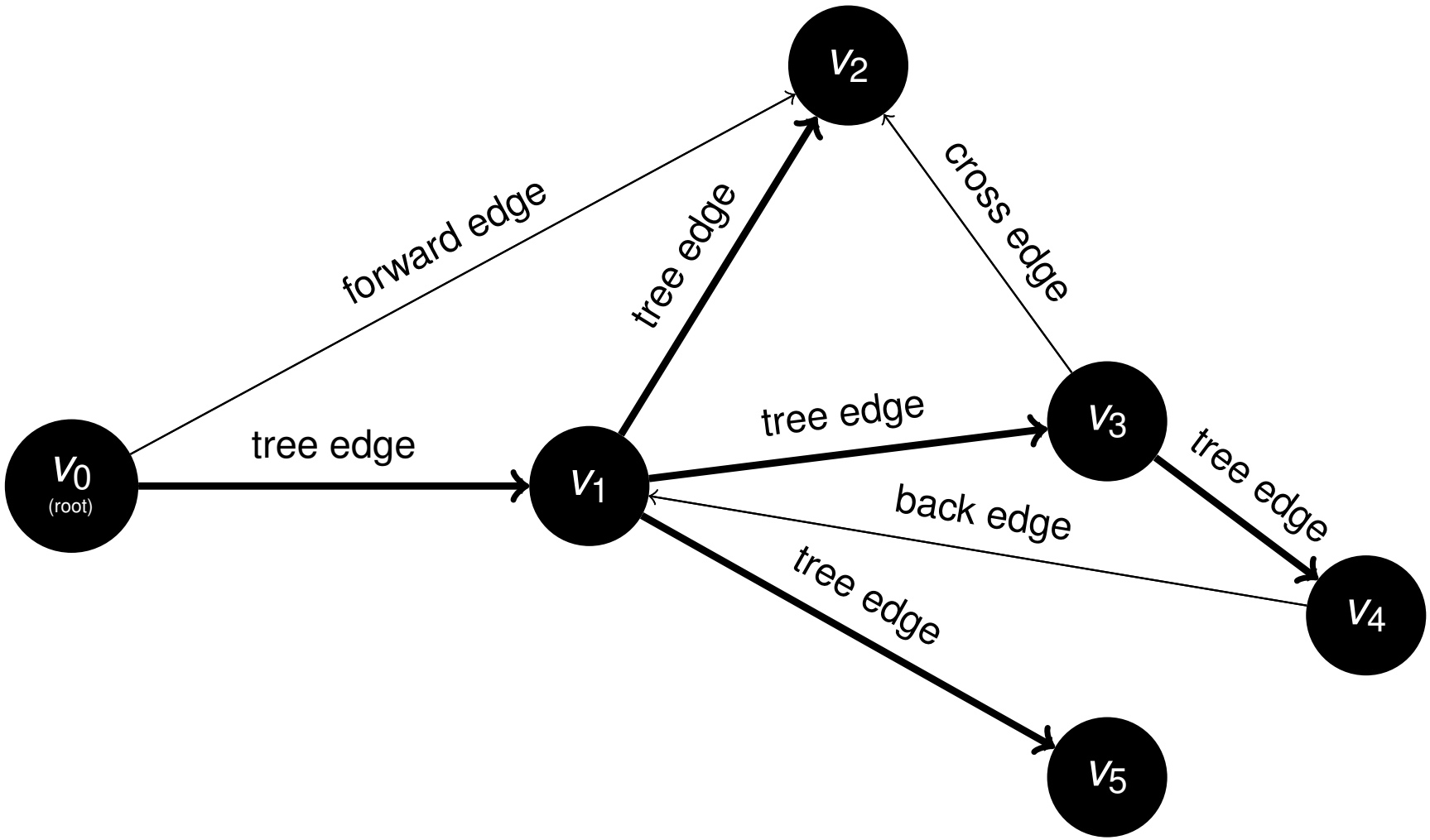
Finding Cycles



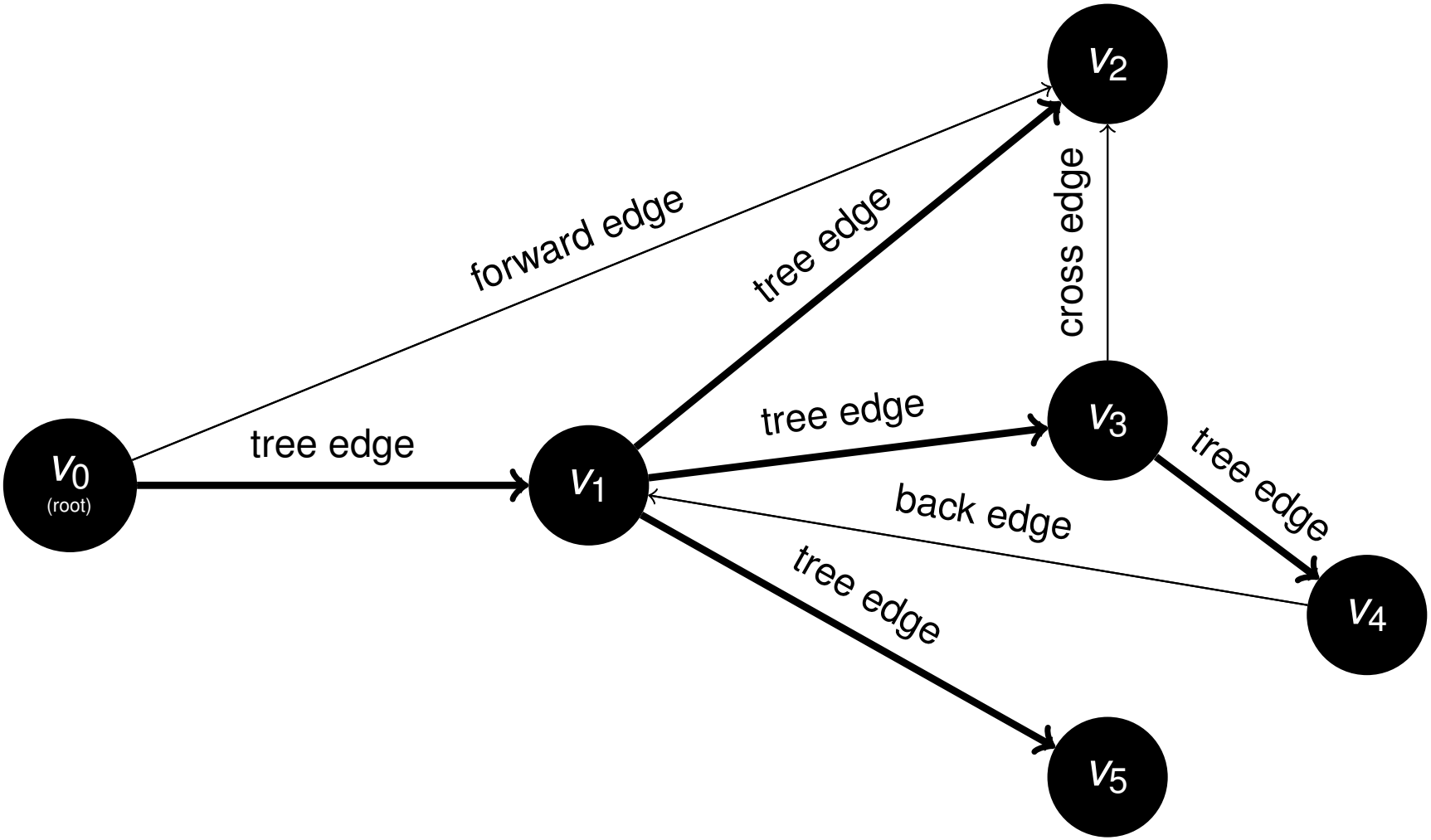
Finding Cycles



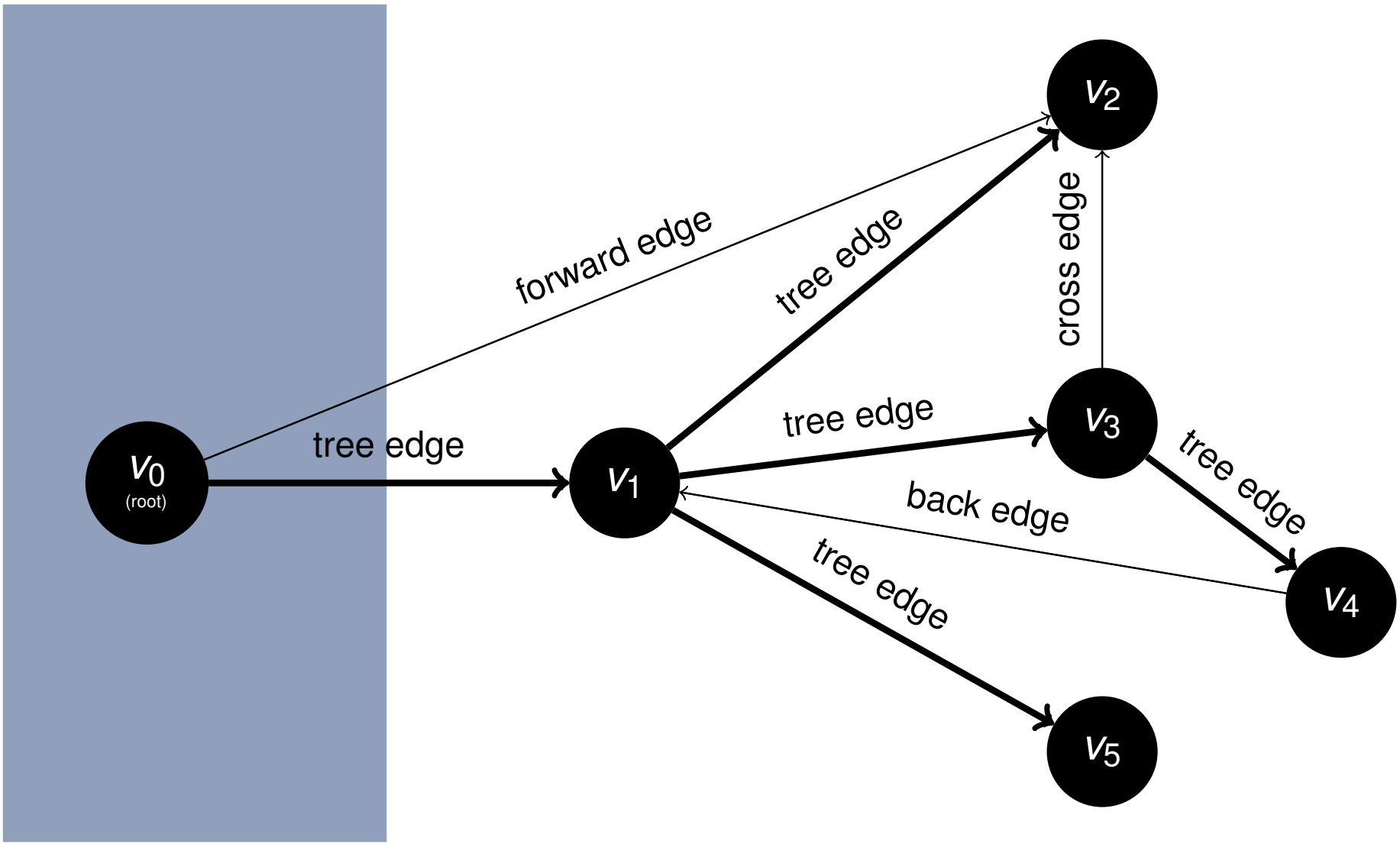
Finding Cycles



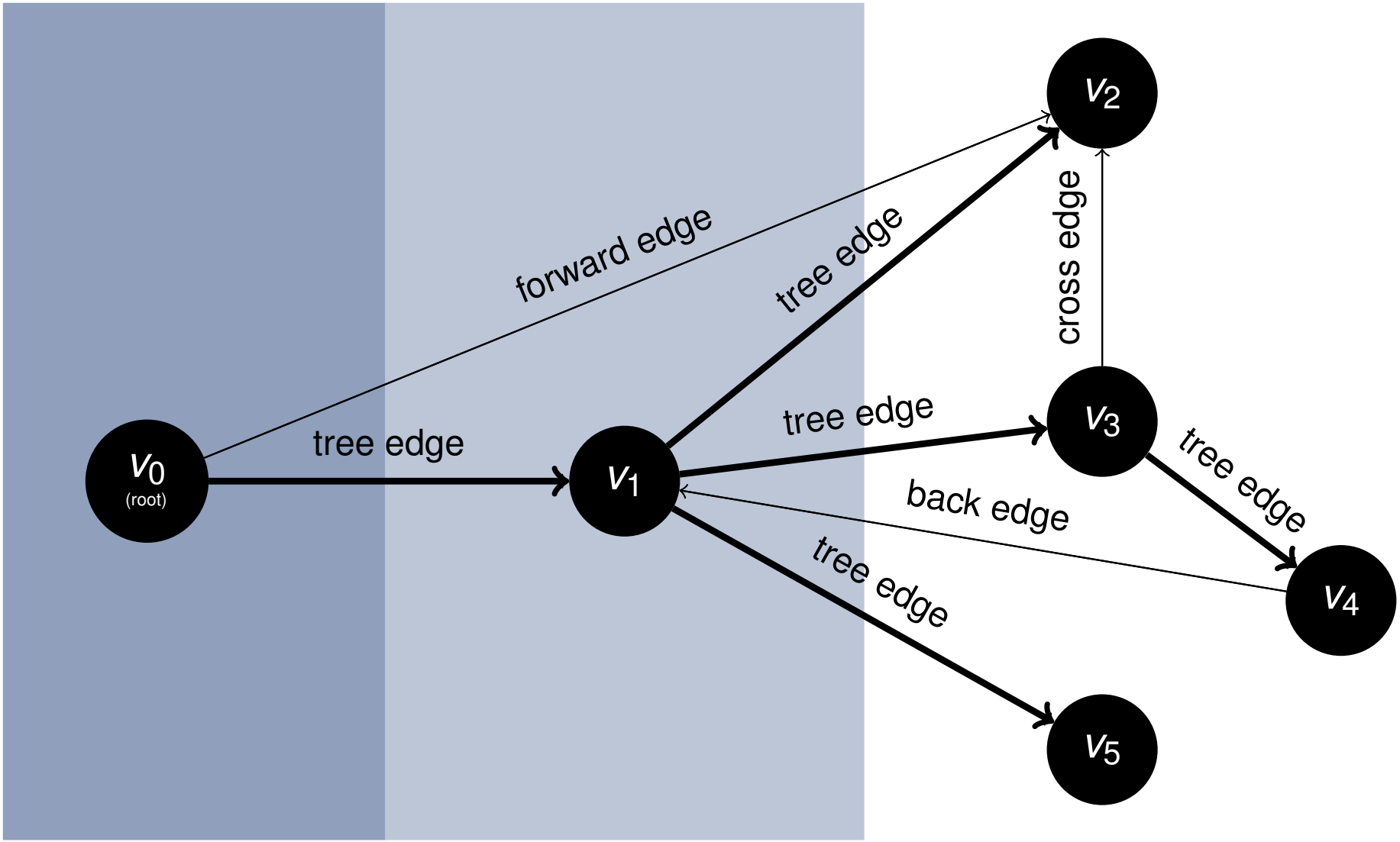
Finding Cycles



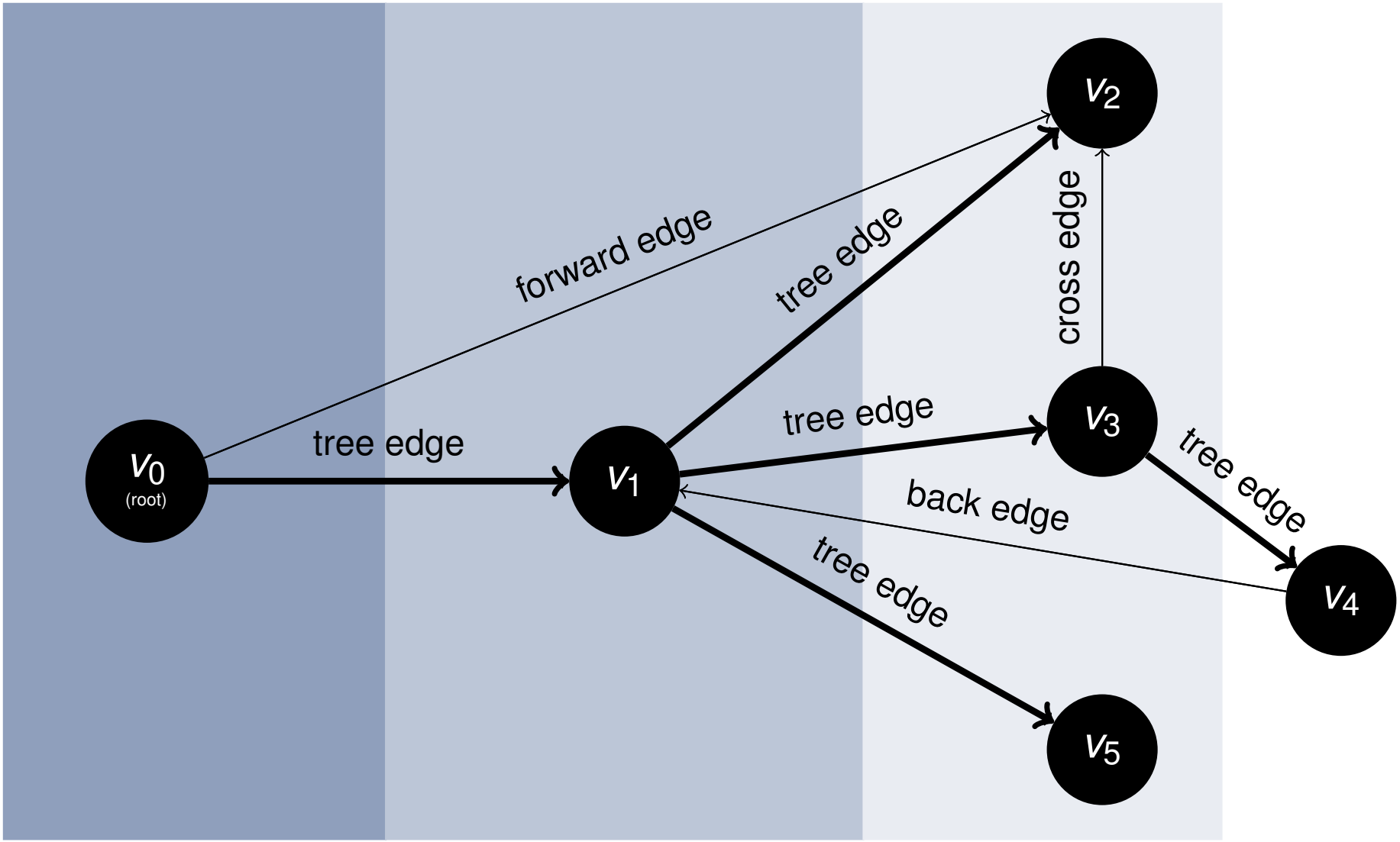
Finding Cycles



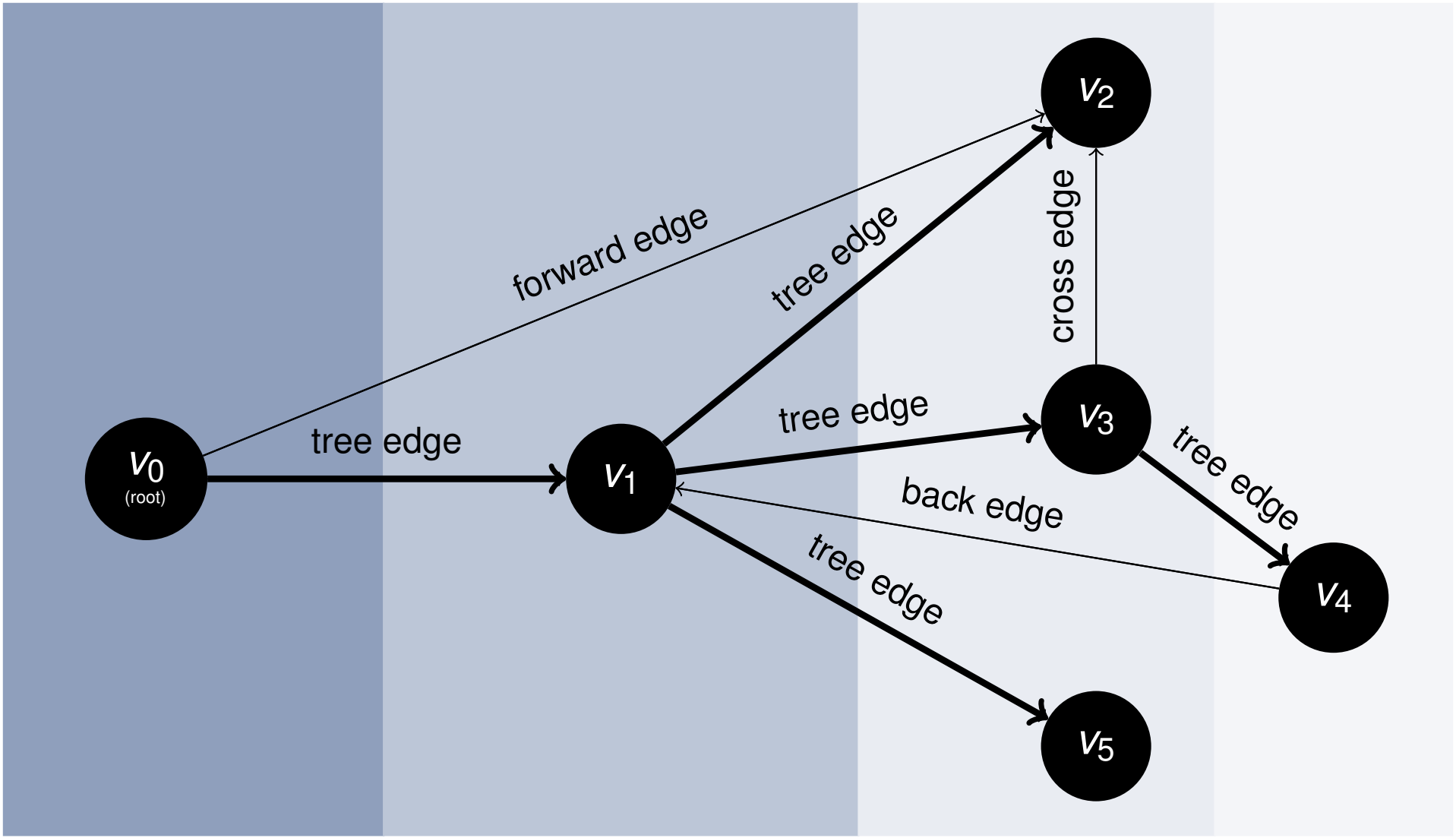
Finding Cycles



Finding Cycles



Finding Cycles



Finding Cycles

```
vector<vector<int>>
G { {1,2},
    {2,3,5},
    {},
    {2,4},
    {1},
    {} };
```

Finding Cycles

```
void cyclefind_dir(int current) {
   cout << current << " ";
   state[current] = 1;
   for (int i=0; i<G[current].size(); ++i) {
      int v = G[current][i];
      if (state[v] == 0) {
         cyclefind_dir(v);
      } else if (state[v] == 1) {
         cout << "BACK EDGE";
      }
   }
   state[current] = 2;
}
```

Topological Sorting

Topological Sorting

- Consider **directed acyclic graph** (**DAG**)

Topological Sorting

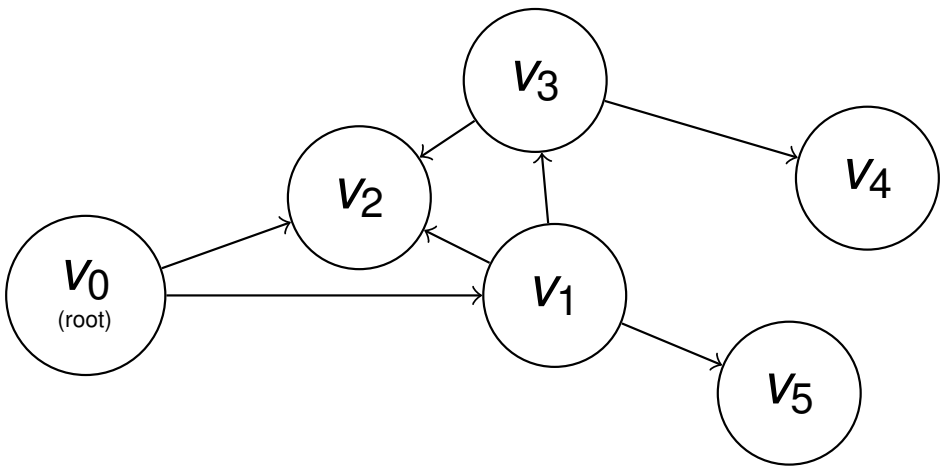
- Consider **directed acyclic graph** (**DAG**)
- Arcs define **partial ordering**

Topological Sorting

- Consider **directed acyclic graph** (**DAG**)
- Arcs define **partial ordering**
- Arc from v to w means $v > w$

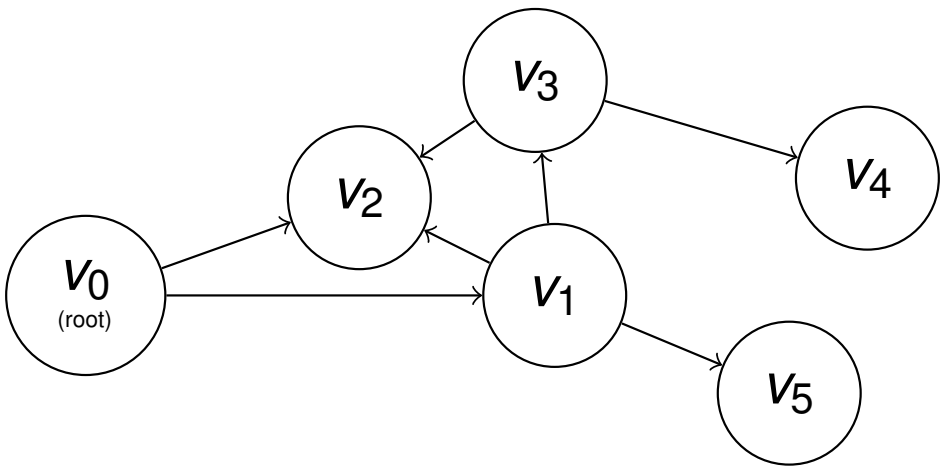
Topological Sorting

- Consider **directed acyclic graph** (**DAG**)
- Arcs define **partial ordering**
- Arc from v to w means $v > w$



Topological Sorting

- Consider **directed acyclic graph** (**DAG**)
- Arcs define **partial ordering**
- Arc from v to w means $v > w$



$$v_2 < v_4 < v_3 < v_5 < v_1 < v_0$$

Topological Sorting

```
vector<vector<int>>
G { {1,2},
    {2,3,5},
    {},
    {2,4},
    {},
    {} };
```

Topological Sorting

- Use DFS

Topological Sorting

- Use DFS
- After recursive call, all “smaller” vertices have been visited

Topological Sorting

- Use DFS
