Kattis

Graphs part 1

Atli Fannar Franklín

October 21, 2024

School of Computer Science

Reykjavík University

Today's material

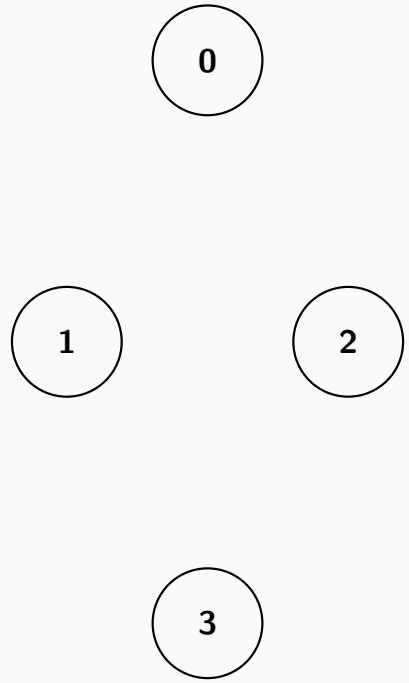
- Graphs
- Storing and representing graphs
- Depth first search (DFS)
- Breadth first search (BFS)
- Tarjan's algorithm
- Topological sorting
- Diameters in graphs

Basics

What is a graph?

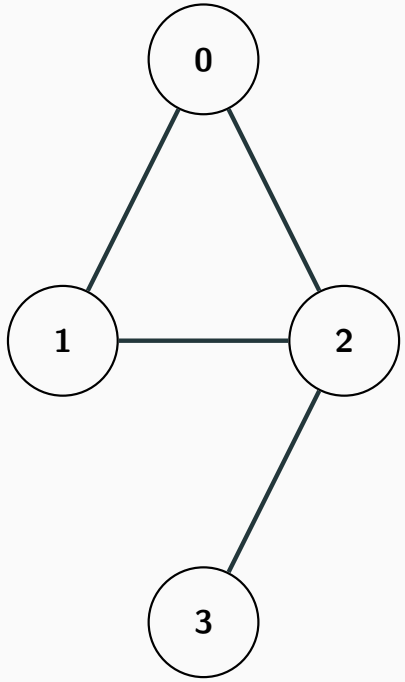
What is a graph?

- Vertices
  - Road intersections
  - Computers
  - Floors in a house
  - Objects



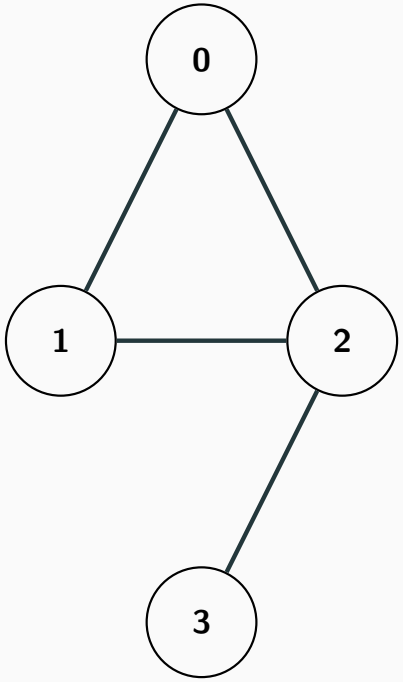
What is a graph?

- Vertices
  - Road intersections
  - Computers
  - Floors in a house
  - Objects
- Edges
  - Roads
  - Ethernet cables
  - Stairs or elevators
  - Relation between objects



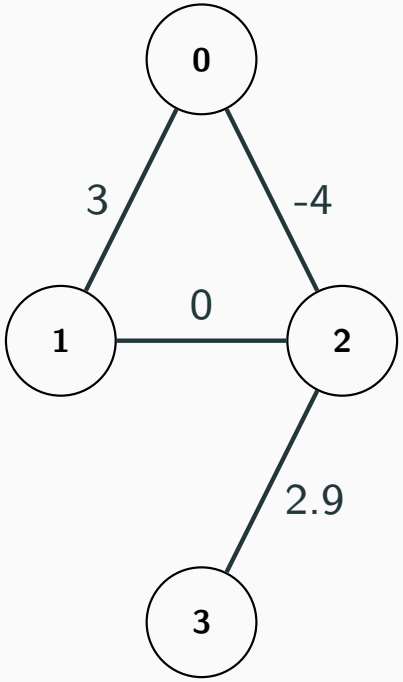
Types of edges

- Unweighted



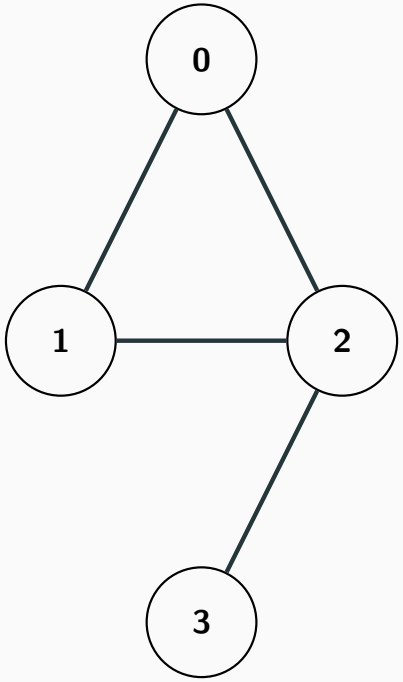
Types of edges

- Unweighted or Weighted



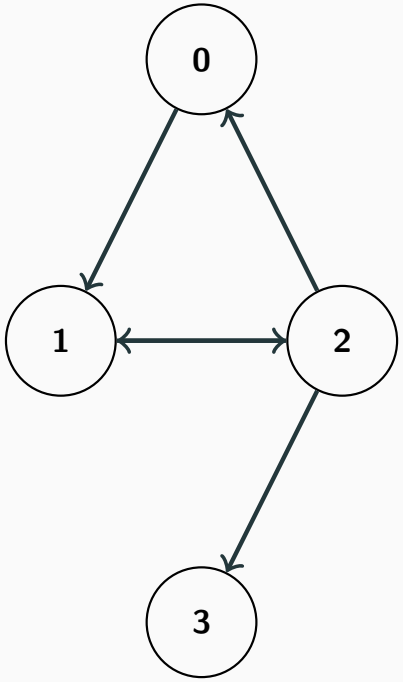
Types of edges

- Unweighted or Weighted
- Undirected



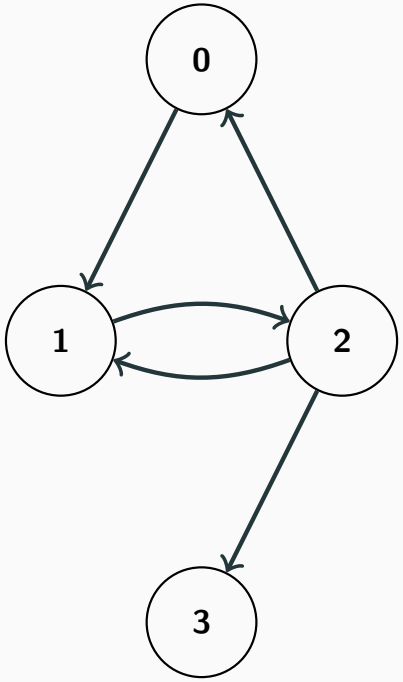
Types of edges

- Unweighted or Weighted
- Undirected or Directed

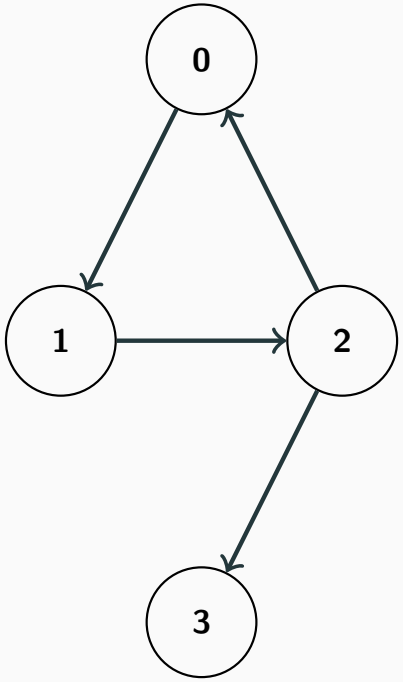


Types of edges

- Unweighted or Weighted
- Undirected or Directed

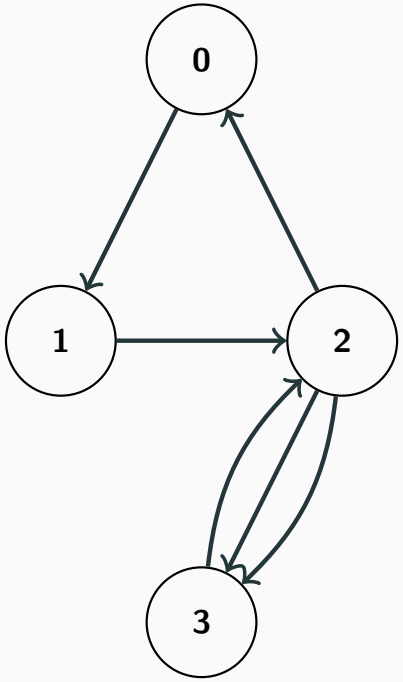


Multigraphs



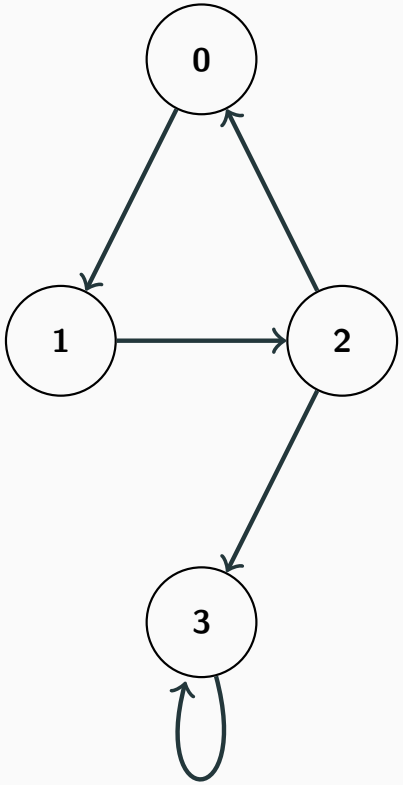
Multigraphs

- Multiple edges



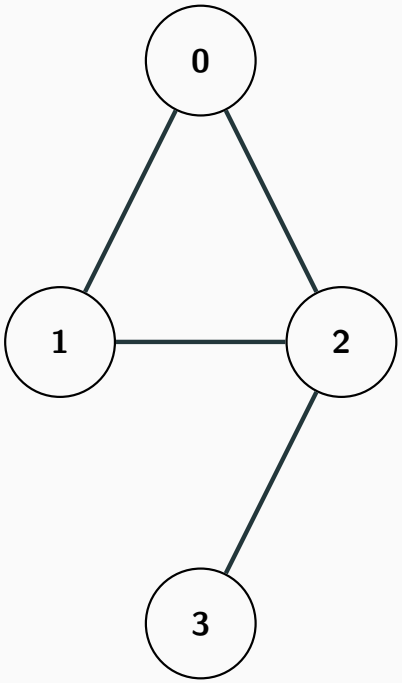
Multigraphs

- Multiple edges
- Self-loops



Adjacency list

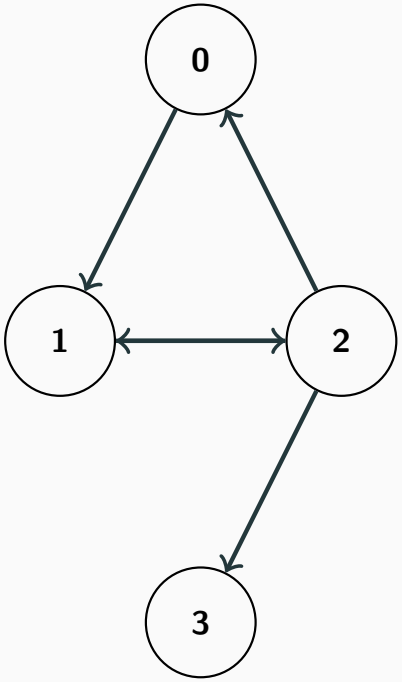
```
0: 1, 2
1: 0, 2
2: 0, 1, 3
3: 2
```




```
vector<int> adj[4];
adj[0].push_back(1);
adj[0].push_back(2);
adj[1].push_back(0);
adj[1].push_back(2);
adj[2].push_back(0);
adj[2].push_back(1);
adj[2].push_back(3);
adj[3].push_back(2);
```

Adjacency list (directed)

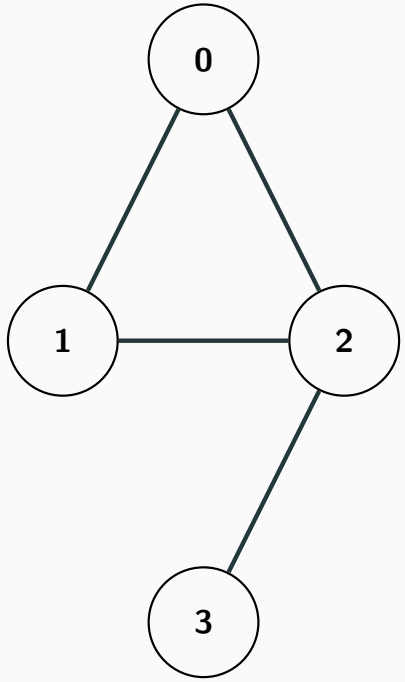
```
0: 1
1: 2
2: 0, 1, 3
3:
```




```
vector<int> adj[4];
adj[0].push_back(1);
adj[1].push_back(2);
adj[2].push_back(0);
adj[2].push_back(1);
adj[2].push_back(3);
```

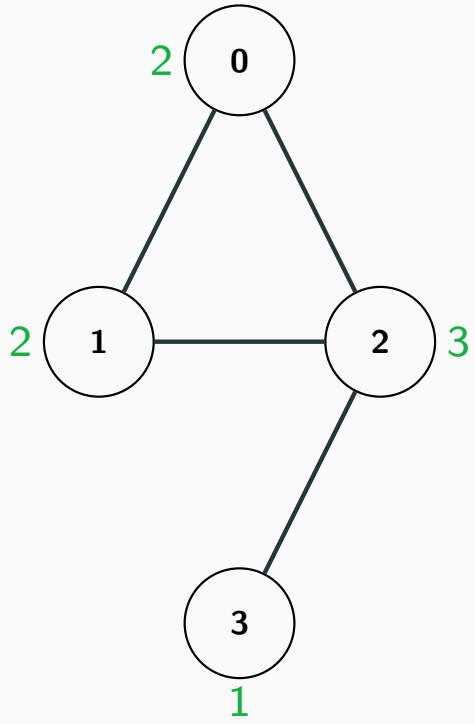
Vertex properties (undirected graph)

- Degree of a vertex
  - Number of adjacent edges
  - Number of adjacent vertices



Vertex properties (undirected graph)

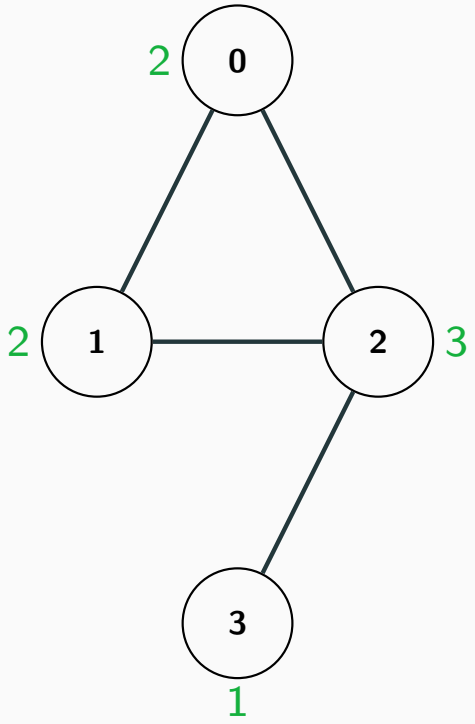
- Degree of a vertex
  - Number of adjacent edges
  - Number of adjacent vertices



Vertex properties (undirected graph)

- Degree of a vertex
  - Number of adjacent edges
  - Number of adjacent vertices
- Handshaking lemma

$$\sum_{v \in V} \deg(v) = 2|E|$$

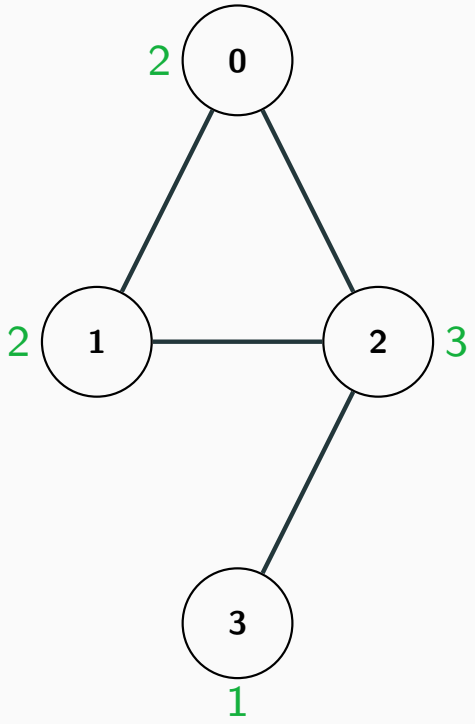


Vertex properties (undirected graph)

- Degree of a vertex
  - Number of adjacent edges
  - Number of adjacent vertices
- Handshaking lemma

$$\sum_{v \in V} \deg(v) = 2|E|$$

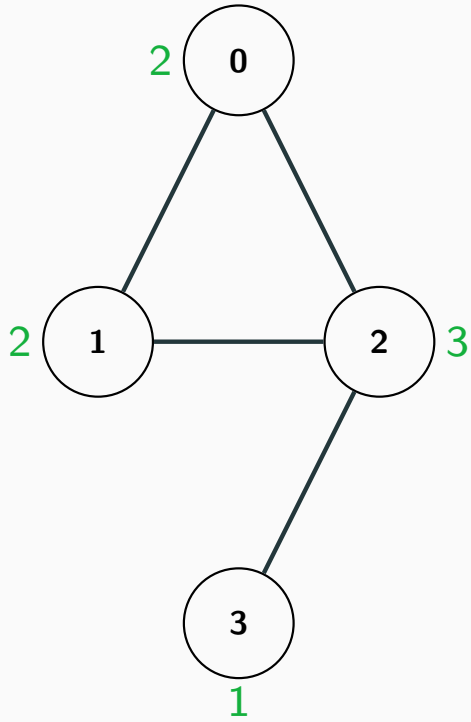
$$2 + 2 + 3 + 1 = 2 \times 4$$



Vertex properties (undirected graph)

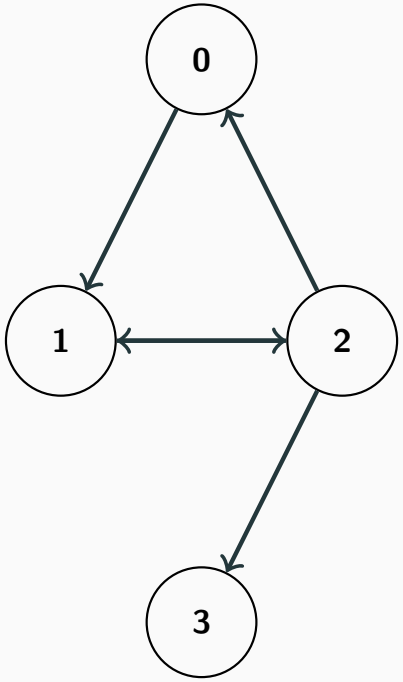
```
0: 1, 2
1: 0, 2
2: 0, 1, 3
3: 2
```

```
adj[0].size() // 2
adj[1].size() // 2
adj[2].size() // 3
adj[3].size() // 1
```



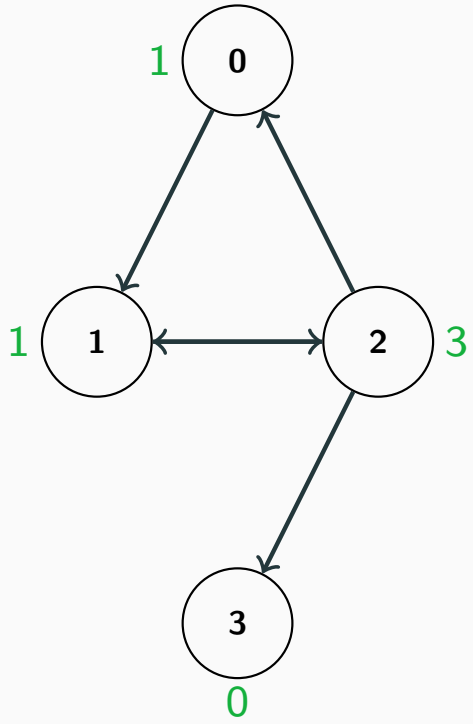
Vertex properties (directed graph)

- Outdegree of a vertex
  - Number of outgoing edges



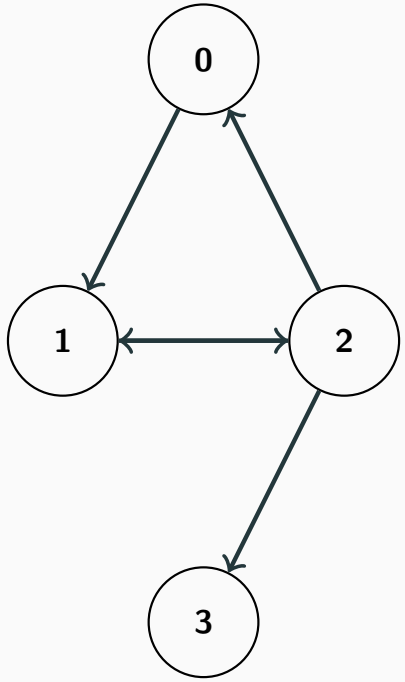
Vertex properties (directed graph)

- Outdegree of a vertex
  - Number of outgoing edges



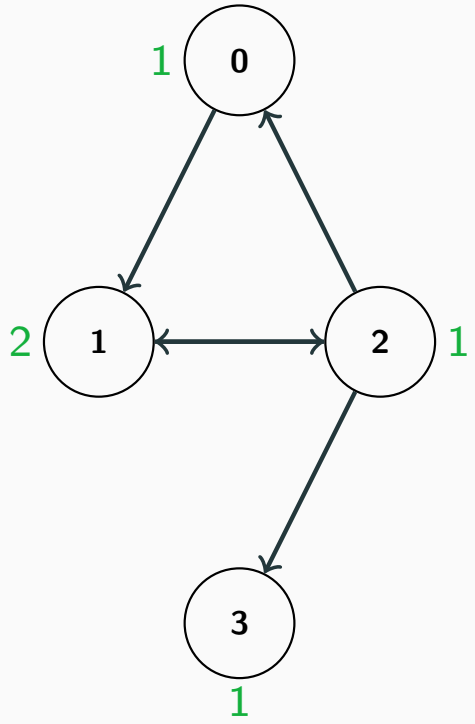
Vertex properties (directed graph)

- Outdegree of a vertex
  - Number of outgoing edges
- Indegree of a vertex
  - Number of incoming edges



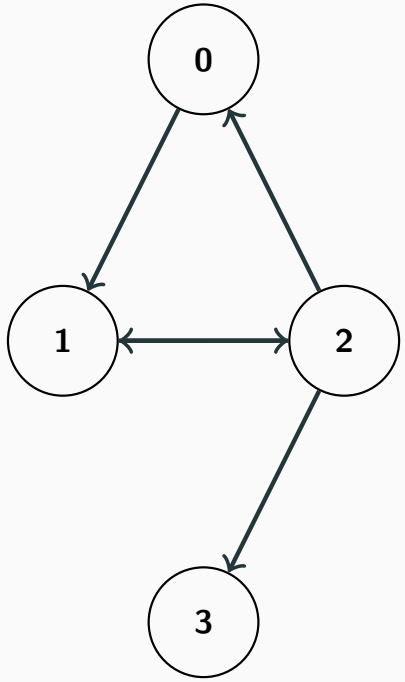
Vertex properties (directed graph)

- Outdegree of a vertex
  - Number of outgoing edges
- Indegree of a vertex
  - Number of incoming edges



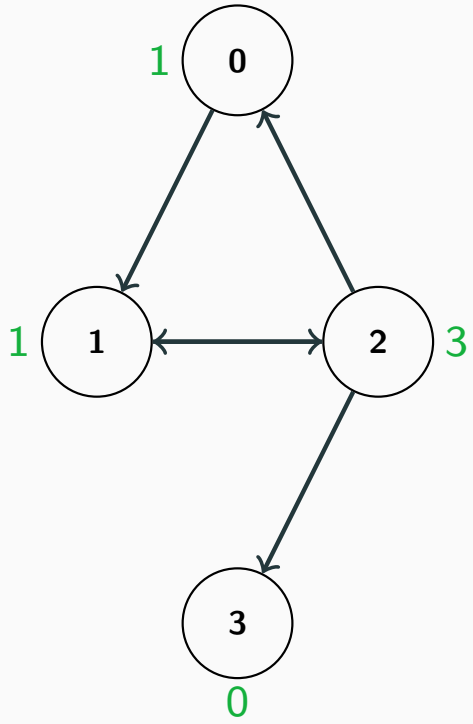
Vertex properties (directed graph)

- Outdegree of a vertex
  - Number of outgoing edges
- Indegree of a vertex
  - Number of incoming edges



Vertex properties (directed graph)

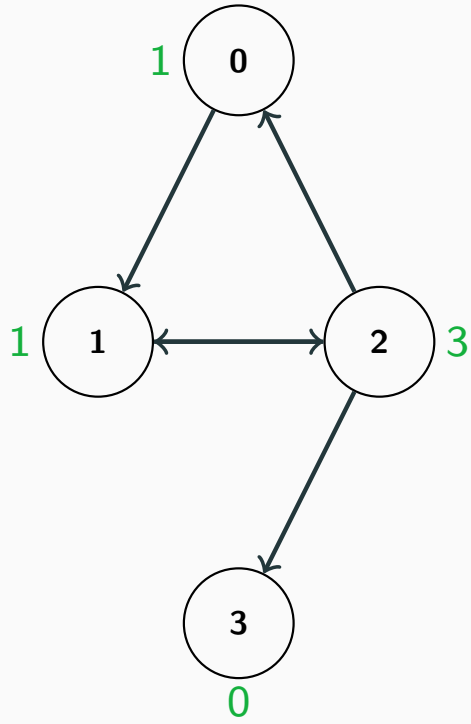
- Outdegree of a vertex
  - Number of outgoing edges
- Indegree of a vertex
  - Number of incoming edges



Adjacency list (directed)

```
0: 1
1: 2
2: 0, 1, 3
3:
```

```
adj[0].size() // 1
adj[1].size() // 1
adj[2].size() // 3
adj[3].size() // 0
```



Paths

- Path / Walk / Trail:

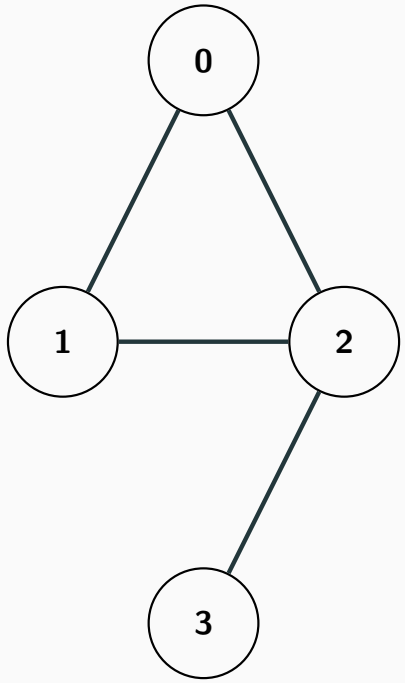
$$e_1 e_2 \dots e_k$$

such that

$$e_i \in E$$

$$e_i = e_j \Rightarrow i = j$$

$$\text{to}(e_i) = \text{from}(e_{i+1})$$



Paths

- Path / Walk / Trail:

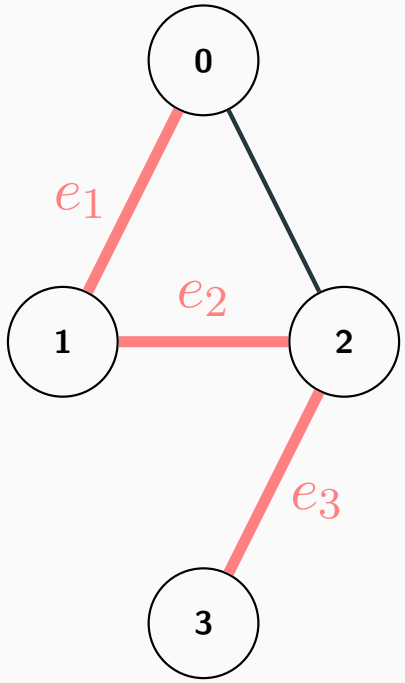
$$e_1 e_2 \dots e_k$$

such that

$$e_i \in E$$

$$e_i = e_j \Rightarrow i = j$$

$$\text{to}(e_i) = \text{from}(e_{i+1})$$



Paths

- Path / Walk / Trail:

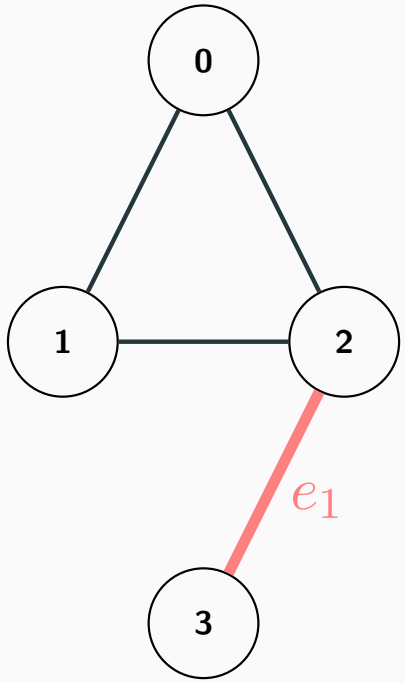
$$e_1 e_2 \dots e_k$$

such that

$$e_i \in E$$

$$e_i = e_j \Rightarrow i = j$$

$$\text{to}(e_i) = \text{from}(e_{i+1})$$



Paths

- Path / Walk / Trail:

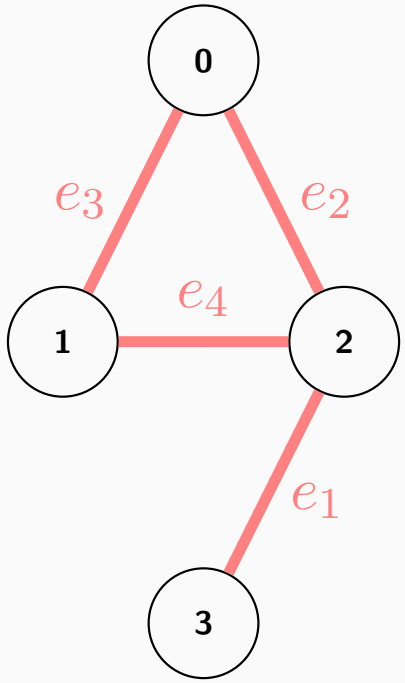
$$e_1 e_2 \dots e_k$$

such that

$$e_i \in E$$

$$e_i = e_j \Rightarrow i = j$$

$$\text{to}(e_i) = \text{from}(e_{i+1})$$



Cycles

- Cycle / Circuit / Tour:

$$e_1 e_2 \dots e_k$$

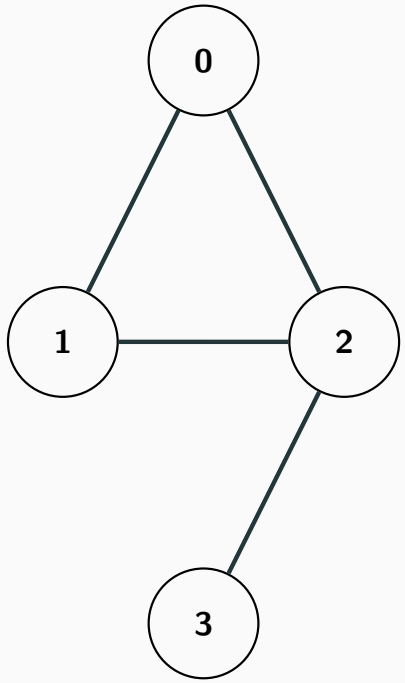
such that

$$e_i \in E$$

$$e_i = e_j \Rightarrow i = j$$

$$\text{to}(e_i) = \text{from}(e_{i+1})$$

$$\text{from}(e_1) = \text{to}(e_k)$$



Cycles

- Cycle / Circuit / Tour:

$$e_1 e_2 \dots e_k$$

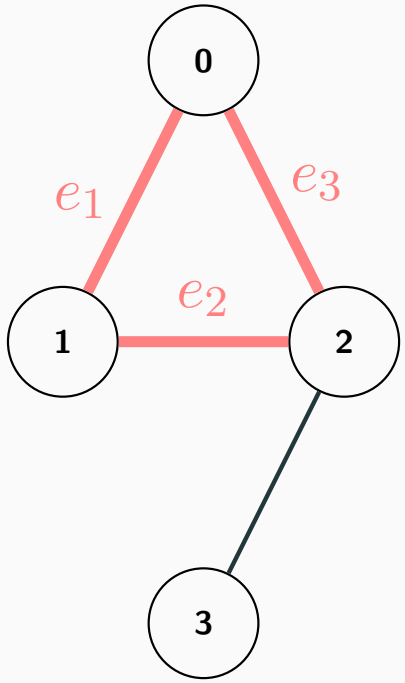
such that

$$e_i \in E$$

$$e_i = e_j \Rightarrow i = j$$

$$\text{to}(e_i) = \text{from}(e_{i+1})$$

$$\text{from}(e_1) = \text{to}(e_k)$$



Cycles

- Cycle / Circuit / Tour:

$$e_1 e_2 \dots e_k$$

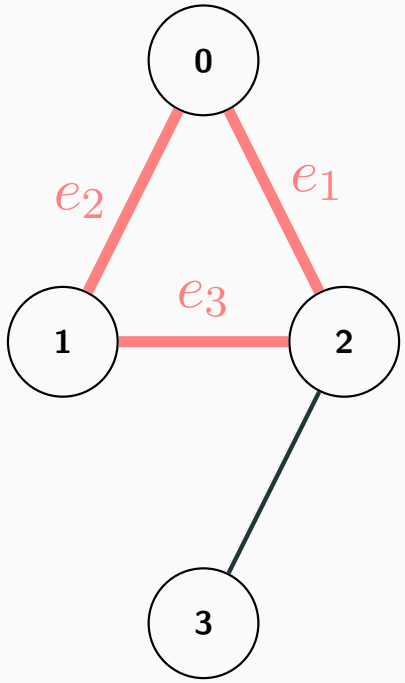
such that

$$e_i \in E$$

$$e_i = e_j \Rightarrow i = j$$

$$\text{to}(e_i) = \text{from}(e_{i+1})$$

$$\text{from}(e_1) = \text{to}(e_k)$$



Cycles

- Cycle / Circuit / Tour:

$$e_1 e_2 \dots e_k$$

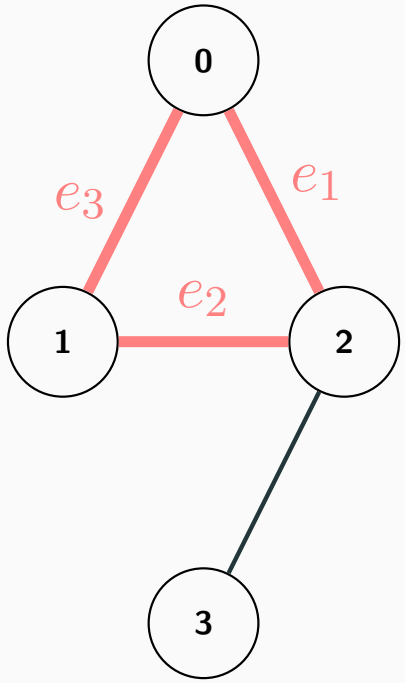
such that

$$e_i \in E$$

$$e_i = e_j \Rightarrow i = j$$

$$\text{to}(e_i) = \text{from}(e_{i+1})$$

$$\text{from}(e_1) = \text{to}(e_k)$$

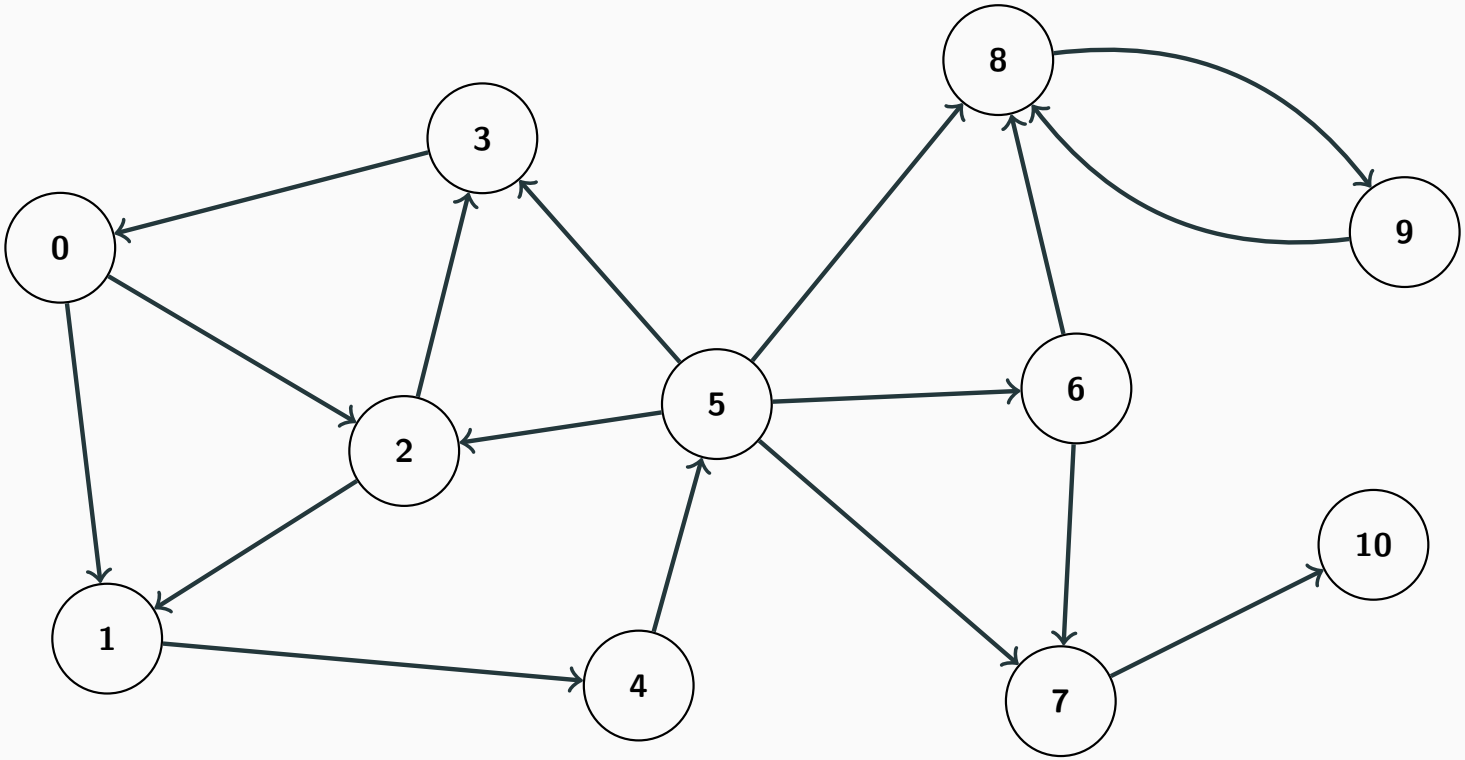


Depth-first search

Depth-first search

- Given a graph (either directed or undirected) and two vertices u and v, does there exist a path from u to v?
- Depth-first search is an algorithm for finding such a path, if one exists
- It traverses the graph in depth-first order, starting from the initial vertex u
- We don't actually have to specify a v, since we can just let it visit all reachable vertices from u (and still same time complexity)
- But what is the time complexity?
- Each vertex is visited once, and each edge is traversed once
- $O(n + m)$

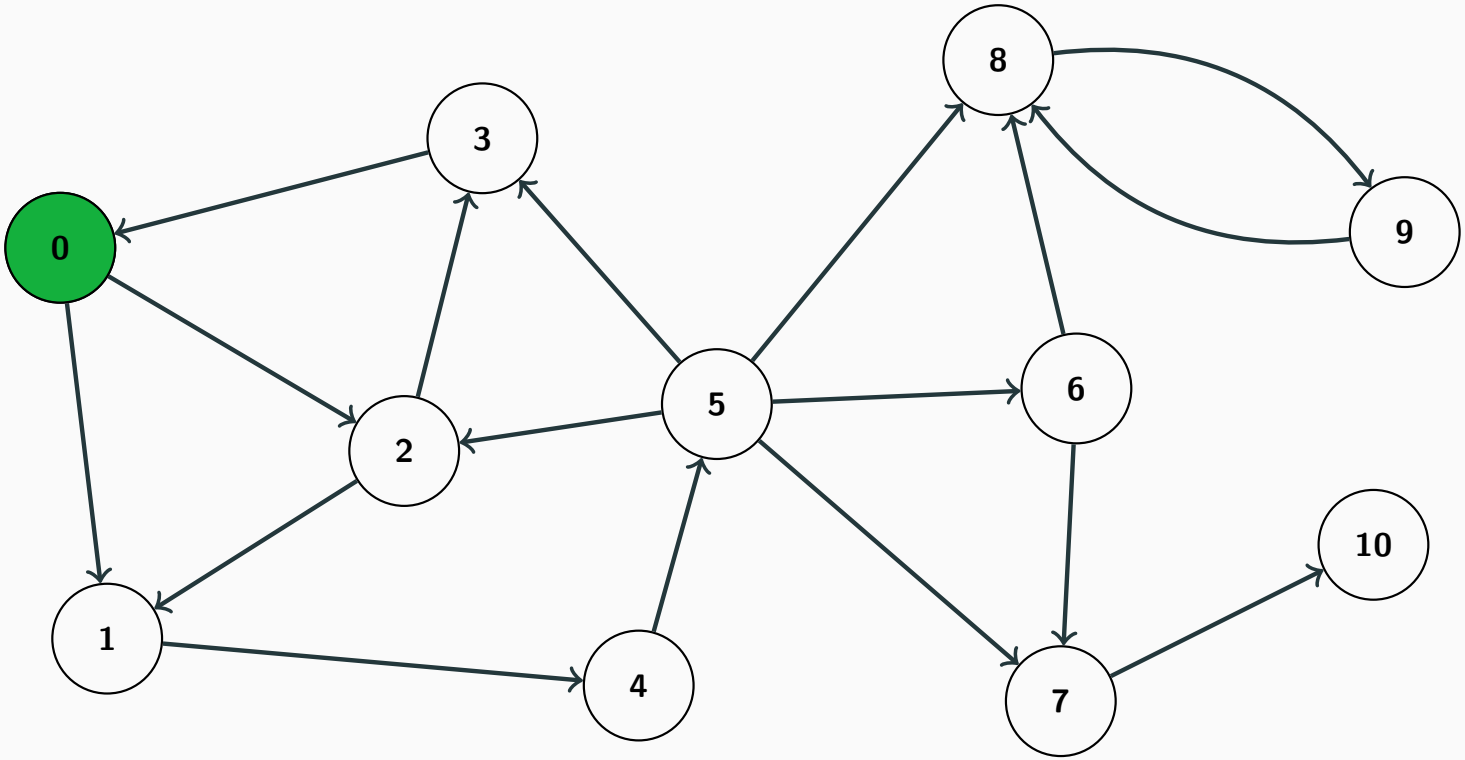
Depth-first search



Stack: |

	0	1	2	3	4	5	6	7	8	9	10
marked	0	0	0	0	0	0	0	0	0	0	0

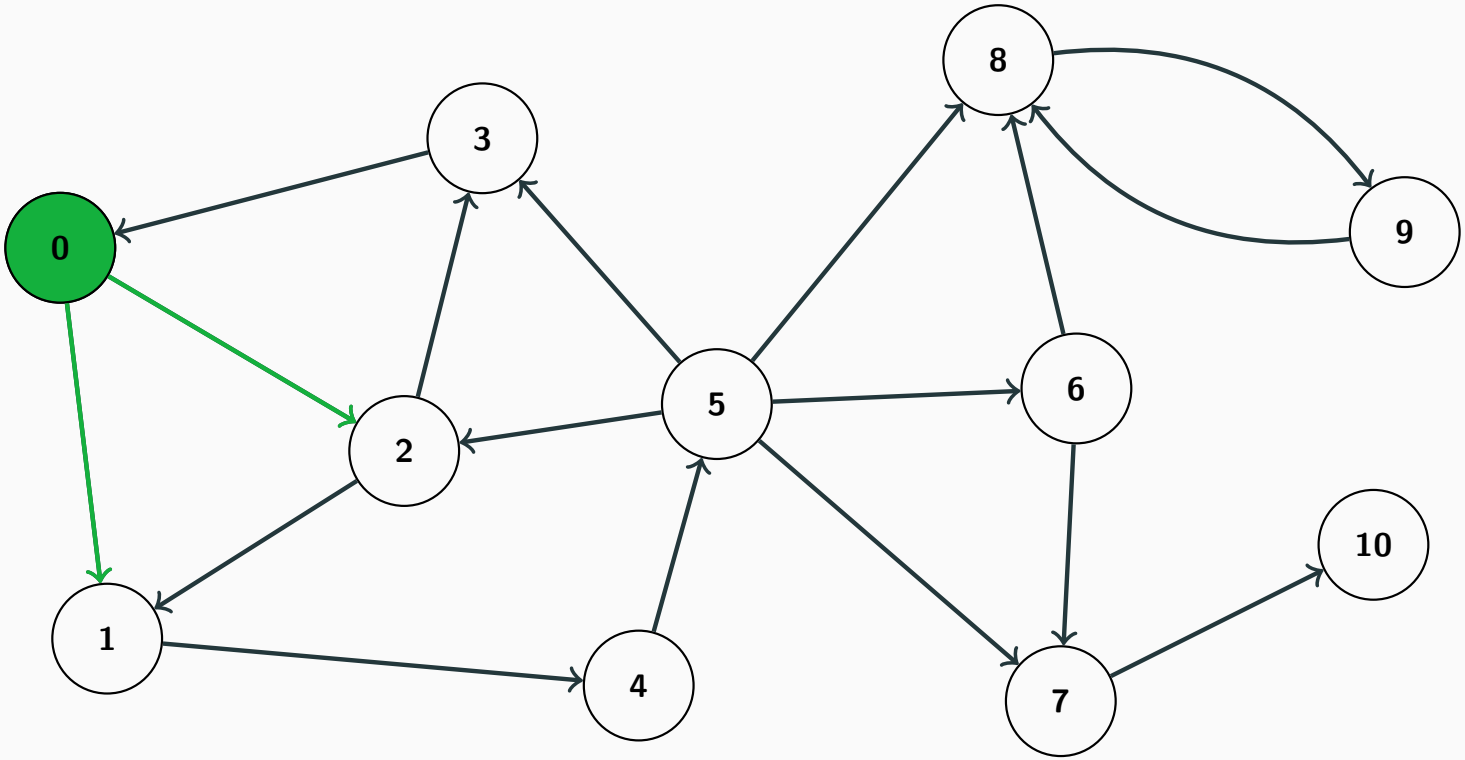
Depth-first search



Stack: 0 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	0	0	0	0	0	0	0	0	0	0

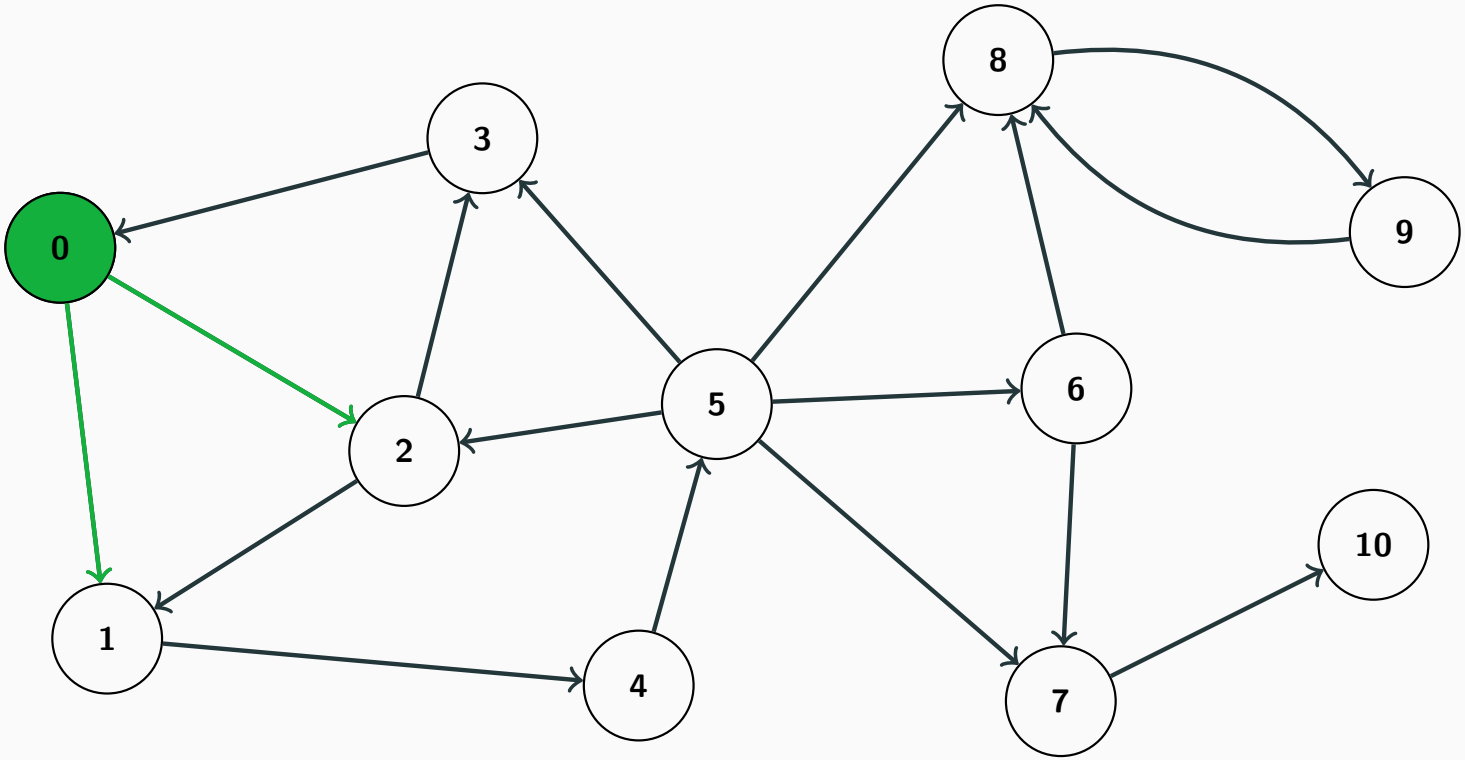
Depth-first search



Stack: 0 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	0	0	0	0	0	0	0	0	0	0

