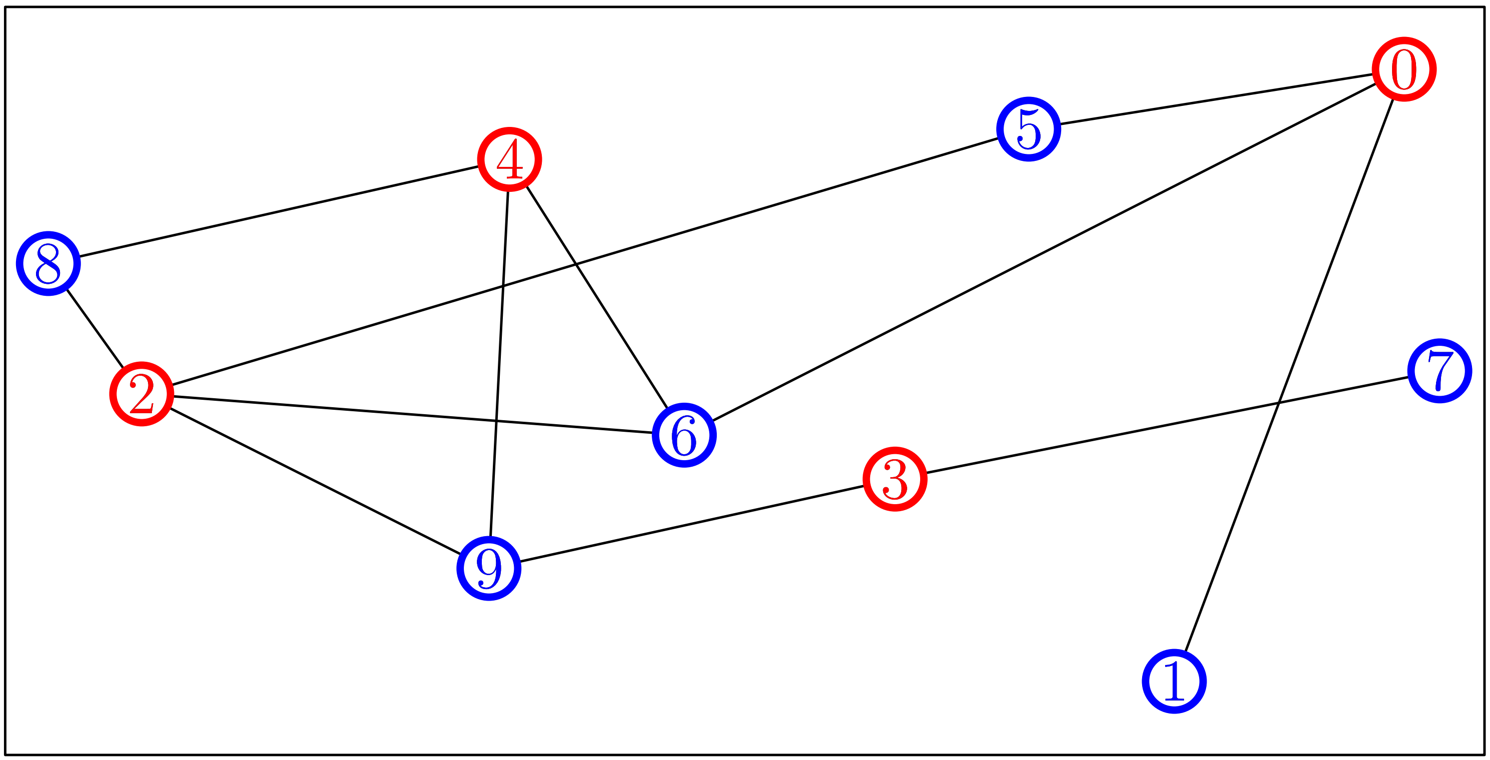
Algorithms and Analysis

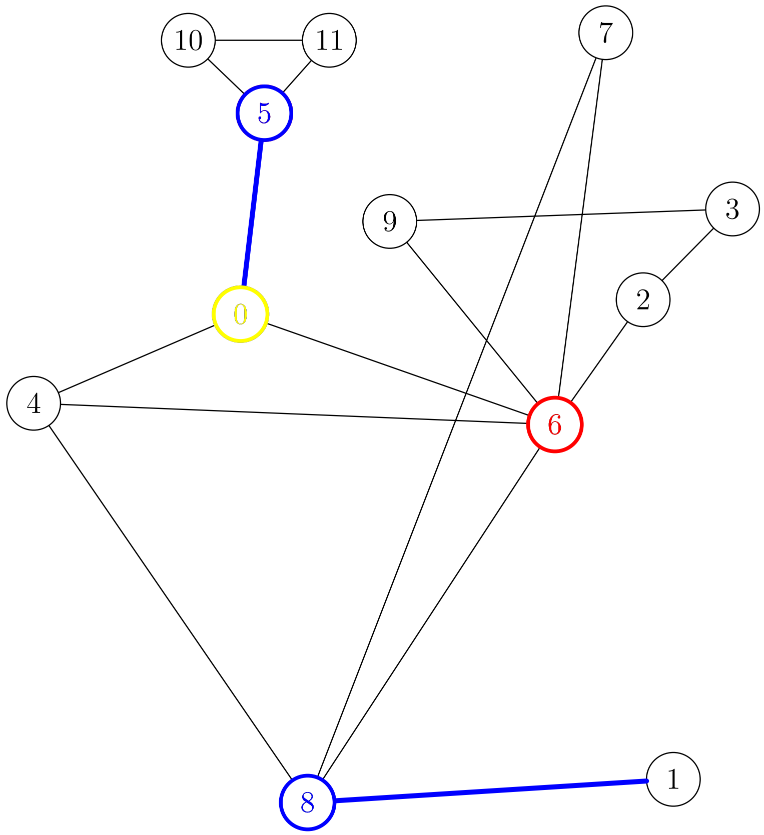
Lesson 20: *Learn to Traverse Graphs*



Breadth First Search, Depth First Search, Topological Sort

Outline

1. **Breadth First Search**
    - BFS applications
2. Depth First Search
    - DFS applications
3. Topological Sort



Basic Graph Algorithms

- Graphs provide an abstraction for a huge number of real world processes: social networks, compute network, road networks, etc.
- Increasing applications focus on very large (sparse) graphs (usually implemented using an adjacency list)
- Require (near) linear time algorithms
- Basic building block are graph traversal algorithms
  - ⋆ Breadth First Search
  - ⋆ Depth First Search

Basic Graph Algorithms

- Graphs provide an abstraction for a huge number of real world processes: social networks, compute network, road networks, etc.
- Increasing applications focus on very large (sparse) graphs (usually implemented using an adjacency list)
- Require (near) linear time algorithms
- Basic building block are graph traversal algorithms
  - ⋆ Breadth First Search
  - ⋆ Depth First Search

Basic Graph Algorithms

- Graphs provide an abstraction for a huge number of real world processes: social networks, compute network, road networks, etc.
- Increasing applications focus on very large (sparse) graphs (usually implemented using an adjacency list)
- Require (near) linear time algorithms
- Basic building block are graph traversal algorithms
  - ⋆ Breadth First Search
  - ⋆ Depth First Search

Basic Graph Algorithms

- Graphs provide an abstraction for a huge number of real world processes: social networks, compute network, road networks, etc.
- Increasing applications focus on very large (sparse) graphs (usually implemented using an adjacency list)
- Require (near) linear time algorithms
- Basic building block are graph traversal algorithms
  - ⋆ Breadth First Search
  - ⋆ Depth First Search

Advanced Generic

- To make these algorithm general purpose (generic) we allow ourselves to call arbitrary functions to act on the vertices and edges at different points in the algorithm
- This introduces a new level of generics which makes the algorithms very powerful
- Increases the steepness of the learning curve to use these algorithms
- Once you get familiar with using these algorithms this level of generics starts to pay off
- Libraries which does this include Boost Graph Library and LEDA in C++; JDSL and JGraphT in Java

Advanced Generic

- To make these algorithm general purpose (generic) we allow ourselves to call arbitrary functions to act on the vertices and edges at different points in the algorithm
- This introduces a new level of generics which makes the algorithms very powerful
- Increases the steepness of the learning curve to use these algorithms
- Once you get familiar with using these algorithms this level of generics starts to pay off
- Libraries which does this include Boost Graph Library and LEDA in C++; JDSL and JGraphT in Java

Advanced Generic

- To make these algorithm general purpose (generic) we allow ourselves to call arbitrary functions to act on the vertices and edges at different points in the algorithm

- This introduces a new level of generics which makes the algorithms very powerful

- Increases the steepness of the learning curve to use these algorithms

- Once you get familiar with using these algorithms this level of generics starts to pay off

- Libraries which does this include Boost Graph Library and LEDA in C++; JDSL and JGraphT in Java

Advanced Generic

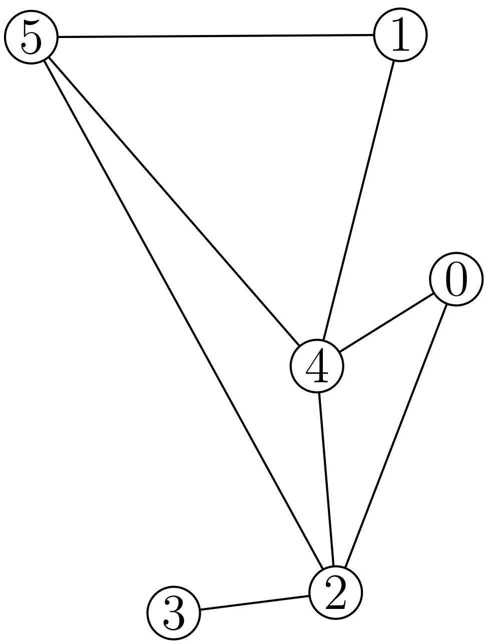
- To make these algorithm general purpose (generic) we allow ourselves to call arbitrary functions to act on the vertices and edges at different points in the algorithm
- This introduces a new level of generics which makes the algorithms very powerful
- Increases the steepness of the learning curve to use these algorithms
- Once you get familiar with using these algorithms this level of generics starts to pay off
- Libraries which does this include Boost Graph Library and LEDA in C++; JDSL and JGraphT in Java

Advanced Generic

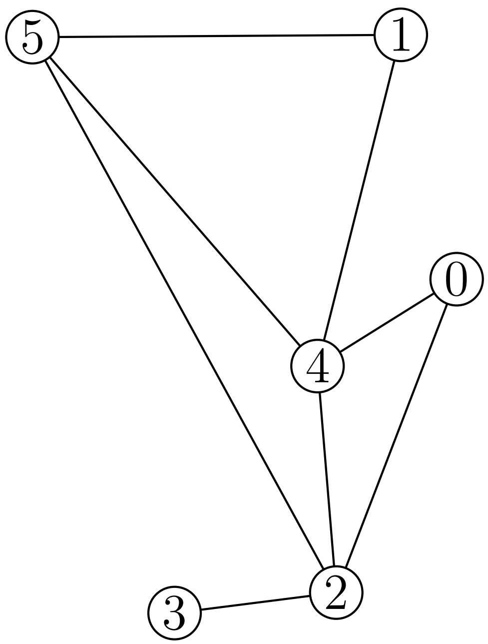
- To make these algorithm general purpose (generic) we allow ourselves to call arbitrary functions to act on the vertices and edges at different points in the algorithm
- This introduces a new level of generics which makes the algorithms very powerful
- Increases the steepness of the learning curve to use these algorithms
- Once you get familiar with using these algorithms this level of generics starts to pay off
- Libraries which does this include Boost Graph Library and LEDA in C++; JDSL and JGraphT in Java

Graph Traversal

- To traverse a graph we start at a (arbitrary) *root* vertex
- We the follow edges to create a tree



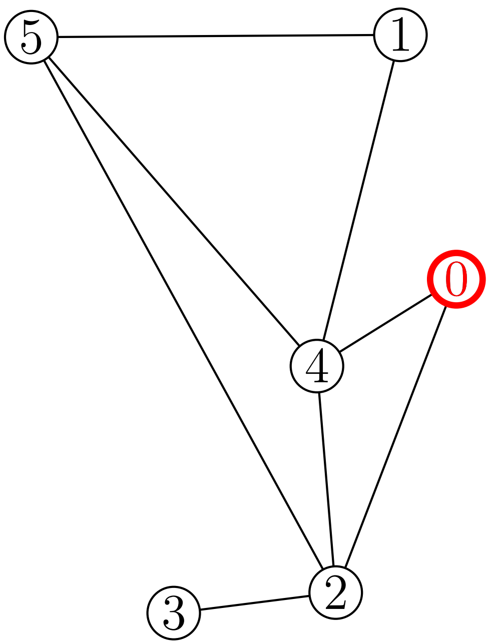
bfs



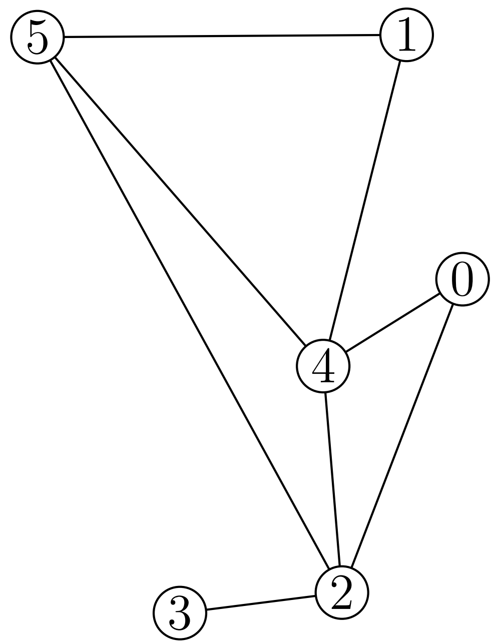
dfs

Graph Traversal

- To traverse a graph we start at a (arbitrary) $root$ vertex
- We the follow edges to create a tree



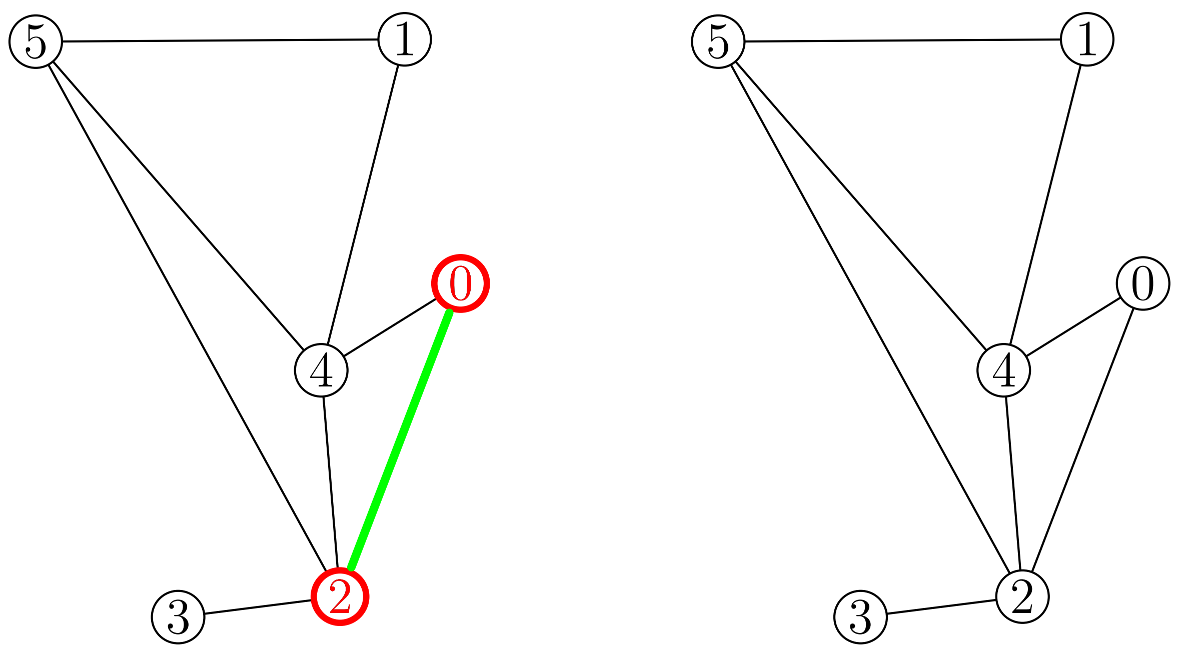
bfs



dfs

Graph Traversal

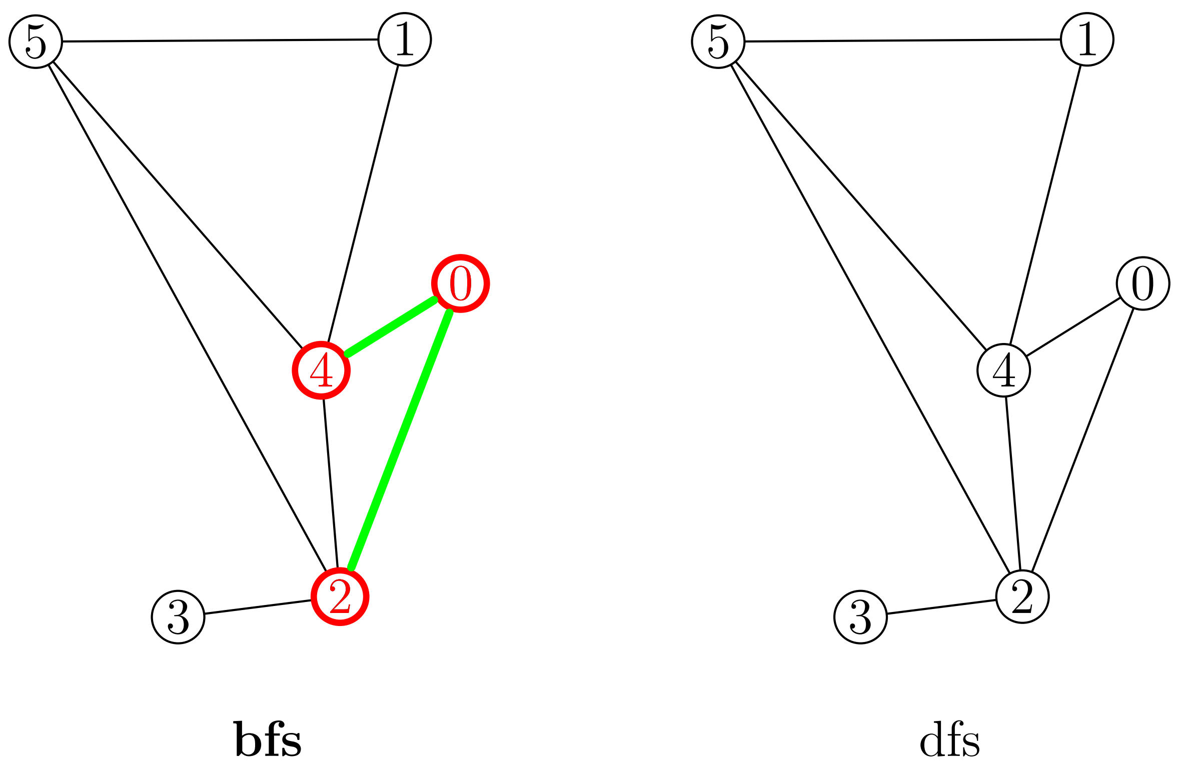
- To traverse a graph we start at a (arbitrary) *root* vertex
- We the follow edges to create a tree



bfs dfs

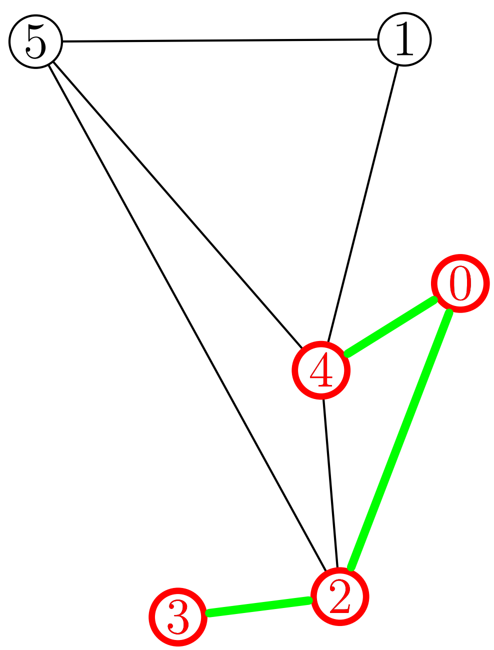
Graph Traversal

- To traverse a graph we start at a (arbitrary) *root* vertex
- We the follow edges to create a tree

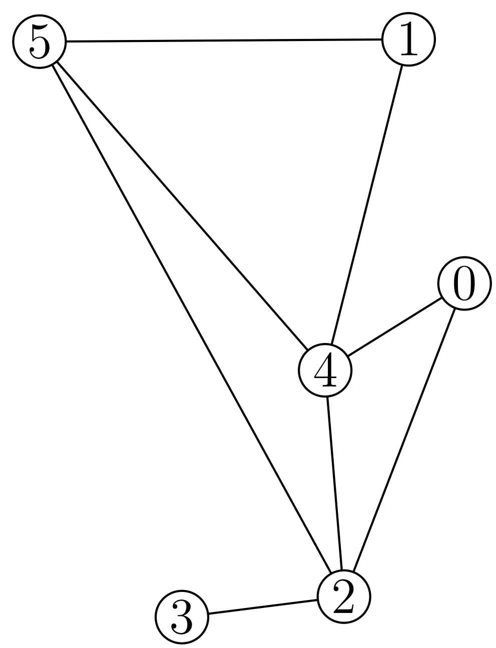


Graph Traversal

- To traverse a graph we start at a (arbitrary) *root* vertex
- We the follow edges to create a tree



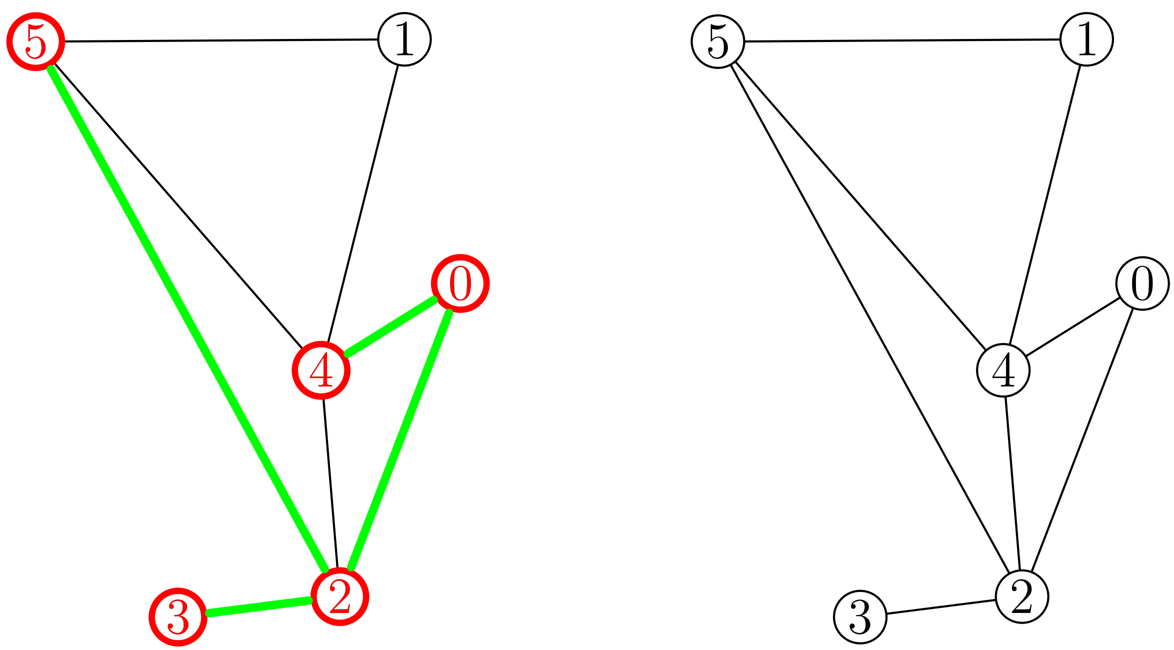
bfs



dfs

Graph Traversal

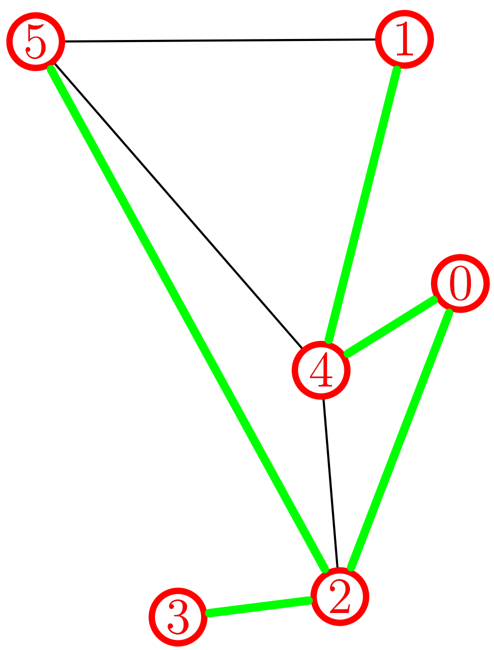
- To traverse a graph we start at a (arbitrary) *root* vertex
- We the follow edges to create a tree



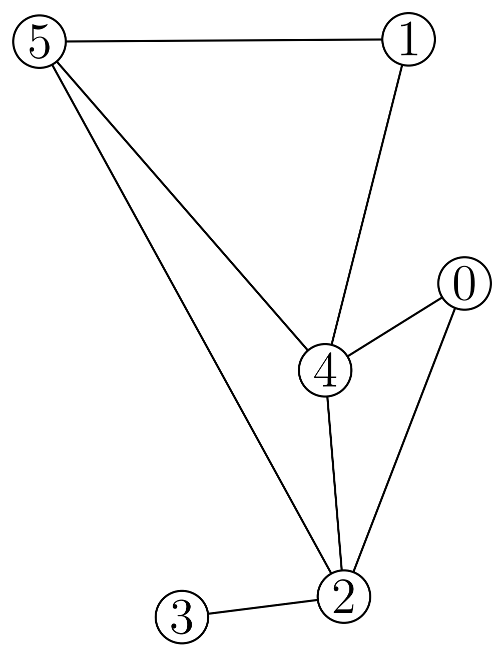
bfs dfs

Graph Traversal

- To traverse a graph we start at a (arbitrary) *root* vertex
- We the follow edges to create a tree



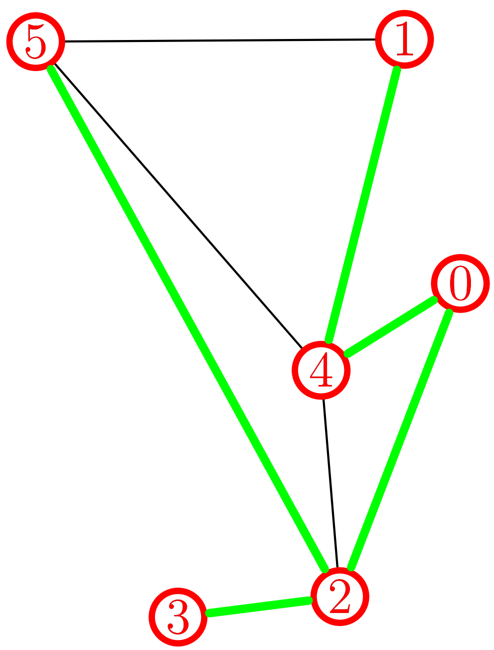
bfs



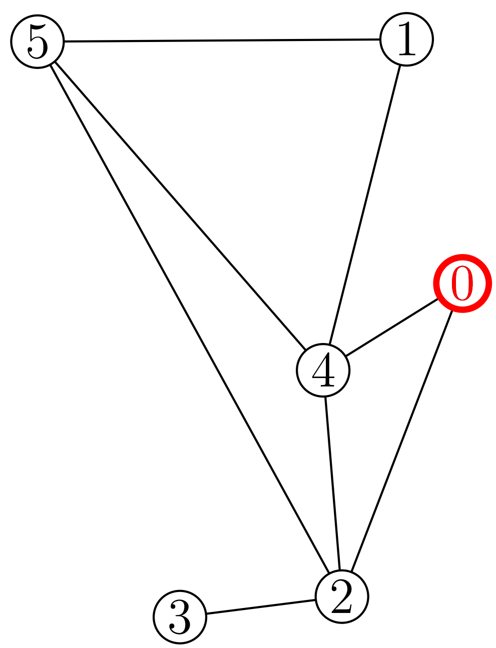
dfs

Graph Traversal

- To traverse a graph we start at a (arbitrary) *root* vertex
- We the follow edges to create a tree



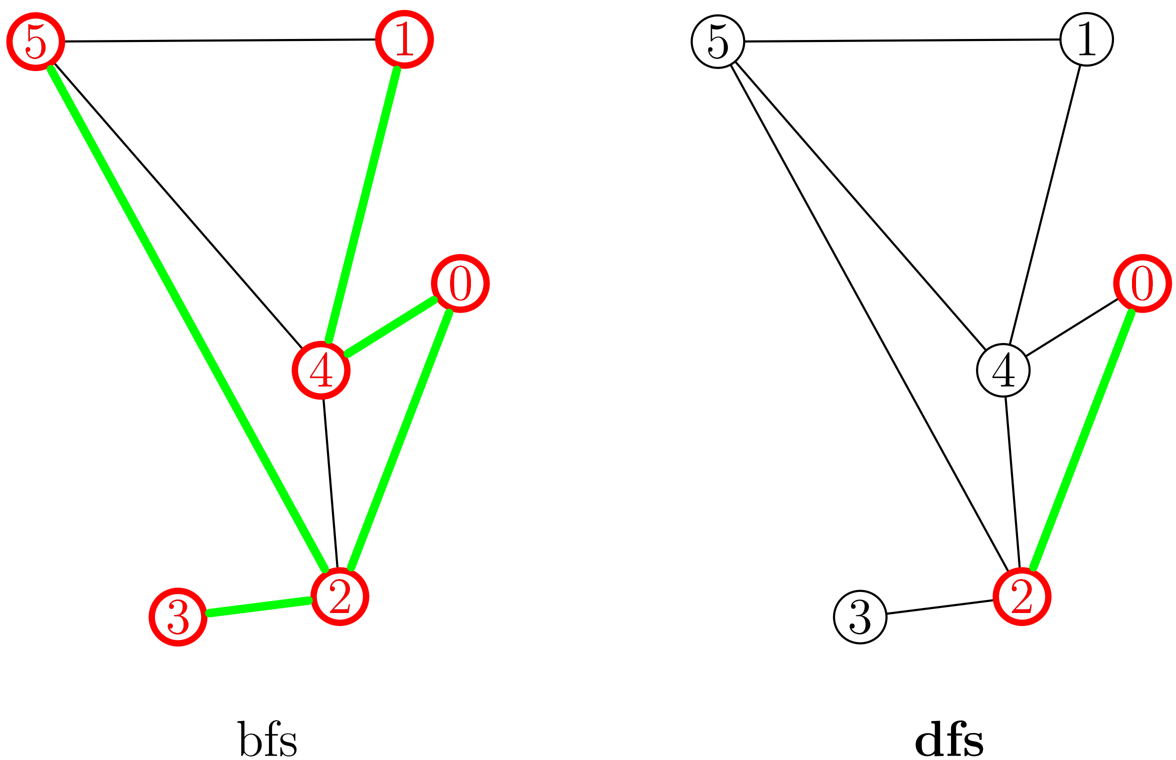
bfs



dfs

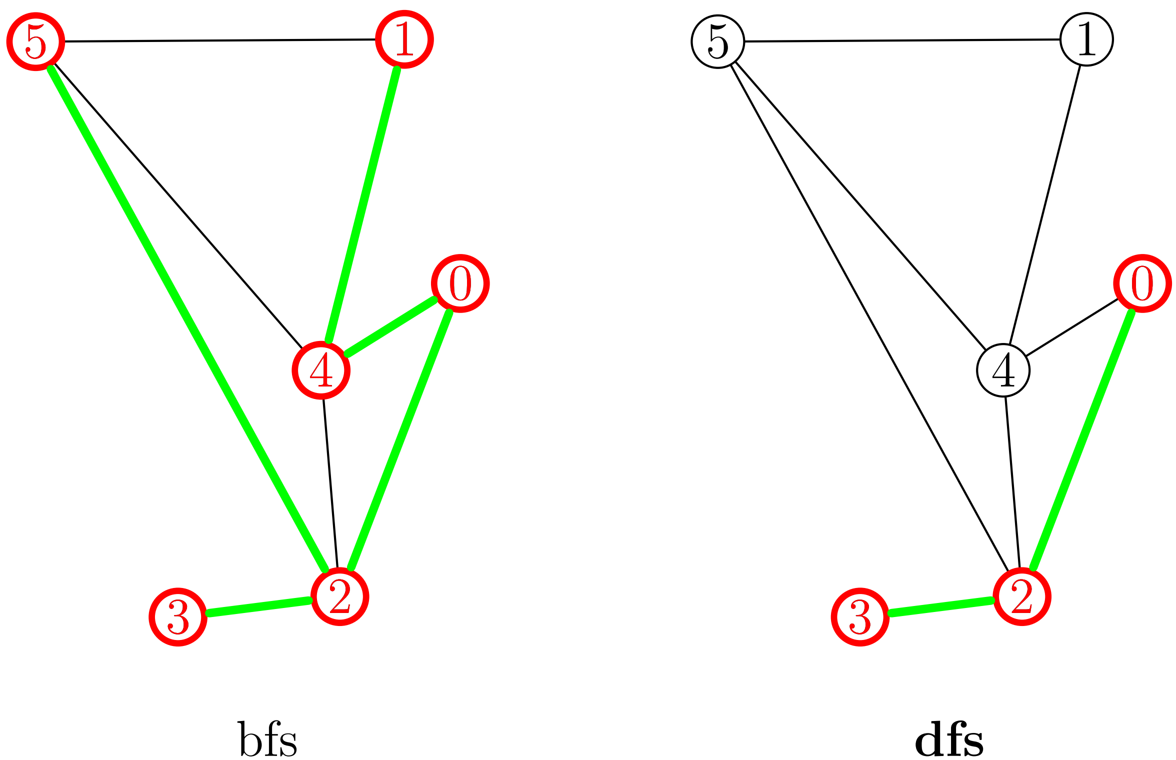
Graph Traversal

- To traverse a graph we start at a (arbitrary) *root* vertex
- We the follow edges to create a tree



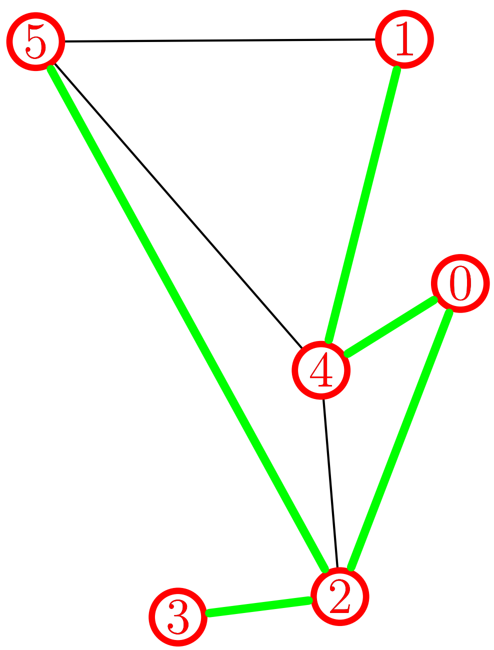
Graph Traversal

- To traverse a graph we start at a (arbitrary) *root* vertex
- We the follow edges to create a tree

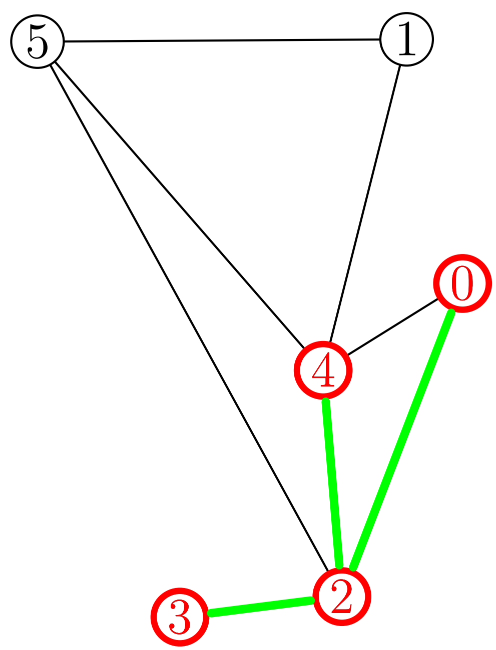


Graph Traversal

- To traverse a graph we start at a (arbitrary) *root* vertex
- We the follow edges to create a tree



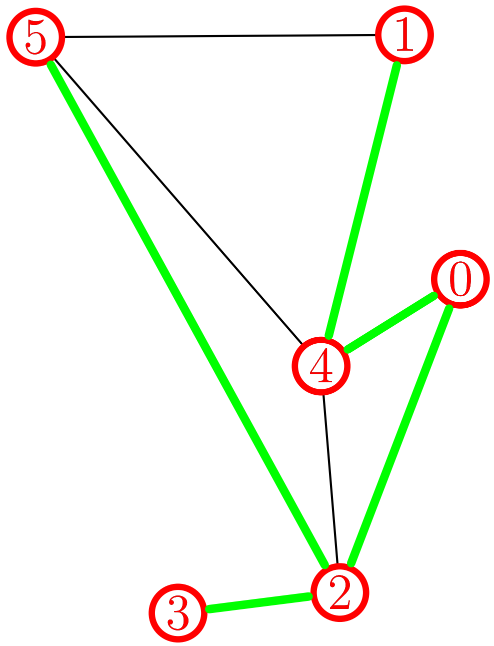
bfs



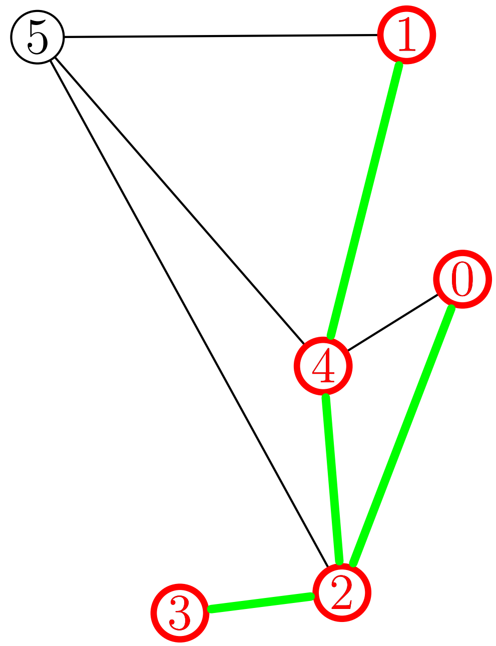
dfs

Graph Traversal

- To traverse a graph we start at a (arbitrary) *root* vertex
- We the follow edges to create a tree



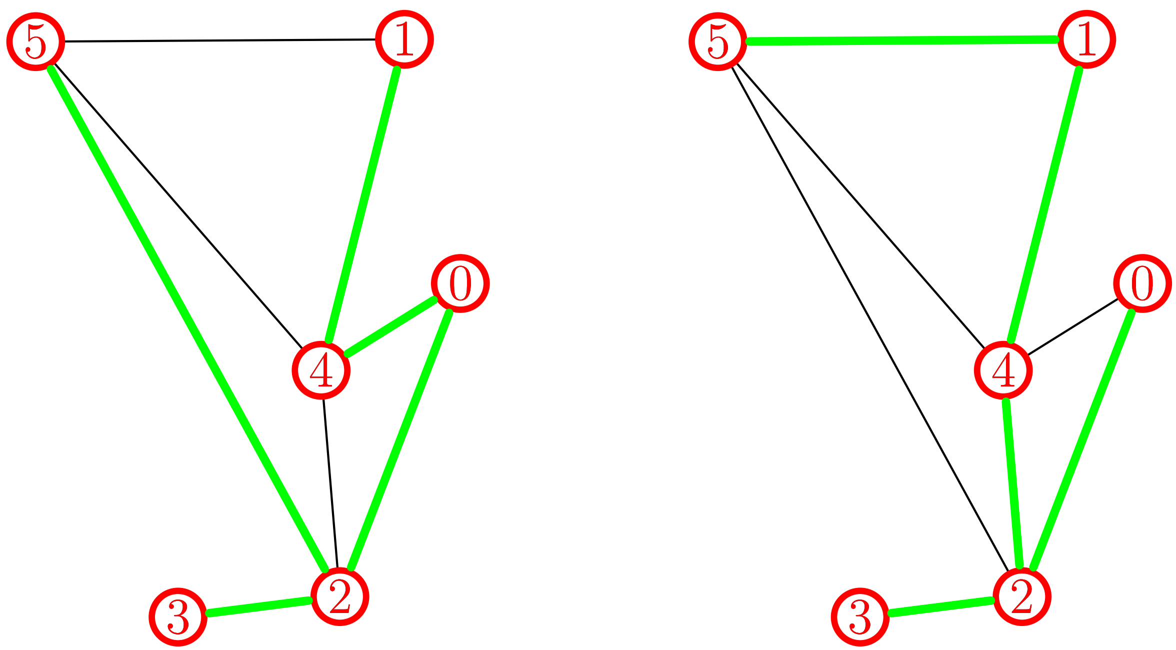
bfs



dfs

Graph Traversal

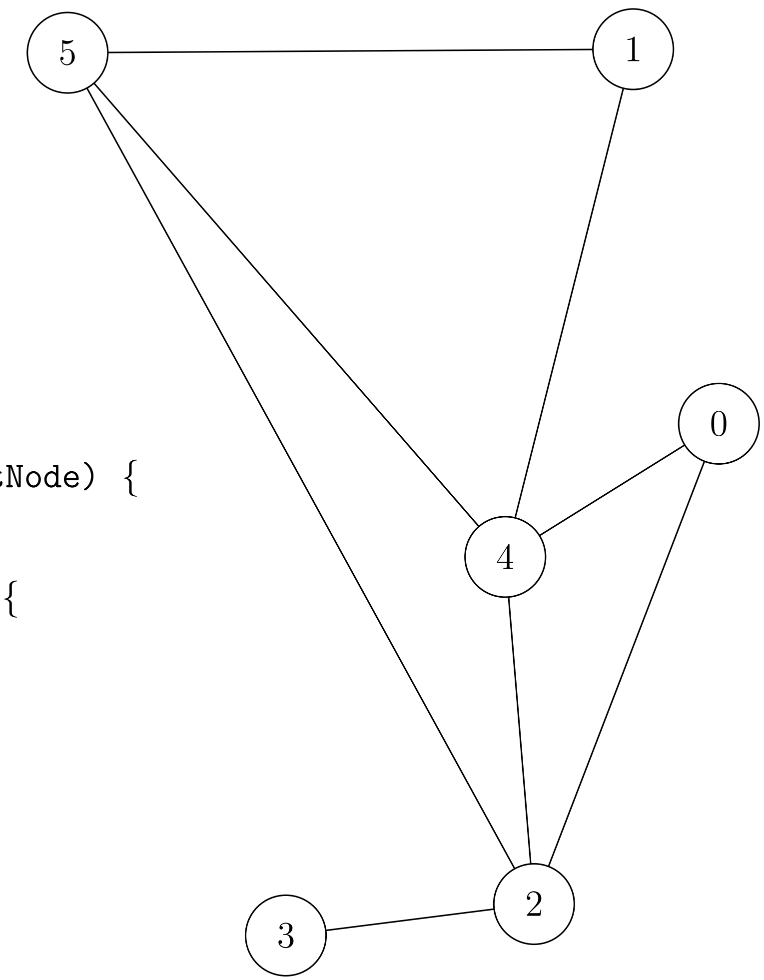
- To traverse a graph we start at a (arbitrary) $root$ vertex
- We the follow edges to create a tree



bfs **dfs**

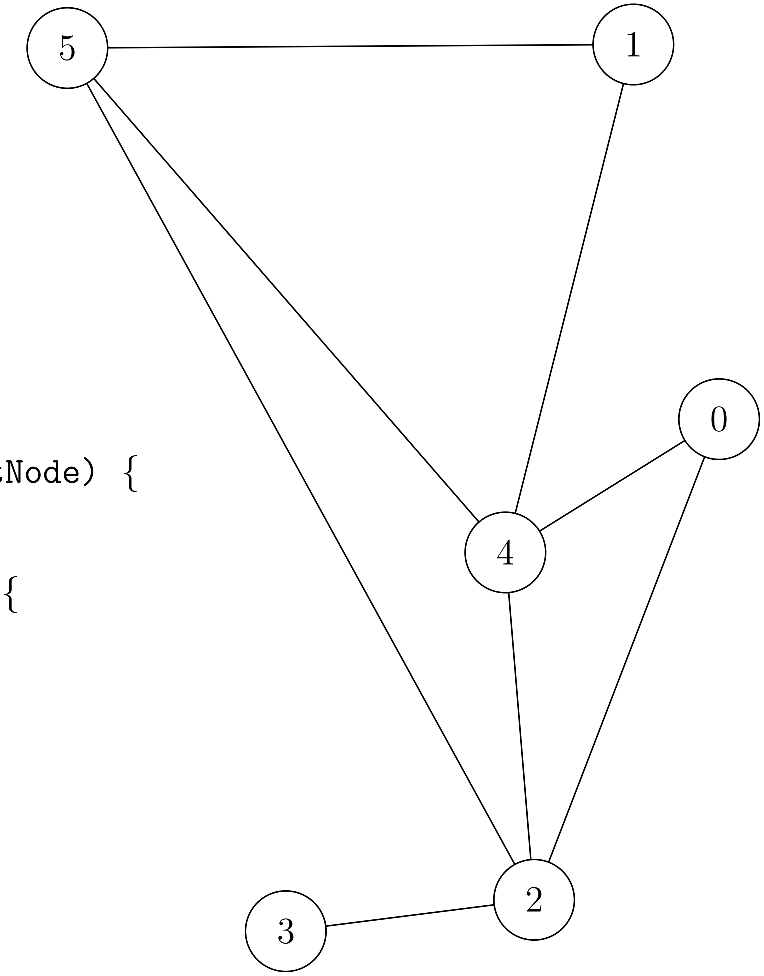
Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```



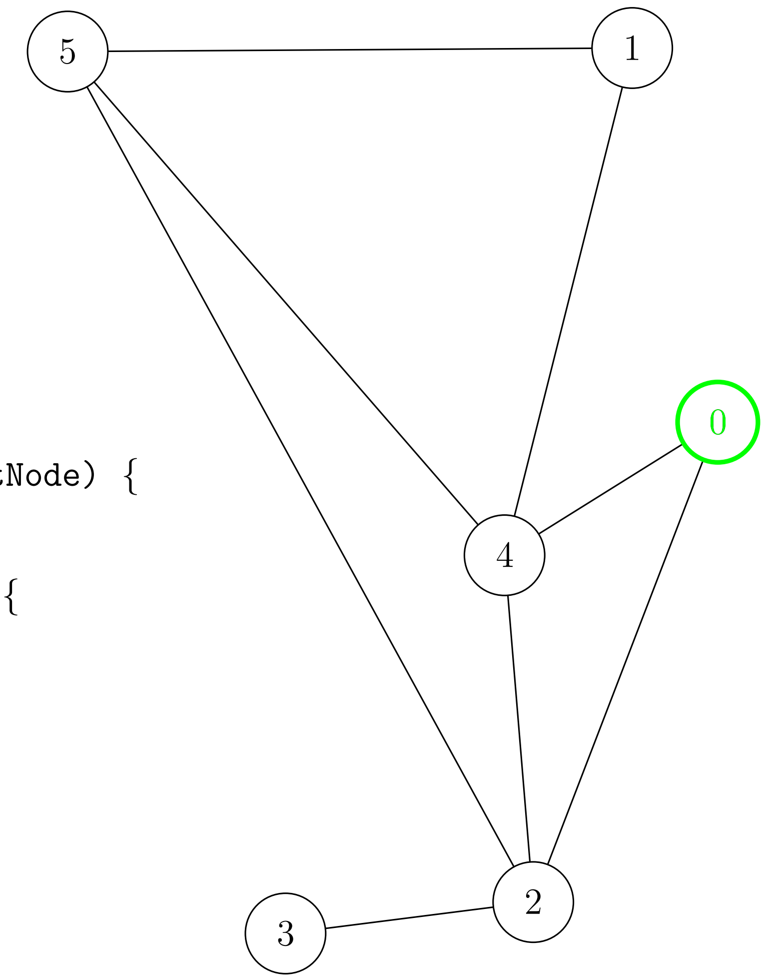
Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```