- After recursive call, all “smaller” vertices have been visited
- Then push `current` to back of stack

Topological Sorting

- Use DFS
- After recursive call, all “smaller” vertices have been visited
- Then push `current` to back of stack

```
void toposort(int current) {
   state[current] = 1;
   for (int i=0; i<G[current].size(); ++i) {
      int v = G[current][i];
      if (state[v] == 0) {
         toposort(v);
      }
   }
   state[current] = 2;
   topo.push_back(current);
}
```

Cheese Patrol

Cheese Patrol

- Surveillance of Tokyo

Cheese Patrol

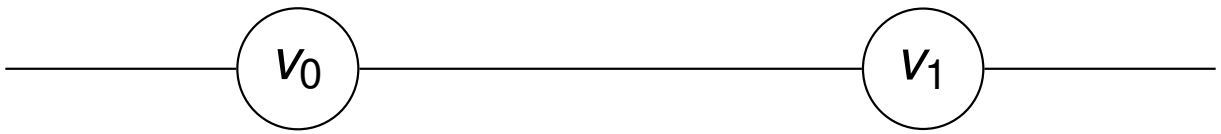
- Surveillance of Tokyo
- Detectives can be placed on streets and intersections

Cheese Patrol

- Surveillance of Tokyo
- Detectives can be placed on streets and intersections
- When on intersection, detectives surveils two streets

Cheese Patrol

- Surveillance of Tokyo
- Detectives can be placed on streets and intersections
- When on intersection, detectives surveils two streets

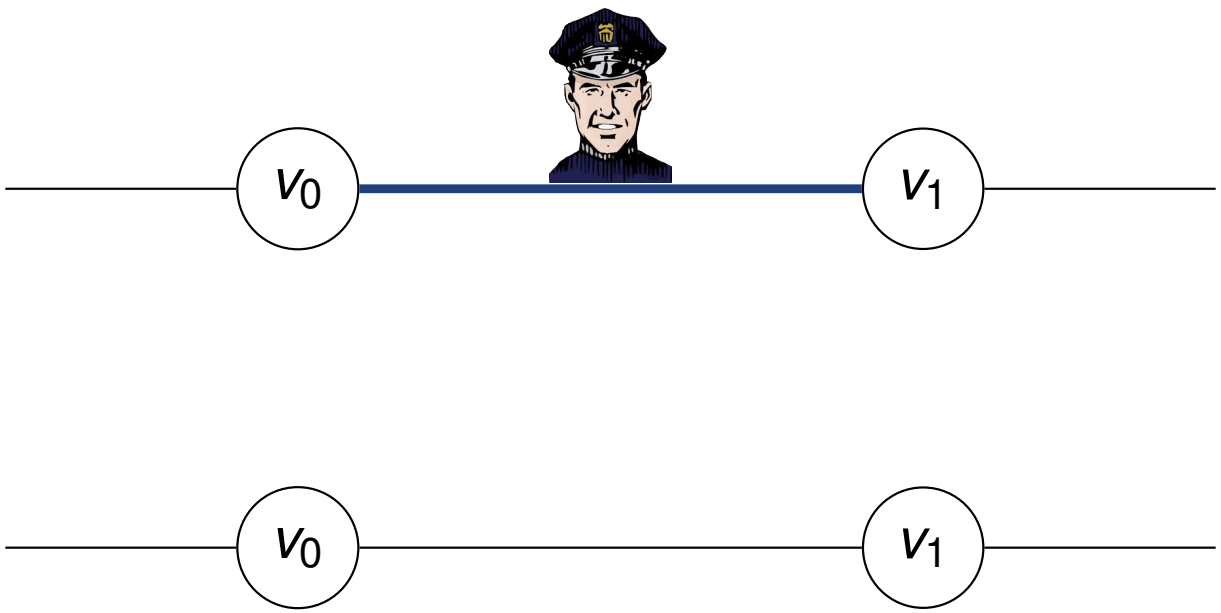


Cheese Patrol

- Surveillance of Tokyo
- Detectives can be placed on streets and intersections
- When on intersection, detectives surveils two streets

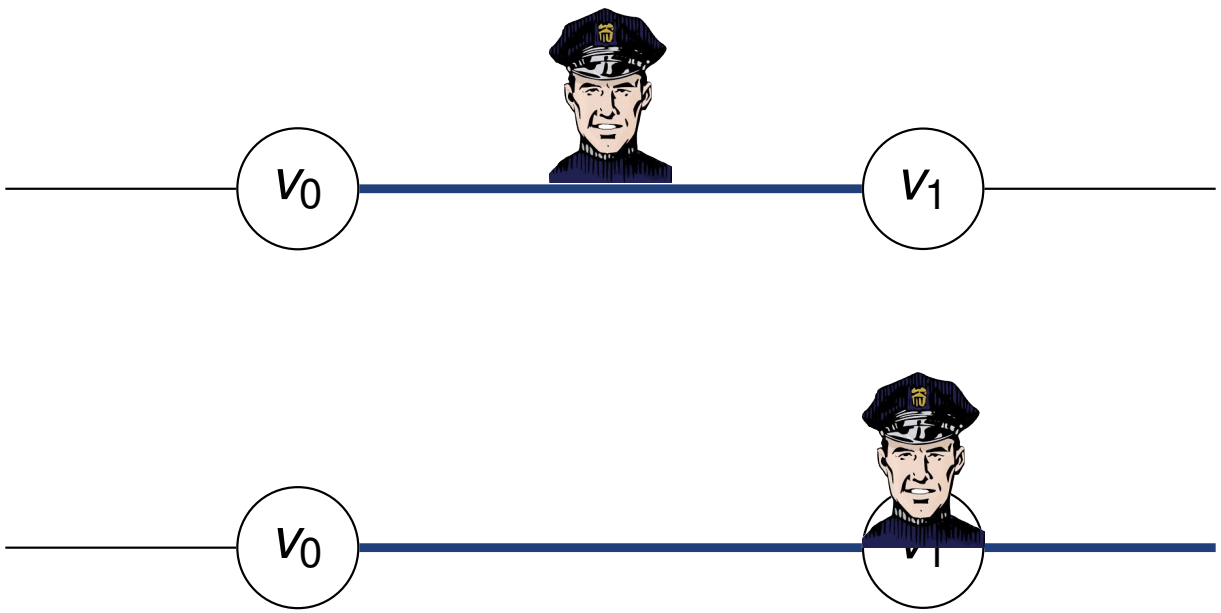
Cheese Patrol

- Surveillance of Tokyo
- Detectives can be placed on streets and intersections
- When on intersection, detectives surveils two streets



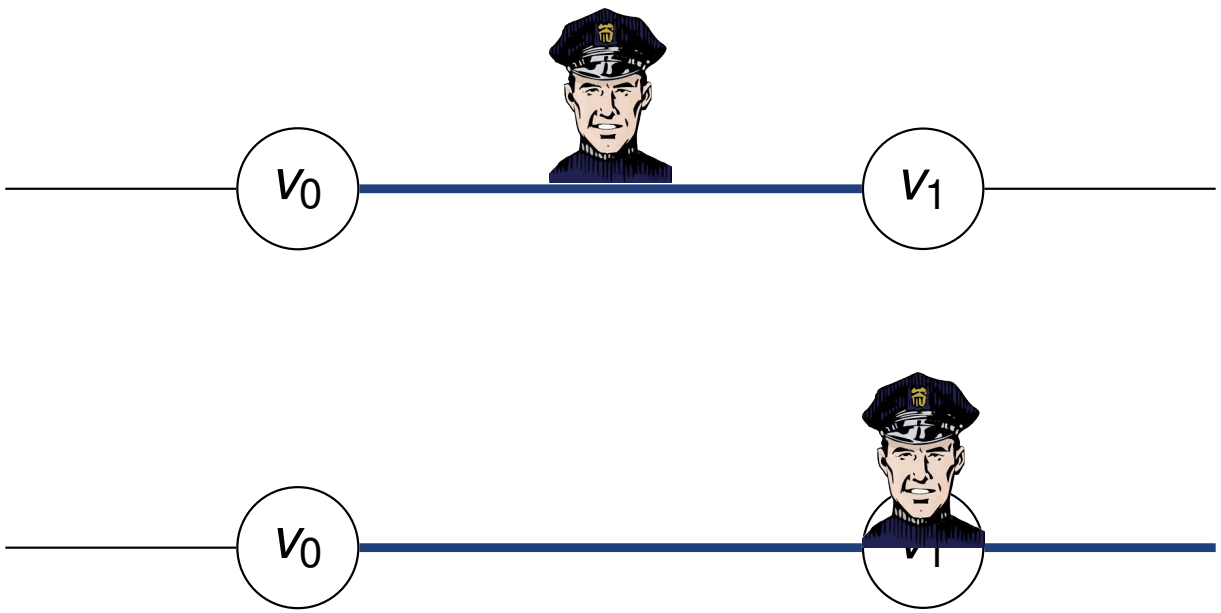
Cheese Patrol

- Surveillance of Tokyo
- Detectives can be placed on streets and intersections
- When on intersection, detectives surveils two streets



Cheese Patrol

- Surveillance of Tokyo
- Detectives can be placed on streets and intersections
- When on intersection, detectives surveils two streets



Surveil complete town with minimum number of detectives

Cheese Patrol

What is the best anyone can do?

Cheese Patrol

What is the best anyone can do?

