Learning Objectives

Learners will be able to...

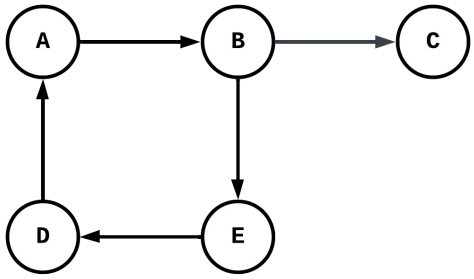
- **Define a cycle in a graph**
- **Detect cycles in directed and undirected graphs**

Graph Cycles

Cycle Detection in Graphs

Cycle detection in graphs is a critical operation in many applications, including network analysis, circuit testing, and dependency resolution. A **cycle** in a graph is a path of edges and vertices wherein a vertex is reachable from itself. Detecting cycles is important to prevent infinite loops in certain algorithms and to ensure data structures like trees remain acyclic.

Consider the following directed graph:



In this graph, there is a cycle involving the vertices A, B, E, and D:

```
A → B → E → D → A.
```

Cycle Detection Concepts

- **Cycle in Directed Graphs**: A directed graph has a cycle if a vertex can be visited more than once as we traverse the graph following the edge directions.
- **Cycle in Undirected Graphs**: An undirected graph has a cycle if there is a path of unique vertices that starts and ends at the same vertex.
- **Detection Techniques**:
    - **Depth-First Search (DFS)**: DFS can be used to detect cycles by tracking visited vertices. If a vertex is encountered more than once, a cycle is detected.
    - **Disjoint Set (Union-Find)**: This is another method for cycle detection in undirected graphs. It checks whether an edge creates a connection between two vertices that are already part of the same set, indicating a cycle.

Key Considerations for Cycle Detection

- **Back Edges**: In DFS, a back edge (an edge that points to an ancestor in a DFS tree) indicates a cycle in directed graphs.
- **Parent Tracking**: In undirected graphs, while using DFS, we need to track the parent of each vertex to distinguish between a genuine back edge and an edge to the parent.
- **Graph Representation**: The choice of graph representation (adjacency list or matrix) can affect the implementation and efficiency of the cycle detection algorithm.
- **Complexity**: The time complexity for cycle detection using DFS is generally $O(V + E)$ for both directed and undirected graphs, where V is the number of vertices and E is the number of edges.

Potential Applications

- **Scheduling and planning algorithms:** cycle detection is crucial as cycles might indicate impossible tasks or dependencies.
- **Computer networks:** cycle detection can help in routing optimization and deadlock prevention.
- **Database management systems:** detecting cycles is essential to avoid deadlock situations where transactions wait indefinitely for each other.
- **Dependency resolution:** cycle detection helps to identify and resolve circular dependencies in software systems.

Cycles in Directed Graphs

Cycle Detection in Directed Graphs

Cycle detection is a fundamental algorithm in graph theory. We will implement cycle detection for a directed graph using depth-first search (DFS). Remember, DFS is a recursive algorithm, so there will be backtracking to previous vertices in the graph. That means we will visit the same vertex more than once. We need to distinguish between visiting the same vertex due to backtracking (no cycle) and visiting the same vertex due to a cycle.

The key idea is to keep track of the already visited vertices (`visited` in the algorithm below) and the vertices being actively explored (`recursionStack` in the algorithm below). The difference between these two is important. If we encounter a vertex in `visited` for a second time (backtracking), no cycle is present. However, if you encounter a vertex in `recursionStack` (the current subtree) for a second time, a cycle is present.

Implementing Cycle Detection

Start by defining a basic `DirectedGraph` class with an adjacency list representation. Vertices in the graph are identified with an integer. Create methods to add directed edges and vertices to the graph.

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <unordered_set>

class DirectedGraph {
private:
    std::unordered_map<int, std::vector<int>> adjList;

public:
    DirectedGraph() {}

    void addVertex(int vertex) {
        adjList[vertex];  // Automatically initializes an empty
        vector if vertex doesn't exist
    }

    void addEdge(int vertex1, int vertex2) {
        adjList[vertex1].push_back(vertex2);
    }
};
```

The `hasCycle` method sets everything up for the recursive method that does the actual cycle detection. This public method does not take any values and returns a boolean value. Start by creating empty sets for the `visited` vertices and the vertices in the `recursionStack`. Iterate over all of the vertices in the graph. For each vertex, ask if the recursive helper method detects a cycle for the current vertex. If so, return true. If the loop ends without finding a cycle, return false.