Breadth First Search

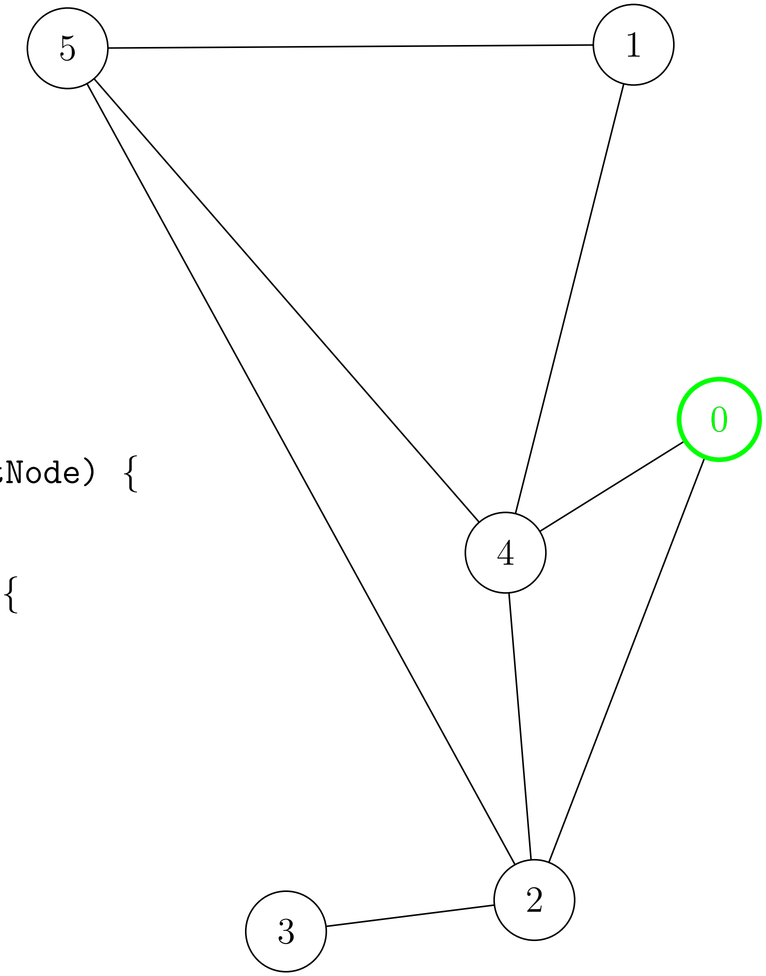
```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

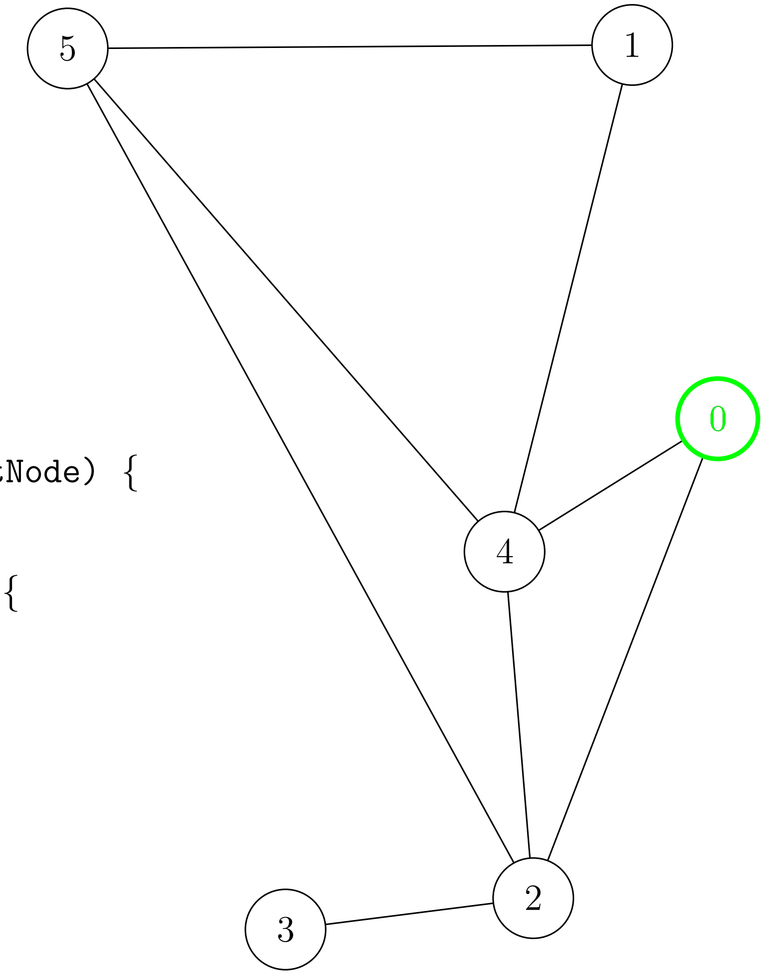
q=



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q= | 0 | | | | | |

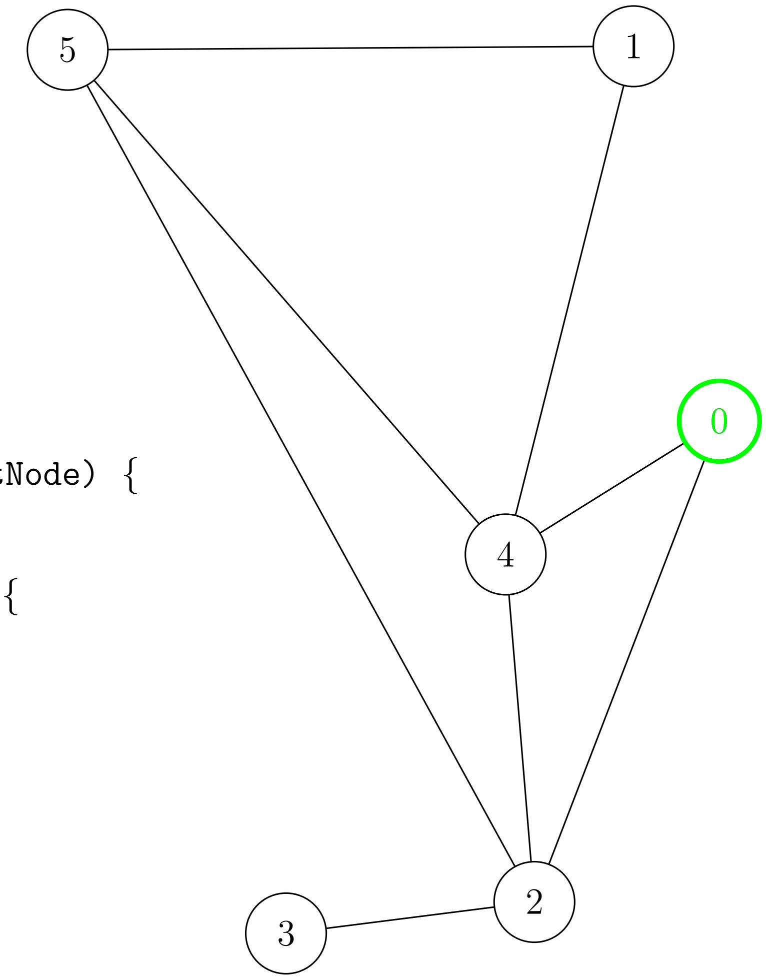


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

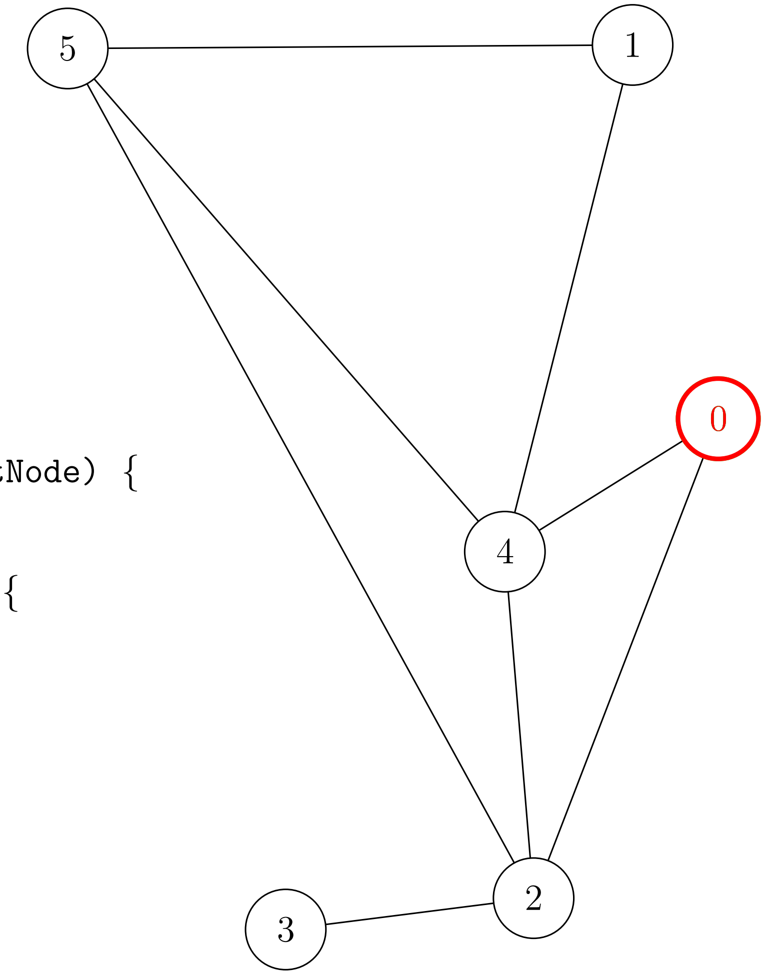
0					



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

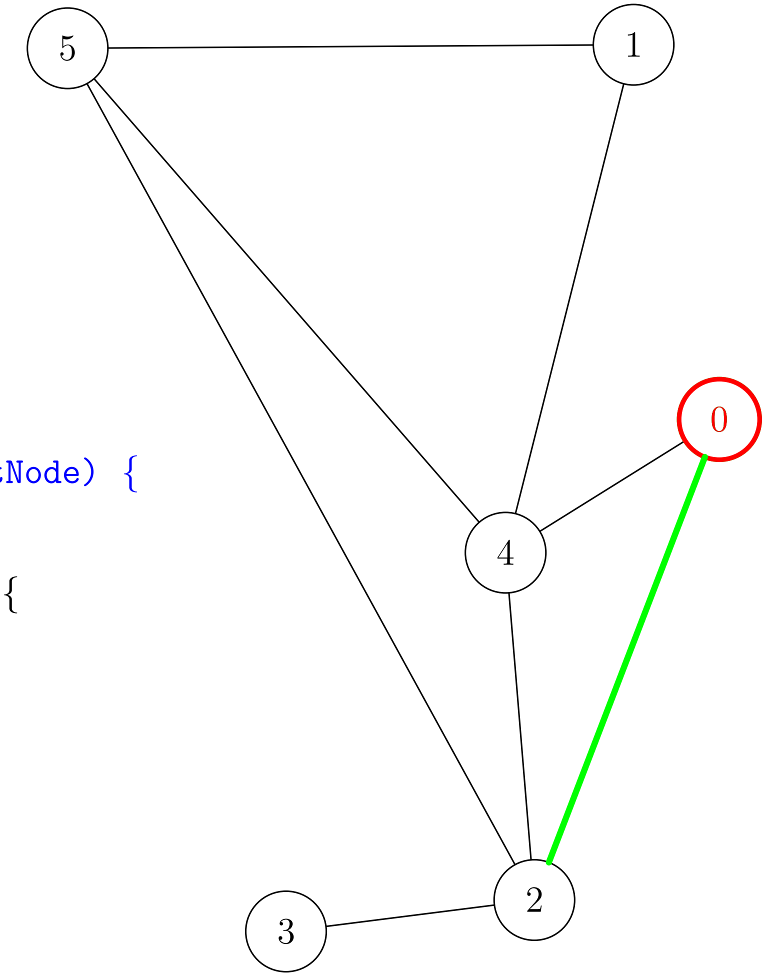
q=



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

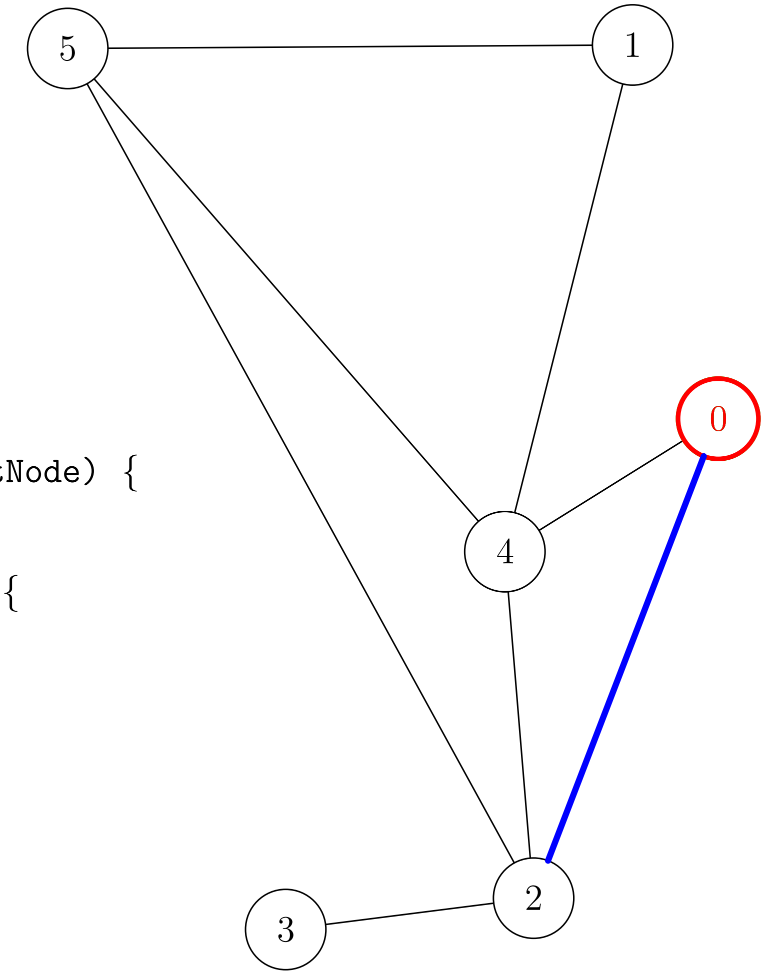
q=



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

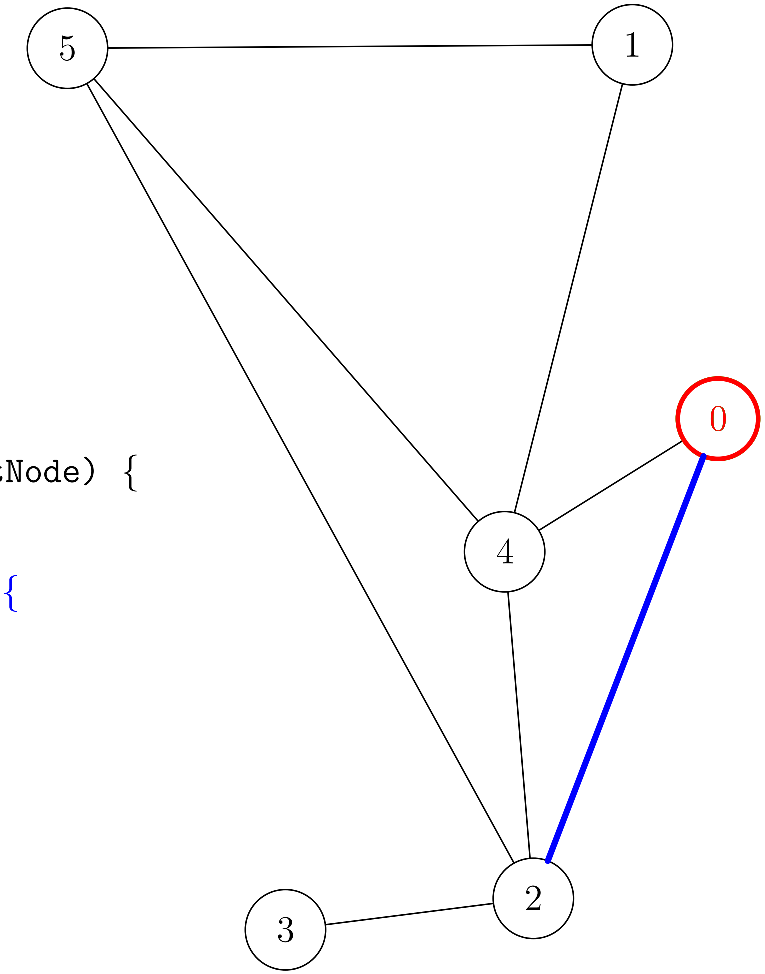
q=					



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

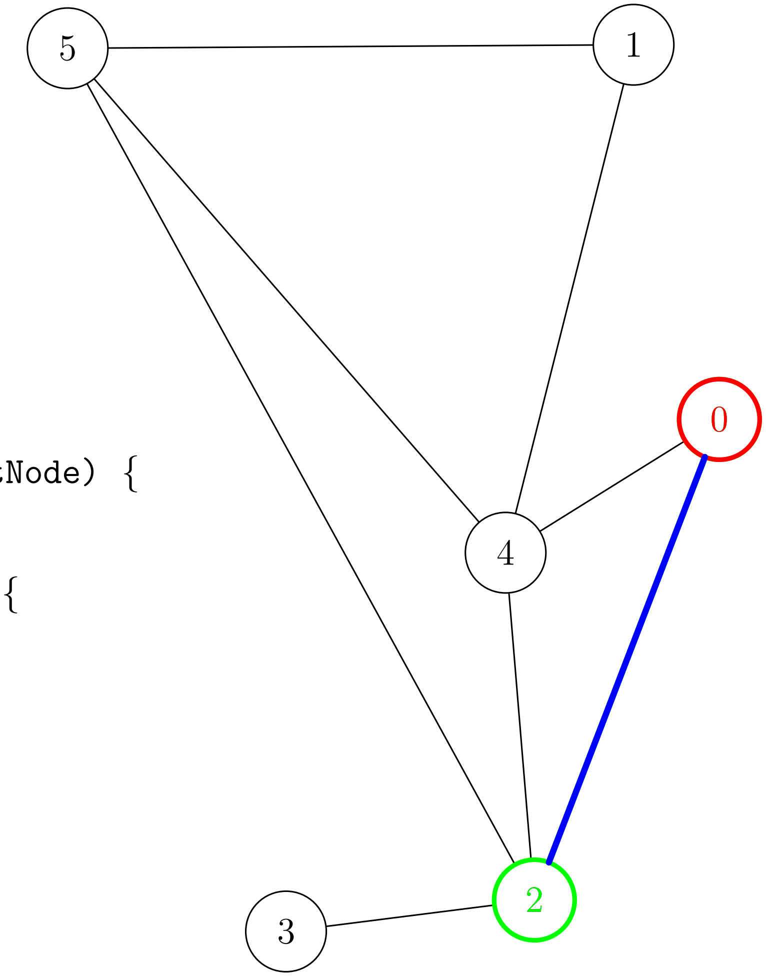
q=



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q= | 2 | | | | | |
|---|---|---|---|---|---|

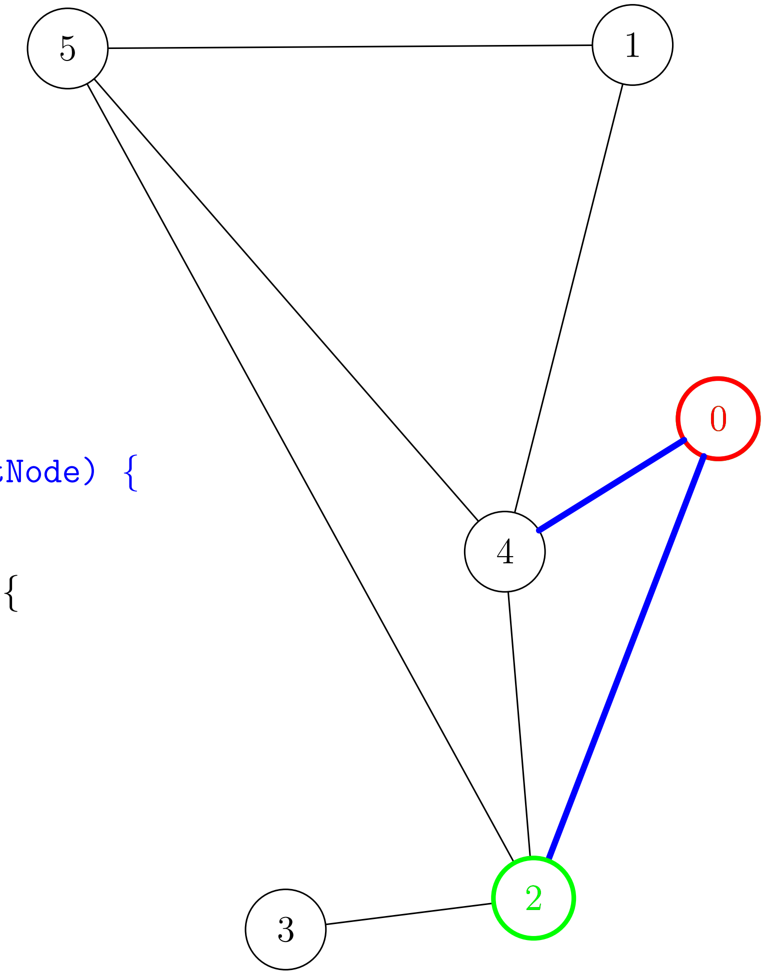


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

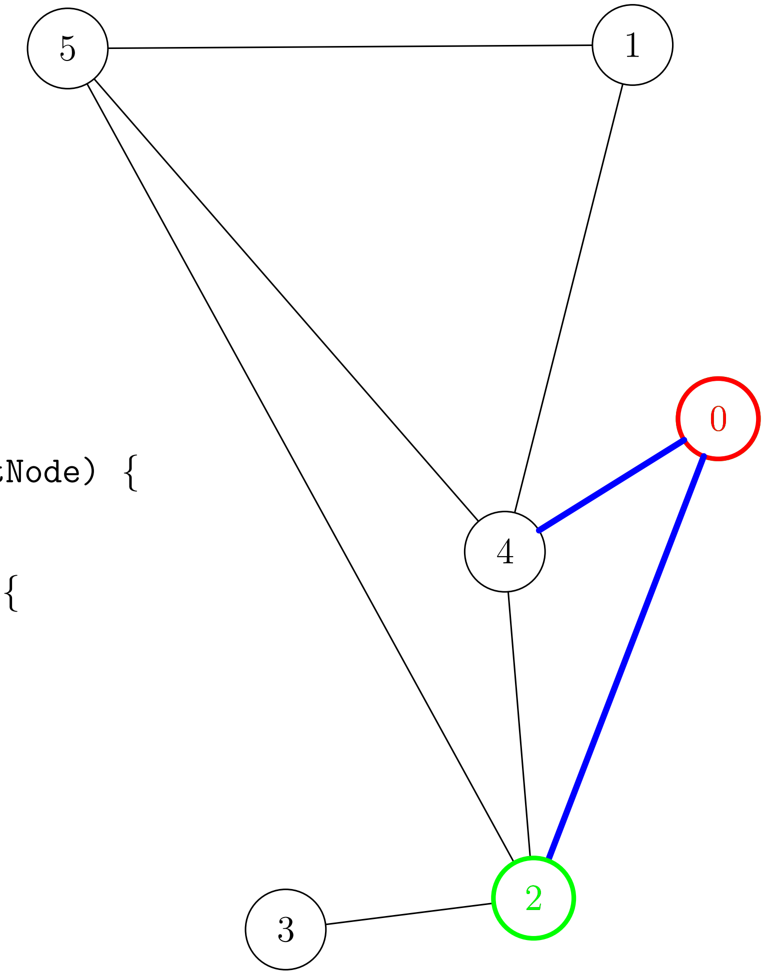
2					



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q= | 2 | | | | | |

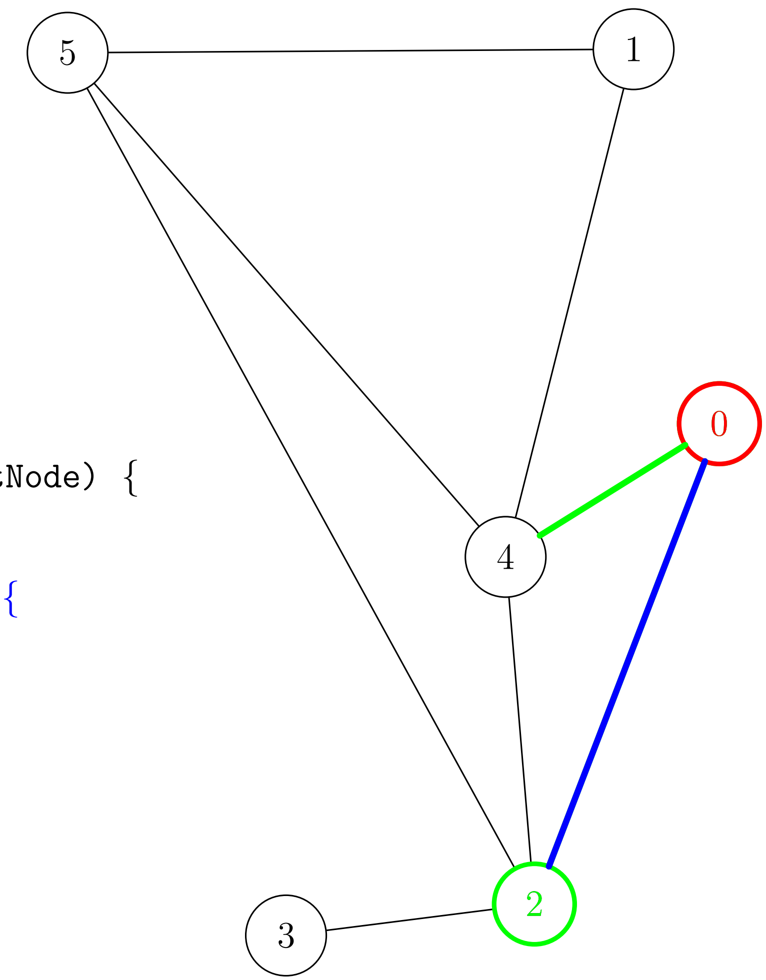


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

2					

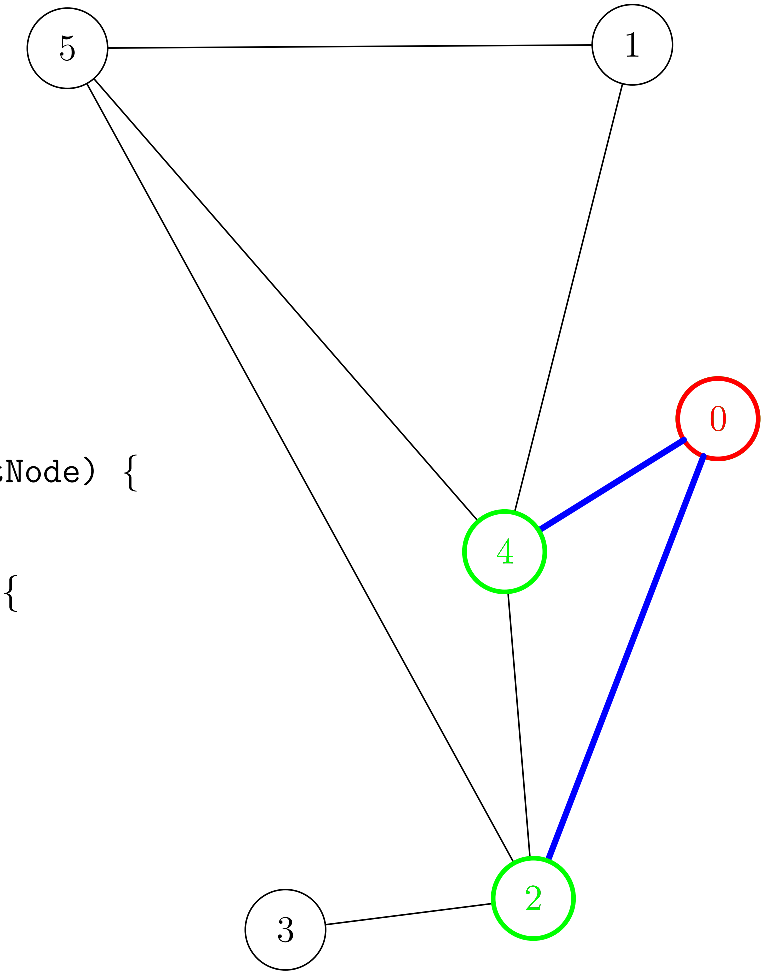


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

2	4				

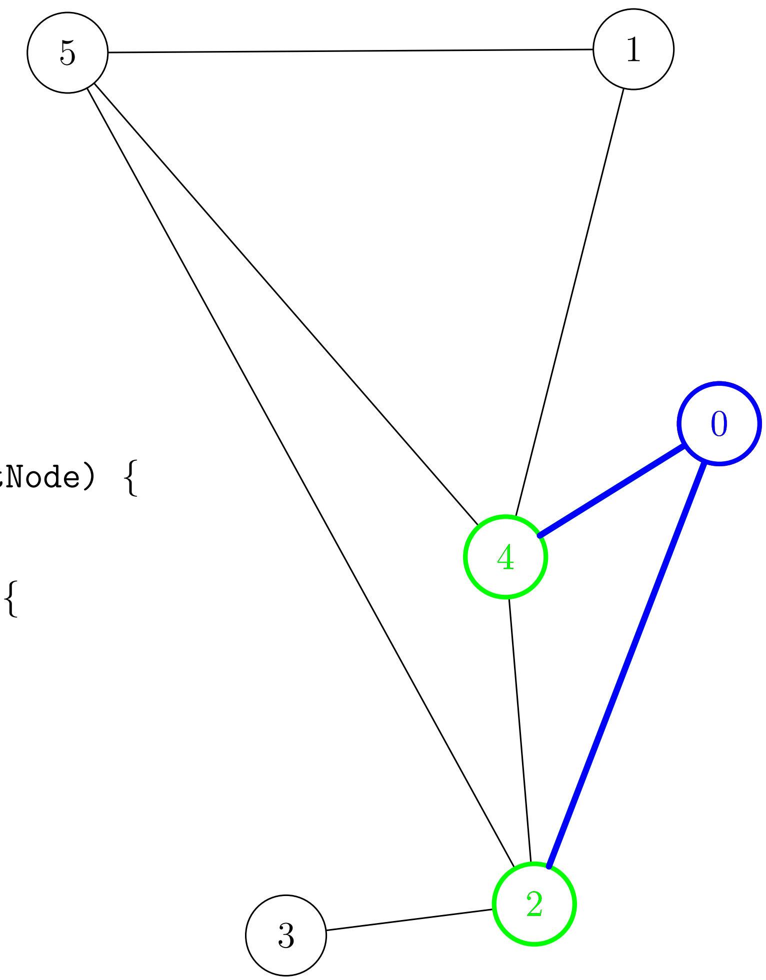


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

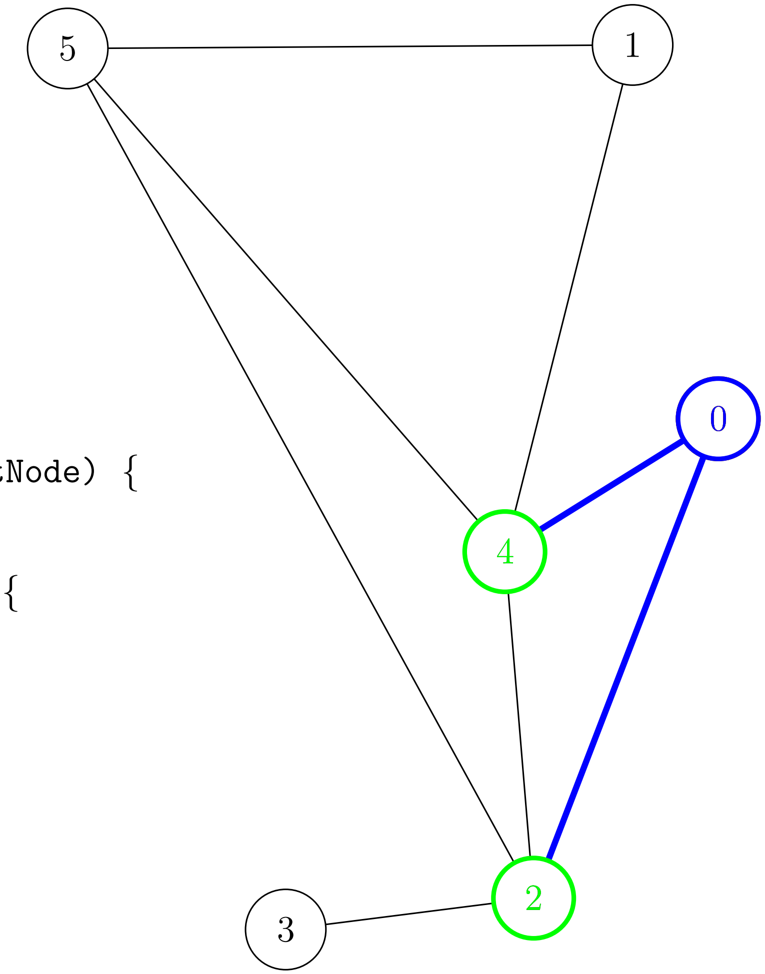
2	4				



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=	2	4			

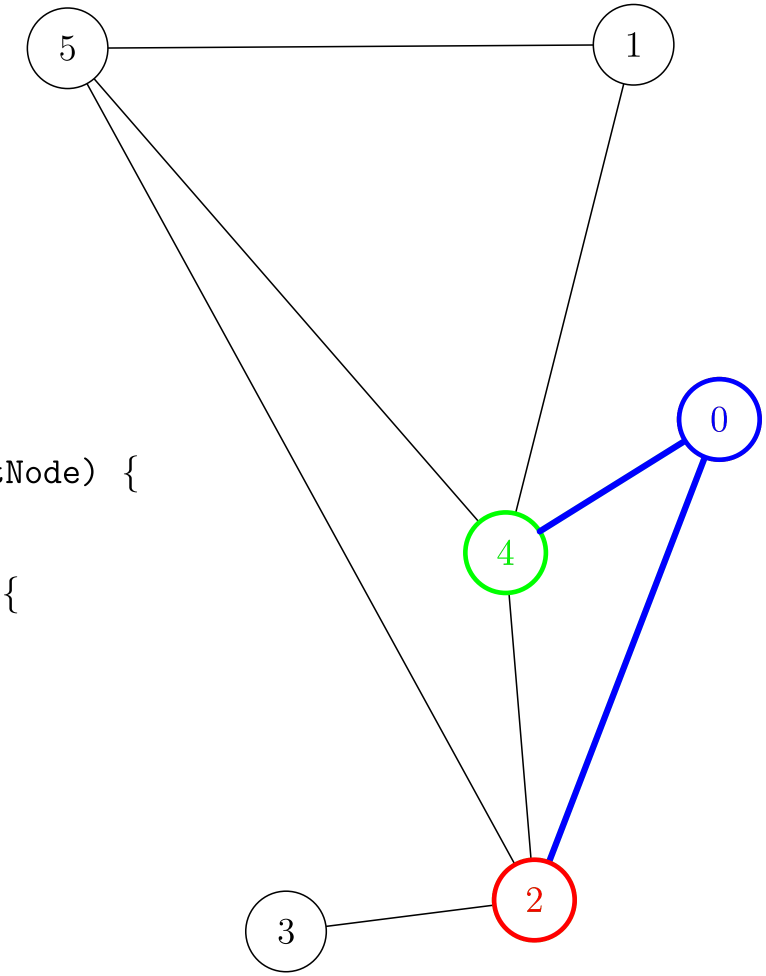


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

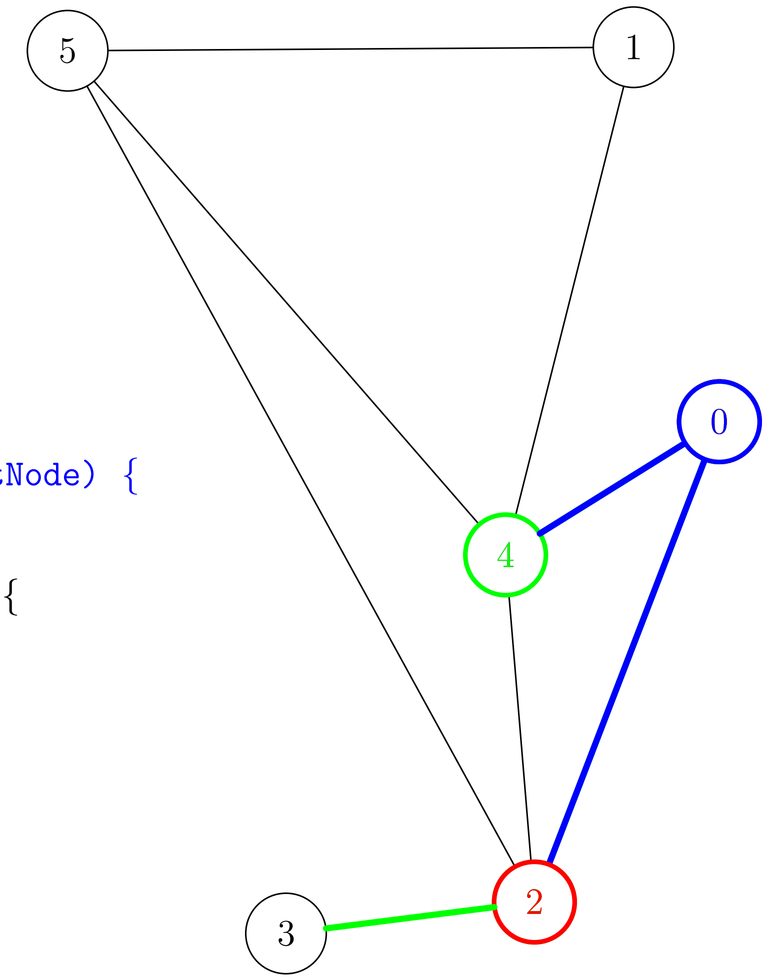
4					



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q= | 4 | | | | | |
|---|---|---|---|---|---|

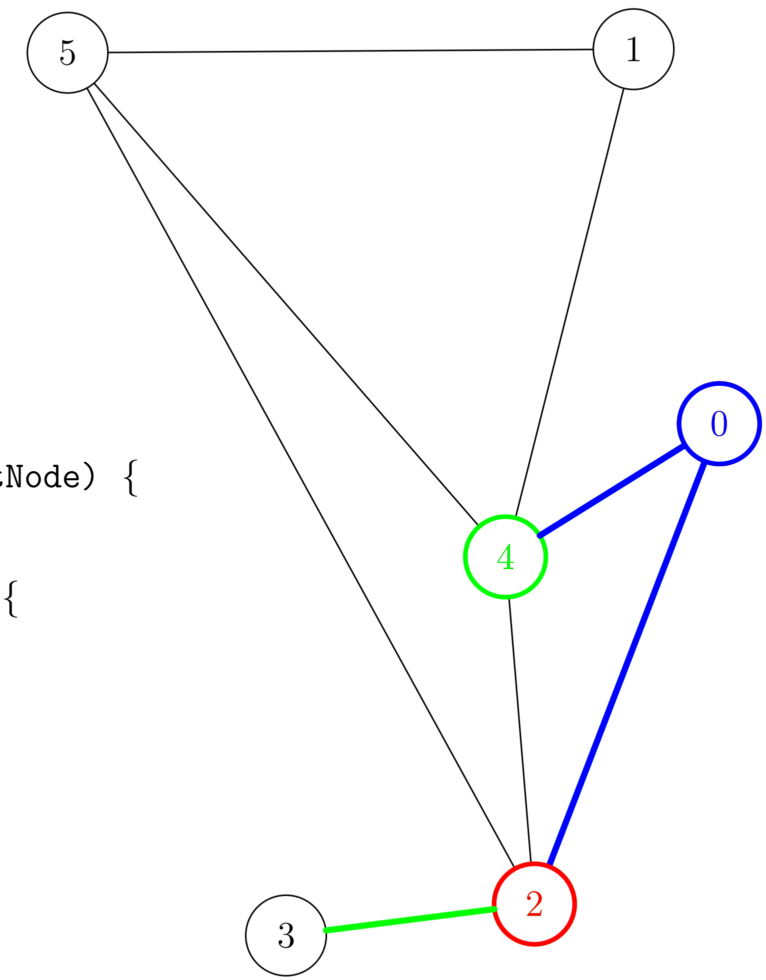


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

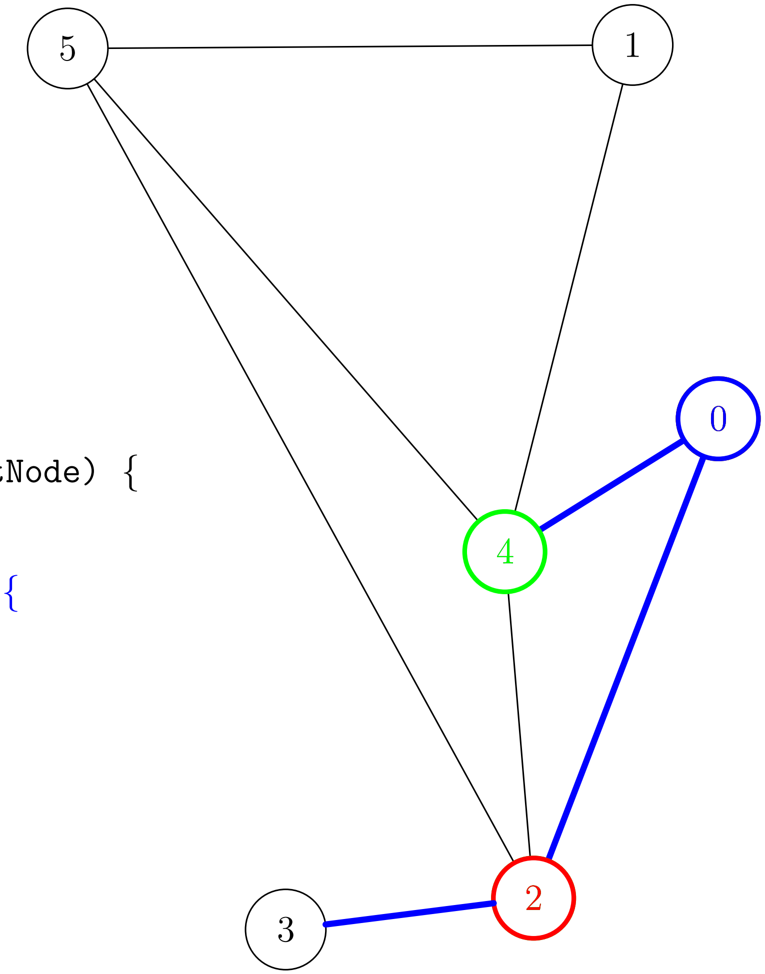
4					



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=	4					

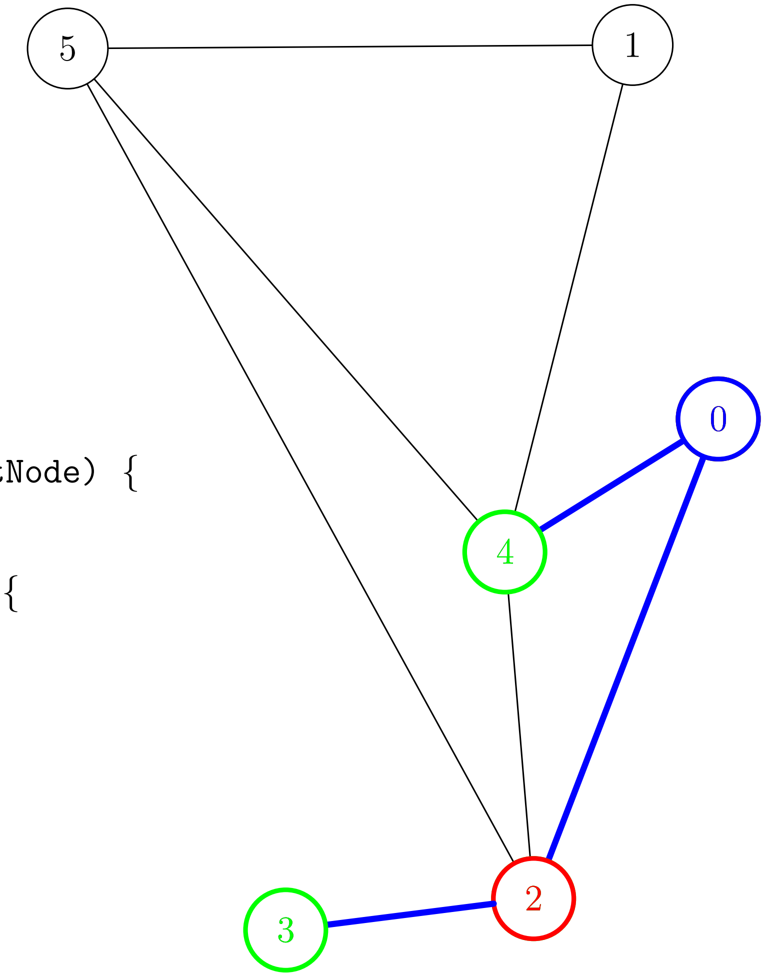


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

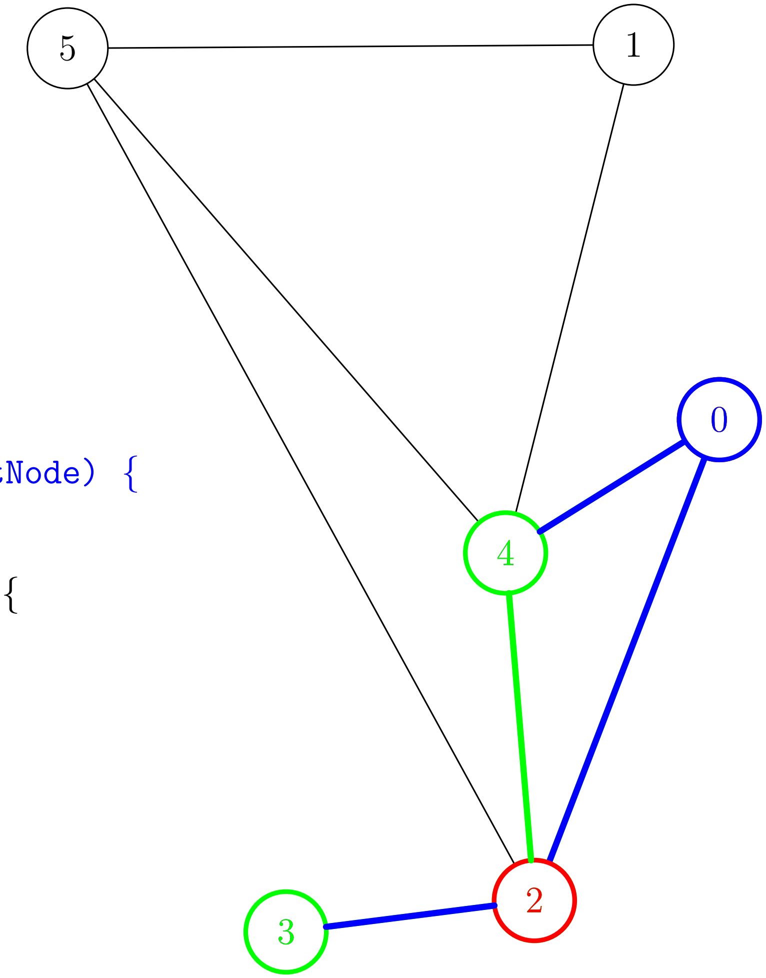
4	3				



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q= | 4 | 3 | | | | |

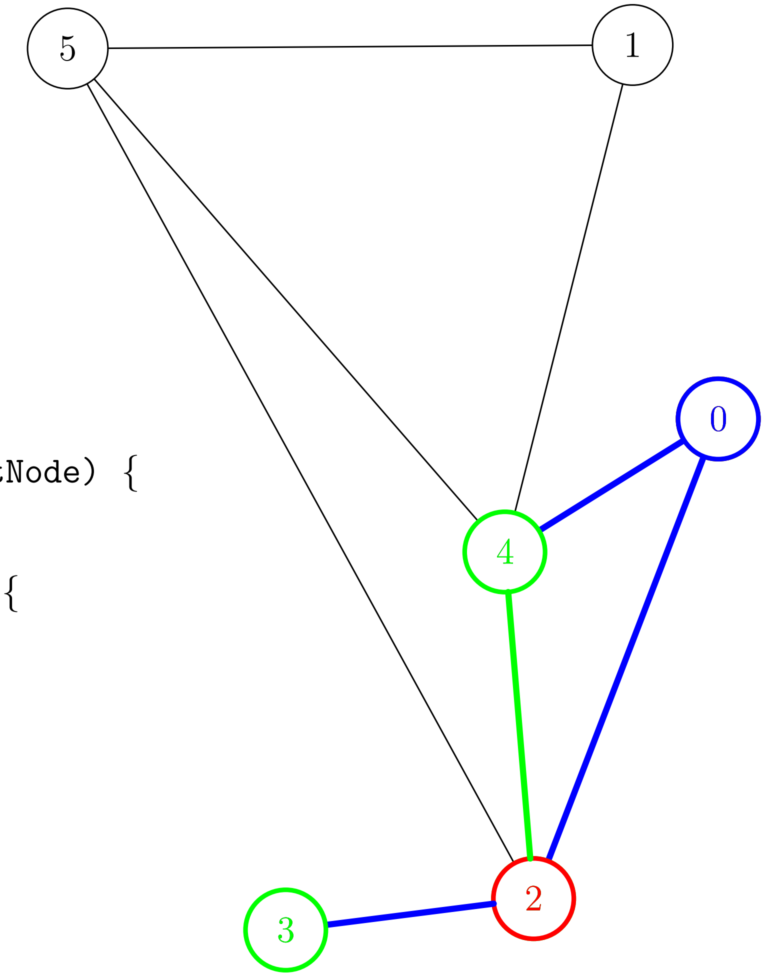


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

4	3				

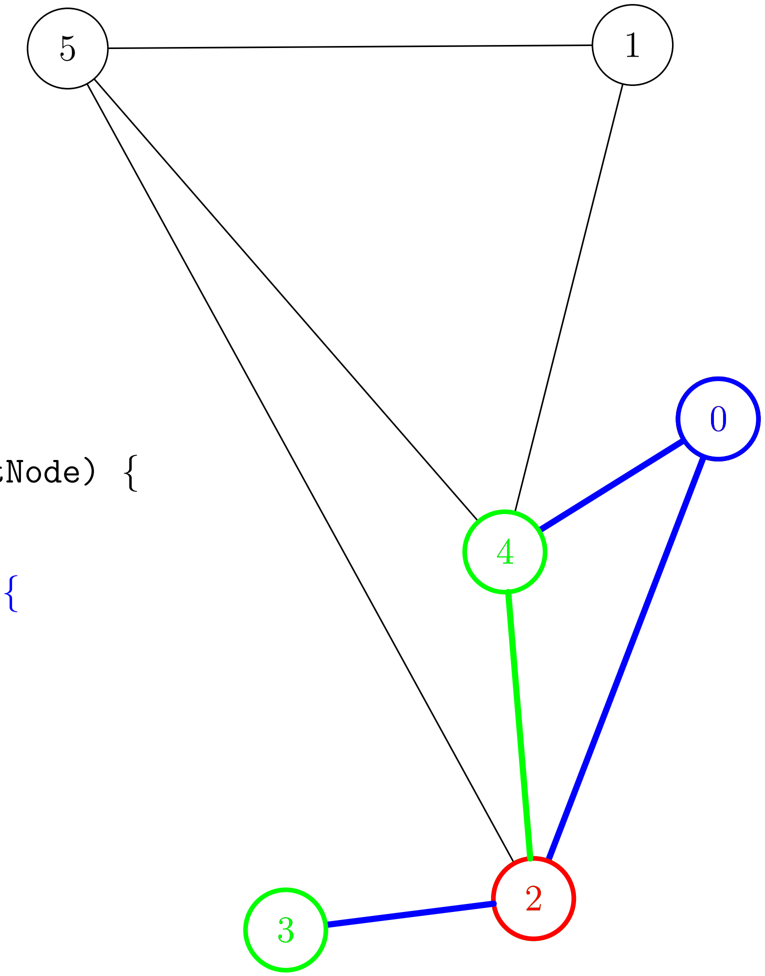


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

4	3				

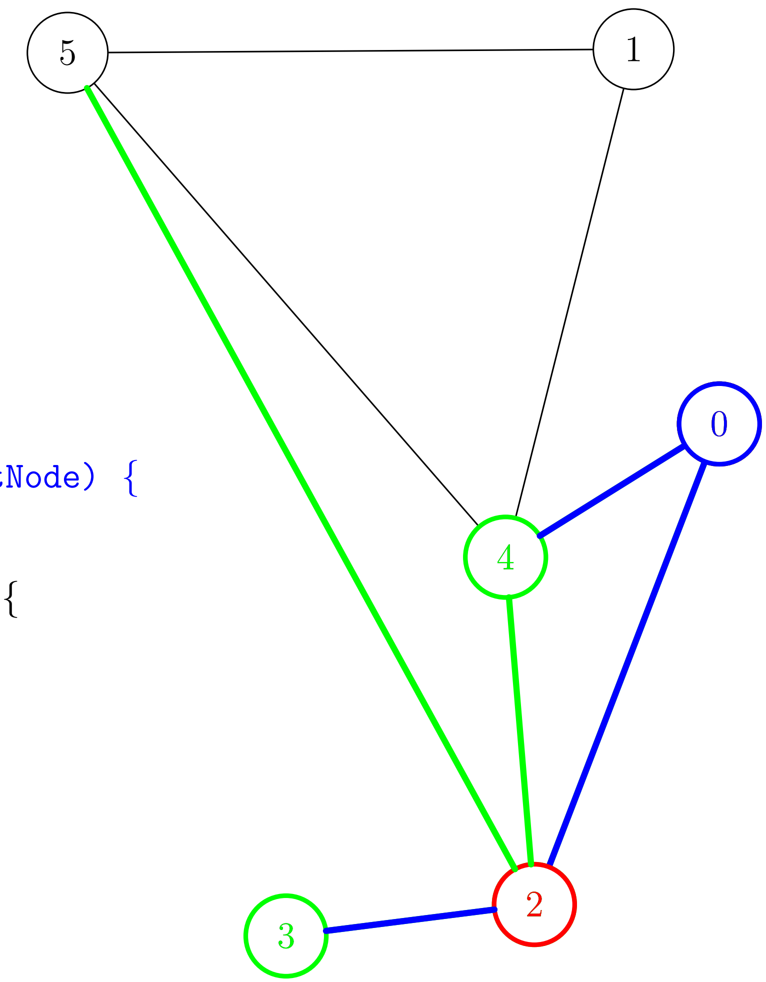


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

4	3				

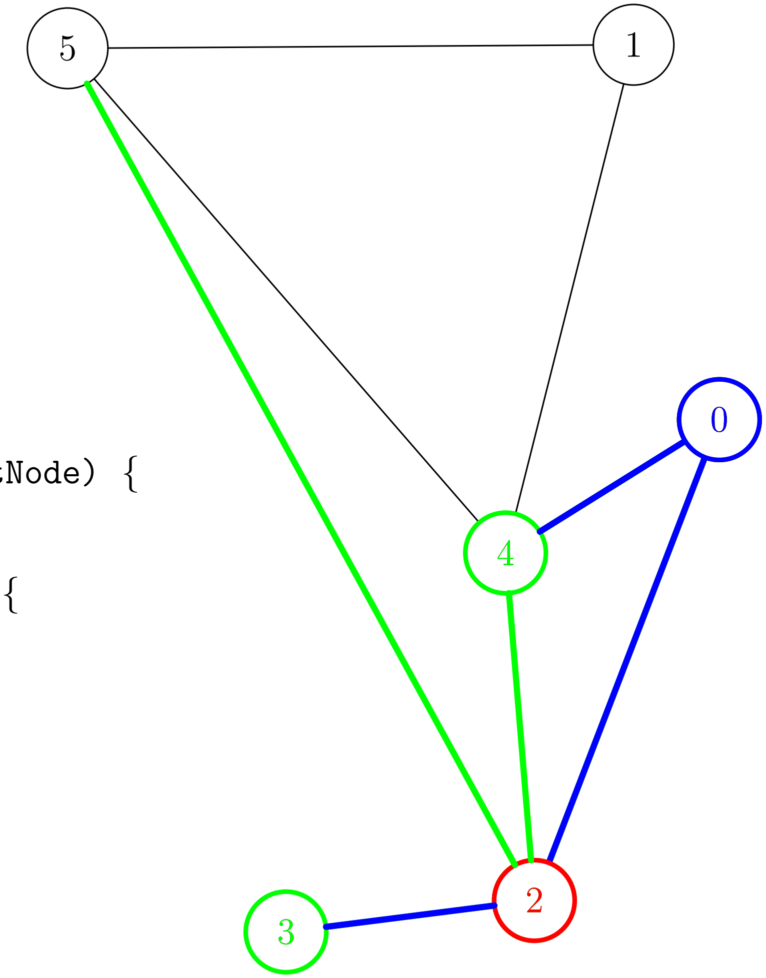


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

4	3				

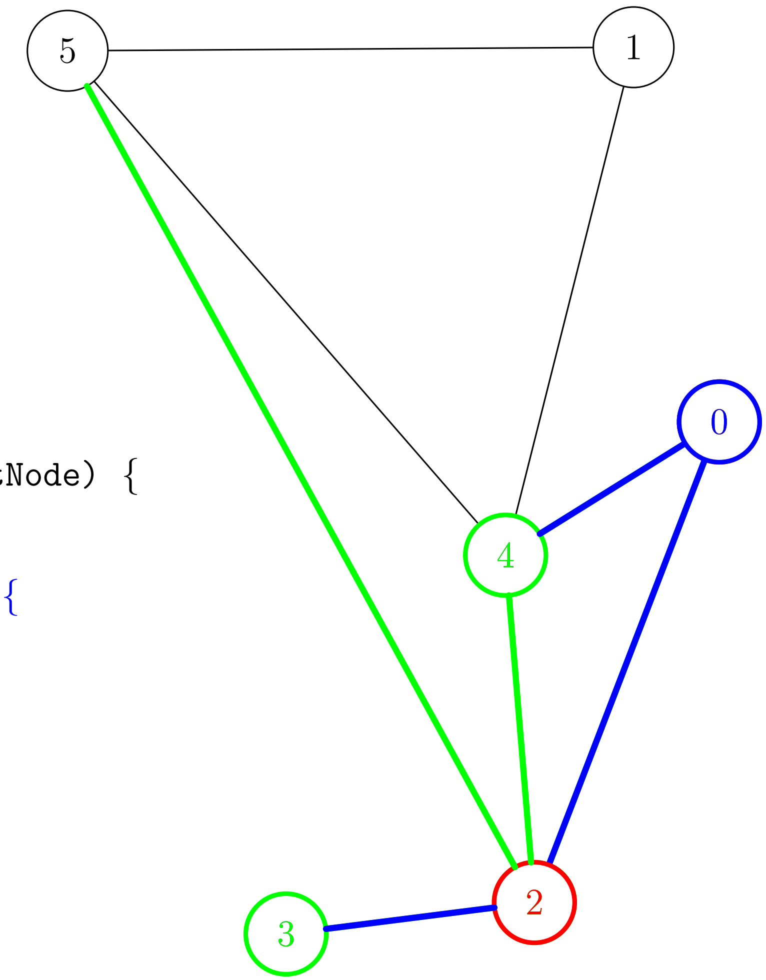


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

4	3				

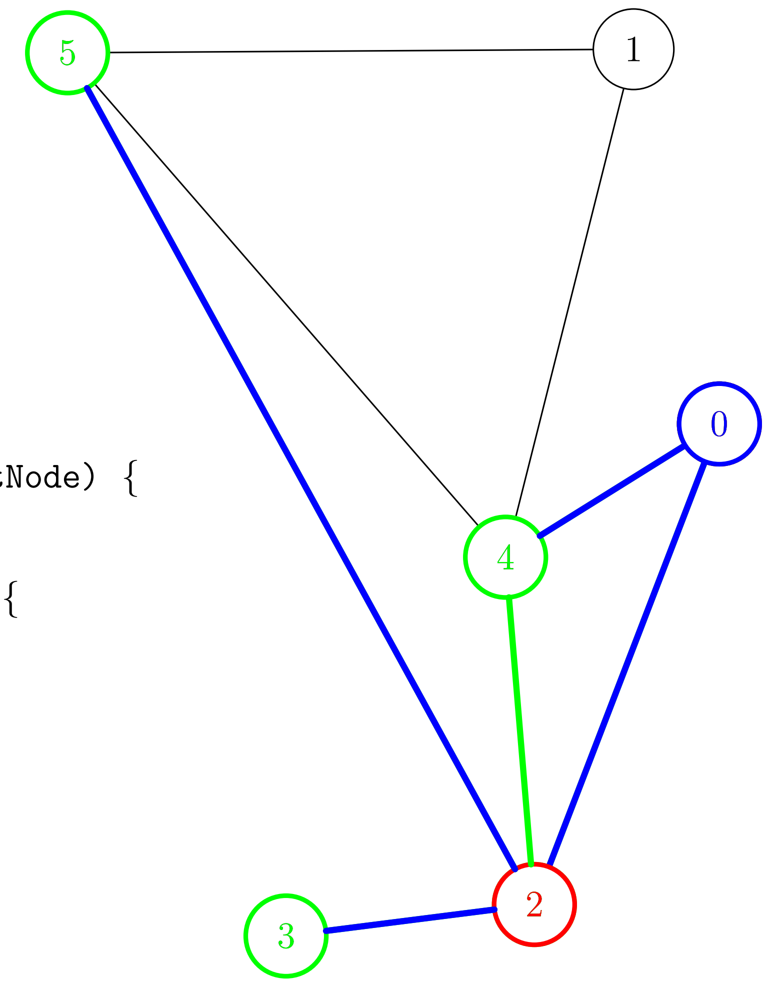


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

4	3	5			

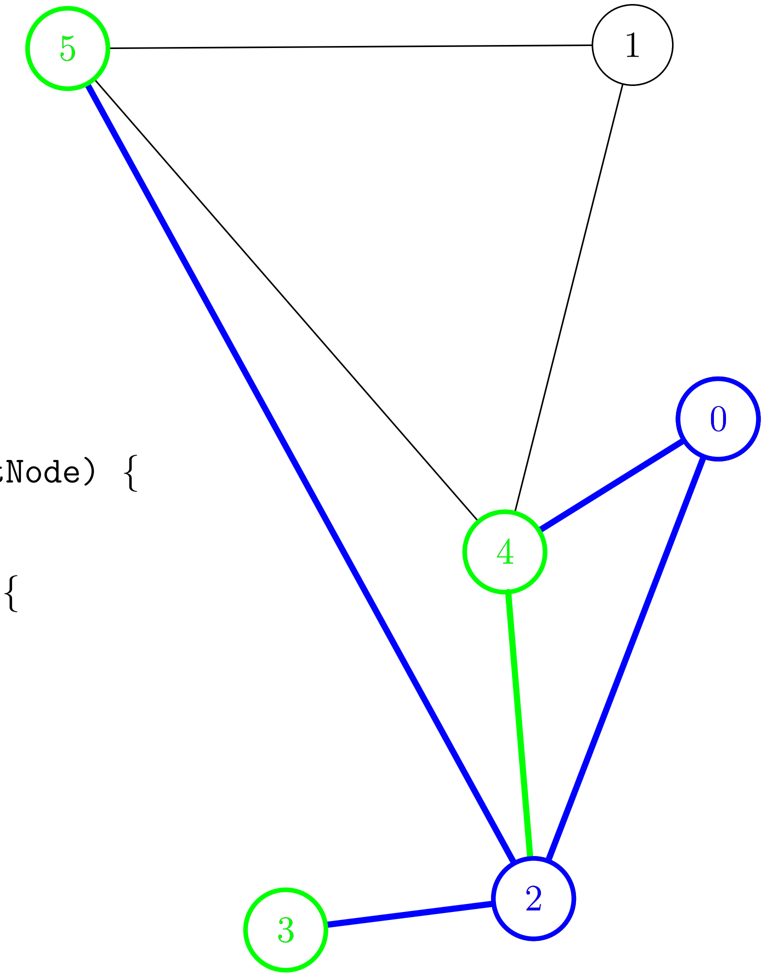


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

4	3	5			

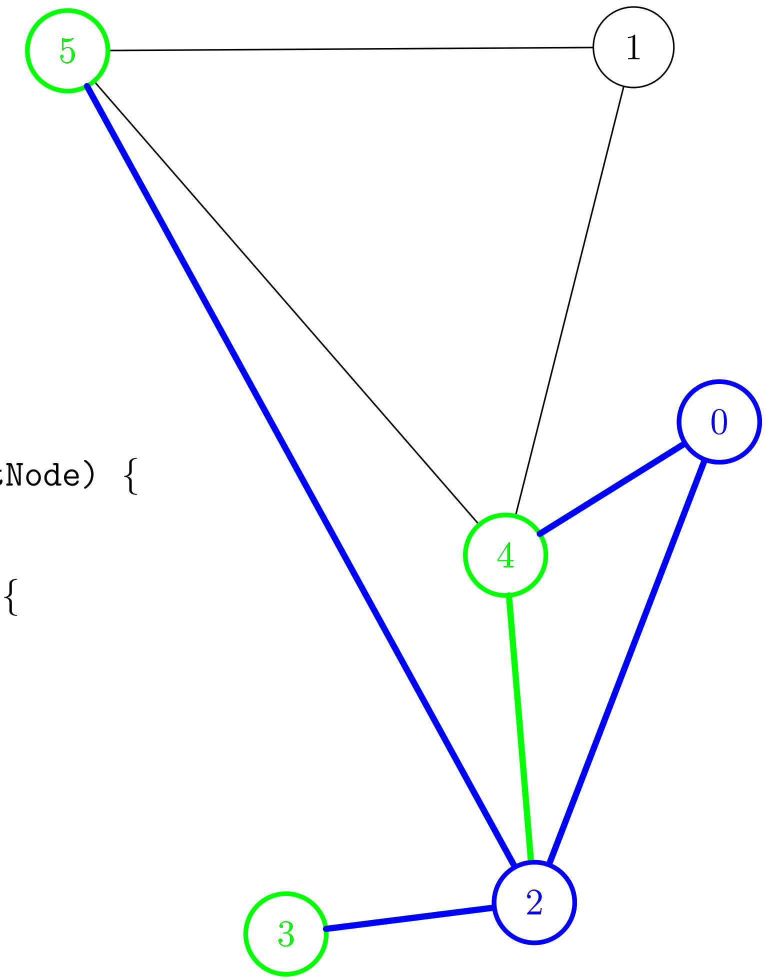


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

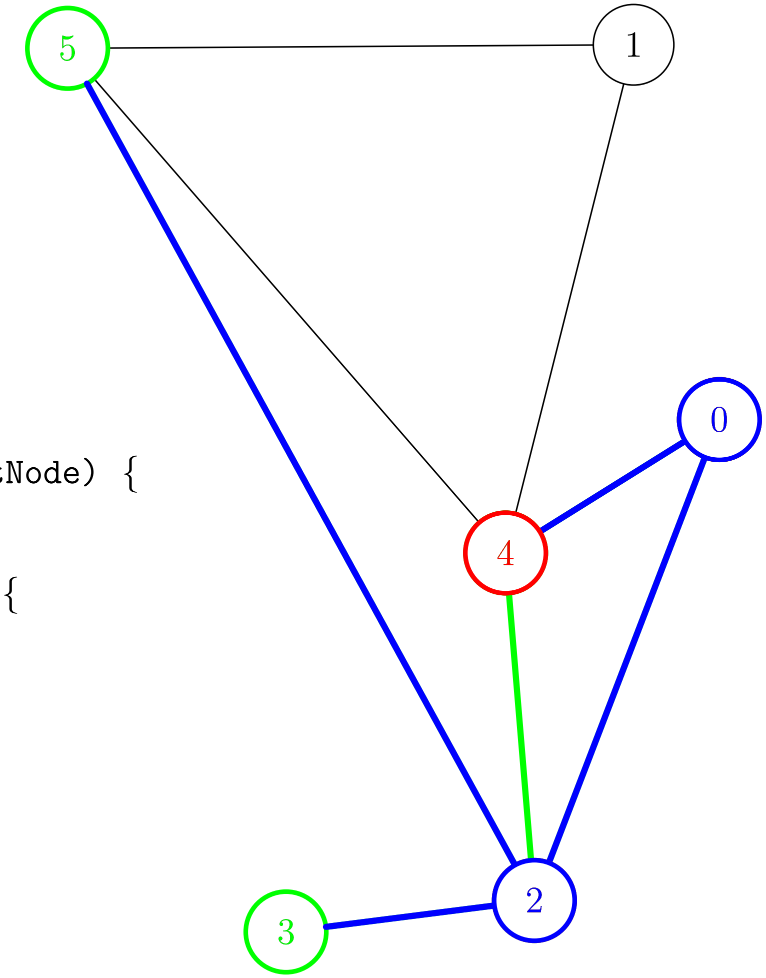
4	3	5			



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q= | 3 | 5 | | | | |

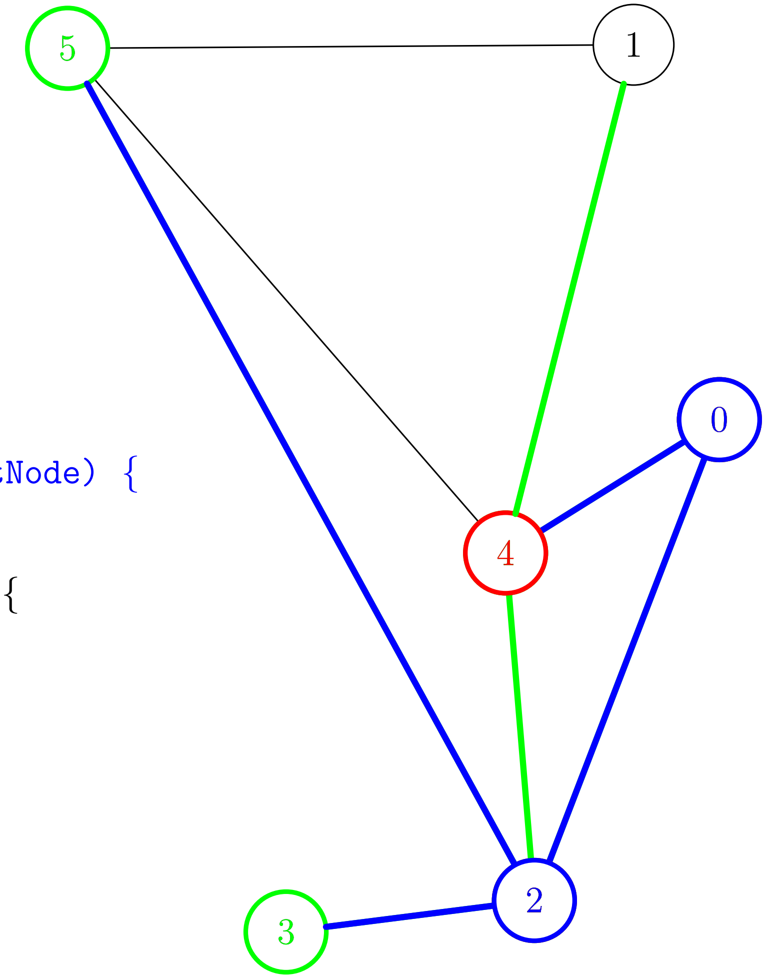


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

3	5				

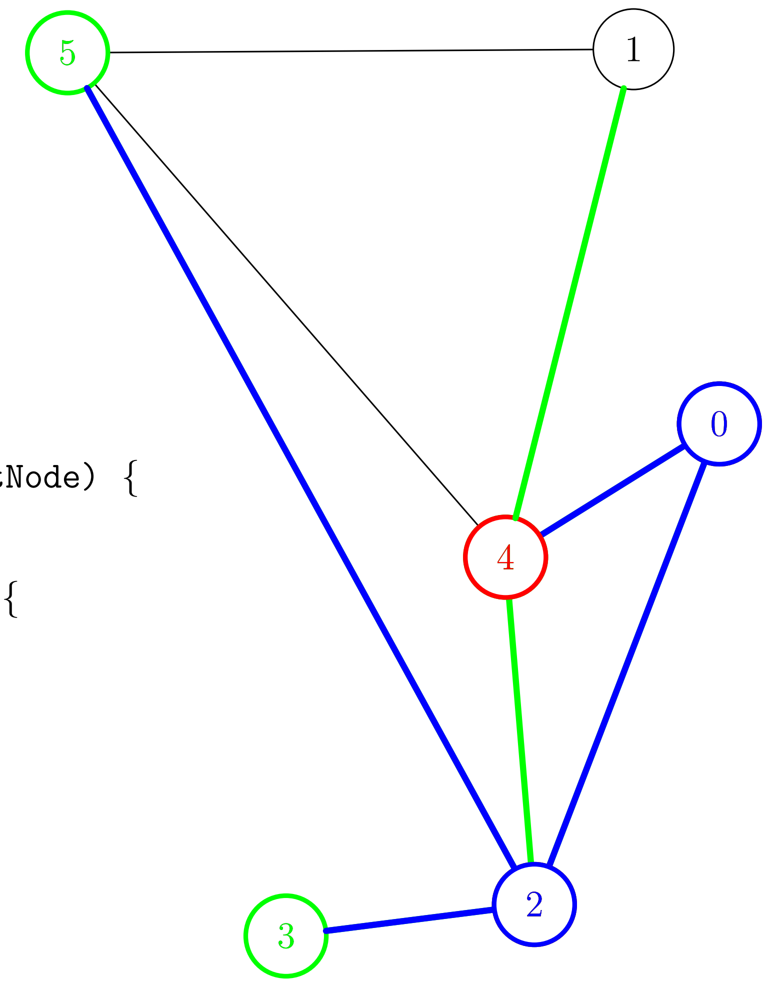


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

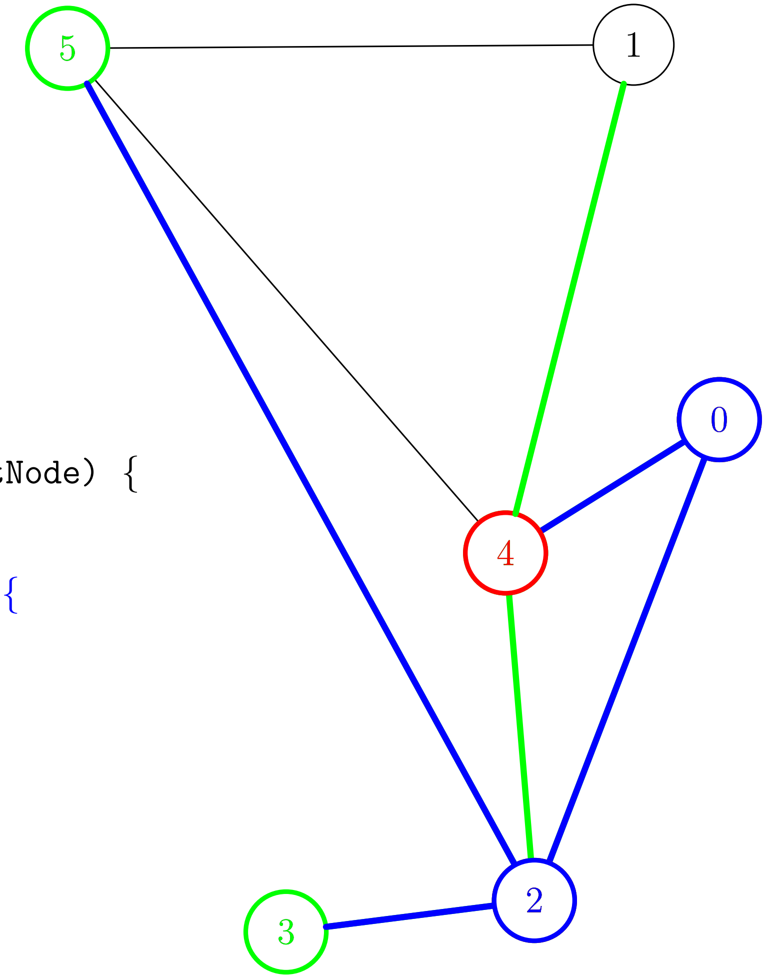
3	5				



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q= | 3 | 5 | | | | |
|---|---|---|---|---|---|

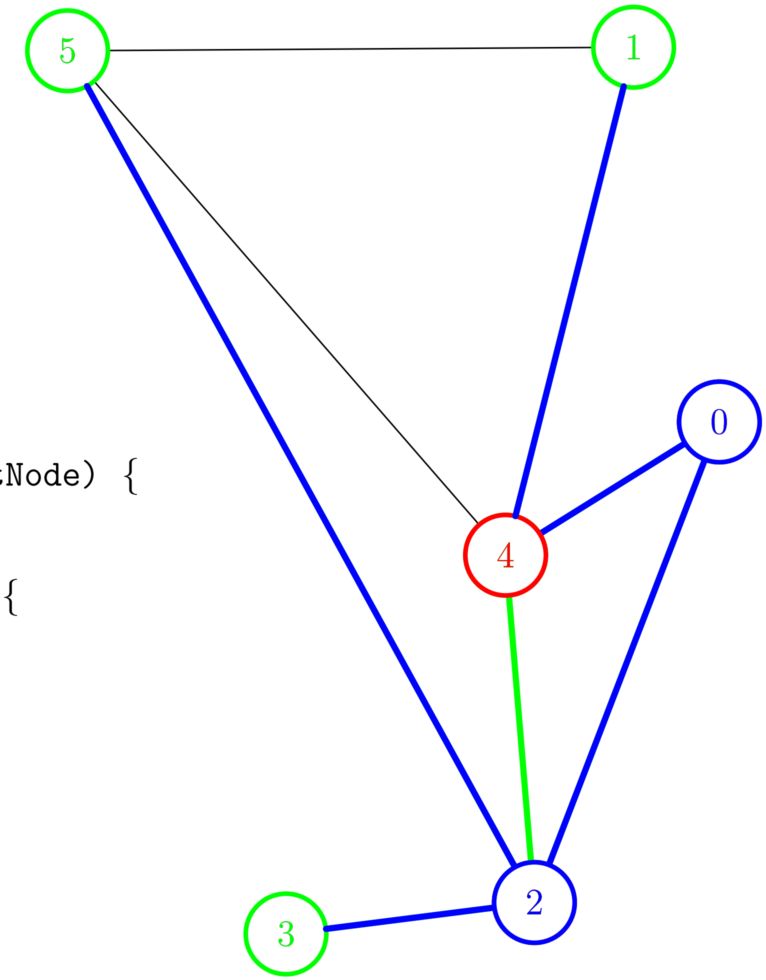


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

3	5	1			

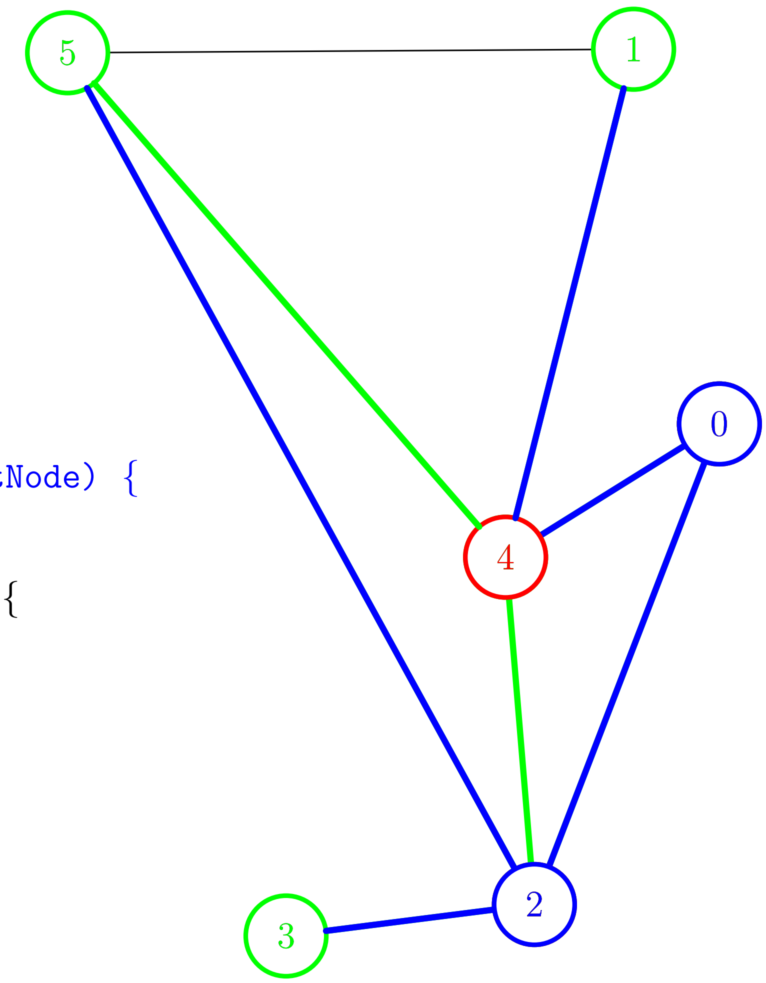


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

3	5	1			

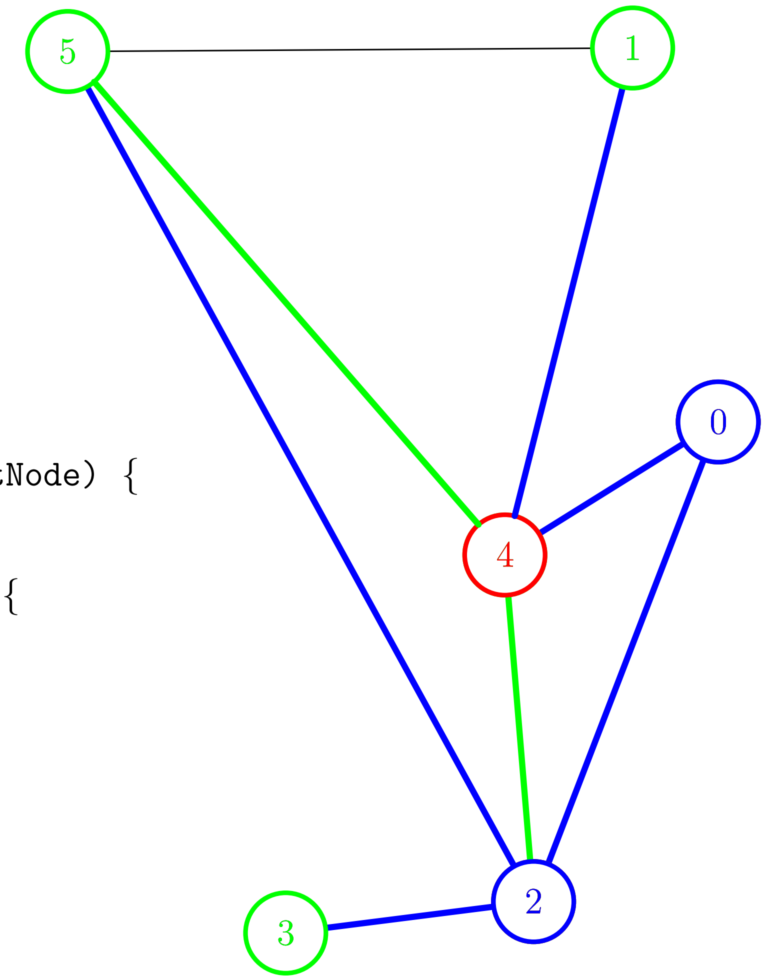


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

3	5	1			

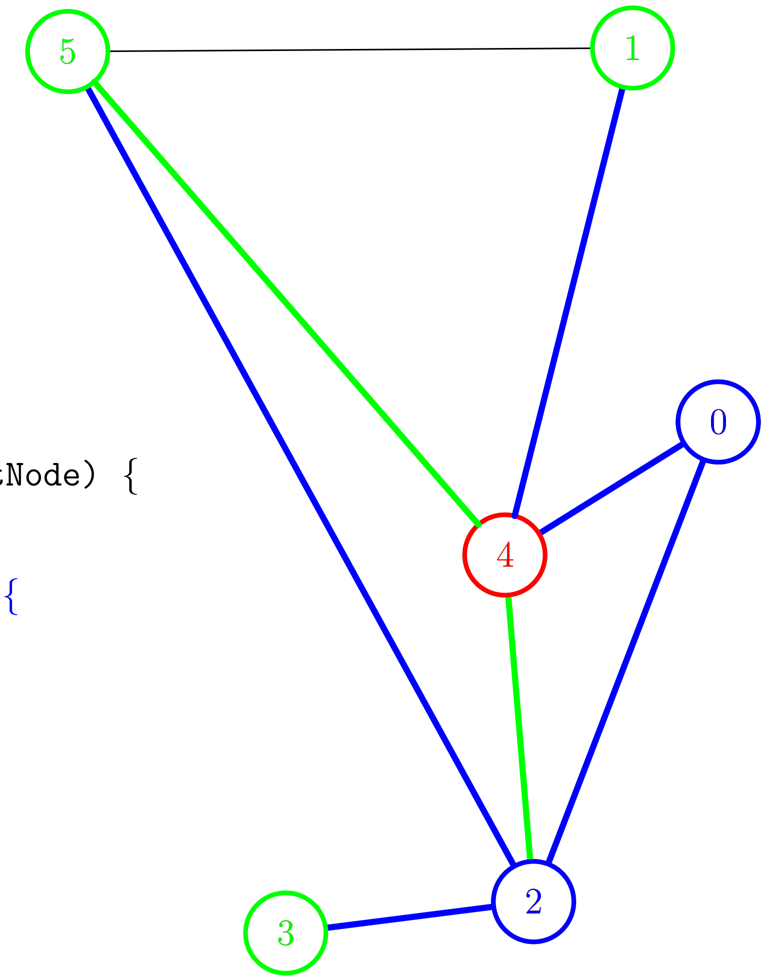


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

3	5	1			

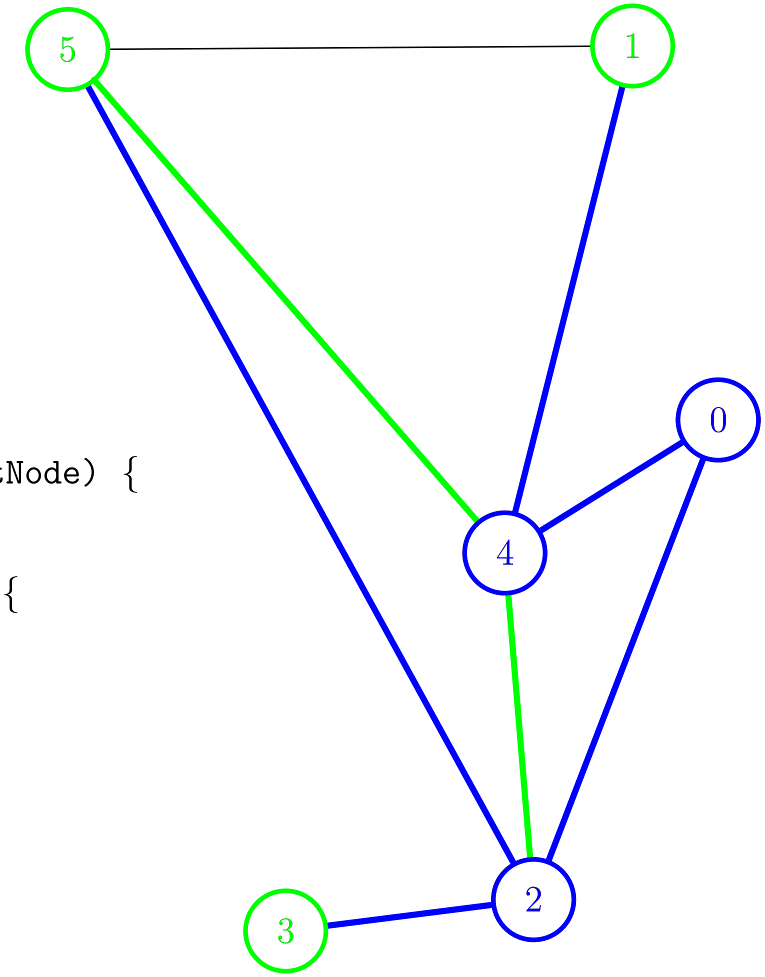


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

3	5	1			

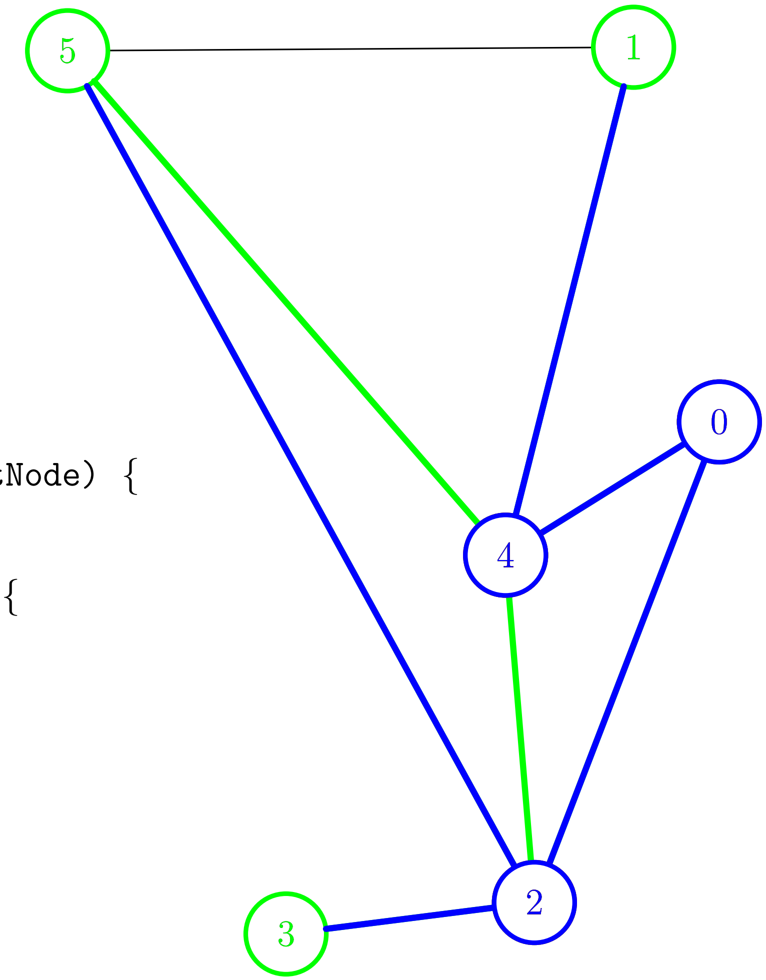


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

3	5	1			

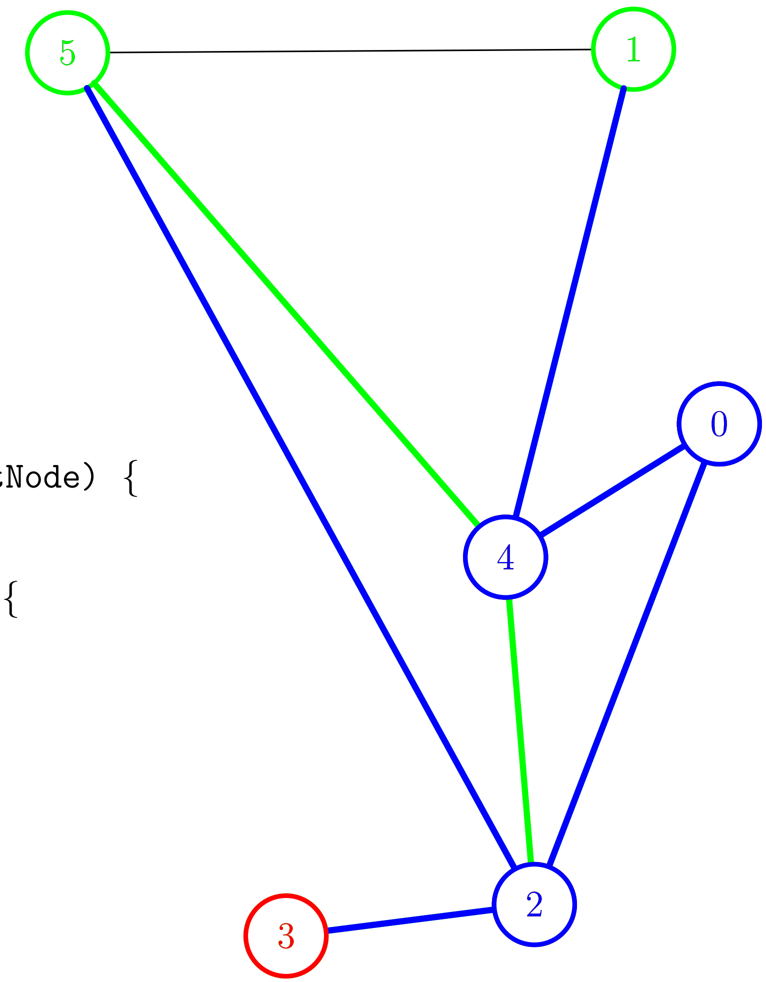


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

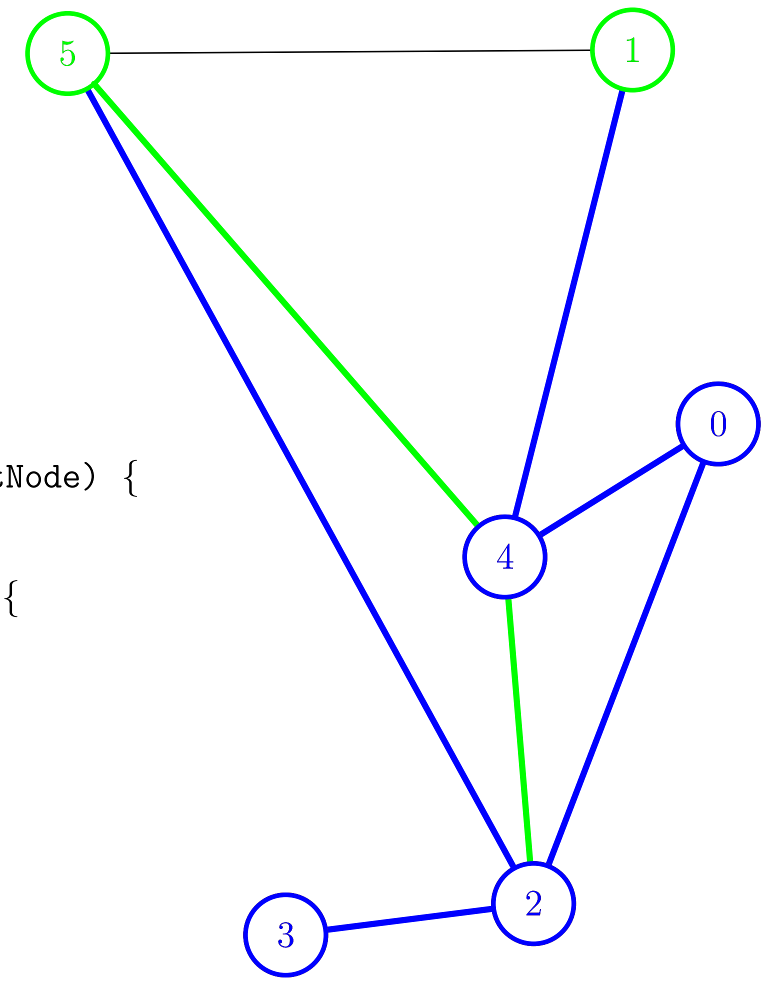
5	1				



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q= | 5 | 1 | | | | |

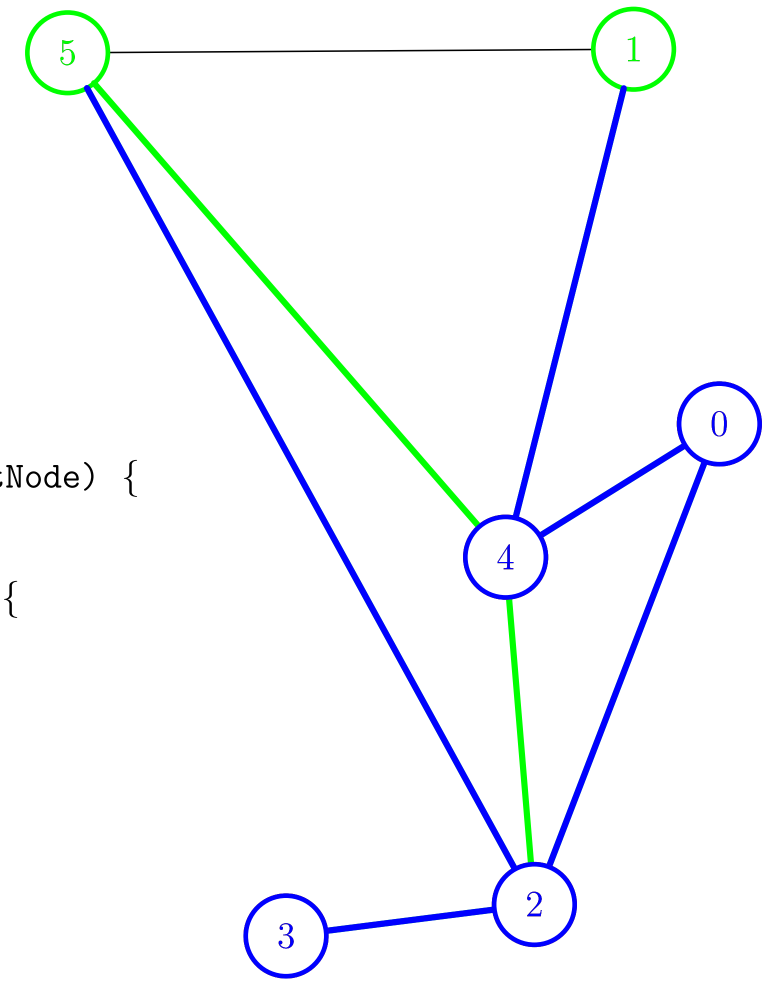


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

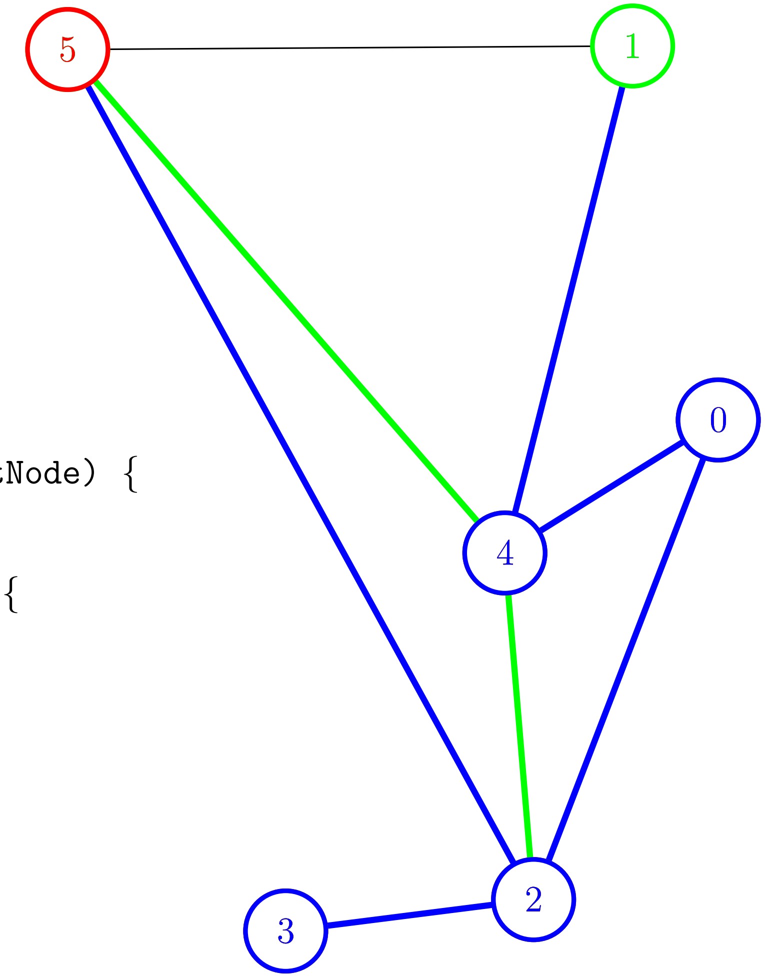
5	1				



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q= | 1 | | | | | |
|---|---|---|---|---|---|

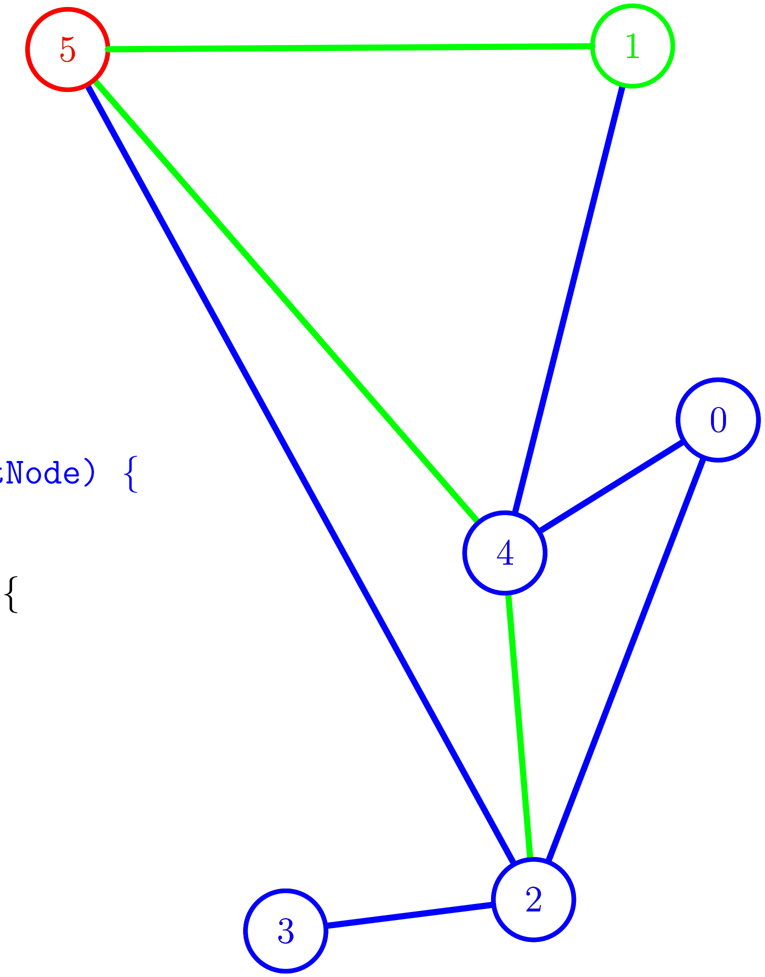


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

1					

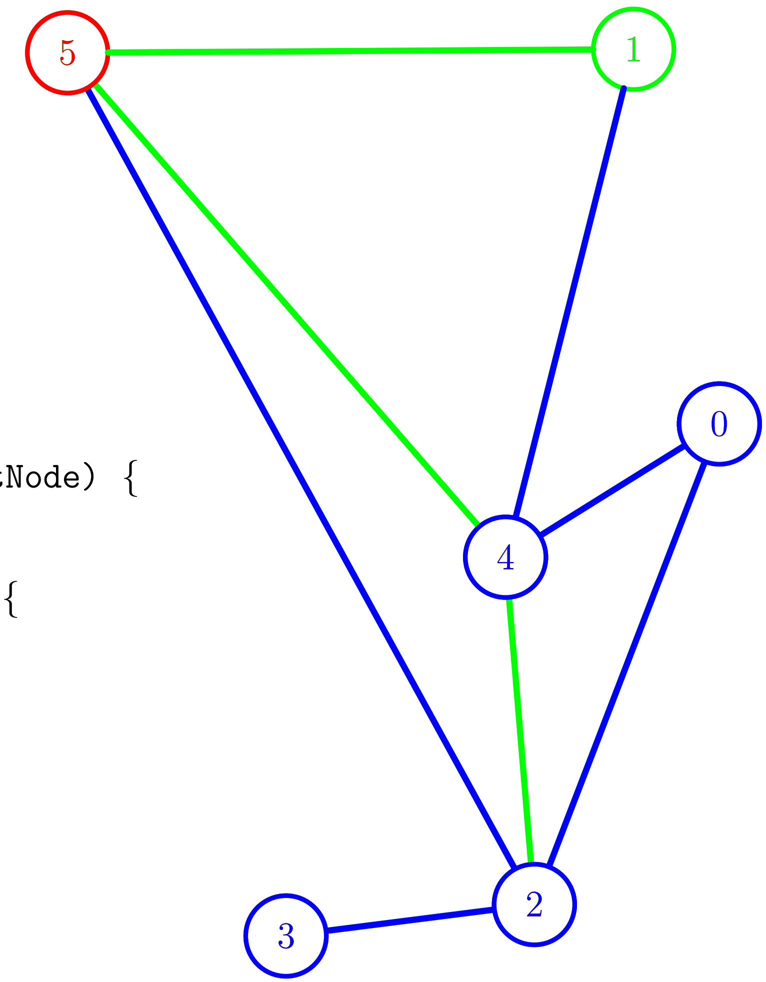


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

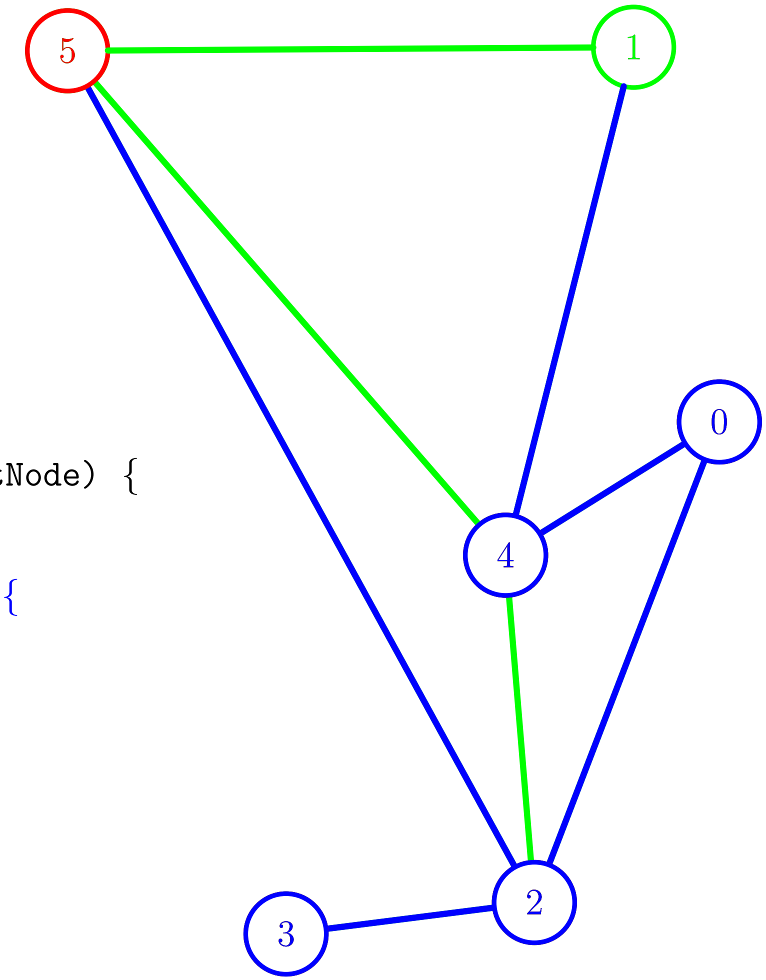
1					



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

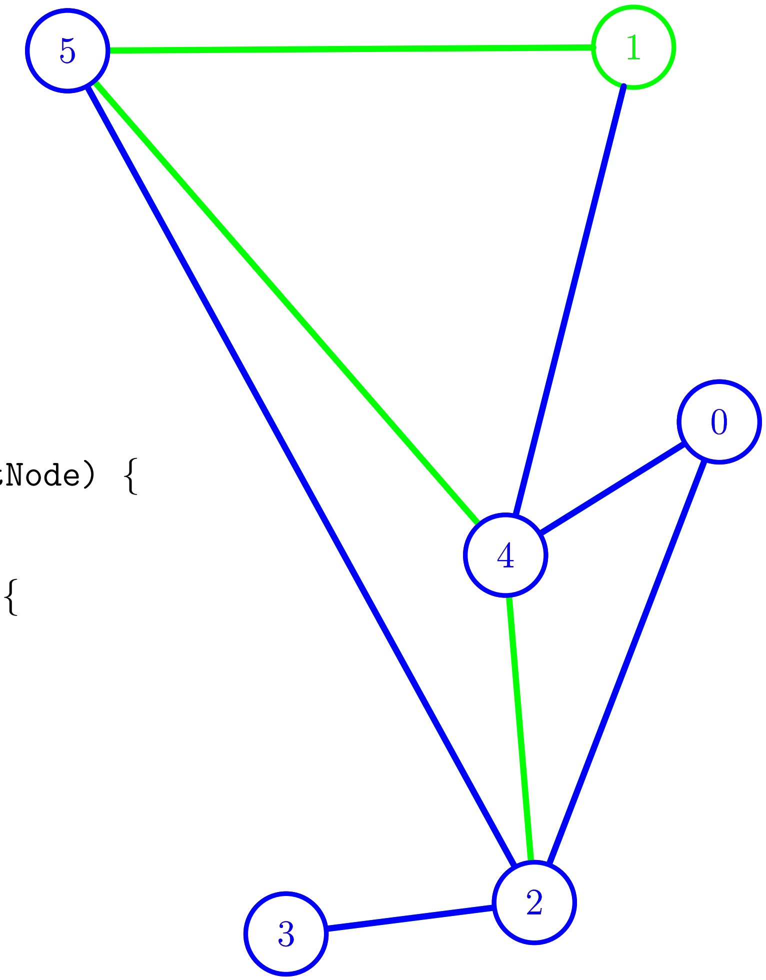
q= | 1 | | | | | |



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=	1					

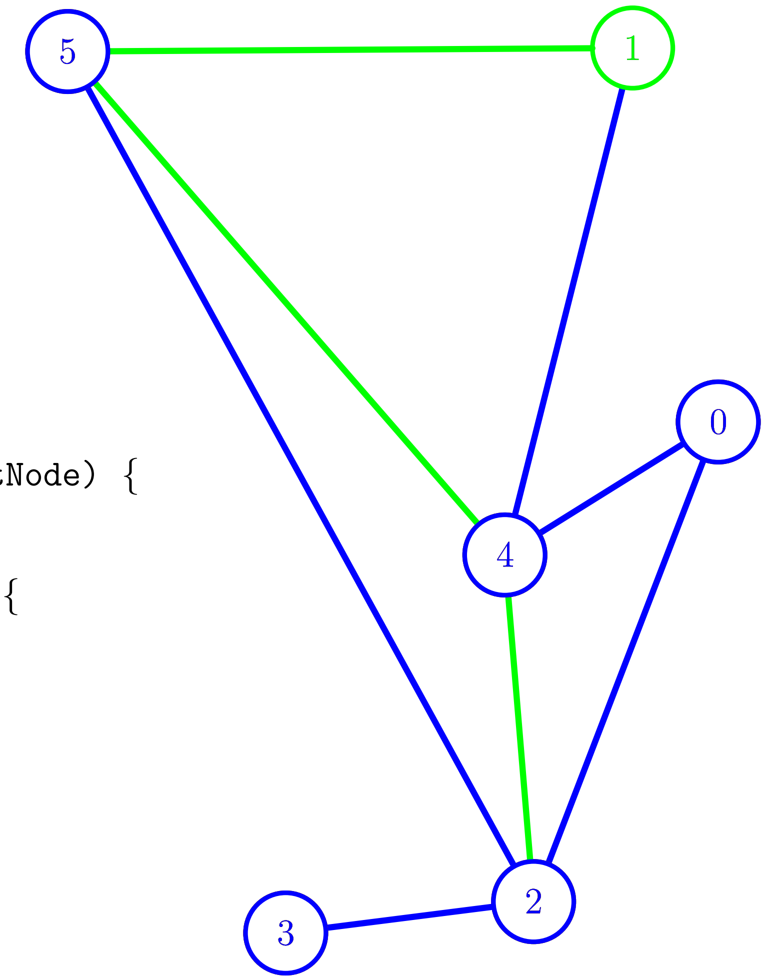


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=

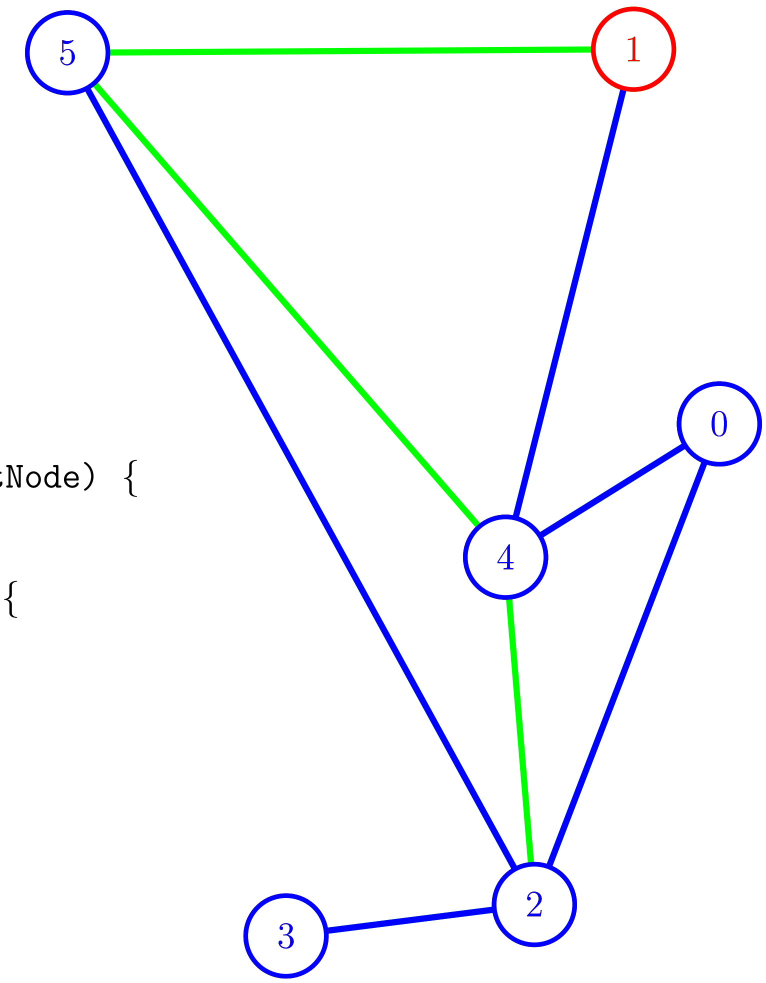
1					



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

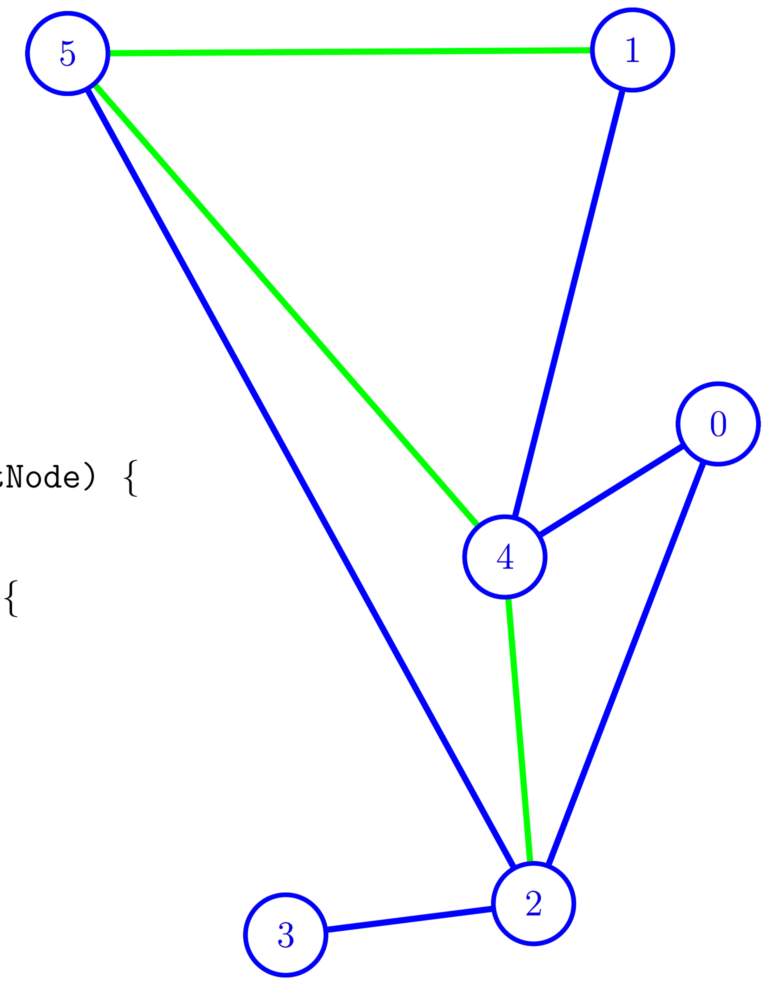
q=



Breadth First Search

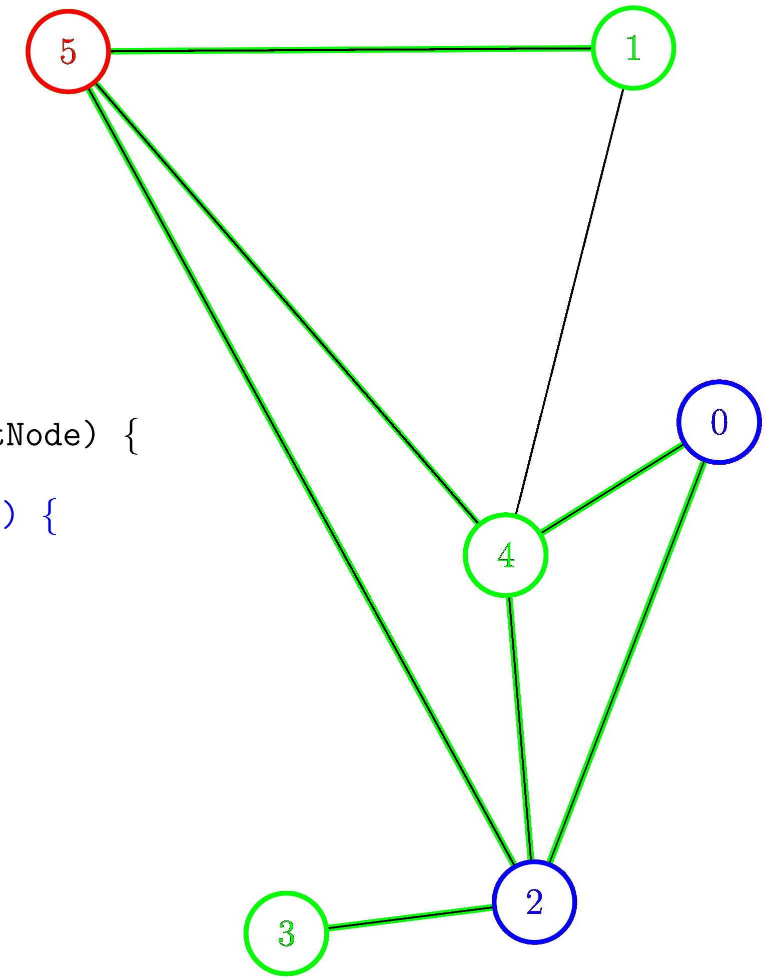
```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] = "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