- Graph G contains ℓ connected components, say $G_1, \ldots, G_\ell$

Cheese Patrol

What is the best anyone can do?

- Graph G contains ℓ connected components, say $G_1, \ldots, G_\ell$
- Detective can never watch streets from different components

Cheese Patrol

What is the best anyone can do?

- Graph G contains ℓ connected components, say $G_1, \ldots, G_\ell$
- Detective can never watch streets from different components
- G_1 has n_1 vertices and m_1 edges and so on

Cheese Patrol

What is the best anyone can do?

- Graph G contains ℓ connected components, say $G_1, \ldots, G_\ell$
- Detective can never watch streets from different components
- G_1 has n_1 vertices and m_1 edges and so on
- So consider G_1

Cheese Patrol

What is the best anyone can do?

- Graph G contains ℓ connected components, say $G_1, \ldots, G_\ell$
- Detective can never watch streets from different components
- G_1 has n_1 vertices and m_1 edges and so on
- So consider G_1

Suppose m_1 is even

Cheese Patrol

What is the best anyone can do?

- Graph G contains ℓ connected components, say $G_1, \ldots, G_\ell$
- Detective can never watch streets from different components
- G_1 has n_1 vertices and m_1 edges and so on
- So consider G_1

Suppose m_1 is even

- It is impossible to use less than $m_1/2$ detectives...

Cheese Patrol

What is the best anyone can do?

- Graph G contains ℓ connected components, say $G_1, \ldots, G_\ell$
- Detective can never watch streets from different components
- G_1 has n_1 vertices and m_1 edges and so on
- So consider G_1

Suppose m_1 is even

- It is impossible to use less than $m_1/2$ detectives...
- because one detective can watch at most two edges

Cheese Patrol

What is the best anyone can do?

- Graph G contains ℓ connected components, say $G_1, \ldots, G_\ell$
- Detective can never watch streets from different components
- G_1 has n_1 vertices and m_1 edges and so on
- So consider G_1

Suppose m_1 is even

- It is impossible to use less than $m_1/2$ detectives...
- because one detective can watch at most two edges

Suppose m_1 is odd

Cheese Patrol

What is the best anyone can do?

- Graph G contains ℓ connected components, say $G_1, \ldots, G_\ell$
- Detective can never watch streets from different components
- G_1 has n_1 vertices and m_1 edges and so on
- So consider G_1

Suppose m_1 is even

- It is impossible to use less than $m_1/2$ detectives...
- because one detective can watch at most two edges

Suppose m_1 is odd

- Then we need $(m_1 + 1)/2$ detectives

Cheese Patrol

What is the best anyone can do?

- Graph G contains ℓ connected components, say $G_1, \ldots, G_\ell$
- Detective can never watch streets from different components
- G_1 has n_1 vertices and m_1 edges and so on
- So consider G_1

Suppose m_1 is even

- It is impossible to use less than $m_1/2$ detectives...
- because one detective can watch at most two edges

Suppose m_1 is odd

- Then we need $(m_1 + 1)/2$ detectives

- $m_1/2 = \lceil m_1/2 \rceil$ for m_1 even
- $(m_1 + 1)/2 = \lceil m_1/2 \rceil$ for m_1 odd

Cheese Patrol

What is the best anyone can do?

- Graph G contains ℓ connected components, say $G_1, \ldots, G_\ell$
- Detective can never watch streets from different components
- G_1 has n_1 vertices and m_1 edges and so on
- So consider G_1

Suppose m_1 is even

- It is impossible to use less than $m_1/2$ detectives...
- because one detective can watch at most two edges

Suppose m_1 is odd

- Then we need $(m_1 + 1)/2$ detectives

- $m_1/2 = \lceil m_1/2 \rceil$ for m_1 even
- $(m_1 + 1)/2 = \lceil m_1/2 \rceil$ for m_1 odd

$$\sum_{i=1}^{\ell} \lceil m_i/2 \rceil \text{ detectives}$$

Cheese Patrol

Observations

- It is never superior to place detective at street

Cheese Patrol

Observations

- It is never superior to place detective at street
- It can always be moved to intersection

Cheese Patrol

Observations

- It is never superior to place detective at street
- It can always be moved to intersection

⇨ This does not make solution worse

Cheese Patrol

Observations

- It is never superior to place detective at street
- It can always be moved to intersection

⇨ This does not make solution worse

- No street needs to be watched by two detectives

Cheese Patrol

Observations

- It is never superior to place detective at street
- It can always be moved to intersection

⇨ This does not make solution worse

- No street needs to be watched by two detectives

Consequences

- Only place detectives at intersections

Cheese Patrol

Observations

- It is never superior to place detective at street
- It can always be moved to intersection

⇨ This does not make solution worse

- No street needs to be watched by two detectives

Consequences

- Only place detectives at intersections
- Have each detective watch two streets

Cheese Patrol

Observations

- It is never superior to place detective at street
- It can always be moved to intersection

⇨ This does not make solution worse

- No street needs to be watched by two detectives

Consequences

- Only place detectives at intersections
- Have each detective watch two streets
- In each component, there may be one exception