```
    bool hasCycle() {
        std::unordered_set<int> visited;
        std::unordered_set<int> recursionStack;

        for (const auto& pair : adjList) {
            int vertex = pair.first;
            if (hasCycleHelper(vertex, visited, recursionStack))
        {
                return true;
            }
        }
        return false;
    }
```

Create the private, recursive method `hasCycleHelper`. This is a helper method that determines if a cycle exists in a given subtree. The method takes an integer representing the `current` vertex, a set of `visited` vertices, and a set of vertices in `recursionStack`. Since this is a recursive method, start by identifying the base cases. If the current vertex is in the recursion stack, return `true` as a cycle exists. If the current vertex is in the set of visited vertices, return `false` as no cycle is present.

```
private:
    // Helper function to detect cycle
    bool hasCycleHelper(int current, std::unordered_set<int>&
        visited,
        std::unordered_set<int>& recursionStack) {
            if (recursionStack.count(current)) {
                return true; // Cycle detected
            }

            if (visited.count(current)) {
                return false;
            }
```

If neither of the above conditions are met, add the current vertex to the set of visited vertices and to the set of vertices in the recursion stack. Next, create a list of neighbors for the current vertex. If the current vertex has neighbors, iterate over them. Recursively check to see if each neighbor has a cycle. If so, return `true`. If the loop ends without finding a cycle, remove the current node from `recursionStack` since its subtree has been explored. Then return `false`.

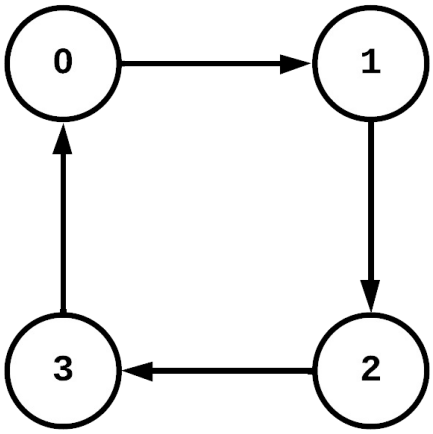
```
        visited.insert(current);
        recursionStack.insert(current);

        const std::vector<int>& edges = adjList[current];
        for (int neighbor : edges) {
            if (hasCycleHelper(neighbor, visited,
        recursionStack)) {
                return true;
            }
        }

        recursionStack.erase(current);
        return false;
    }
```

Testing the Code

Now, let's create a `graph` and test our cycle detection method. The example graph we are going to use looks like this:



Create the `main` method in your program by copying and pasting the code below into the editor.

```
int main() {
    DirectedGraph graph;
    graph.addVertex(0);
    graph.addVertex(1);
    graph.addVertex(2);
    graph.addVertex(3);

    graph.addEdge(0, 1);
    graph.addEdge(1, 2);
    graph.addEdge(2, 3);
    graph.addEdge(3, 0);

    std::cout << "Does the graph have a cycle? " <<
        (graph.hasCycle() ? "true" : "false") << std::endl;

    return 0;
}
```

▼ **Code**

> Your code should look like this:

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <unordered_set>

class DirectedGraph {
private:
    std::unordered_map<int, std::vector<int>> adjList;

public:
    DirectedGraph() {}

    void addVertex(int vertex) {
        adjList[vertex];  // Automatically initializes an
        empty vector if vertex doesn't exist
    }

    void addEdge(int vertex1, int vertex2) {
        adjList[vertex1].push_back(vertex2); // Directed
        edge
    }

    bool hasCycle() {
        std::unordered_set<int> visited;
        std::unordered_set<int> recursionStack;

        for (const auto& pair : adjList) {
            int vertex = pair.first;
            if (hasCycleHelper(vertex, visited,
        recursionStack)) {
                return true;
            }
        }
        return false;
```

```
    }

private:
    bool hasCycleHelper(int current,
        std::unordered_set<int>& visited,
        std::unordered_set<int>& recursionStack) {
        if (recursionStack.count(current)) {
            return true; // Cycle detected
        }

        if (visited.count(current)) {
            return false;
        }

        visited.insert(current);
        recursionStack.insert(current);

        const std::vector<int>& edges = adjList[current];
        for (int neighbor : edges) {
            if (hasCycleHelper(neighbor, visited,
        recursionStack)) {
                return true;
            }
        }

        recursionStack.erase(current);
        return false;
    }
};

int main() {
    DirectedGraph graph;
    graph.addVertex(0);
    graph.addVertex(1);
    graph.addVertex(2);
    graph.addVertex(3);

    graph.addEdge(0, 1);
    graph.addEdge(1, 2);
    graph.addEdge(2, 3);
    graph.addEdge(3, 0);

    std::cout << "Does the graph have a cycle? " <<
        (graph.hasCycle() ? "true" : "false") << std::endl;

    return 0;
}
```