q=



Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode = q.dequeue()
    processVertexEarly(currentNode)
    foreach neighbour ∈ Neighbourhood(currentNode) {
      processEdge(currentNode, neighbour)
      if (state[currentNode] = "undiscovered") {
        state[neighbour] = "discovered"
        parent[neighbour] = currentNode
        q.enqueue(neighbour)
      }
    }
  processVertexLate(currentNode)
  }
}
```



q=						
	5	1				

Applications of Breadth First Search

- Breadth first search can be used to find the shortest path from a source node to a destination node for an **unweighted** graph
  - ⋆ Run `bfs(graph,source)`
  - ⋆ Use parent information to find path from destination back to source
- BFS (as well as DFS) can be used to find connected components
  - ⋆ Use $processVertexEarly$ to mark vertices connected to the current connected component
  - ⋆ Run bfs from all vertices that are not labelled

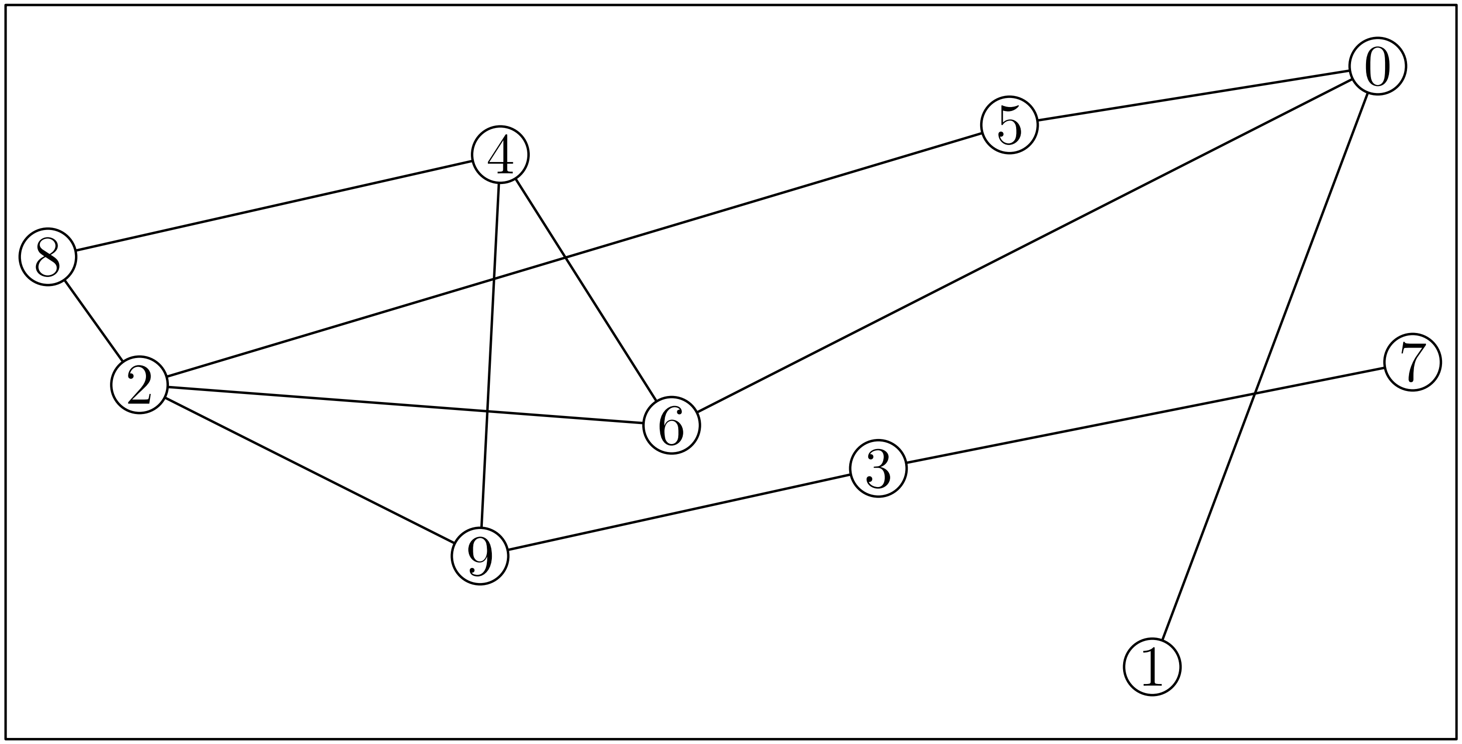
Applications of Breadth First Search

- Breadth first search can be used to find the shortest path from a source node to a destination node for an **unweighted** graph
  - ⋆ Run `bfs(graph,source)`
  - ⋆ Use parent information to find path from destination back to source

- BFS (as well as DFS) can be used to find connected components
  - ⋆ Use $processVertexEarly$ to mark vertices connected to the current connected component
  - ⋆ Run bfs from all vertices that are not labelled

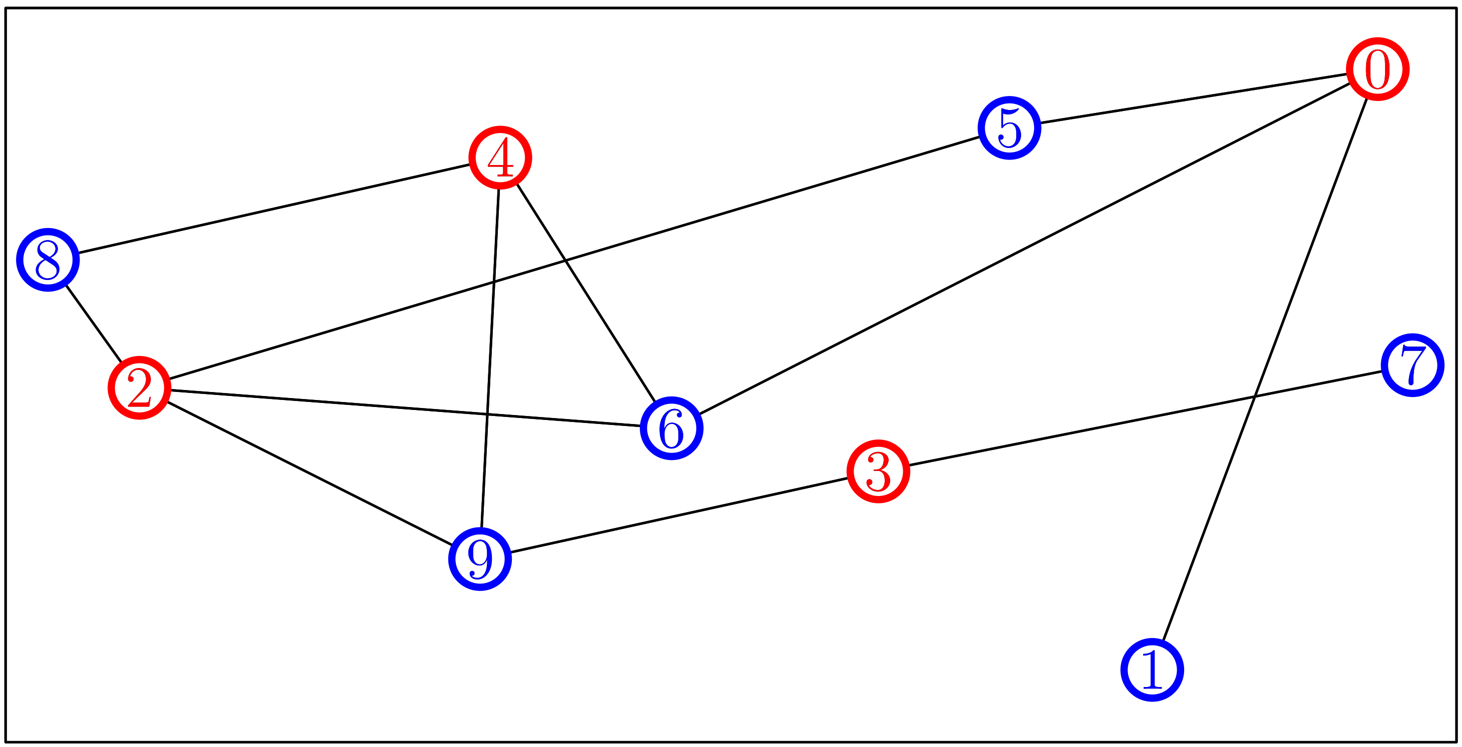
Bipartite Graphs

- Bipartite graphs are graphs where the vertices can be split into two sets so that there are no edges between vertices in the same graph



Bipartite Graphs

- Bipartite graphs are graphs where the vertices can be split into two sets so that there are no edges between vertices in the same graph



Checking Bipartiteness (Two-colourability)

- Each edge must connect nodes from different sets

```
isBipartite(graph) {
   colour = List(graph.noNodes(), "uncoloured");
   bipartite = true;

   foreach node in graph {
      if (colour[node] == "uncoloured") {
         colour[node] = white;
         bfs(graph, node);
      }
   }
   return bipartite;
}

processEdge(node1, node2) {
   if (colour[node1] == colour[node2])
       bipartite = false;
   colour[node2] = (colour[node1]=="white")? "black":"white";
}
```

Checking Bipartiteness (Two-colourability)

- Each edge must connect nodes from different sets