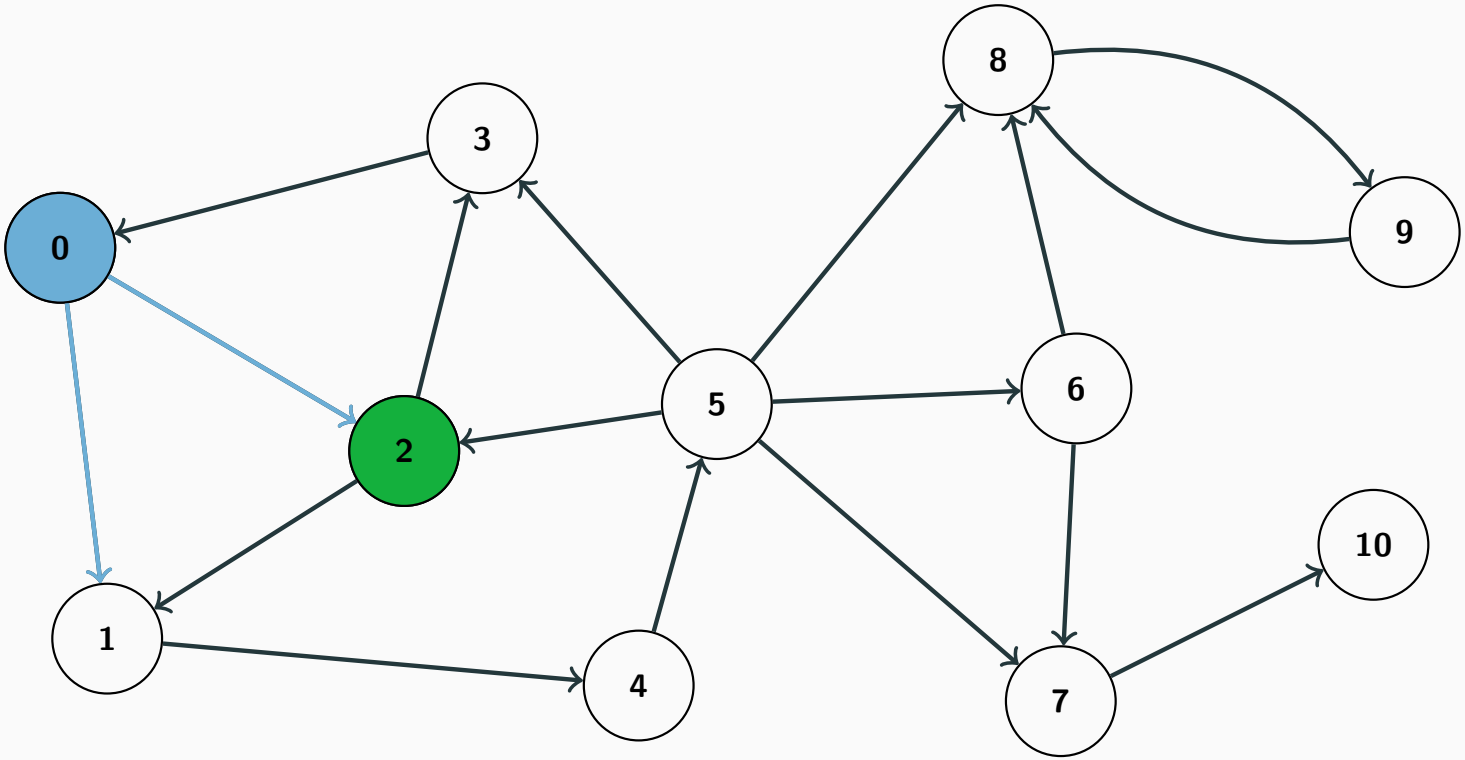
Depth-first search



Stack: 0 | 2 1

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	0	0	0	0	0	0	0

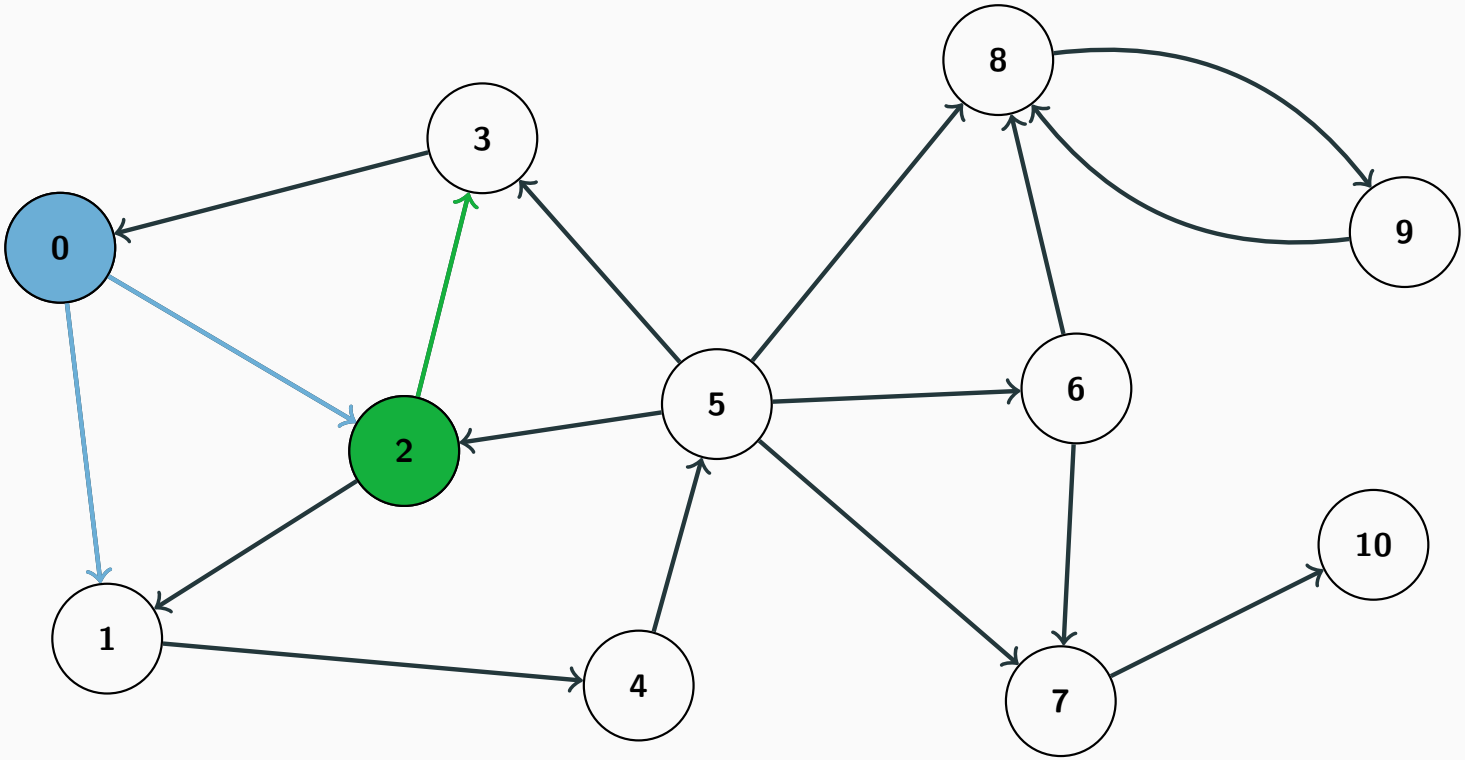
Depth-first search



Stack: 2 | 1

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	0	0	0	0	0	0	0

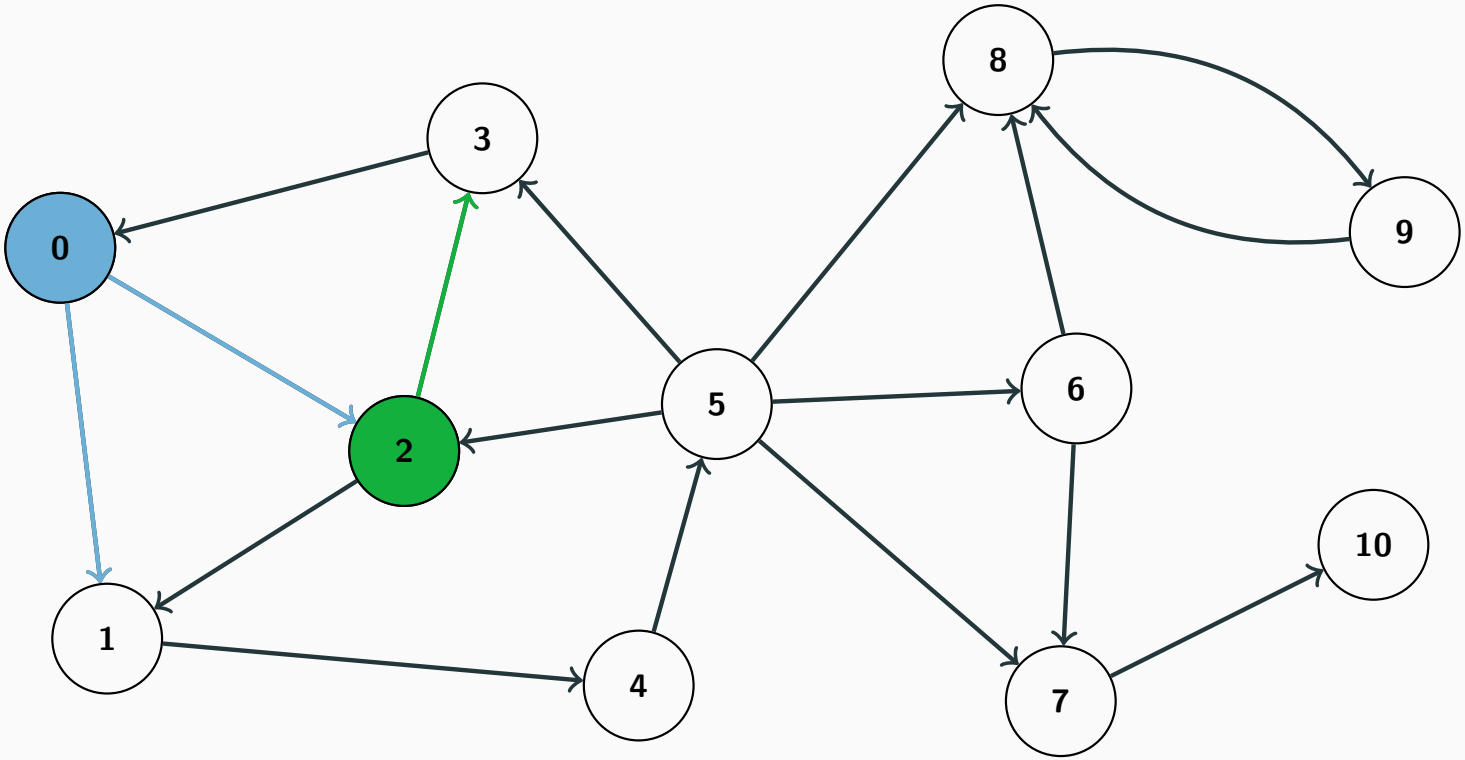
Depth-first search



Stack: 2 | 1

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	0	0	0	0	0	0	0

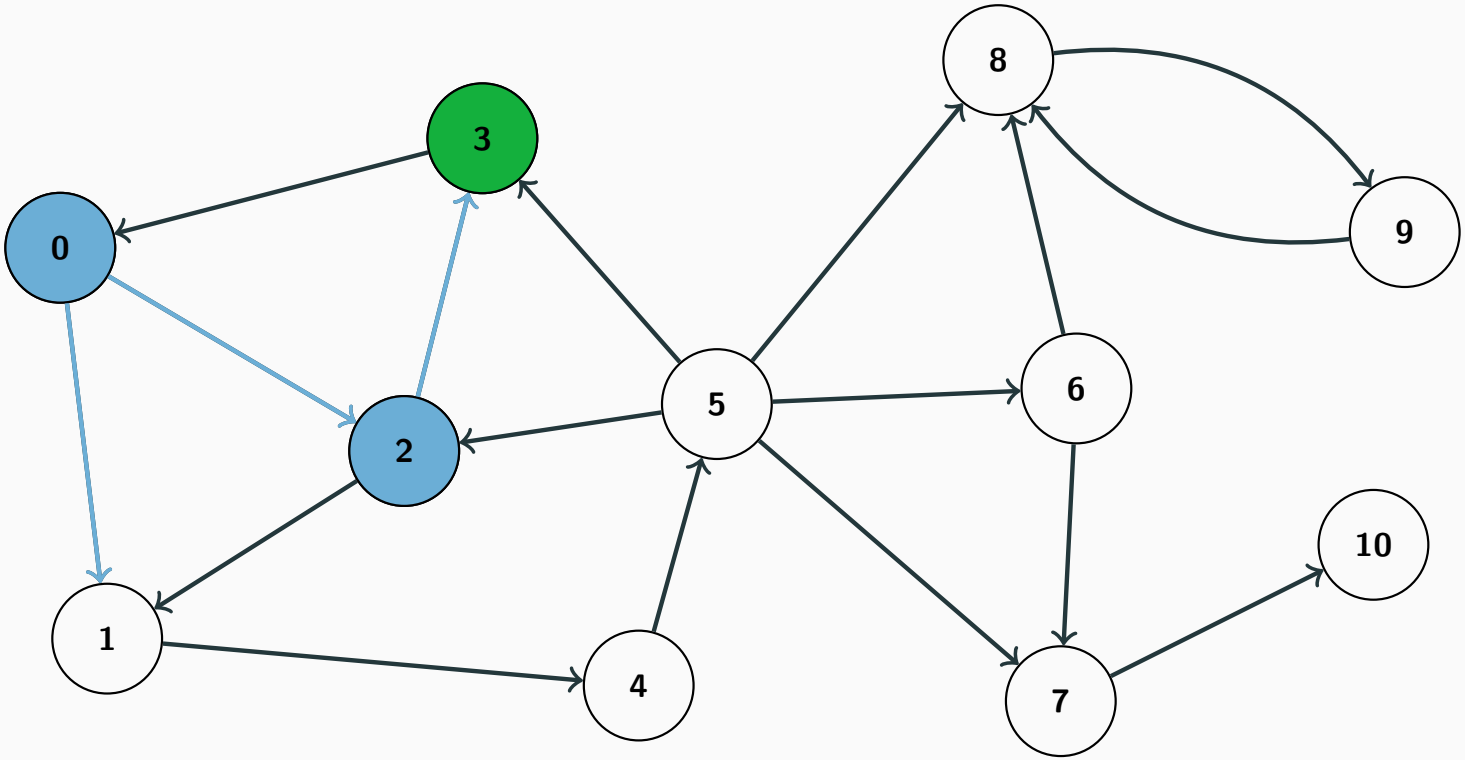
Depth-first search



Stack: 2 | 3 1

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	0	0	0	0	0	0	0

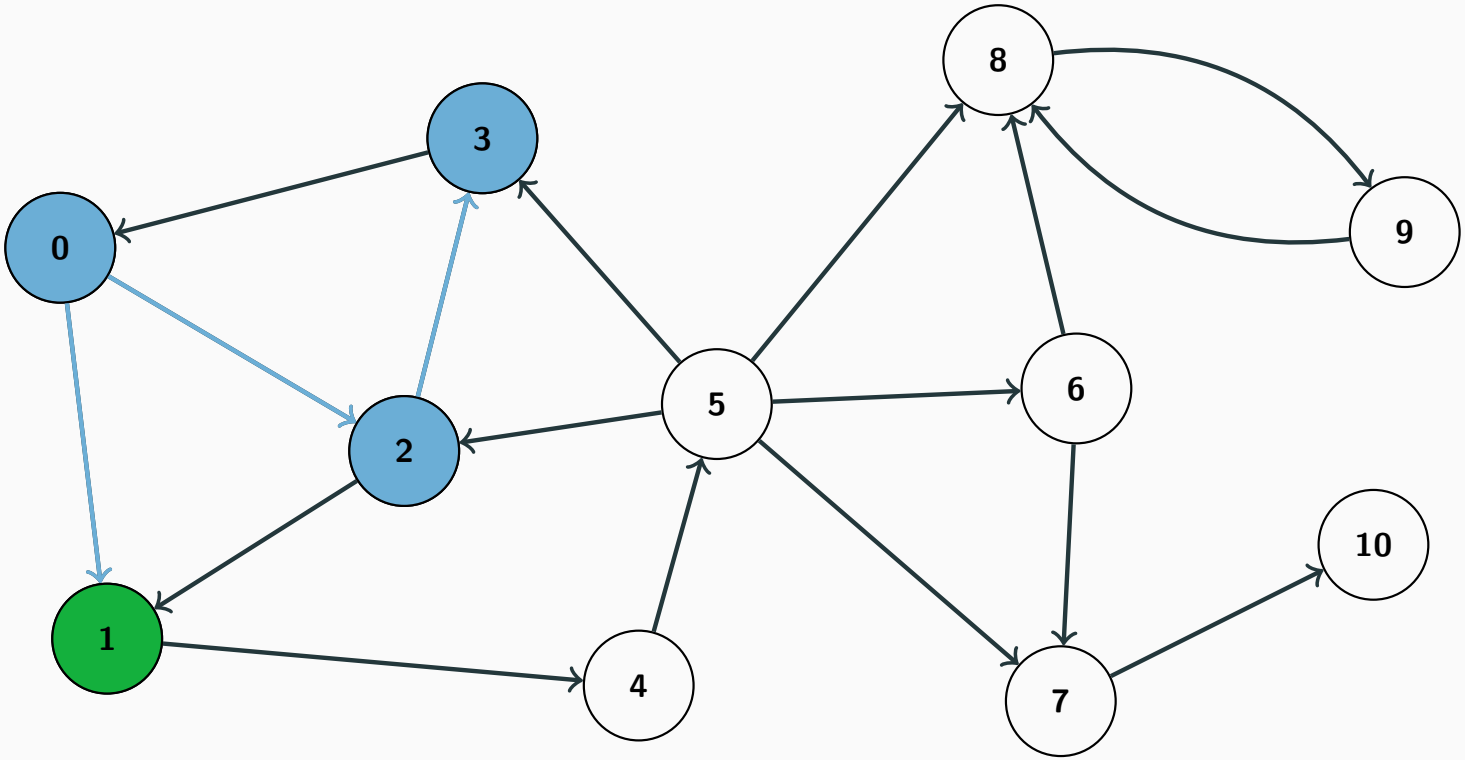
Depth-first search



Stack: 3 | 1

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	0	0	0	0	0	0	0

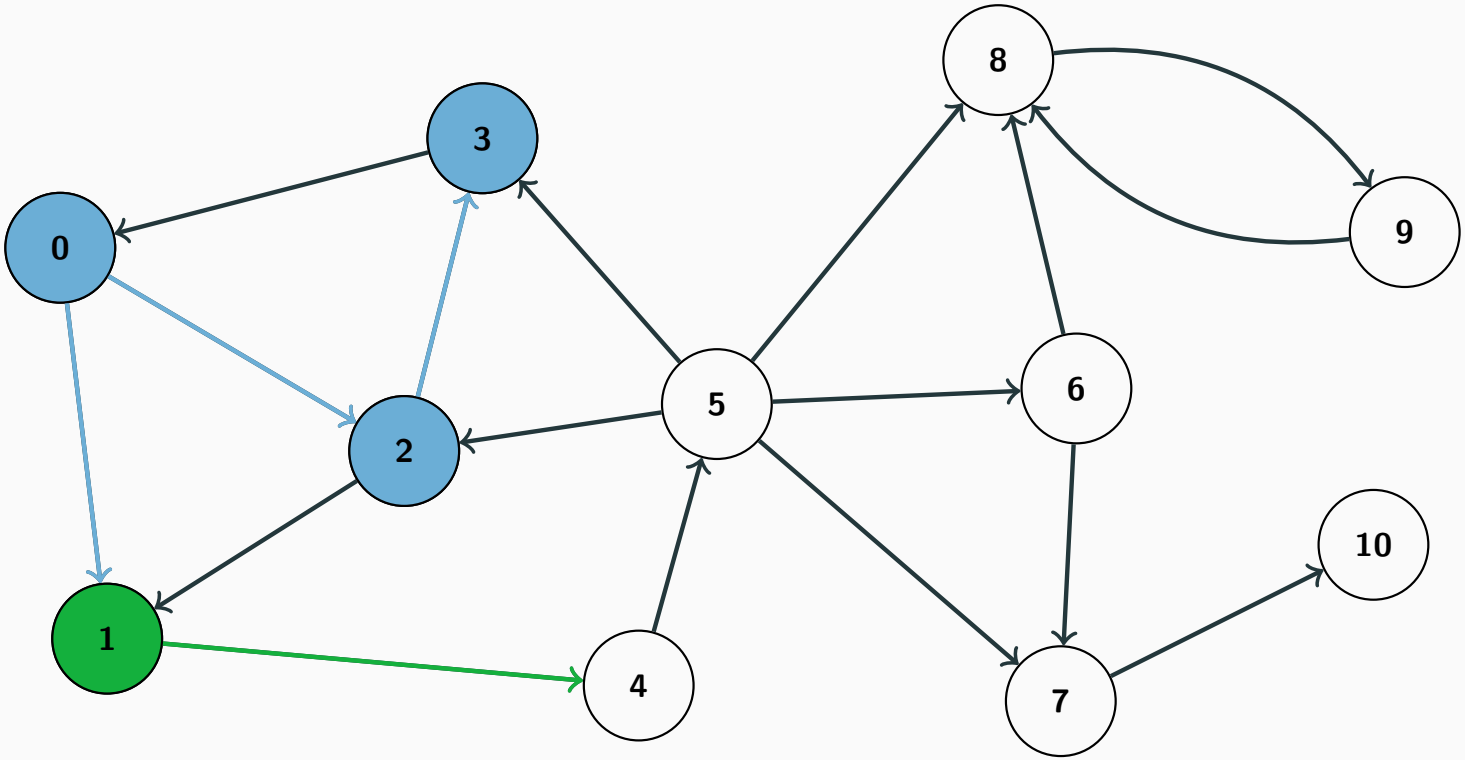
Depth-first search



Stack: 1 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	0	0	0	0	0	0	0

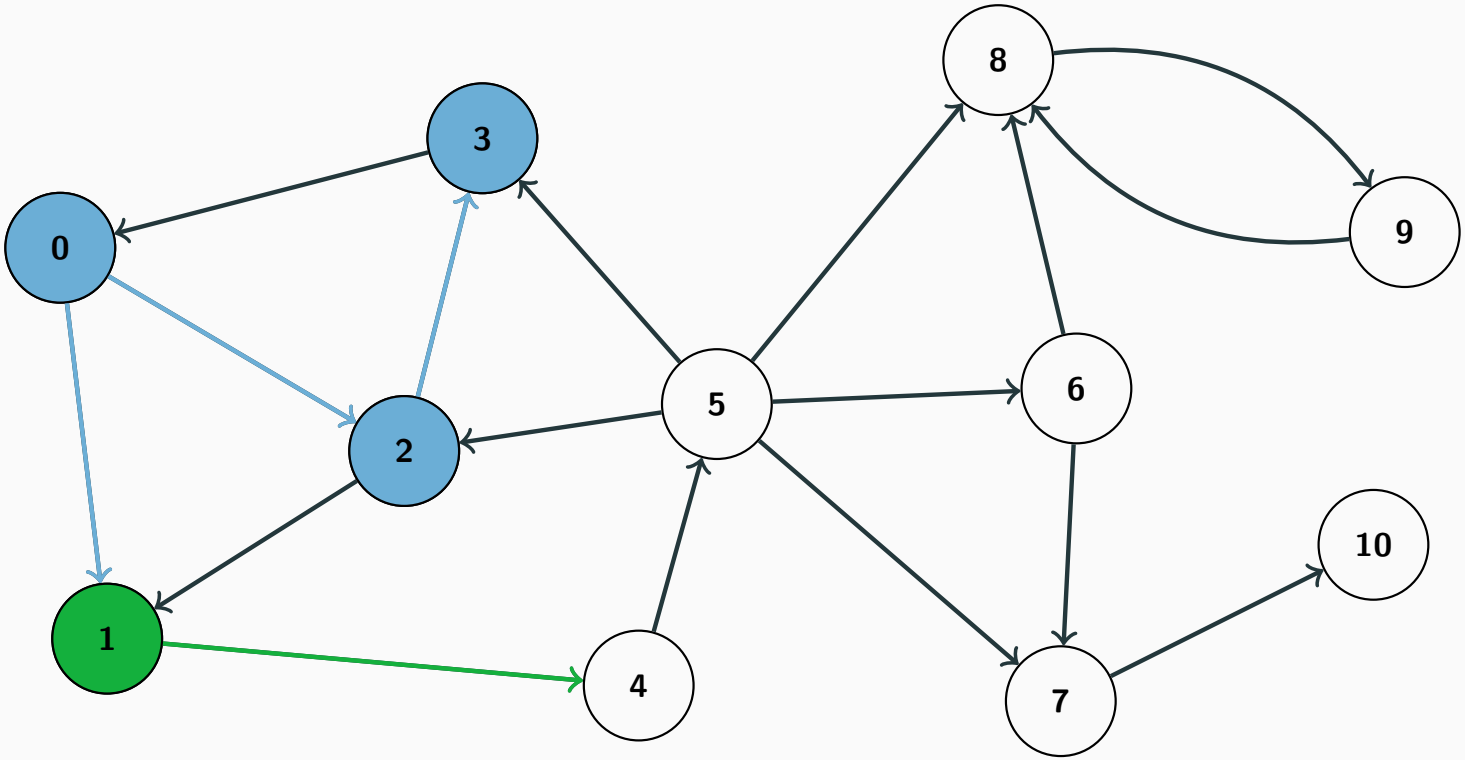
Depth-first search



Stack: 1 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	0	0	0	0	0	0	0

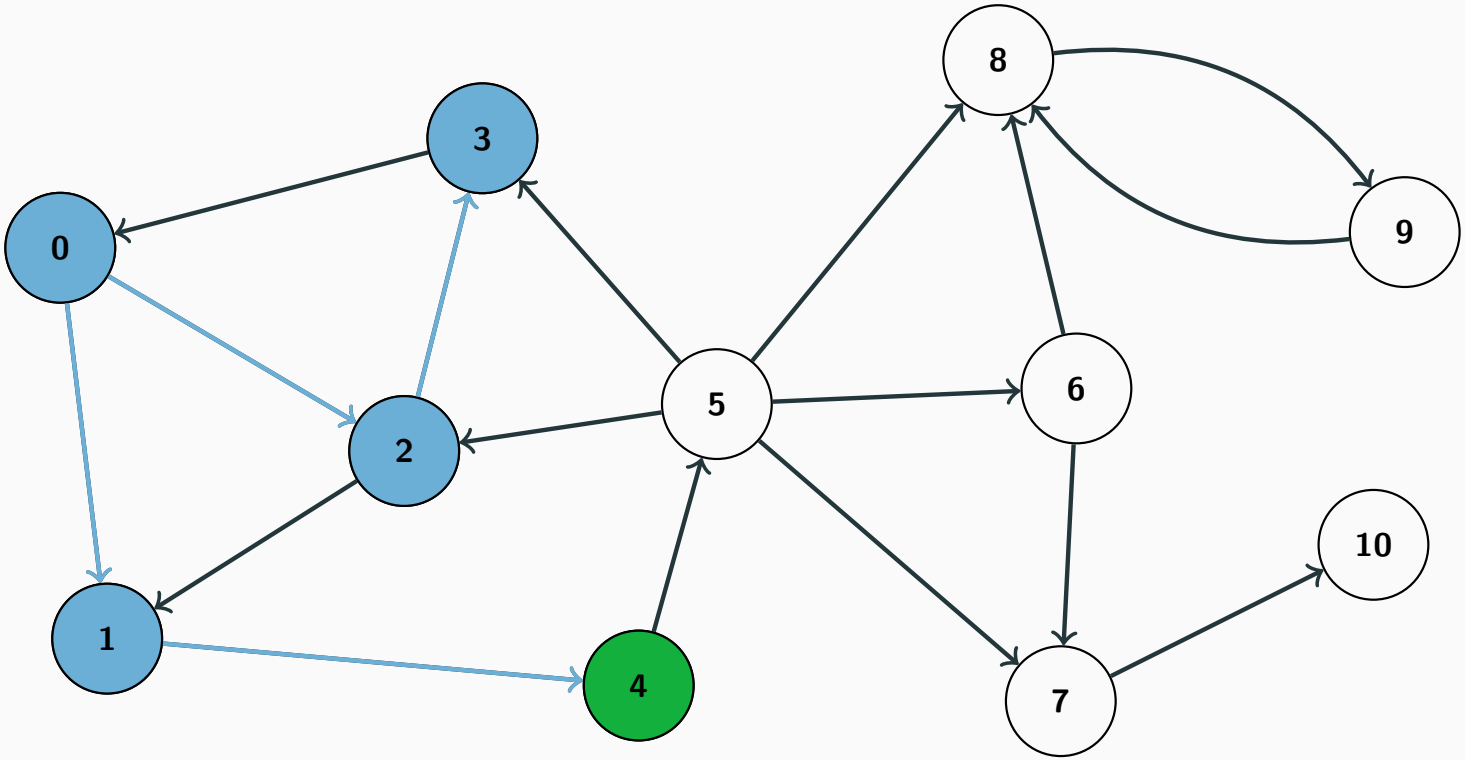
Depth-first search



Stack: 1 | 4

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	0	0	0	0	0	0

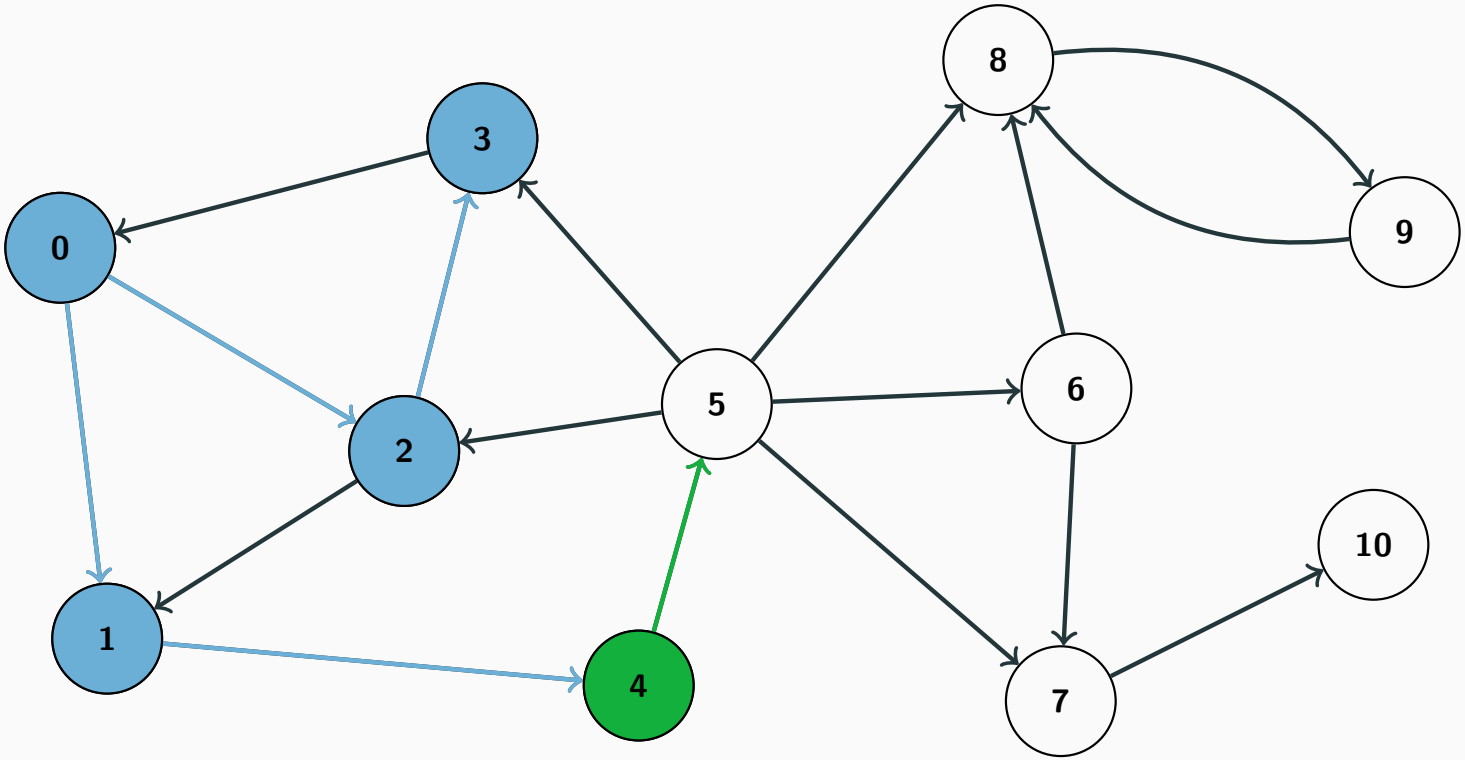
Depth-first search



Stack: 4 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	0	0	0	0	0	0

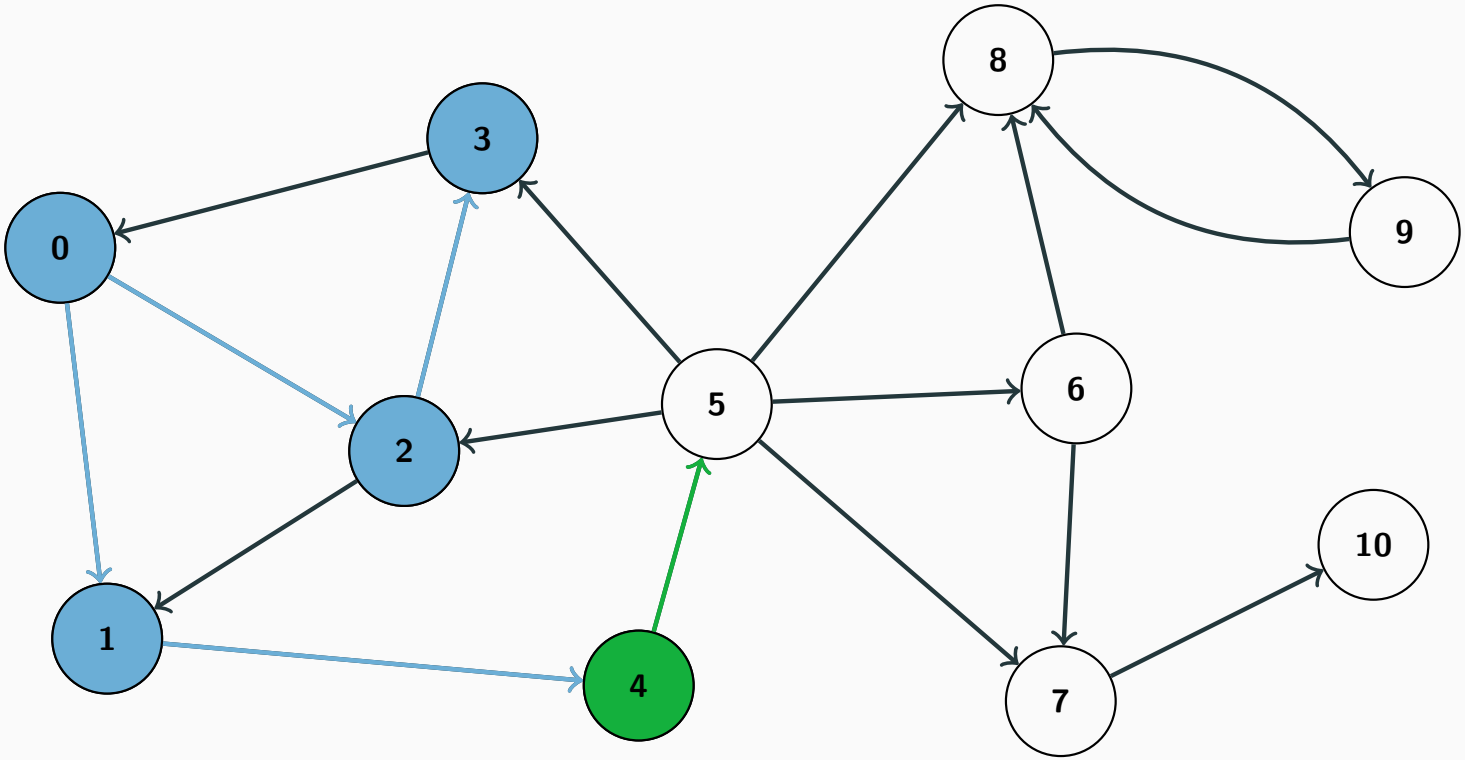
Depth-first search



Stack: 4 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	0	0	0	0	0	0

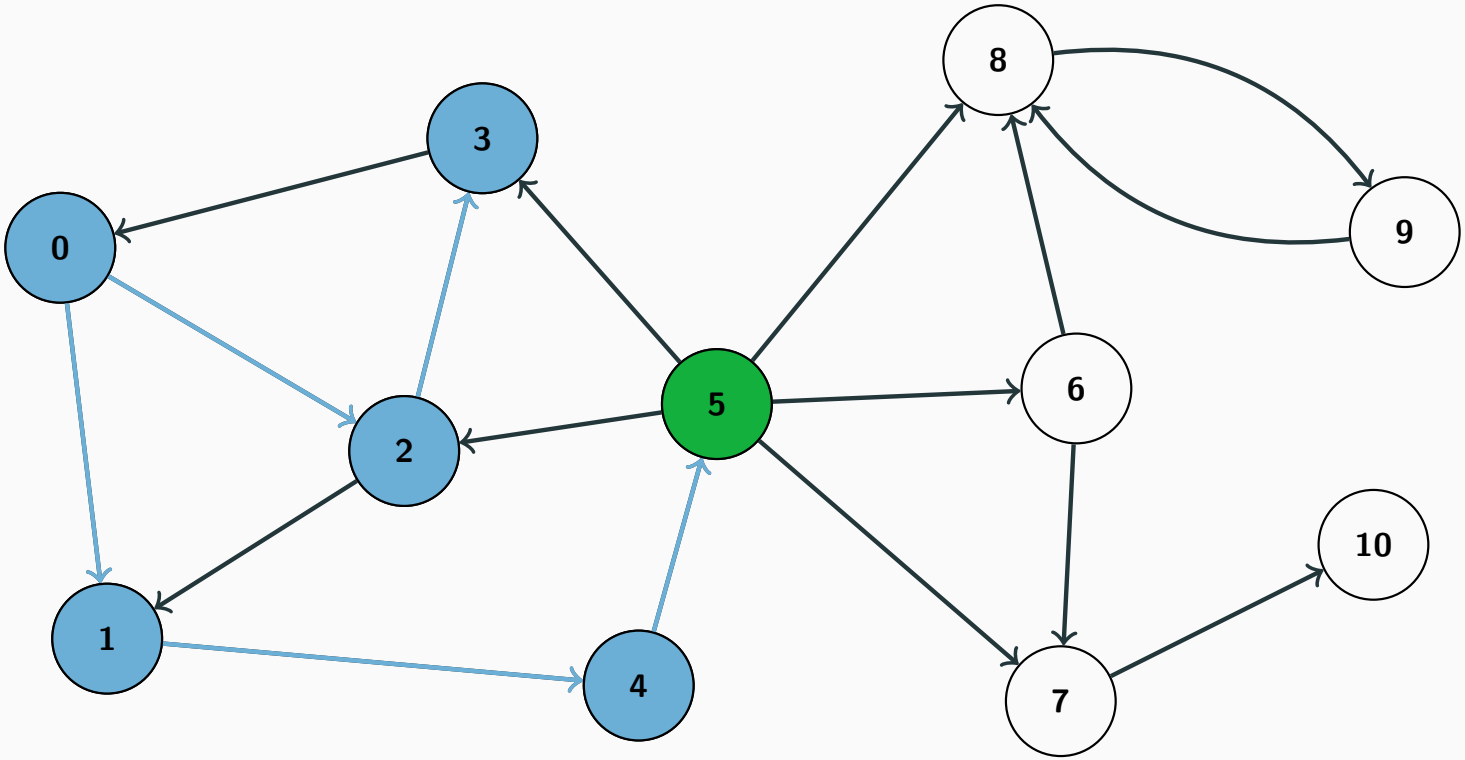
Depth-first search



Stack: 4 | 5

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

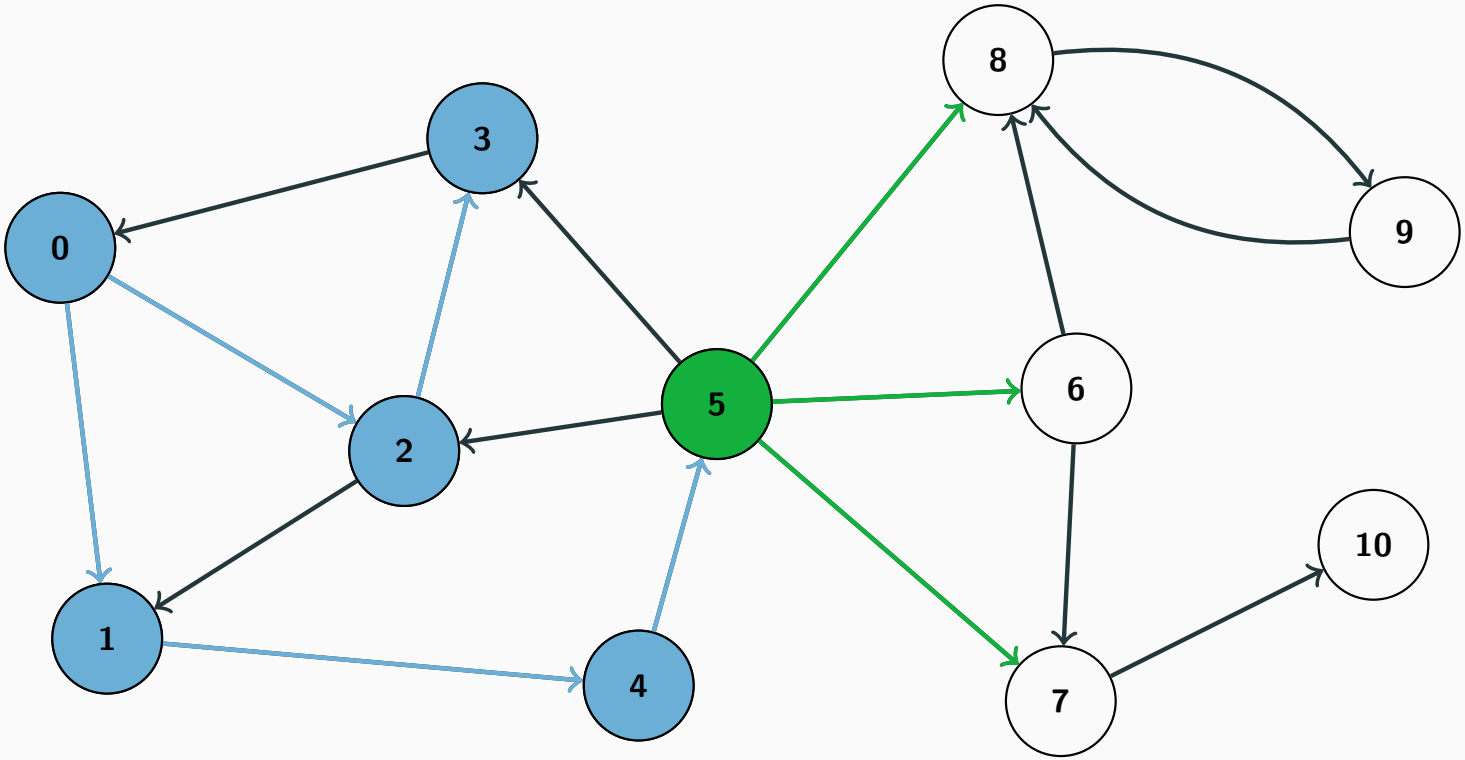
Depth-first search



Stack: 5 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

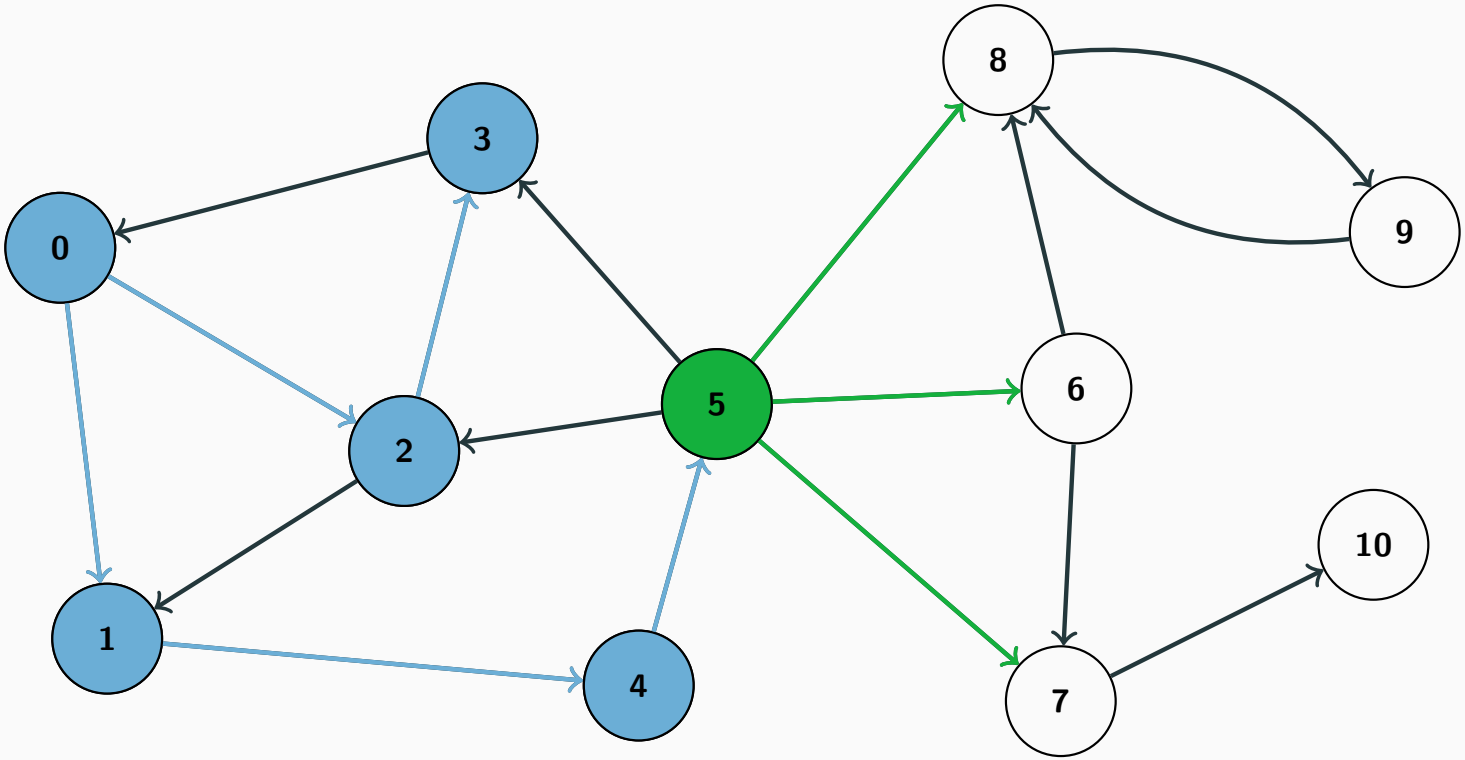
Depth-first search



Stack: 5 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

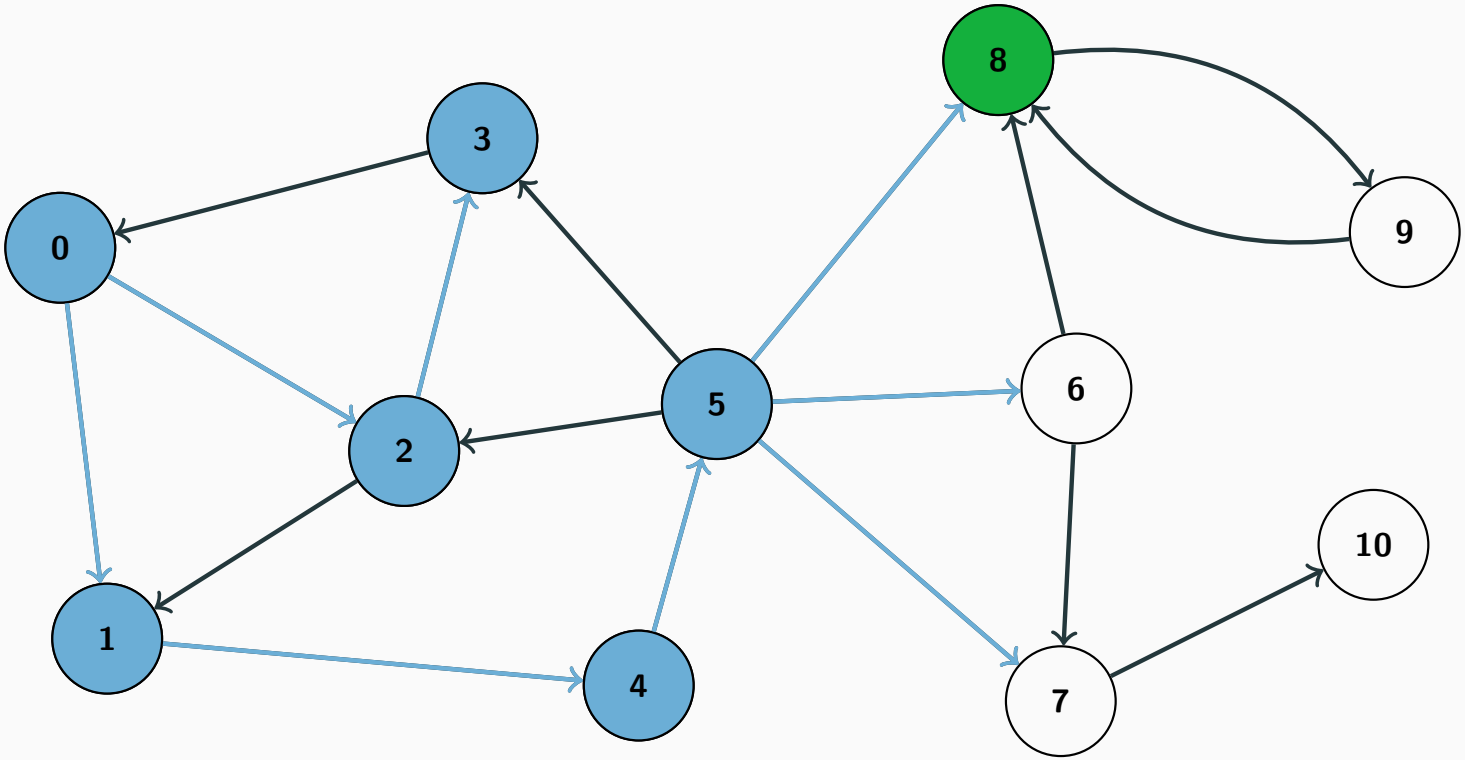
Depth-first search



Stack: 5 | 8 6 7

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

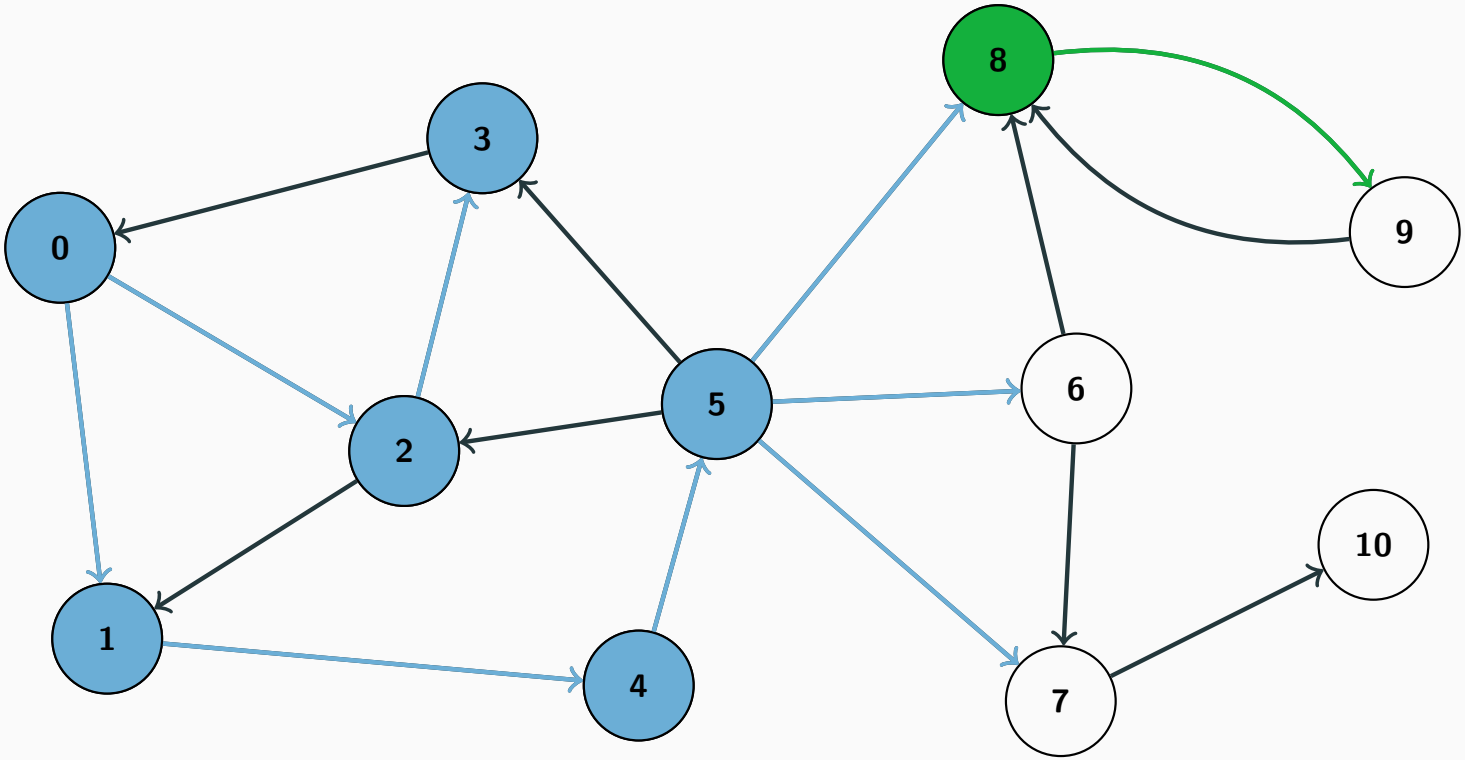
Depth-first search



Stack: 8 | 6 7

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

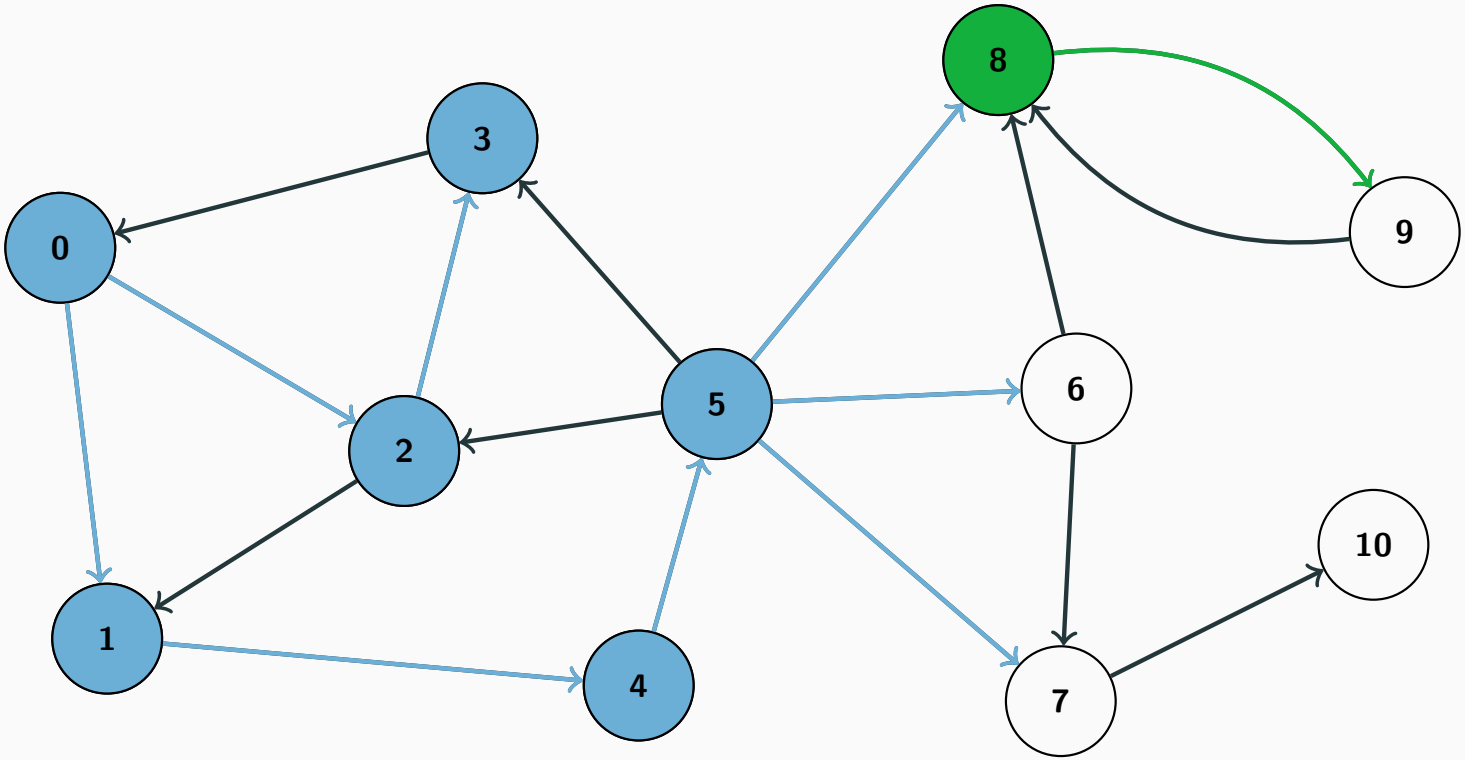
Depth-first search



Stack: 8 | 6 7

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

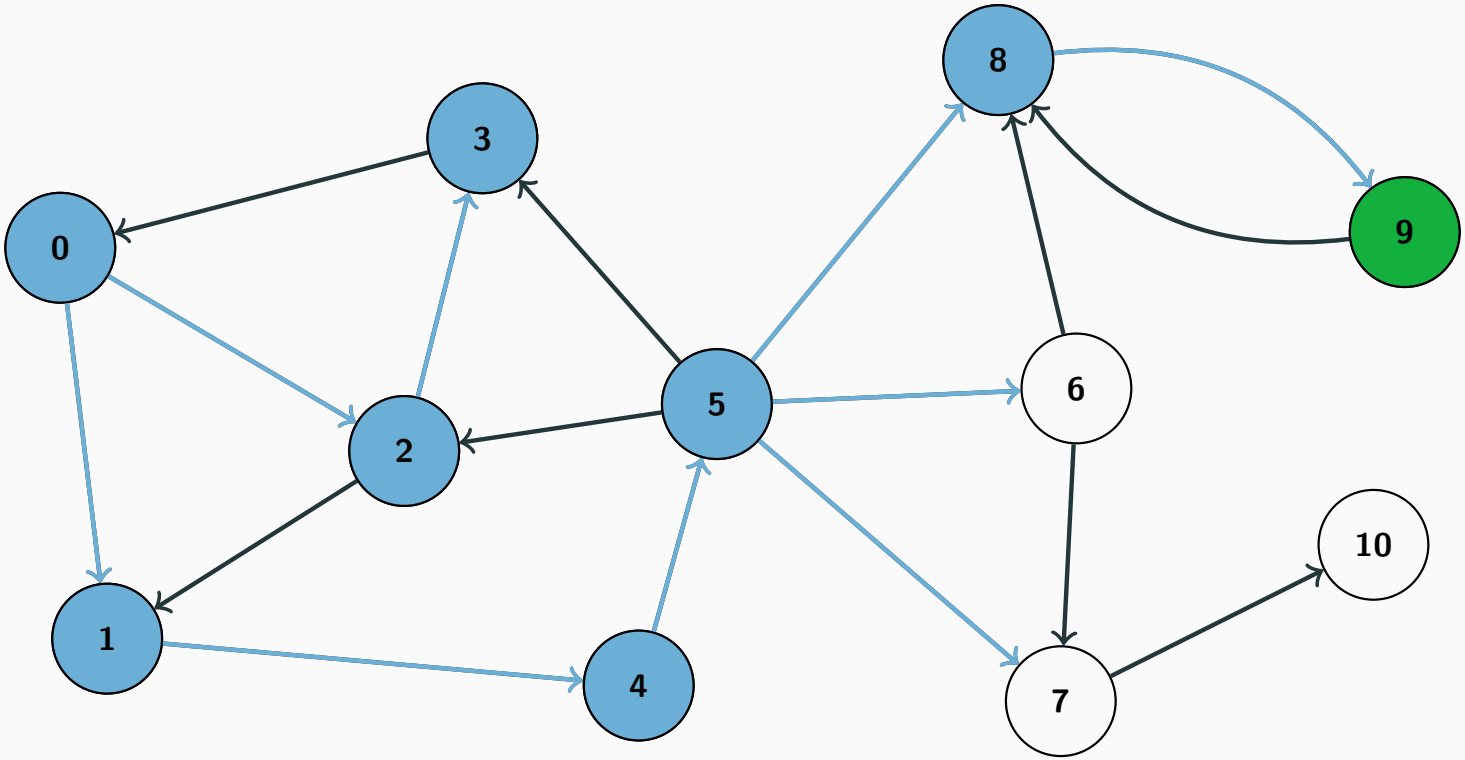
Depth-first search



Stack: 8 | 9 6 7

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	0

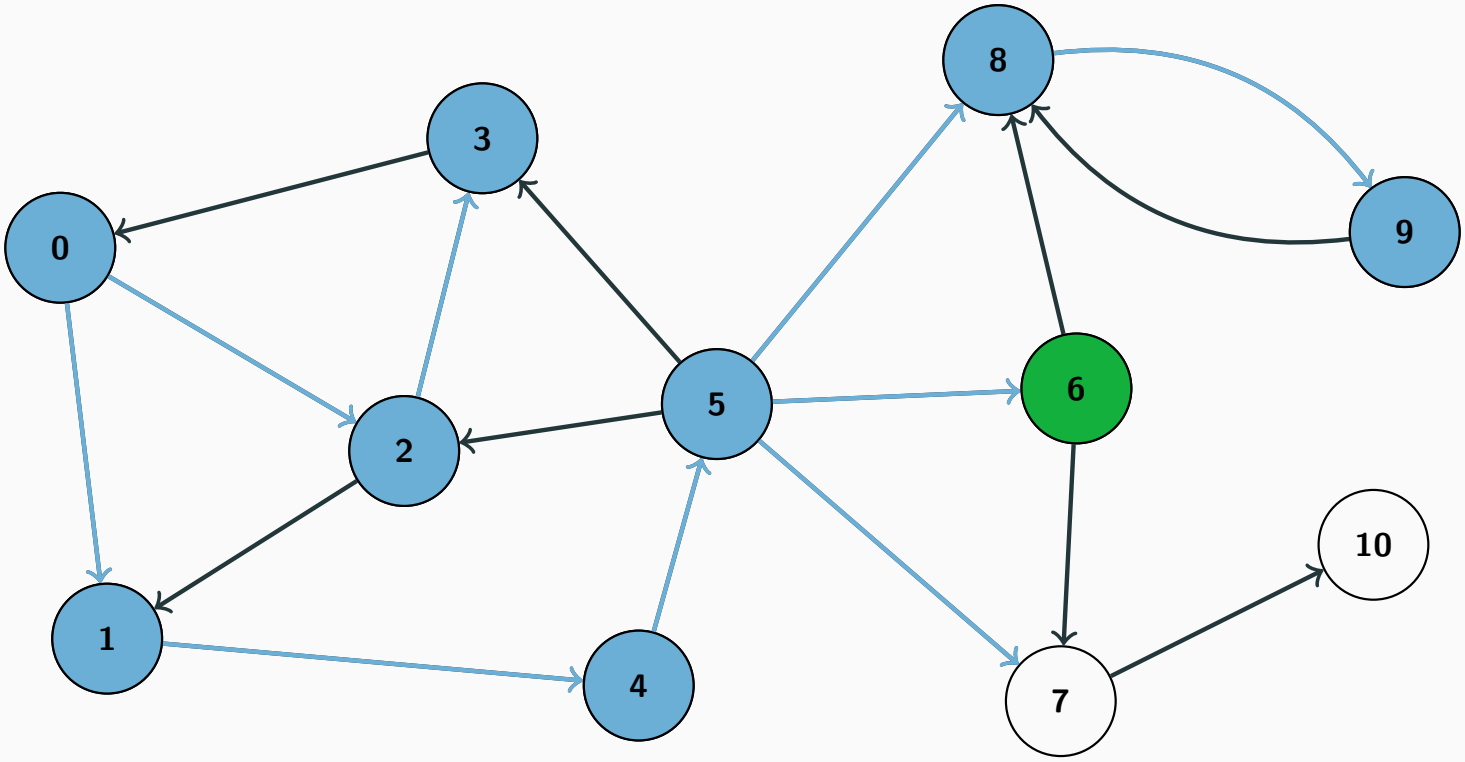
Depth-first search



Stack: 9 | 6 7

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	0

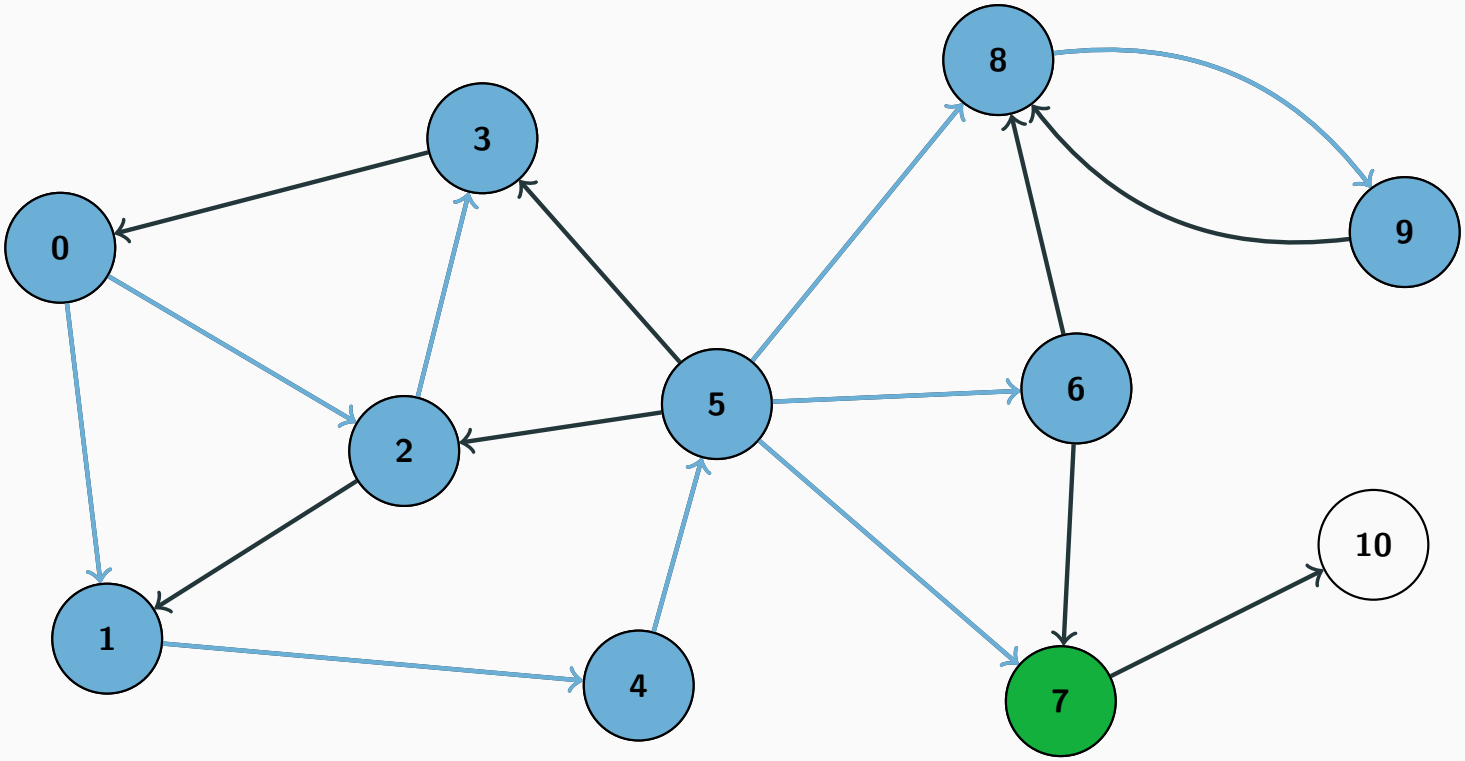
Depth-first search



Stack: 6 | 7

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	0

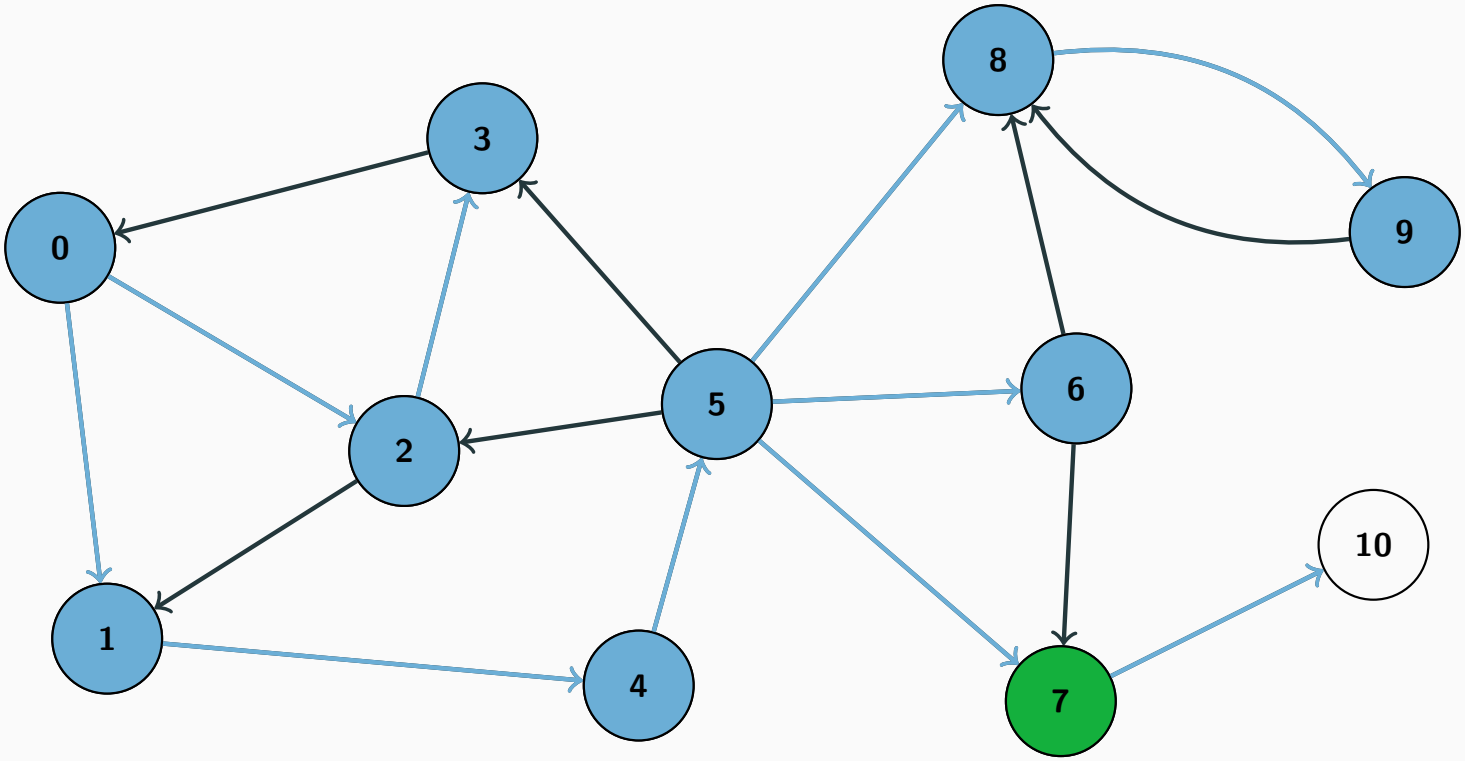
Depth-first search



Stack: 7 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	0

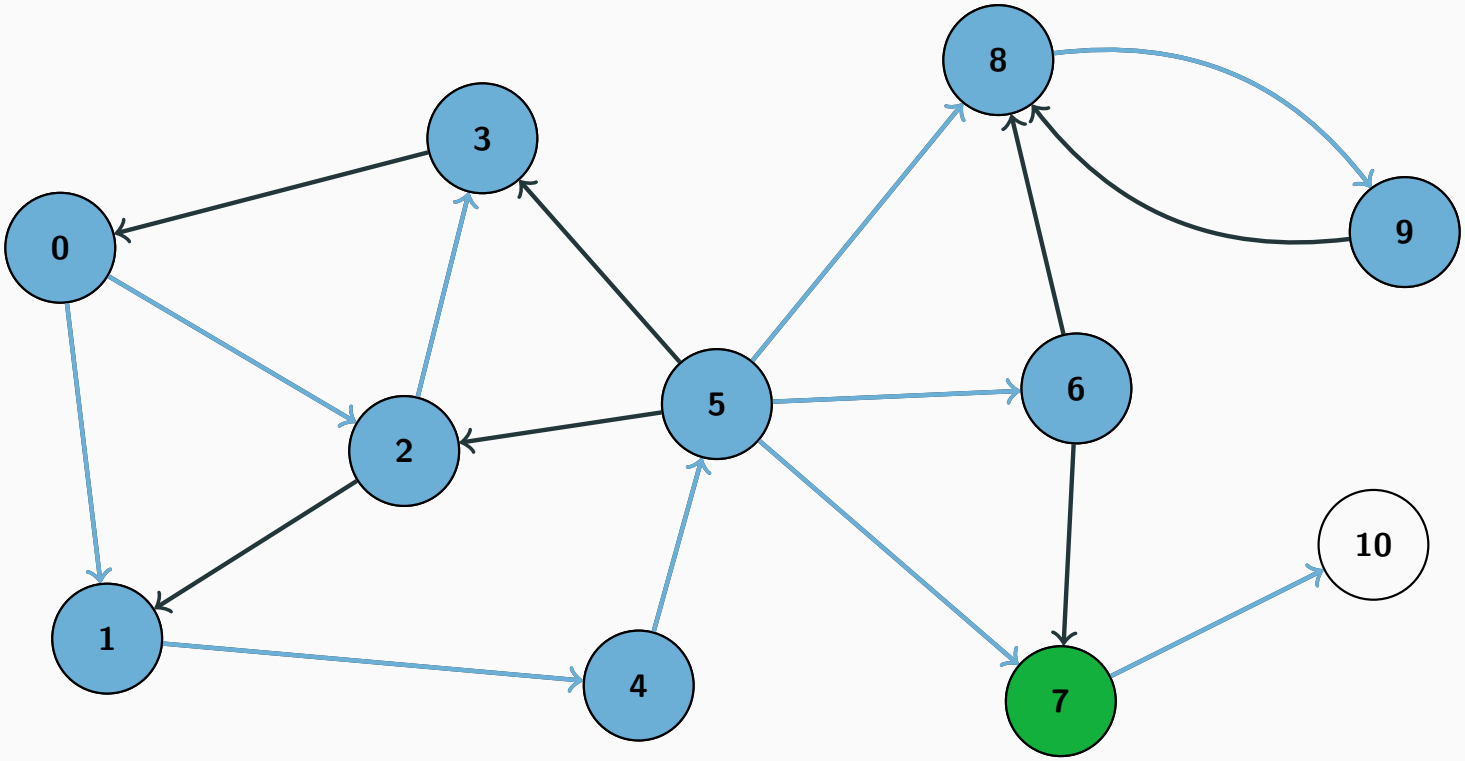
Depth-first search



Stack: 7 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	0

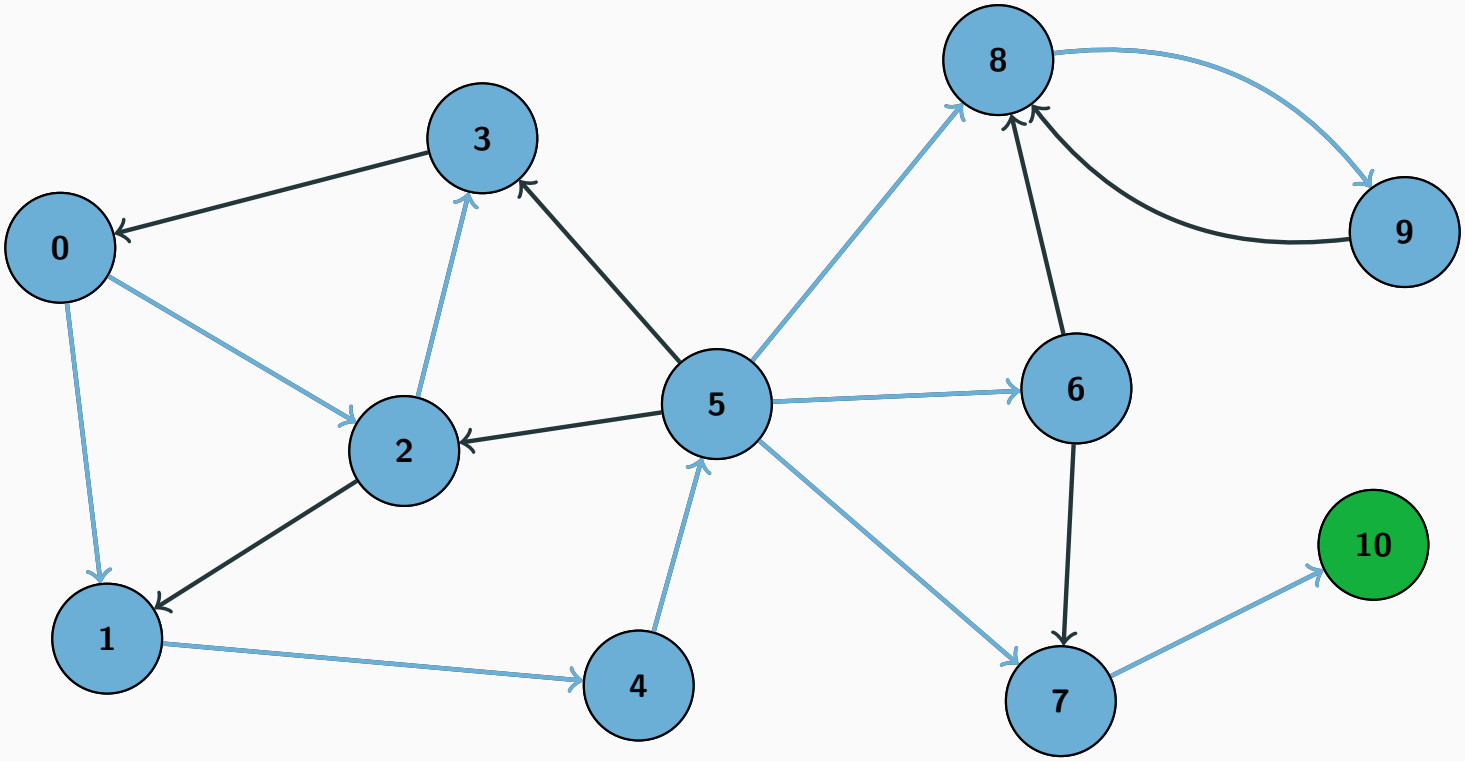
Depth-first search



Stack: 7 | 10

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

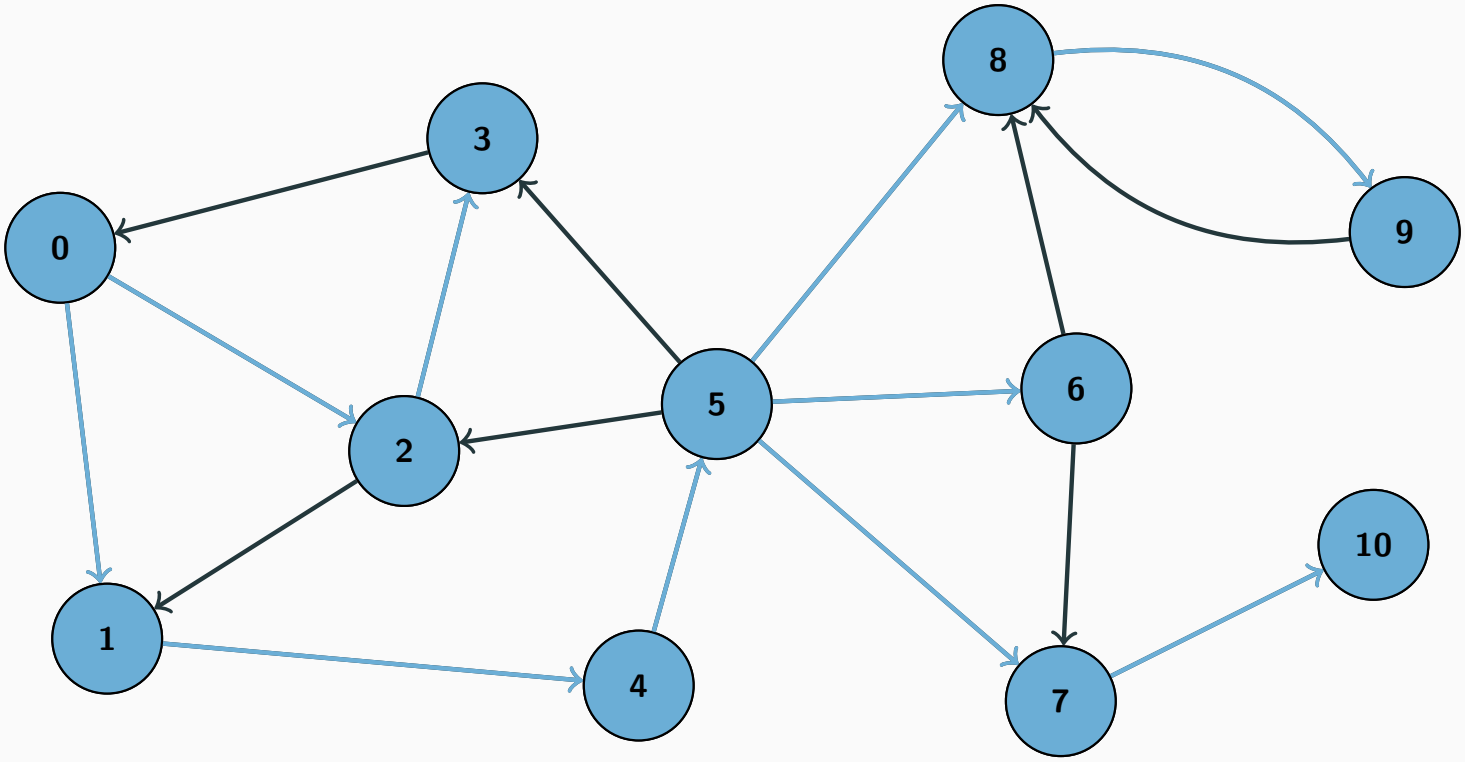
Depth-first search



Stack: 10 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

Depth-first search



Stack: |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

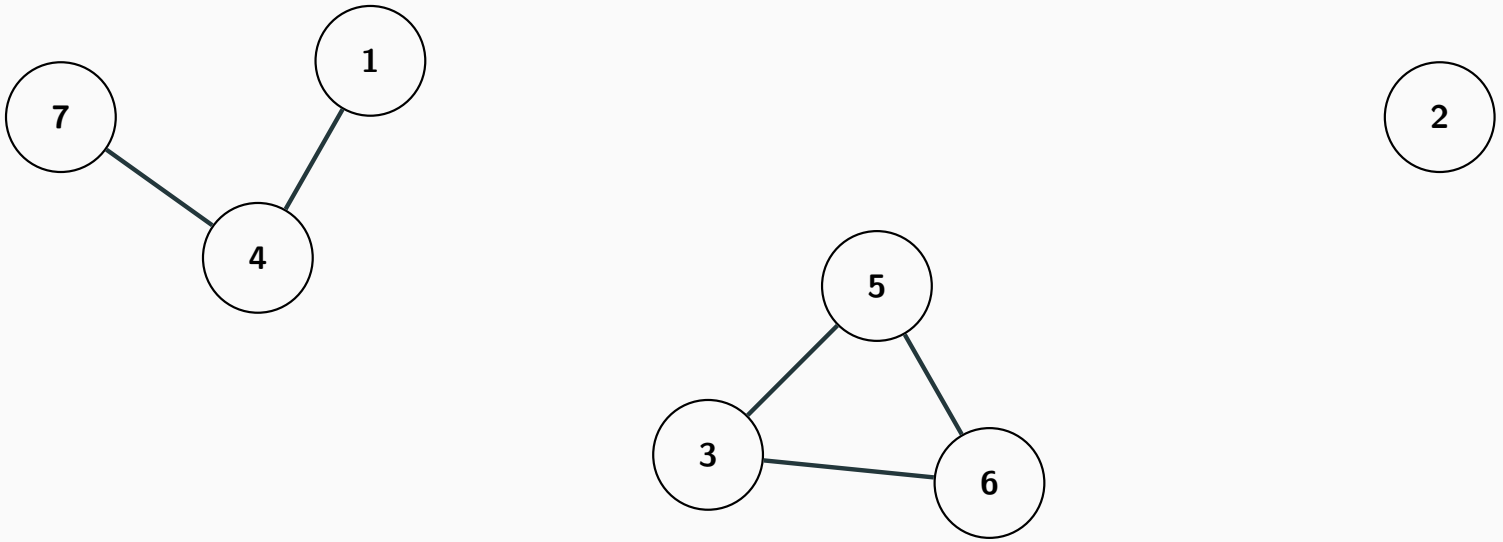
Depth-first search

```
vector<int> adj[1000];
vector<bool> visited(1000, false);

void dfs(int u) {
    if (visited[u]) {
        return;
    }

    visited[u] = true;

    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        dfs(v);
    }
}
```

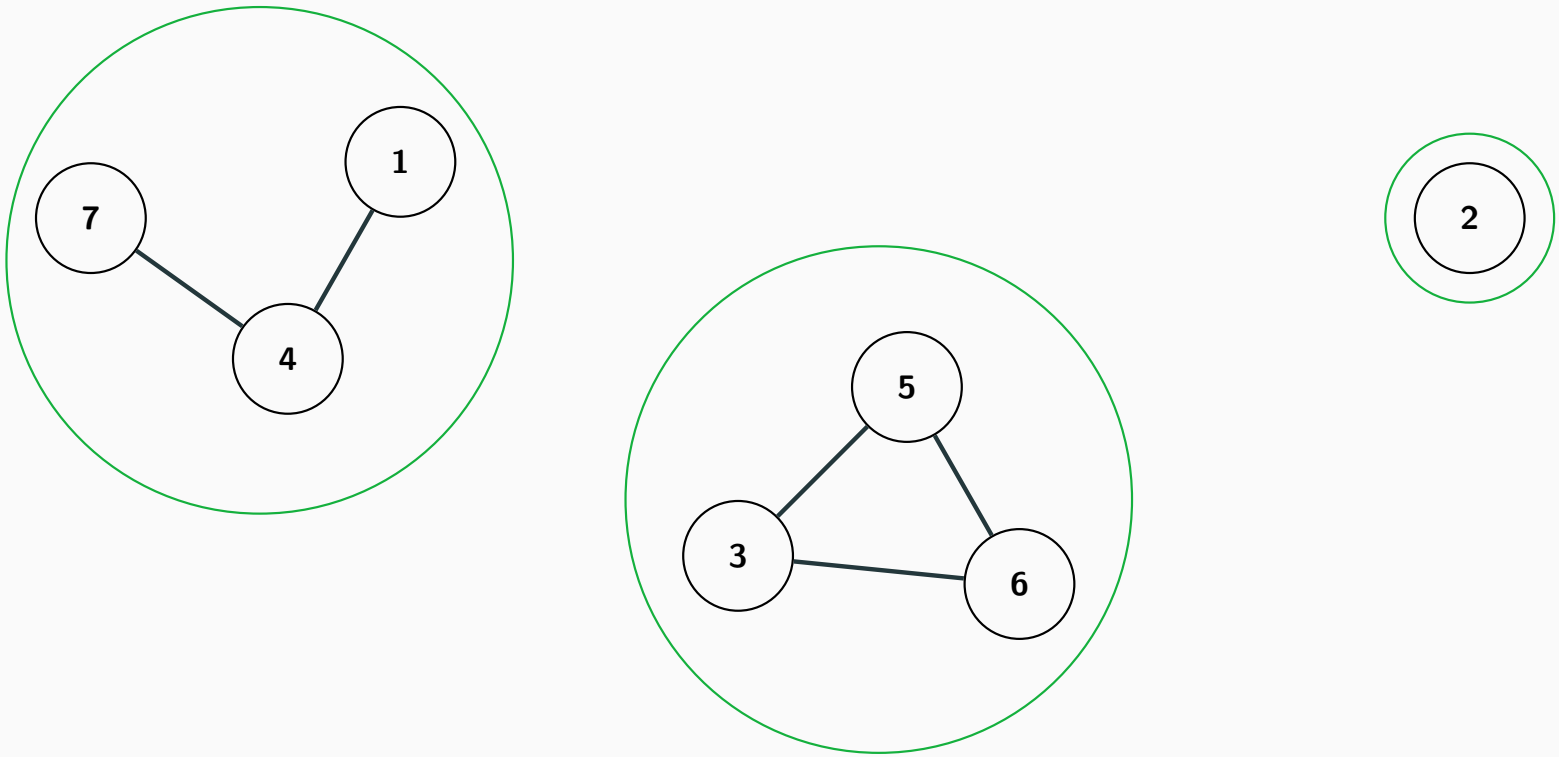
Connected components

- An *undirected graph* can be partitioned into connected components
- A connected component is a maximal subset of the vertices such that each pair of vertices is reachable from each other
- We've already seen this in a couple of problems, but we've been using Union-Find to keep track of the components