You should see the following output:

```
Does the graph have a cycle? true
```

challenge

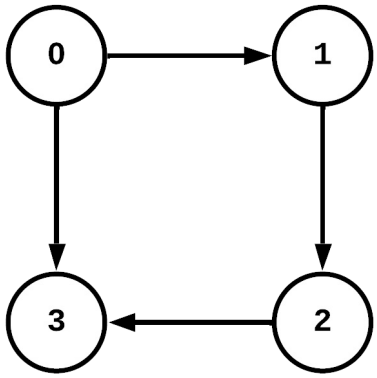
Try this variation

Update the last edge in the example graph to look like the following:

```
graph.addEdge(0, 3);
```

▼ **What is happening?**

> The graph no longer has a cycle since the direction of the last edge was reversed. The graph now looks like this:



This implementation of cycle detection in directed graphs is a fundamental skill in graph algorithms. It can be extended and modified for various applications, including detecting feedback loops in systems, finding circular dependencies in modules, and more.

Cycles in Undirected Graphs

Cycle Detection in Undirected Graphs

Before we explore the cycle detection algorithm for undirected graphs, we need to first understand how cycles differ in directed and undirected graphs. In a directed graph, if we traverse a graph and encounter a previously visited vertex, a cycle is present. This is a very simple definition because you can only traverse a graph in a specified manner.

In an undirected graph, if vertex A is connected to vertex B, then B is also connected to A. So, I can go from A to B and back to A again. Vertex A was already visited, so does that mean a cycle is present? No, it does not. When we traverse an undirected graph, we are going to mark the preceding vertex as the "parent." If we encounter an already visited vertex and that vertex is not the parent, then a cycle is present.

Detecting cycles in undirected graphs requires a different approach compared to directed graphs. In undirected graphs, we need to ensure that the presence of an edge connecting back to an ancestor does not automatically imply a cycle, as it could just be the bidirectional nature of the graph. We will use a depth-first search (DFS) approach, with an additional check to differentiate between backtracking and actual cycles. Remember Cycle in Undirected Graphs: An undirected graph has a cycle if there is a path of unique vertices that starts and ends at the same vertex.

Implementing Cycle Detection

We'll use a similar `UndirectedGraph` class as before. A `std::map` will contain an adjacency list. The `addVertex` method adds a vertex to the list, and the `addEdge` method adds an edge. However, adding an edge will now create a bidirectional connection between two vertices.

```cpp
#include <iostream>
#include <map>
#include <vector>
#include <set>

class UndirectedGraph {
private:
    std::map<int, std::vector<int>> adjList;

public:
    void addVertex(int vertex) {
        adjList[vertex];
    }

    void addEdge(int vertex1, int vertex2) {
        adjList[vertex1].push_back(vertex2);
        adjList[vertex2].push_back(vertex1);
    }
};
```

Just as with directed graphs, we will create the `hasCycle` method. This time we are only going to have a single set of the visited vertices. We do not need to keep track of the recursion stack for cycle detection in undirected graphs. Iterate over the vertices in the graph. If the vertex has not already been visited, call the `hasCycleHelper` method. If this method returns `true`, then there is a cycle and the `hasCycle` method should return `true`. If the loop ends, return `false` since there are no cycles.

```cpp
    bool hasCycle() {
        std::set<int> visited;

        for (const auto& pair : adjList) {
            int vertex = pair.first;
            if (visited.find(vertex) == visited.end()) {
                if (hasCycleHelper(vertex, visited, -1)) {
                    return true;
                }
            }
        }
        return false;
    }
```