```
isBipartite(graph) {
   colour = List(graph.noNodes(), "uncoloured");
   bipartite = true;

   foreach node in graph {
      if (colour[node] == "uncoloured") {
         colour[node] = white;
         bfs(graph, node);
      }
   }
   return bipartite;
}

processEdge(node1, node2) {
   if (colour[node1] == colour[node2])
       bipartite = false;
   colour[node2] = (colour[node1]=="white")? "black":"white";
}
```

Checking Bipartiteness (Two-colourability)

- Each edge must connect nodes from different sets

```
isBipartite(graph) {
   colour = List(graph.noNodes(), "uncoloured");
   bipartite = true;

   foreach node in graph {
      if (colour[node] == "uncoloured") {
         colour[node] = white;
         bfs(graph, node);
      }
   }
   return bipartite;
}

processEdge(node1, node2) {
   if (colour[node1] == colour[node2])
       bipartite = false;
   colour[node2] = (colour[node1]=="white")? "black":"white";
}
```

Checking Bipartiteness (Two-colourability)

- Each edge must connect nodes from different sets

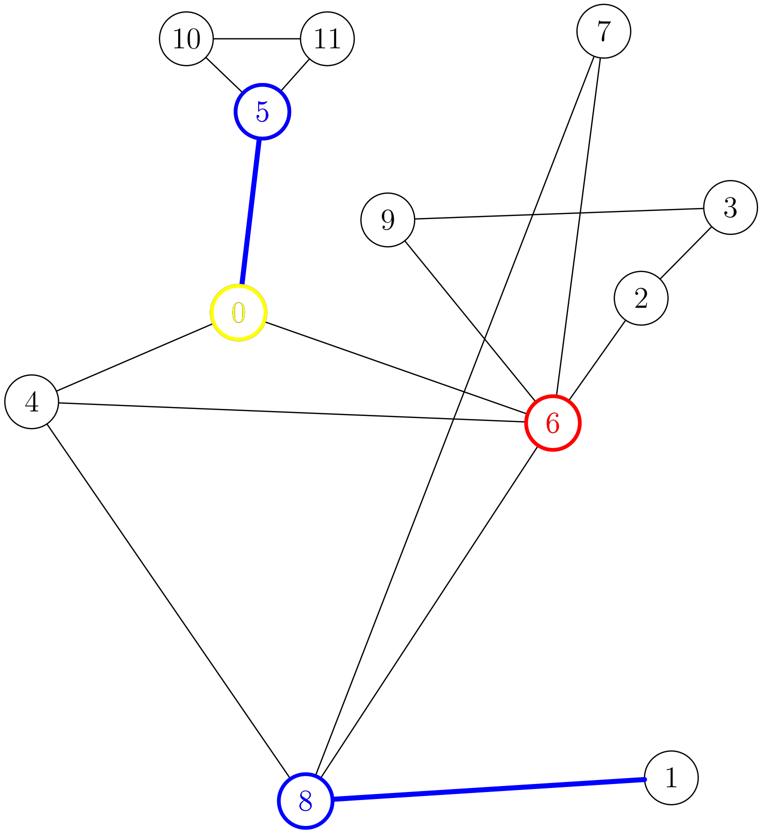
```
isBipartite(graph) {
   colour = List(graph.noNodes(), "uncoloured");
   bipartite = true;

   foreach node in graph {
      if (colour[node] == "uncoloured") {
         colour[node] = white;
         bfs(graph, node);
      }
   }
   return bipartite;
}