Connected components



Connected components



Connected components

- Also possible to find these components using depth-first search
- Pick some vertex we don't know anything about, and do a depth-first search from that vertex
- All vertices reachable from that starting vertex are in the same component
- Repeat this process until you have all the components
- Time complexity is $O(n + m)$

Connected components

```
vector<int> adj[1000];
vector<int> component(1000, -1);
void find_component(int cur_comp, int u) {
    if (component[u] != -1) {
        return;
    }

    component[u] = cur_comp;

    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        find_component(cur_comp, v);
    }
}
int components = 0;
for (int u = 0; u < n; u++) {
    if (component[u] == -1) {
        find_component(components, u);
        components++;
    }
}
```

Depth-first search tree

- When we do a depth-first search from a certain vertex, the edges that we go over form a tree
- When we go from a vertex to another vertex that we haven't visited before, the edge that we take is called a *forward edge*
- When we go from a vertex to another vertex that we've already visited before, the edge that we take is called a *backward edge*
- To be more specific: the forward edges form a tree

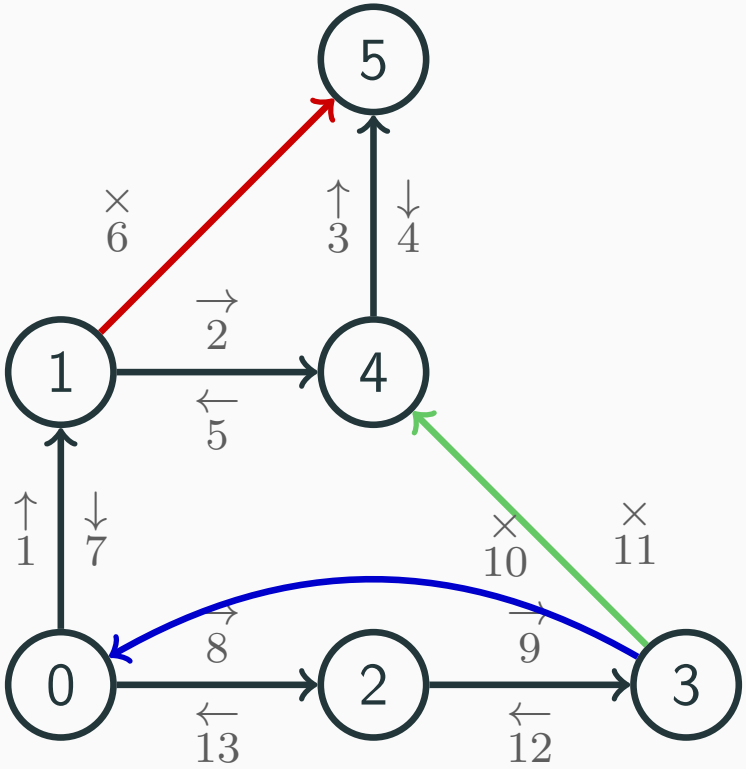
- *see example*

Depth-first search tree

- This tree of forward edges, along with the backward edges, can be analyzed to get a lot of information about the original graph

- For example: a backward edge represents a cycle in the original graph
- If there are no backward edges, then there are no cycles in the original graph (i.e. the graph is acyclic)

Depth-first search tree



Tree edges

Forward edge

Cross edge

Back edge

Analyzing the DFS tree

- Let's take a closer look at the depth-first search tree
- First, let's number each of the vertices in the order that we visit them in the depth-first search
- For each vertex, we want to know the smallest number of a vertex that we reached when exploring the subtree rooted at the current vertex
- Why? We'll see in a bit..

Analyzing the DFS tree

```
const int n = 1000;
vector<int> adj[n];
vector<int> low(n), num(n, -1);
int curnum = 0;
void analyze(int u, int p) {
    low[u] = num[u] = curnum++;
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (v == p) continue;
        if (num[v] == -1) {
            analyze(v, u);
            low[u] = min(low[u], low[v]);
        } else {
            low[u] = min(low[u], num[v]);
        }
    }
}
for (int u = 0; u < n; u++) {
    if (num[u] == -1) {
        analyze(u, -1);
    }
}
```

Analyzing the DFS tree

- Time complexity of this is just $O(n + m)$, since this is basically just one depth-first search

- Now, as promised, let's see some applications of this

Bridges

- We have a **connected undirected** graph
- Find an edge, so that if you remove that edge the graph is no longer connected

- Naive algorithm: Try removing edges, one at a time, and find the connected components of the resulting graph
- That's pretty inefficient, $O(m(n+m))$

Bridges

- Let's take a look at the values that we computed in the DFS tree
- We see that a forward edge (u, v) is a bridge if and only if `low[v]>num[u]`
- Simple to extend our analyze function to return all bridges
- Again, this is just $O(n + m)$

Bridges

```
const int n = 1000;
vector<int> adj[n];
vector<int> low(n), num(n, -1);
int curnum = 0;

vector<pair<int, int> > bridges;

void find_bridges(int u, int p) {
    low[u] = num[u] = curnum++;
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (v == p) continue;
        if (num[v] == -1) {
            find_bridges(v, u);
            low[u] = min(low[u], low[v]);
        } else {
            low[u] = min(low[u], num[v]);
        }

        if (low[v] > num[u]) {