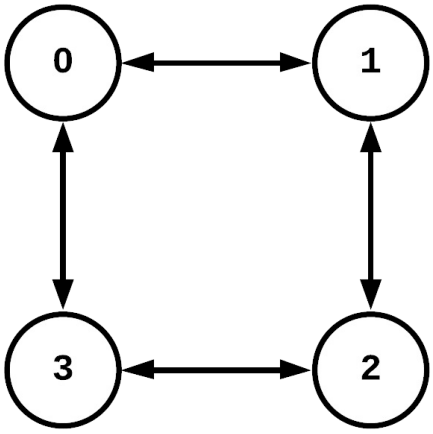
The `hasCycleHelper` method is a private helper method that returns a boolean value. The method takes an integer representing the current vertex, a set of vertices that have already been visited, and an integer representing the parent vertex. Add the current vertex to the set of visited vertices. Then get a list of the neighbors for the `current` vertex. Check if there are any neighbors. If there are no neighbors, return `false` since there is no cycle. If there are neighbors, iterate over them.

```
private:
    bool hasCycleHelper(int current, std::set<int>& visited, int
        parent) {
        visited.insert(current);

        const std::vector<int>& edges = adjList[current];
        for (int neighbor : edges) {
            if (visited.find(neighbor) == visited.end()) {
                if (hasCycleHelper(neighbor, visited, current))
        {
                    return true;
                }
            } else if (neighbor != parent) {
                return true;
            }
        }
        return false;
    }
```

Testing the Code

Now, let's create a graph and test our cycle detection method. The example graph we are going to use looks like this:



Create the main method in your program by copying and pasting the code below into the editor.

```
int main() {
    UndirectedGraph graph;
    graph.addVertex(0);
    graph.addVertex(1);
    graph.addVertex(2);
    graph.addVertex(3);

    graph.addEdge(0, 1);
    graph.addEdge(1, 2);
    graph.addEdge(2, 3);
    graph.addEdge(3, 0);

    std::cout << "Does the graph have a cycle? " <<
        (graph.hasCycle() ? "true" : "false") << std::endl;

    return 0;
}
```

▼ Code

> Your code should look like this:

```
#include <iostream>
#include <map>
#include <vector>
#include <set>

class UndirectedGraph {
private:
    std::map<int, std::vector<int>> adjList;

public:
    void addVertex(int vertex) {
        adjList[vertex];
    }

    void addEdge(int vertex1, int vertex2) {
        adjList[vertex1].push_back(vertex2);
        adjList[vertex2].push_back(vertex1);
    }

    bool hasCycle() {
        std::set<int> visited;

        for (const auto& pair : adjList) {
            int vertex = pair.first;
            if (visited.find(vertex) == visited.end()) {
                if (hasCycleHelper(vertex, visited, -1)) {
                    return true;
                }
            }
        }
        return false;
    }
```

```
private:
    bool hasCycleHelper(int current, std::set<int>&
        visited, int parent) {
        visited.insert(current);

        const std::vector<int>& edges = adjList[current];
        for (int neighbor : edges) {
            if (visited.find(neighbor) == visited.end()) {
                if (hasCycleHelper(neighbor, visited,
        current)) {
                    return true;
                }
            } else if (neighbor != parent) {
                return true;
            }
        }
        return false;
    }
};

int main() {
    UndirectedGraph graph;
    graph.addVertex(0);
    graph.addVertex(1);
    graph.addVertex(2);
    graph.addVertex(3);

    graph.addEdge(0, 1);
    graph.addEdge(1, 2);
    graph.addEdge(2, 3);
    graph.addEdge(3, 0);

    std::cout << "Does the graph have a cycle? " <<
        (graph.hasCycle() ? "true" : "false") << std::endl;

    return 0;
}
```

You should see the following output:

```
Does the graph have a cycle? true
```

challenge

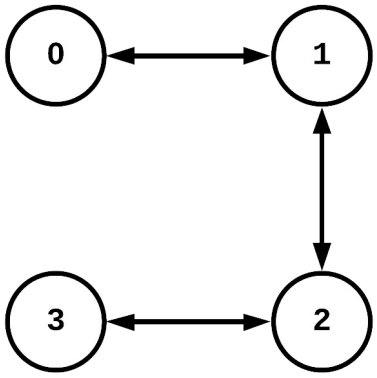
Try this variation

Update the example such that there are only three edges:

```
graph.addEdge(0, 1);
graph.addEdge(1, 2);
graph.addEdge(2, 3);
```

▼ **What is happening?**

> The graph no longer has a cycle since there is no edge that leads back to a previously visited vertex (ignoring parents). The graph now looks like this:



Cycle detection in undirected graphs also recursively performs DFS, checking for cycles. The `parent` parameter is crucial to avoid false cycle detection due to the bidirectional nature of undirected graphs. It is fundamental for applications like network routing, circuit design, and more. Understanding this concept lays the groundwork for more advanced graph algorithms and paves the way for exploring other complex aspects of graph theory.

Formative Assessment 1

Formative Assessment 2