processEdge(node1, node2) {
   if (colour[node1] == colour[node2])
       bipartite = false;
   colour[node2] = (colour[node1]=="white")? "black":"white";
}
```

Outline

1. Breadth First Search
   - BFS applications
2. **Depth First Search**
   - DFS applications
3. Topological Sort



Depth First Search

- Depth first search is essentially like breadth first search except we replace the queue by a stack
- In practice it is often implemented using recursion rather than a stack
- It proves useful to keep a record of the traversal **time** for each vertex
  - ⋆ the clock ticks each time a vertex is entered or exited

Depth First Search

- Depth first search is essentially like breadth first search except we replace the queue by a stack
- In practice it is often implemented using recursion rather than a stack
- It proves useful to keep a record of the traversal **time** for each vertex
  - the clock ticks each time a vertex is entered or exited

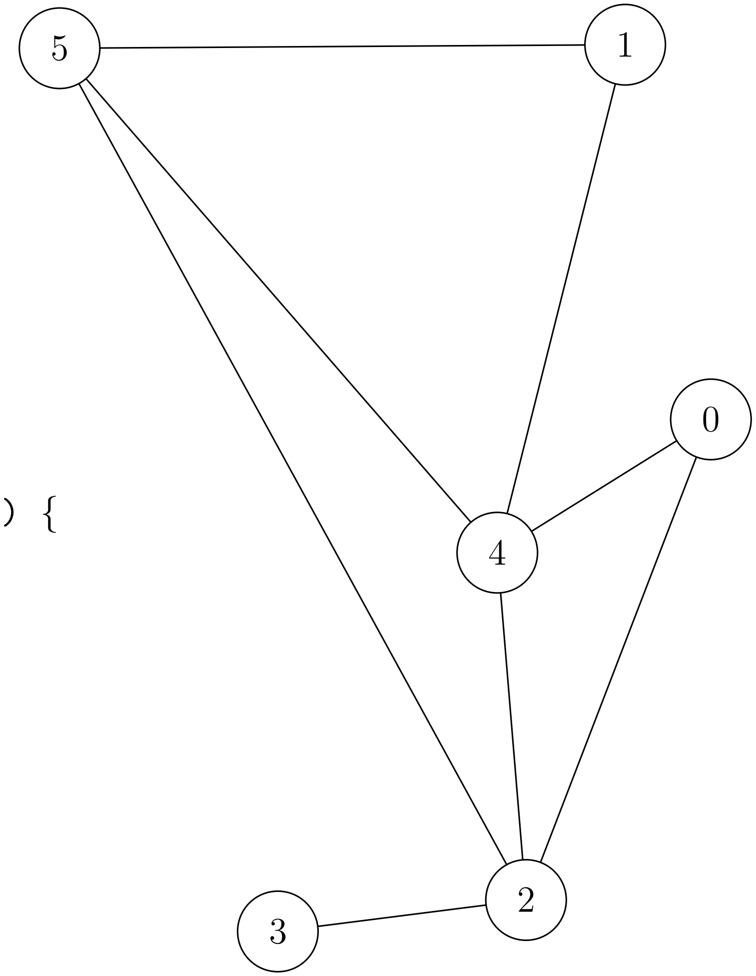
Depth First Search

- Depth first search is essentially like breadth first search except we replace the queue by a stack
- In practice it is often implemented using recursion rather than a stack
- It proves useful to keep a record of the traversal **time** for each vertex
  - ⋆ the clock ticks each time a vertex is entered or exited

Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

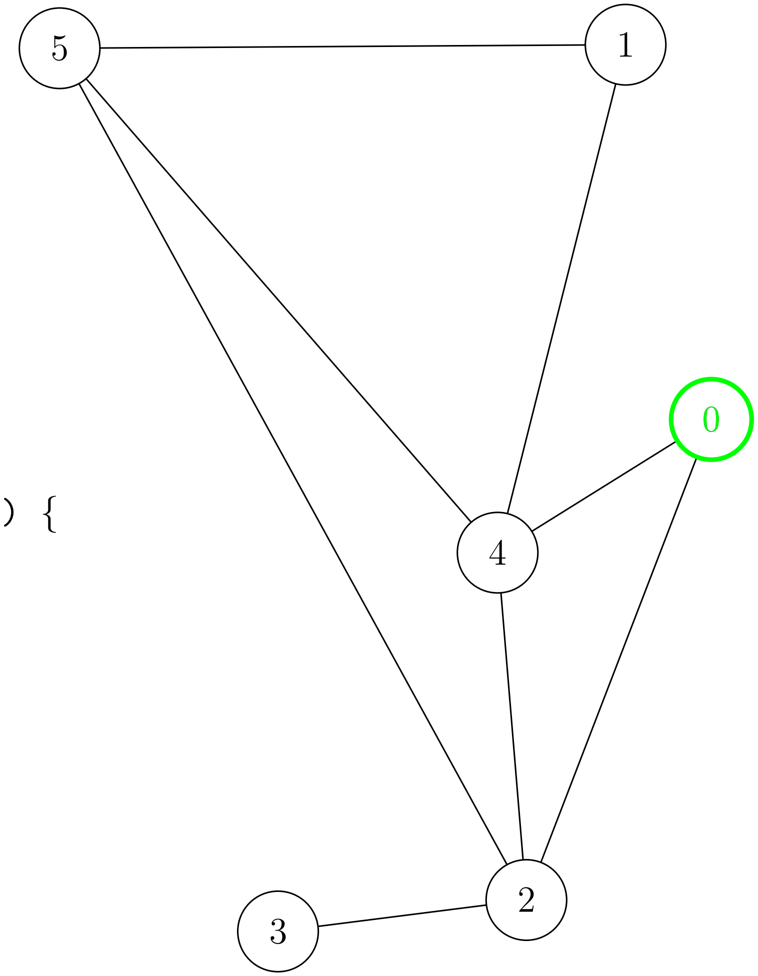
node=0 neighbour=nil time=0



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

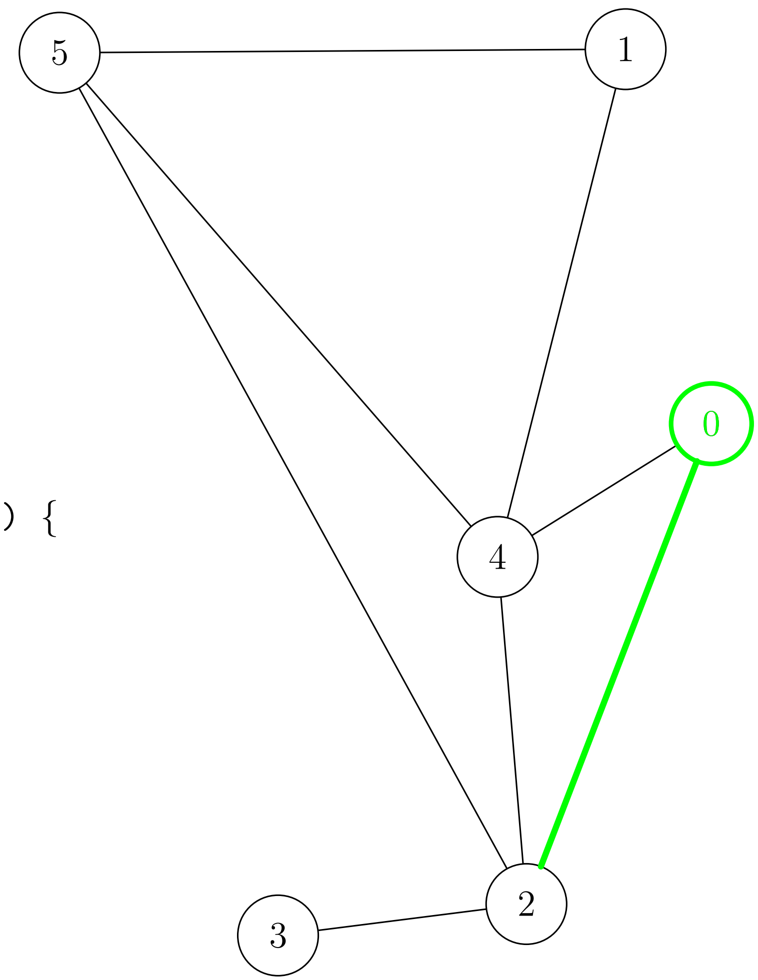
node=0   neighbour=nil   time=1



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

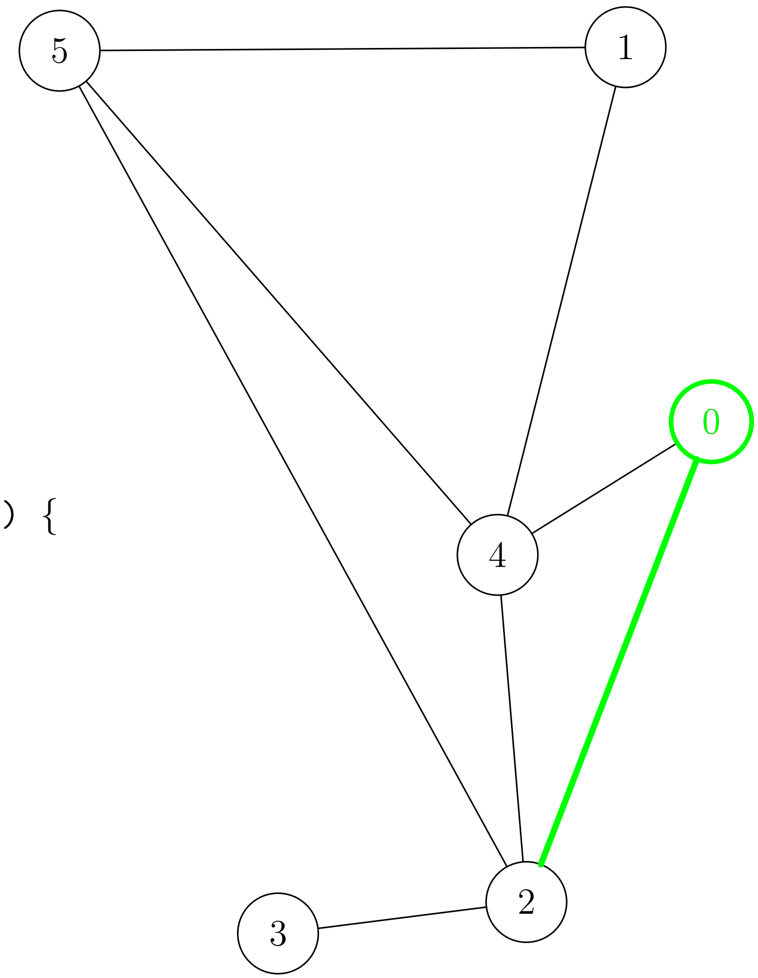
node=0   neighbour=2   time=1



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

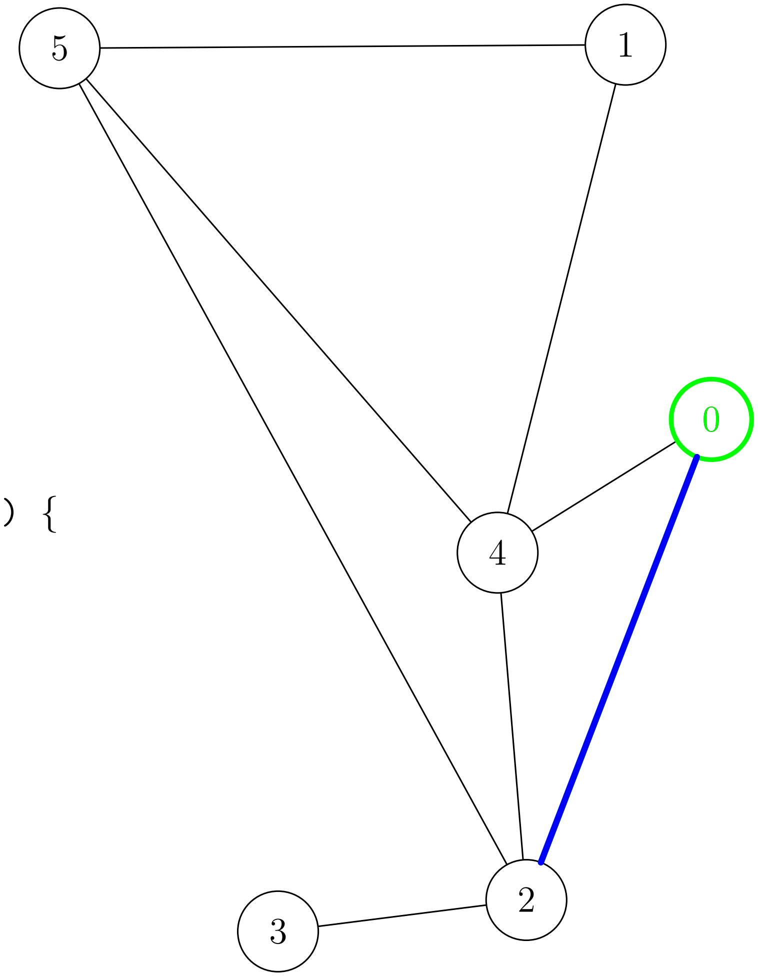
node=0 neighbour=2 time=1



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

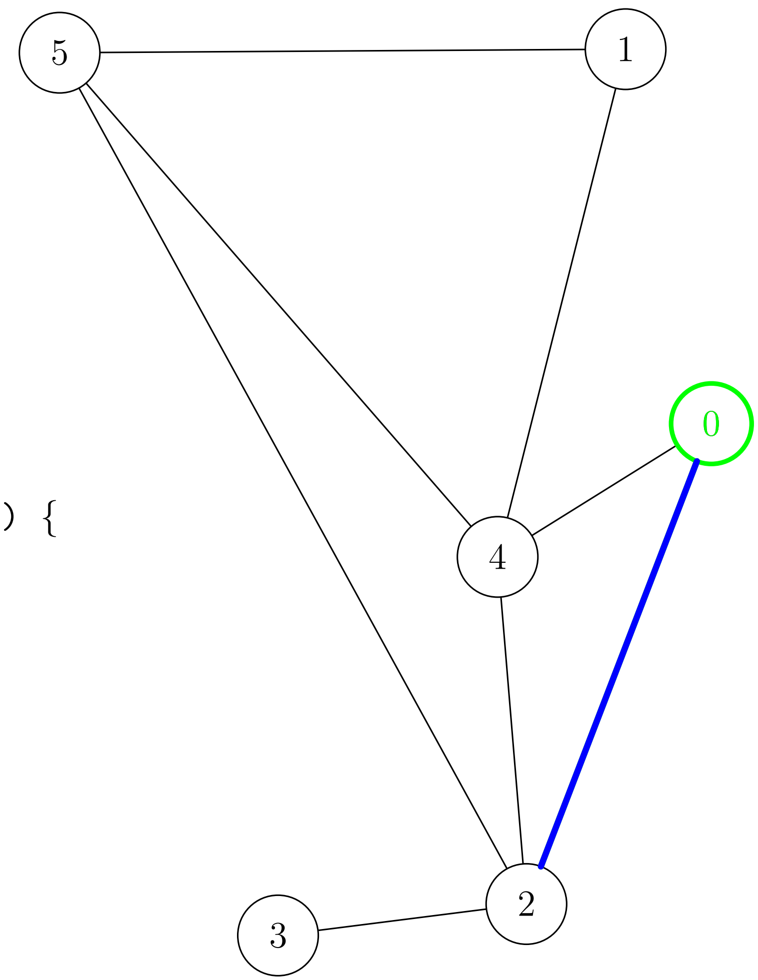
node=0 neighbour=2 time=1



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

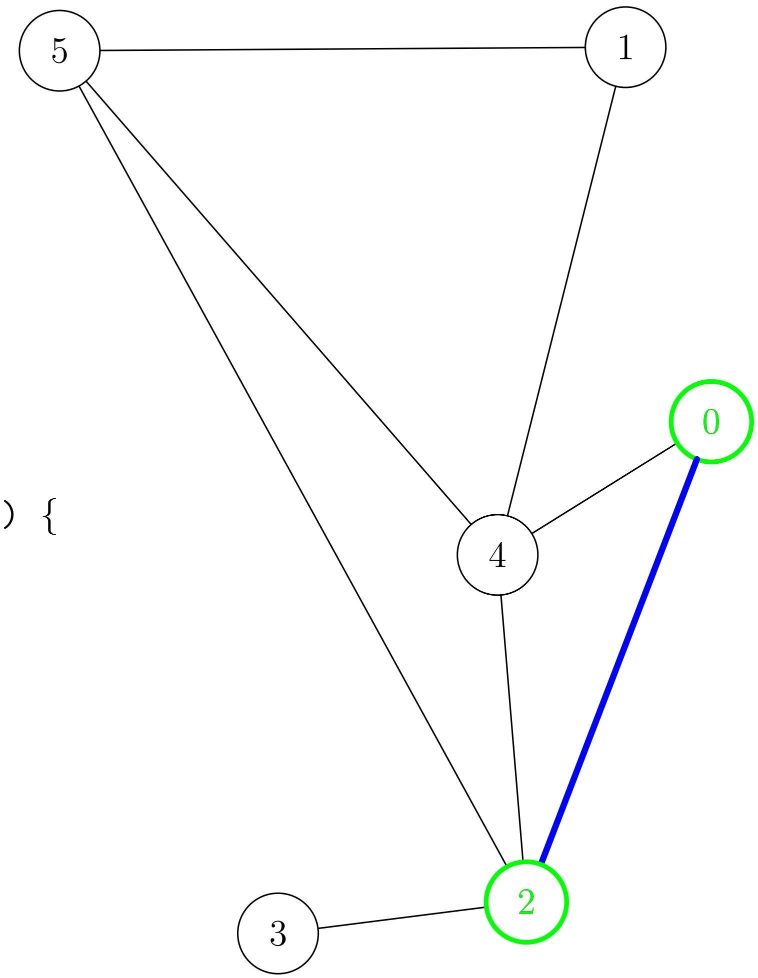
node=0   neighbour=2   time=1



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

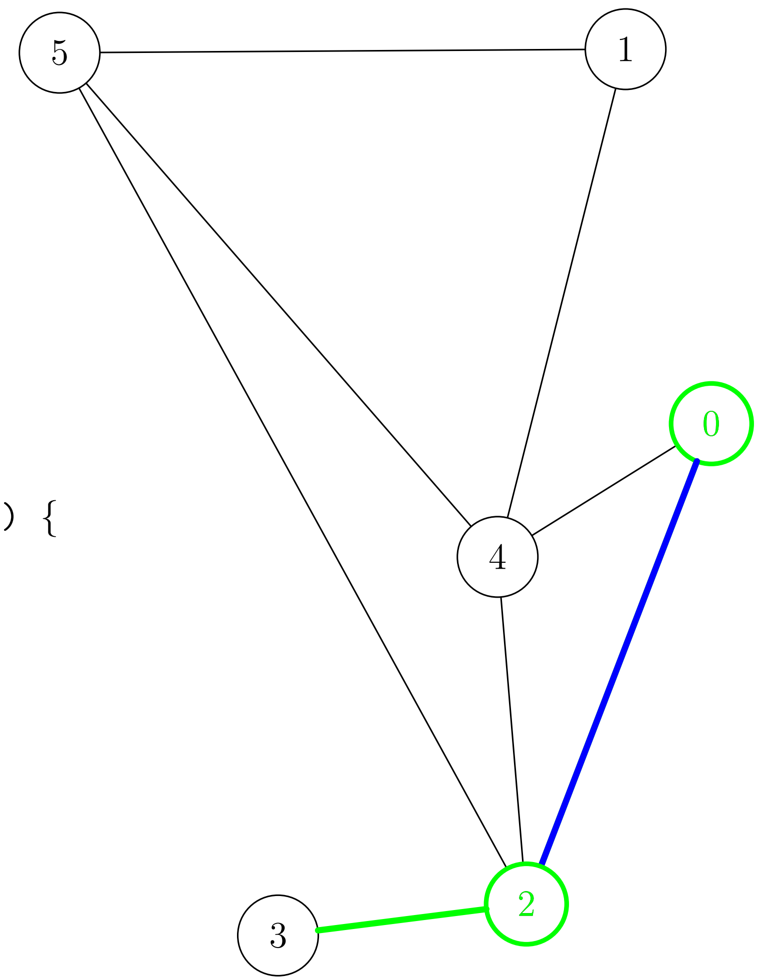
node=2 neighbour=nil time=2



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

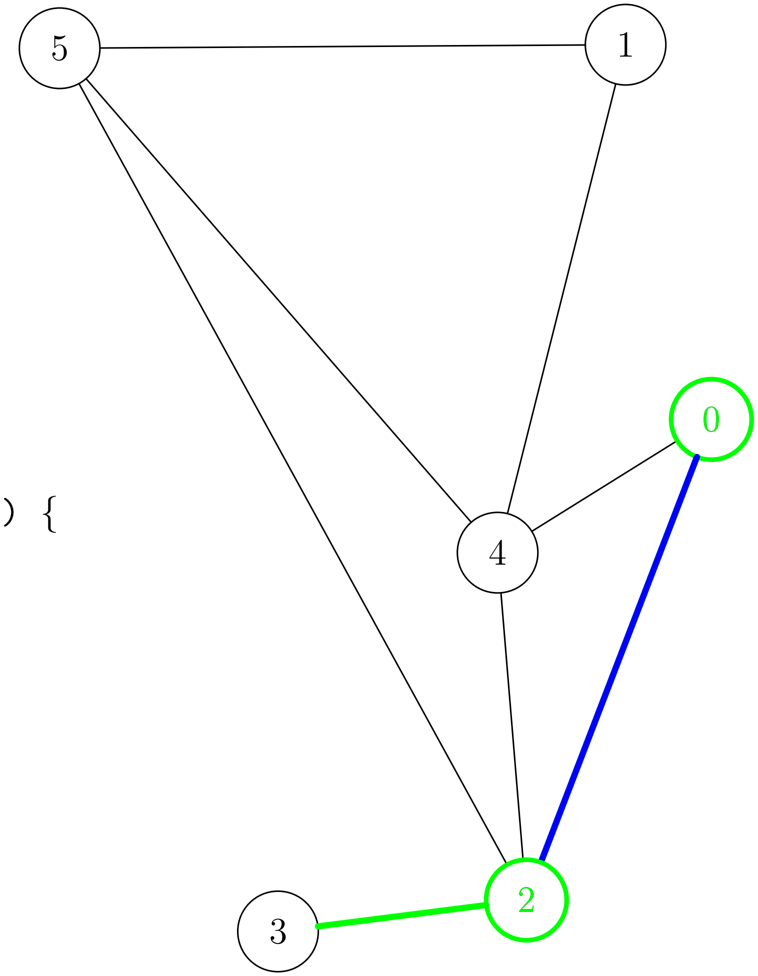
node=2   neighbour=3   time=2



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

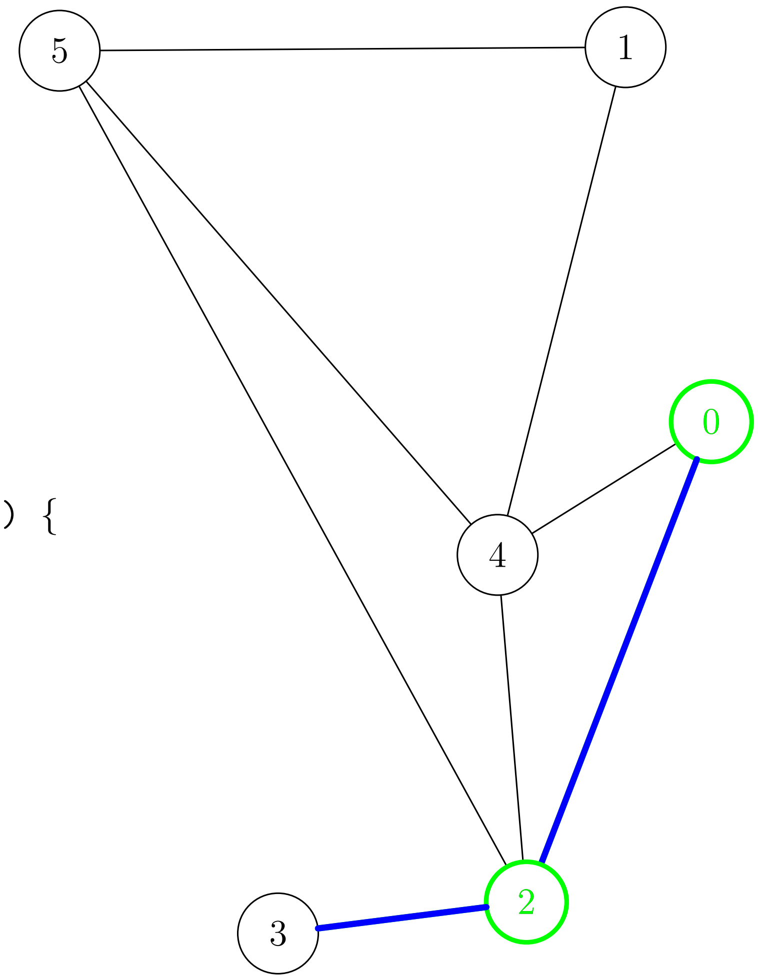
node=2   neighbour=3   time=2



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

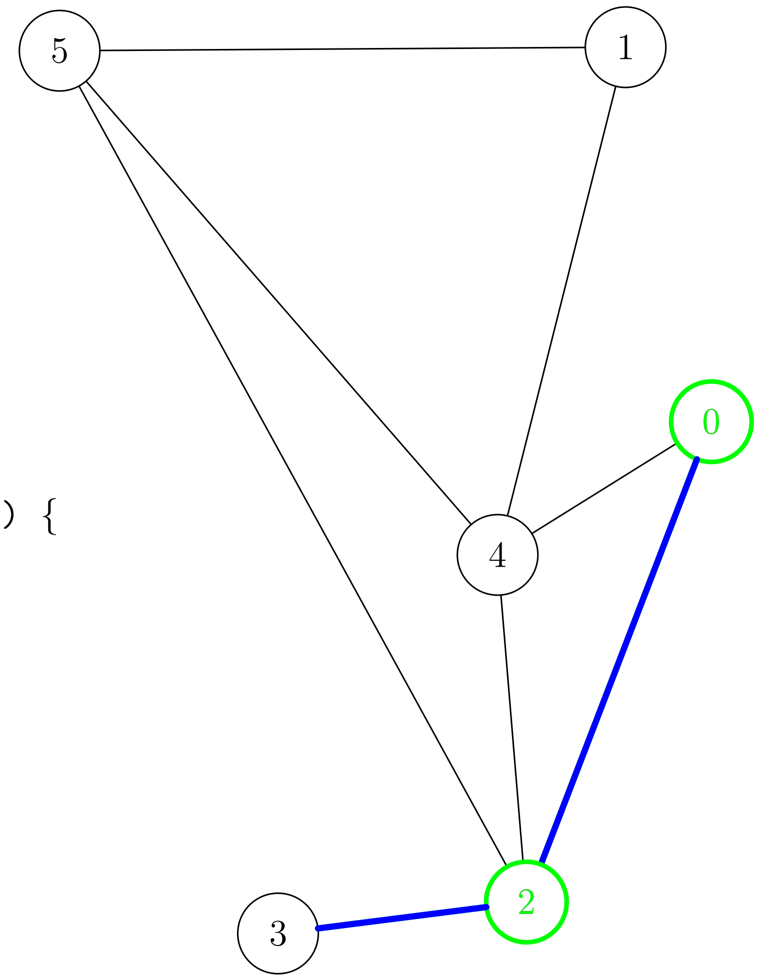
node=2 neighbour=3 time=2



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

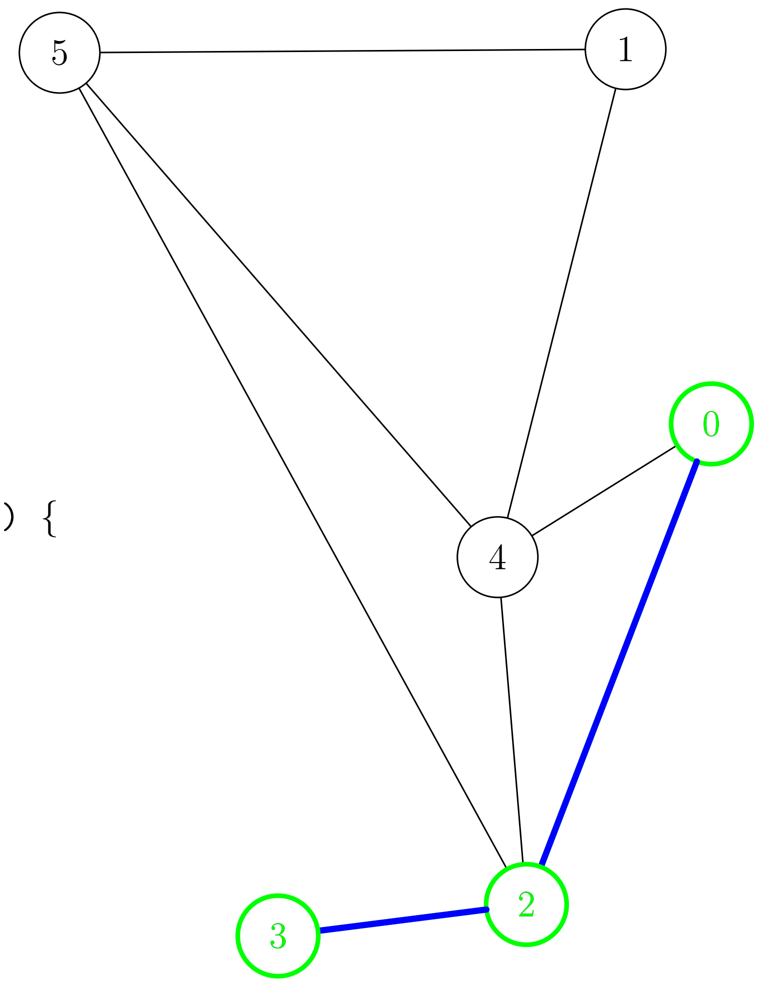
node=2 neighbour=3 time=2



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

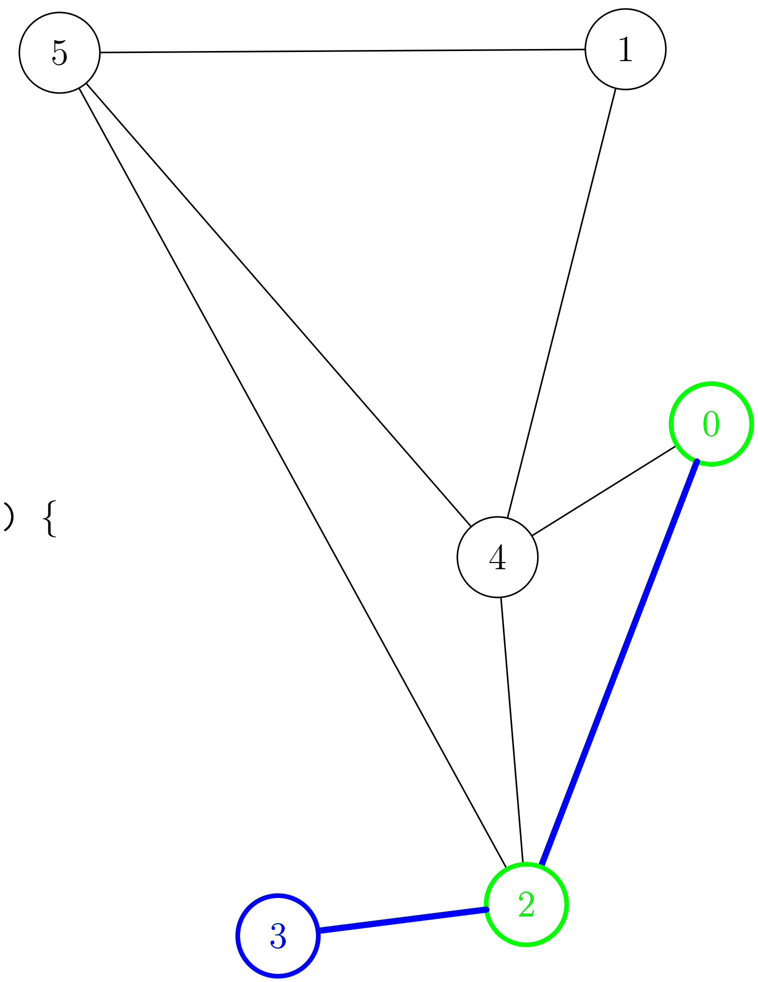
node=3 neighbour=nil time=3



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

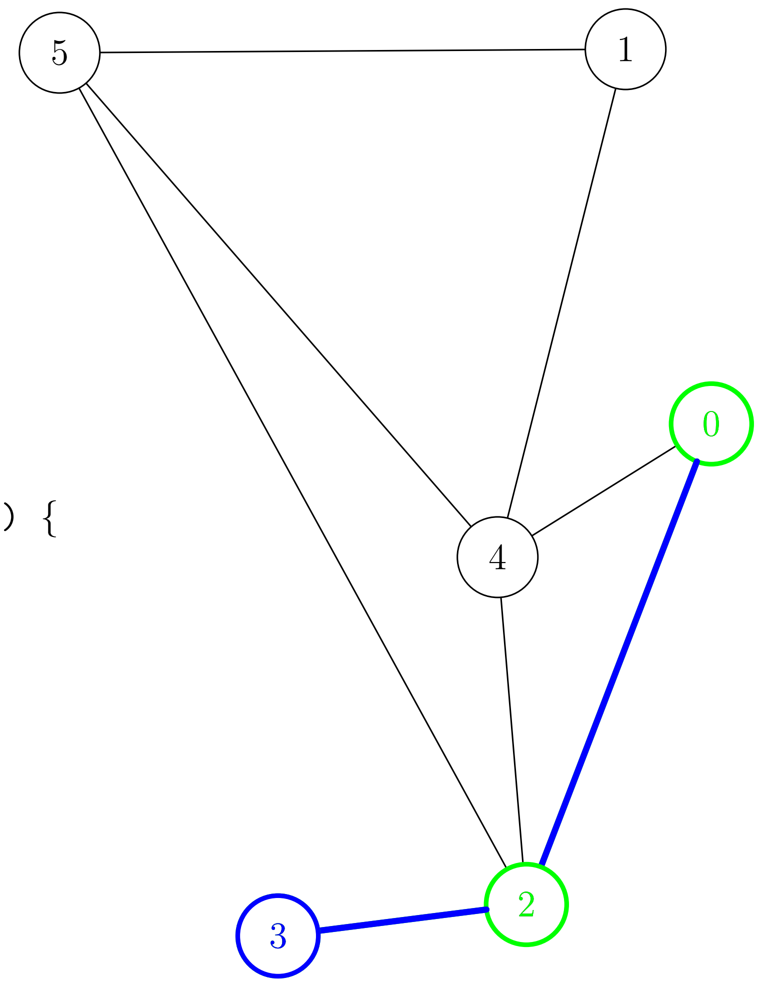
node=3   neighbour=nil time=4



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

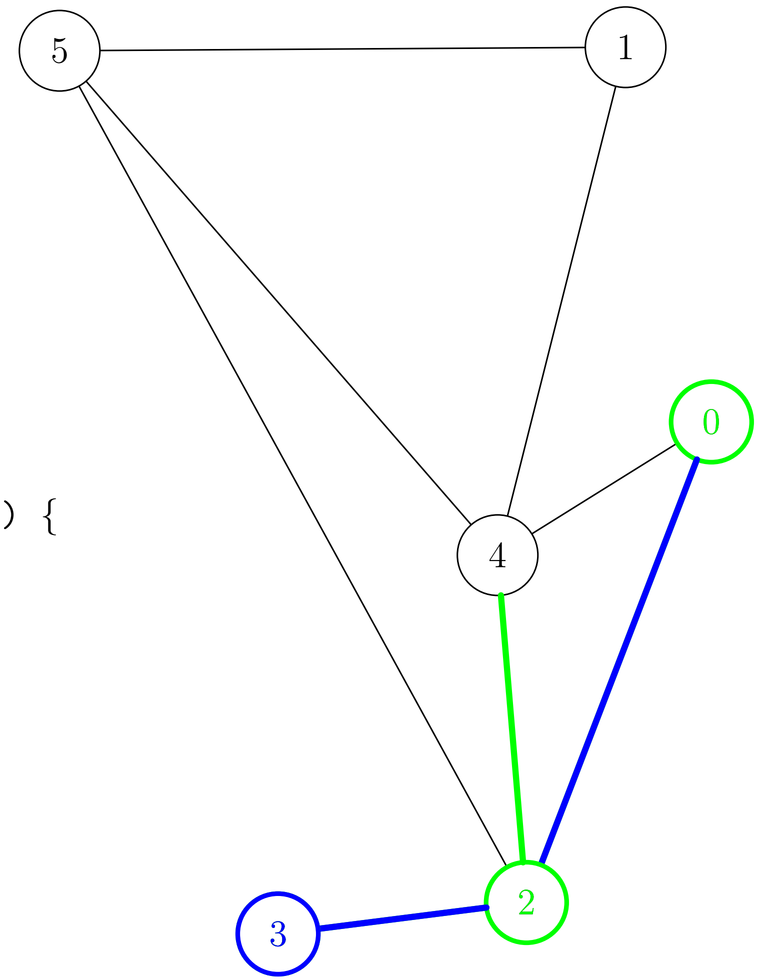
node=2 neighbour=3 time=4



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

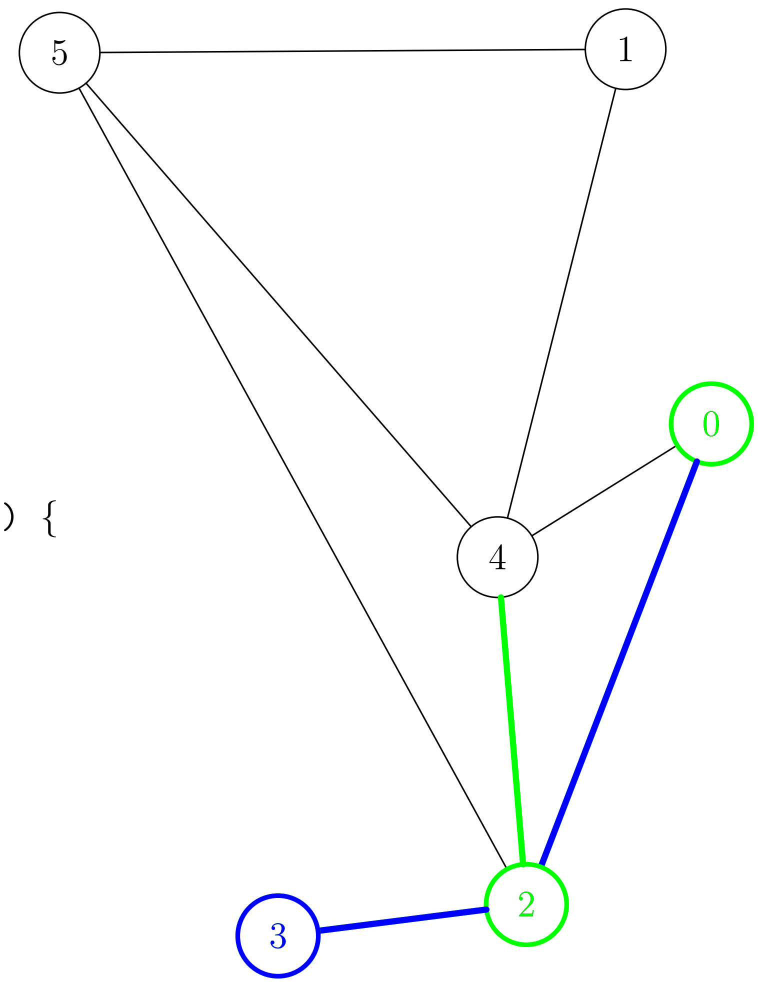
node=2 neighbour=4 time=4



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

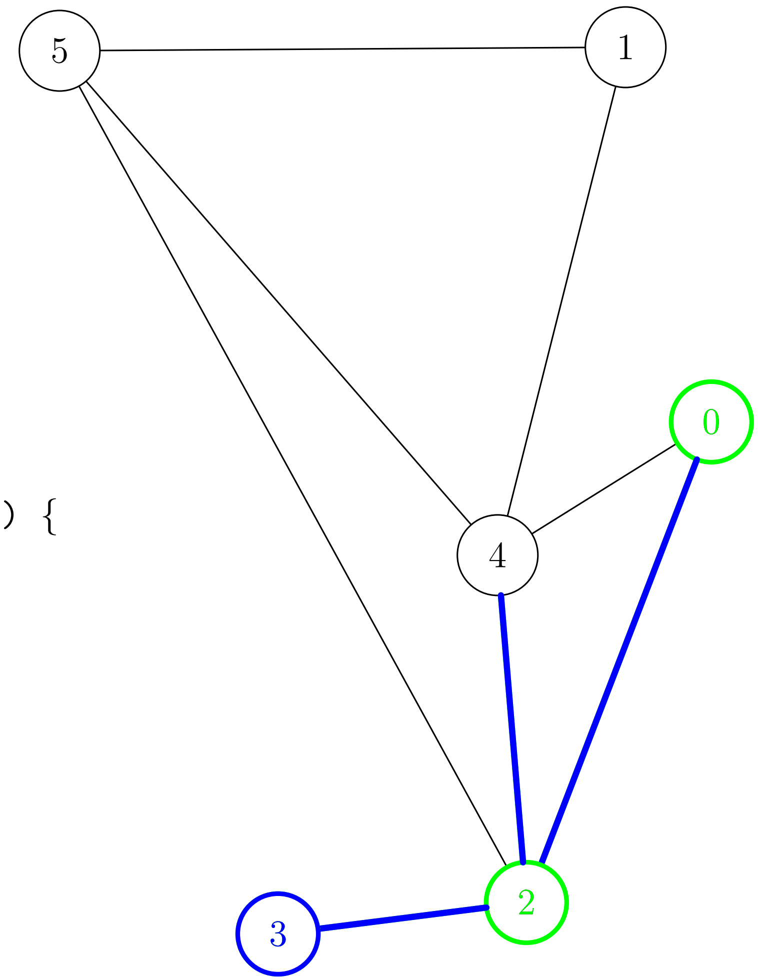
node=2 neighbour=4 time=4



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

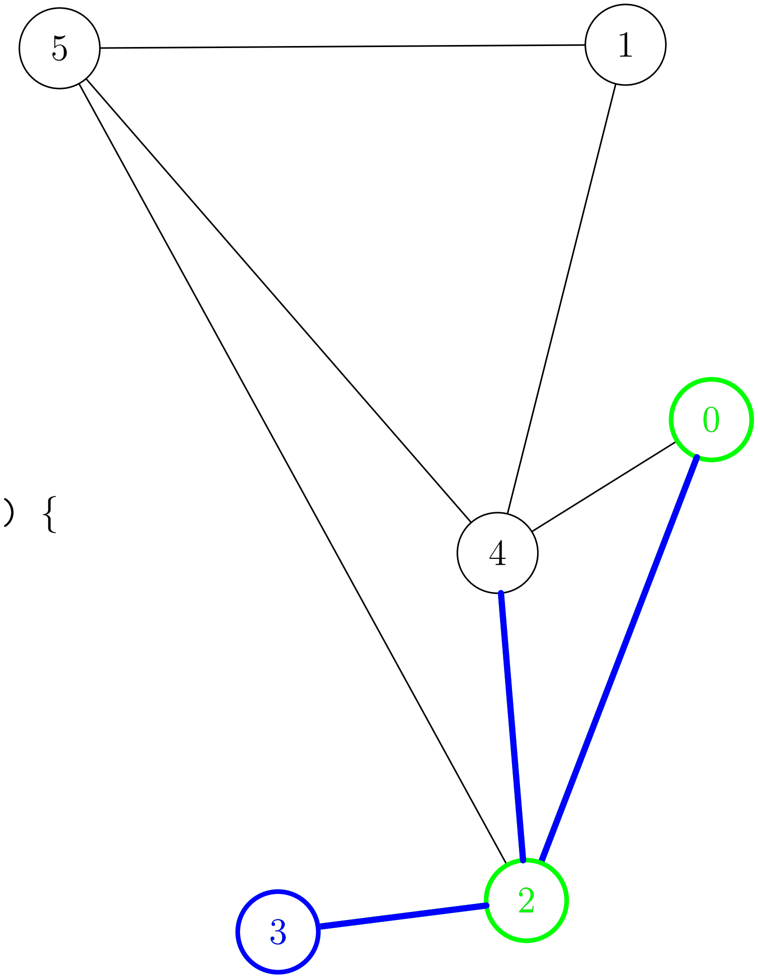
node=2 neighbour=4 time=4



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

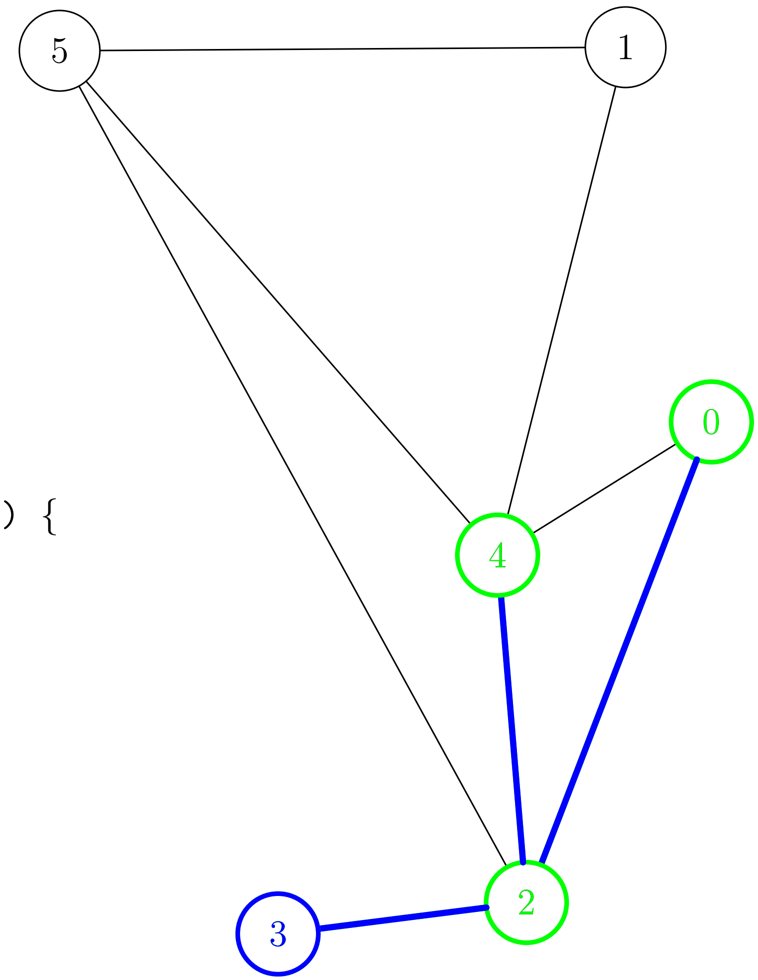
node=2 neighbour=4 time=4



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

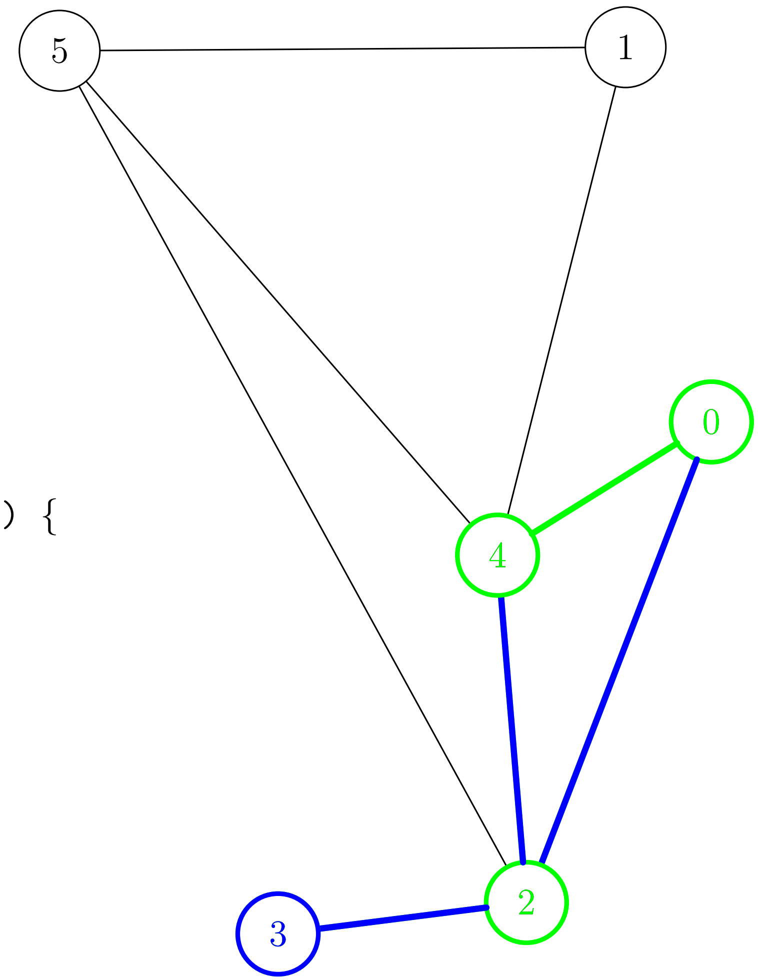
node=4 neighbour=nil time=5



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

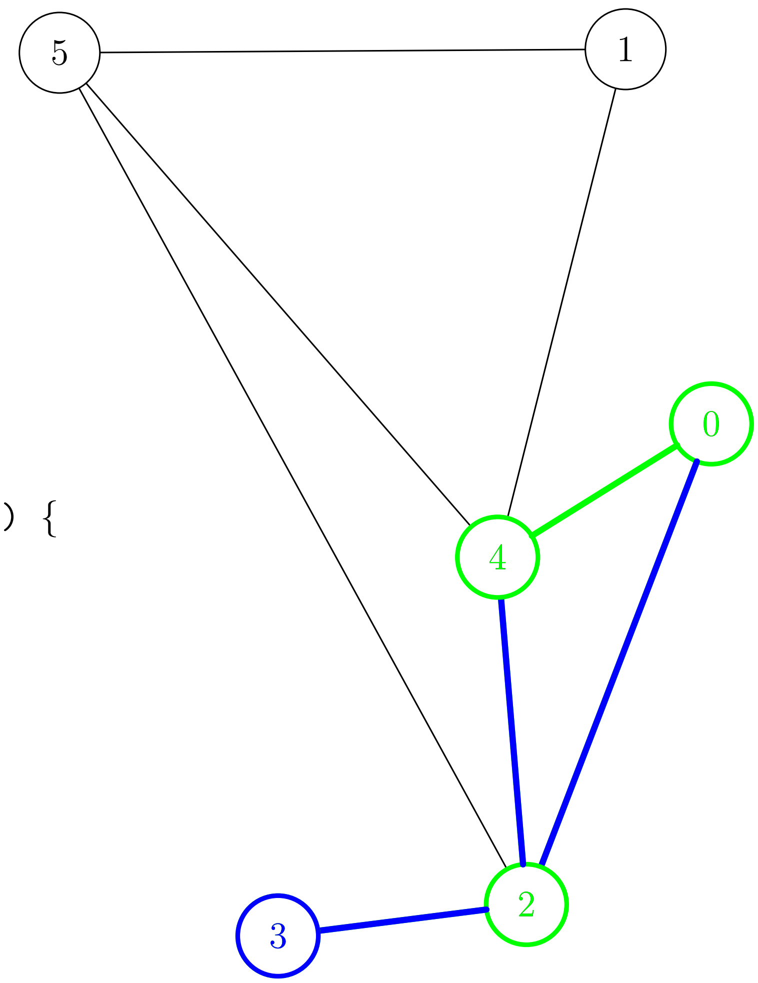
node=4 neighbour=nil time=5



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

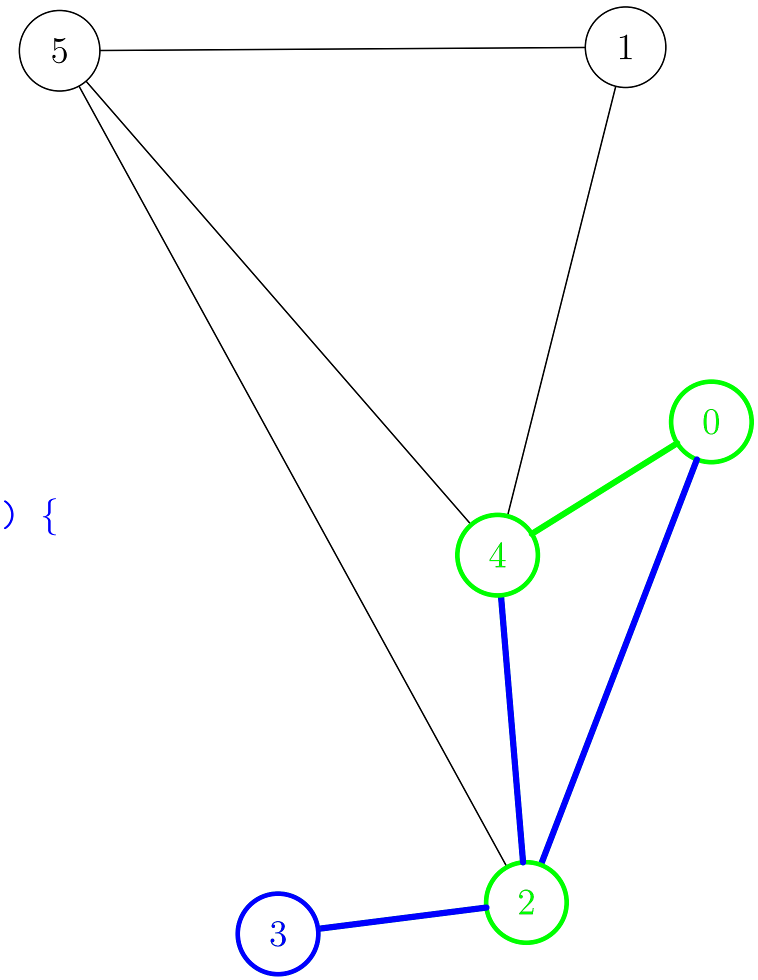
node=4 neighbour=nil time=5



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

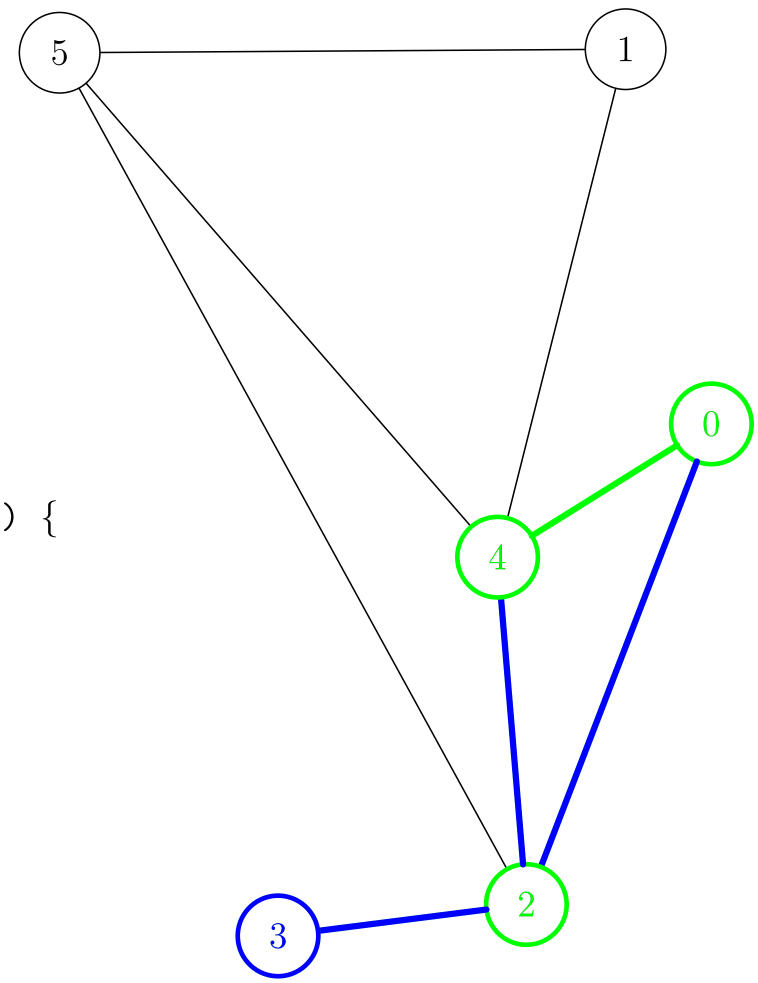
node=4 neighbour=nil time=5



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

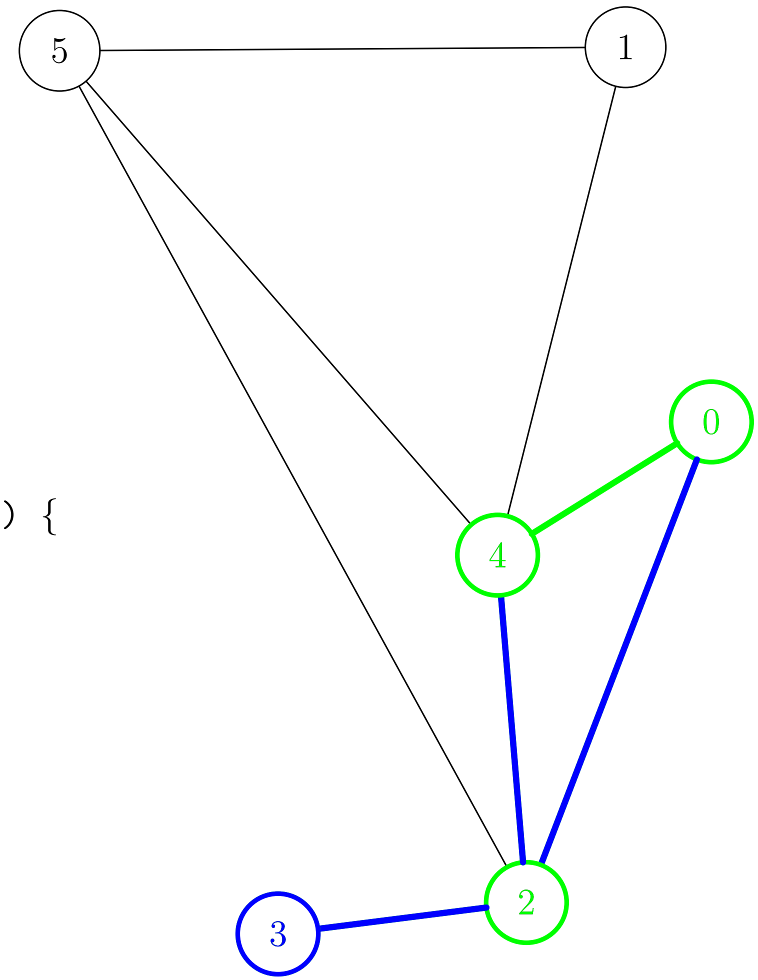
node=4 neighbour=nil time=5



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

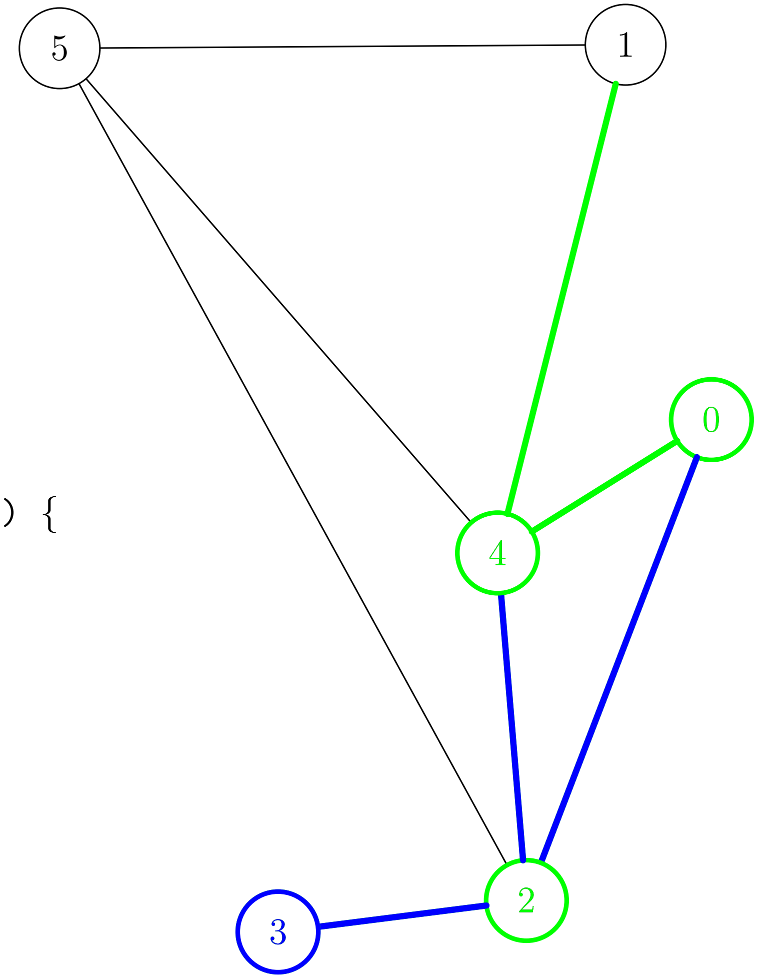
node=4   neighbour=nil   time=5



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

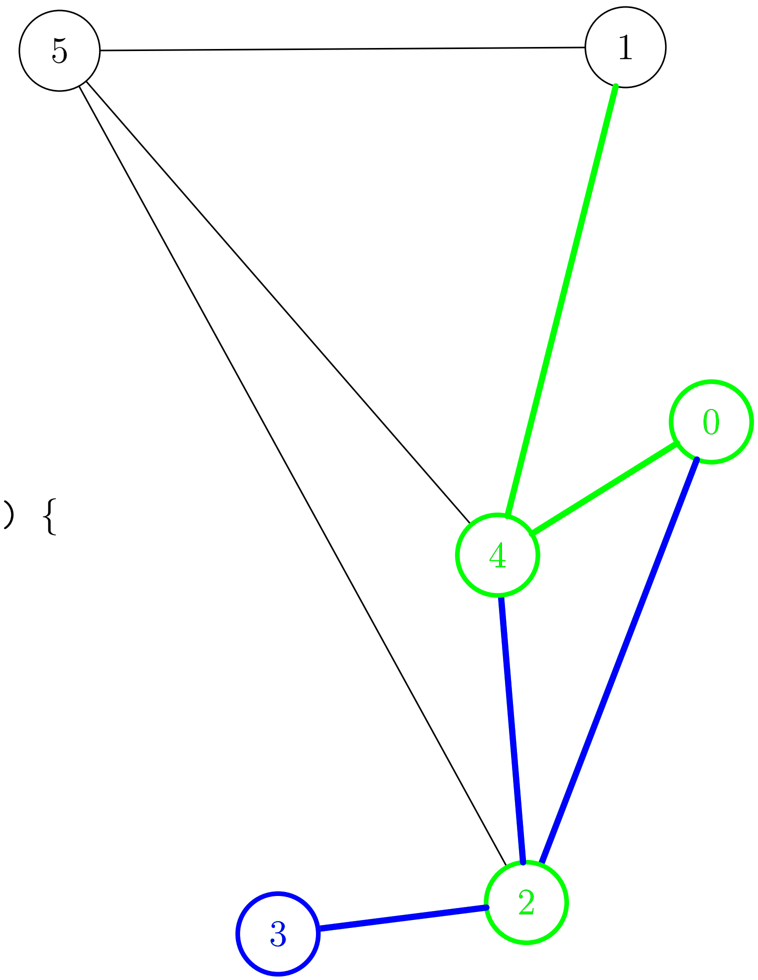
node=4 neighbour=1 time=5



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

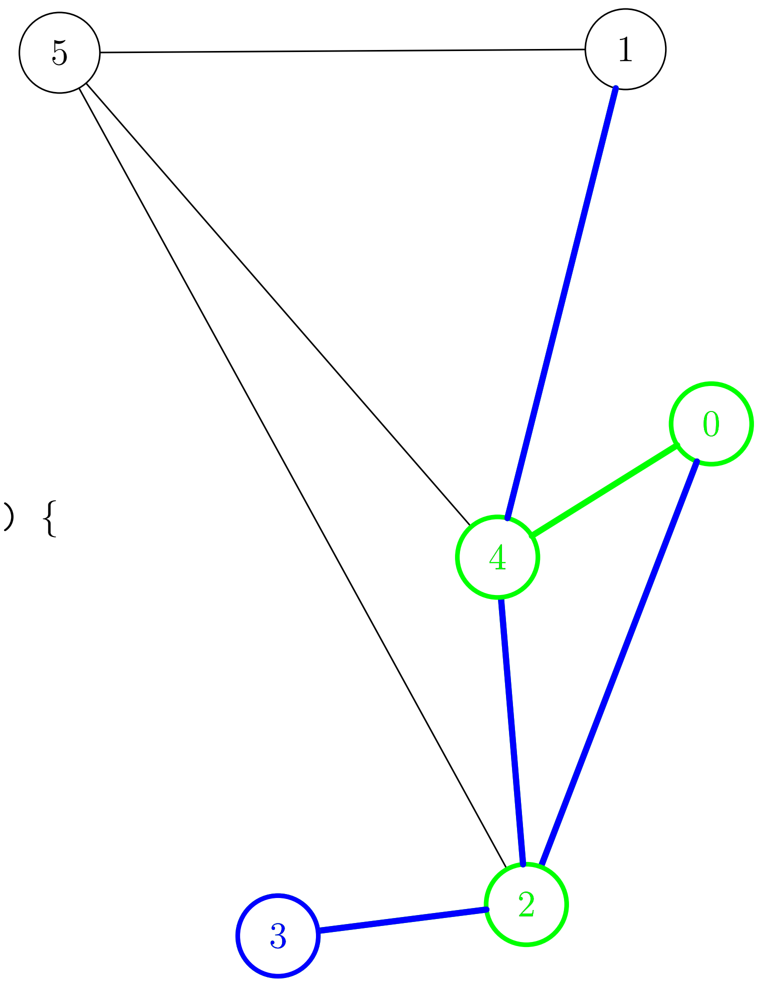
node=4   neighbour=1   time=5



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

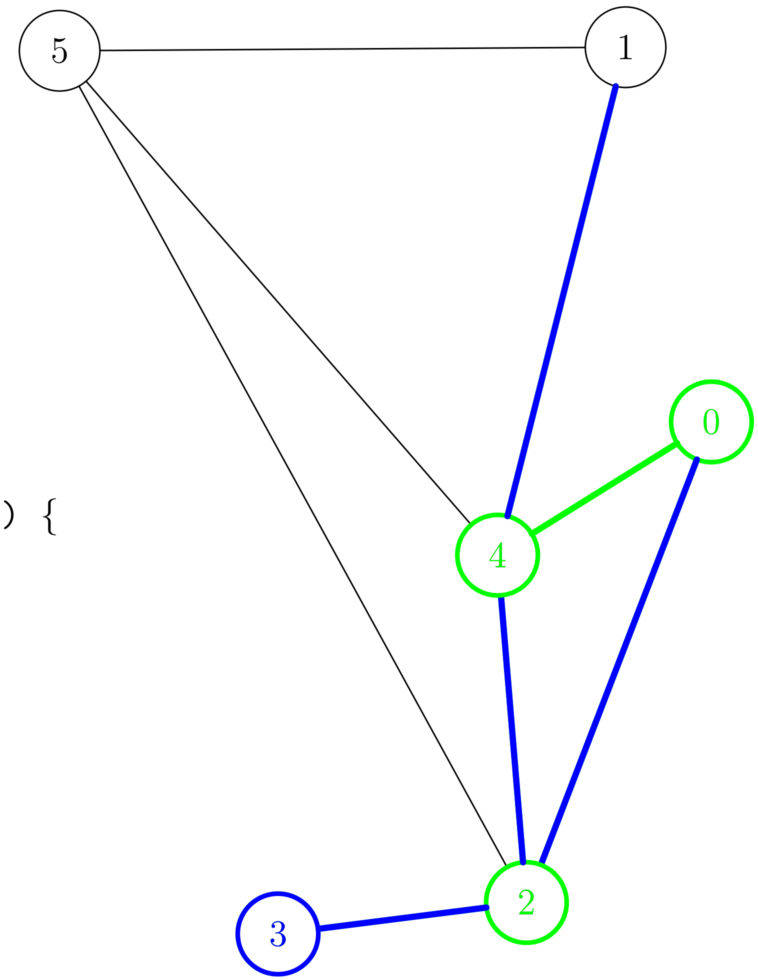
node=4   neighbour=1   time=5



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

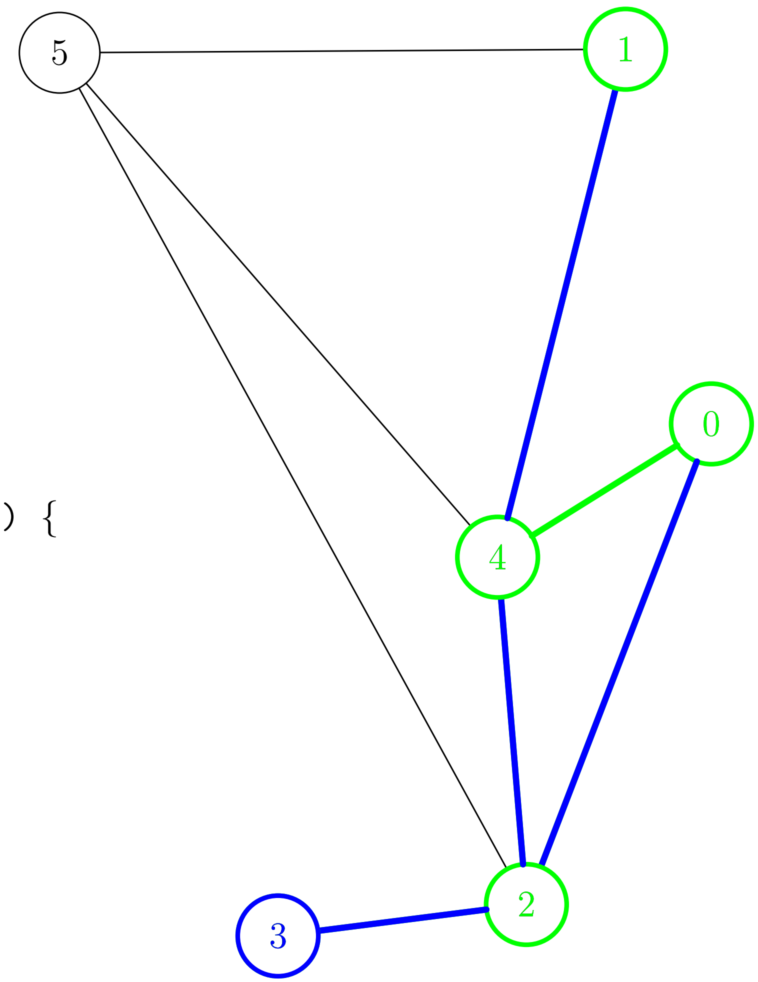
