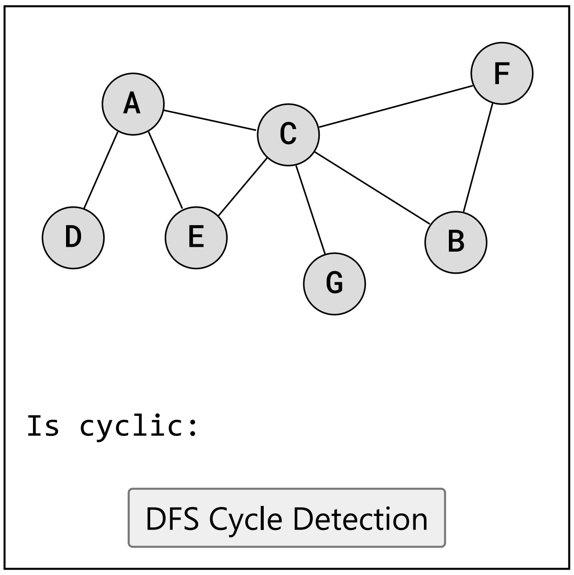
DSA Graphs Cycle Detection

❮ Previous Next ❯

Cycles in Graphs

A cycle in a Graph is a path that starts and ends at the same vertex, where no edges are repeated. It is similar to walking through a maze and ending up exactly where you started.



A cycle can be defined slightly different depending on the situation. A self-loop for example, where an edge goes from and to the same vertex, might or might not be considered a cycle,

Cycle Detection

It is important to be able to detect cycles in Graphs because cycles can indicate problems or special conditions in many applications like networking, scheduling, and circuit design.

The two most common ways to detect cycles are:

1. **Depth First Search (DFS):** DFS traversal explores the Graph and marks vertices as visited. A cycle is detected when the current vertex has an adjacent vertex that has already been visited.
2. **Union-Find:** This works by initially defining each vertex as a group, or a subset. Then these groups are joined for every edge. Whenever a new edge is explored, a cycle is detected if two vertices already belong to the same group.

How cycle detection with DFS and Union-Find work, and how they are implemented, are explained in more detail below.

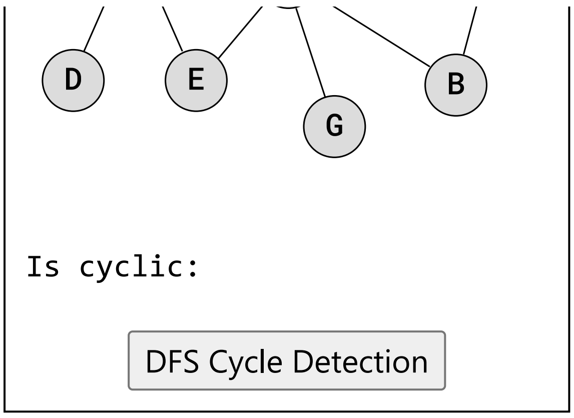
DFS Cycle Detection for Undirected Graphs

To detect cycles in an undirected Graph using Depth First Search (DFS), we use a code very similar to the DFS traversal code on the previous page, with just a few changes.

How it works:

1. Start DFS traversal on each unvisited vertex (in case the Graph is not connected).
2. During DFS, mark vertices as visited, and run DFS on the adjacent vertices (recursively).
3. If an adjacent vertex is already visited and is not the parent of the current vertex, a cycle is detected, and `True` is returned.
4. If DFS traversal is done on all vertices and no cycles are detected, `False` is returned.

Run the animation below to see how DFS cycle detection runs on a specific Graph, starting in vertex A (this is the same as the previous animation).



The DFS traversal starts in vertex A because that is the first vertex in the adjacency matrix. Then, for every new vertex visited, the traversal method is called recursively on all adjacent vertices that have not been visited yet. The cycle is detected when vertex F is visited, and it is discovered that the adjacent vertex C has already been visited.

Example

Python:

```python
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = 1
            self.adj_matrix[v][u] = 1

    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
            self.vertex_data[vertex] = data

    def print_graph(self):
        print("Adjacency Matrix:")
        for row in self.adj_matrix:
            print(' '.join(map(str, row)))
        print("\nVertex Data:")
```

```python

g = Graph(7)

g.add_vertex_data(0, 'A')
g.add_vertex_data(1, 'B')
g.add_vertex_data(2, 'C')
g.add_vertex_data(3, 'D')
g.add_vertex_data(4, 'E')
g.add_vertex_data(5, 'F')
g.add_vertex_data(6, 'G')

g.add_edge(3, 0)  # D - A
g.add_edge(0, 2)  # A - C
g.add_edge(0, 3)  # A - D
g.add_edge(0, 4)  # A - E
g.add_edge(4, 2)  # E - C
g.add_edge(2, 5)  # C - F
g.add_edge(2, 1)  # C - B
g.add_edge(2, 6)  # C - G
g.add_edge(1, 5)  # B - F

```

Try it Yourself »

Line 66: The DFS cycle detection starts when the `is_cyclic()` method is called.