Cheese Patrol

Assume graph is a tree

Cheese Patrol

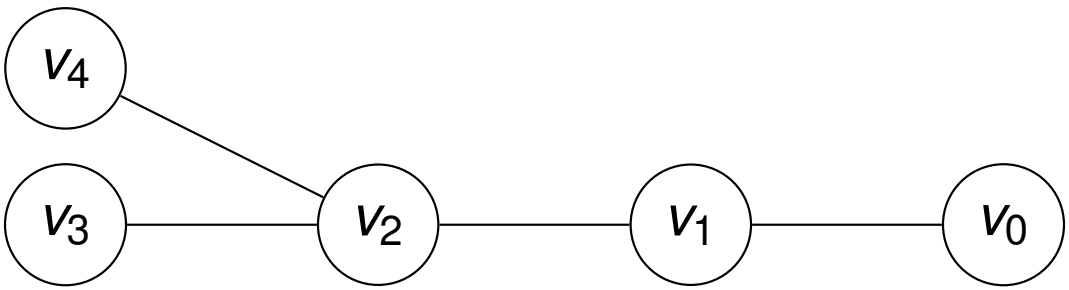
Assume graph is a tree

Careful about leaves

Cheese Patrol

Assume graph is a tree

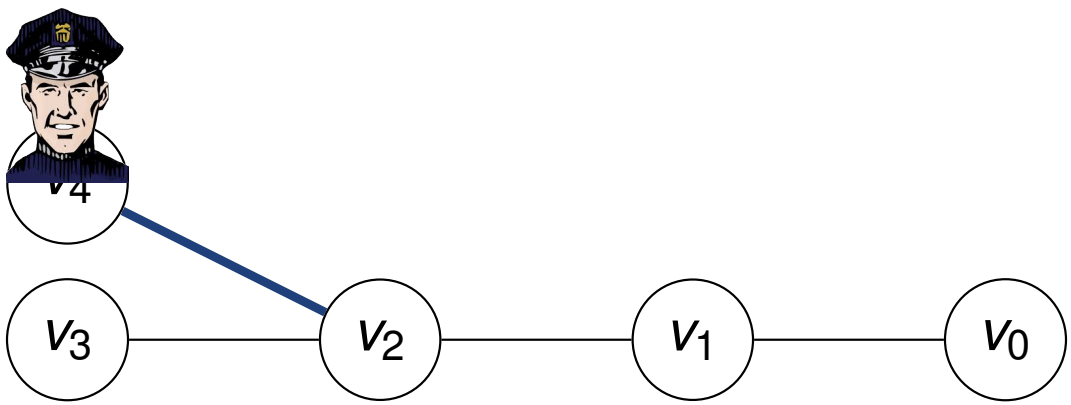
Careful about leaves



Cheese Patrol

Assume graph is a tree

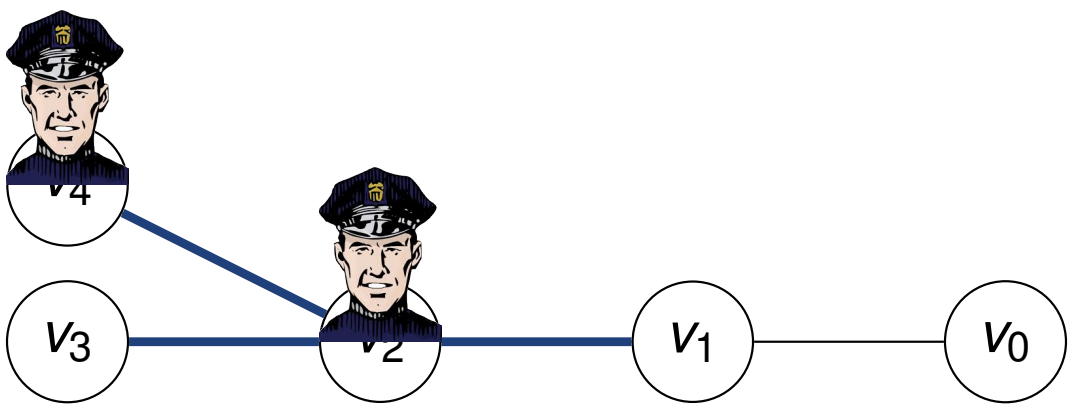
Careful about leaves



Cheese Patrol

Assume graph is a tree

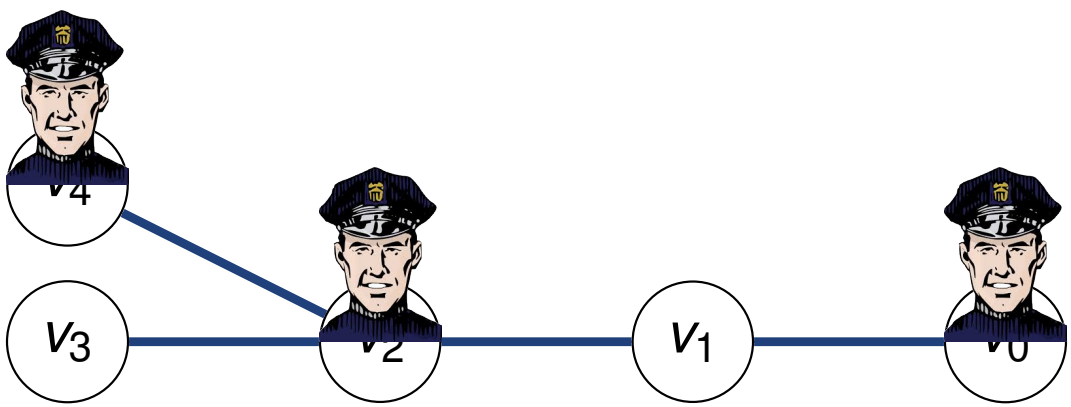
Careful about leaves



Cheese Patrol

Assume graph is a tree

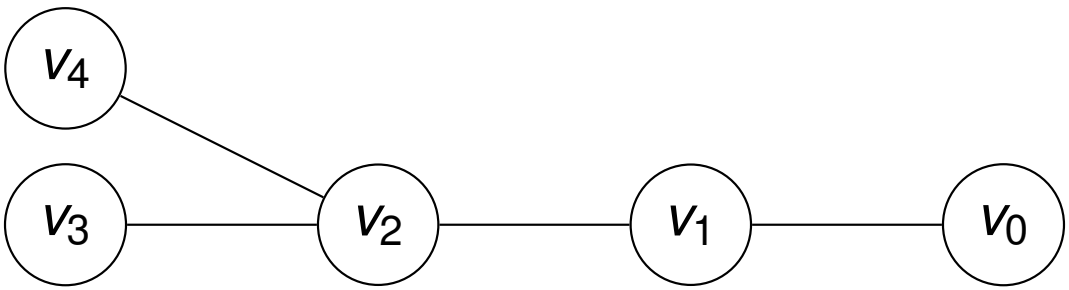
Careful about leaves



Cheese Patrol

Assume graph is a tree

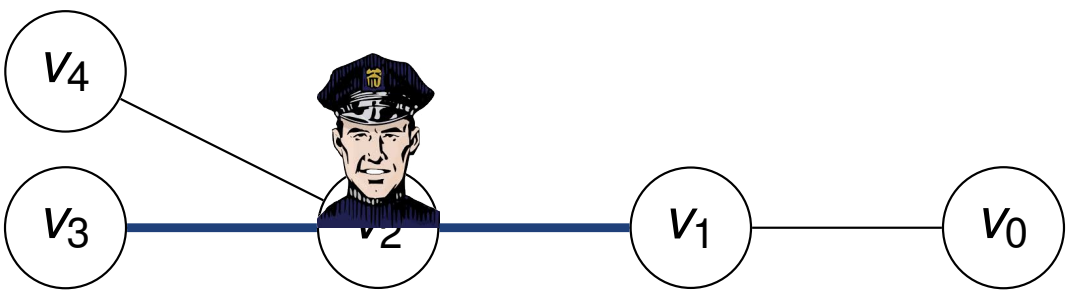
Careful about leaves



Cheese Patrol

Assume graph is a tree

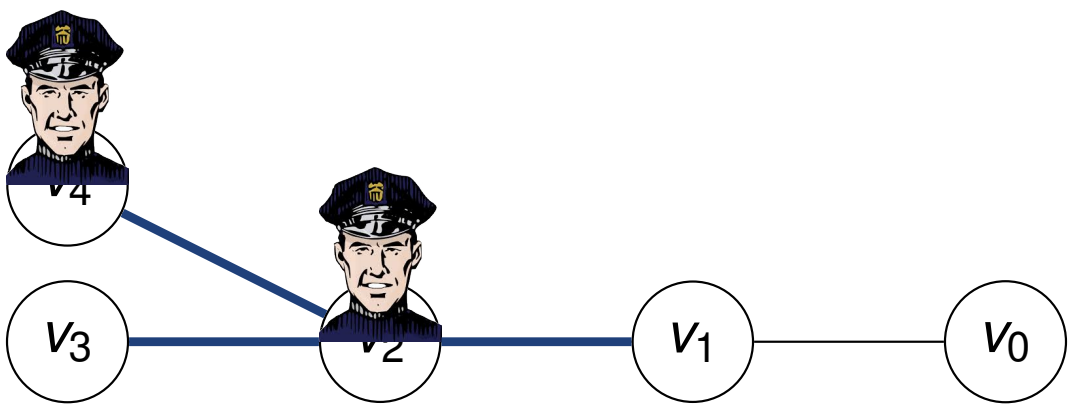
Careful about leaves



Cheese Patrol

Assume graph is a tree