node=4 neighbour=1 time=5



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

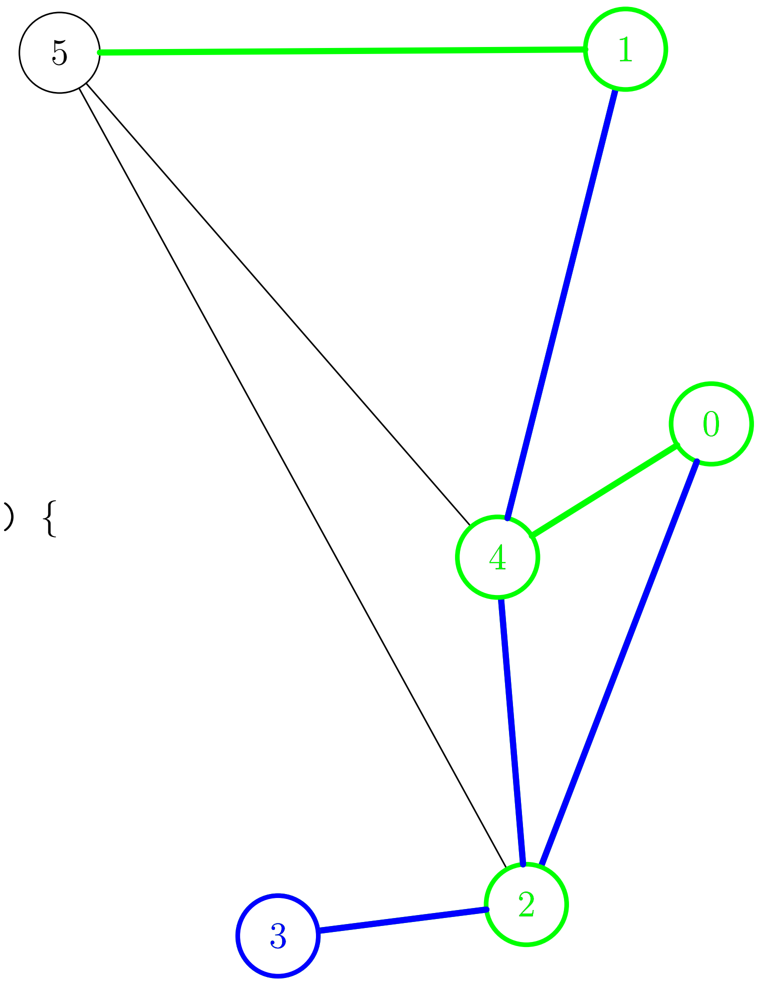
node=1 neighbour=nil time=6



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

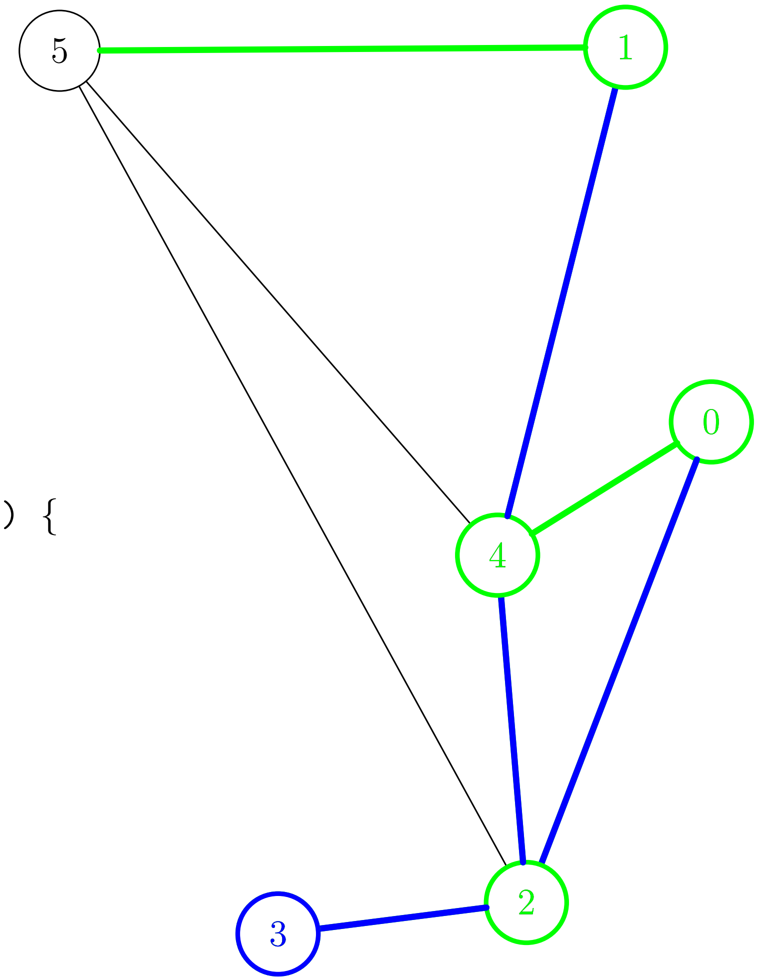
node=1 neighbour=5 time=6



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

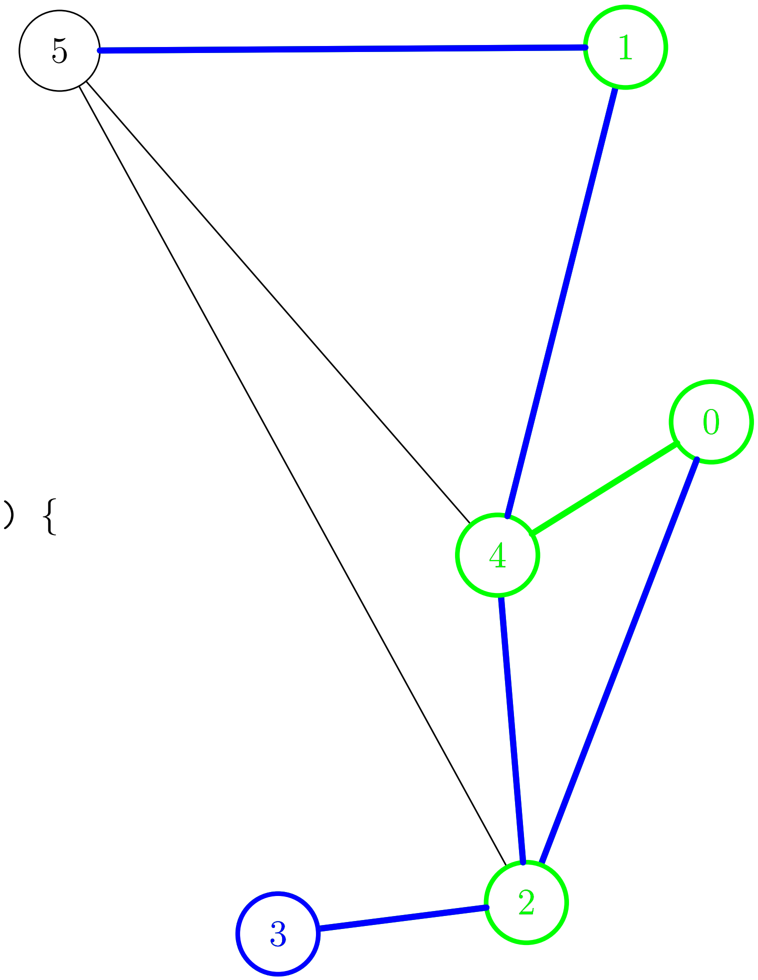
node=1   neighbour=5   time=6



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

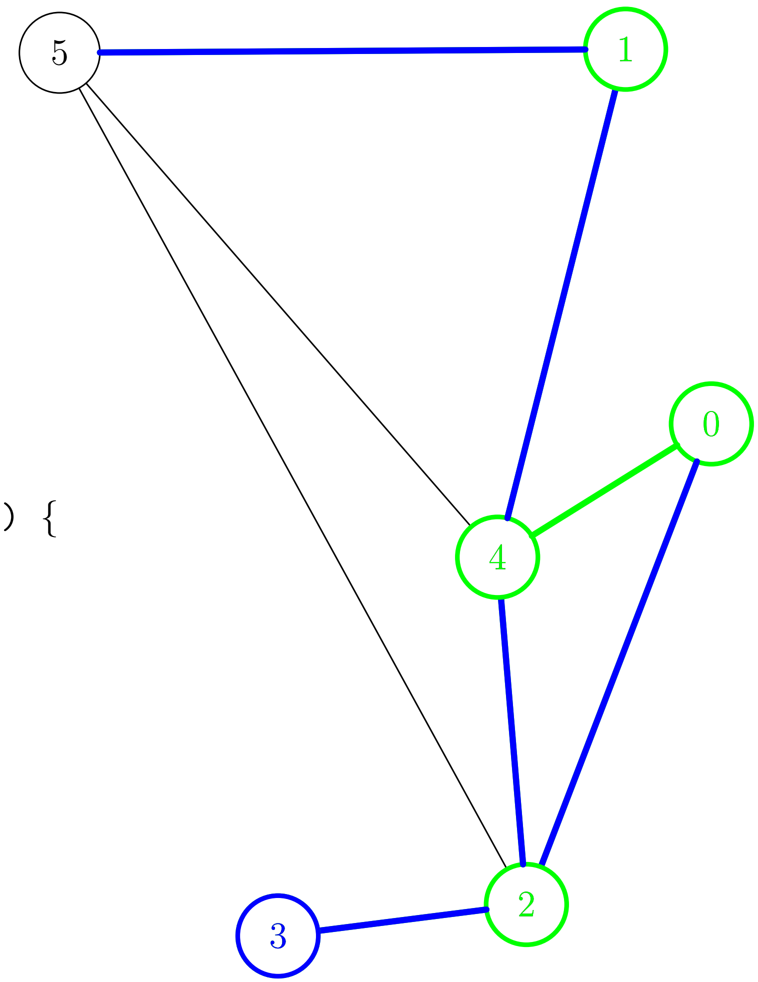
node=1   neighbour=5   time=6



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

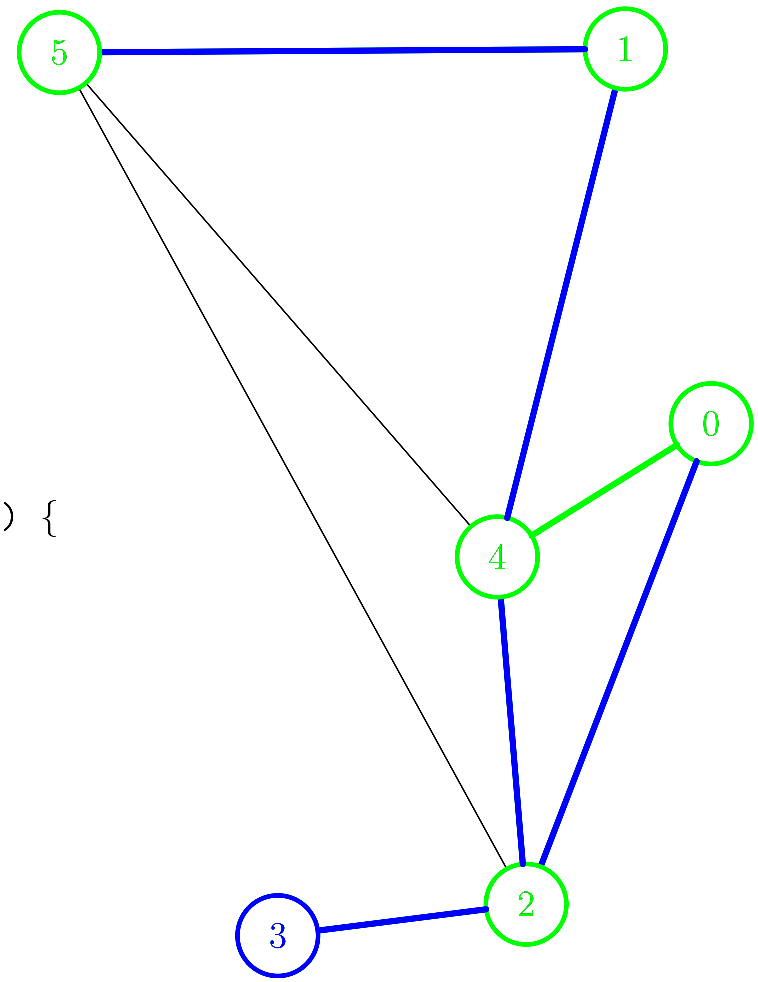
node=1 neighbour=5 time=6



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

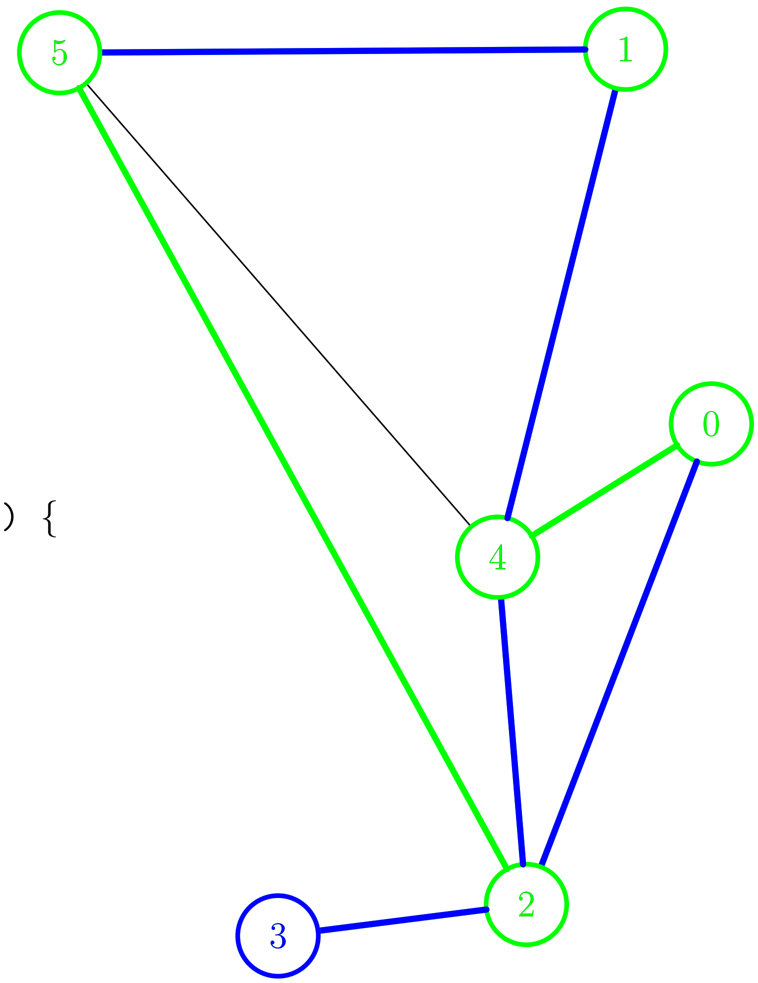
node=5 neighbour=nil time=7



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

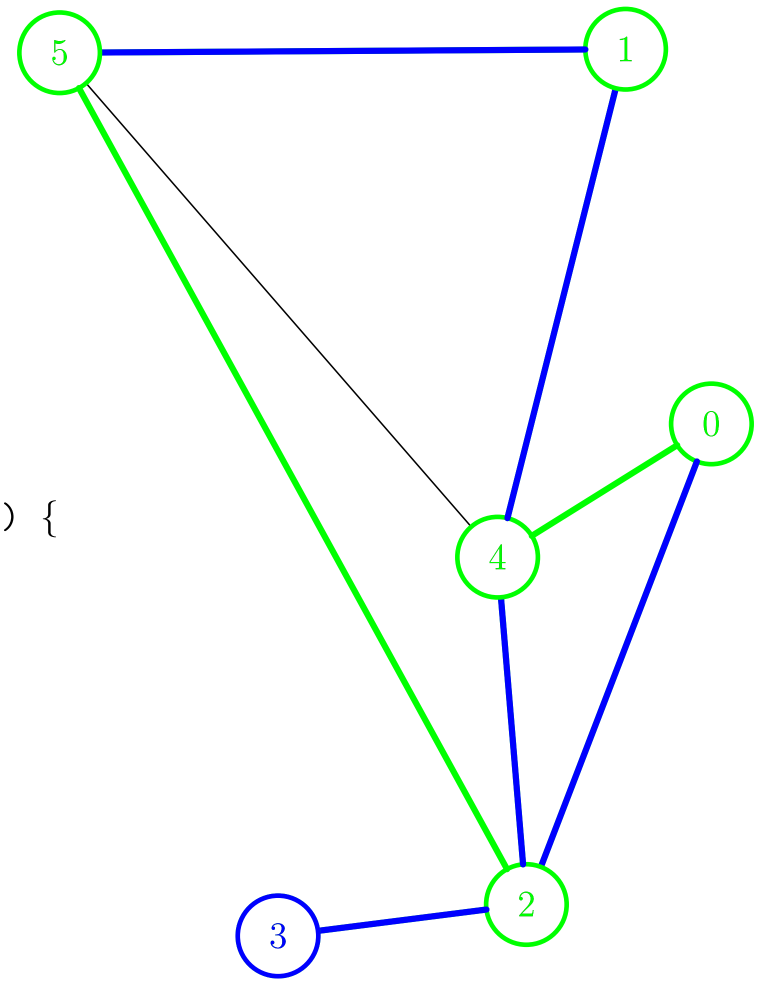
node=5   neighbour=2   time=7



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

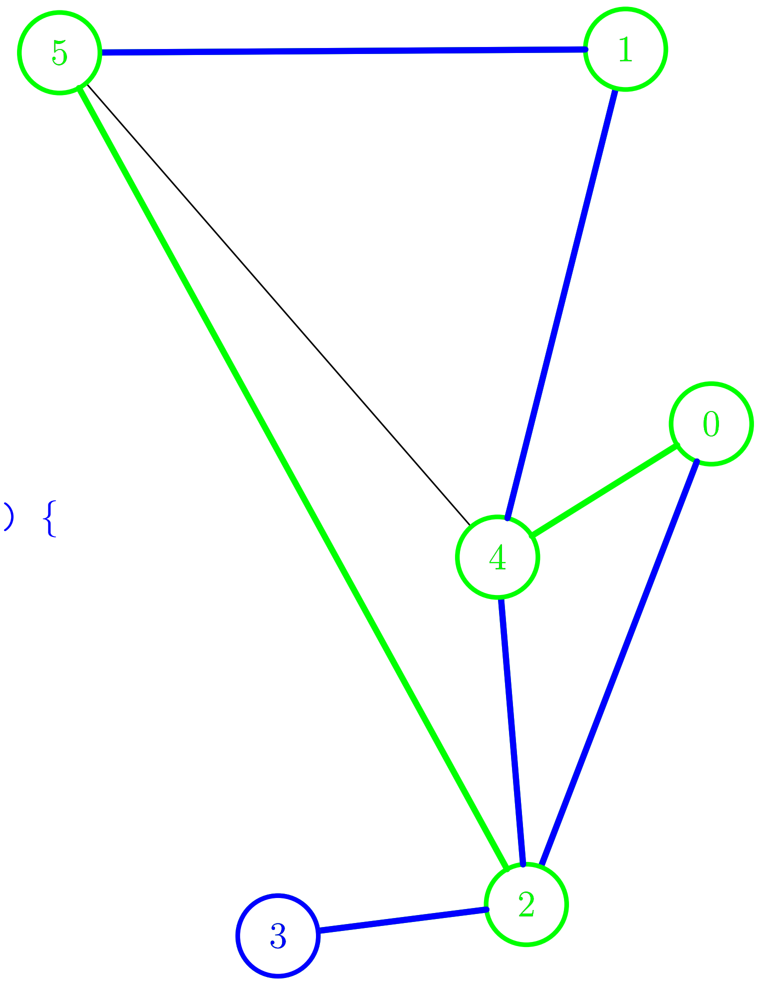
node=5 neighbour=2 time=7



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

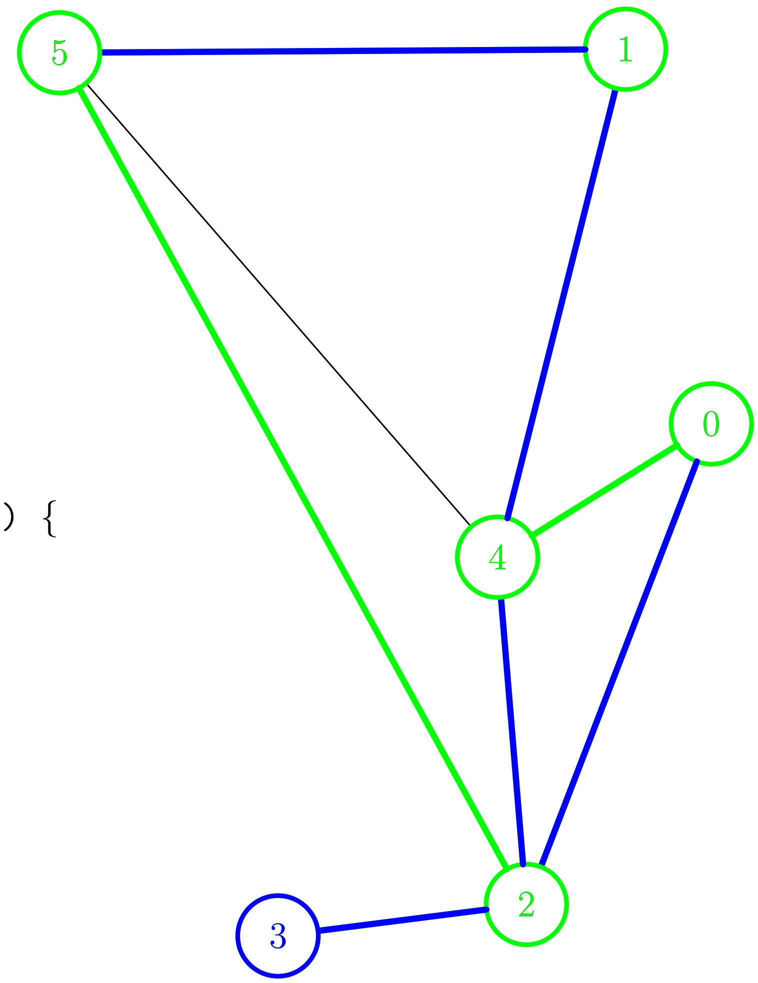
node=5 neighbour=2 time=7



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

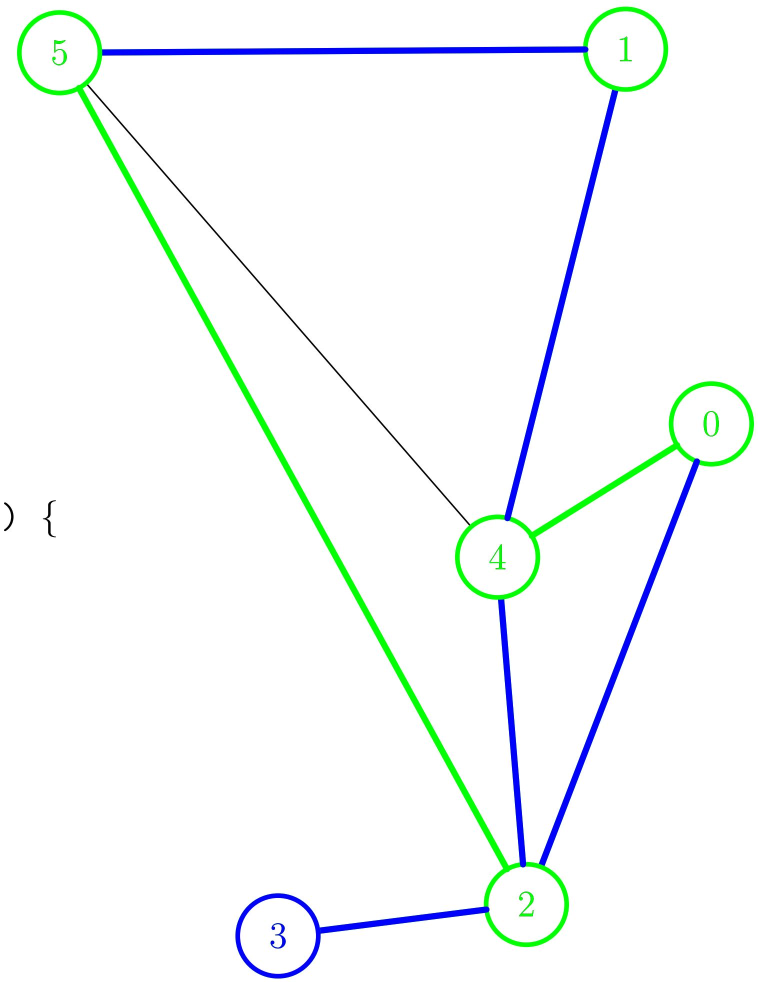
node=5   neighbour=2   time=7



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

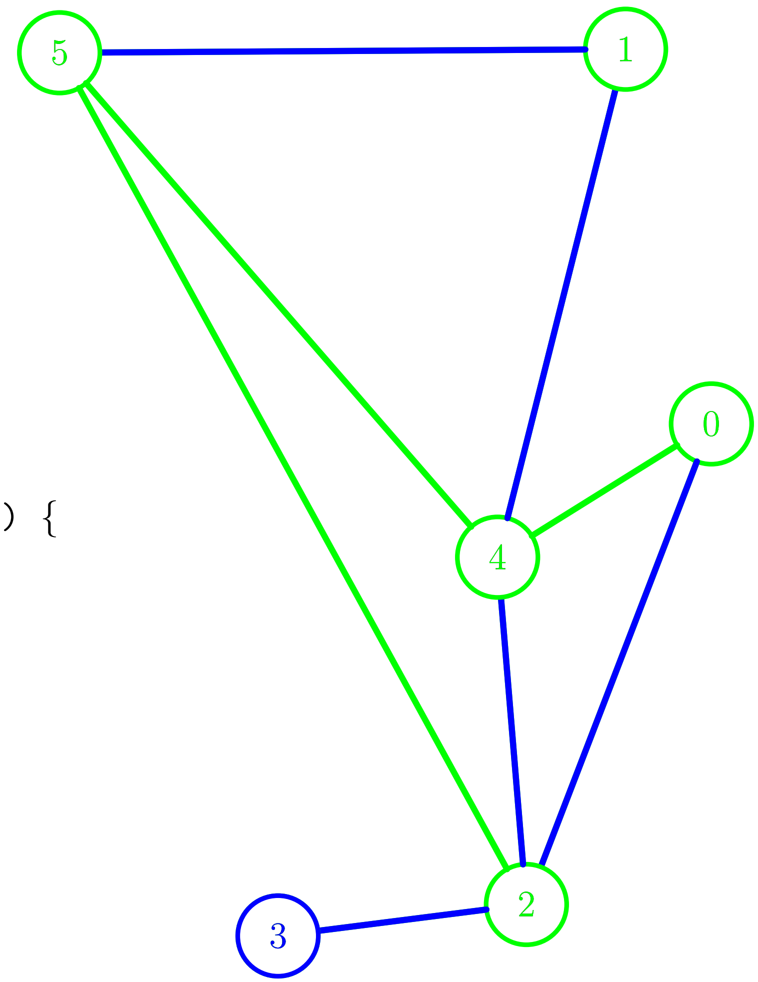
node=5   neighbour=2   time=7



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

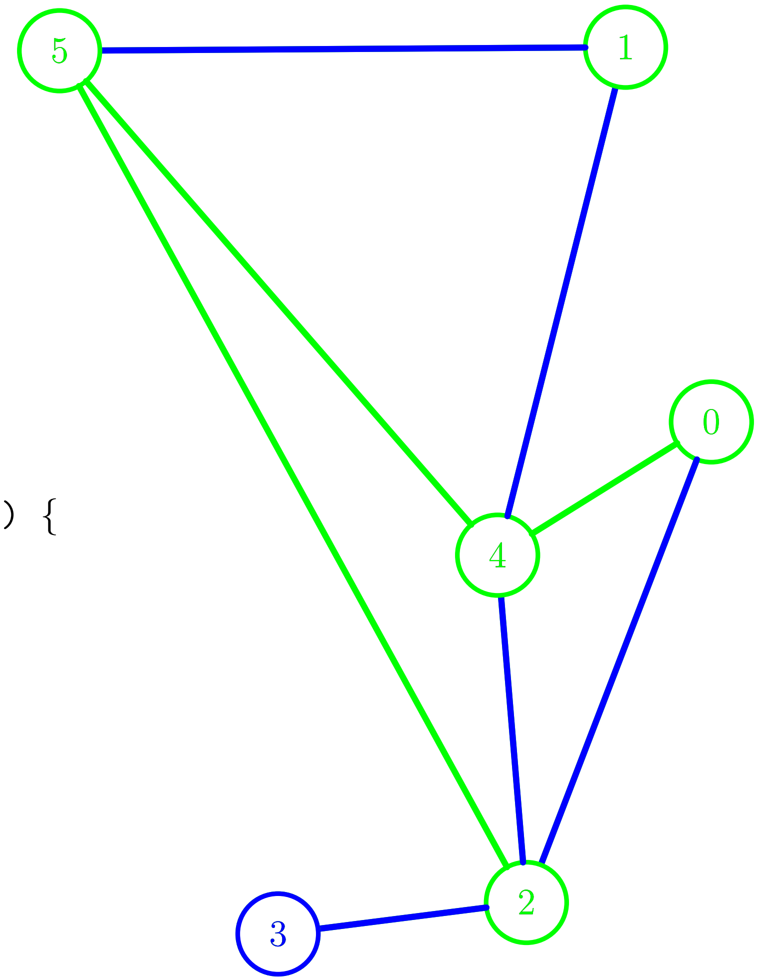
node=5 neighbour=4 time=7



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

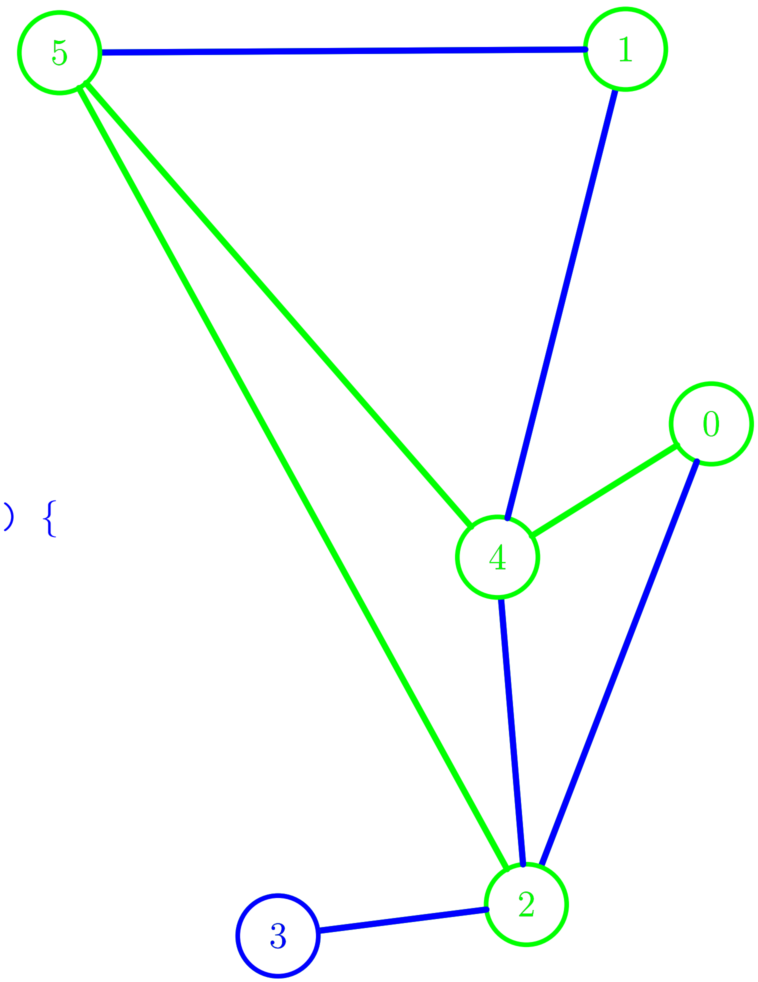
node=5 neighbour=4 time=7



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

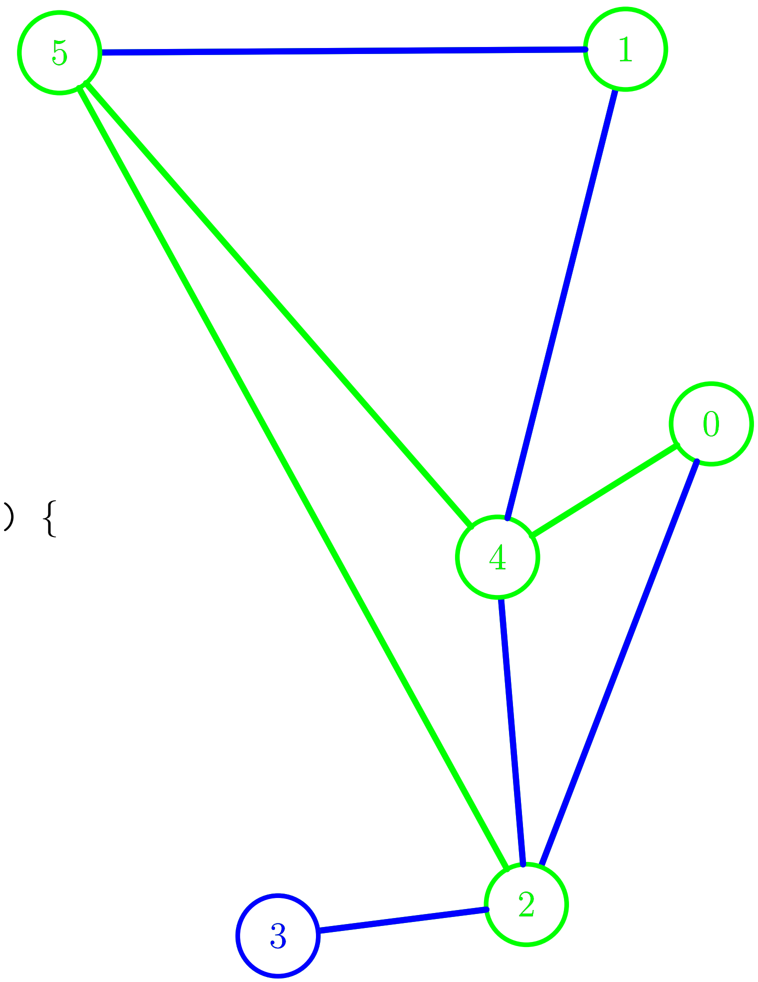
node=5 neighbour=4 time=7



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

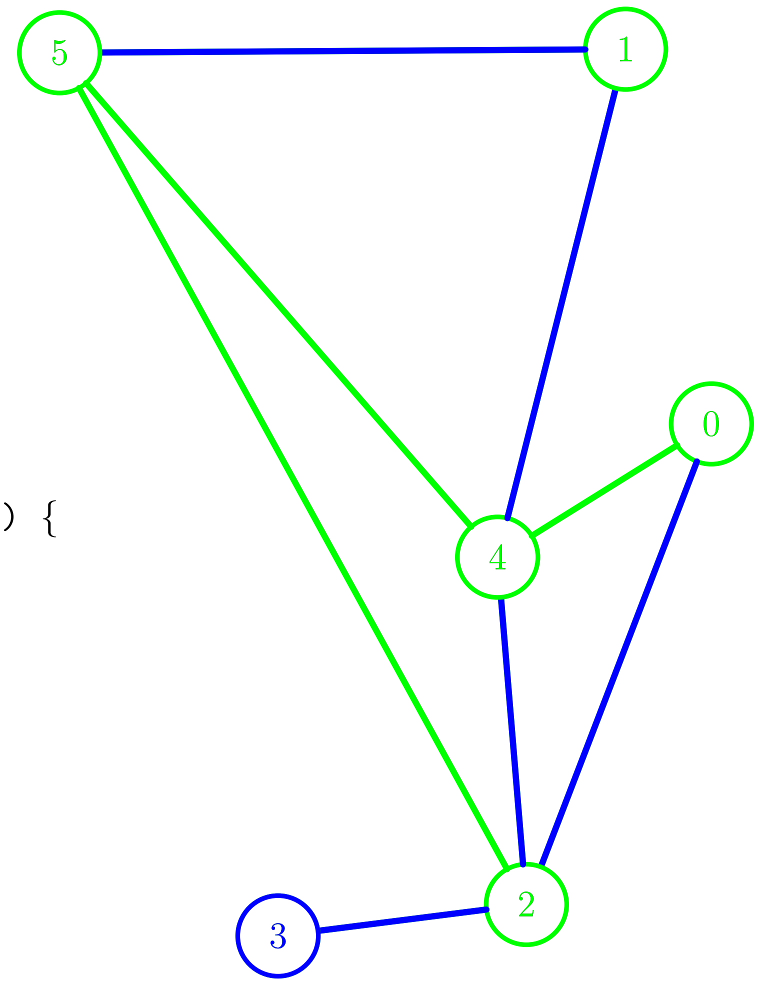
node=5 neighbour=4 time=7



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

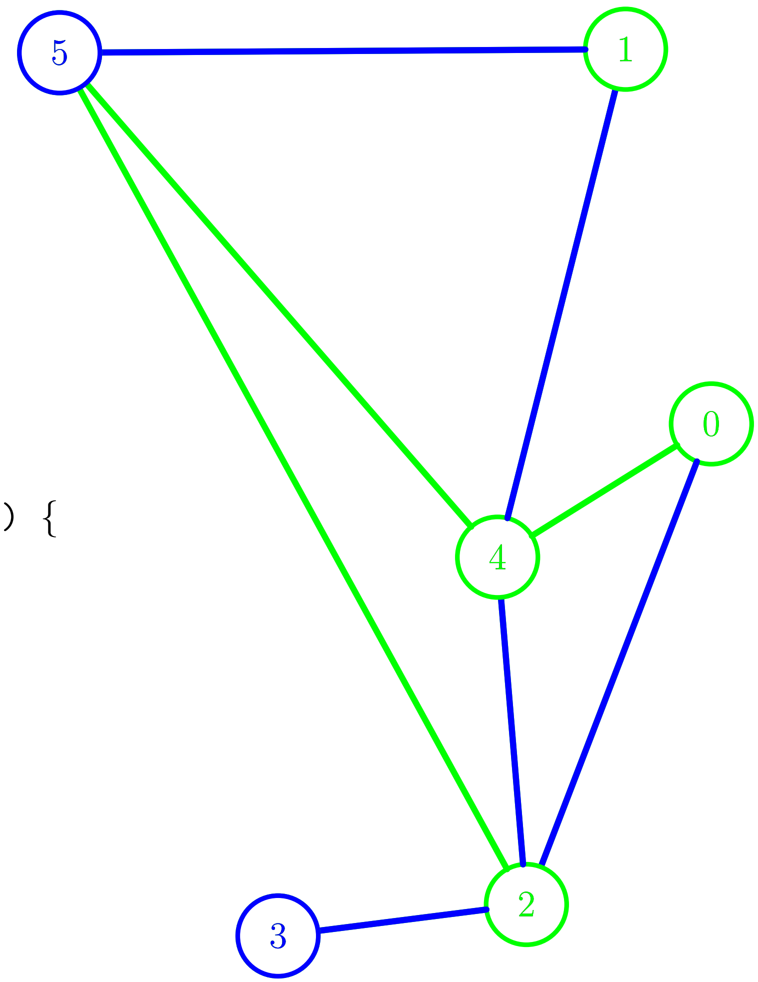
node=5 neighbour=4 time=7



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

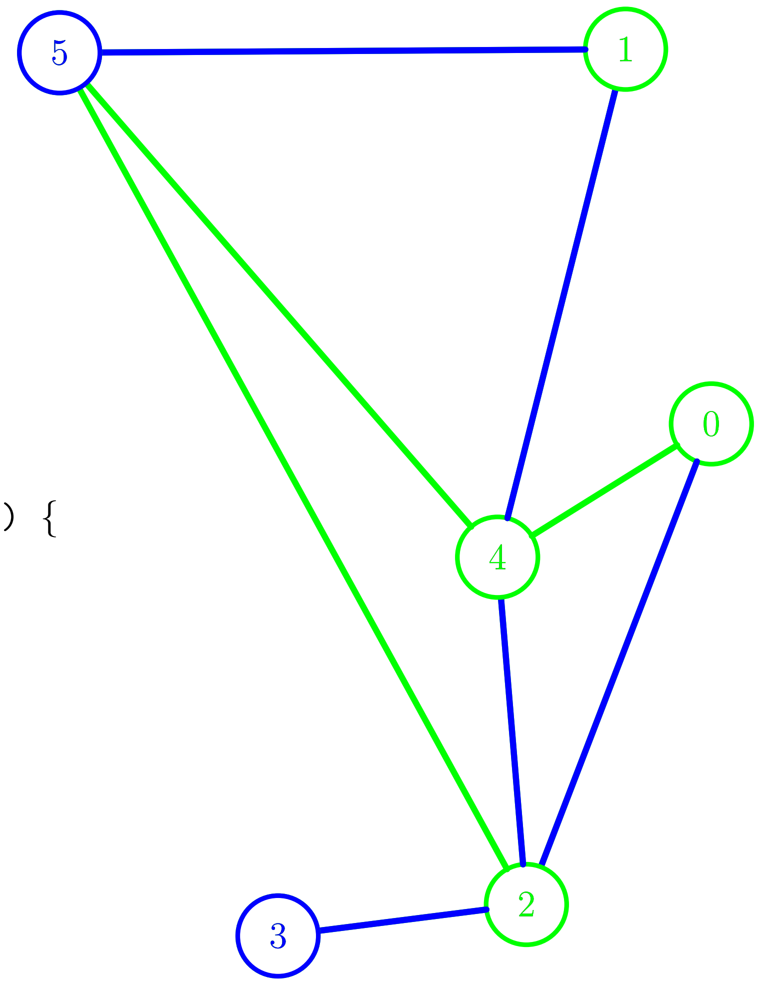
node=5 neighbour=nil time=8



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

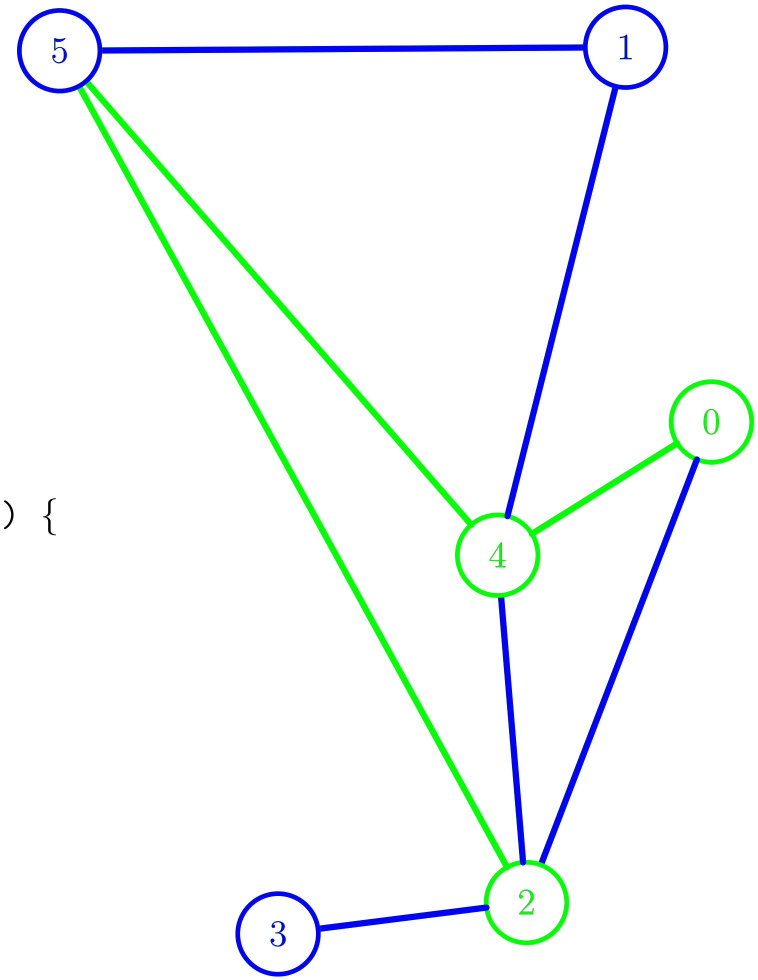
node=1   neighbour=5   time=8



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

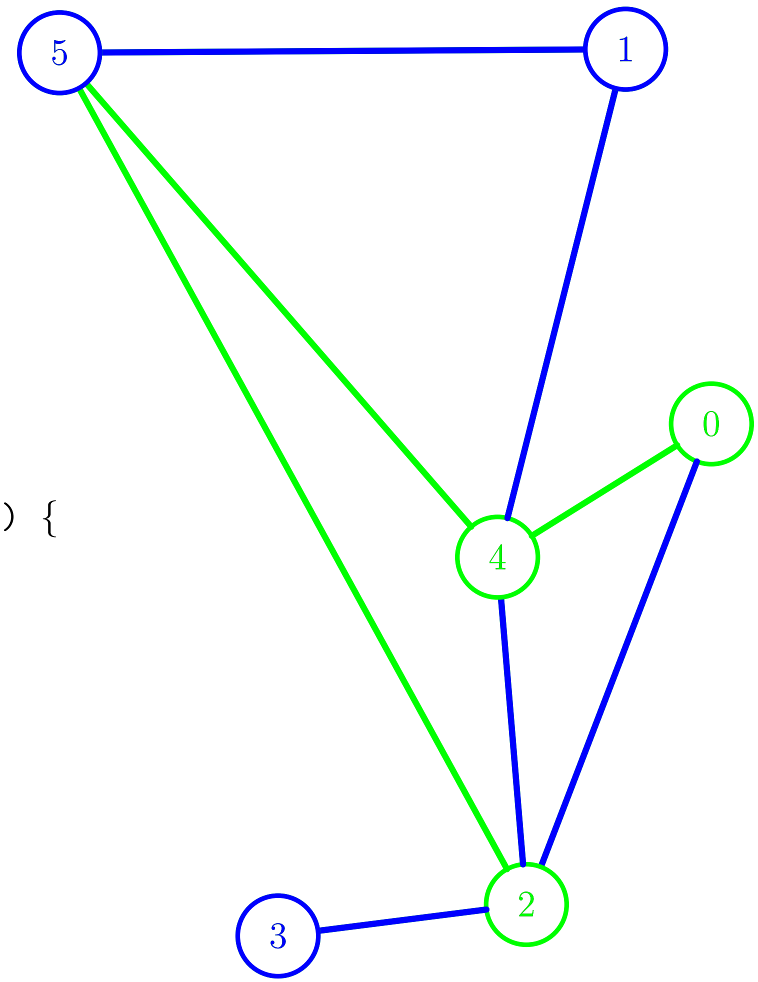
node=1 neighbour=nil time=9



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

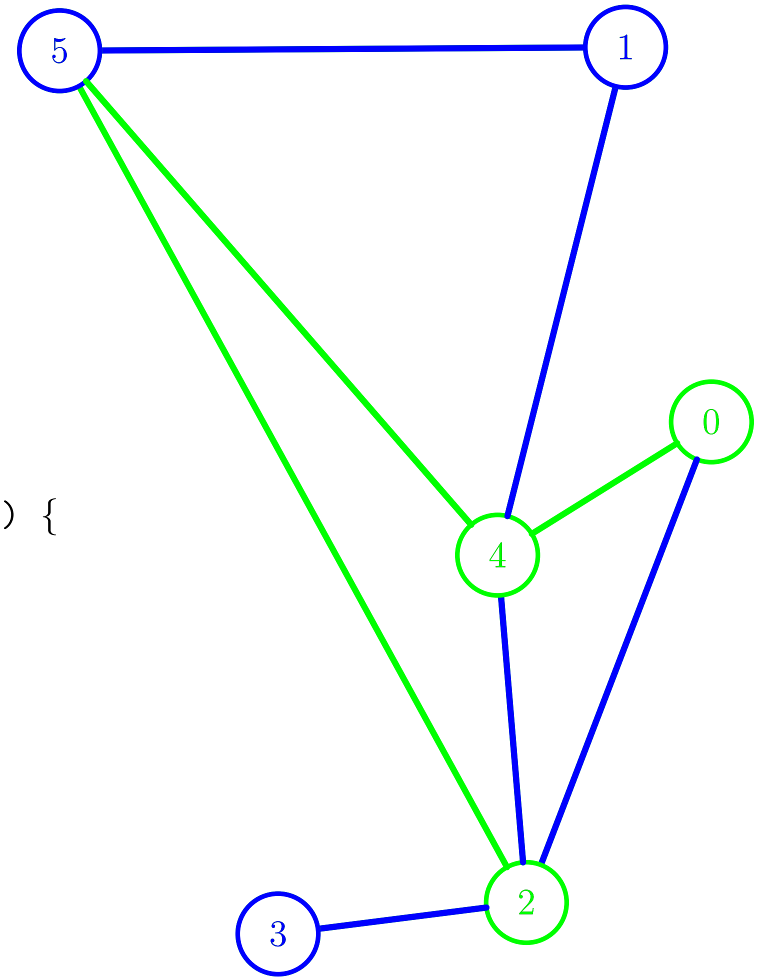
node=4 neighbour=1 time=9



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

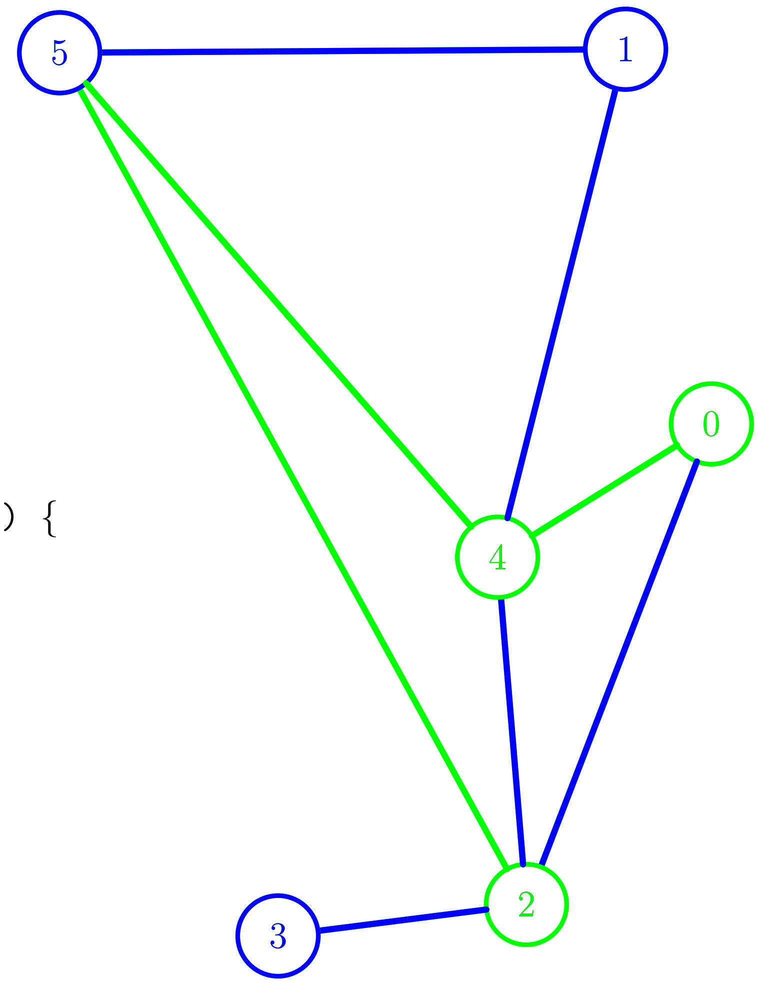
node=4 neighbour=5 time=9



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

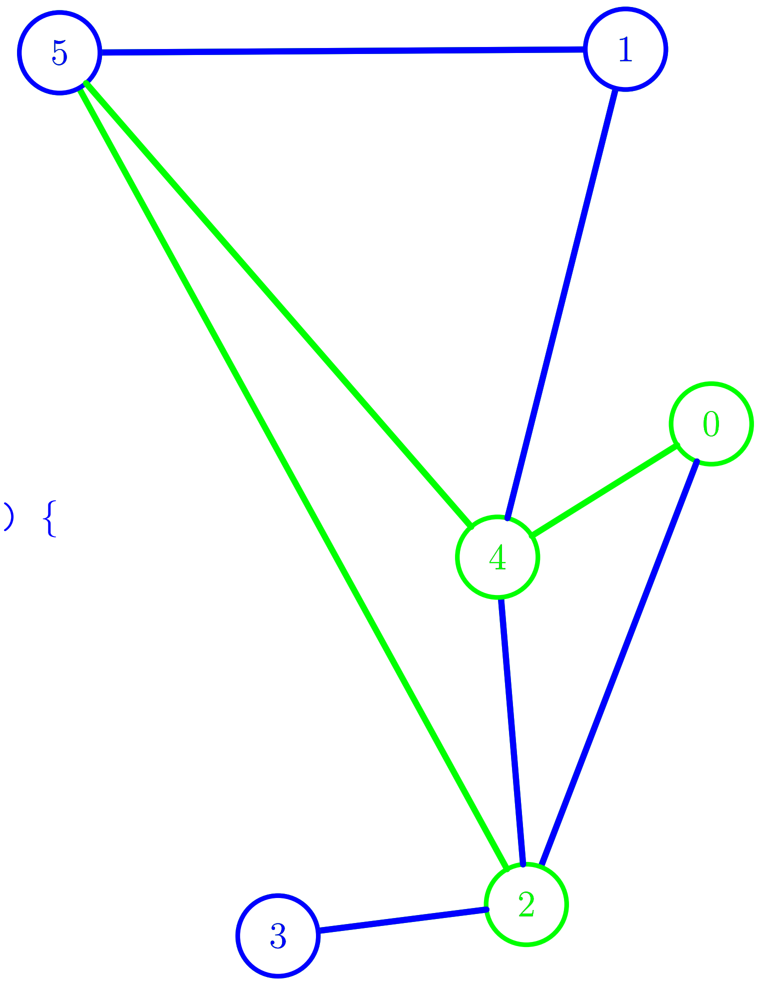
node=4   neighbour=5   time=9



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

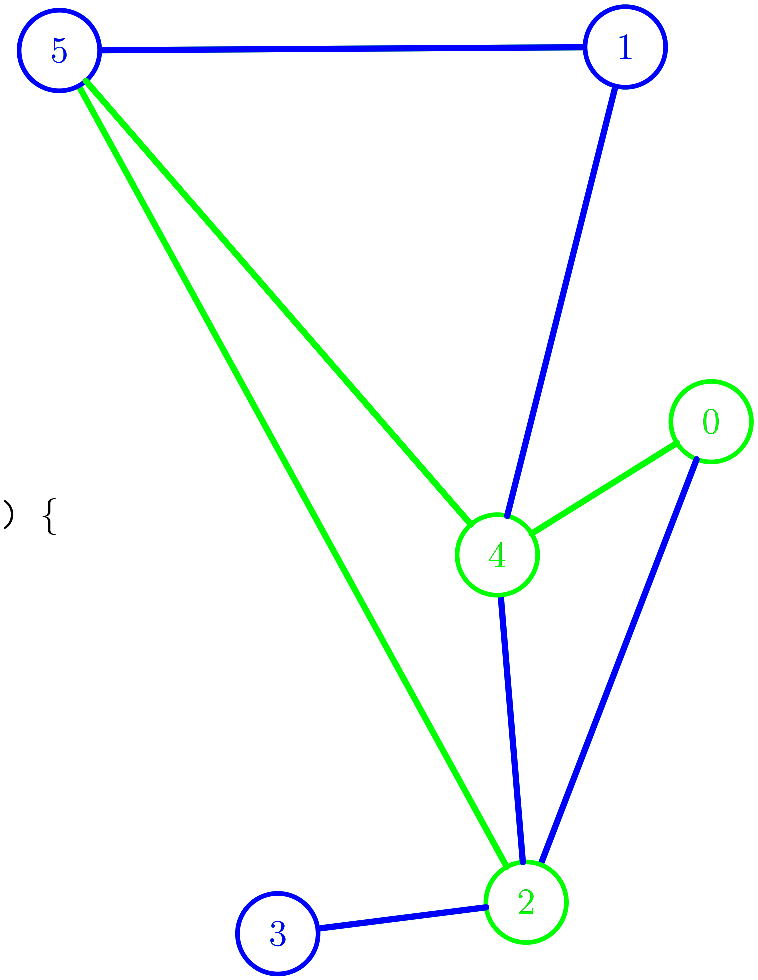
node=4 neighbour=5 time=9



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

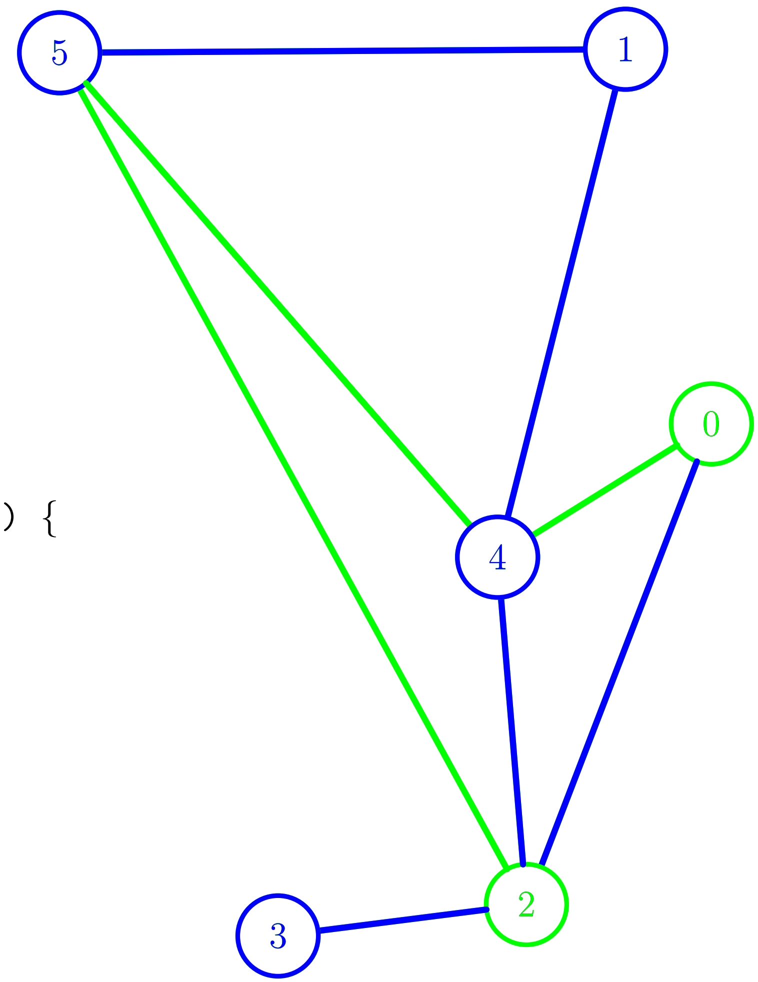
node=4 neighbour=5 time=9



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

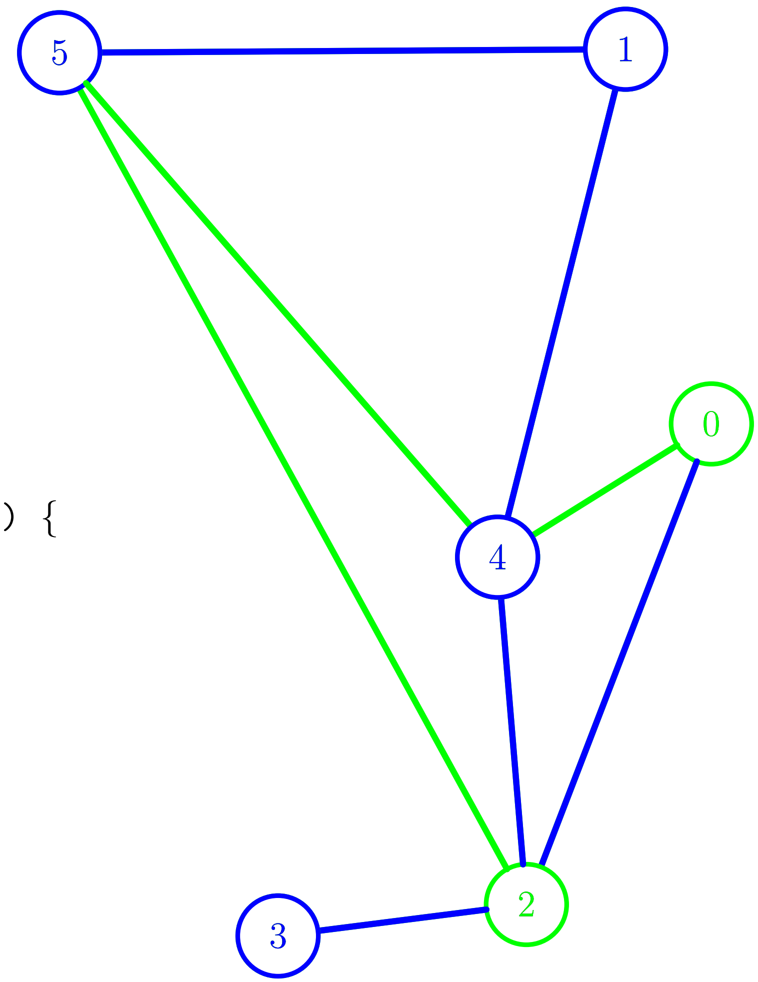
node=4 neighbour=nil time=10



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

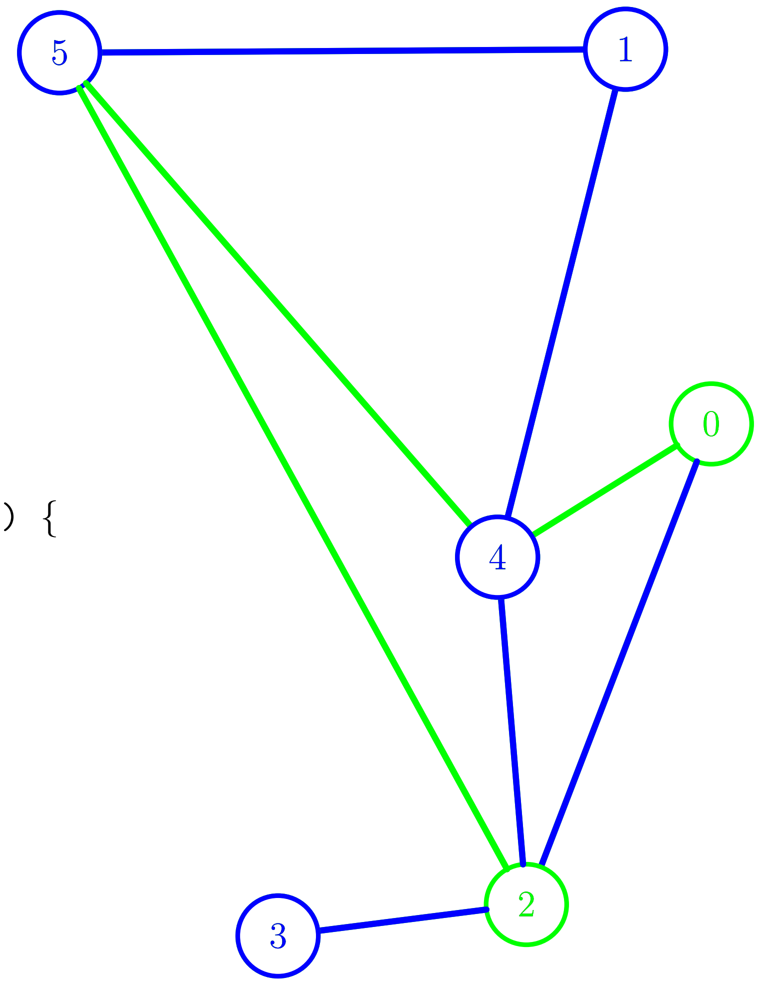
node=2   neighbour=4   time=10



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

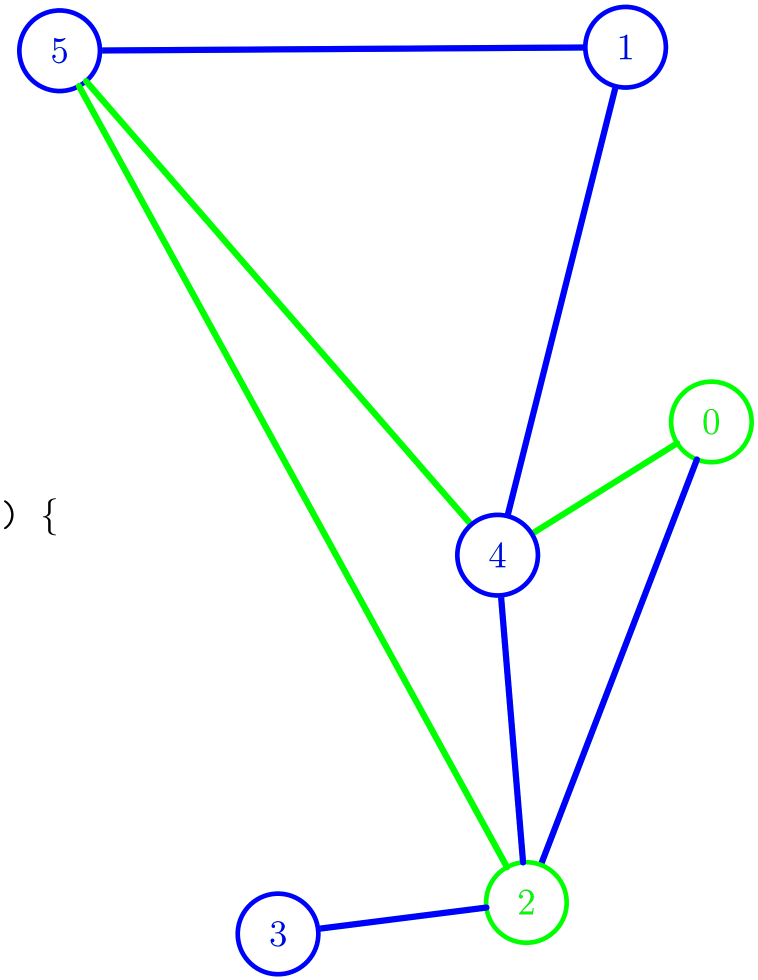
node=2   neighbour=5   time=10



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

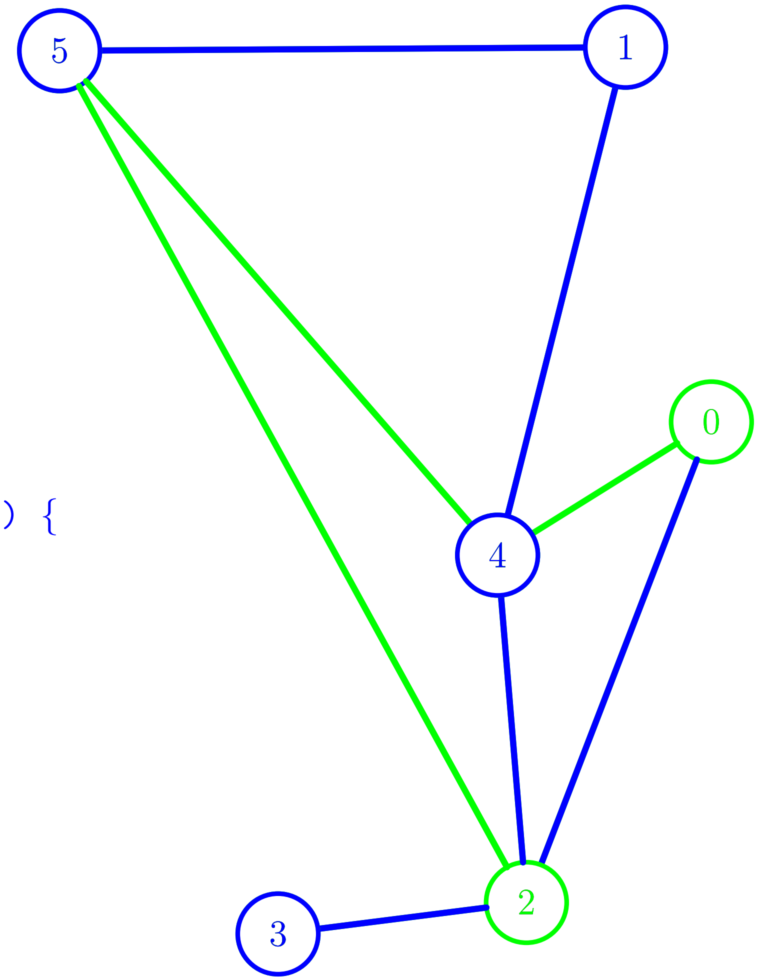
node=2 neighbour=5 time=10



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

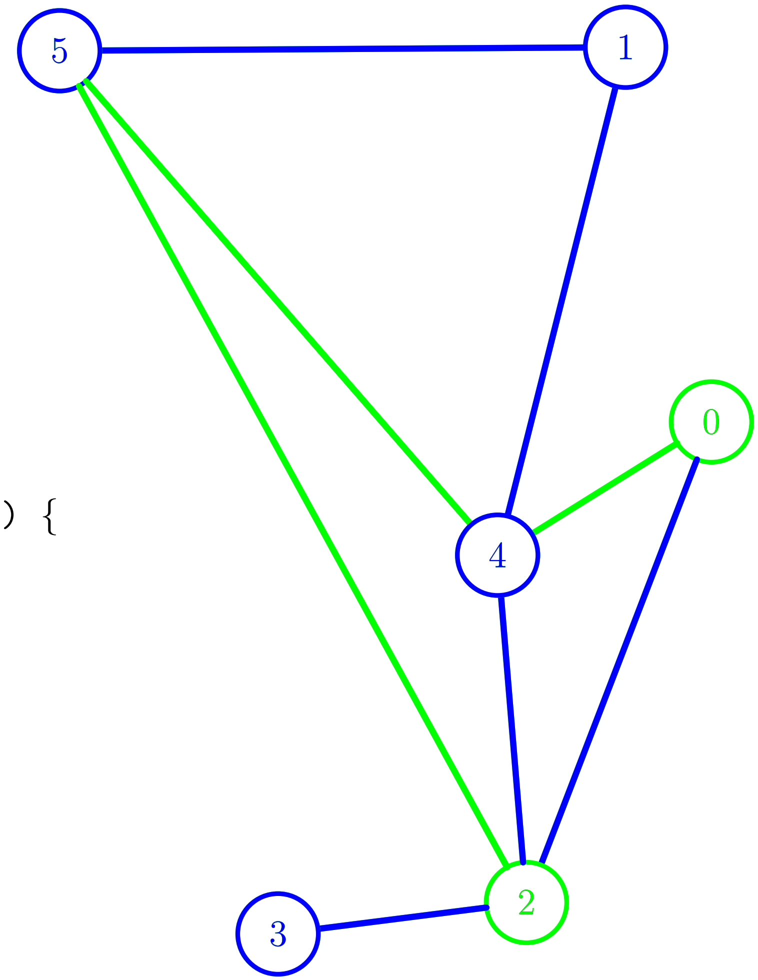
node=2   neighbour=5   time=10



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

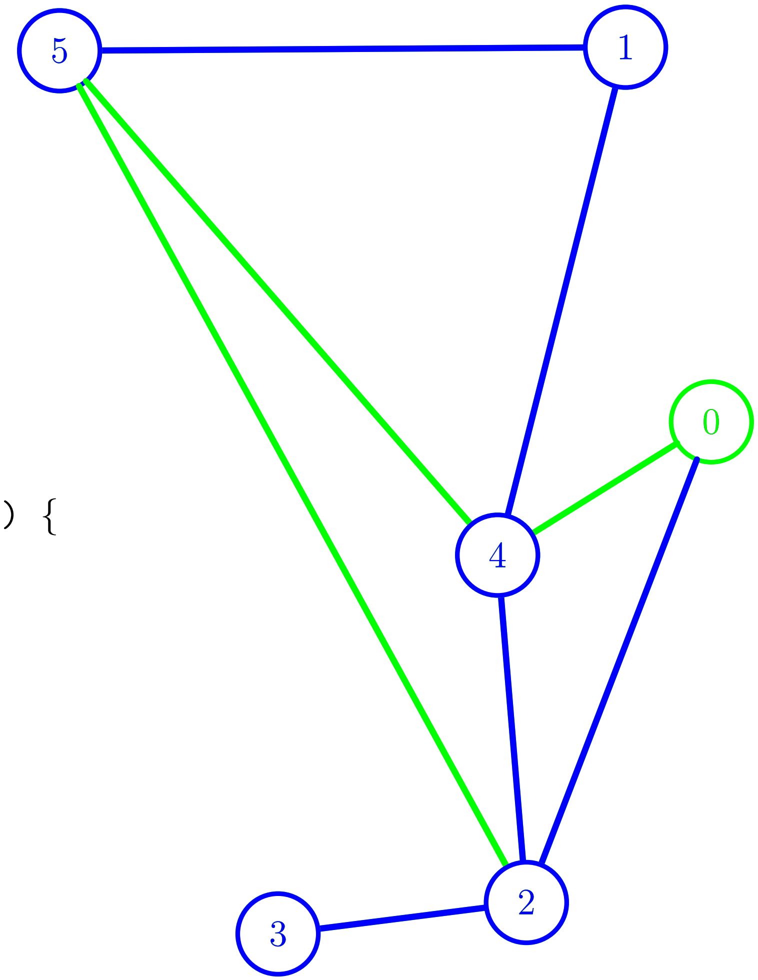
node=2   neighbour=5   time=10



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

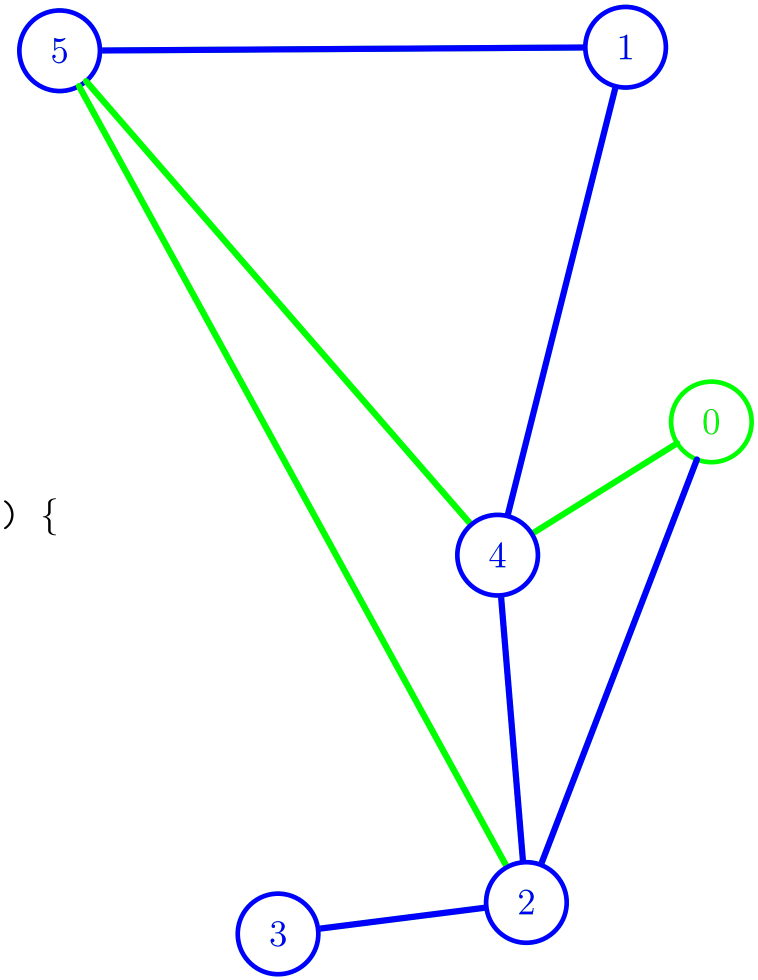
node=2 neighbour=nil time=11



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

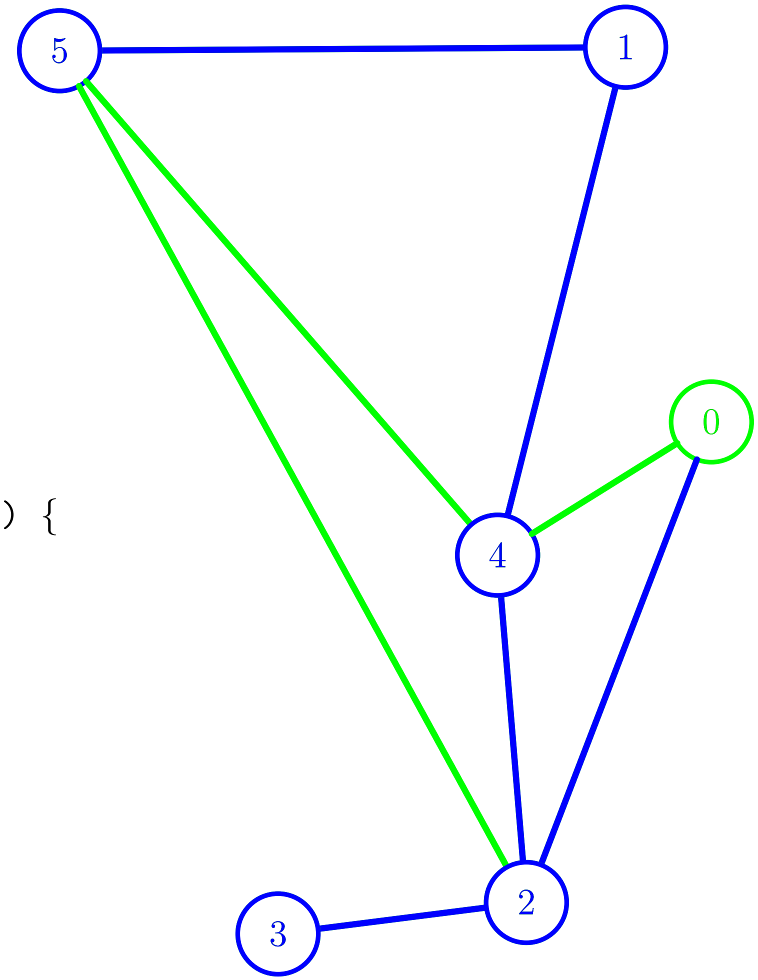
node=0   neighbour=2   time=11



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

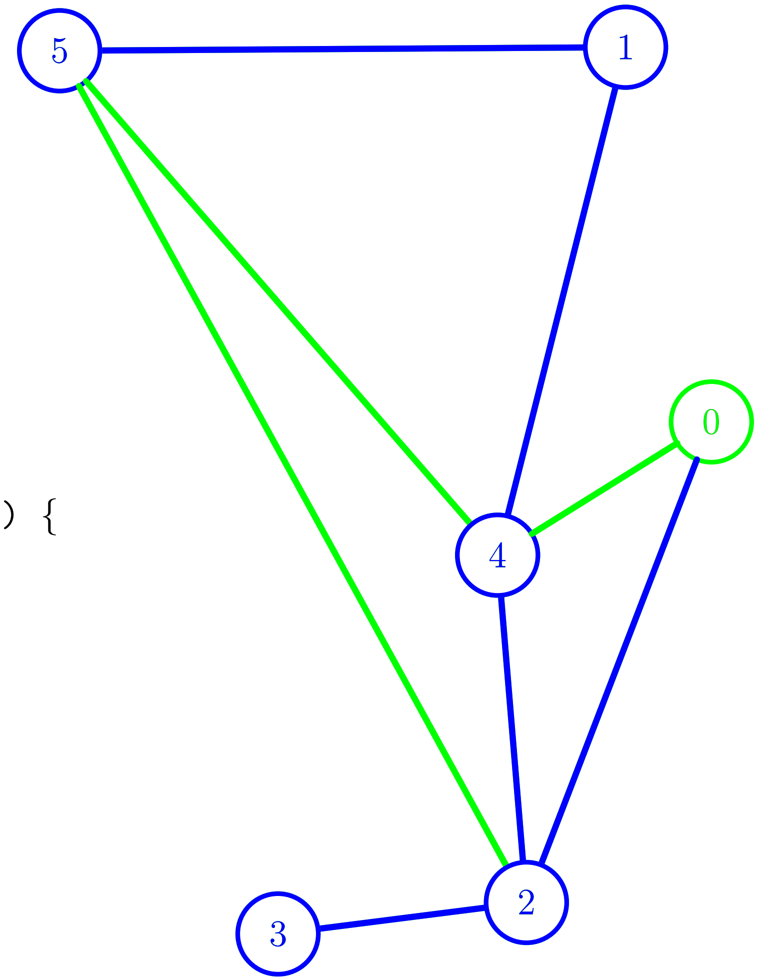
node=0 neighbour=4 time=11



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

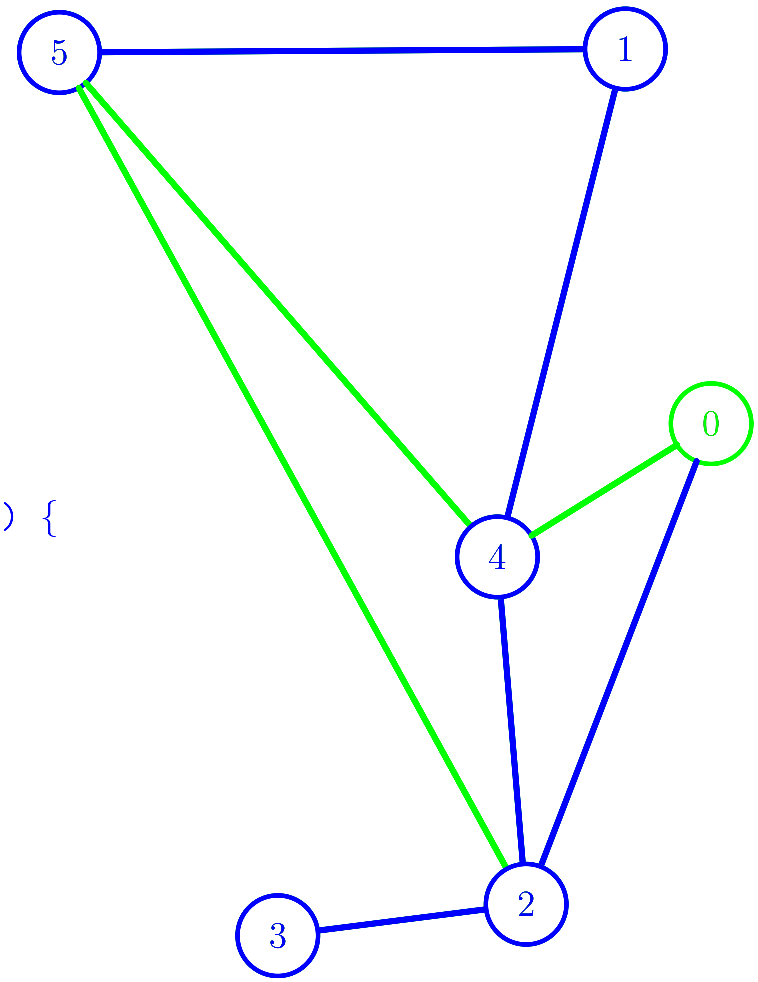
node=0 neighbour=4 time=11



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

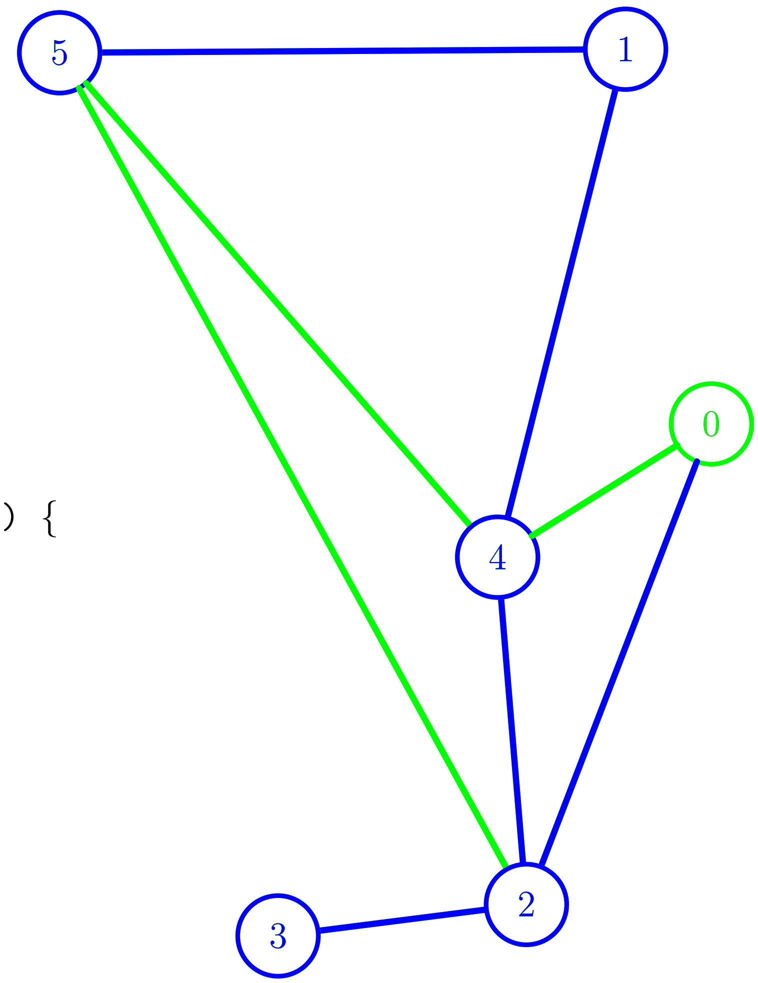
node=0 neighbour=4 time=11



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

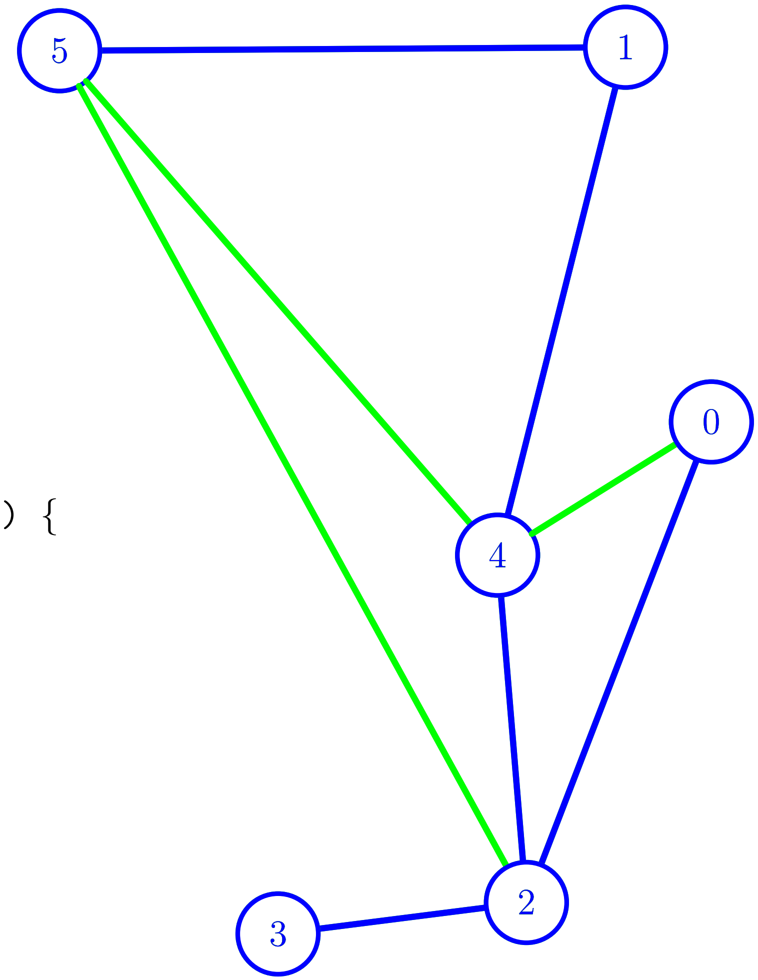
node=0 neighbour=4 time=11



Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

node=0 neighbour=nil time=12



Applications of DFS

- Depth first search has many applications
- Suppose we want to check if the graph is a tree (i.e. has no cycles)
- The only edges that are allowed are parent edges