            bridges.push_back(make_pair(u, v));
        }
    }
}

for (int u = 0; u < n; u++) {
    if (num[u] == -1) {
        find_bridges(u, -1);
    }
}
```

Strongly connected components

- We know how to find connected components in undirected graphs
- But what about directed graphs?

- Such components behave a bit differently in directed graphs, especially since if v is reachable from u, it doesn't mean that u is reachable from v

- The definition remains the same, though
- A strongly connected component is a maximal subset of the vertices such that each pair of vertices is reachable from each other

Strongly connected components

- The connected components algorithm won't work here
- Instead we can use the depth-first search tree of the graph to find these components

Strongly connected components

```
vector<int> adj[100];
vector<int> low(100), num(100, -1);
vector<bool> incomp(100, false);
int curnum = 0;

stack<int> comp;

void scc(int u) {

    // scc code...
}

for (int i = 0; i < n; i++) {
    if (num[i] == -1) {
        scc(i);
    }
}
```

Strongly connected components

```
void scc(int u) {
    comp.push(u);
    incomp[u] = true;

    low[u] = num[u] = curnum++;
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (num[v] == -1) {
            scc(v);
            low[u] = min(low[u], low[v]);
        } else if (incomp[v]) {
            low[u] = min(low[u], num[v]);
        }
    }

    if (num[u] == low[u]) {
        printf("comp: ");
        while (true) {
            int cur = comp.top();
            comp.pop();
            incomp[cur] = false;
            printf("%d, ", cur);
            if (cur == u) {
                break;
            }
        }

        printf("\n");
    }
}
```

Strongly connected components

- Time complexity?
- Basically just the DFS analyze function (which was $O(n+m)$), with one additional loop to construct the component
- But each vertex is only in one component...
- Time complexity still just $O(n+m)$

Example problem: Tourist safety

Tourists sometimes like to explore the cities they're in. They start at their hotel, and then drive randomly around the city. This can be dangerous, however, as when the tourist becomes tired and wants to go home, it may not be possible!

Given a directed road network, determine which road intersections are tourist-safe. A road intersection u is tourist-safe if for each road intersection v reachable from u, u is reachable from v.

Topological sort

- We have n tasks
- Each task i has a list of tasks that must be completed before we can start task i
- Find an order in which we can process the tasks
- Can be represented as a directed graph
  - Each task is a vertex in the graph
  - If task j should be finished before task i, then we add a directed edge from vertex i to vertex j
- Notice that this can't be solved if the graph contains a cycle
- A modified depth-first search can be used to find an ordering in $O(n + m)$ time, or determine that one does not exist

Topological sort

```
vector<int> adj[1000];
vector<bool> visited(1000, false);
vector<int> order;

void topsort(int u) {
    if (visited[u]) {
        return;
    }

    visited[u] = true;
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        topsort(v);
    }

    order.push_back(u);
}

for (int u = 0; u < n; u++) {
    topsort(u);
}
```

Topological sort

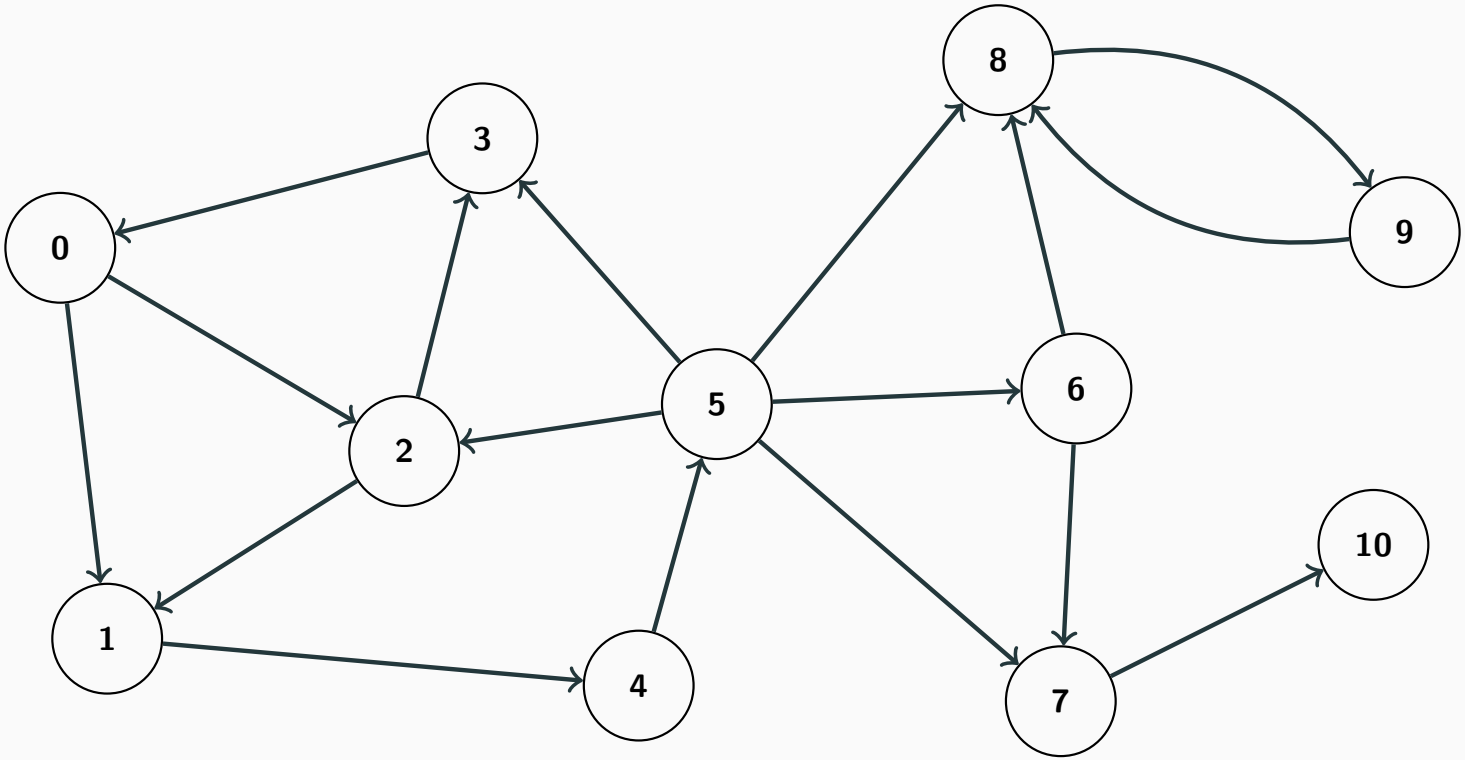
- We will see this again for BFS, as there is another way to solve this
- In my experience many students find the other solution easier to work with

Breadth-first search

Breadth-first search

- There's another search algorithm called Breadth-first search
- Only difference is the order in which it visits the vertices
- It goes in order of increasing distance from the source vertex

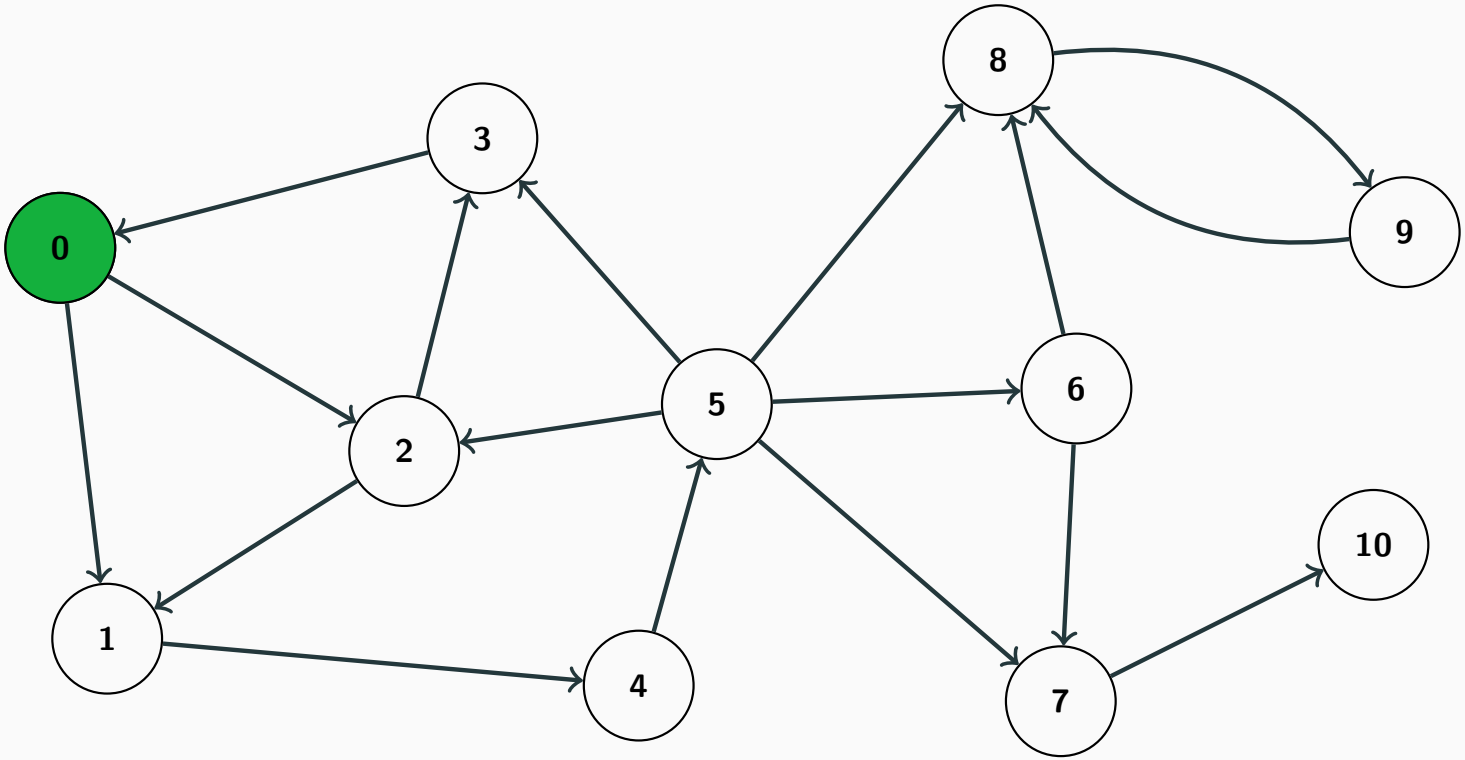
Breadth-first search



Queue:

	0	1	2	3	4	5	6	7	8	9	10
marked	0	0	0	0	0	0	0	0	0	0	0

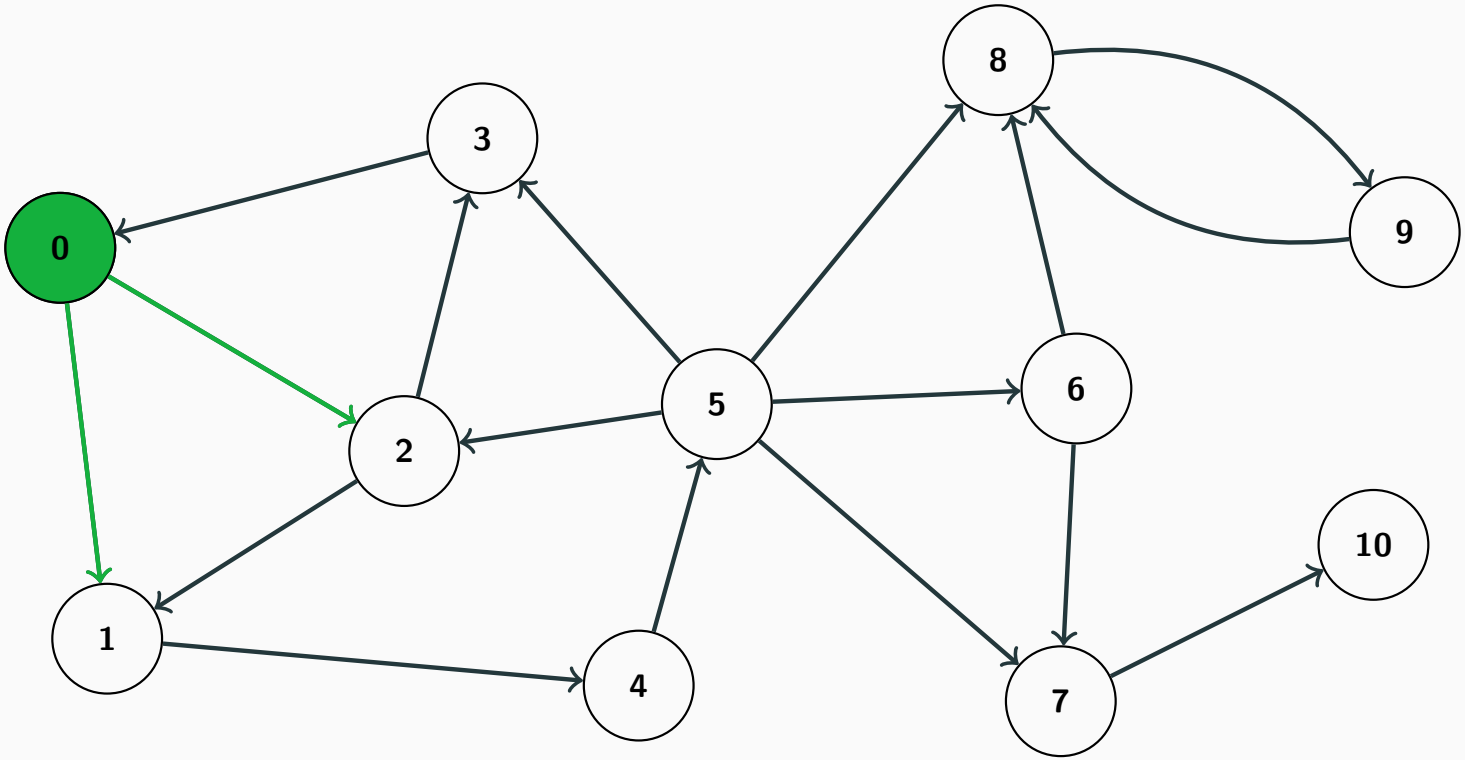
Breadth-first search



Queue: 0

	0	1	2	3	4	5	6	7	8	9	10
marked	1	0	0	0	0	0	0	0	0	0	0

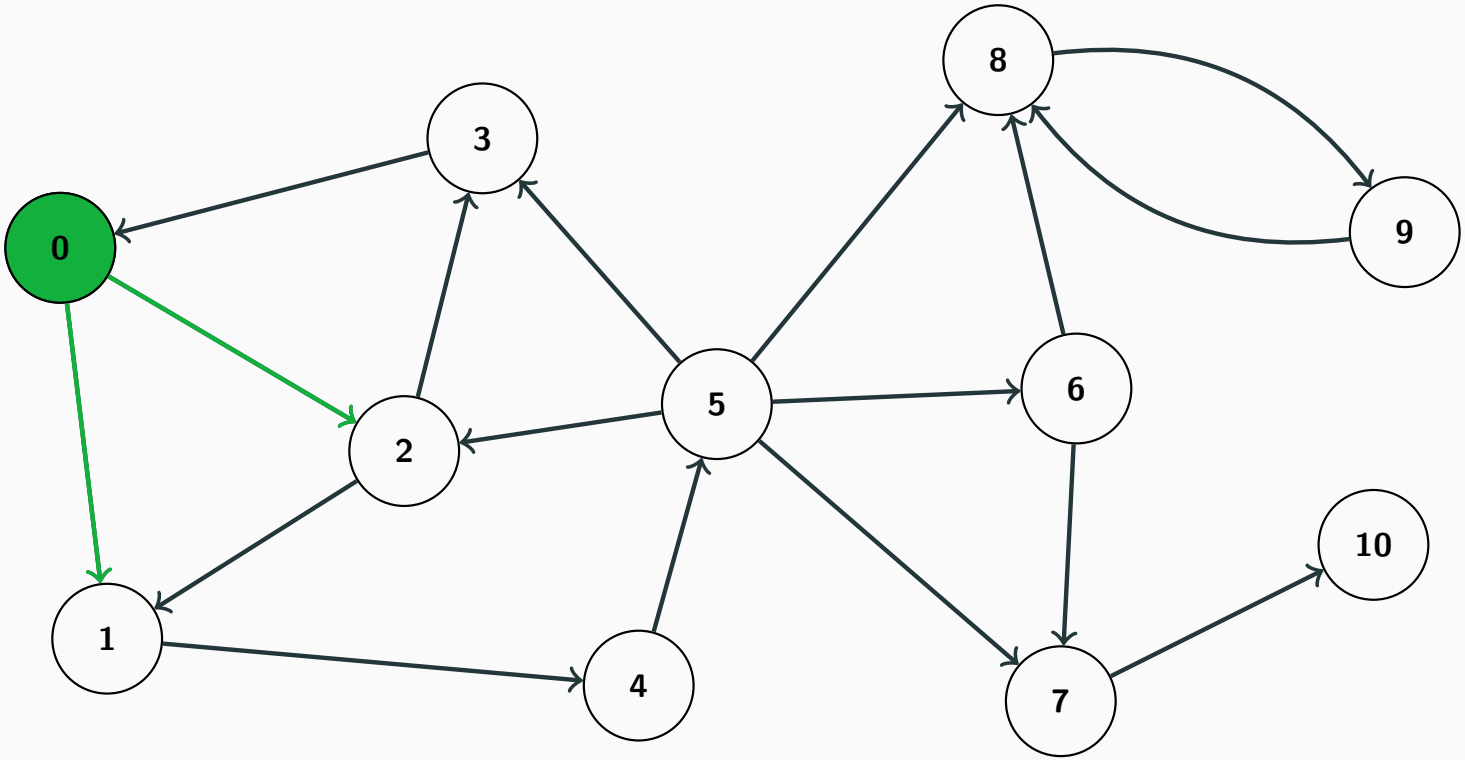
Breadth-first search



Queue: 0

	0	1	2	3	4	5	6	7	8	9	10
marked	1	0	0	0	0	0	0	0	0	0	0

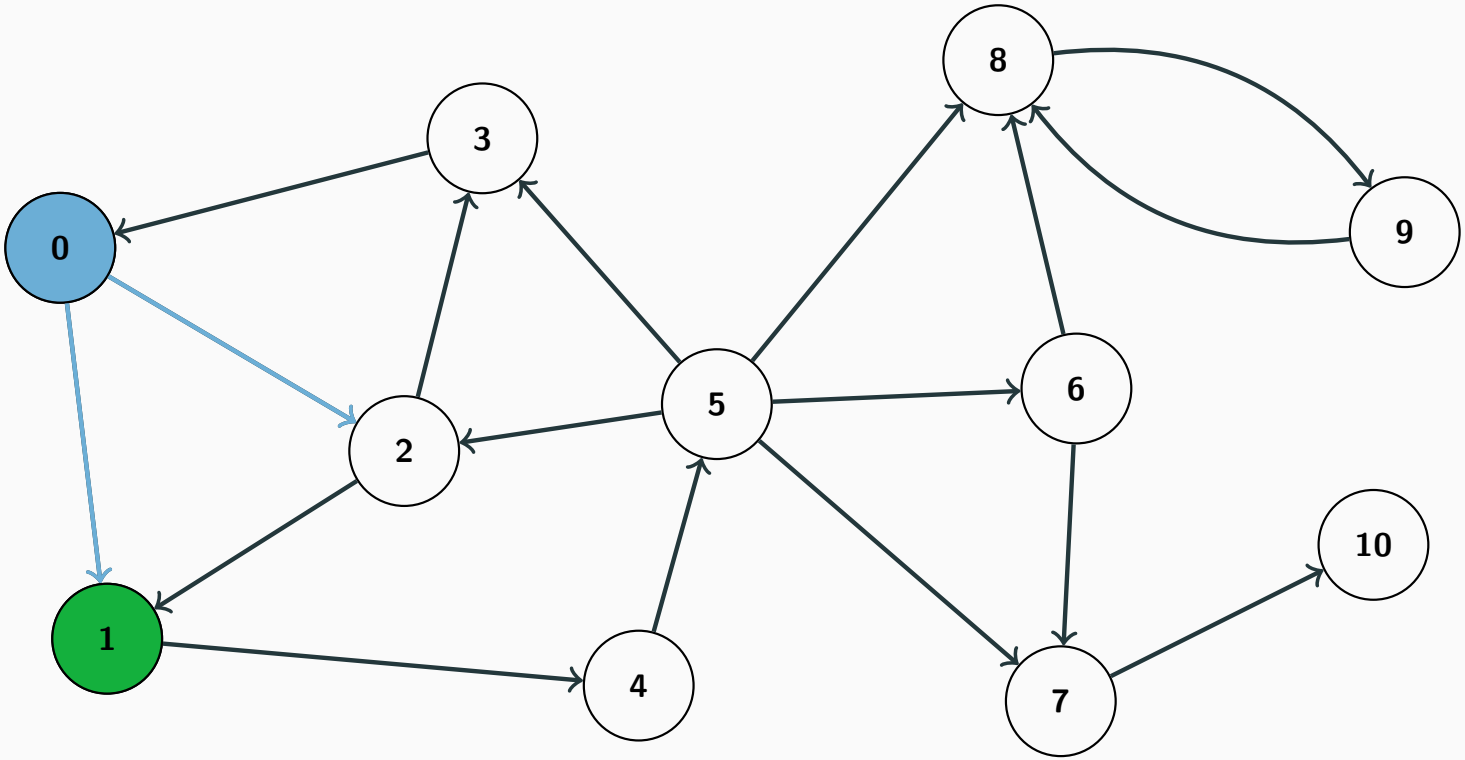
Breadth-first search



Queue: 0 1 2

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	0	0	0	0	0	0	0

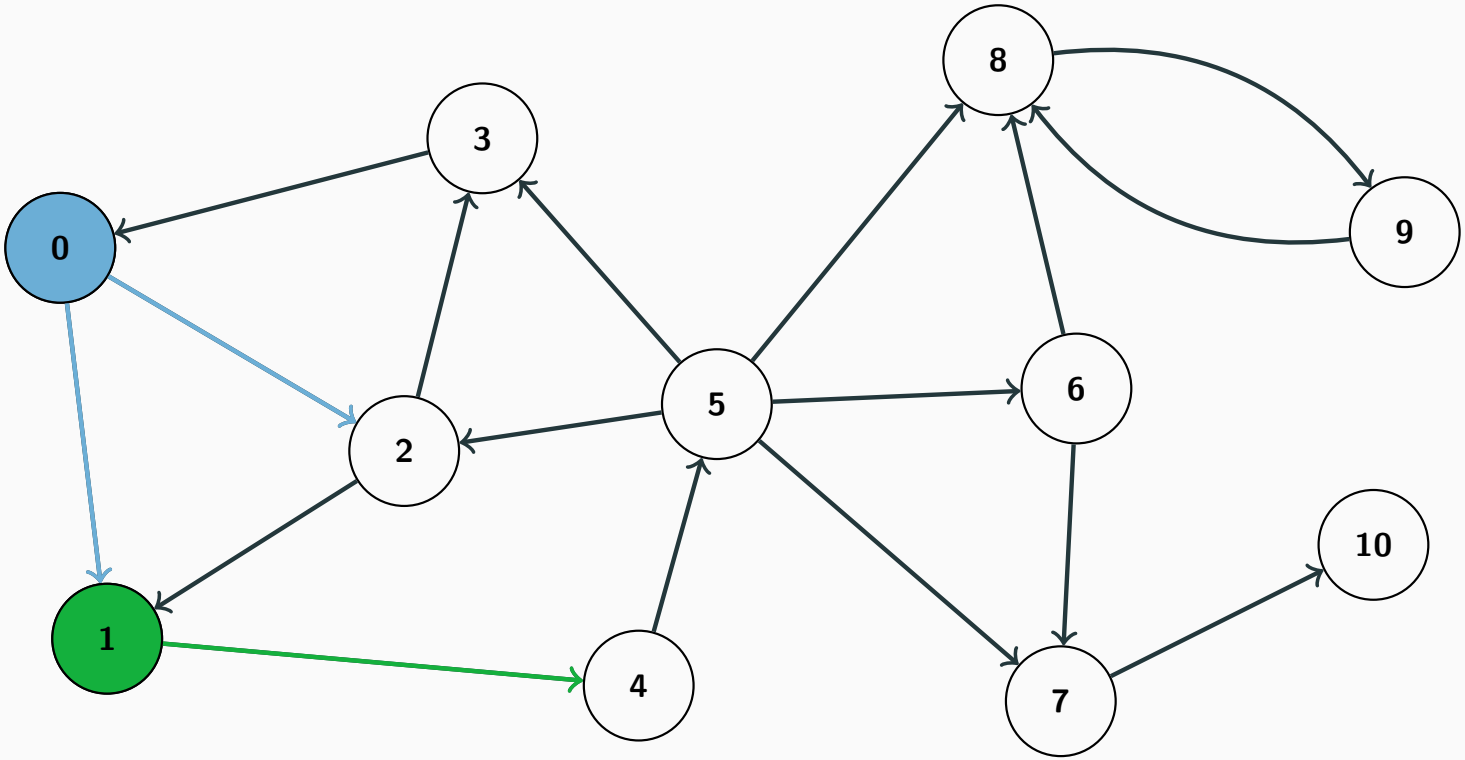
Breadth-first search



Queue: 1 2

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	0	0	0	0	0	0	0

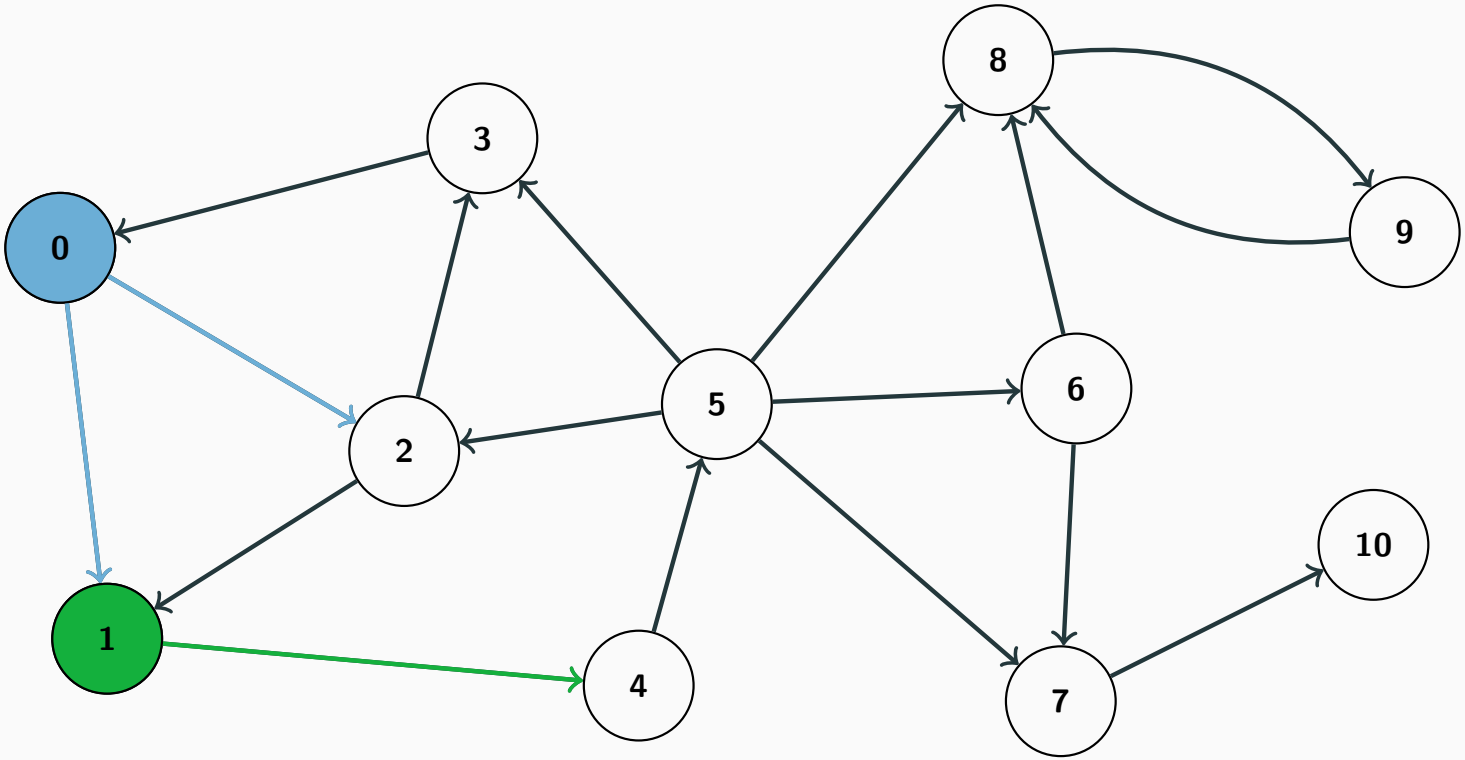
Breadth-first search



Queue: 1 2

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	0	0	0	0	0	0	0

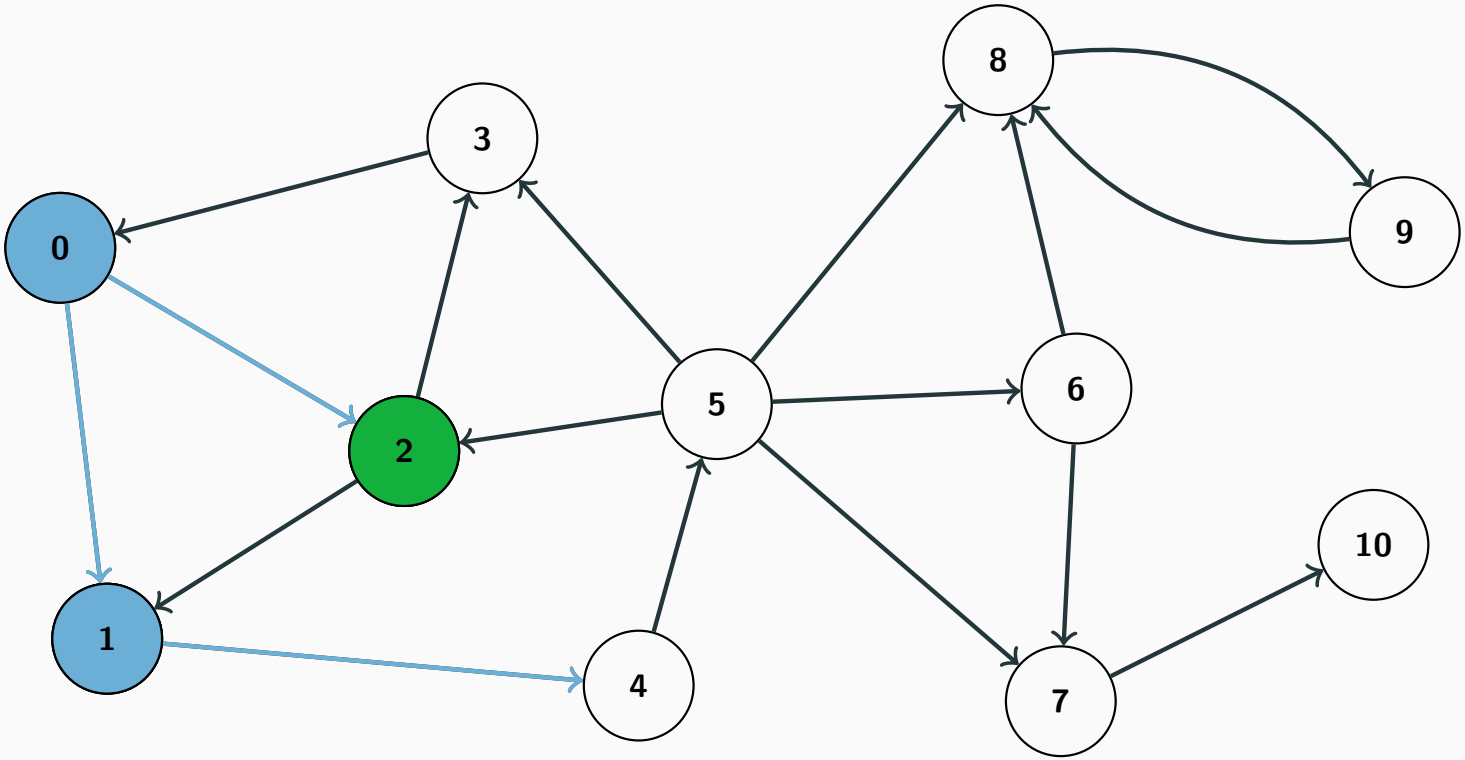
Breadth-first search



Queue: 1 2 4

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	1	0	0	0	0	0	0

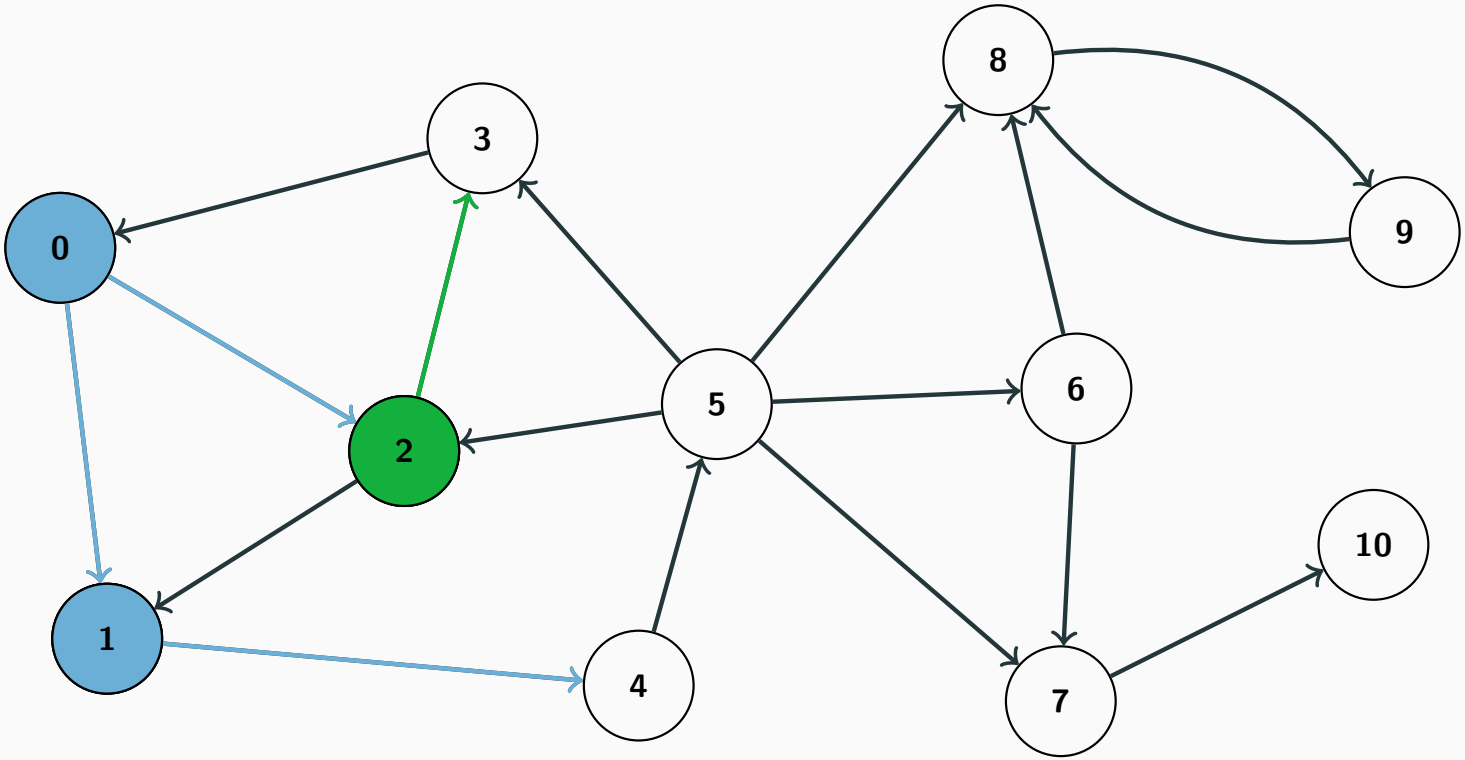
Breadth-first search



Queue: 2 4

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	1	0	0	0	0	0	0

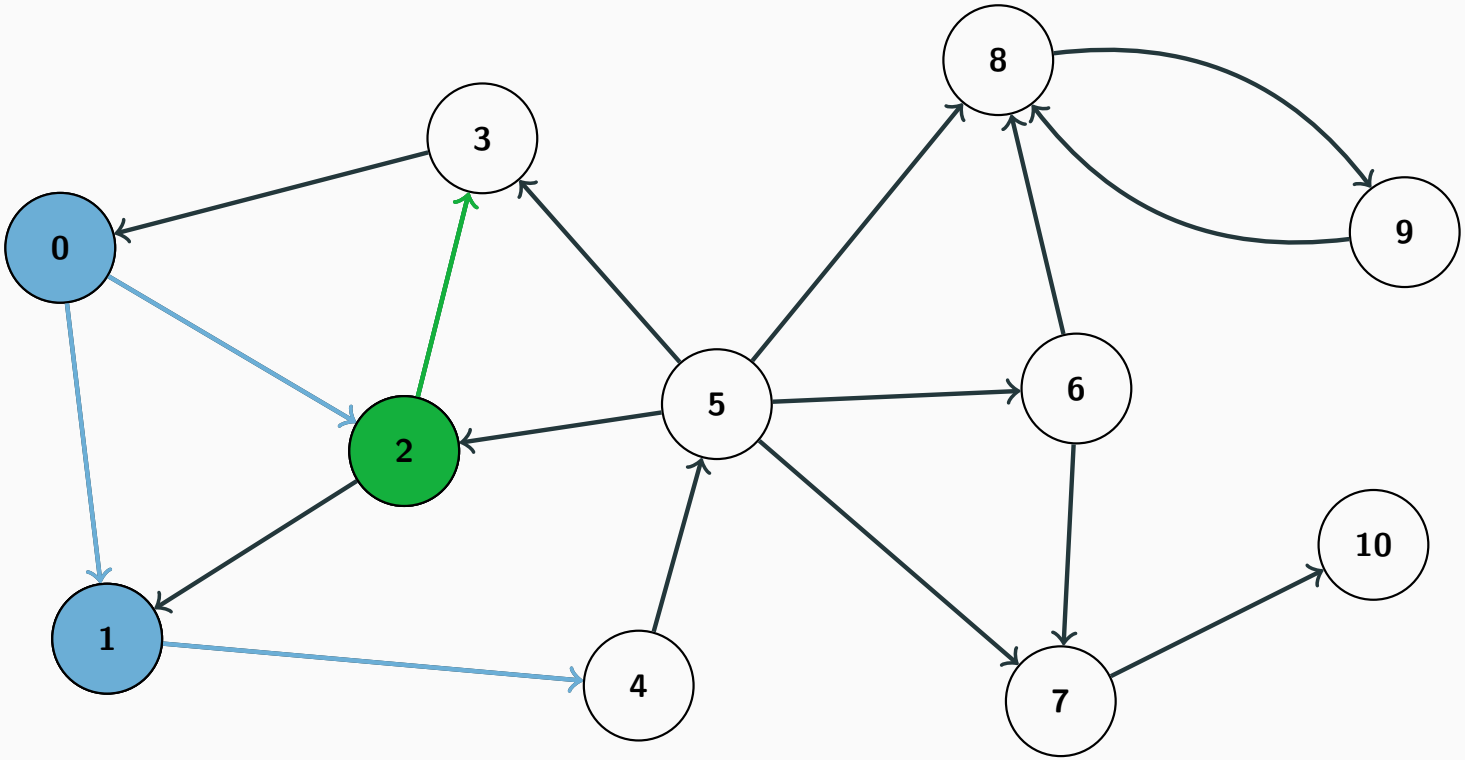
Breadth-first search



Queue: 2 4

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	1	0	0	0	0	0	0

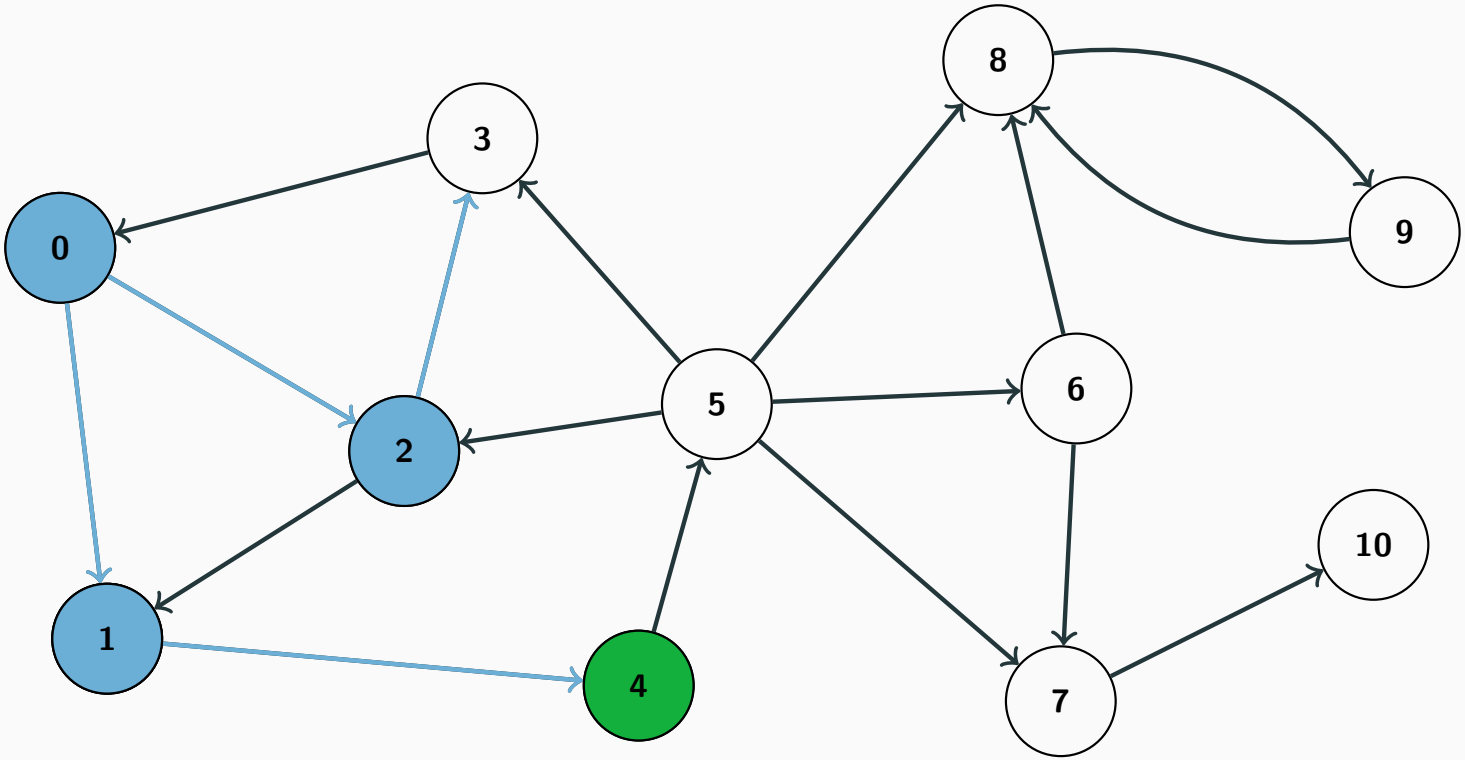
Breadth-first search



Queue: 2 4 3

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	0	0	0	0	0	0

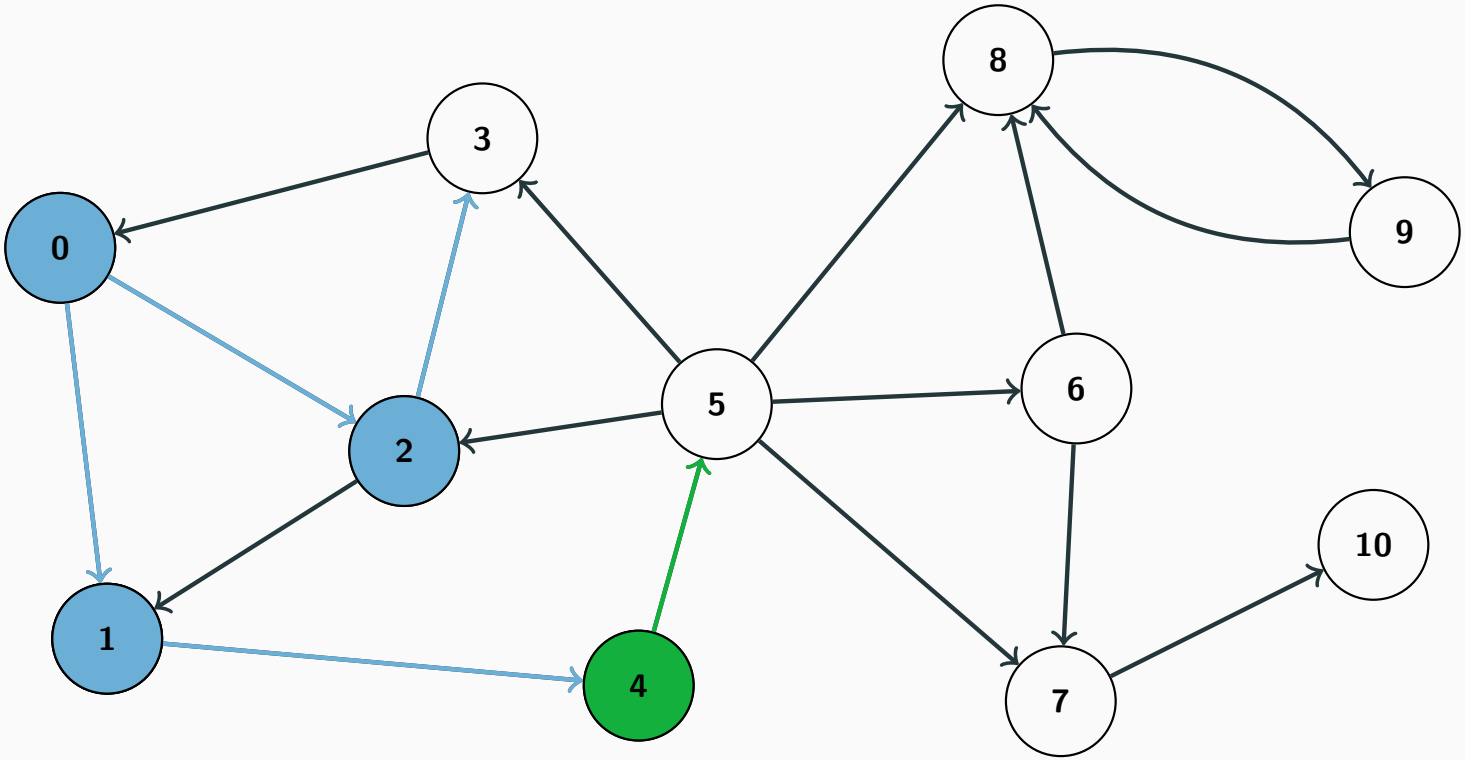
Breadth-first search



Queue: 4 3

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	0	0	0	0	0	0

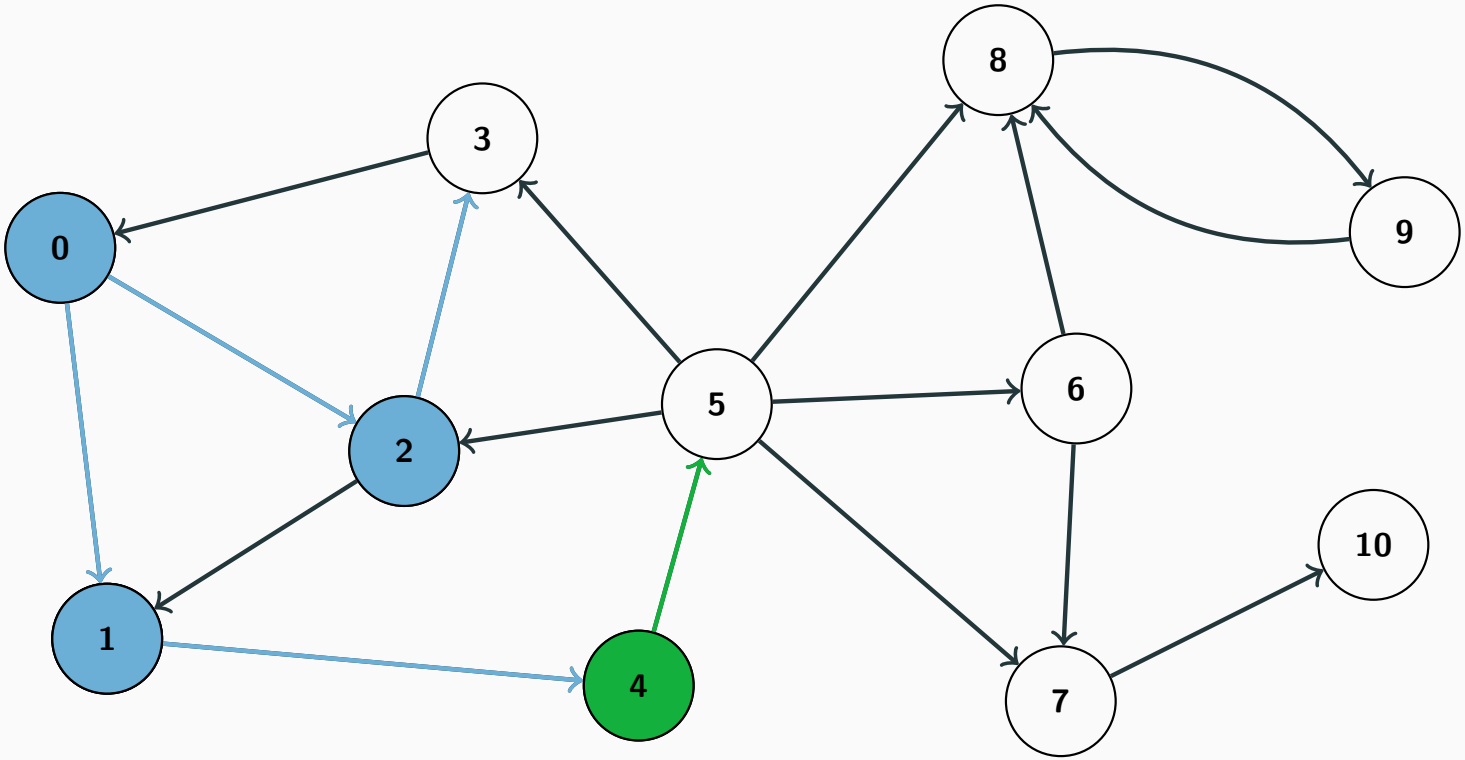
Breadth-first search



Queue: 4 3

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	0	0	0	0	0	0

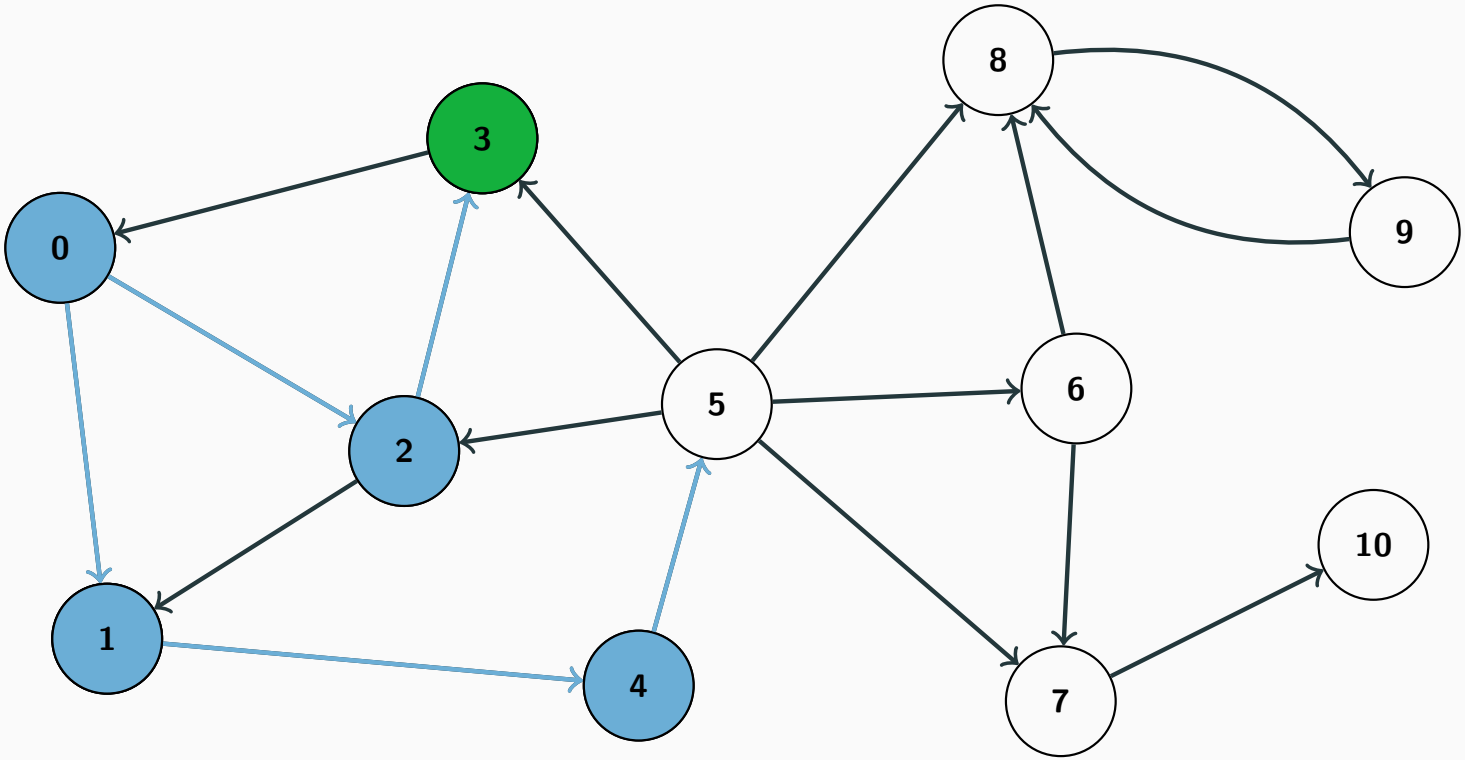
Breadth-first search



Queue: 4 3 5

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

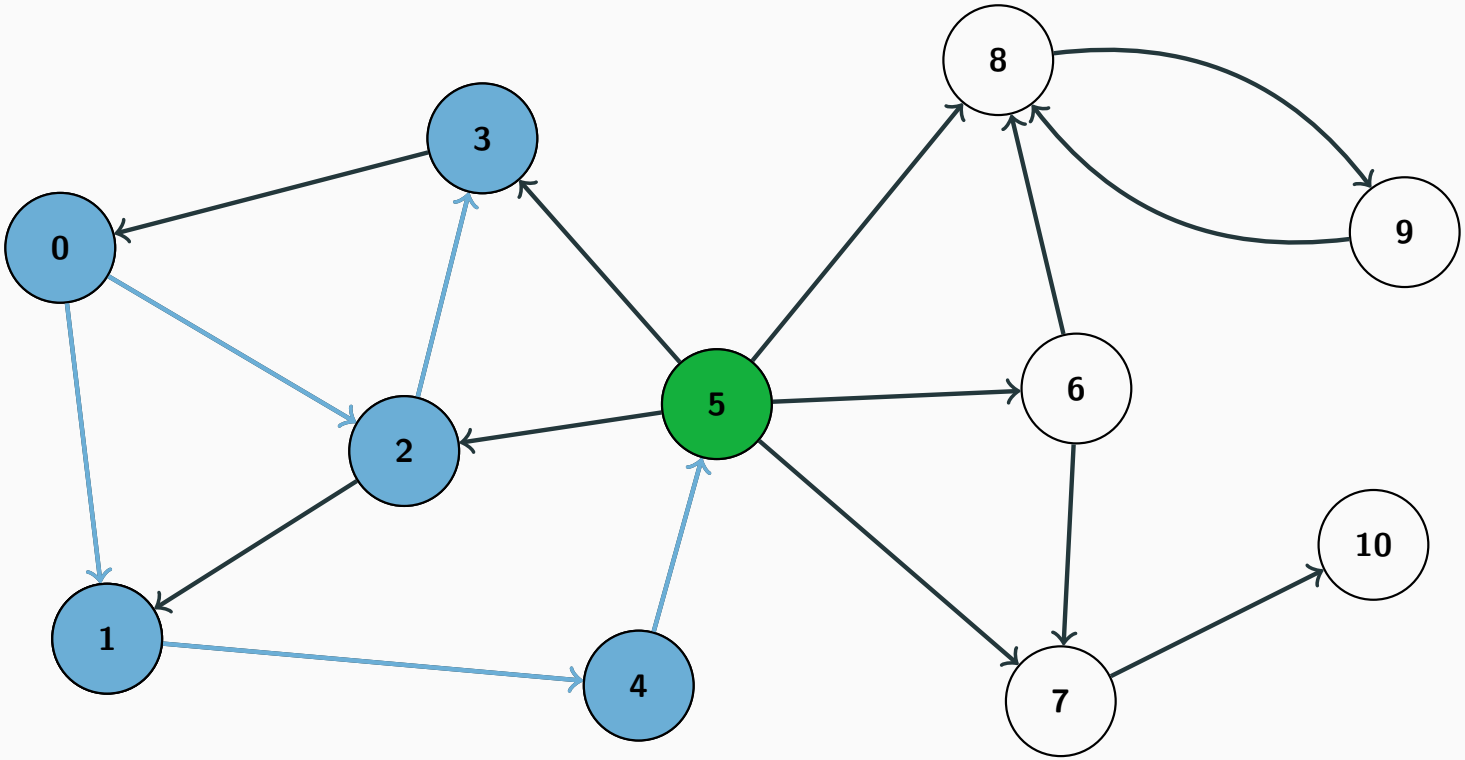
Breadth-first search



Queue: 3 5

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

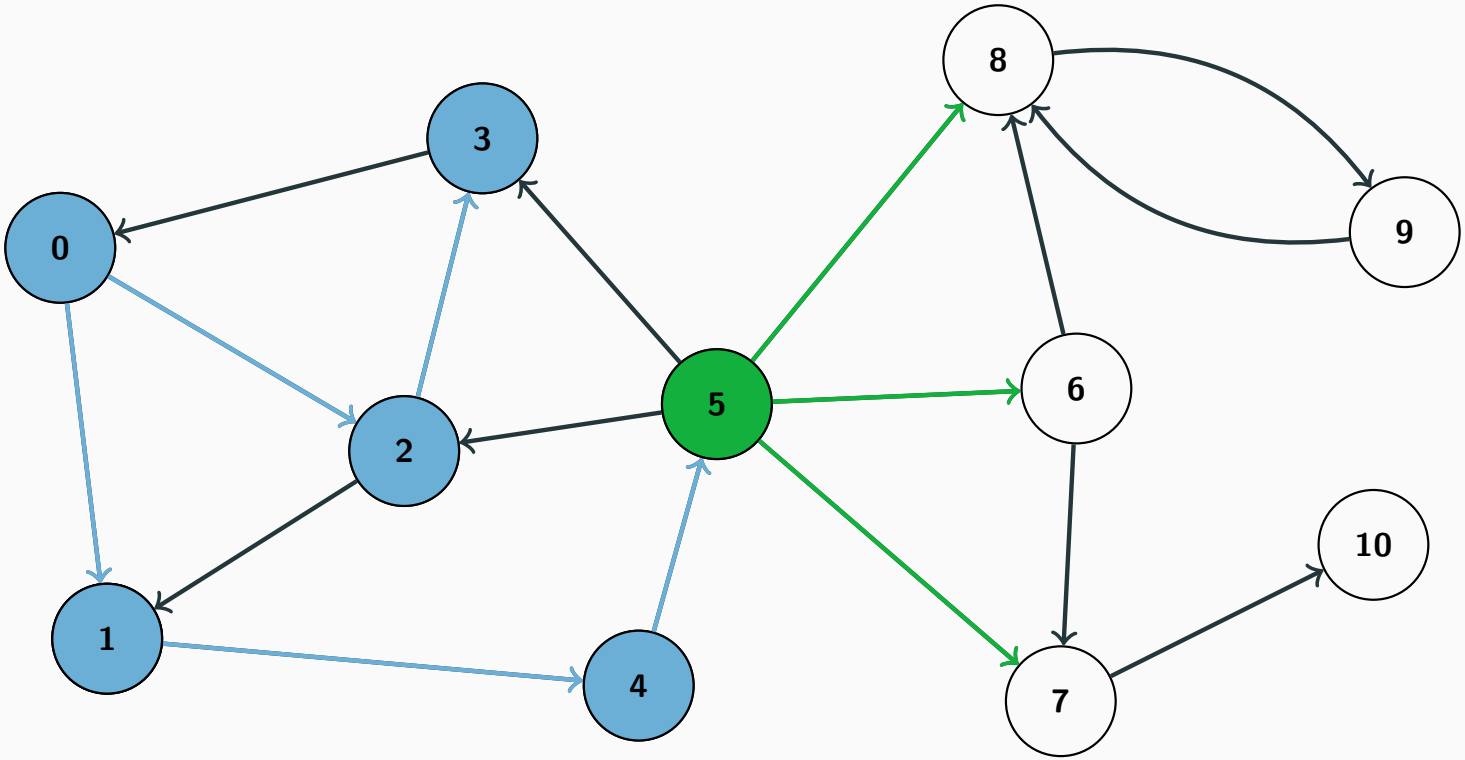
Breadth-first search



Queue: 5

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

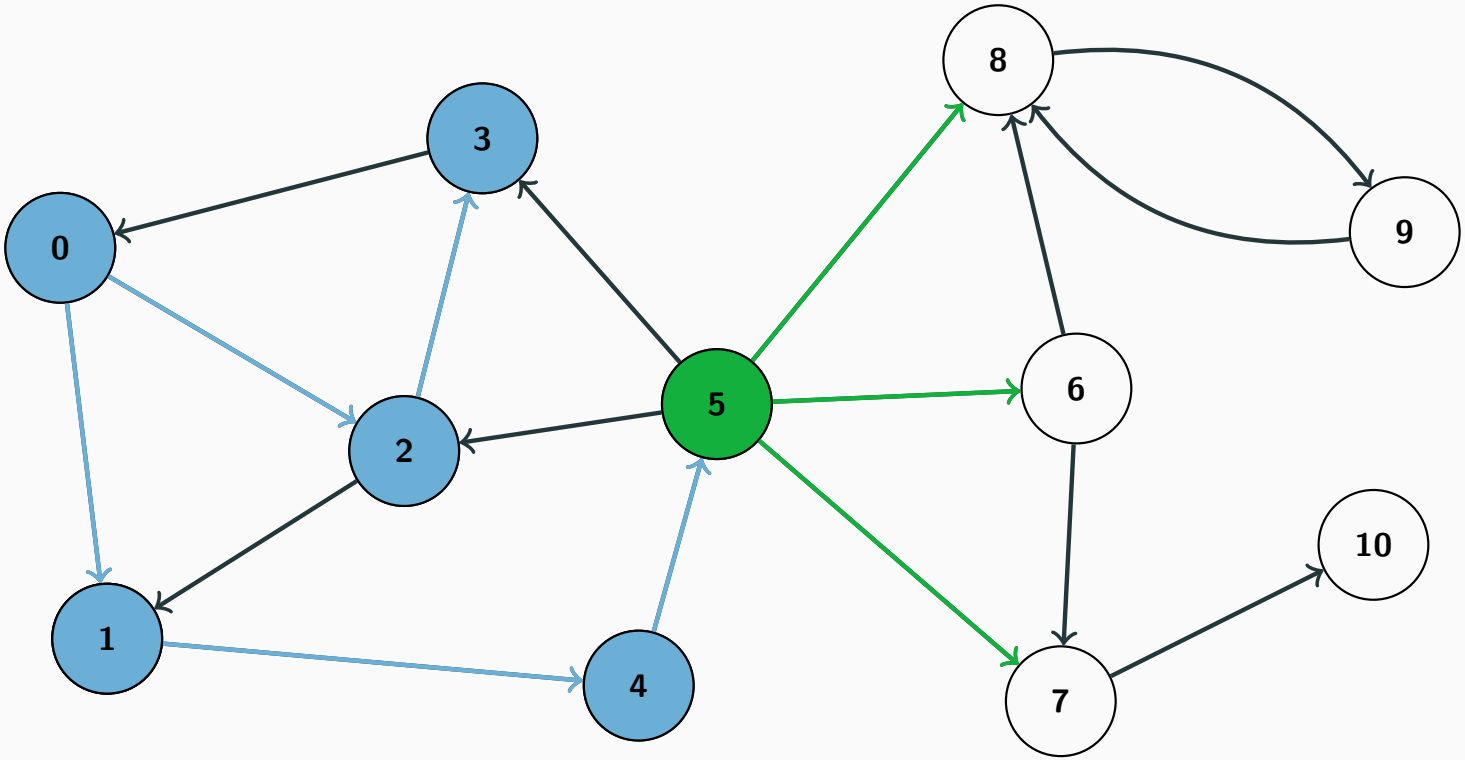
Breadth-first search



Queue: 5

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

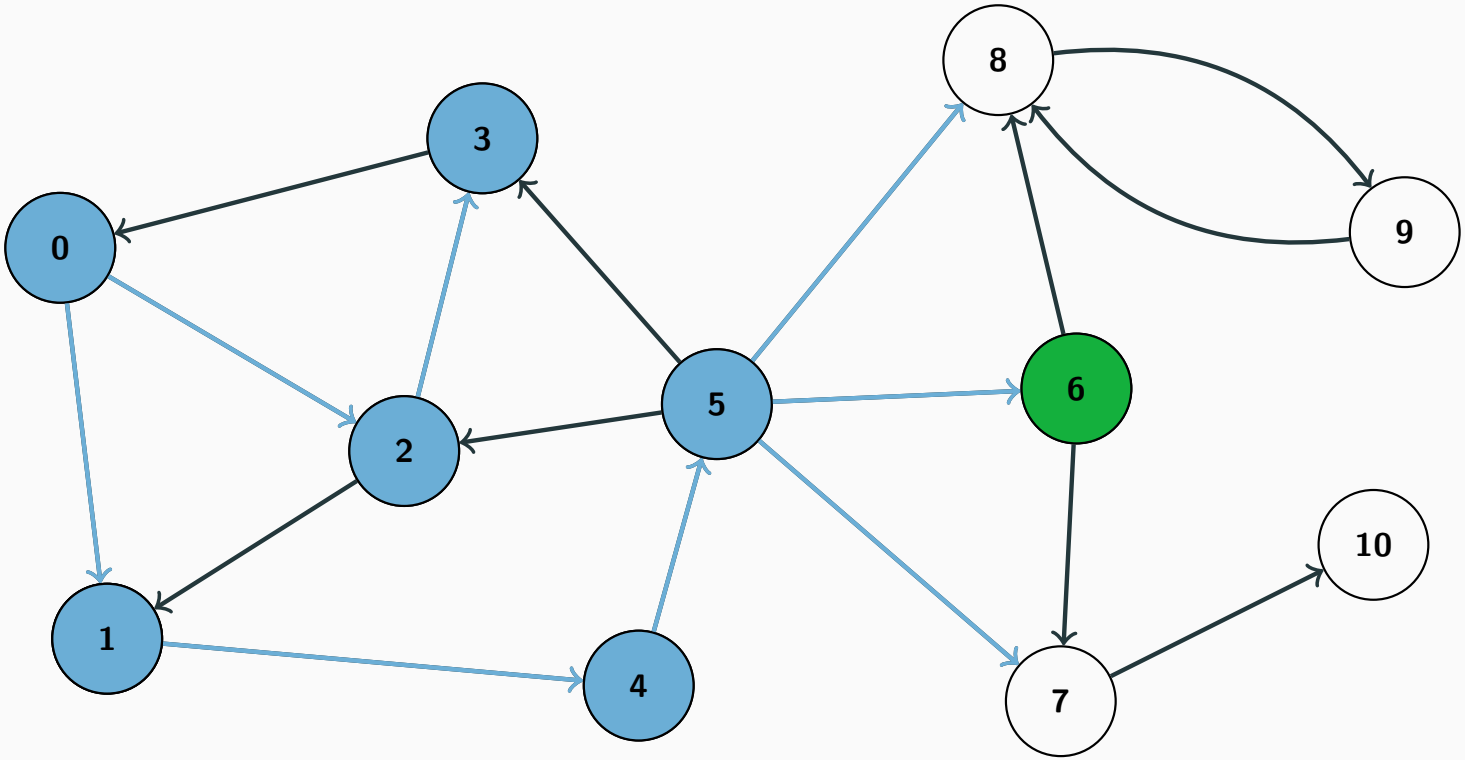
Breadth-first search



Queue: 5 6 7 8

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

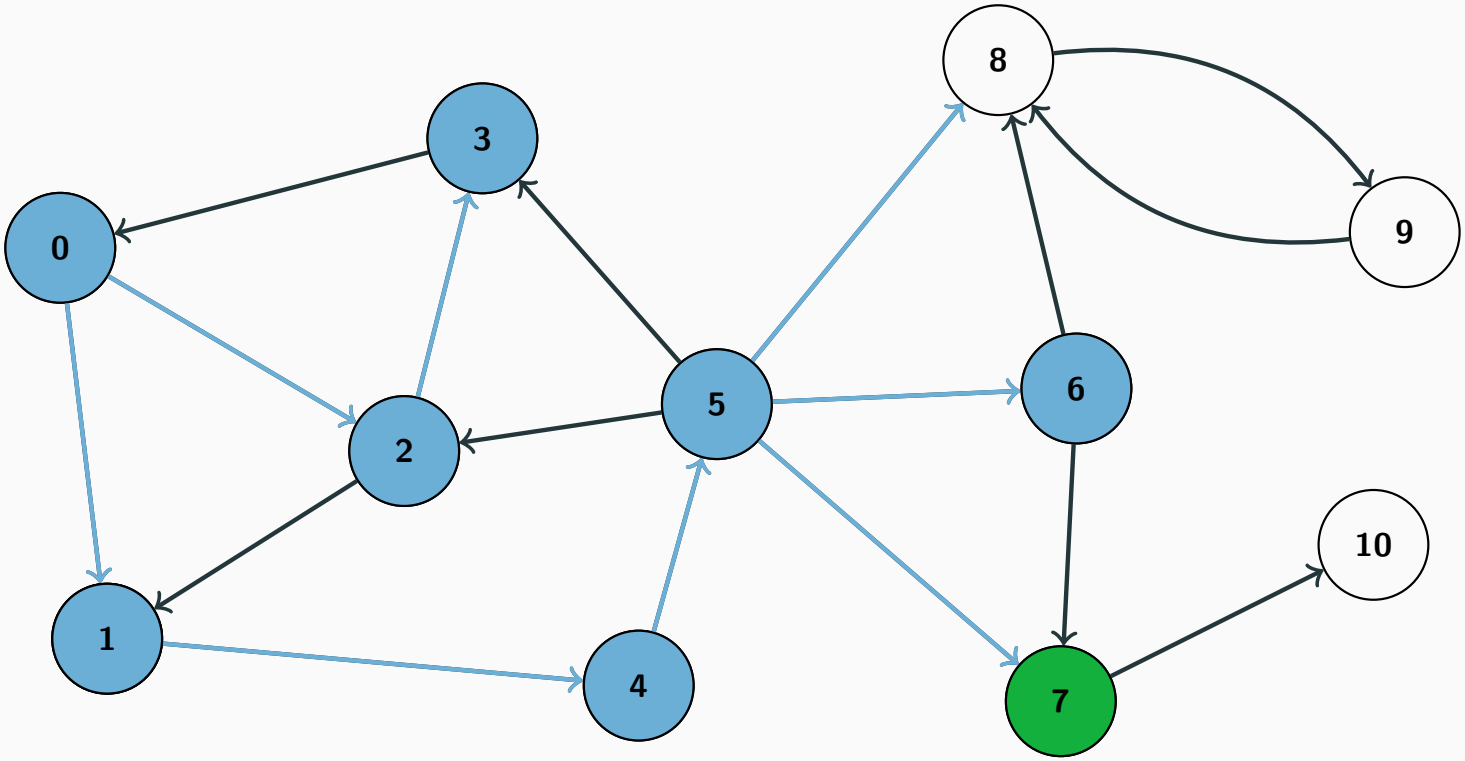
Breadth-first search



Queue: 6 7 8

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