Line 37: The `visited` array is first set to `false` for all vertices, because no vertices are visited yet at this point.

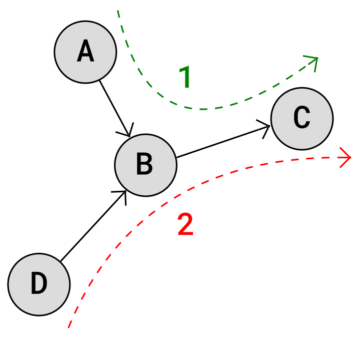
Line 38-42: DFS cycle detection is run on all vertices in the Graph. This is to make sure all vertices are visited in case the Graph is not connected. If a node is already visited, there must be a cycle, and `True` is returned. If all nodes are visited just ones, which means no cycles are detected, `False` is returned.

Line 24-34: This is the part of the DFS cycle detection that visits a vertex, and then visits adjacent vertices recursively. A cycle is detected and `True` is returned if an adjacent vertex has already been visited, and it is not the parent node.

DFS Cycle Detection for Directed Graphs

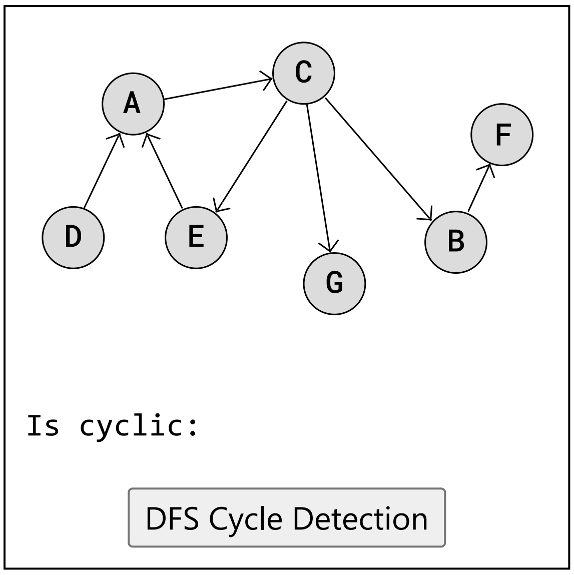
To detect cycles in Graphs that are directed, the algorithm is still very similar as for undirected Graphs, but the code must be modified a little bit because for a directed Graph, if we come to an adjacent node that has already been visited, it does not necessarily mean that there is a cycle.

Just consider the following Graph where two paths are explored, trying to detect a cycle:



In path 1, the first path to be explored, vertices A->B->C are visited, no cycles detected.

node has been visited before in the same path.



To implement DFS cycle detection on a directed Graph, like in the animation above, we need to remove the symmetry we have in the adjacency matrix for undirected Graphs. We also need to use a `recStack` array to keep track of visited vertices in the current recursive path.

Example

Python:

```
class Graph:
    # ......
    def add_edge(self, u, v):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = 1

    # ......
    def dfs_util(self, v, visited, recStack):
        visited[v] = True
        recStack[v] = True
        print("Current vertex:",self.vertex_data[v])

        for i in range(self.size):
```

```
20
21            recStack[v] = False
22            return False
23
24        def is_cyclic(self):
25            visited = [False] * self.size

27            for i in range(self.size):
28                if not visited[i]:
29                    print() #new line
30                    if self.dfs_util(i, visited, recStack):
31                        return True
32            return False
33
34    g = Graph(7)
35
36    # ......
37
38    g.add_edge(3, 0)  # D -> A
39    g.add_edge(0, 2)  # A -> C
40    g.add_edge(2, 1)  # C -> B
41    g.add_edge(2, 4)  # C -> E
42    g.add_edge(1, 5)  # B -> F
43    g.add_edge(4, 0)  # E -> A
44    g.add_edge(2, 6)  # C -> G
45
46    g.print_graph()
47
48    print("Graph has cycle:", g.is_cyclic())
```

Try it Yourself »

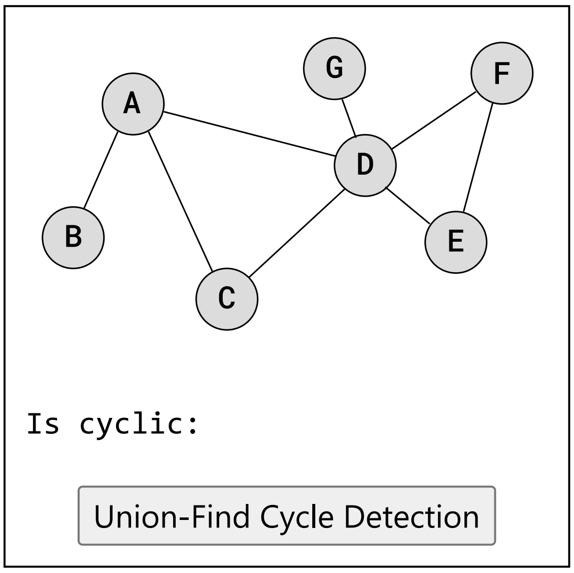
Line 6: This line is removed because it is only applicable for undirected Graphs.