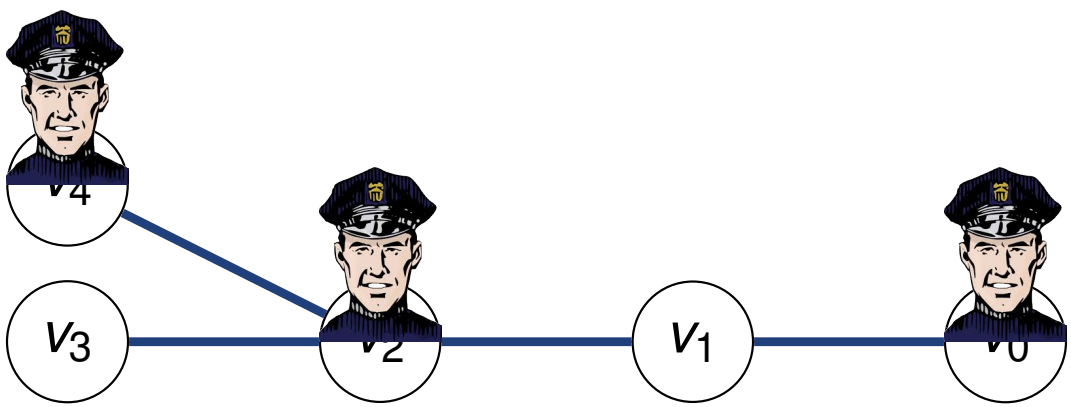
Careful about leaves



Cheese Patrol

Assume graph is a tree

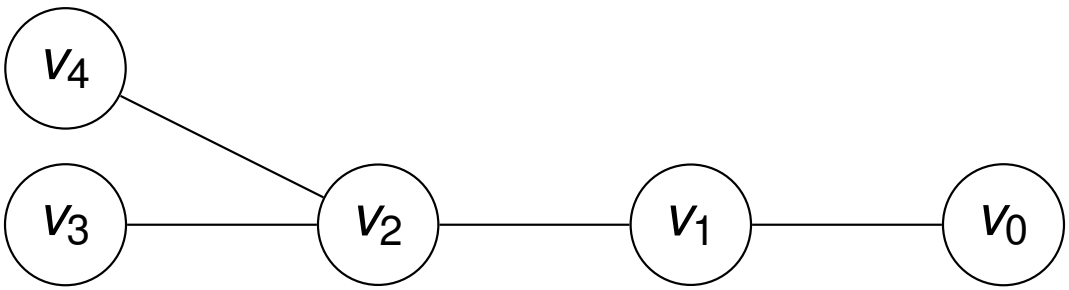
Careful about leaves



Cheese Patrol

Assume graph is a tree

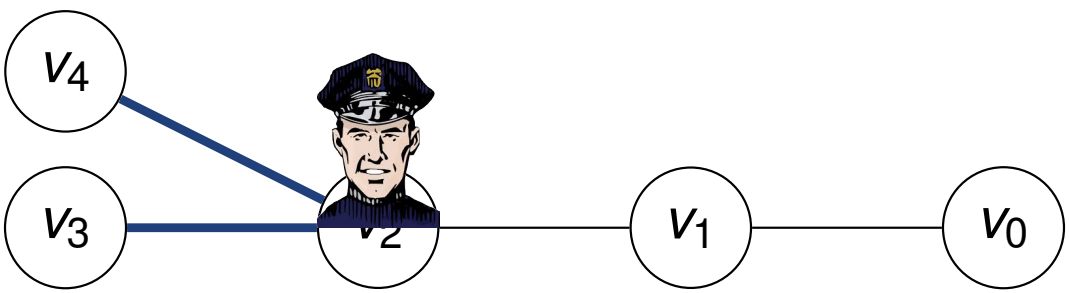
Careful about leaves



Cheese Patrol

Assume graph is a tree

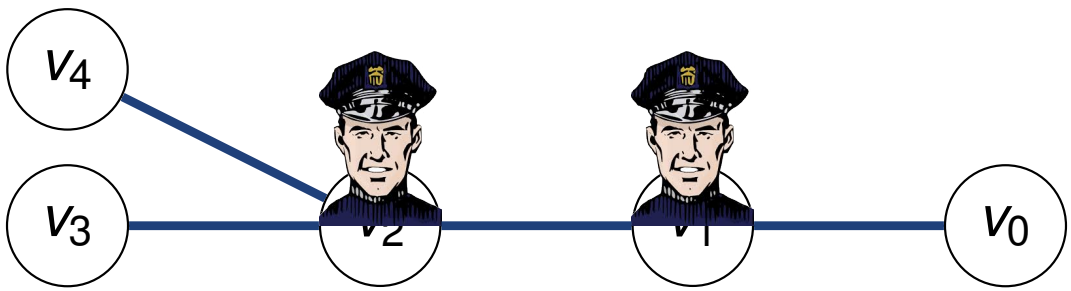
Careful about leaves



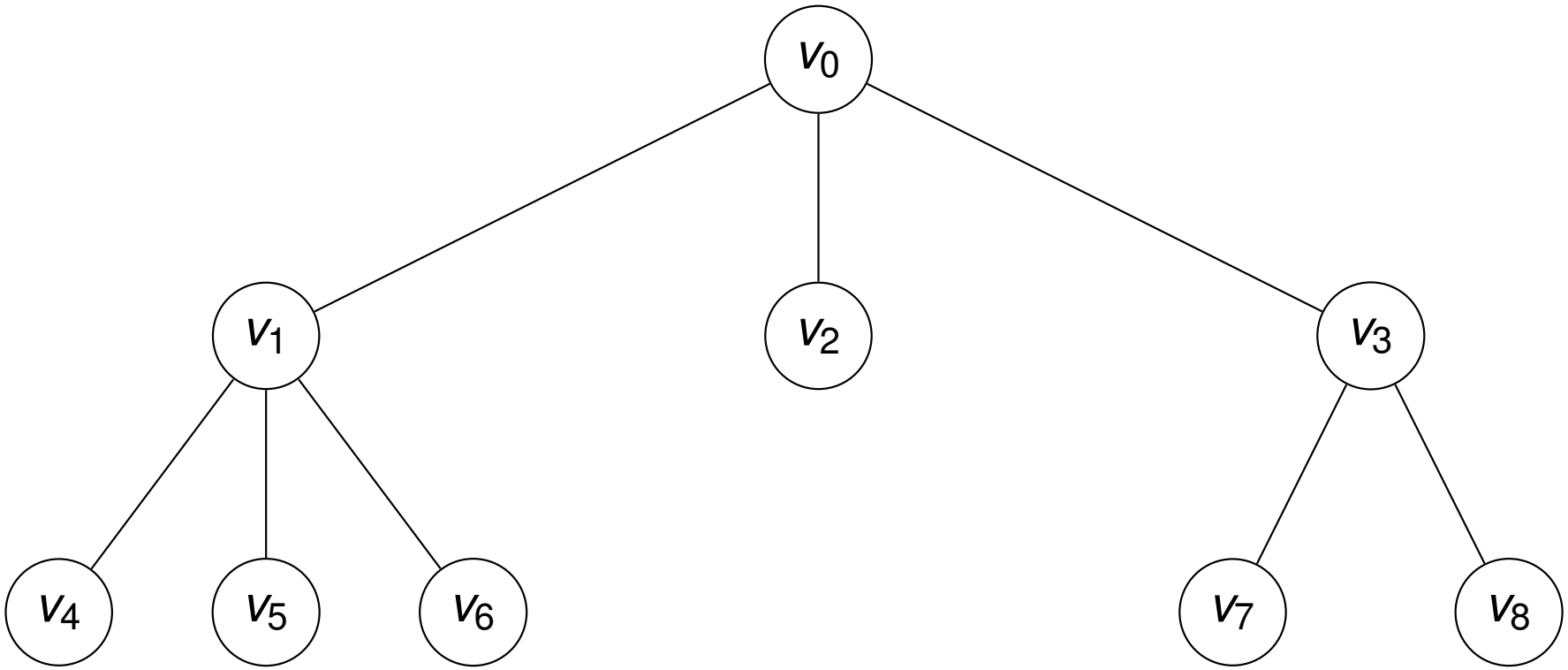
Cheese Patrol

Assume graph is a tree

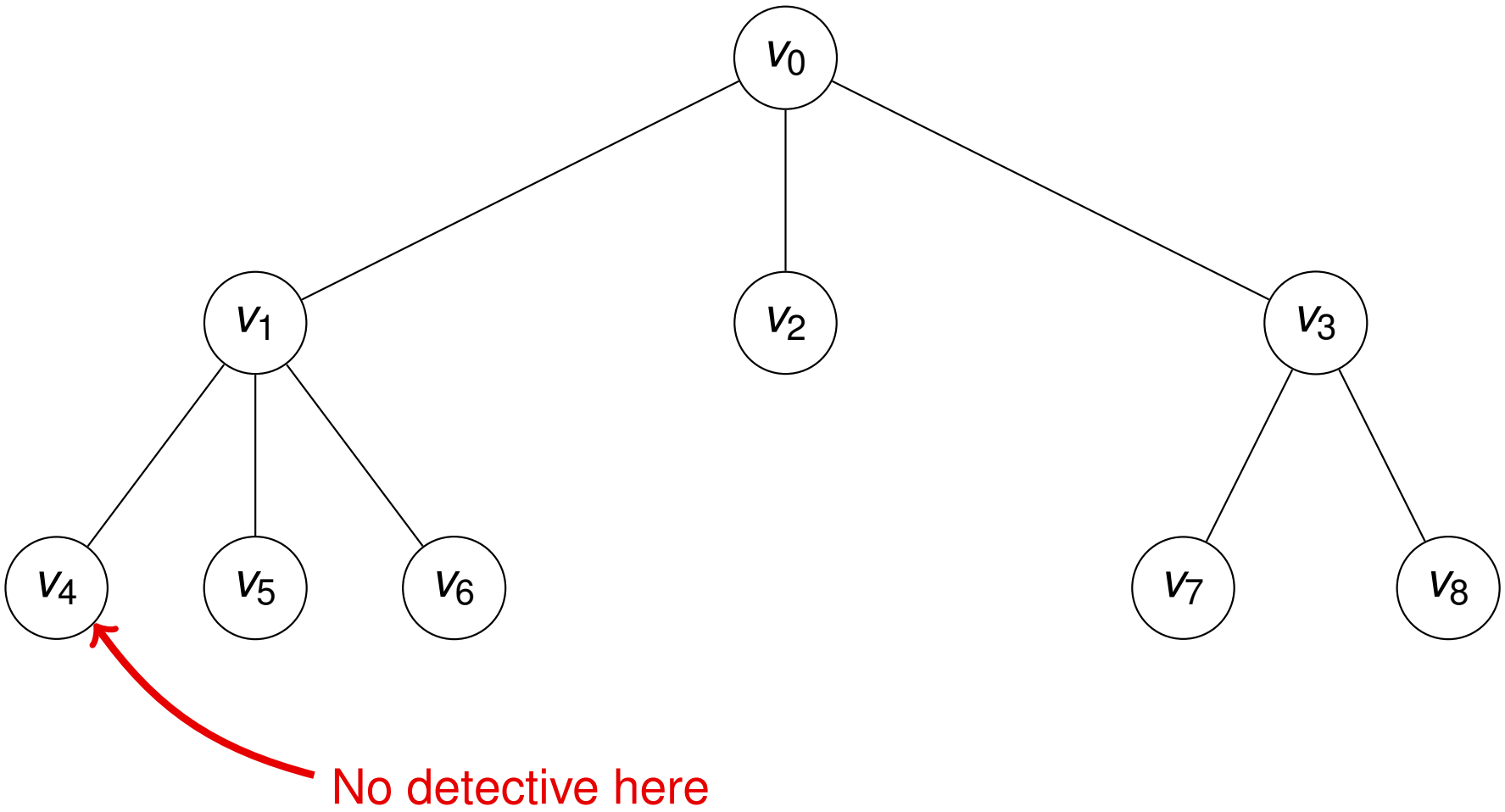
Careful about leaves



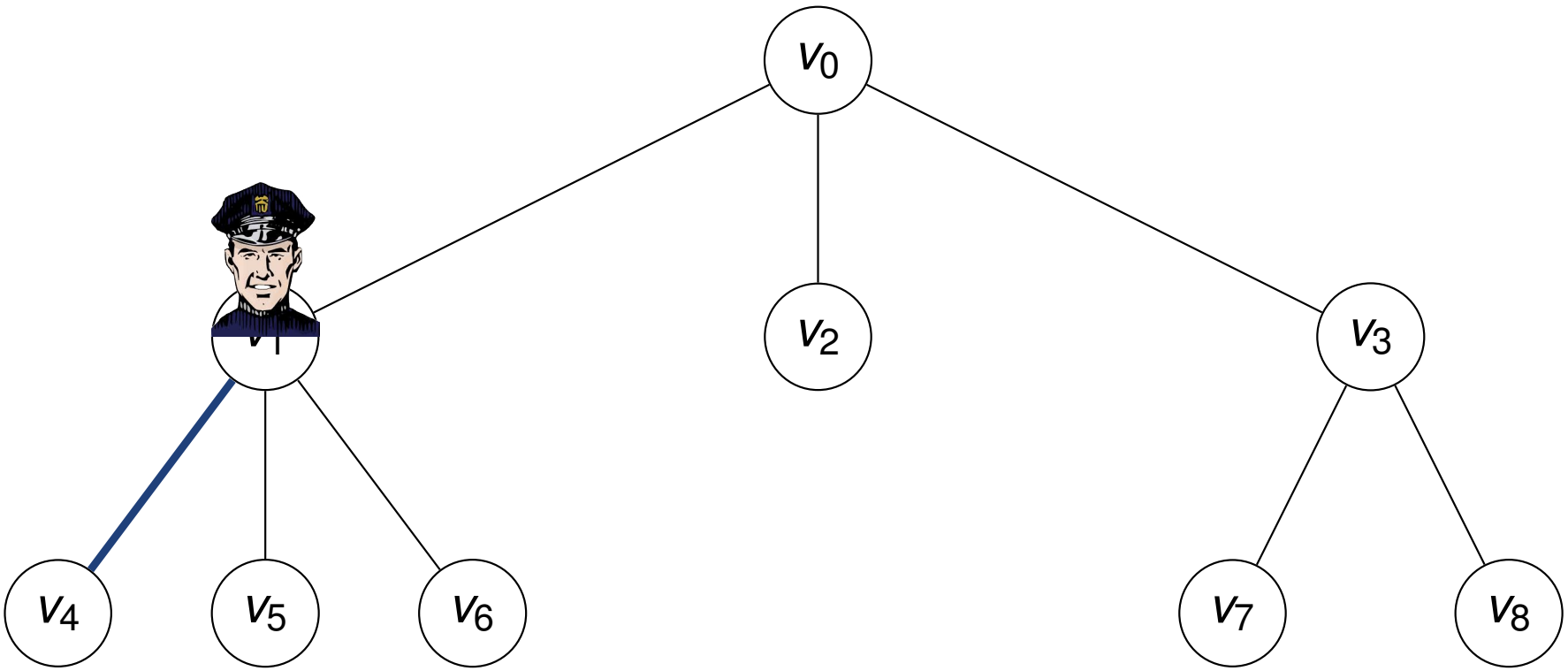
Cheese Patrol



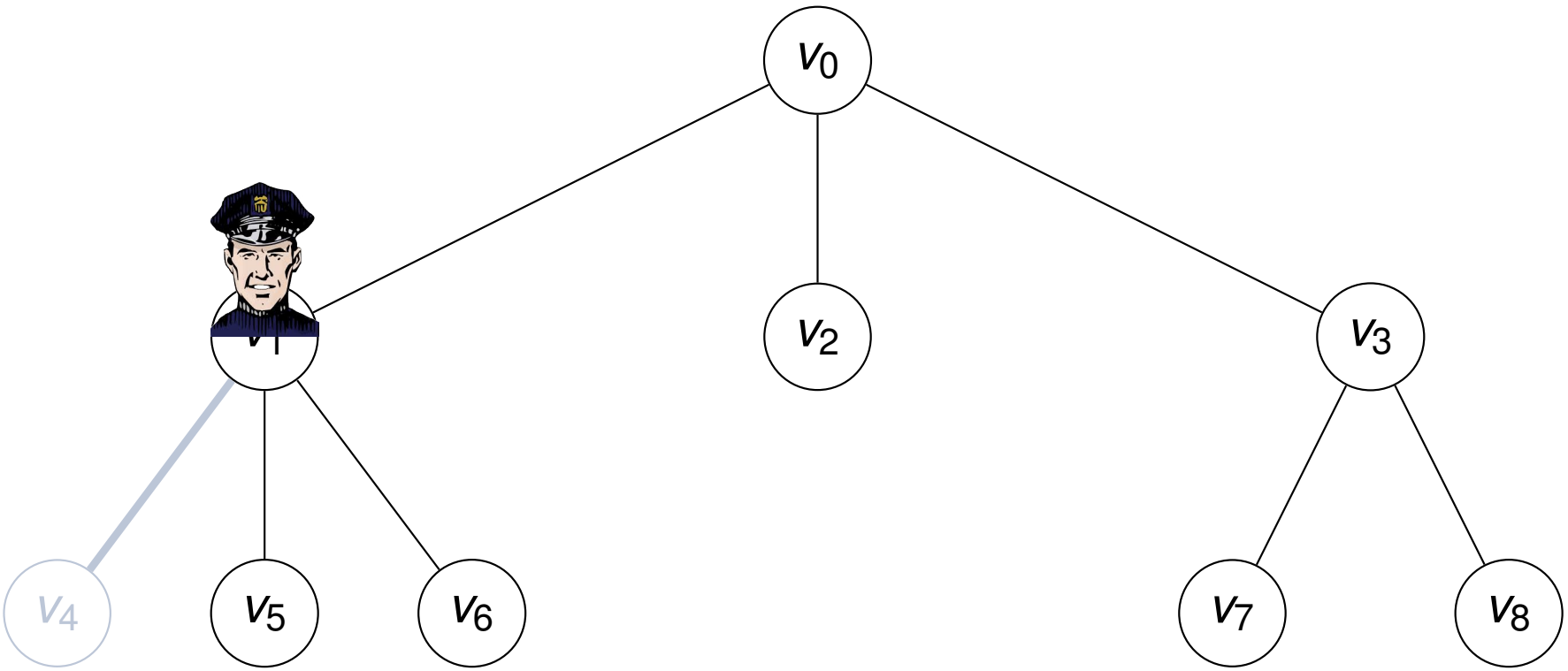
Cheese Patrol



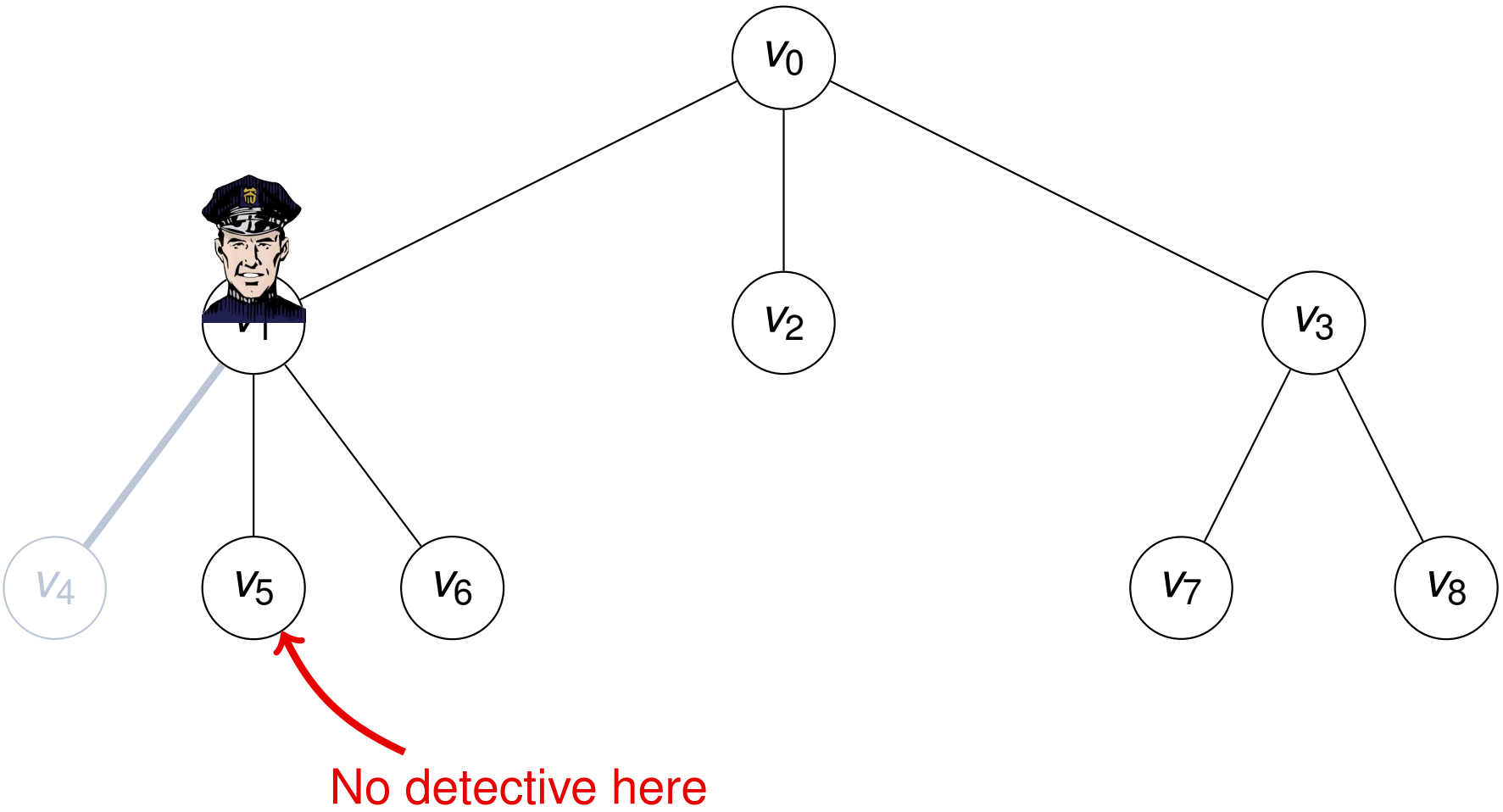
Cheese Patrol



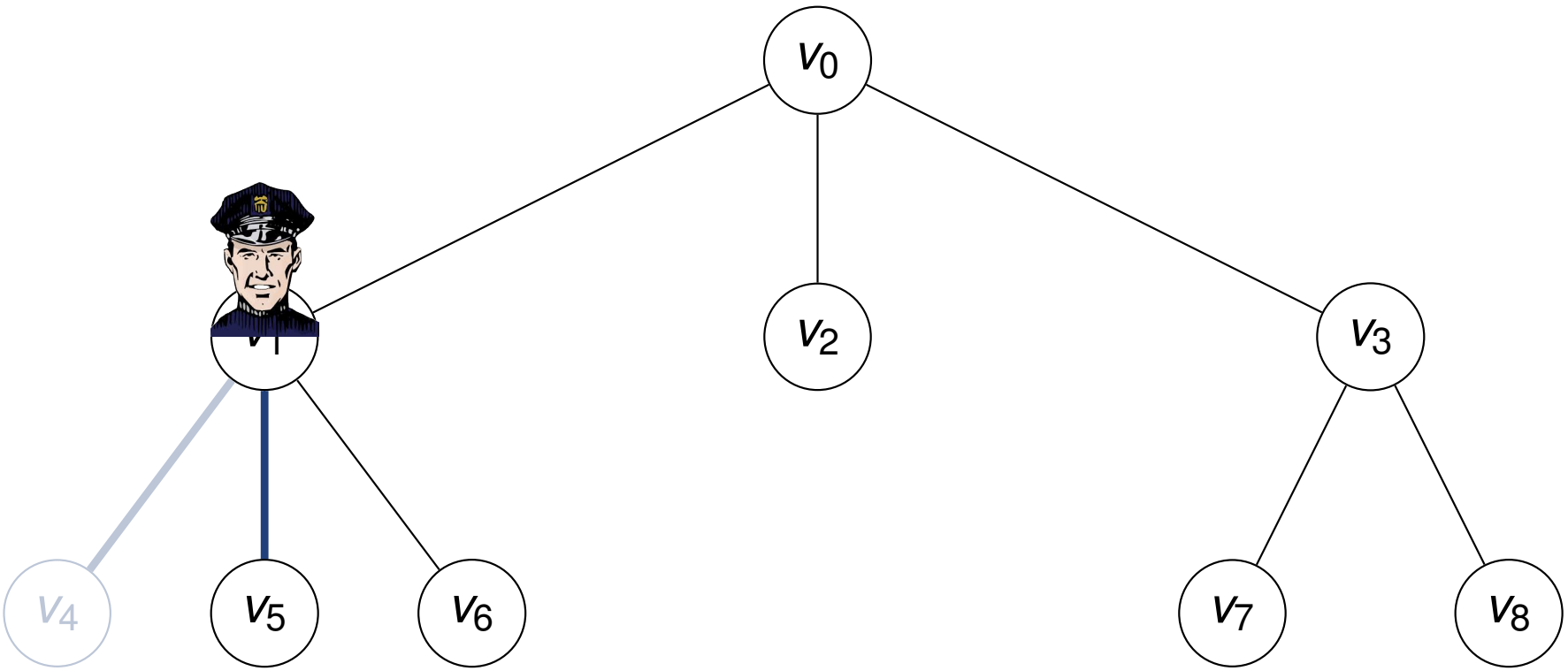
Cheese Patrol



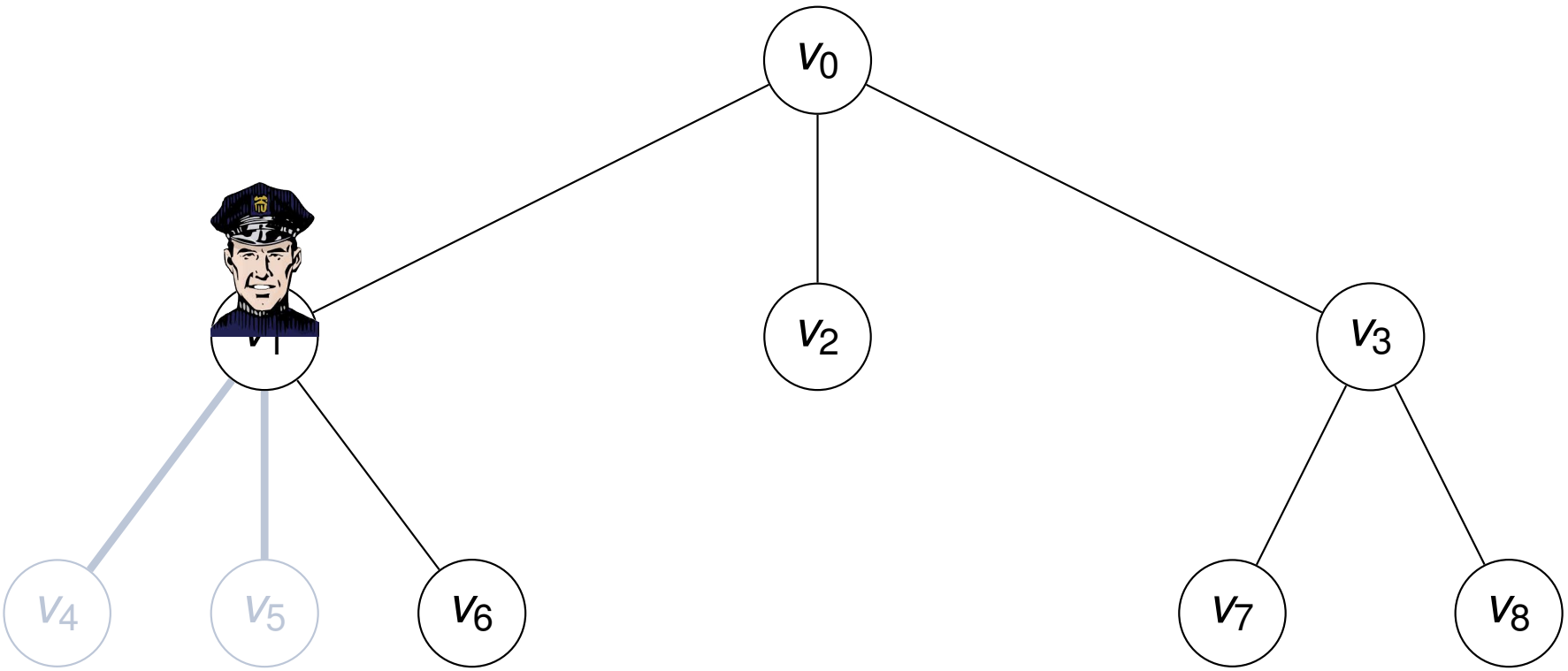