```
processEdges(node1, node2) {
  if (parent[node1] ≠ node2) {
    isTree ←false
    finish ←true
  }
}
```

(note that we set `finish` to stop DFS prematurely)

Applications of DFS

- Depth first search has many applications
- Suppose we want to check if the graph is a tree (i.e. has no cycles)
- The only edges that are allowed are parent edges

```
processEdges(node1, node2) {
  if (parent[node1] ≠ node2) {
    isTree ←false
    finish ←true
  }
}
```

(note that we set `finish` to stop DFS prematurely)

Applications of DFS

- Depth first search has many applications
- Suppose we want to check if the graph is a tree (i.e. has no cycles)
- The only edges that are allowed are parent edges

```
processEdges(node1, node2) {
  if (parent[node1] ≠ node2) {
    isTree ←false
    finish ←true
  }
}
```

(note that we set `finish` to stop DFS prematurely)

Biconnected Graphs and Articulation Vertices

- If we removed a vertex (and all its edges) from a graph would the graph become disconnected?
- Such nodes are called **articulation vertices**
- Graphs with no articulation vertices are said to be **biconnected**
- In many applications articulation vertices are important, e.g. they represent single point of failures in communication networks
- A brute force method for identifying articulation vertices is to remove each and check for connectivity—this would take $O(n(m+n))$ time

Biconnected Graphs and Articulation Vertices

- If we removed a vertex (and all its edges) from a graph would the graph become disconnected?

- Such nodes are called **articulation vertices**

- Graphs with no articulation vertices are said to be **biconnected**

- In many applications articulation vertices are important, e.g. they represent single point of failures in communication networks

- A brute force method for identifying articulation vertices is to remove each and check for connectivity—this would take $O(n(m+n))$ time

Biconnected Graphs and Articulation Vertices

- If we removed a vertex (and all its edges) from a graph would the graph become disconnected?

- Such nodes are called **articulation vertices**

- Graphs with no articulation vertices are said to be **biconnected**

- In many applications articulation vertices are important, e.g. they represent single point of failures in communication networks

- A brute force method for identifying articulation vertices is to remove each and check for connectivity—this would take $O(n(m+n))$ time

Biconnected Graphs and Articulation Vertices

- If we removed a vertex (and all its edges) from a graph would the graph become disconnected?
- Such nodes are called **articulation vertices**
- Graphs with no articulation vertices are said to be **biconnected**
- In many applications articulation vertices are important, e.g. they represent single point of failures in communication networks
- A brute force method for identifying articulation vertices is to remove each and check for connectivity—this would take $O(n(m+n))$ time

Biconnected Graphs and Articulation Vertices

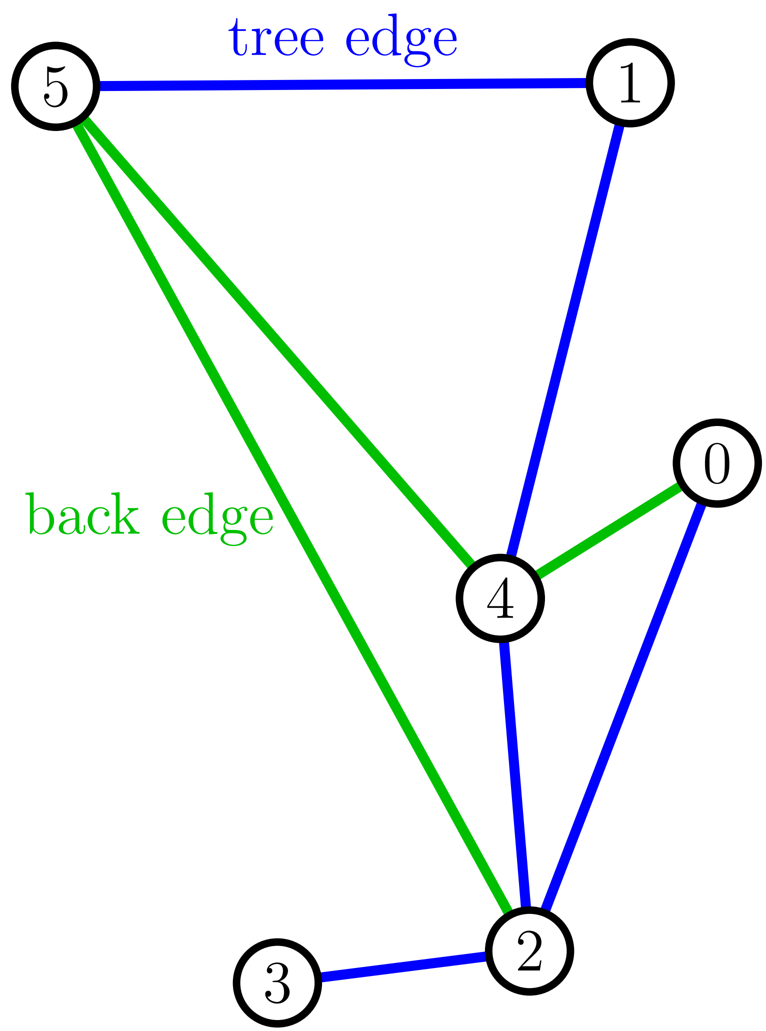
- If we removed a vertex (and all its edges) from a graph would the graph become disconnected?
- Such nodes are called **articulation vertices**
- Graphs with no articulation vertices are said to be **biconnected**
- In many applications articulation vertices are important, e.g. they represent single point of failures in communication networks
- A brute force method for identifying articulation vertices is to remove each and check for connectivity—this would take $O(n(m+n))$ time

Biconnected Graphs and Articulation Vertices

- If we removed a vertex (and all its edges) from a graph would the graph become disconnected?
- Such nodes are called **articulation vertices**
- Graphs with no articulation vertices are said to be **biconnected**
- In many applications articulation vertices are important, e.g. they represent single point of failures in communication networks
- A brute force method for identifying articulation vertices is to remove each and check for connectivity—this would take $O(n(m+n))$ time. Can we do this any faster?

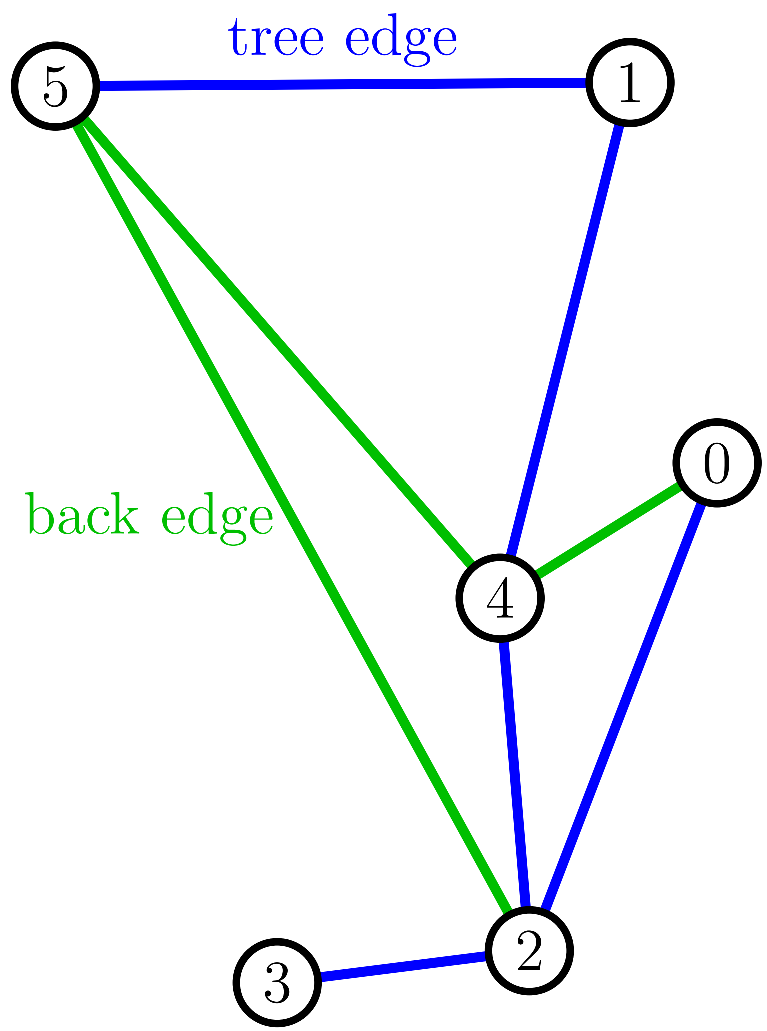
Single Pass Algorithm

- In DFS we divide the edges into tree edges that define the search tree (edges between nodes and parents) and back edges which take us back to vertices we have already seen
- Without the back edges we have a tree where all non-leaf nodes are articulation nodes
- The back edges secure the edges to the rest of the tree



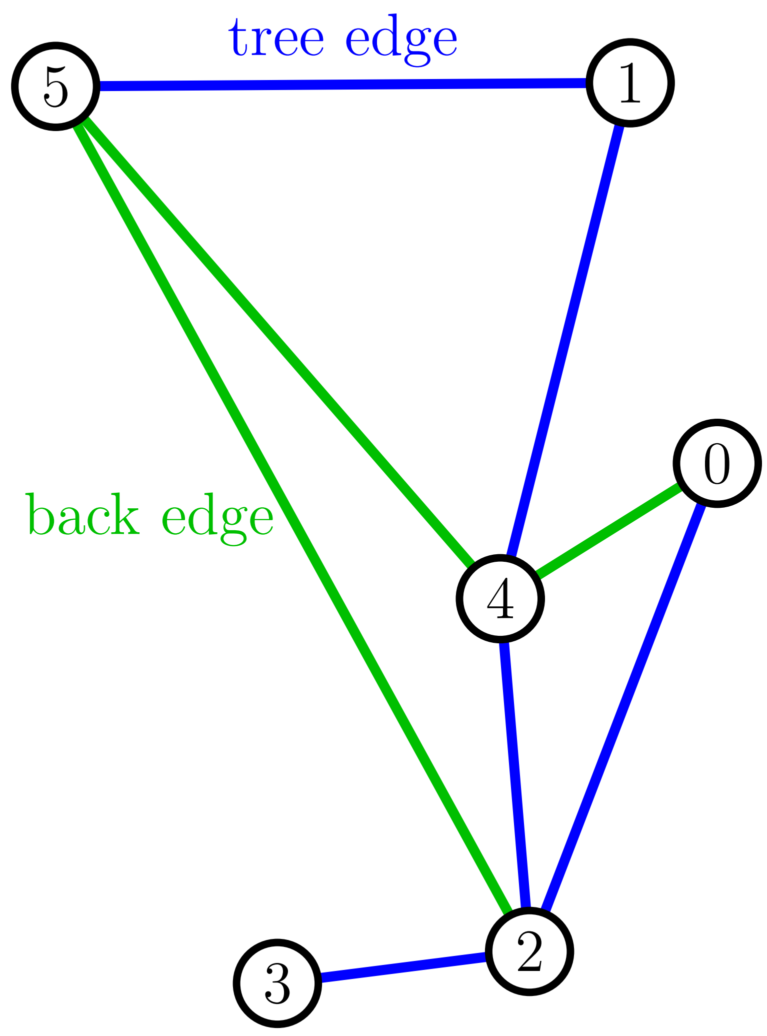
Single Pass Algorithm

- In DFS we divide the edges into tree edges that define the search tree (edges between nodes and parents) and back edges which take us back to vertices we have already seen
- Without the back edges we have a tree where all non-leaf nodes are articulation nodes
- The back edges secure the edges to the rest of the tree



Single Pass Algorithm

- In DFS we divide the edges into tree edges that define the search tree (edges between nodes and parents) and back edges which take us back to vertices we have already seen
- Without the back edges we have a tree where all non-leaf nodes are articulation nodes
- The back edges secure the edges to the rest of the tree



Reachable Ancestors

- Need to check if there exist back edges to nodes that have been visited earlier
- We maintain an array noting the reachable ancestors of all nodes
- This is initialised in the `processVertexEarly` method to the node itself
- In the `processEdge` method, if the edge is a

  back edge we update the reachable ancestor (we check the $entryTime$ to see if the edge leads to a vertex which was discovered earlier)

  tree edge we maintain a count of the number of tree edges connected to the vertex (used to determine if we are at a leaf node)

Reachable Ancestors

- Need to check if there exist back edges to nodes that have been visited earlier
- We maintain an array noting the reachable ancestors of all nodes
- This is initialised in the `processVertexEarly` method to the node itself
- In the `processEdge` method, if the edge is a

  back edge we update the reachable ancestor (we check the $entryTime$ to see if the edge leads to a vertex which was discovered earlier)

  tree edge we maintain a count of the number of tree edges connected to the vertex (used to determine if we are at a leaf node)

Reachable Ancestors

- Need to check if there exist back edges to nodes that have been visited earlier
- We maintain an array noting the reachable ancestors of all nodes
- This is initialised in the `processVertexEarly` method to the node itself
- In the `processEdge` method, if the edge is a

back edge we update the reachable ancestor (we check the $entryTime$ to see if the edge leads to a vertex which was discovered earlier)

tree edge we maintain a count of the number of tree edges connected to the vertex (used to determine if we are at a leaf node)

Reachable Ancestors

- Need to check if there exist back edges to nodes that have been visited earlier
- We maintain an array noting the reachable ancestors of all nodes
- This is initialised in the `processVertexEarly` method to the node itself
- In the `processEdge` method, if the edge is a

  back edge we update the reachable ancestor (we check the $entryTime$ to see if the edge leads to a vertex which was discovered earlier)

  tree edge we maintain a count of the number of tree edges connected to the vertex (used to determine if we are at a leaf node)

Reachable Ancestors

- Need to check if there exist back edges to nodes that have been visited earlier
- We maintain an array noting the reachable ancestors of all nodes
- This is initialised in the `processVertexEarly` method to the node itself
- In the `processEdge` method, if the edge is a

back edge we update the reachable ancestor (we check the $entryTime$ to see if the edge leads to a vertex which was discovered earlier)

tree edge we maintain a count of the number of tree edges connected to the vertex (used to determine if we are at a leaf node)

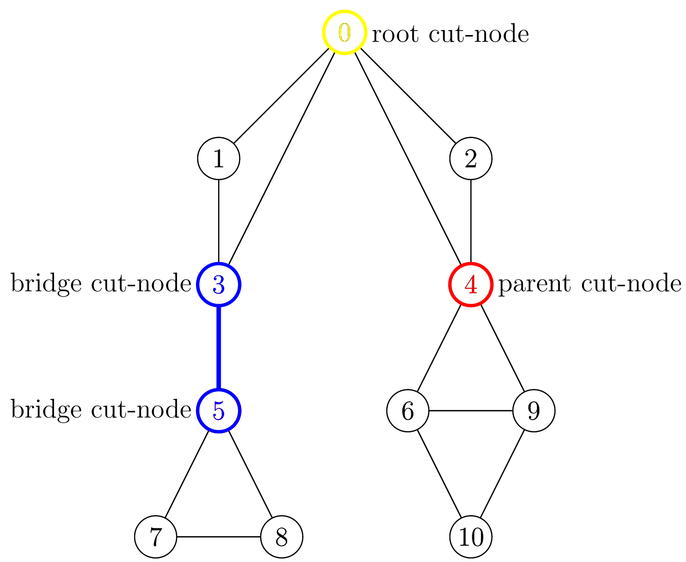
Types of Articulated Vertices

- Key is to recognise that articulated vertices only occur in three version

Root cut-nodes Occur when the root has more than one child

Bridge cut-nodes Occurs when the earliest reachable vertex (not including the tree edge to the parent) is the vertex itself. The parent will be an articulation node as will be the node itself if it is not a leaf node

Parent cut-nodes If the earliest reachable vertex is it parent then the parent is an articulation node



- These are determined in `processVertexLate` method.

Types of Articulated Vertices

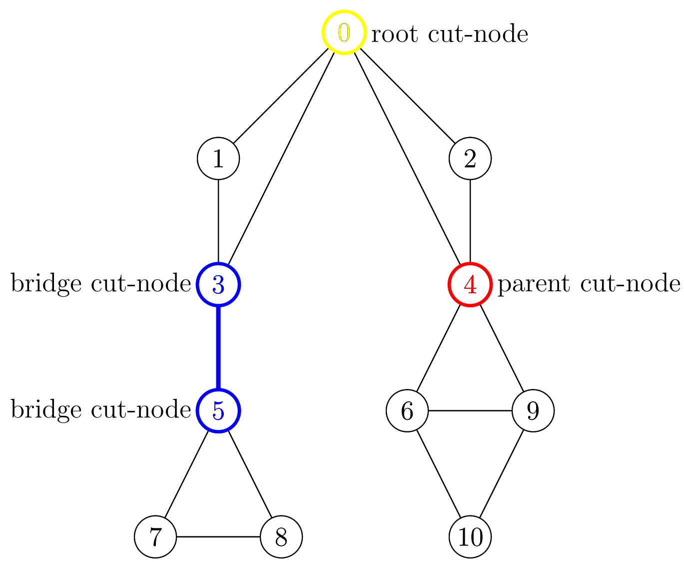
- Key is to recognise that articulated vertices only occur in three version

  Root cut-nodes Occur when the root has more than one child

  Bridge cut-nodes Occurs when the earliest reachable vertex (not including the tree edge to the parent) is the vertex itself. The parent will be an articulation node as will be the node itself if it is not a leaf node

  Parent cut-nodes If the earliest reachable vertex is it parent then the parent is an articulation node

- These are determined in `processVertexLate` method.



Types of Articulated Vertices

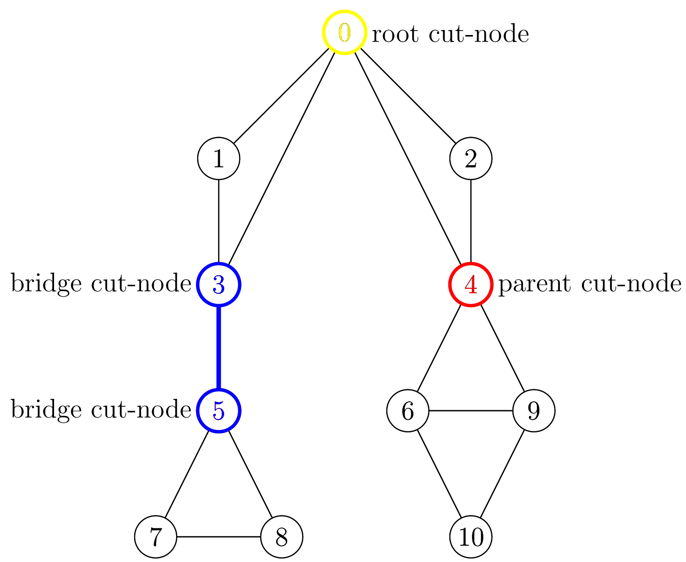
- Key is to recognise that articulated vertices only occur in three version

  Root cut-nodes Occur when the root has more than one child

  Bridge cut-nodes Occurs when the earliest reachable vertex (not including the tree edge to the parent) is the vertex itself. The parent will be an articulation node as will be the node itself if it is not a leaf node

  Parent cut-nodes If the earliest reachable vertex is it parent then the parent is an articulation node

- These are determined in `processVertexLate` method.



Types of Articulated Vertices

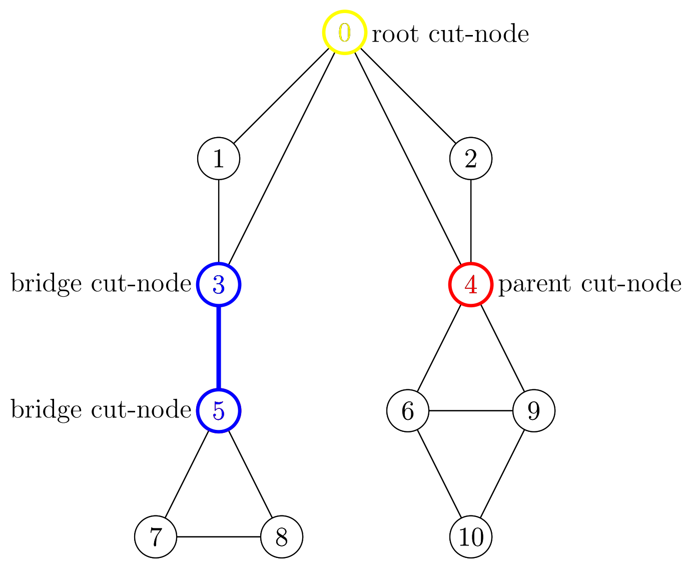
- Key is to recognise that articulated vertices only occur in three version

  Root cut-nodes Occur when the root has more than one child

  Bridge cut-nodes Occurs when the earliest reachable vertex (not including the tree edge to the parent) is the vertex itself. The parent will be an articulation node as will be the node itself if it is not a leaf node

  Parent cut-nodes If the earliest reachable vertex is it parent then the parent is an articulation node

- These are determined in `processVertexLate` method.



Types of Articulated Vertices

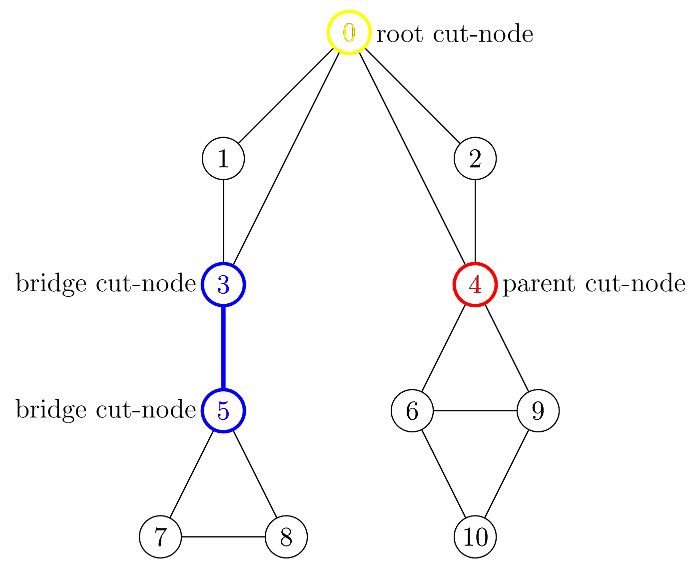
- Key is to recognise that articulated vertices only occur in three version

  Root cut-nodes Occur when the root has more than one child

  Bridge cut-nodes Occurs when the earliest reachable vertex (not including the tree edge to the parent) is the vertex itself. The parent will be an articulation node as will be the node itself if it is not a leaf node

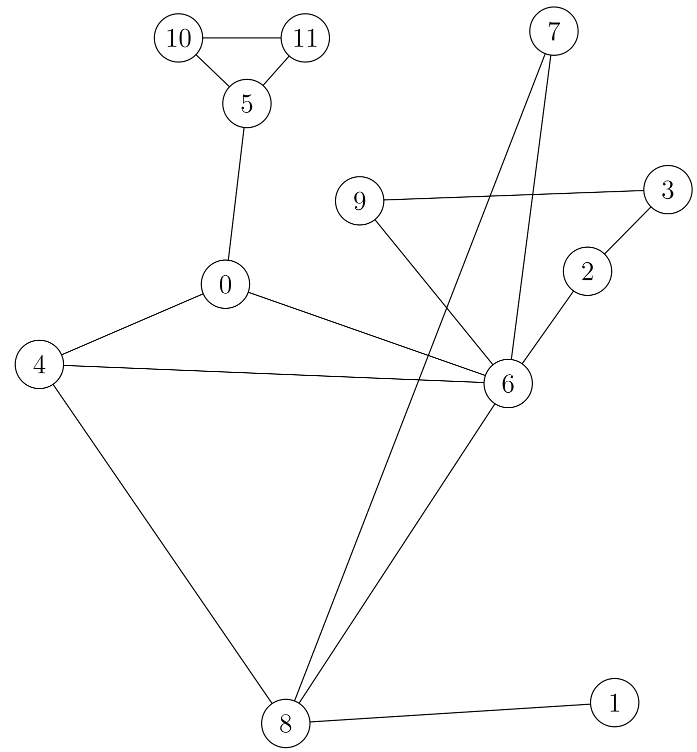
  Parent cut-nodes If the earliest reachable vertex is it parent then the parent is an articulation node

- These are determined in `processVertexLate` method.



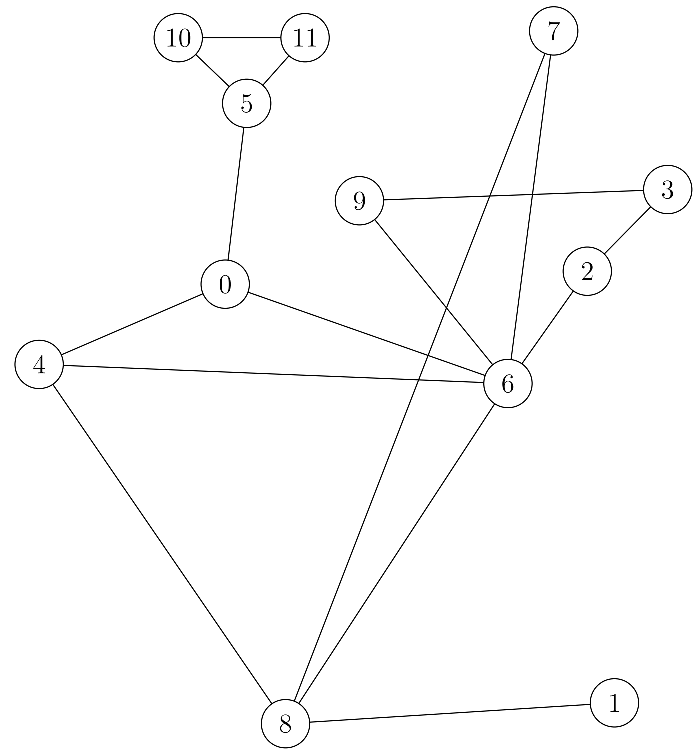
Biconnectivity Summary

- Algorithmic details are not too important
- One pass ($O(n+m)$) algorithm
- Uses `processVertexEarly`, `procesEdge` and `processVerexLate` methods
- Bridge cuts also shows articulation edges



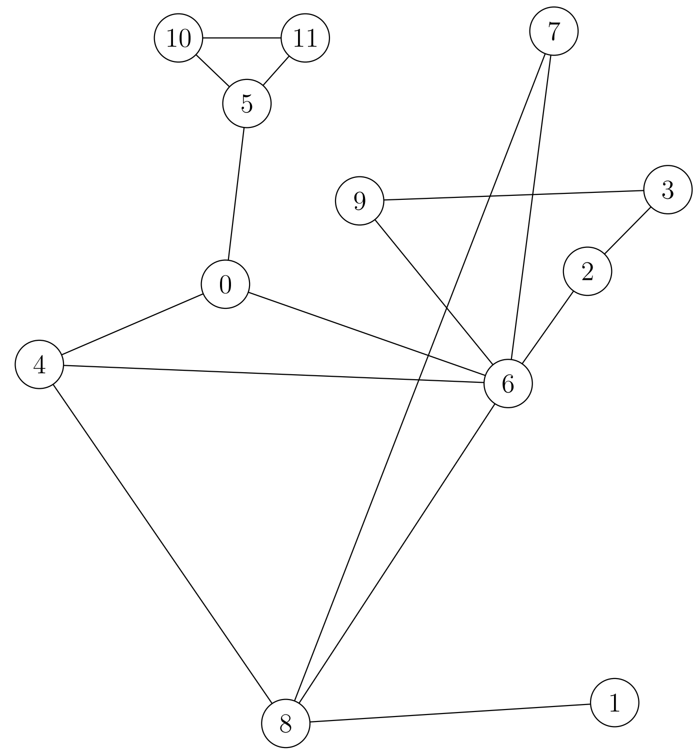
Biconnectivity Summary

- Algorithmic details are not too important
- One pass ($O(n+m)$) algorithm
- Uses `processVertexEarly`, `procesEdge` and `processVerexLate` methods
- Bridge cuts also shows articulation edges



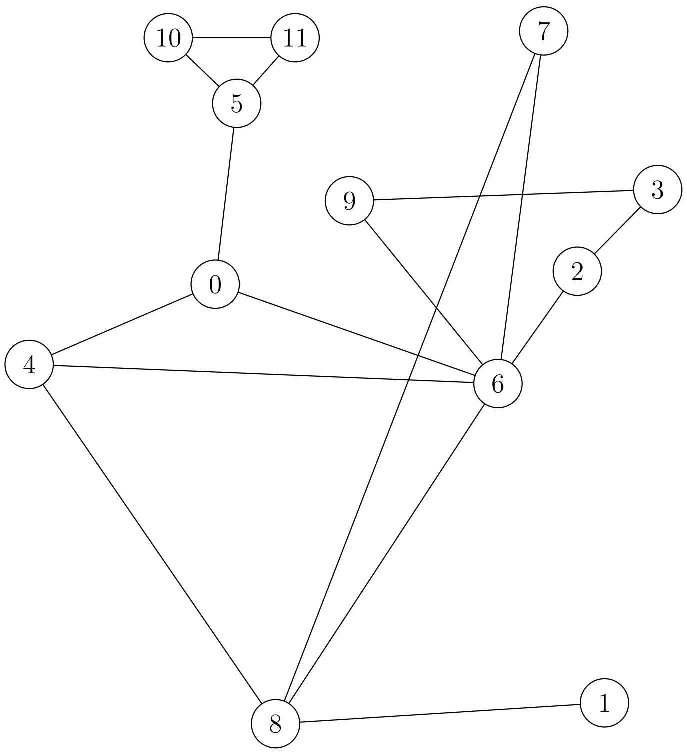
Biconnectivity Summary

- Algorithmic details are not too important
- One pass ($O(n+m)$) algorithm
- Uses `processVertexEarly`, `procesEdge` and `processVerexLate` methods
- Bridge cuts also shows articulation edges



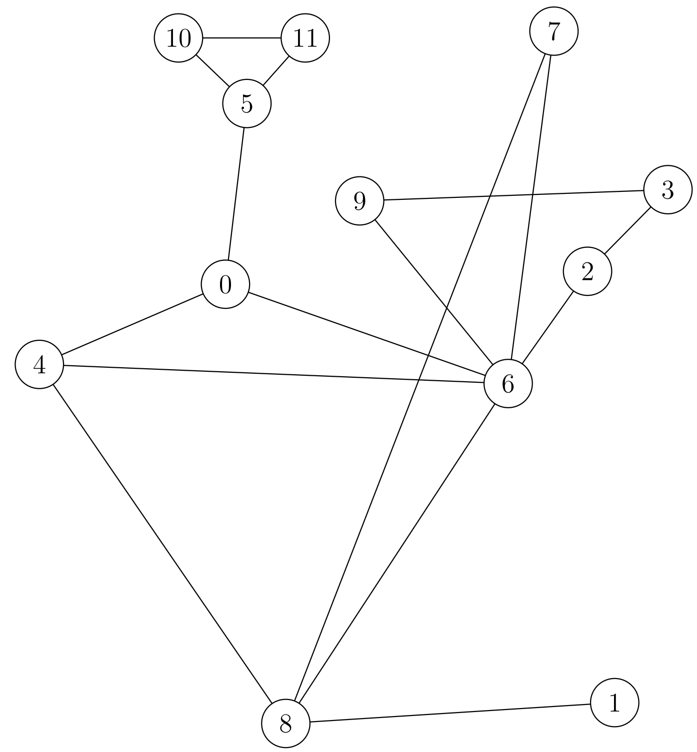
Biconnectivity Summary

- Algorithmic details are not too important
- One pass ($O(n+m)$) algorithm
- Uses `processVertexEarly`, `procesEdge` and `processVerexLate` methods
- Bridge cuts also shows articulation edges



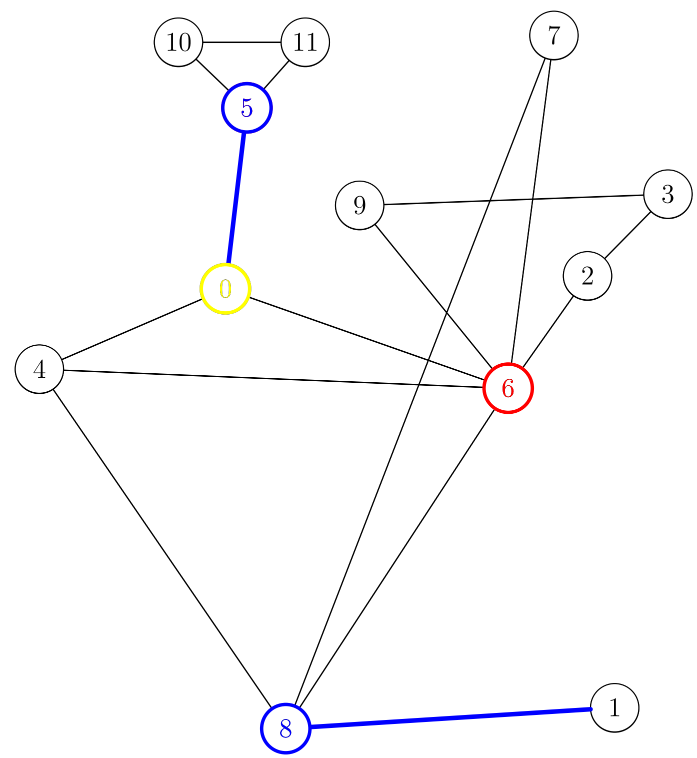
Biconnectivity Summary

- Algorithmic details are not too important
- One pass ($O(n+m)$) algorithm
- Uses `processVertexEarly`, `procesEdge` and `processVerexLate` methods
- Bridge cuts also shows articulation edges



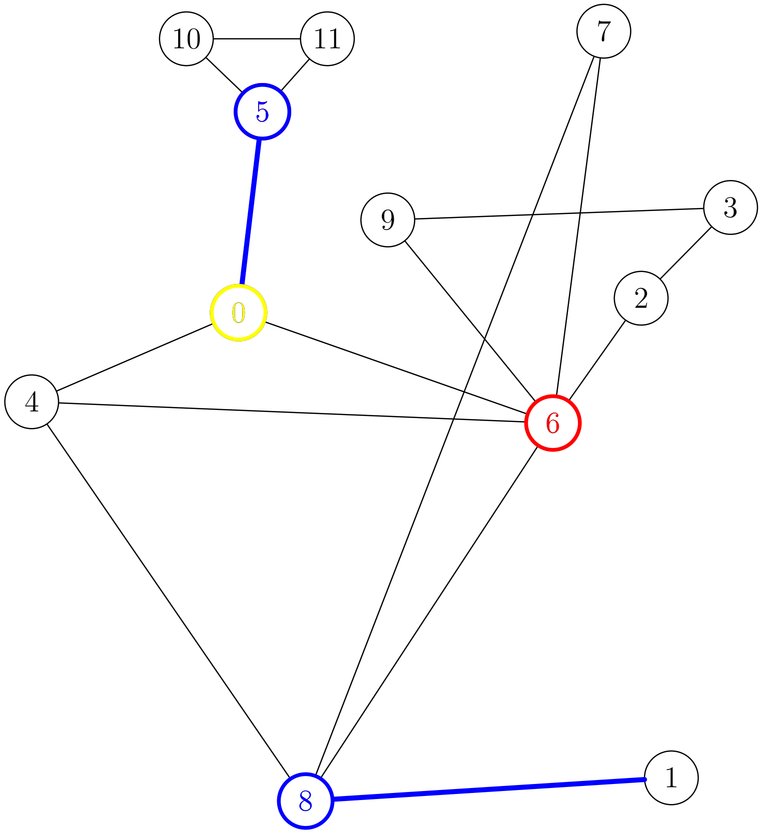
Biconnectivity Summary

- Algorithmic details are not too important
- One pass ($O(n+m)$) algorithm
- Uses `processVertexEarly`, `procesEdge` and `processVerexLate` methods
- Bridge cuts also shows articulation edges



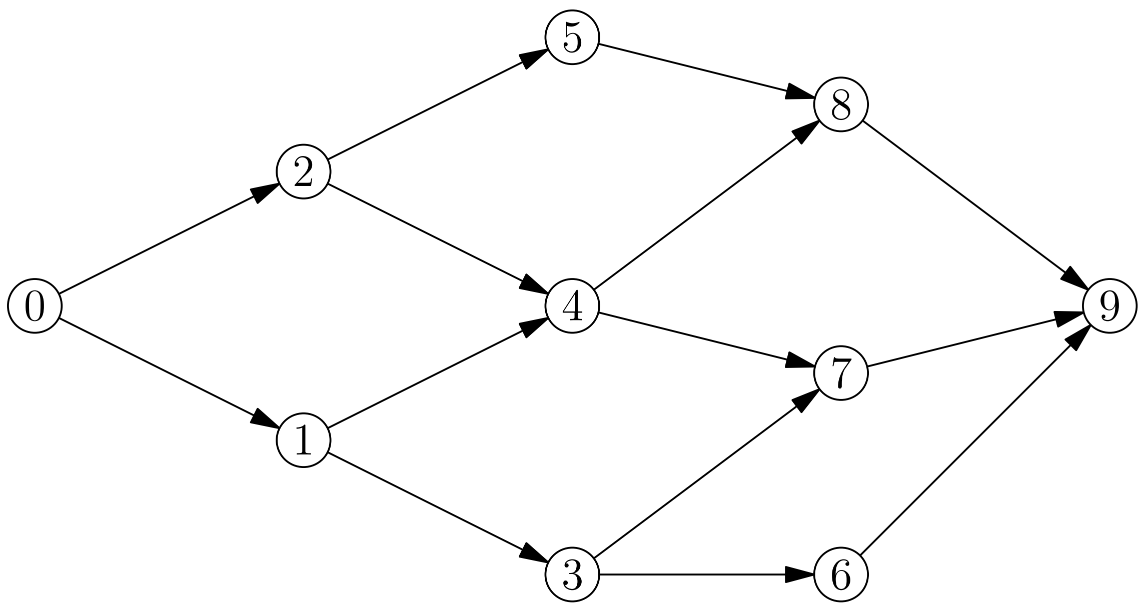
Outline

1. Breadth First Search
   - BFS applications
2. Depth First Search
   - DFS applications
3. **Topological Sort**



DAGs

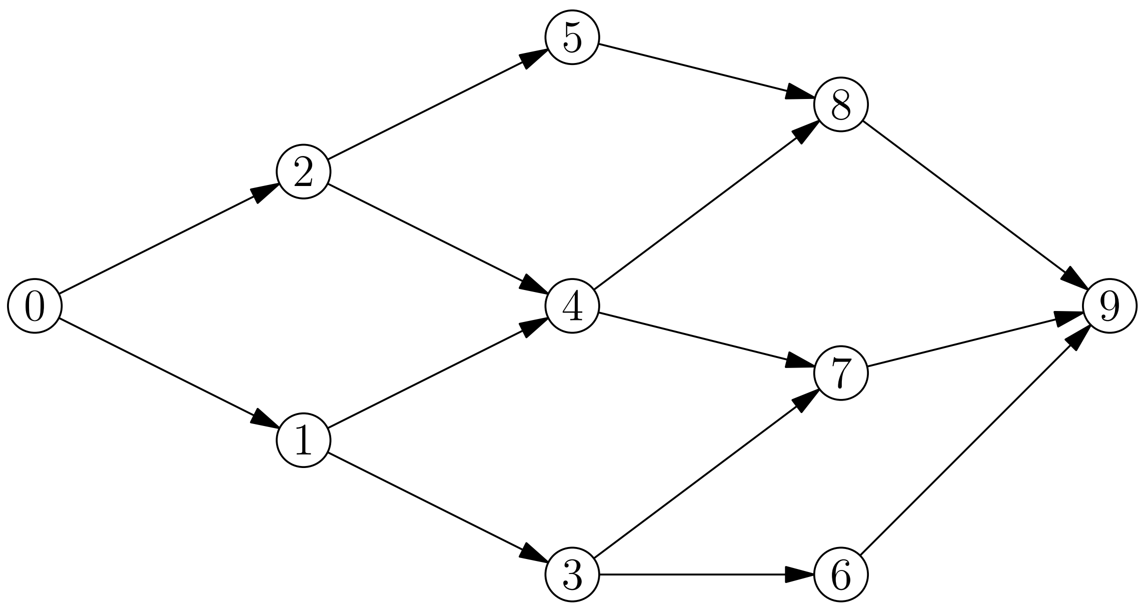
- Directed acyclic graphs or DAGs are directed graphs without cycles
- They are often used to represent complex processes
  - ⋆ Vertices are processes
  - ⋆ Directed edge (i, j) indicates process i needs to occur before process j



DAGs

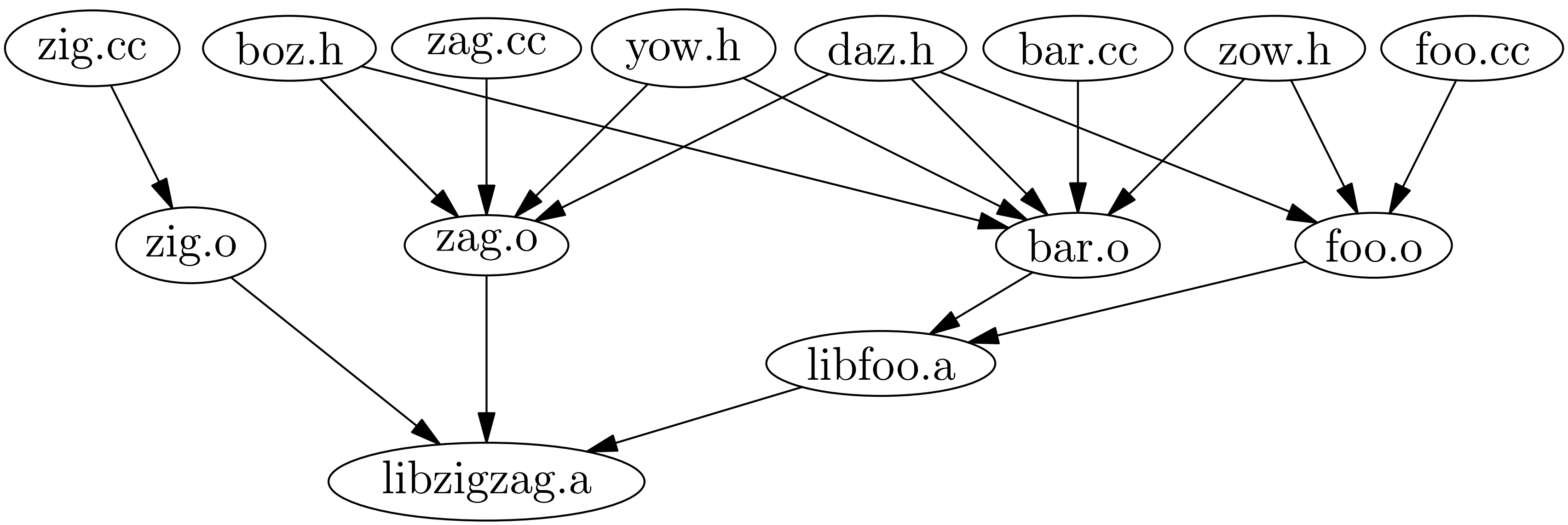
- Directed acyclic graphs or DAGs are directed graphs without cycles

- They are often used to represent complex processes
  - ⋆ Vertices are processes
  - ⋆ Directed edge (i, j) indicates process i needs to occur before process j



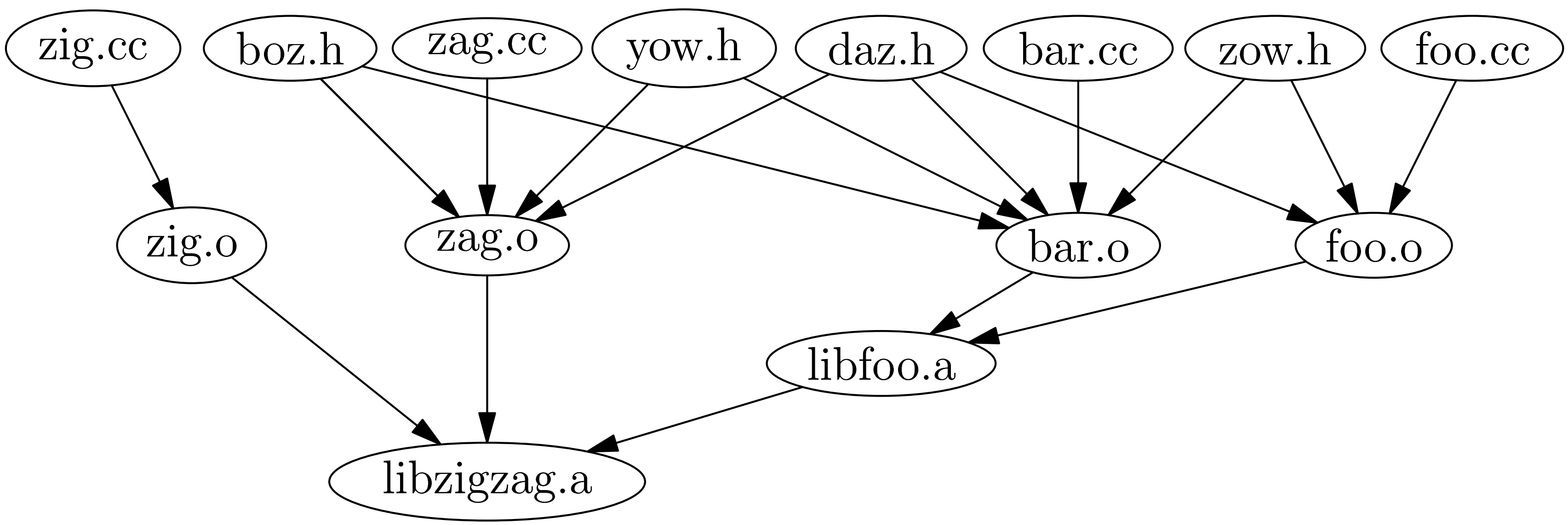
Program Compilation

- One example of a DAG is in compiling programs
- Some programs depend on other programs so they need compiling first



Program Compilation

- One example of a DAG is in compiling programs
- Some programs depend on other programs so they need compiling first



Other Applications

- The same problem occurs in compiling classes
  - ⋆ The implementation of a class can depend on the implementation of other graphs
  - ⋆ What order should you compile the classes?
- In taking a degree various modules have other modules as prerequisites
- In your graduation ceremony there is the VC, Provost, Dean, HOS, Professors, etc.

Other Applications

- The same problem occurs in compiling classes
  - ⋆ The implementation of a class can depend on the implementation of other graphs
  - ⋆ What order should you compile the classes?

- In taking a degree various modules have other modules as prerequisites

- In your graduation ceremony there is the VC, Provost, Dean, HOS, Professors, etc.

Other Applications

- The same problem occurs in compiling classes
  - ⋆ The implementation of a class can depend on the implementation of other graphs
  - ⋆ What order should you compile the classes?
- In taking a degree various modules have other modules as prerequisites
- In your graduation ceremony there is the VC, Provost, Dean, HOS, Professors, etc.

Other Applications

- The same problem occurs in compiling classes
  - ⋆ The implementation of a class can depend on the implementation of other graphs
  - ⋆ What order should you compile the classes?
- In taking a degree various modules have other modules as prerequisites—what order should a student study the modules in?
- In your graduation ceremony there is the VC, Provost, Dean, HOS, Professors, etc.

Other Applications

- The same problem occurs in compiling classes
  - ⋆ The implementation of a class can depend on the implementation of other graphs
  - ⋆ What order should you compile the classes?
- In taking a degree various modules have other modules as prerequisites—what order should a student study the modules in?
- In your graduation ceremony there is the VC, Provost, Dean, HOS, Professors, etc.

Other Applications

- The same problem occurs in compiling classes
  - ⋆ The implementation of a class can depend on the implementation of other graphs
  - ⋆ What order should you compile the classes?
- In taking a degree various modules have other modules as prerequisites—what order should a student study the modules in?
- In your graduation ceremony there is the VC, Provost, Dean, HOS, Professors, etc.—in what order should they process?

Other Applications

- The same problem occurs in compiling classes
  - ⋆ The implementation of a class can depend on the implementation of other graphs
  - ⋆ What order should you compile the classes?

- In taking a degree various modules have other modules as prerequisites—what order should a student study the modules in?

- In your graduation ceremony there is the VC, Provost, Dean, HOS, Professors, etc.—in what order should they process?

- If the graphs were not acyclic it would be impossible process them!

Topological Sort

- Given a DAG a topological sort outputs an ordered list of vertices which respects the ordering imposed by the edges

- That is, for each edge (i, j), vertex i will occur before vertex j

- Any DAG will have at least one topological sort, but most DAGs will have many topological sorts

- Topological sort is not a "sort", but it is a useful algorithm for some applications

Topological Sort

- Given a DAG a topological sort outputs an ordered list of vertices which respects the ordering imposed by the edges
- That is, for each edge (i, j), vertex i will occur before vertex j
- Any DAG will have at least one topological sort, but most DAGs will have many topological sorts
- Topological sort is not a "sort", but it is a useful algorithm for some applications

Topological Sort

- Given a DAG a topological sort outputs an ordered list of vertices which respects the ordering imposed by the edges
- That is, for each edge (i, j), vertex i will occur before vertex j
- Any DAG will have at least one topological sort, but most DAGs will have many topological sorts
- Topological sort is not a "sort", but it is a useful algorithm for some applications

Topological Sort

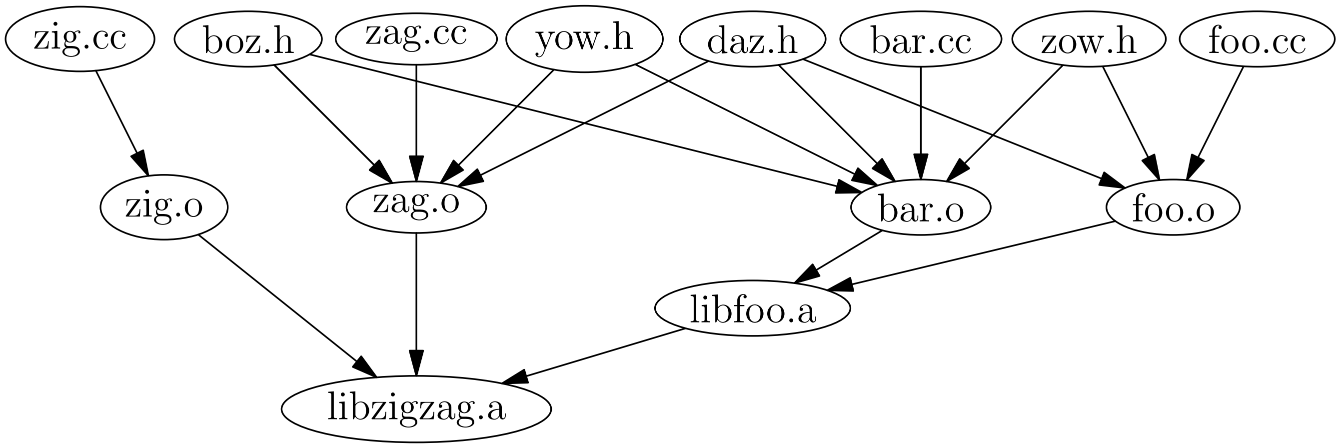
- Given a DAG a topological sort outputs an ordered list of vertices which respects the ordering imposed by the edges
- That is, for each edge (i, j), vertex i will occur before vertex j
- Any DAG will have at least one topological sort, but most DAGs will have many topological sorts
- Topological sort is not a "sort", but it is a useful algorithm for some applications

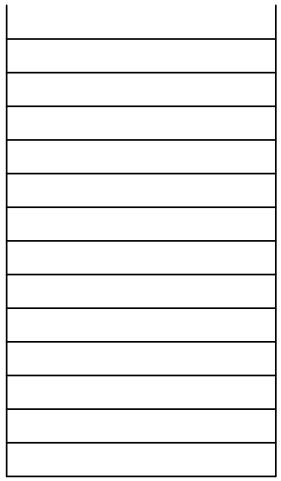
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes

Performing a Topological Sort

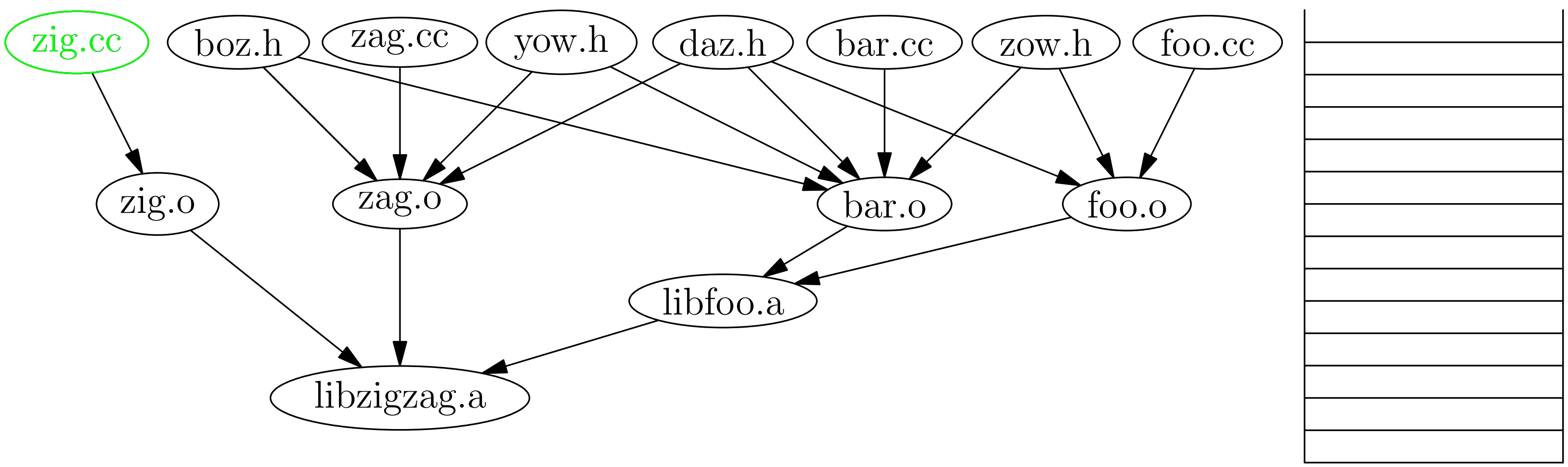
- A topological sort is generated by a reversed order list of how DFS processes nodes





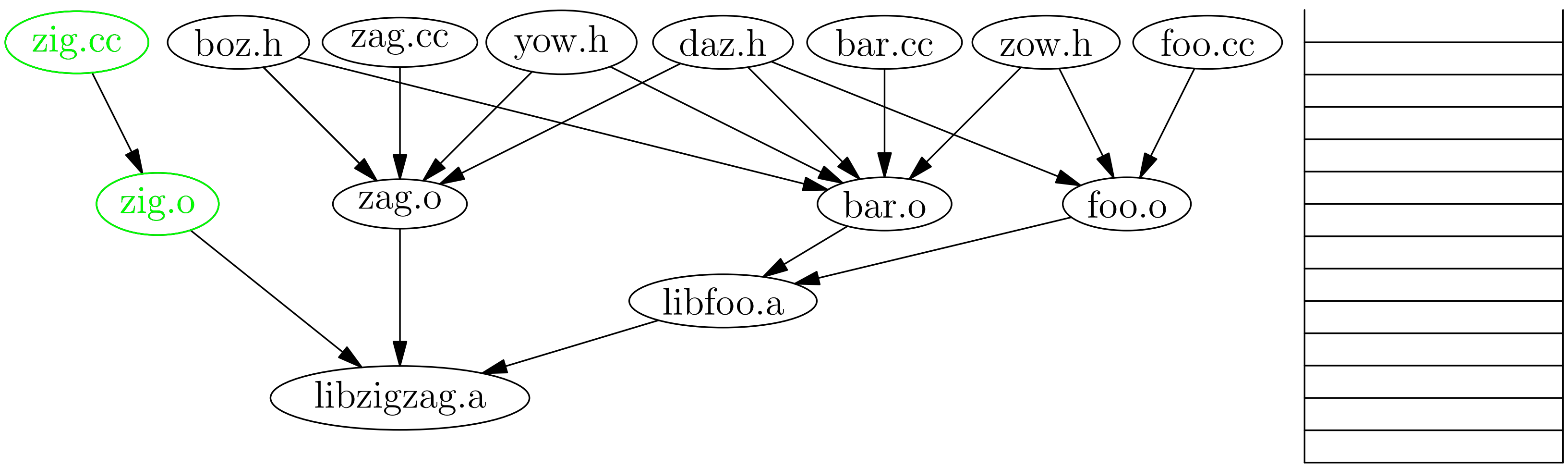
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



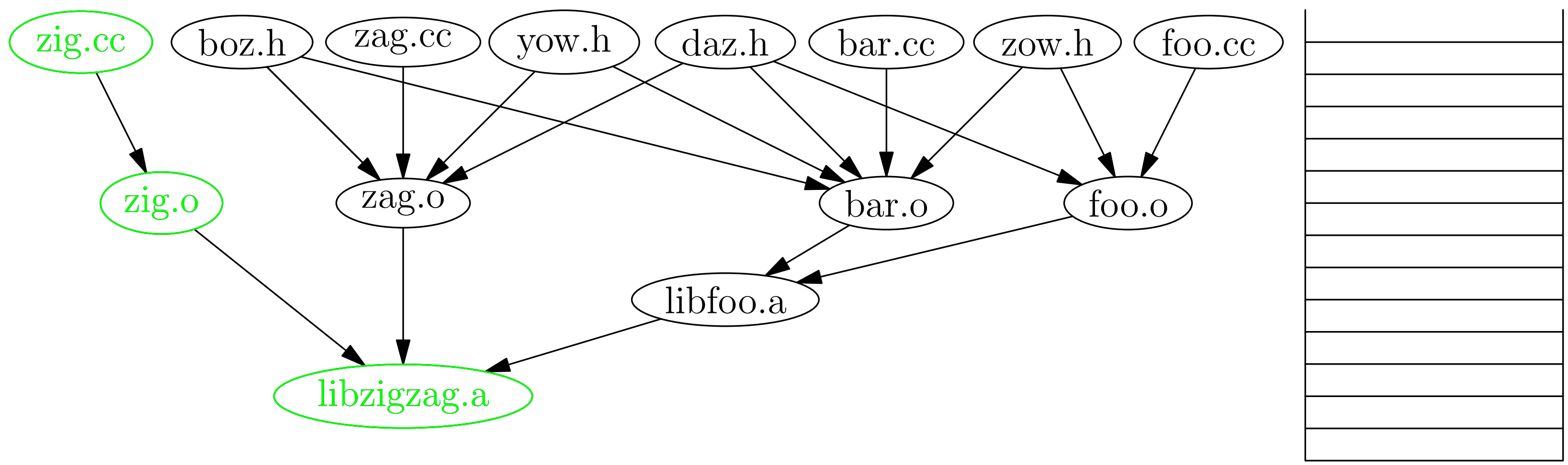
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



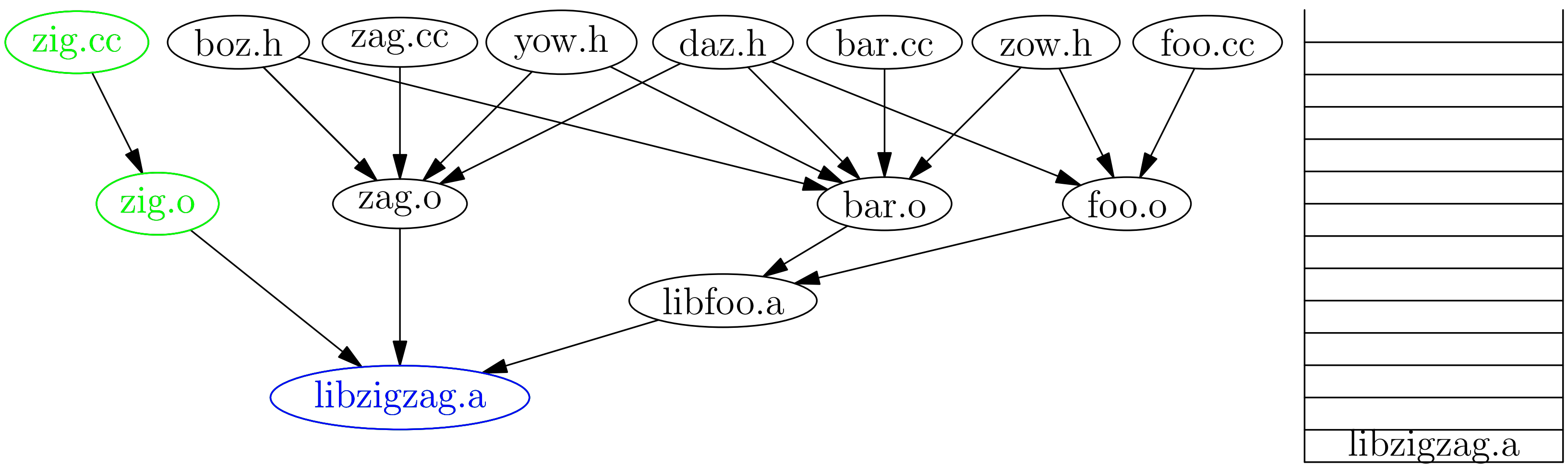
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



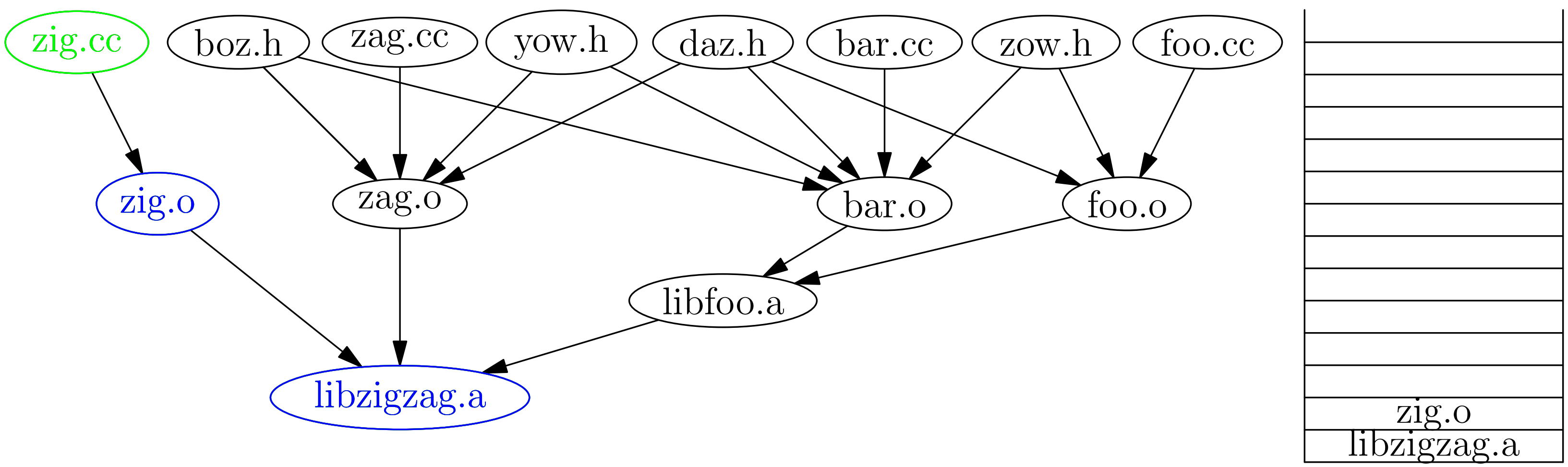
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



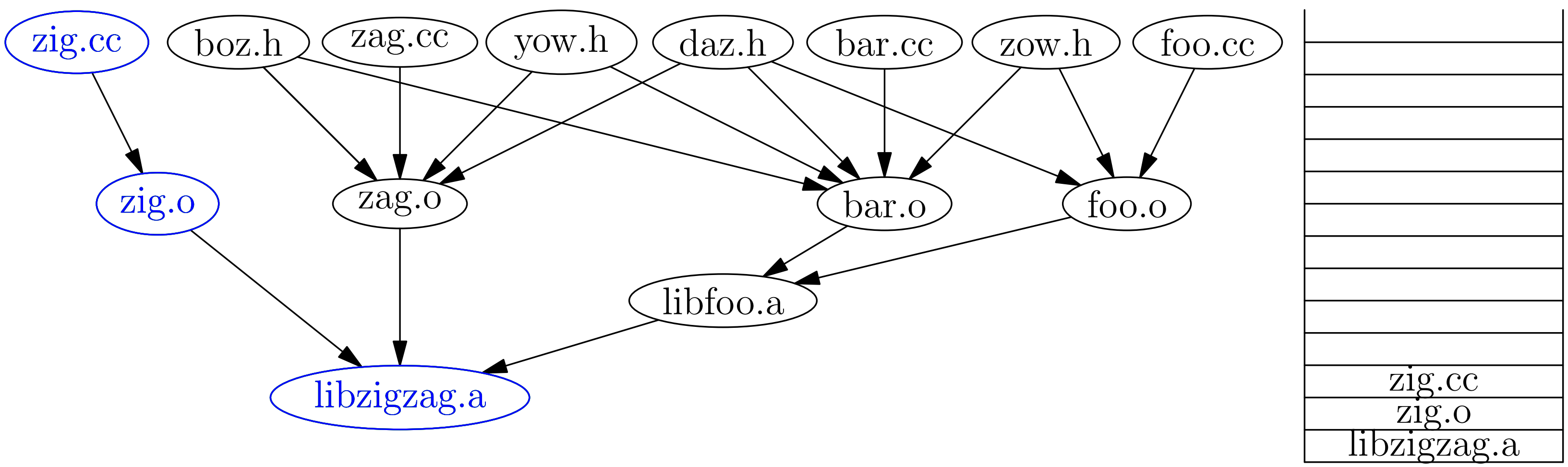
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



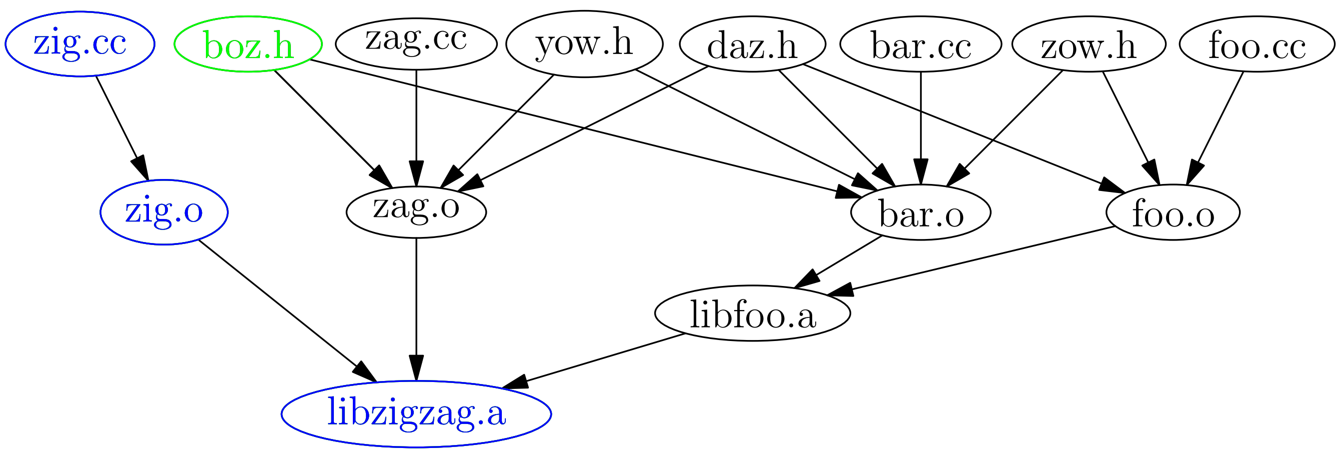
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



Performing a Topological Sort

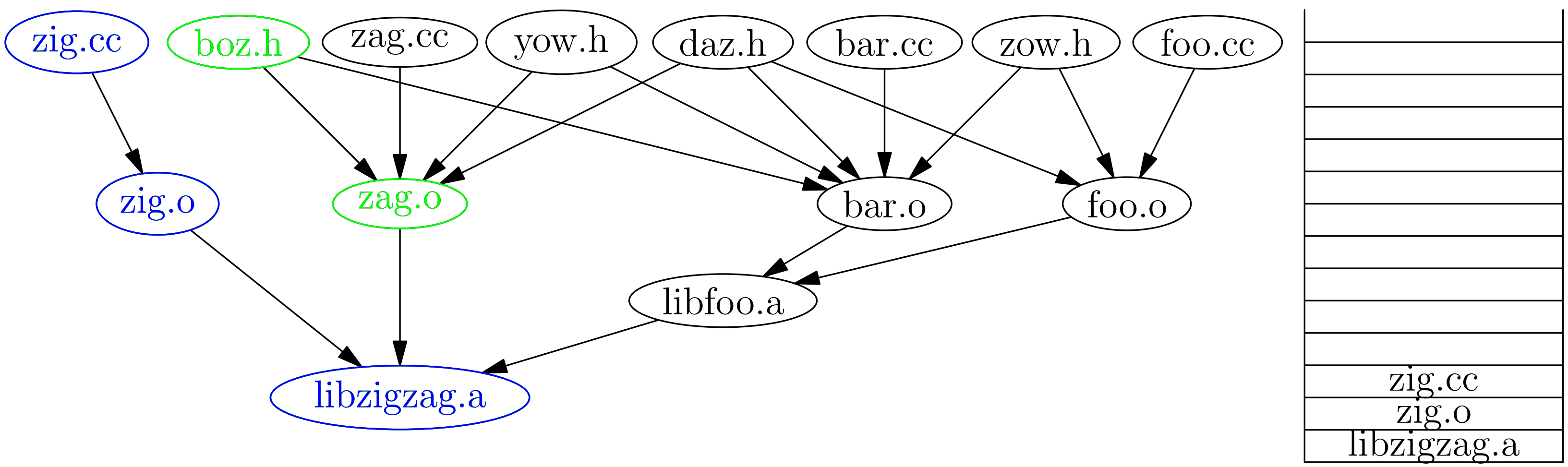
- A topological sort is generated by a reversed order list of how DFS processes nodes



zig.cc
zig.o
libzigzag.a

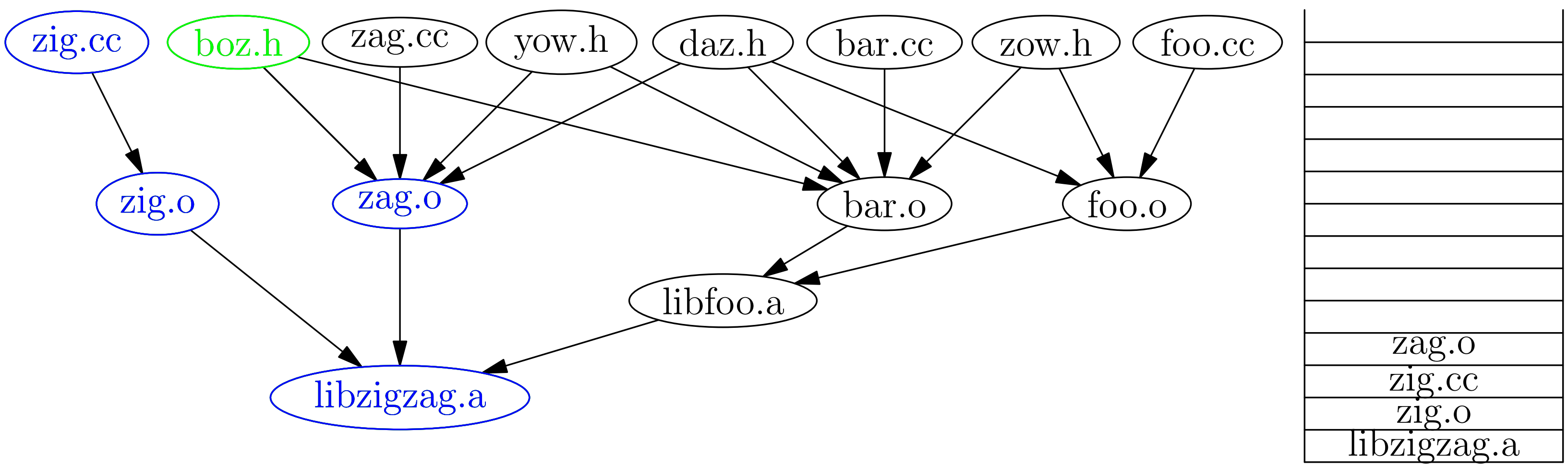
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



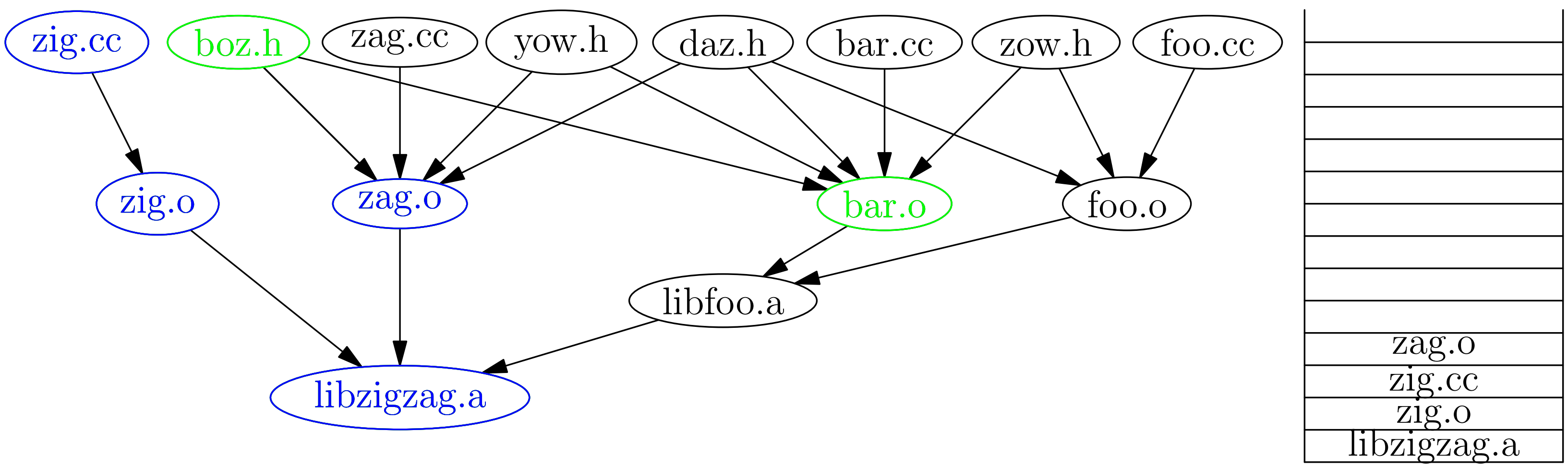
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



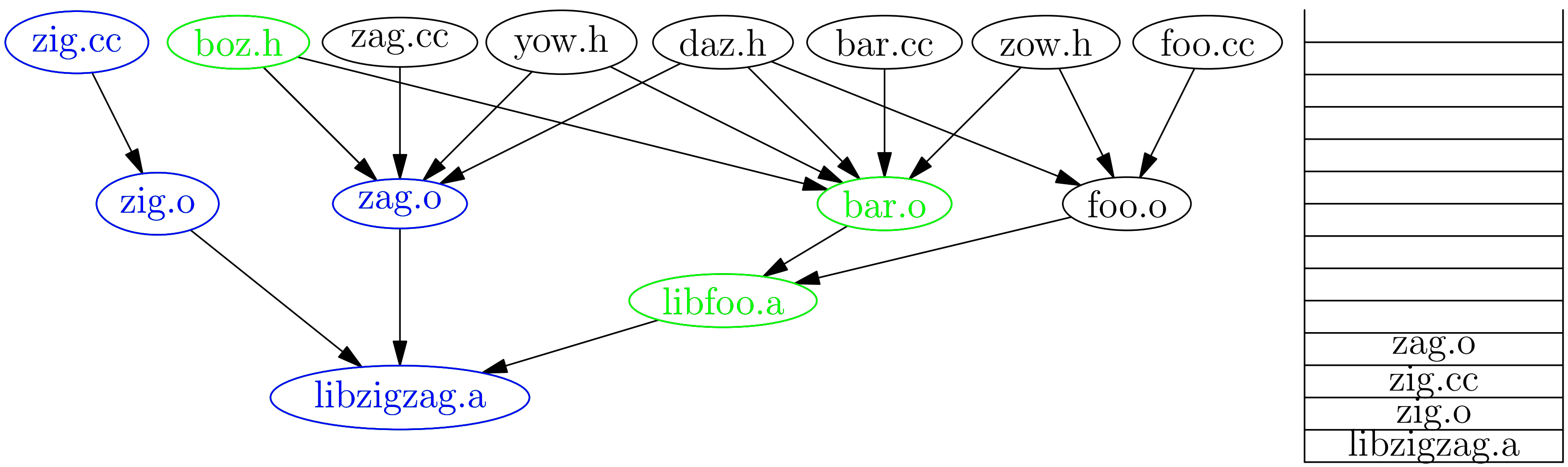
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



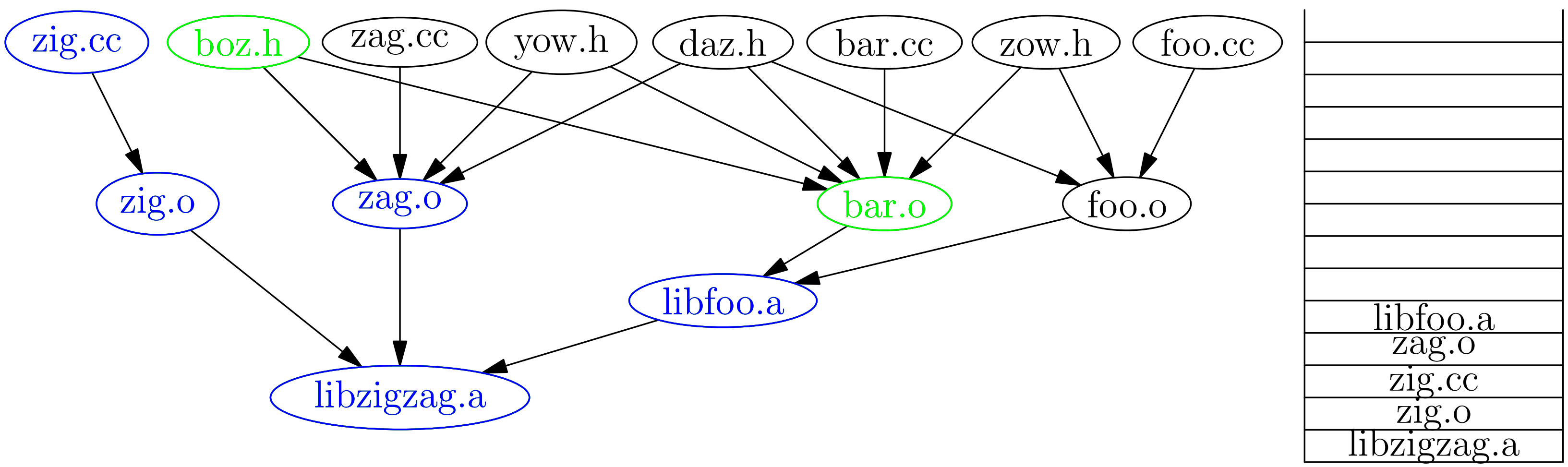
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



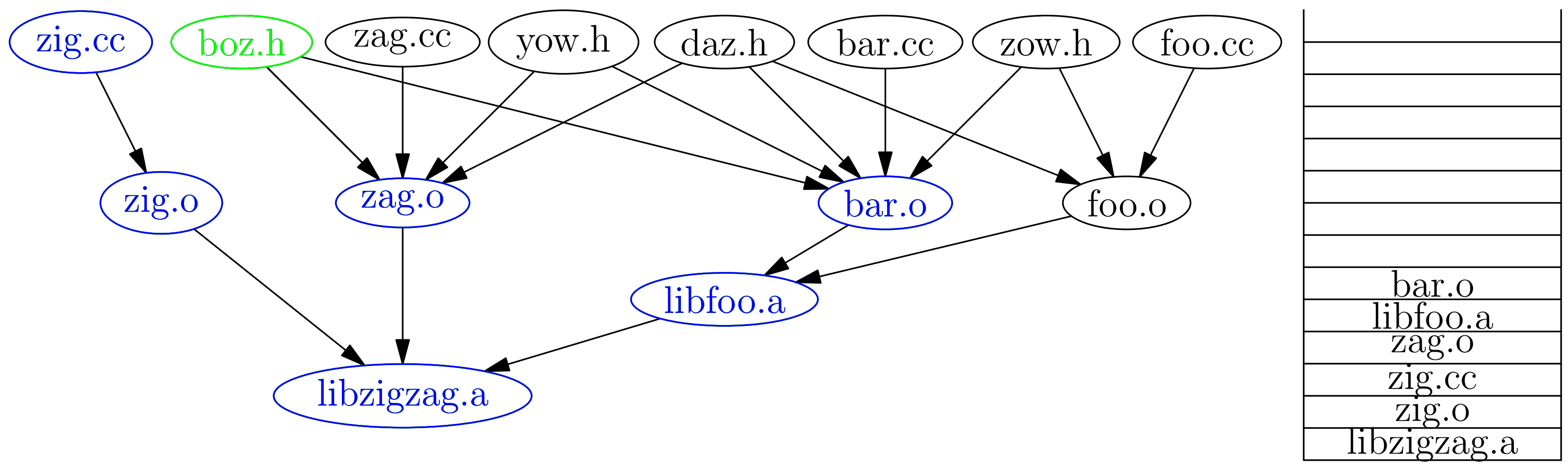
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



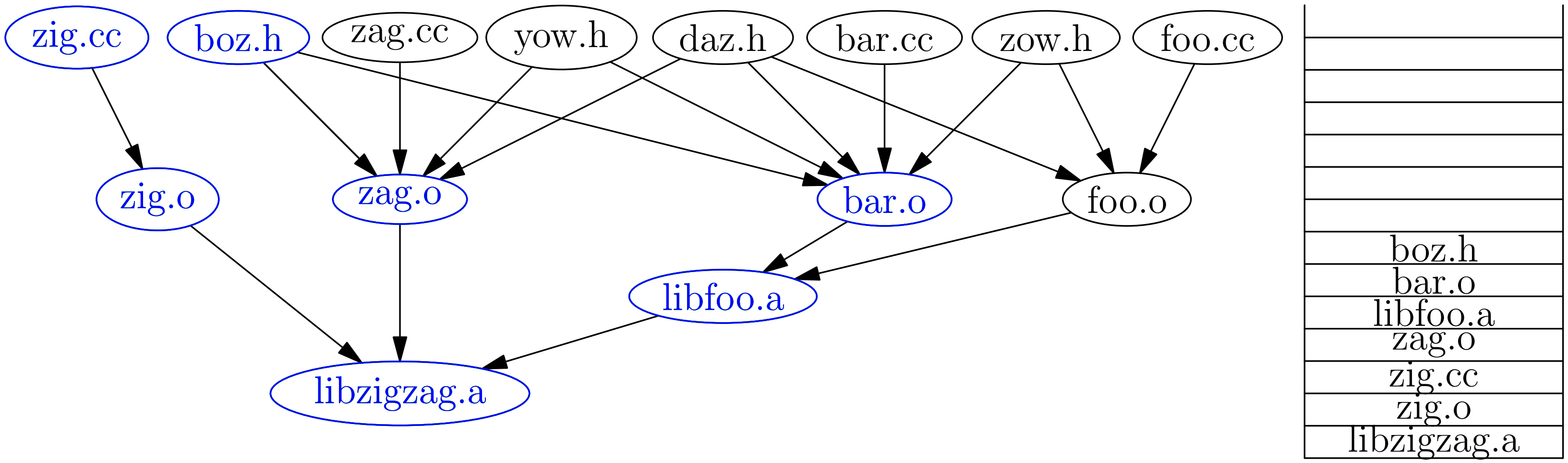
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



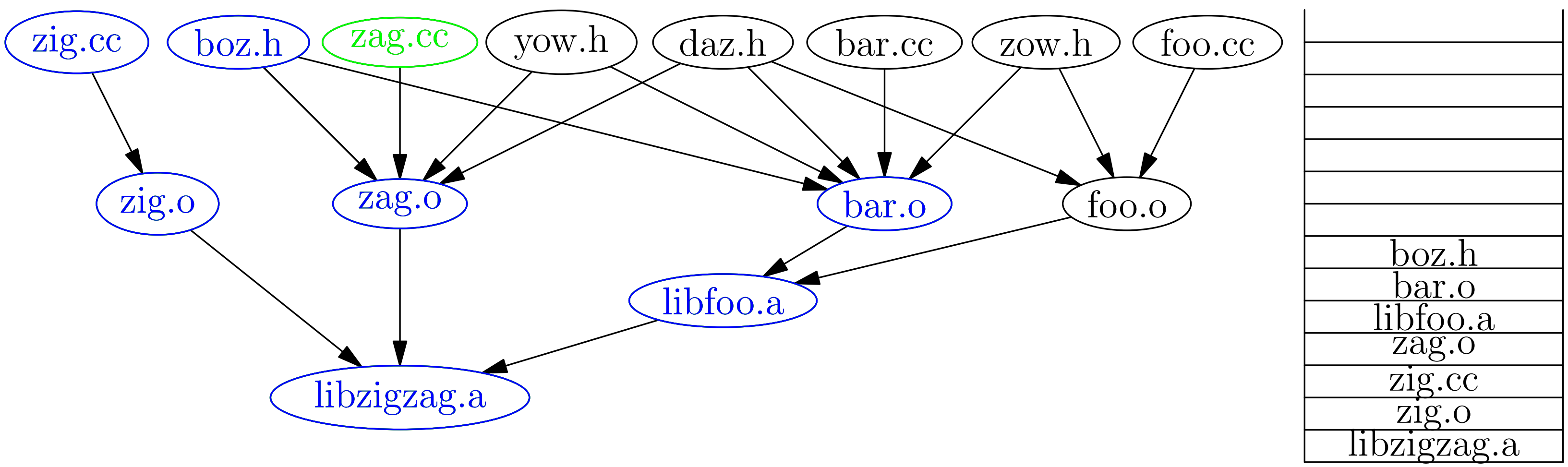
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



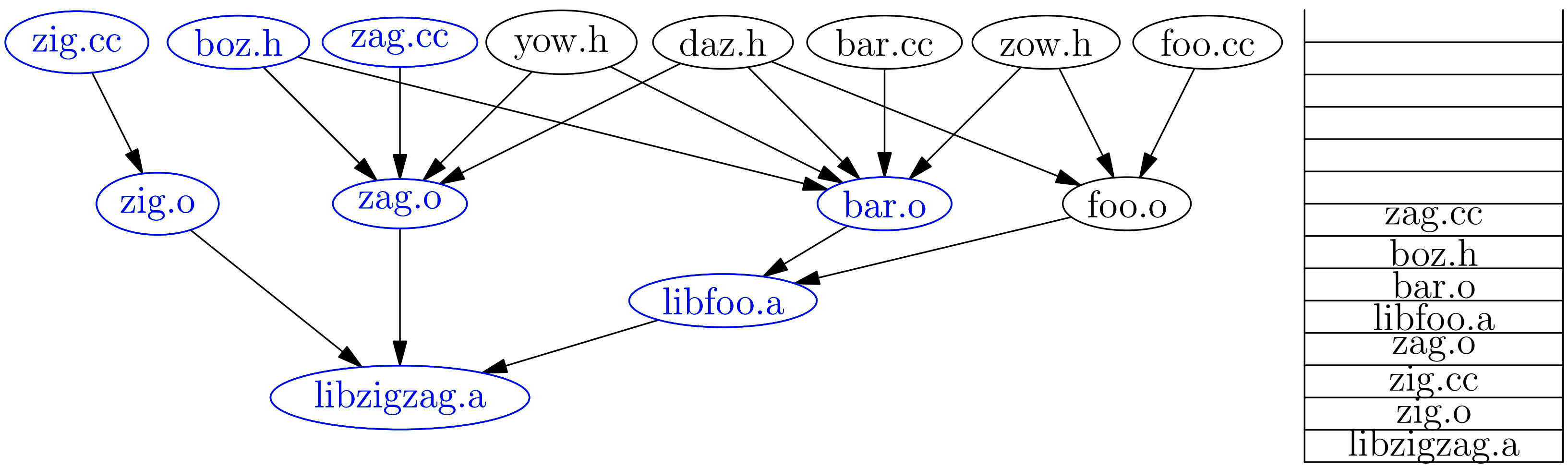
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



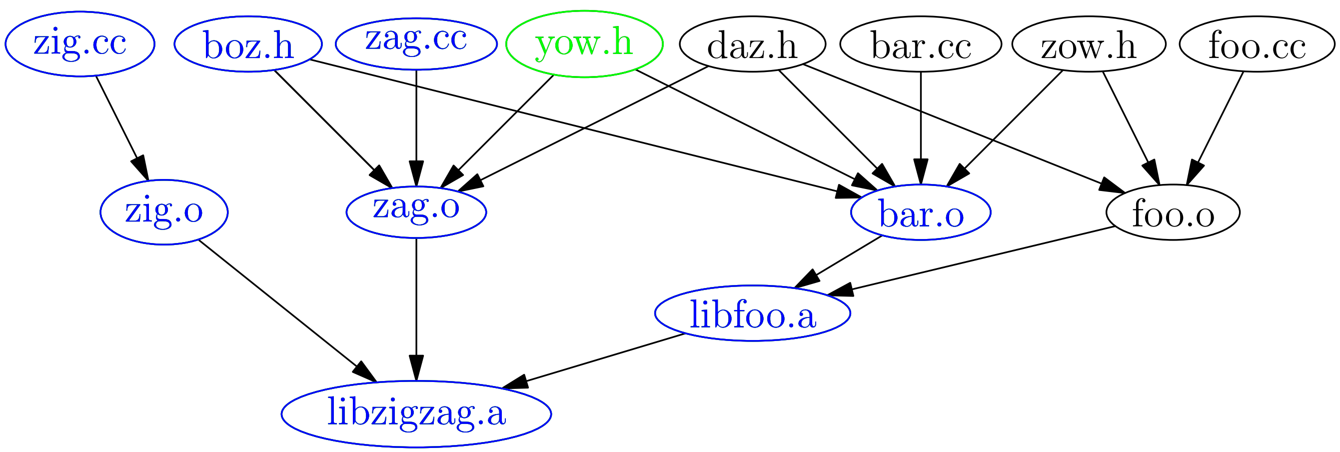
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



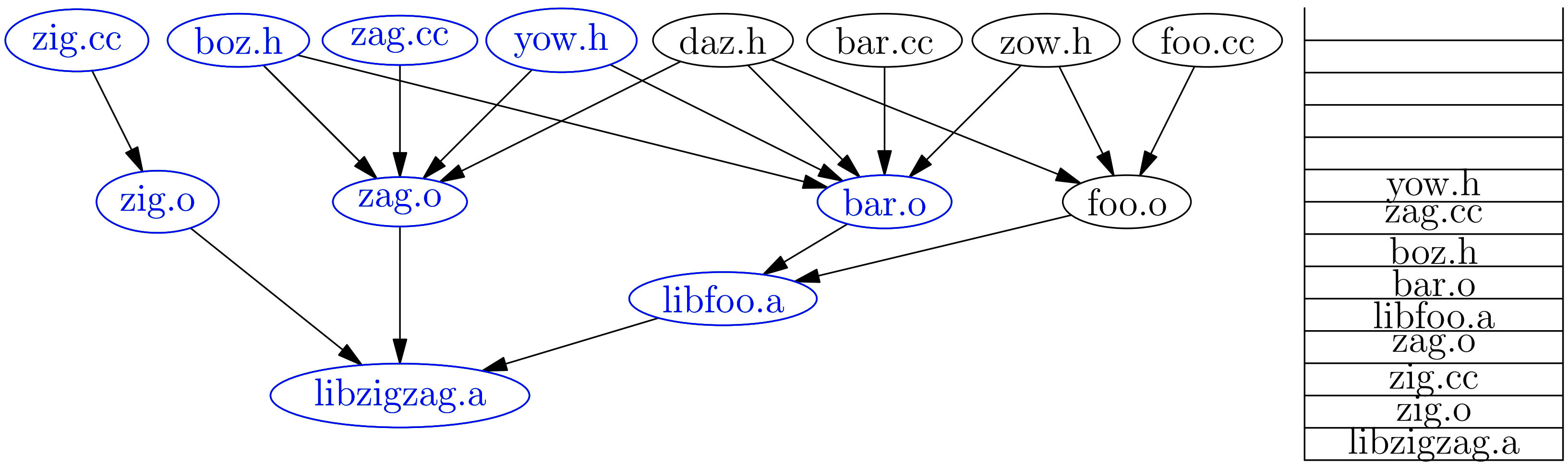
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



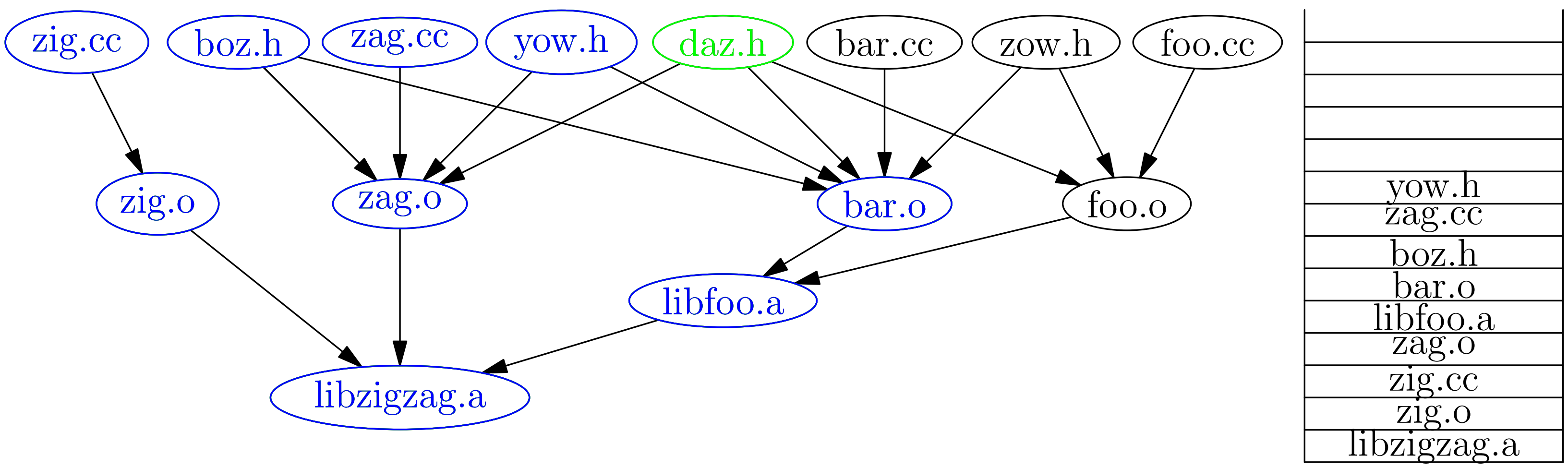
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



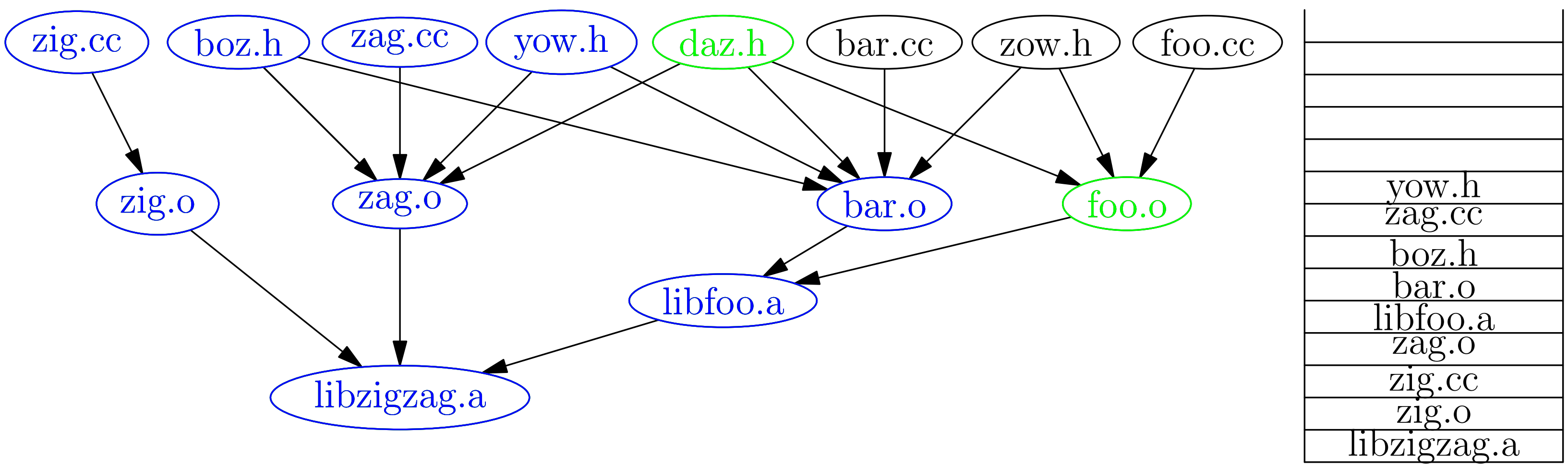
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



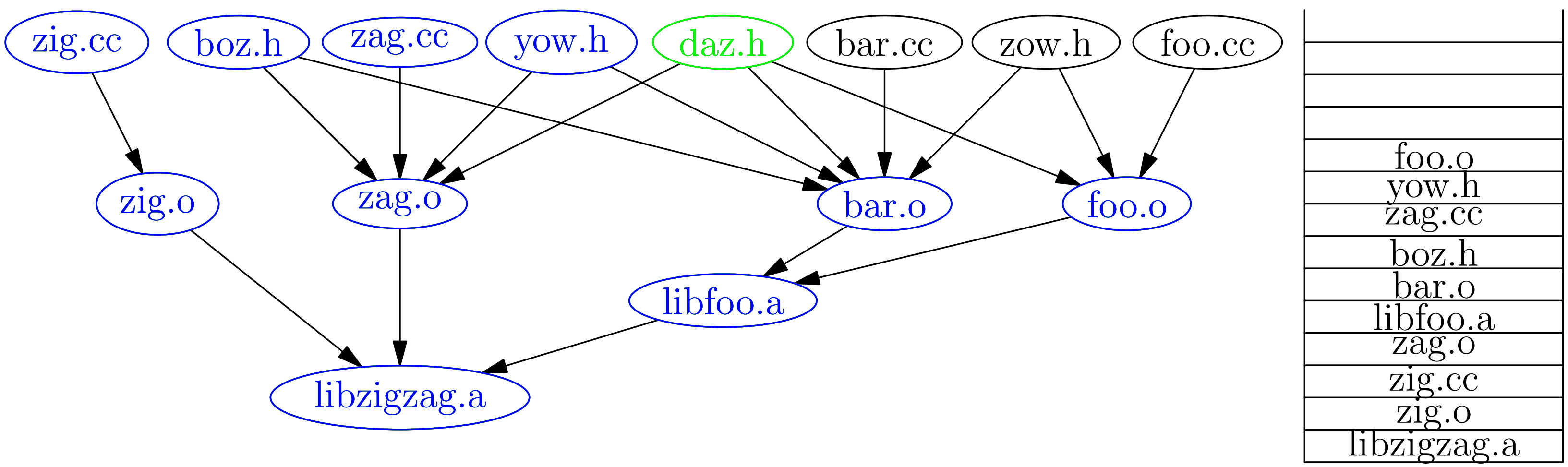
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



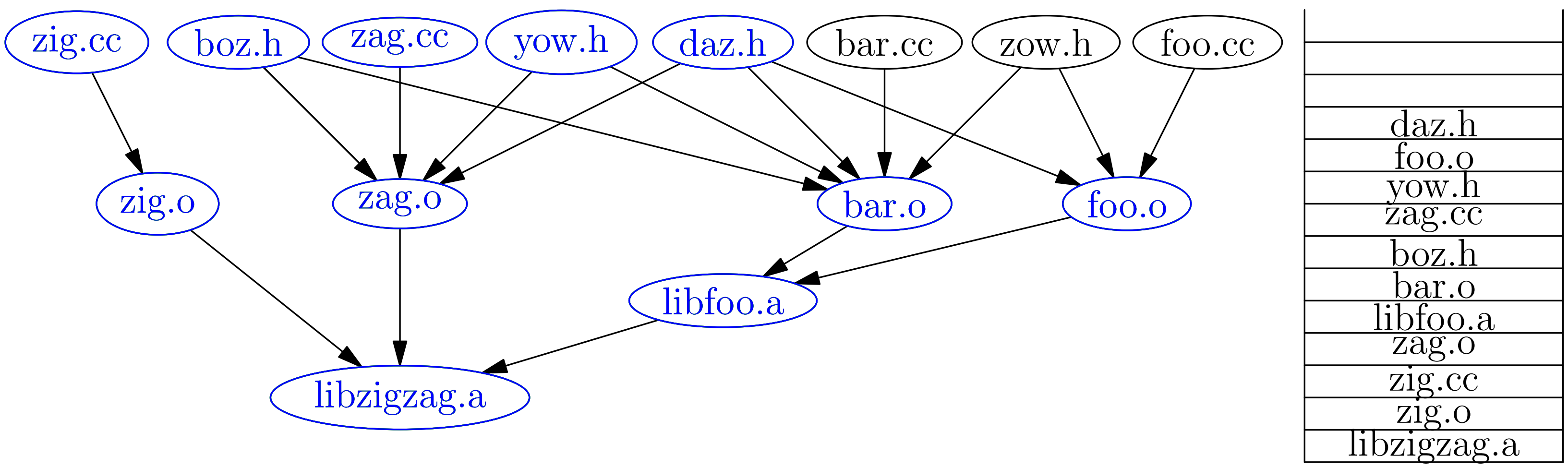
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



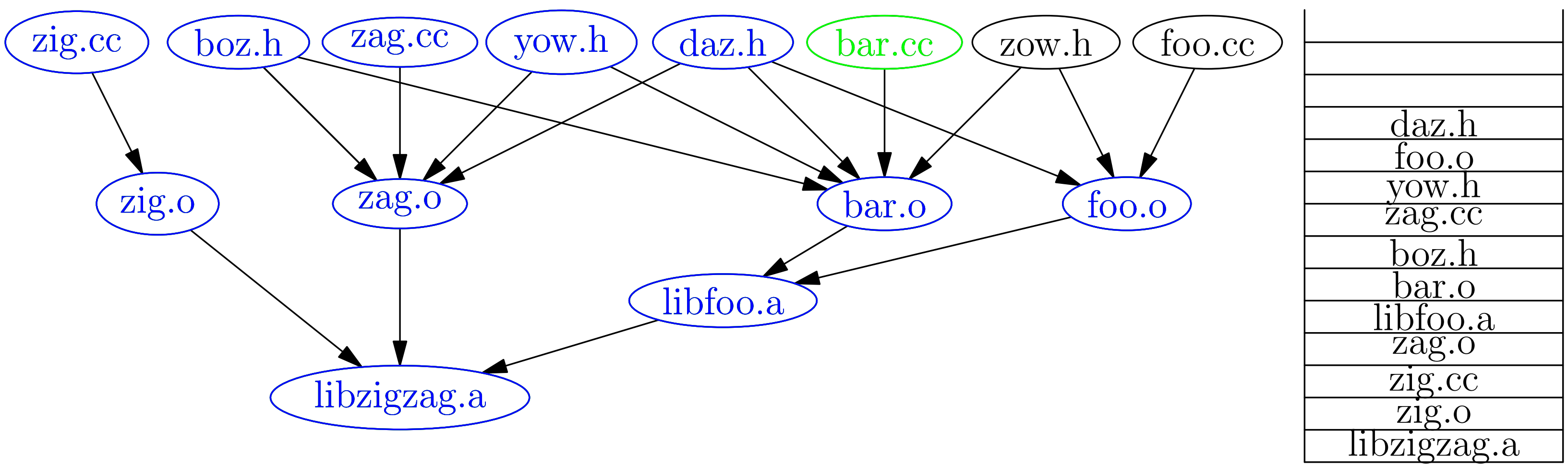
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



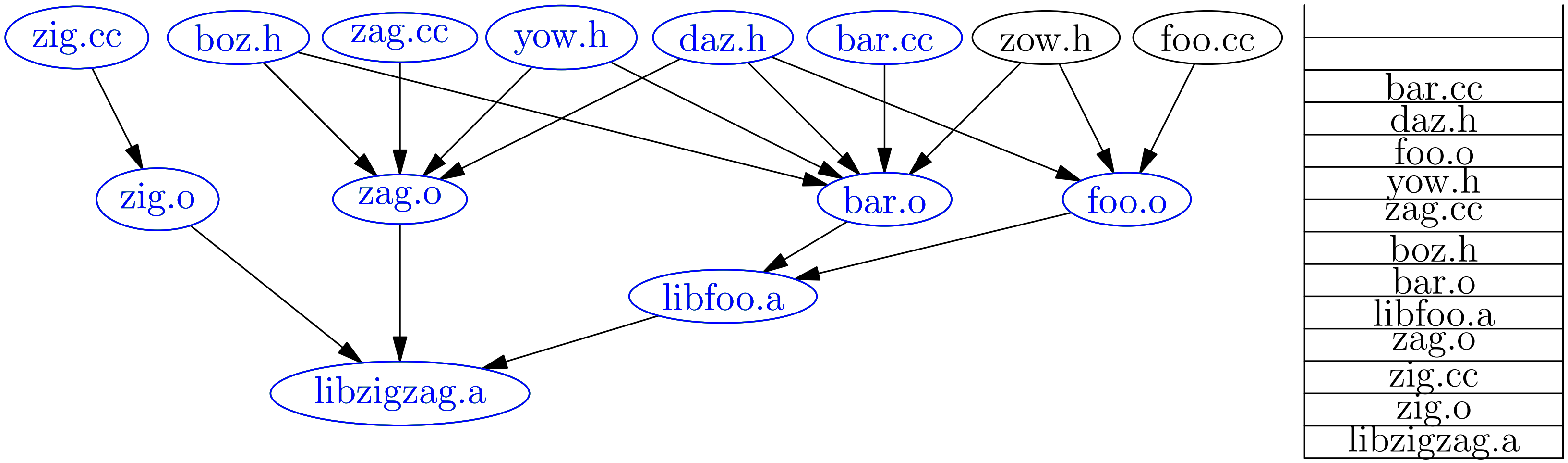
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



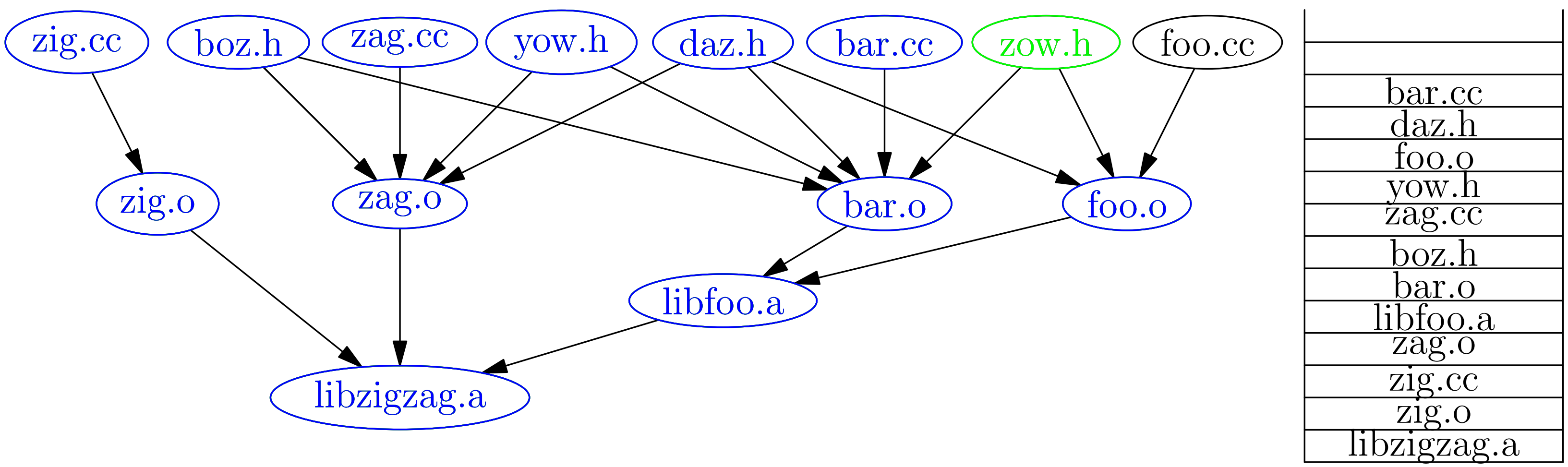
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



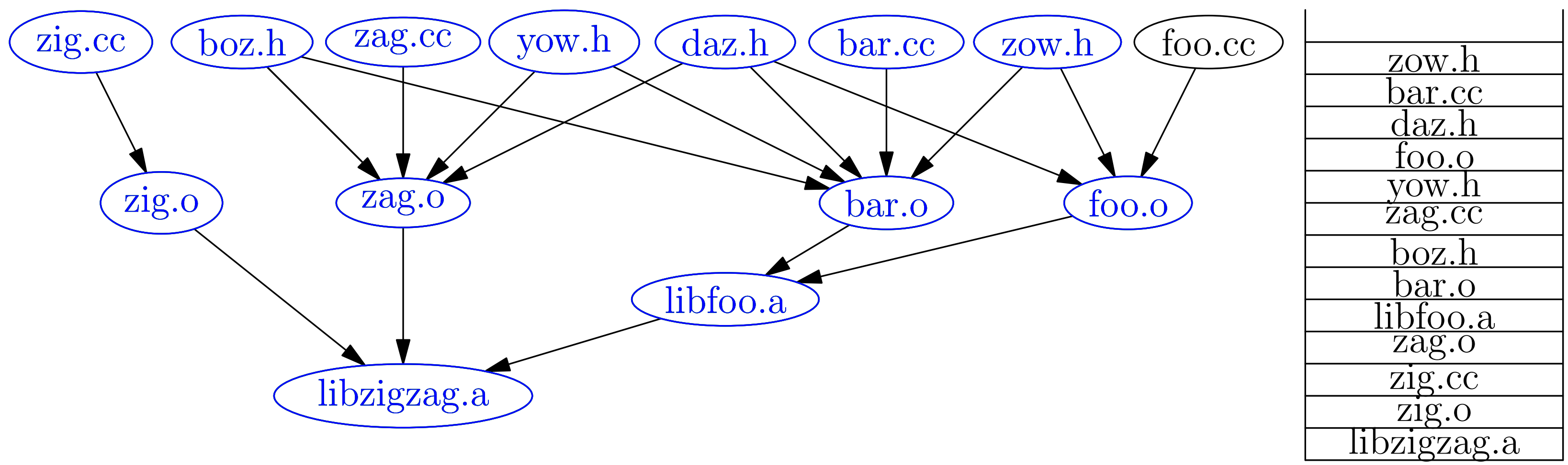
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



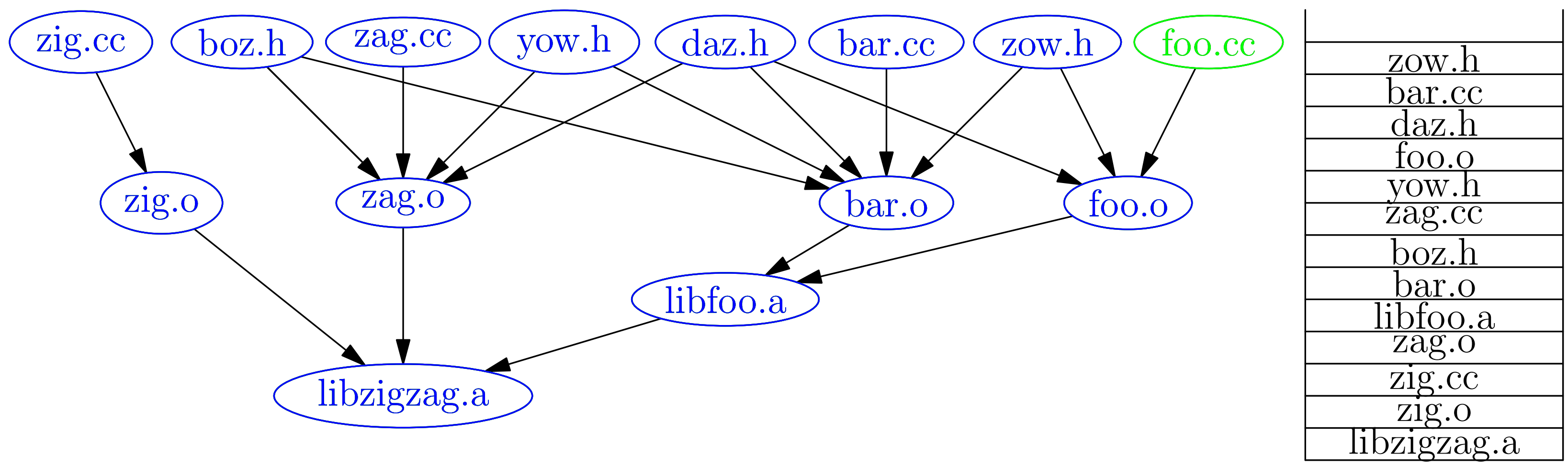
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



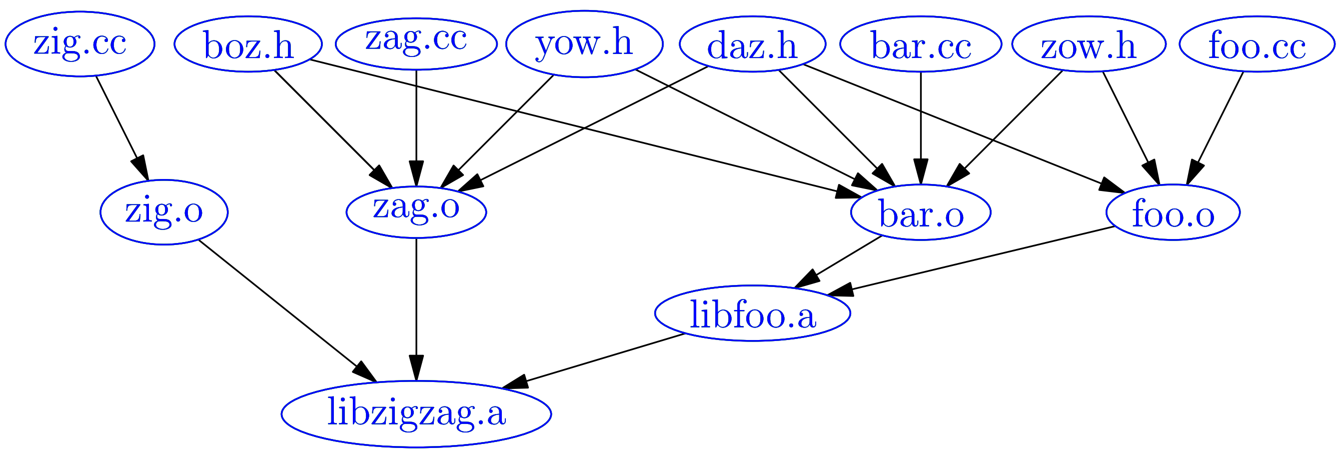
Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



foo.cc
zow.h
bar.cc
daz.h
foo.o
yow.h
zag.cc
boz.h
bar.o
libfoo.a
zag.o
zig.cc
zig.o
libzigzag.a

DFS on Digraphs

DFS for undirected graphs

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] ≠ "discovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

DFS on Digraphs

DFS for directed graphs

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] ≠ "discovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed" ∨ graph is directed) {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

Implementing Topological Sort

- Given our DFS programme we define