Line 26: The `recStack` array keeps an overview over which vertices have been visited during a recursive exploration of a path.

Union-Find Cycle Detection

Detecting cycles using Union-Find is very different from using Depth First Search.

Union-Find cycle detection works by first putting each node in its own subset (like a bag or container). Then, for every edge, the subsets belonging to each vertex are merged. For an edge, if the vertices already belong to the same subset, it means that we have found a cycle.



In the animation above, Union-Find cycle detection explores the edges in the Graph. As edges are explored, the subset of vertex A grows to also include vertices B, C, and D. The cycle is detected when the edge between A and D is explored, and it is discovered that both A and D already belong to the same subset.

The edges between D, E, and F also construct a circle, but this circle is not detected because the algorithm stops (returns `True`) when the first circle is detected.

Union-Find cycle detection is only applicable for Graphs that are undirected.

Union-Find cycle detection is implemented using the adjacency matrix representation, so setting up the Graph structure with vertices and edges is basically the same as in previous examples.

Example

```
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size
        self.parent = [i for i in range(size)]  # Union-Find array

    def add_edge(self, u, v):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = 1
            self.adj_matrix[v][u] = 1

    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
            self.vertex_data[vertex] = data

    def find(self, i):
        if self.parent[i] == i:
            return i
        return self.find(self.parent[i])

    def union(self, x, y):
        x_root = self.find(x)
        y_root = self.find(y)
        print('Union:',self.vertex_data[x],'+',self.vertex_data[y])
        self.parent[x_root] = y_root
        print(self.parent,'\n')

    def is_cyclic(self):
        for i in range(self.size):
            for j in range(i + 1, self.size):
                if self.adj_matrix[i][j]:
                    x = self.find(i)
                    y = self.find(j)
                    if x == y:
                        return True
                    self.union(x, y)
        return False

g = Graph(7)
```

```
46  g.add_vertex_data(4, 'E')
47  g.add_vertex_data(5, 'F')
48  g.add_vertex_data(6, 'G')
49
50  g.add_edge(1, 0)  # B - A
51  g.add_edge(0, 3)  # A - D
52  g.add_edge(0, 2)  # A - C
53  g.add_edge(2, 3)  # C - D
54  g.add_edge(3, 4)  # D - E
55  g.add_edge(3, 5)  # D - F
56  g.add_edge(3, 6)  # D - G
57  g.add_edge(4, 5)  # E - F
58
59  print("Graph has cycle:", g.is_cyclic())
```

Try it Yourself »

Line 6: The `parent` array contains the root vertex for every subset. This is used to detect a cycle by checking if two vertices on either side of an edge already belong to the same subset.

Line 17: The `find` method finds the root of the set that the given vertex belongs to.

Line 22: The `union` method combines two subsets.

Line 29: The `is_cyclic` method uses the `find` method to detect a cycle if two vertices `x` and `y` are already in the same subset. If a cycle is not detected, the `union` method is used to combine the subsets.

DSA Exercises

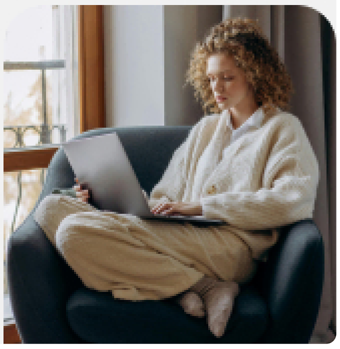
Test Yourself With Exercises

What is a cycle in a Graph?

```
A cycle in a Graph is a path
that starts and ends at the
same        , where no
are repeated.
```

Submit Answer »

Start the Exercise



PLUS SPACES

GET CERTIFIED FOR TEACHERS

FOR BUSINESS CONTACT US

Top Tutorials

HTML Tutorial
CSS Tutorial
JavaScript Tutorial
How To Tutorial
SQL Tutorial
Python Tutorial
W3.CSS Tutorial
Bootstrap Tutorial
PHP Tutorial
Java Tutorial
C++ Tutorial
jQuery Tutorial

Top References

HTML Reference
CSS Reference
JavaScript Reference
SQL Reference
Python Reference
W3.CSS Reference
Bootstrap Reference
PHP Reference
HTML Colors
Java Reference

CSS Examples
JavaScript Examples
How To Examples
SQL Examples
Python Examples
W3.CSS Examples
Bootstrap Examples
PHP Examples
Java Examples
XML Examples
jQuery Examples

CSS Certificate
JavaScript Certificate
Front End Certificate
SQL Certificate
Python Certificate
PHP Certificate
jQuery Certificate
Java Certificate
C++ Certificate
C# Certificate
XML Certificate

FORUM ABOUT ACADEMY