Cheese Patrol



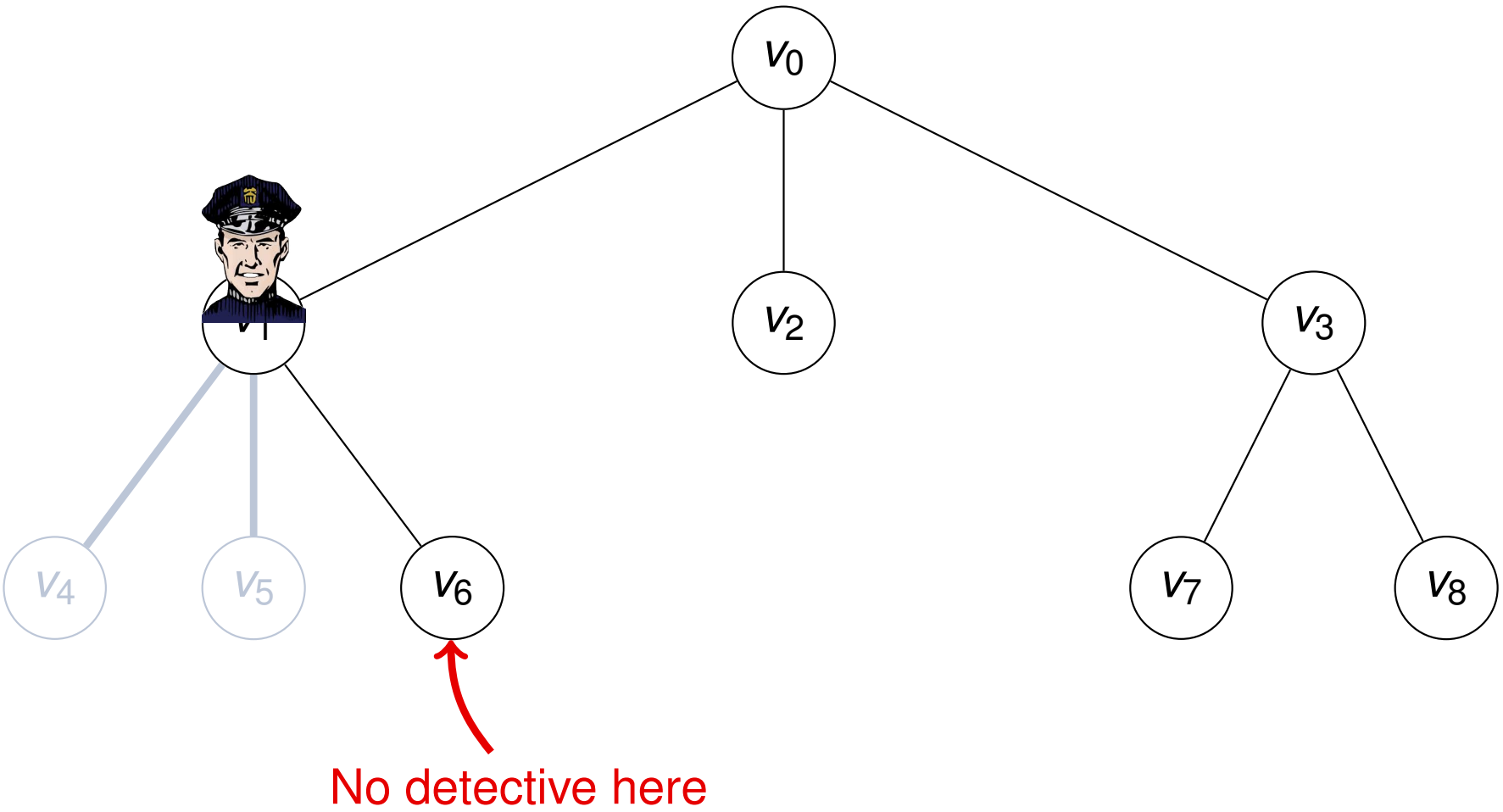
Cheese Patrol



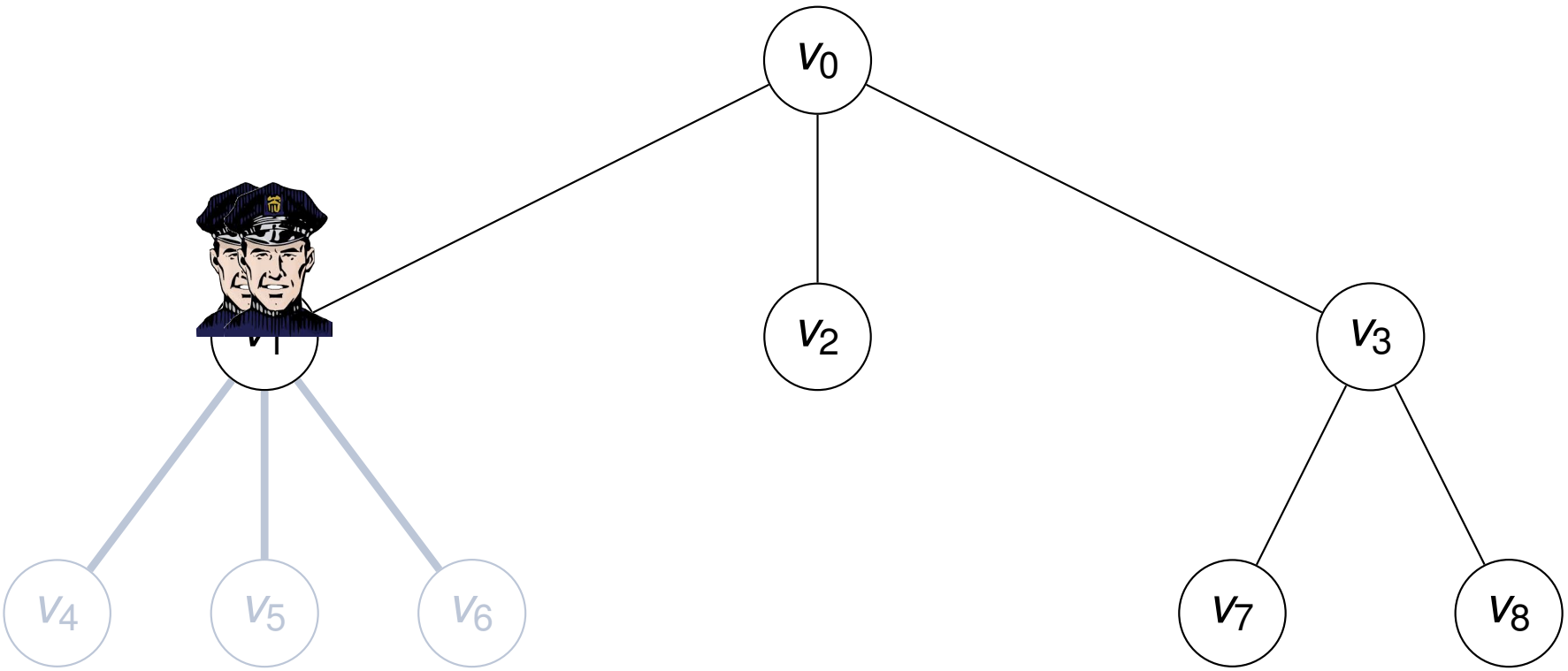
Cheese Patrol



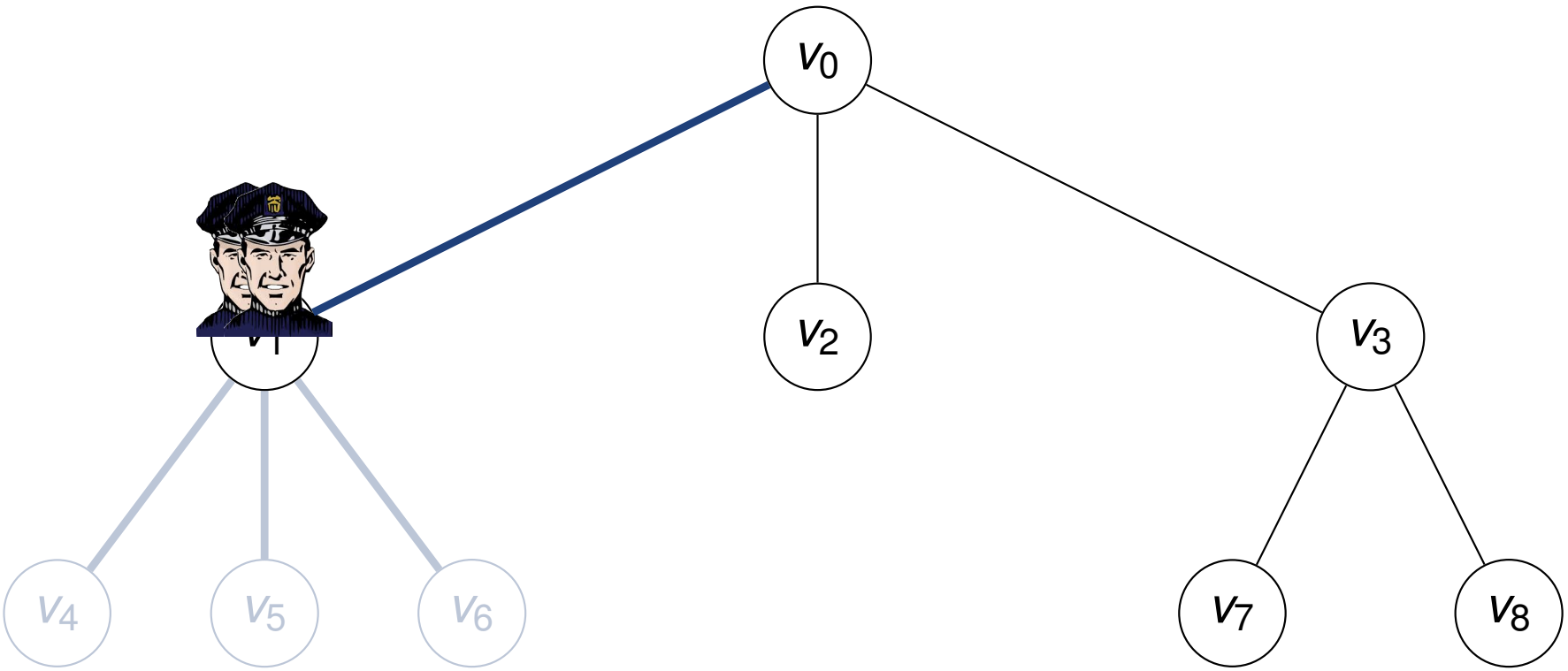
Cheese Patrol



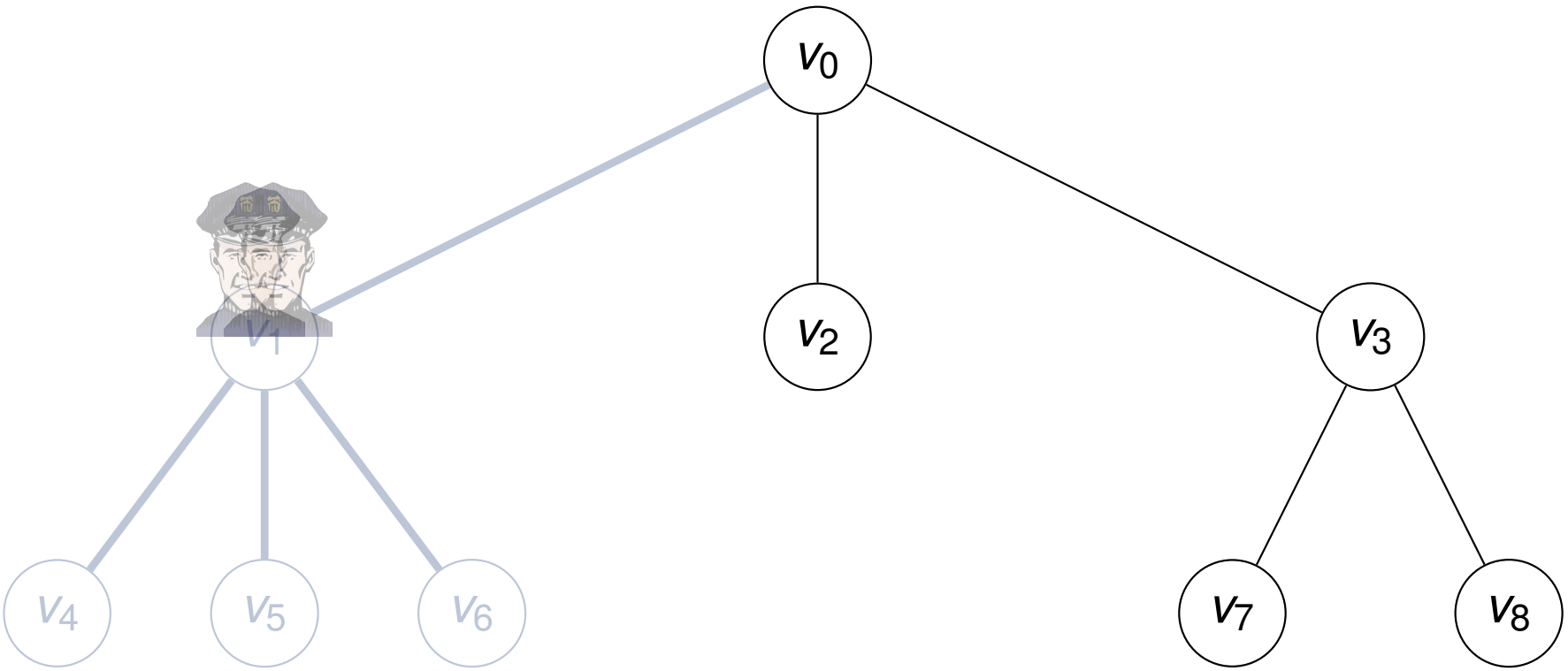
Cheese Patrol



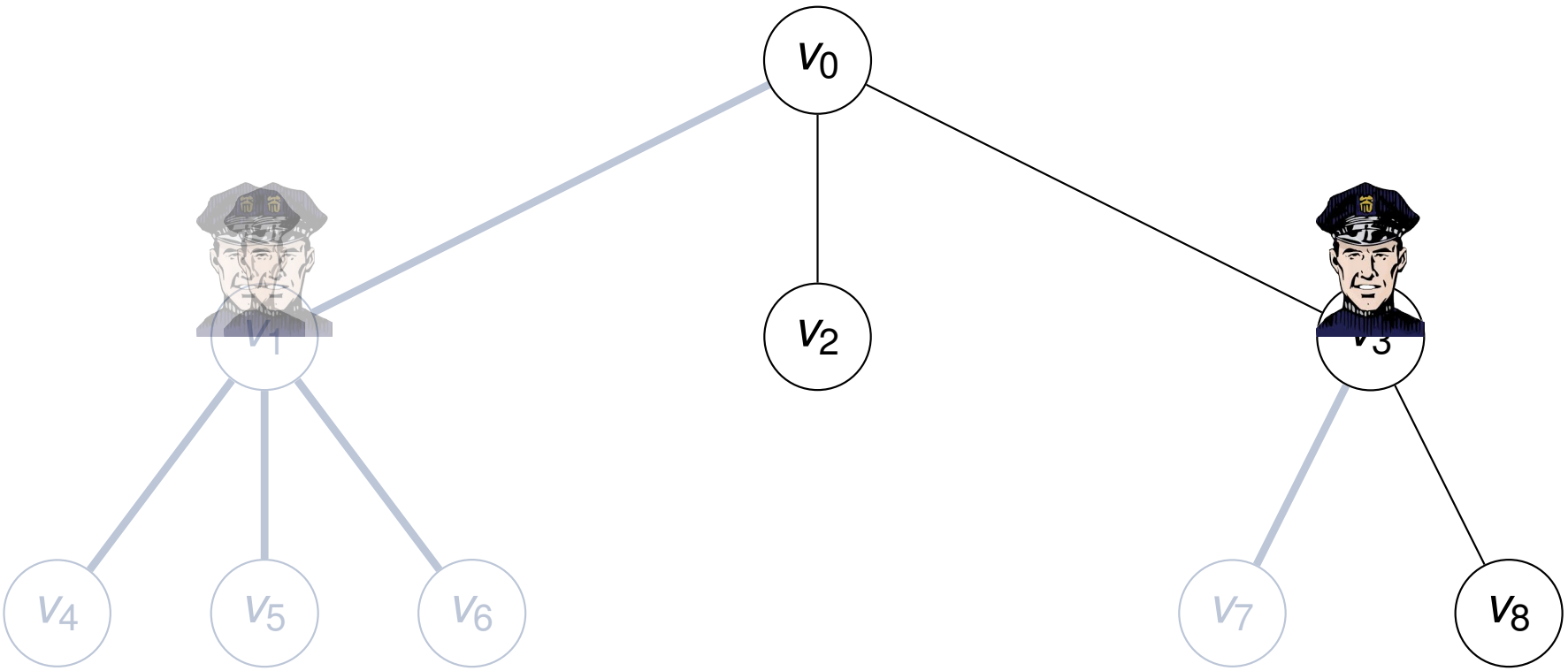
Cheese Patrol



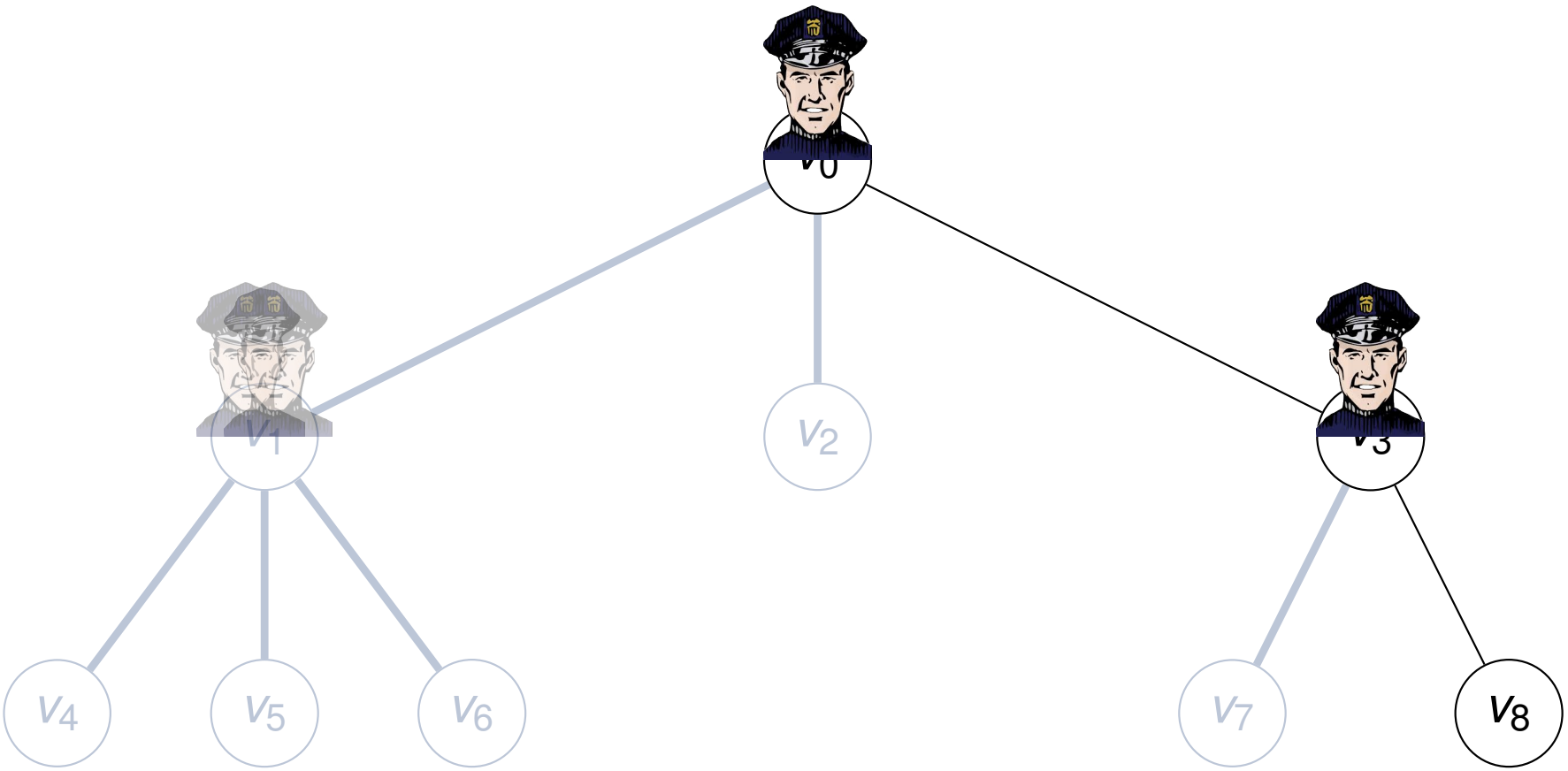
Cheese Patrol



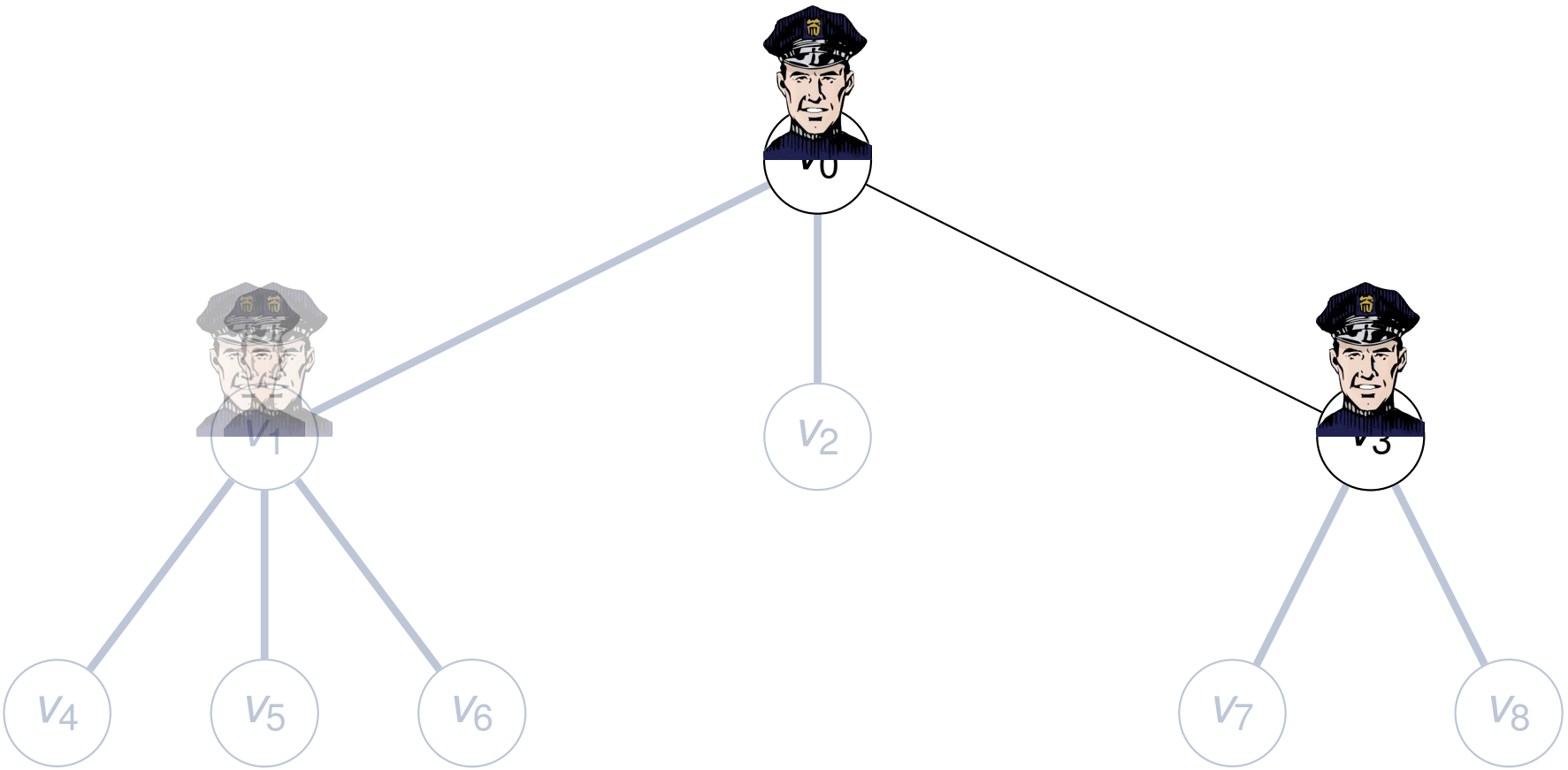
Cheese Patrol



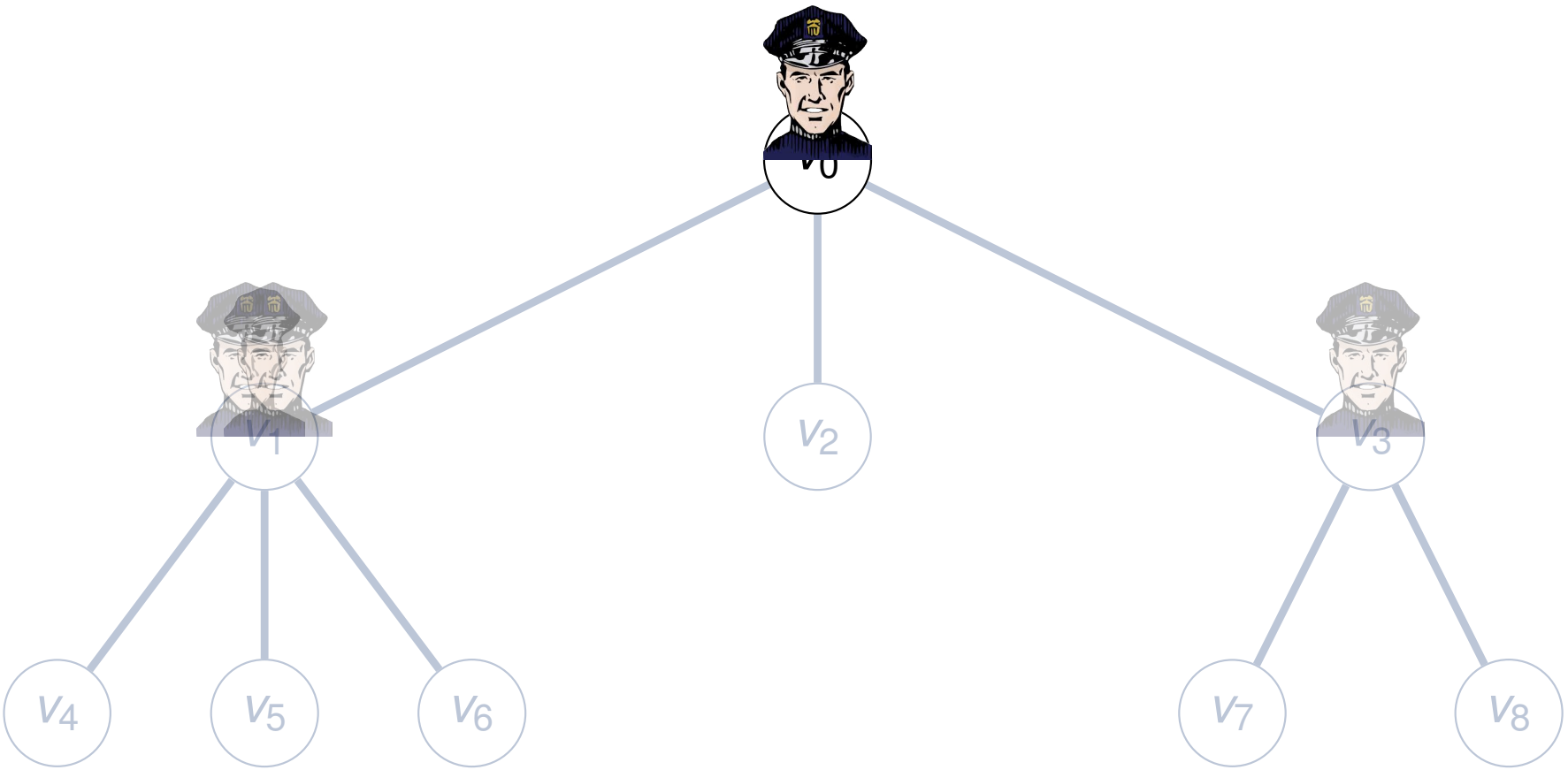
Cheese Patrol



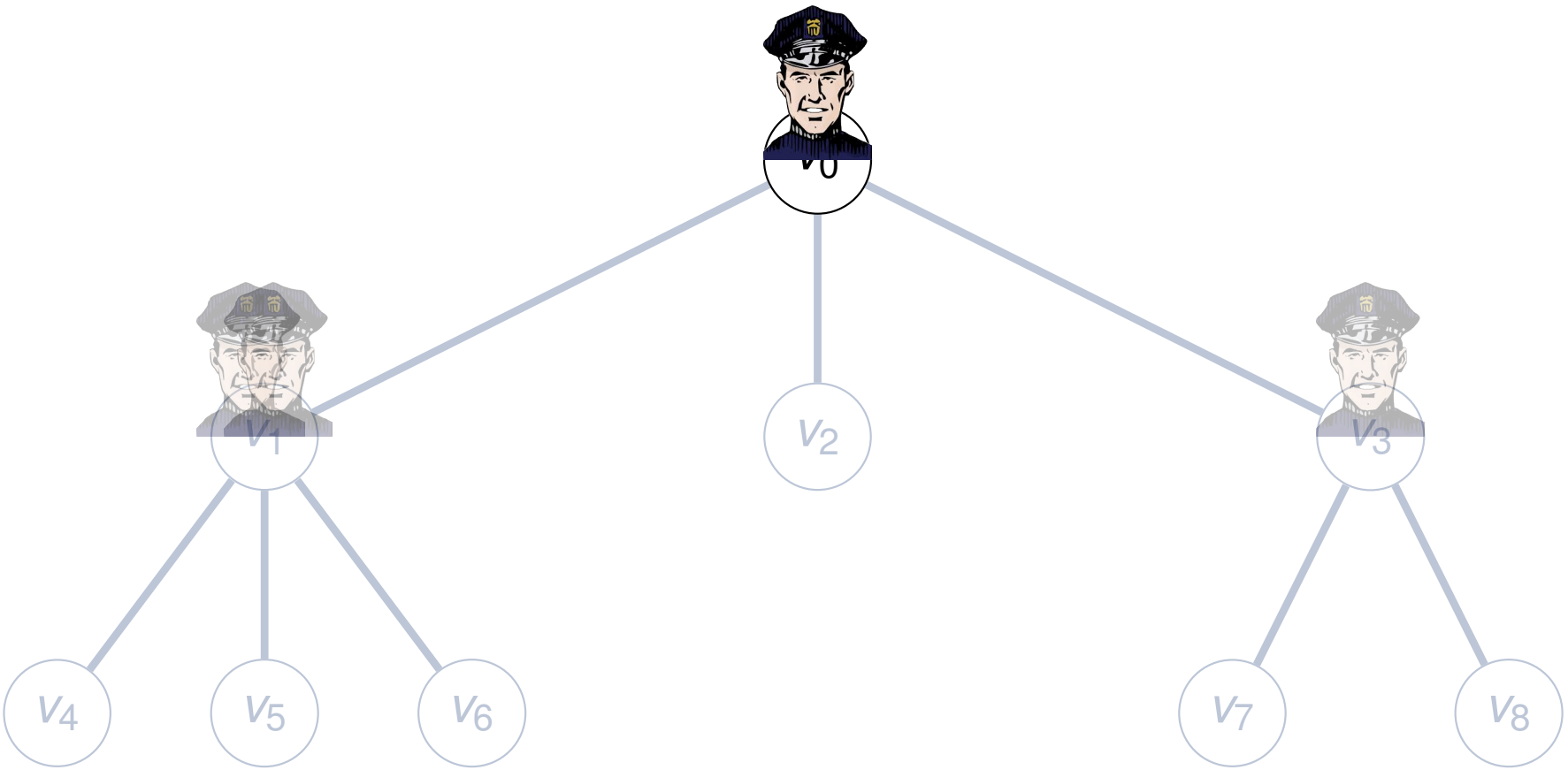
Cheese Patrol



Cheese Patrol



Cheese Patrol



Result: 8 edges, 4 detectives

Cheese Patrol

When we are done with a subtree

Cheese Patrol

When we are done with a subtree

- We used an optimal number of detectives

Cheese Patrol

When we are done with a subtree

- We used an optimal number of detectives
- If subtree has even number of edges, half as many detectives

Cheese Patrol

When we are done with a subtree

- We used an optimal number of detectives
- If subtree has even number of edges, half as many detectives
- If odd number of edges, one det. has capacity for one edge

Cheese Patrol

When we are done with a subtree

- We used an optimal number of detectives
- If subtree has even number of edges, half as many detectives
- If odd number of edges, one det. has capacity for one edge

⇨ Used to watch edge to parent

Cheese Patrol

When we are done with a subtree

- We used an optimal number of detectives
- If subtree has even number of edges, half as many detectives
- If odd number of edges, one det. has capacity for one edge