```
topologicalSort(graph) {
  Stack stack
  for node ∈ graph.vertexSet()
    if (¬discovered[node])
      dfs(graph, node)

  List topSortList
  while (¬stack.isEmpty())
    topSortList.add(stack.pop())

  return topSortList
}
```

Enhance DFS

- Requires us to define a couple of helper function

```
processVertexLast(node) {
  stack.push(node)
}

processEdge(currentNode, neighbour) {
  if (state[neighbour] == "processed") {
    print "error:␣graph␣not␣a␣DAG"
    finished = true
  }
}
```

Enhance DFS

- Requires us to define a couple of helper function

```
processVertexLast(node) {
  stack.push(node)
}

processEdge(currentNode, neighbour) {
  if (state[neighbour] == "processed") {
    print "error:␣graph␣not␣a␣DAG"
    finished = true
  }
}
```

Implementation Issues

- Most awkward part of the implementation is that the `topologicalSort` algorithm needs access to dfs structures (`discovered[]`)
- `processVertexLast(node)` needs access to the stack
- Need to be able to redefine `processVertexFirst`, `processEdge` and `processVertexLast`
- Different languages and libraries cope with this differently
  - ⋆ Java: JDSL, JGraphT
  - ⋆ C++: Boost Graph Library, LEDA

Implementation Issues

- Most awkward part of the implementation is that the `topologicalSort` algorithm needs access to dfs structures (`discovered[]`)

- `processVertexLast(node)` needs access to the stack

- Need to be able to redefine `processVertexFirst`, `processEdge` and `processVertexLast`

- Different languages and libraries cope with this differently
  - ⋆ Java: JDSL, JGraphT
  - ⋆ C++: Boost Graph Library, LEDA

Implementation Issues

- Most awkward part of the implementation is that the `topologicalSort` algorithm needs access to dfs structures (`discovered[]`)
- `processVertexLast(node)` needs access to the stack
- Need to be able to redefine `processVertexFirst`, `processEdge` and `processVertexLast`
- Different languages and libraries cope with this differently
  - ⋆ Java: JDSL, JGraphT
  - ⋆ C++: Boost Graph Library, LEDA

Implementation Issues

- Most awkward part of the implementation is that the `topologicalSort` algorithm needs access to dfs structures (`discovered[]`)
- `processVertexLast(node)` needs access to the stack
- Need to be able to redefine `processVertexFirst`, `processEdge` and `processVertexLast`
- Different languages and libraries cope with this differently
  - ⋆ Java: JDSL, JGraphT
  - ⋆ C++: Boost Graph Library, LEDA

Other Applications

- DFS is used for many other classic problems

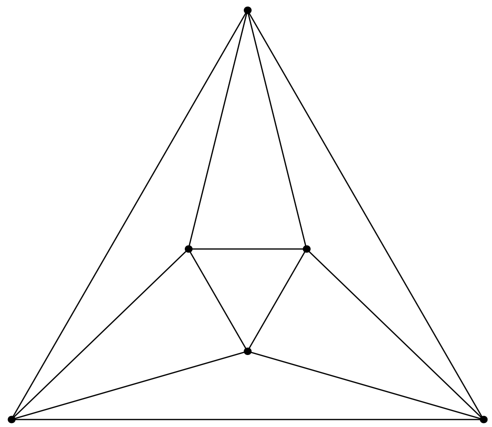
- Euler Cycles

- Strongly Connected Components

Other Applications

- DFS is used for many other classic problems

- Euler Cycles

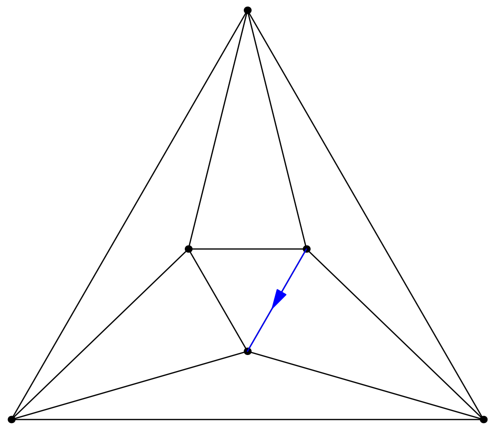


- Strongly Connected Components

Other Applications

- DFS is used for many other classic problems

- Euler Cycles

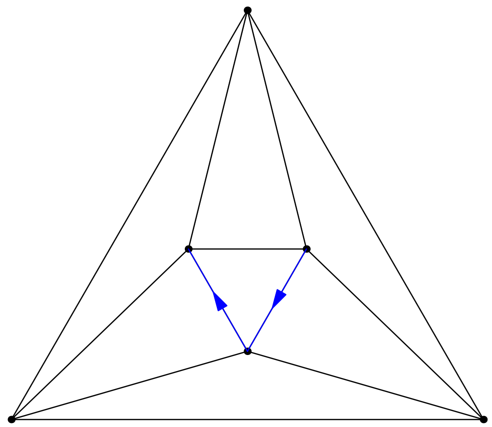


- Strongly Connected Components

Other Applications

- DFS is used for many other classic problems

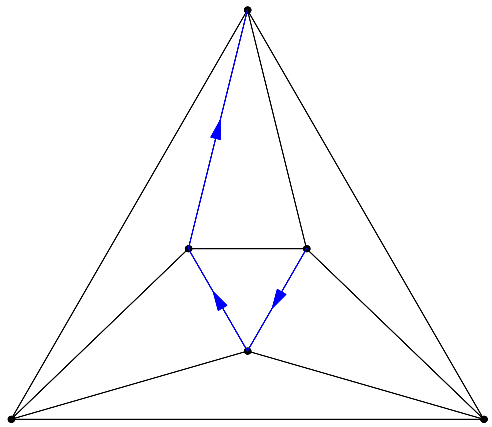
- Euler Cycles



- Strongly Connected Components

Other Applications

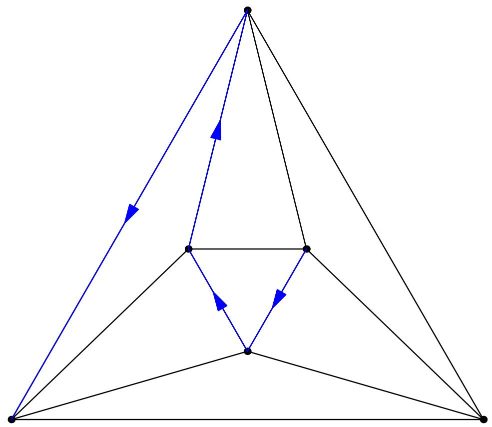
- DFS is used for many other classic problems
- Euler Cycles



- Strongly Connected Components

Other Applications

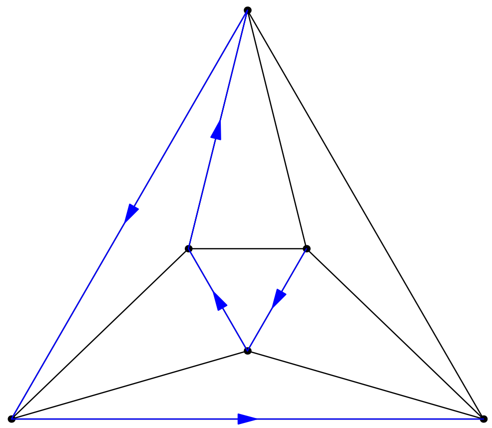
- DFS is used for many other classic problems
- Euler Cycles



- Strongly Connected Components

Other Applications

- DFS is used for many other classic problems
- Euler Cycles

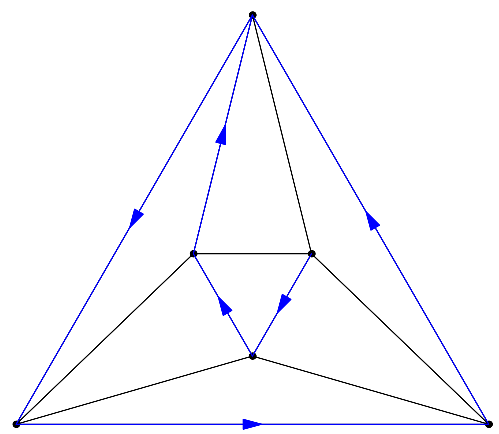


- Strongly Connected Components

Other Applications

- DFS is used for many other classic problems

- Euler Cycles

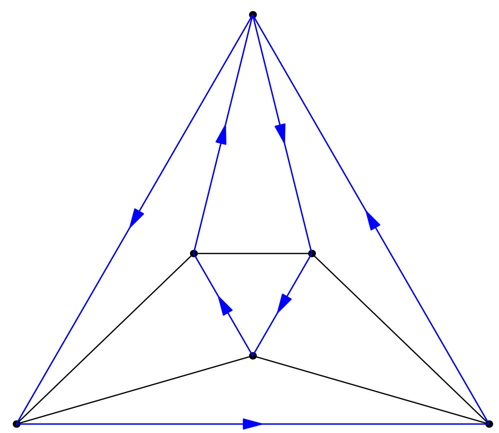


- Strongly Connected Components

Other Applications

- DFS is used for many other classic problems

- Euler Cycles

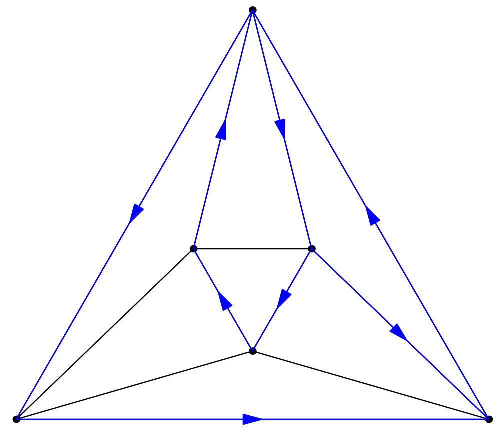


- Strongly Connected Components

Other Applications

- DFS is used for many other classic problems

- Euler Cycles

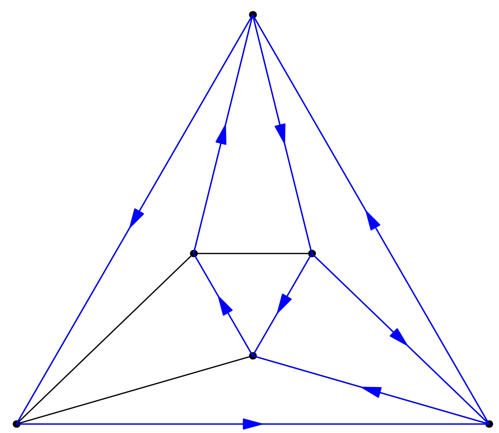


- Strongly Connected Components

Other Applications

- DFS is used for many other classic problems

- Euler Cycles

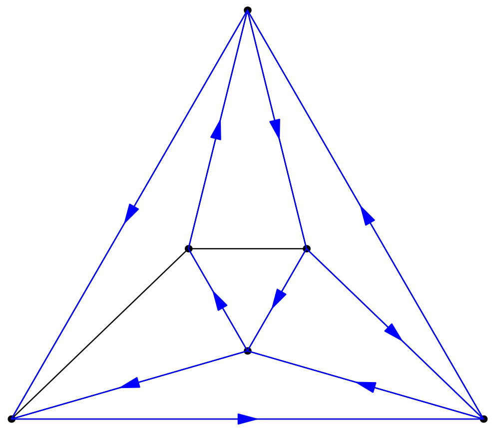


- Strongly Connected Components

Other Applications

- DFS is used for many other classic problems

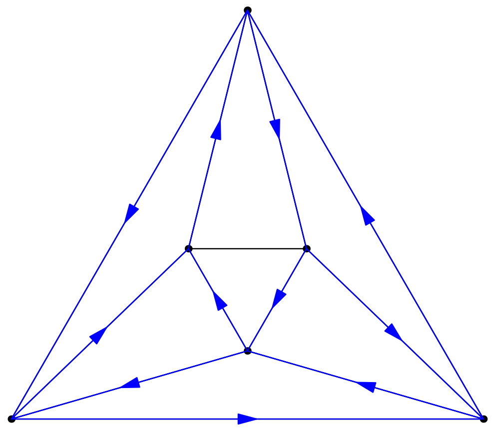
- Euler Cycles



- Strongly Connected Components

Other Applications

- DFS is used for many other classic problems
- Euler Cycles

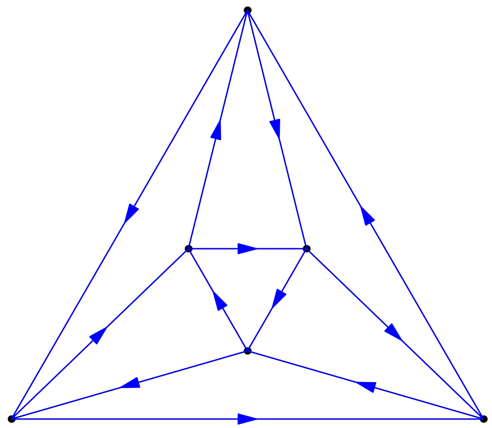


- Strongly Connected Components

Other Applications

- DFS is used for many other classic problems

- Euler Cycles

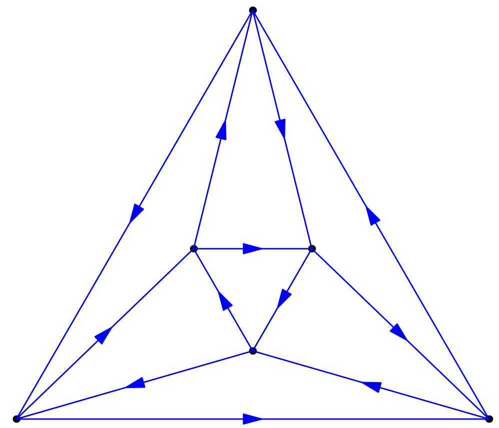


- Strongly Connected Components

Other Applications

- DFS is used for many other classic problems

- Euler Cycles

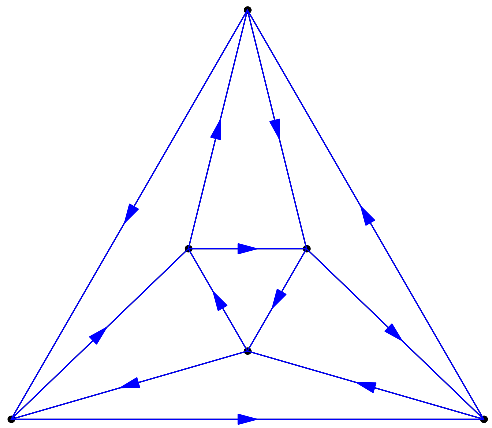


- Strongly Connected Components

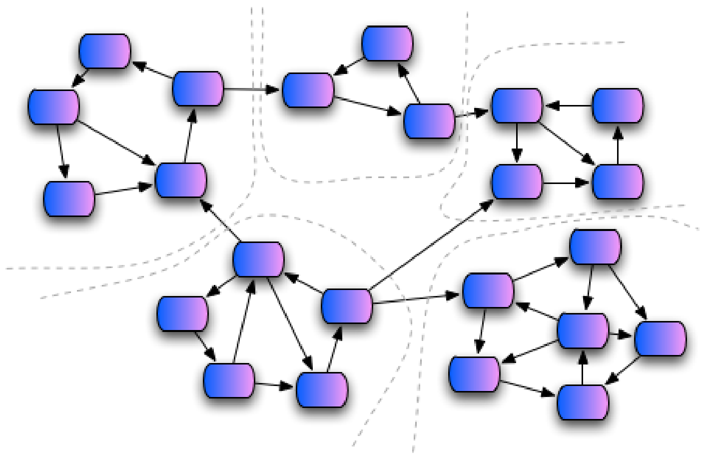
Other Applications

- DFS is used for many other classic problems

- Euler Cycles



- Strongly Connected Components



Lessons

- Breadth first and depth first search are different methods for traversing graphs
- They are used as part of many specific algorithms for discovering graph properties
- Breadth first search is particularly important for finding shortest paths in unweighted graphs
- Depth first search is used in a whole host of applications (finding articulation points, Euler cycles, strongly connected components)
- One of the most used application is in topological sort (finding an ordering of processes represented by a DAG)

Lessons

- Breadth first and depth first search are different methods for traversing graphs
- They are used as part of many specific algorithms for discovering graph properties
- Breadth first search is particularly important for finding shortest paths in unweighted graphs
- Depth first search is used in a whole host of applications (finding articulation points, Euler cycles, strongly connected components)
- One of the most used application is in topological sort (finding an ordering of processes represented by a DAG)

Lessons

- Breadth first and depth first search are different methods for traversing graphs
- They are used as part of many specific algorithms for discovering graph properties
- Breadth first search is particularly important for finding shortest paths in unweighted graphs
- Depth first search is used in a whole host of applications (finding articulation points, Euler cycles, strongly connected components)
- One of the most used application is in topological sort (finding an ordering of processes represented by a DAG)

Lessons

- Breadth first and depth first search are different methods for traversing graphs
- They are used as part of many specific algorithms for discovering graph properties
- Breadth first search is particularly important for finding shortest paths in unweighted graphs
- Depth first search is used in a whole host of applications (finding articulation points, Euler cycles, strongly connected components)
- One of the most used application is in topological sort (finding an ordering of processes represented by a DAG)

Lessons

- Breadth first and depth first search are different methods for traversing graphs
- They are used as part of many specific algorithms for discovering graph properties
- Breadth first search is particularly important for finding shortest paths in unweighted graphs
- Depth first search is used in a whole host of applications (finding articulation points, Euler cycles, strongly connected components)
- One of the most used application is in topological sort (finding an ordering of processes represented by a DAG)