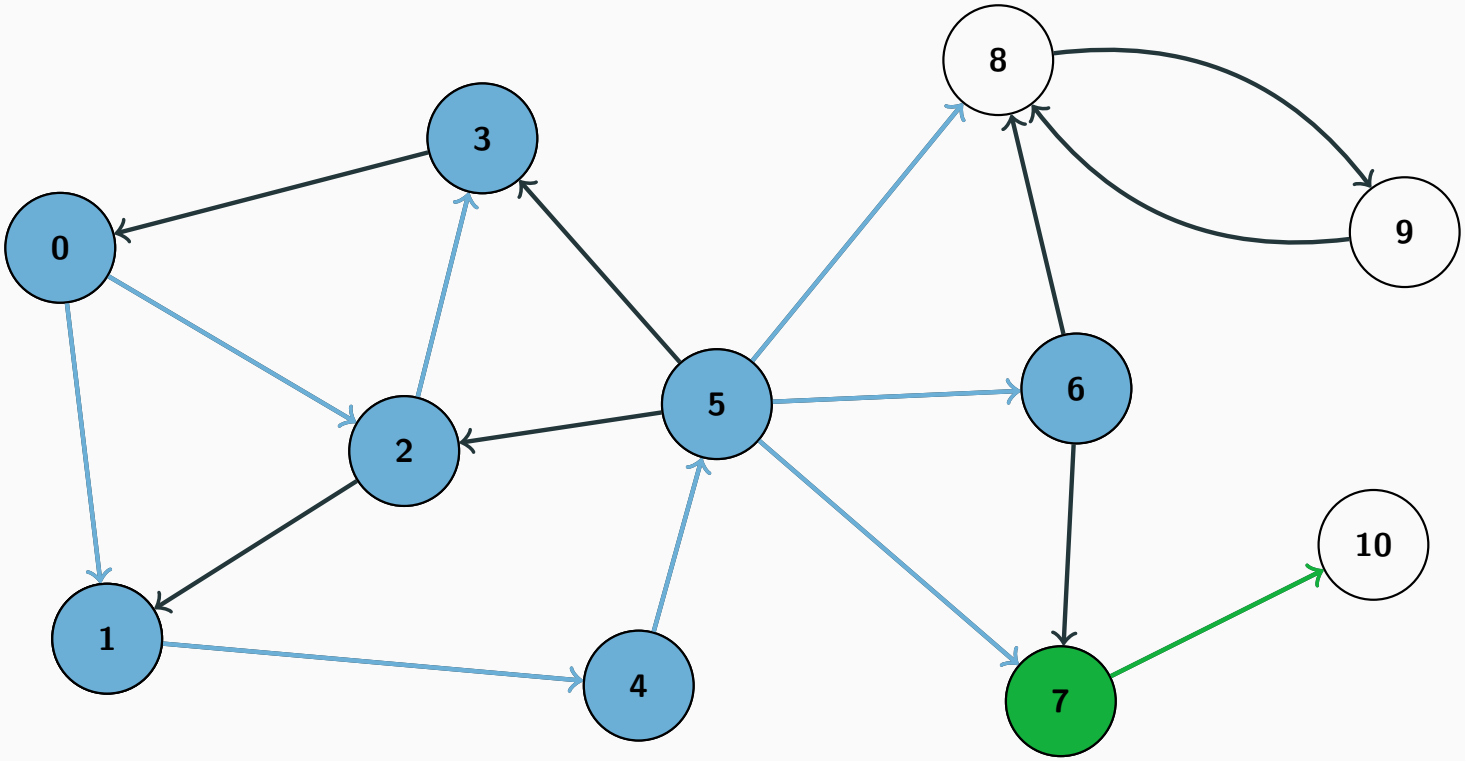
Breadth-first search



Queue: 7 8

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

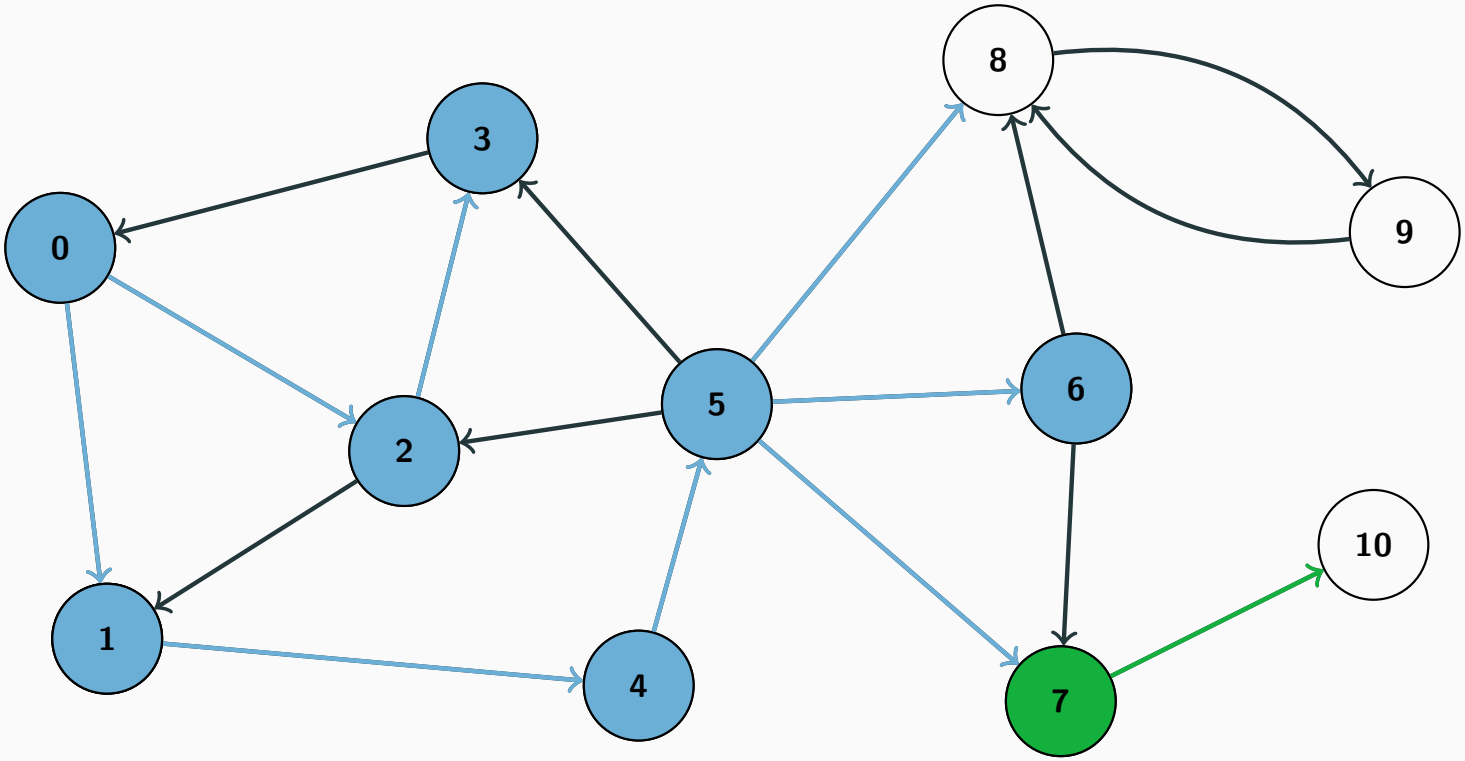
Breadth-first search



Queue: 7 8

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

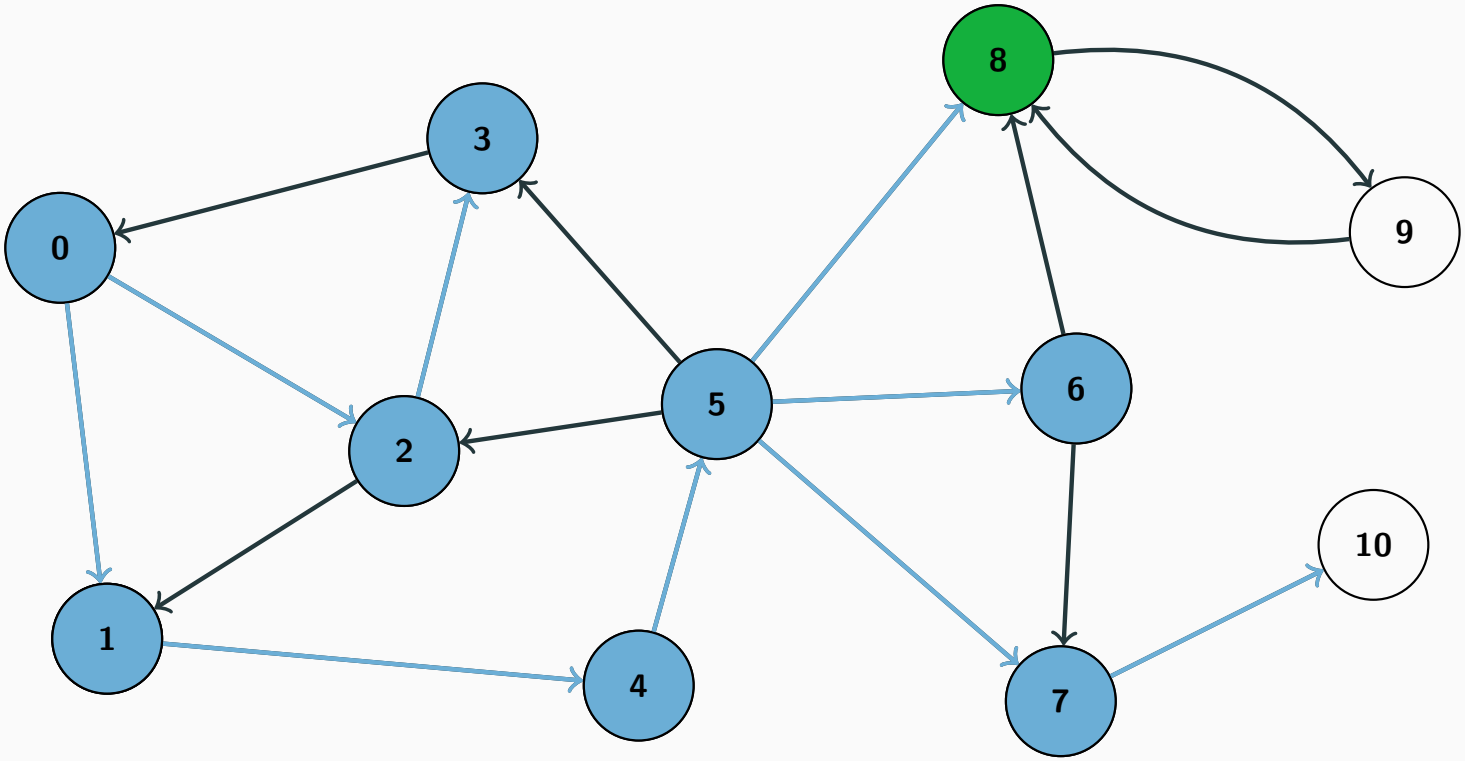
Breadth-first search



Queue: 7 8 10

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	1

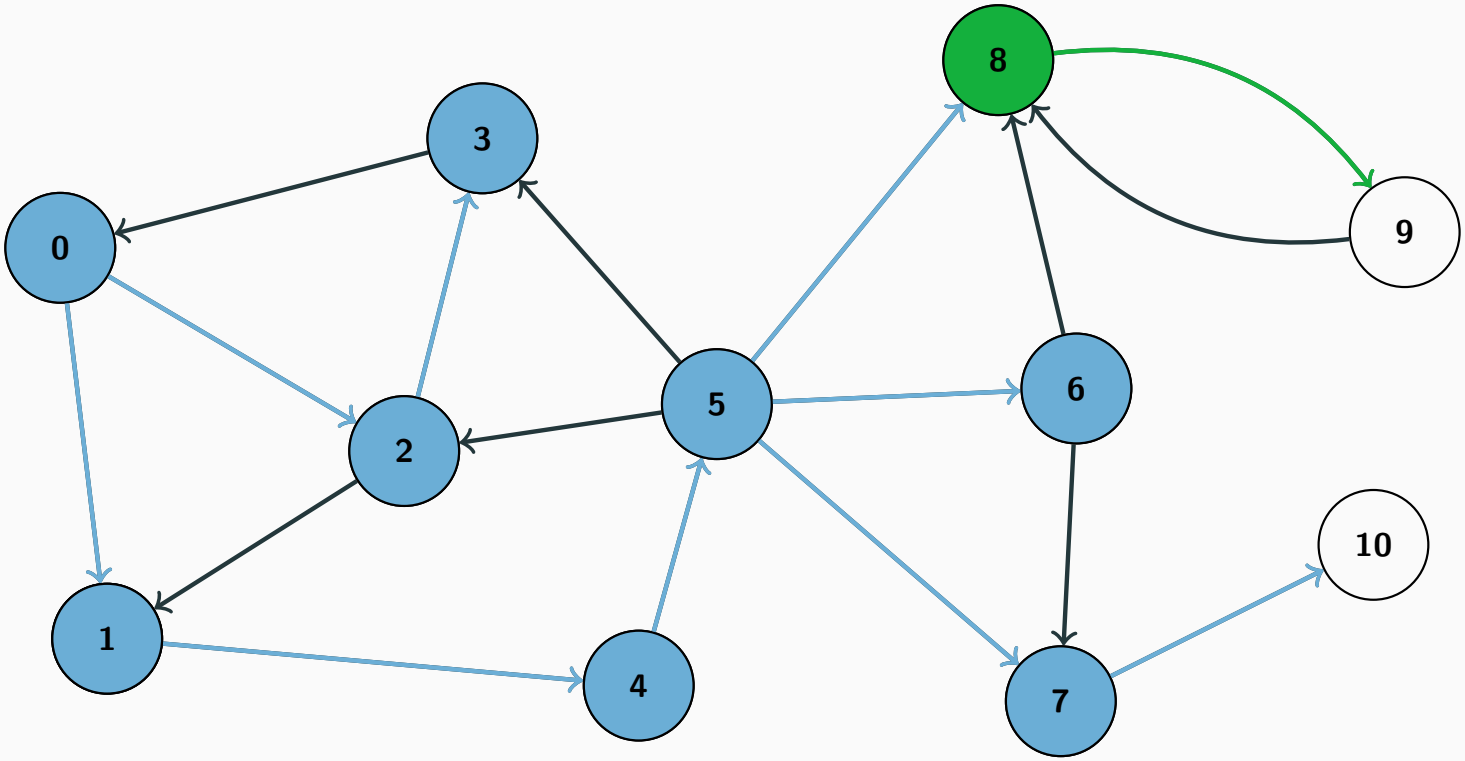
Breadth-first search



Queue: 8 10

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	1

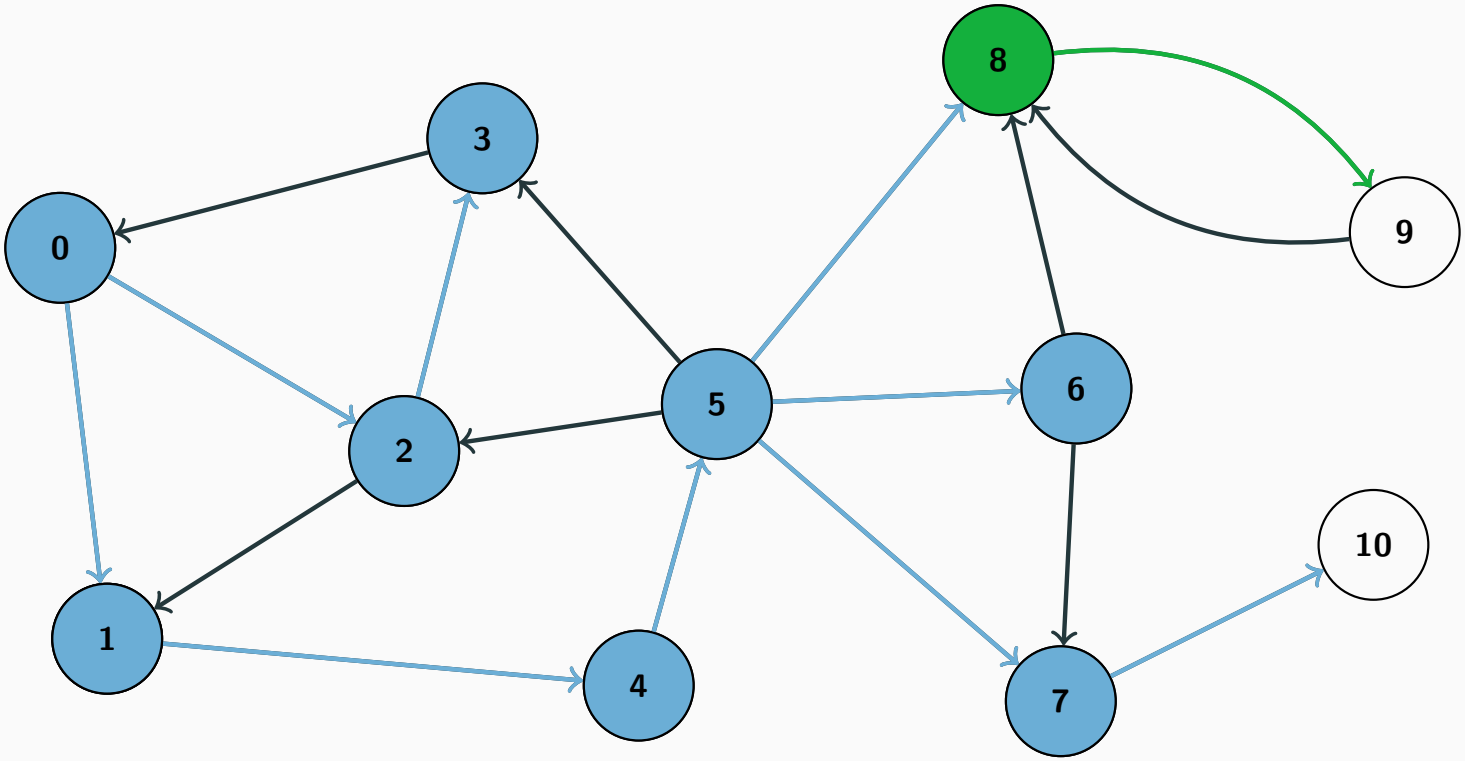
Breadth-first search



Queue: 8 10

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	1

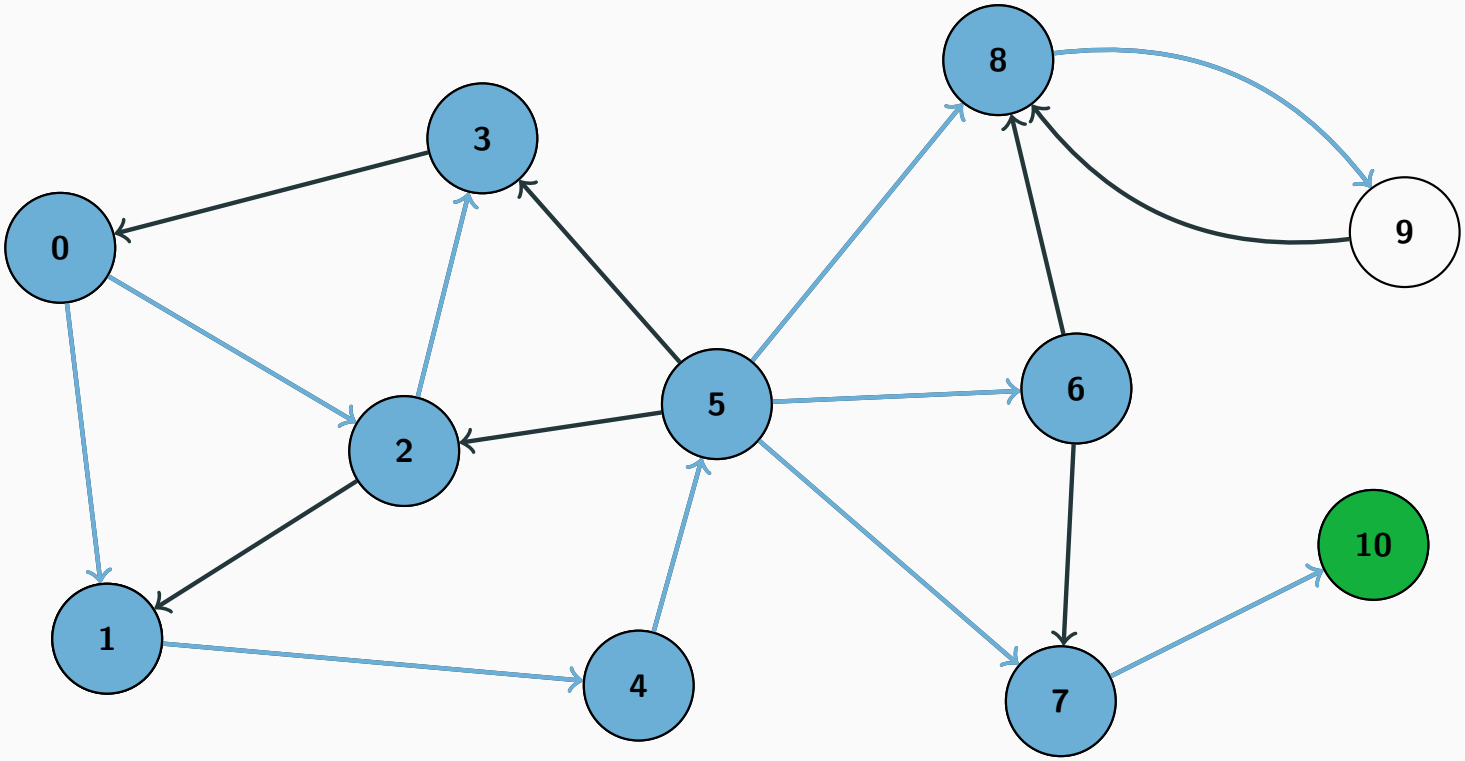
Breadth-first search



Queue: 8 10 9

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

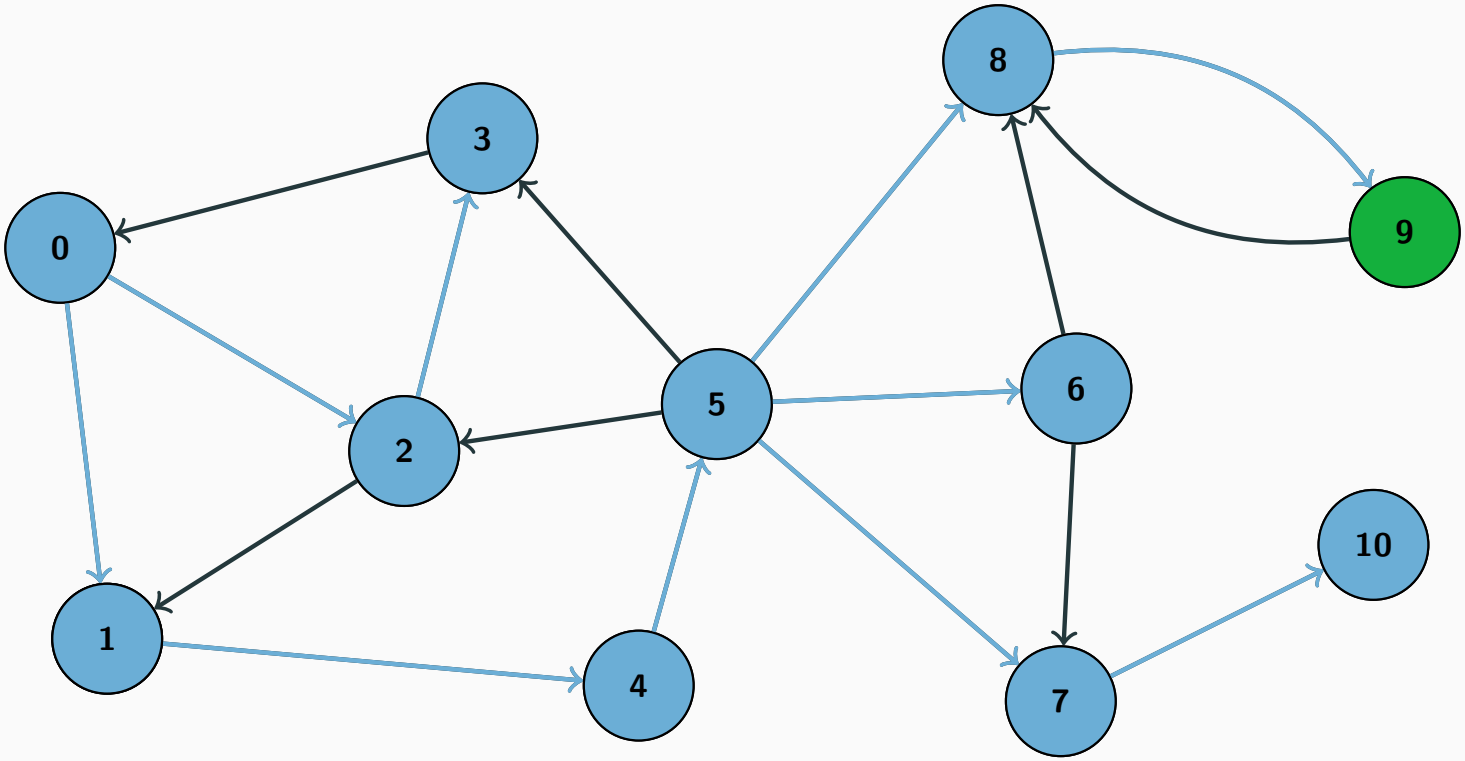
Breadth-first search



Queue: 10 9

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

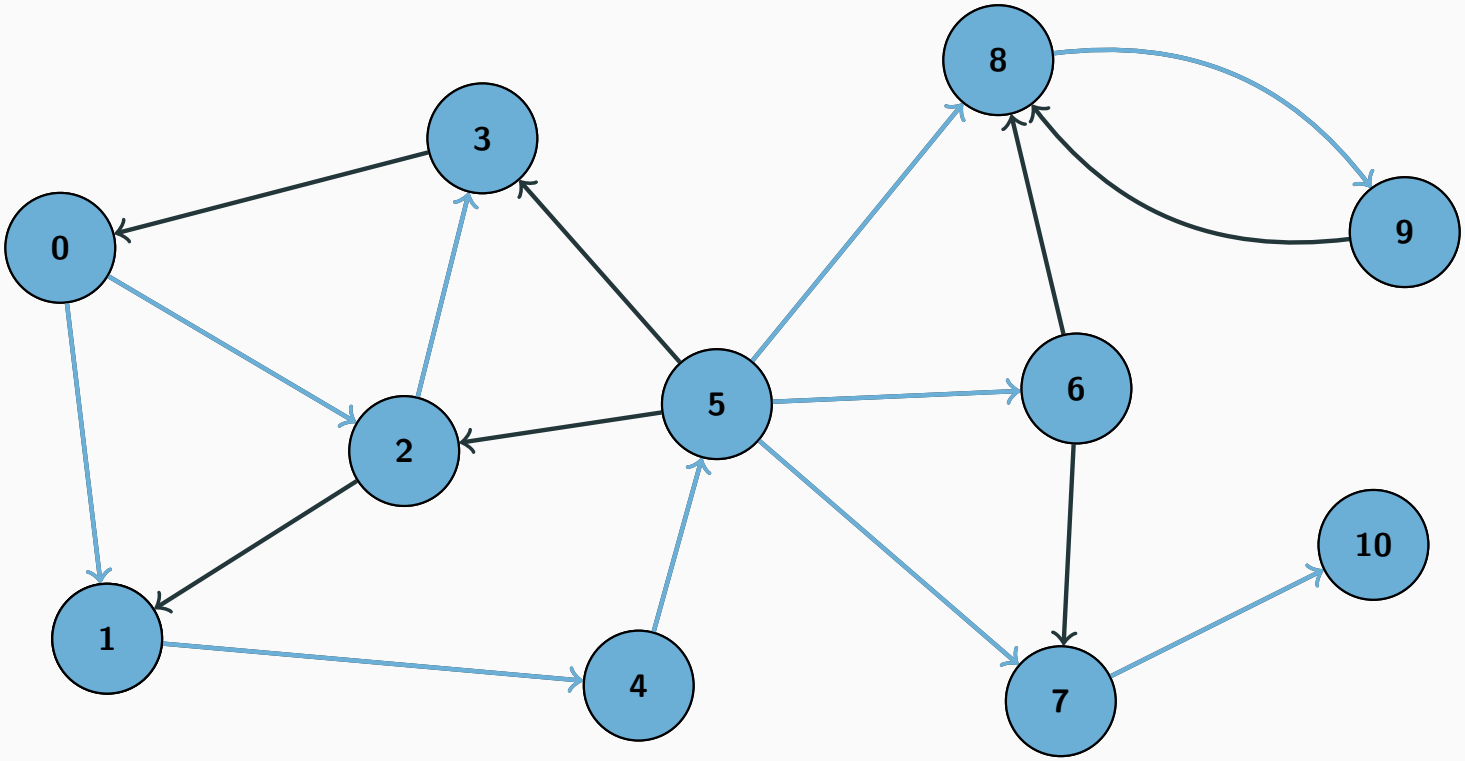
Breadth-first search



Queue: 9

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

Breadth-first search



Queue:

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

Breadth-first search

```
vector<int> adj[1000];
vector<bool> visited(1000, false);

queue<int> Q;
Q.push(start);
visited[start] = true;

while (!Q.empty()) {
    int u = Q.front(); Q.pop();

    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (!visited[v]) {
            Q.push(v);
            visited[v] = true;
        }
    }
}
```

Shortest path in unweighted graphs

- We have an unweighted graph, and want to find the shortest path from A to B
- That is, we want to find a path from A to B with the minimum number of edges

- Breadth-first search goes through the vertices in increasing order of distance from the start vertex
- Just do a single breadth-first search from A, until we find B
- Or let the search continue through the whole graph, and then we have the shortest paths from A to all other vertices
- Shortest path from A to all other vertices: $O(n + m)$

Shortest path in unweighted graphs

```
vector<int> adj[1000];
vector<int> dist(1000, -1);

queue<int> Q;
Q.push(A);
dist[A] = 0;

while (!Q.empty()) {
    int u = Q.front(); Q.pop();

    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (dist[v] == -1) {
            Q.push(v);
            dist[v] = 1 + dist[u];
        }
    }
}

printf("%d\n", dist[B]);
```