⇨ Used to watch edge to parent

- Unless parent is root

Cheese Patrol

Now assume graph is arbitrary

Cheese Patrol

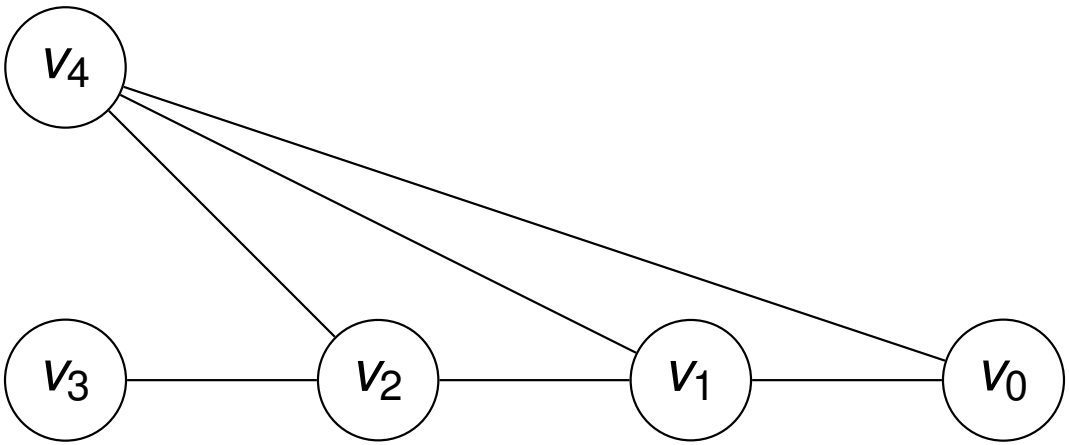
Now assume graph is arbitrary

Somewhat the same problem

Cheese Patrol

Now assume graph is arbitrary

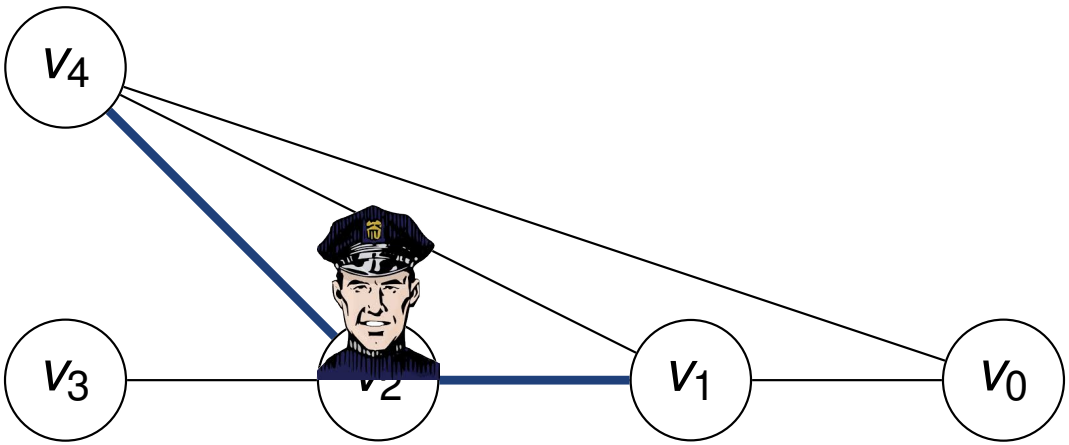
Somewhat the same problem



Cheese Patrol

Now assume graph is arbitrary

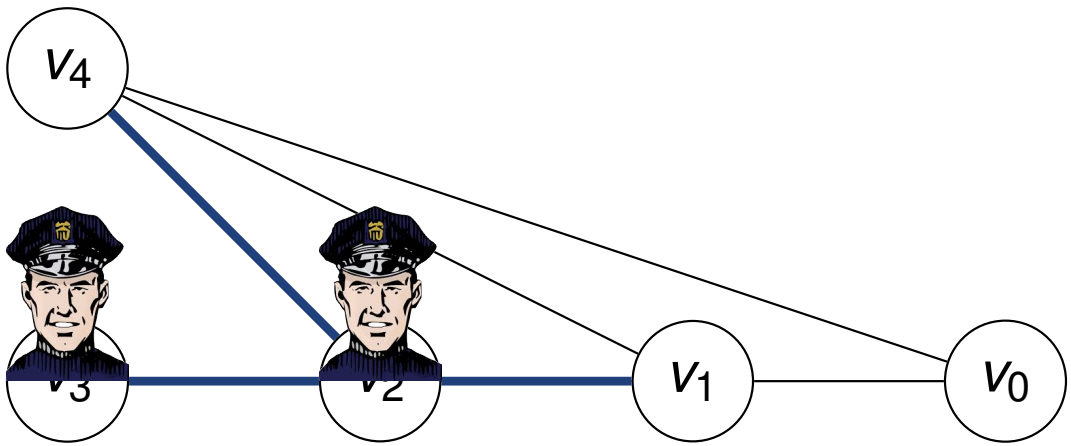
Somewhat the same problem



Cheese Patrol

Now assume graph is arbitrary

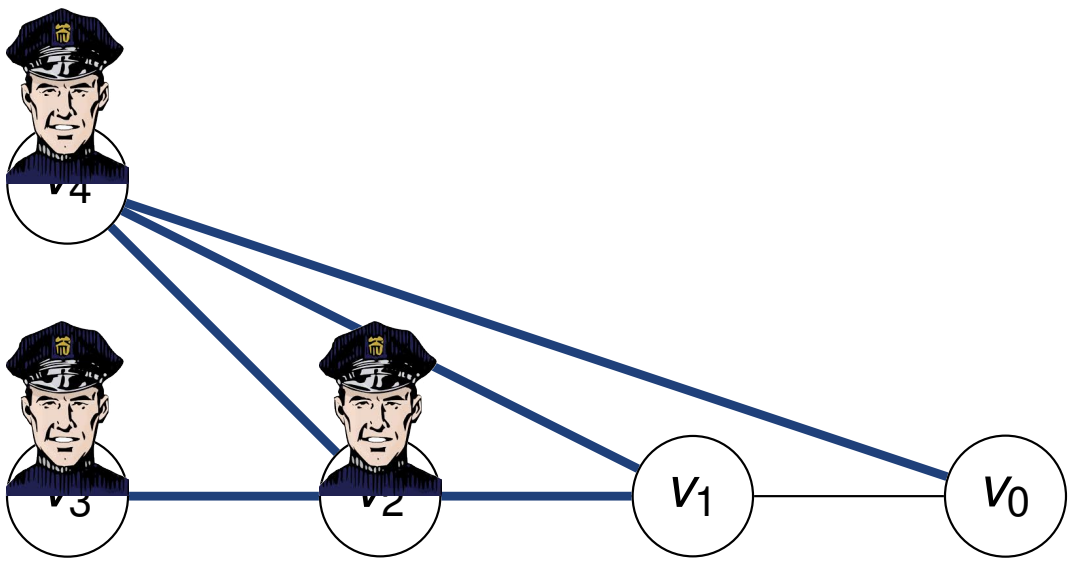
Somewhat the same problem



Cheese Patrol

Now assume graph is arbitrary

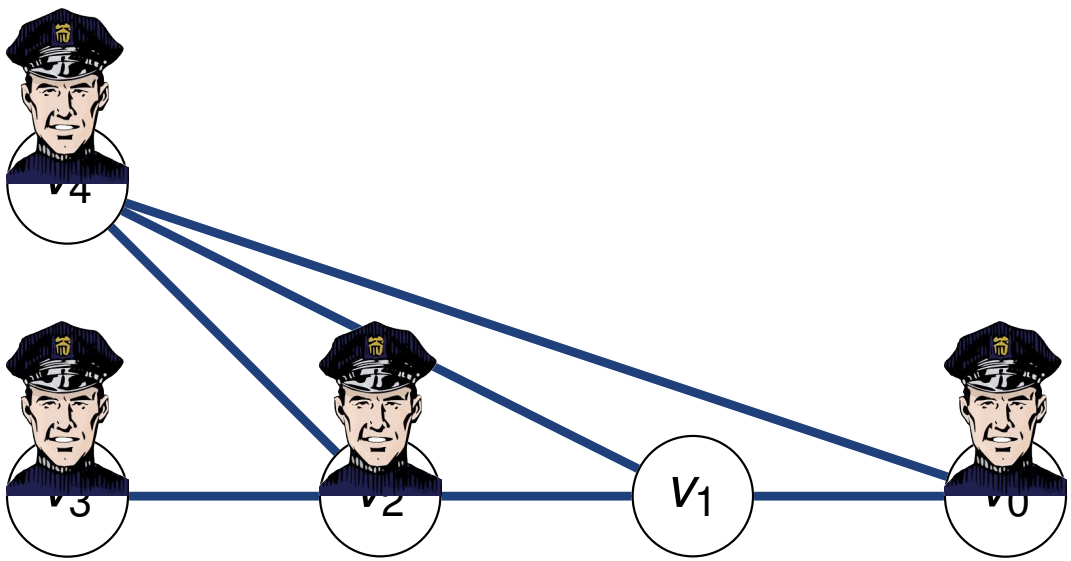
Somewhat the same problem



Cheese Patrol

Now assume graph is arbitrary

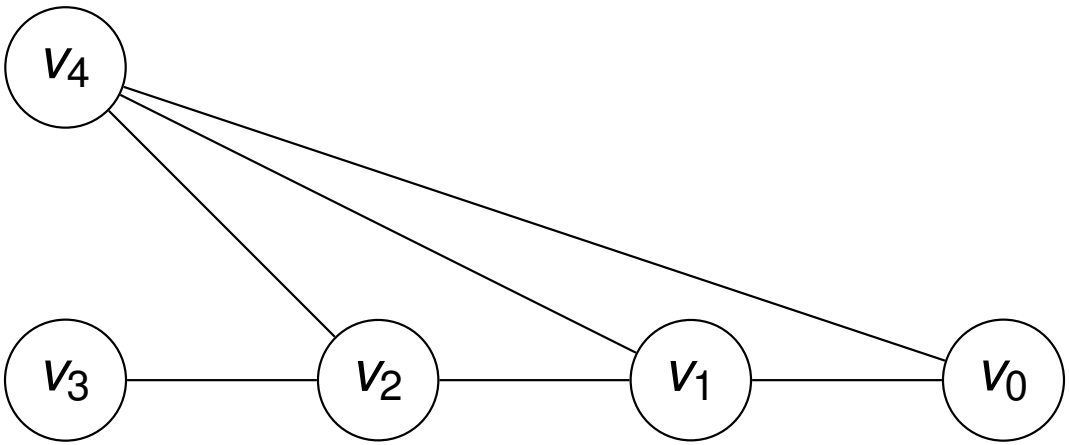
Somewhat the same problem



Cheese Patrol

Now assume graph is arbitrary

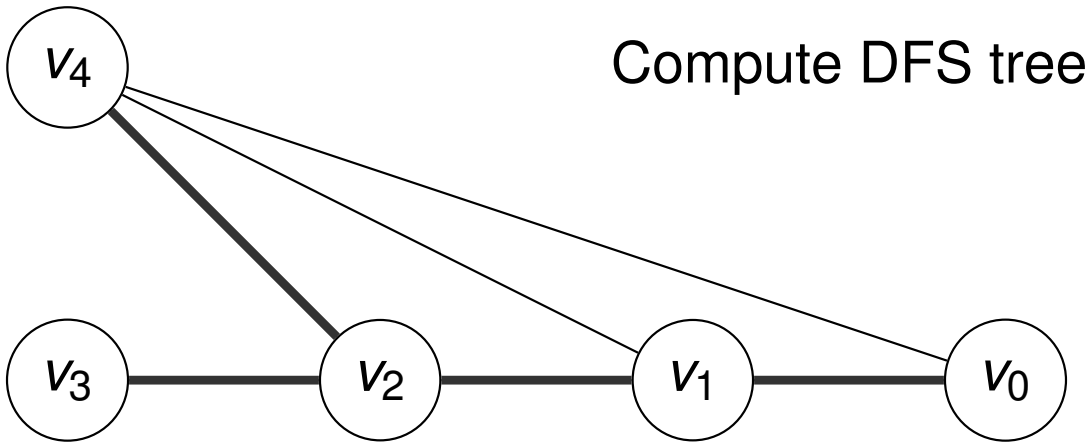
Somewhat the same problem



Cheese Patrol

Now assume graph is arbitrary

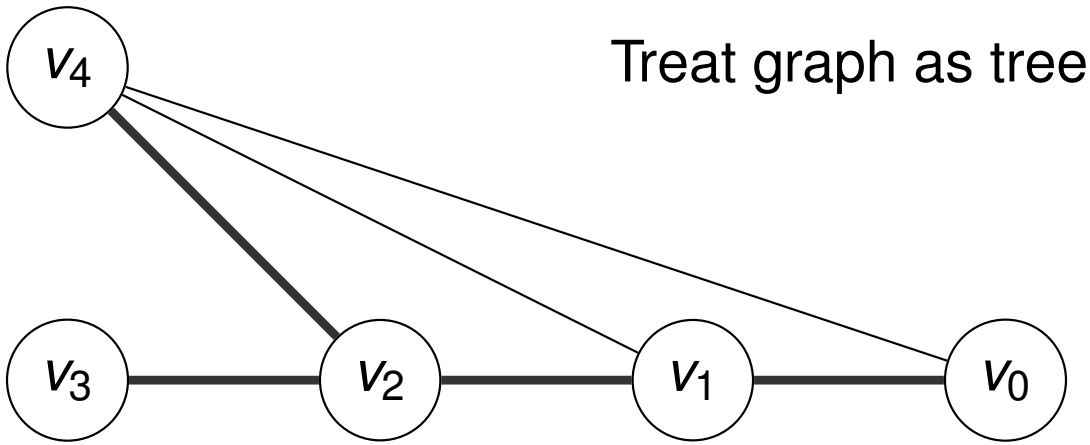
Use DFS to avoid this problem



Cheese Patrol

Now assume graph is arbitrary

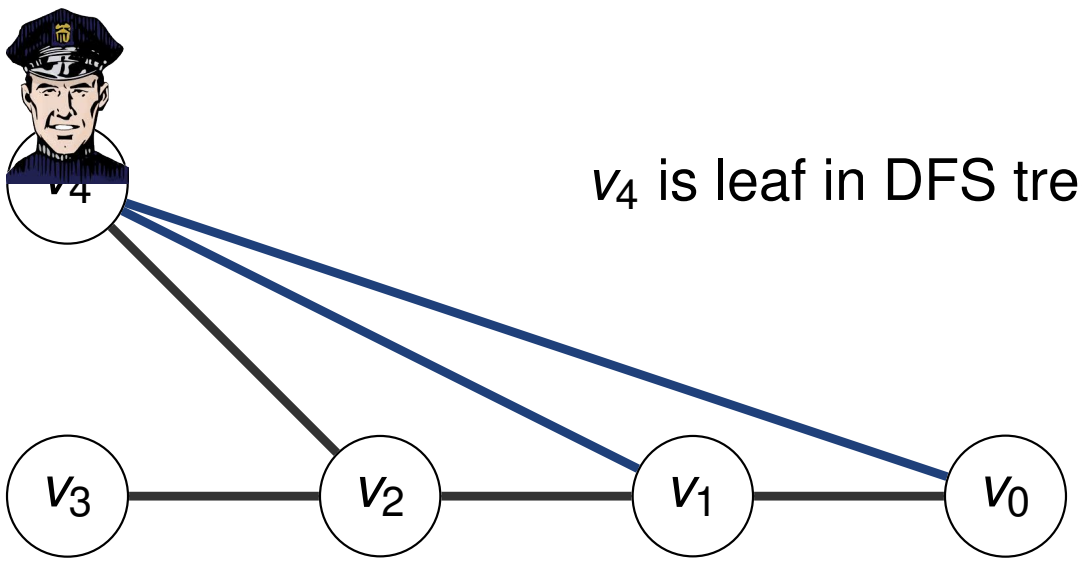
Use DFS to avoid this problem



Cheese Patrol

Now assume graph is arbitrary

Use DFS to avoid this problem

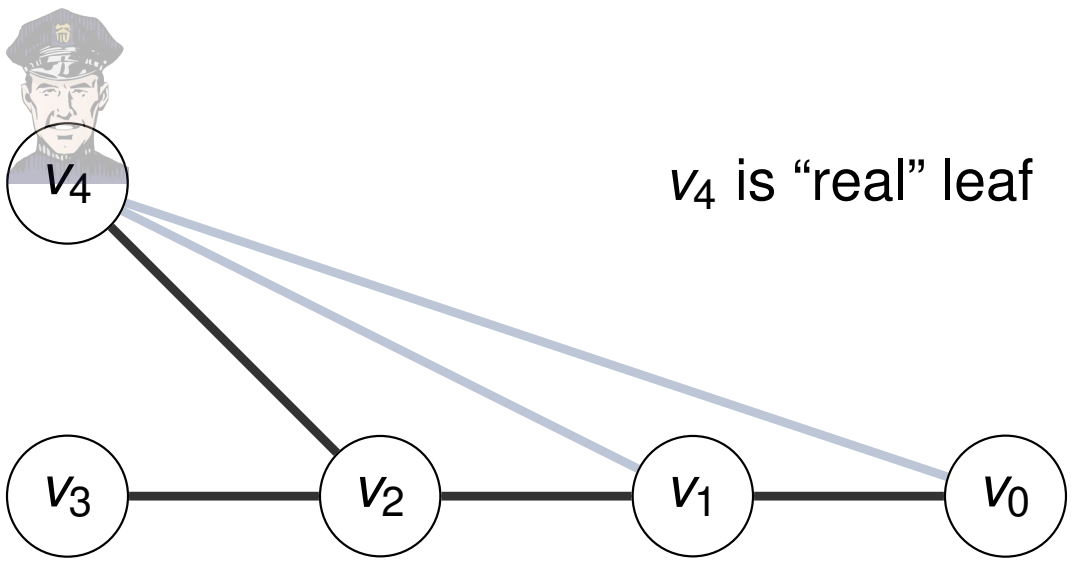


v_4 is leaf in DFS tree

Cheese Patrol

Now assume graph is arbitrary

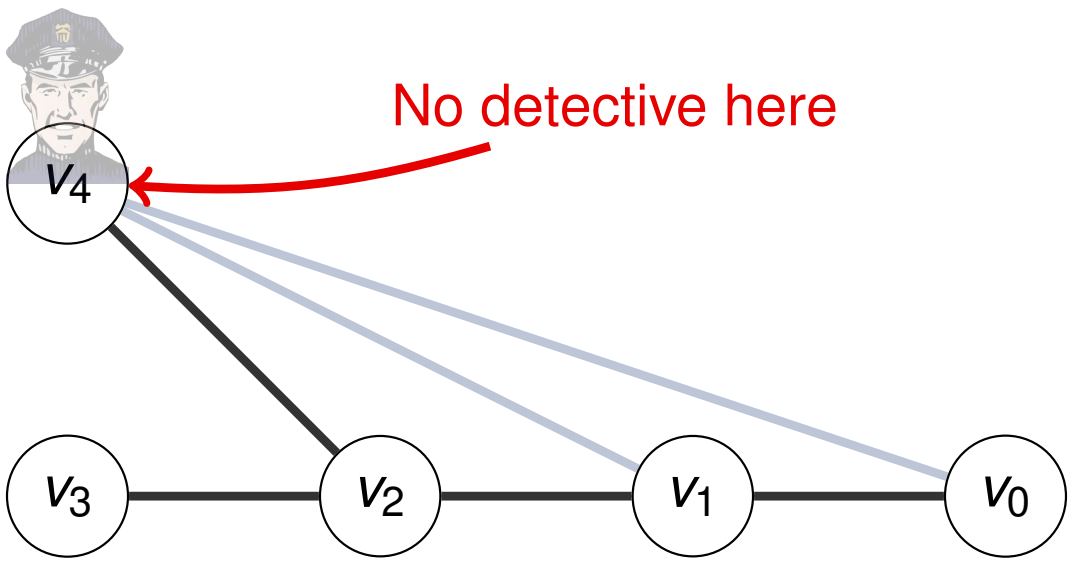
Use DFS to avoid this problem



Cheese Patrol

Now assume graph is arbitrary

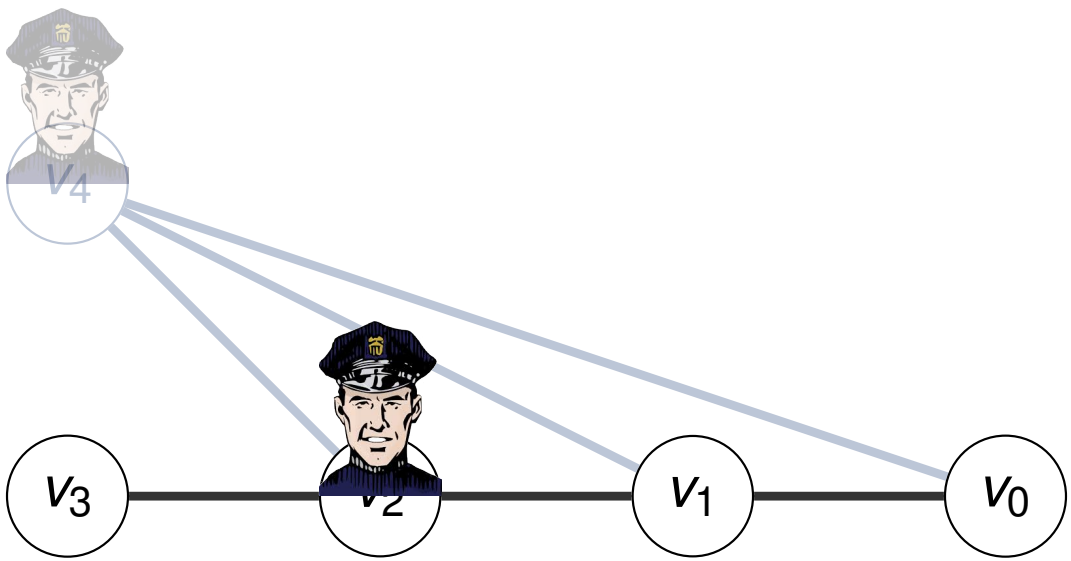
Use DFS to avoid this problem



Cheese Patrol

Now assume graph is arbitrary

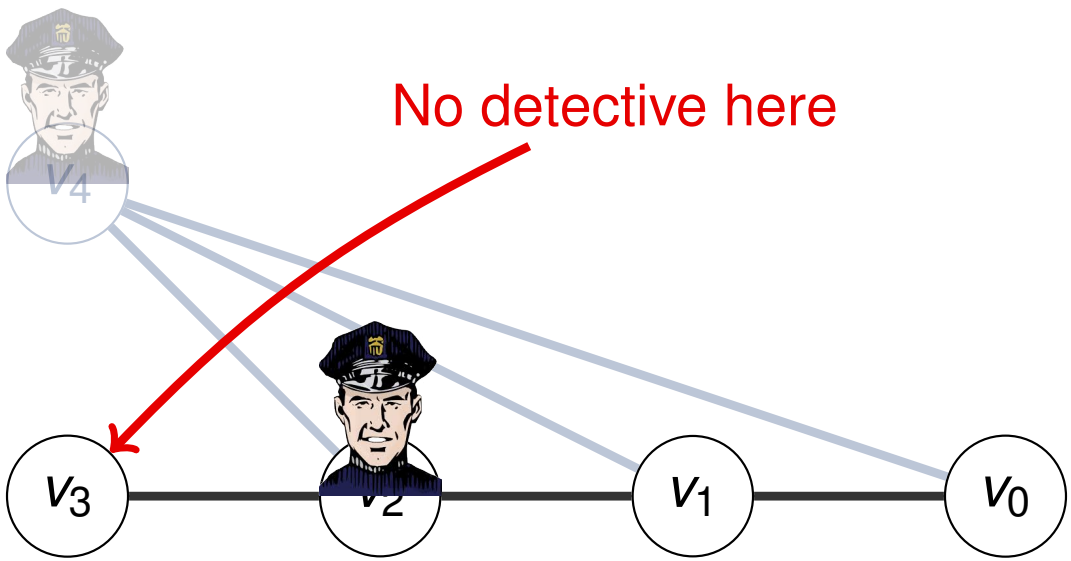
Use DFS to avoid this problem



Cheese Patrol

Now assume graph is arbitrary

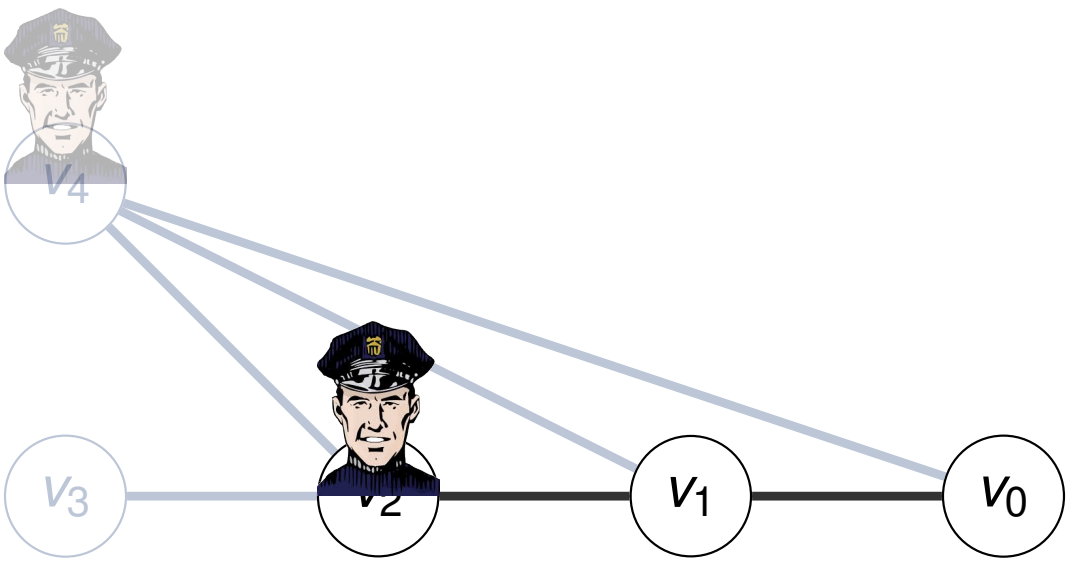
Use DFS to avoid this problem



Cheese Patrol

Now assume graph is arbitrary

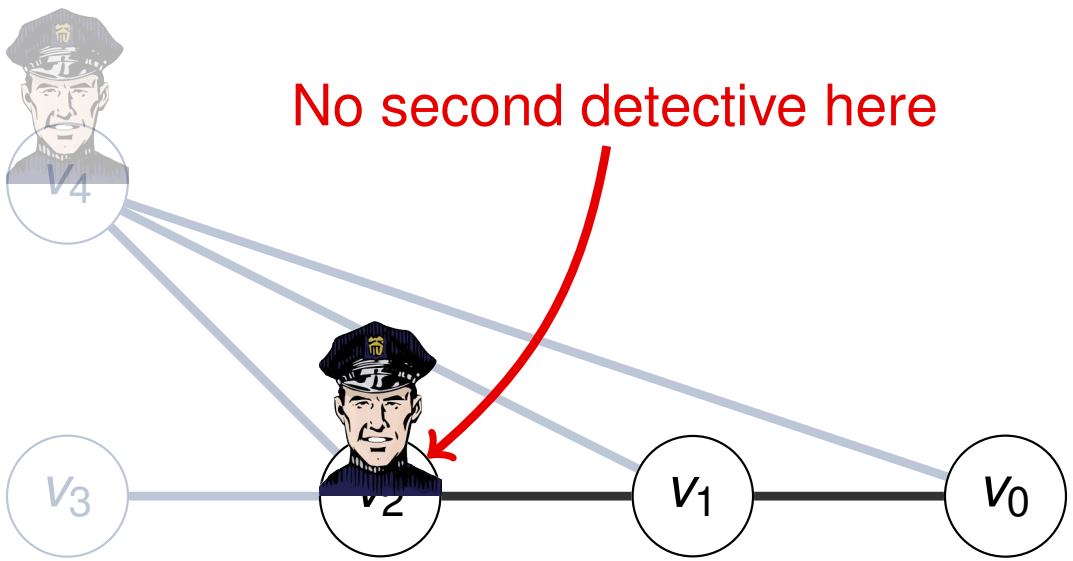
Use DFS to avoid this problem



Cheese Patrol

Now assume graph is arbitrary

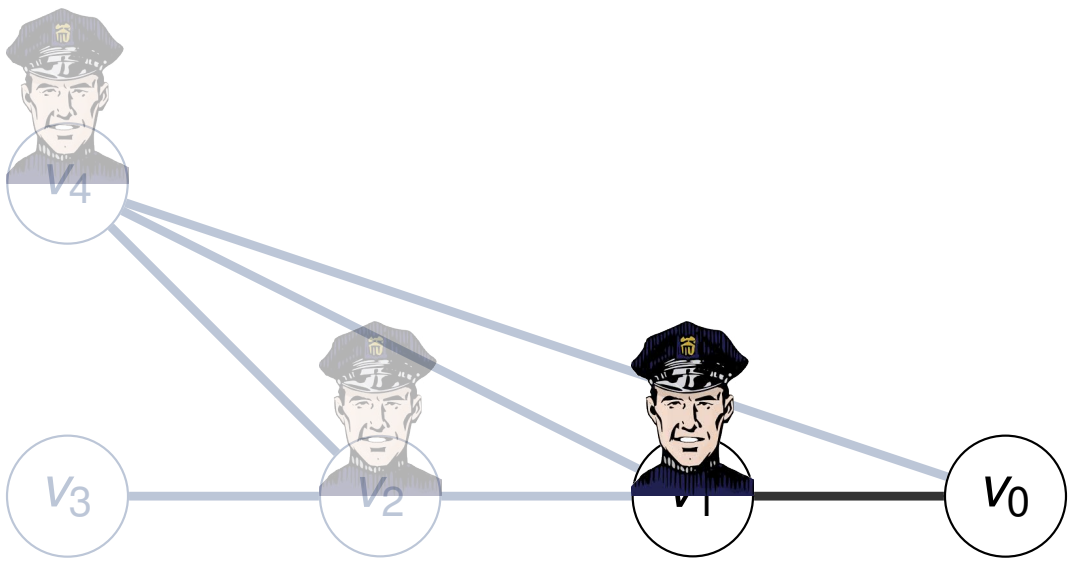
Use DFS to avoid this problem



Cheese Patrol

Now assume graph is arbitrary

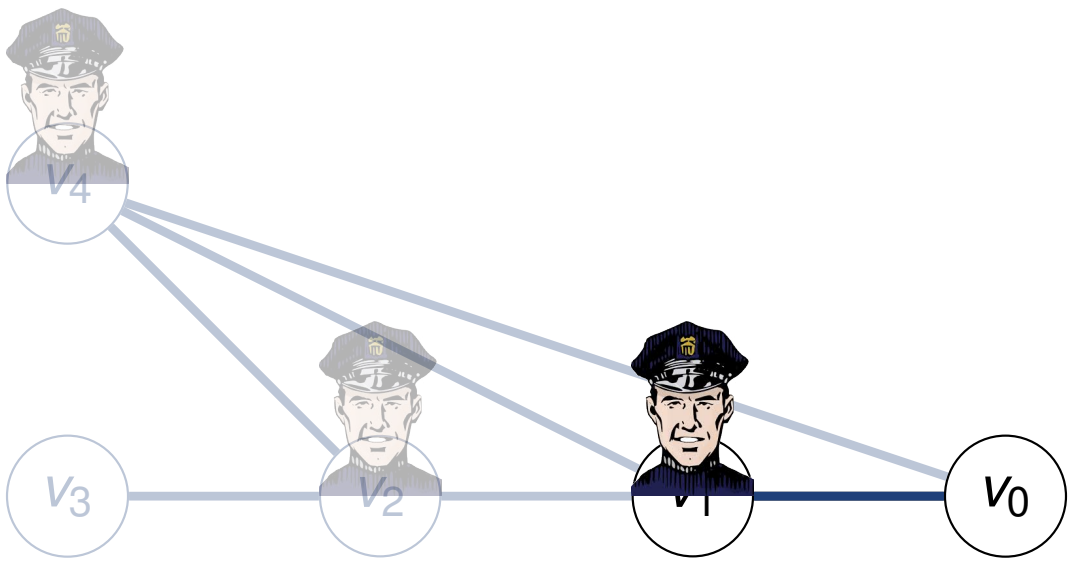
Use DFS to avoid this problem



Cheese Patrol

Now assume graph is arbitrary

Use DFS to avoid this problem



Thanks for the attention