0/1 weights

- This does not work for weighted graphs in general, but what if the weights are only 0 or 1?
- If we have a double ended queue, this can be made to work
- For weights 1 we push to the back as usual
- But for "freebie" weights 0 we can push to the front of the queue, letting them cut ahead of the others because the edge is "for free"
- This version is called 0/1 BFS
- The idea can be extended to weights in $0, 1, \ldots, k$ and is in that case called Dial's algorithm

Back to topsort

- But how can we use this idea to do a topsort?
- Well, we can always just try to greedily take a value to go first, one that doesn't have anything pointing into it
- If the graph is acyclic, this will finish taking all the vertices
- This is known as Kahn's algorithm

Kahn

```
def kahn(graph):
        indegree = [0 for _ in range(len(graph))]
        for v in range(len(graph)):
                for w in graph[v]:
                        indegree[w] += 1
        q = Queue()
        for v in range(len(graph)):
                if indegree[v] == 0:
                        q.push(v)
        result = []
        while len(q) > 0:
                current = q.pop()
                result.append(current)
                for neighbour in graph[current]:
                        indegree[neighbour] -= 1
                        if indegree[neighbour] == 0:
                                q.push(neighbour)
        return result
```

Diameter of graphs

- We call the longest shortest distance in a graph its diameter
- We could find that by running BFS from every vertex and then collecting the answer
- What if the graph is a tree, can we do better then?
- A tree is a connected graph with no cycles, or equivalently a connected graph on n vertices with $n - 1$ edges

Diameter of graphs

- It can be shown that in a tree every path between two vertices is unique
- Furthermore we can show the existence of a centroid (or two in some cases), a vertex who is on every longest path in the graph
- The centroid is then also at most $\lceil d/2 \rceil$ distance from every other vertex where d is the diameter

Diameter of graphs

- Thus we just start at some vertex v, find the vertex furthest away, call it w
- Then we find the vertex furthest away from w, call it u
- Then the diameter is simply the distance of u from w
- And the centroid will be the middle (or middle two) vertices on the path from u to w