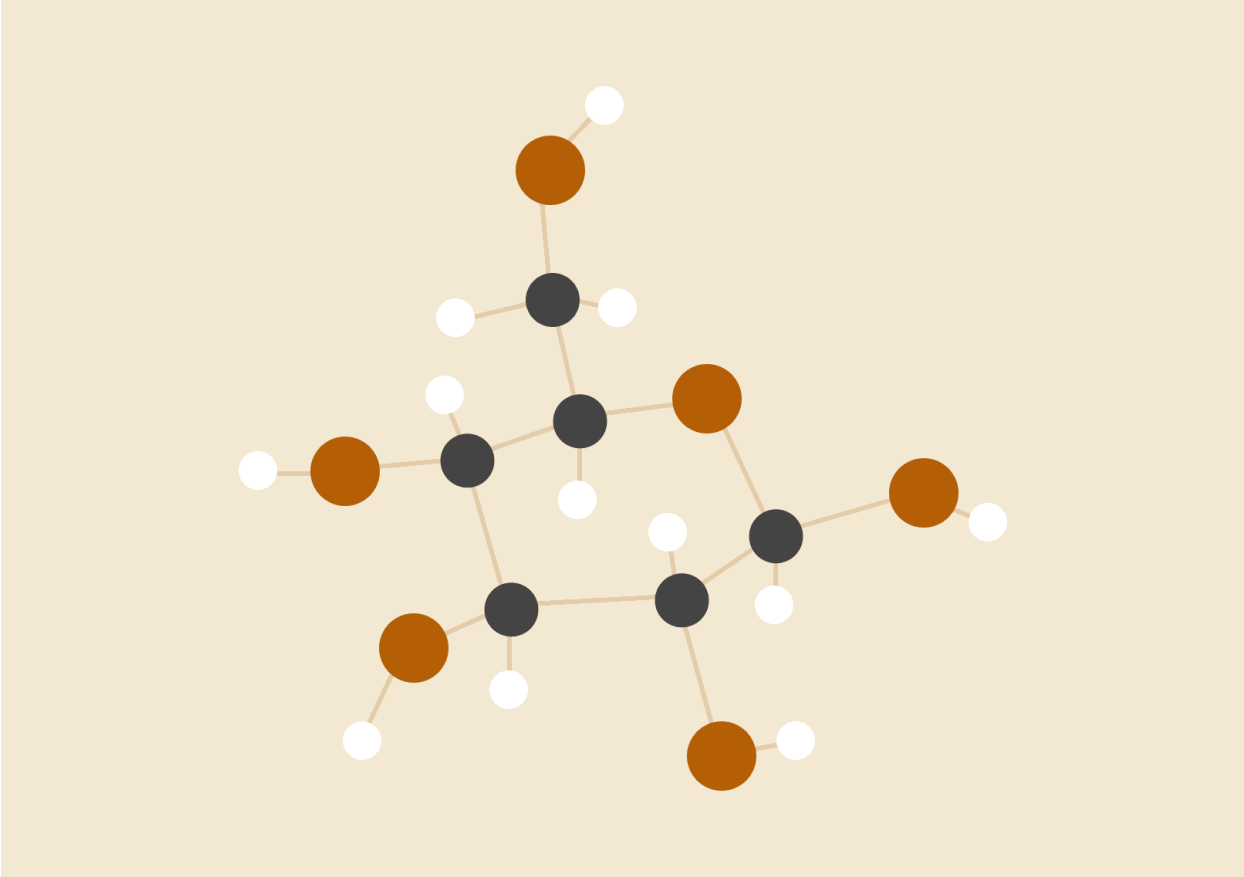
DiGraph cycle detection

Implementation of directed graph cycles and loops detection using DFS technique



Alexander Momot

15.08.2022

Introduction

According to a graph theory, the graph is a mathematical structures used to model pairwise relation between objects. Usually, graphs in this context is made up from vertices and edges. Edges could and could not connect vertices one and more times, edges can be undirected and directed.

If the given graph contains vertices and edges with the directions (e.g. arrows), it's called a **directed graph**. Assuming that the edge has a direction, and if there are enough nodes (vertices) and edges between them, it's possible to find a route from the starting point "A" to the ending point "A" which we're calling a "cycle". There is also possibility that the point "A" is pointing out to itself, in that case we're calling it the "loop"

According to the challenge description, in practice we favor directed graphs without cycles a.k.a. Directed Acyclic Graph, so the main task here is to find a loop or cycle (if it exists) and return a backtrack or full route from the start to the end of that cycle.

During the challenge it's necessary to:

1) Discover existing solutions and pick the algorithm
2) Find or implement a graph storage, or use some common library
3) Create an application on python which implements the search of cycle
4) Implement a backtrack array/list storage to track the loop
5) Find or implement a library to draw a graph
6) Show a graph using graphical library, colorize the backtrack as red
7) Add some tests with different cases
8) Document the solution

Requirements

1. Python 3.8.9+
2. Pipenv
3. Matplotlib
4. networkx

How to start

- Please check that python and pipenv is installed in your system:
    - **~ $ python3 --version**
    - **~ $ pipenv --version**
- Initialize the local environment and install the dependencies locally, then use venv:
    - **~ $ cd acyclic-digraph-solver**
    - **~/acyclic-digraph-solver $ pipenv sync**
    - **~/acyclic-digraph-solver $ pipenv shell**
- Finally, run the application using common entry point ./main.py with number of vertices:
    - **~/acyclic-digraph-solver $ ./main.py**
    - **~/acyclic-digraph-solver $ ./main.py 10**
    - **~/acyclic-digraph-solver $ ./main.py 100**

In case if you want to run all test cases, you can call a tests module or just call a unittest framework directly:

- **~/acyclic-digraph-solver $ ./tests.py**
- **~/acyclic-digraph-solver $ python -m unittest tests.py**

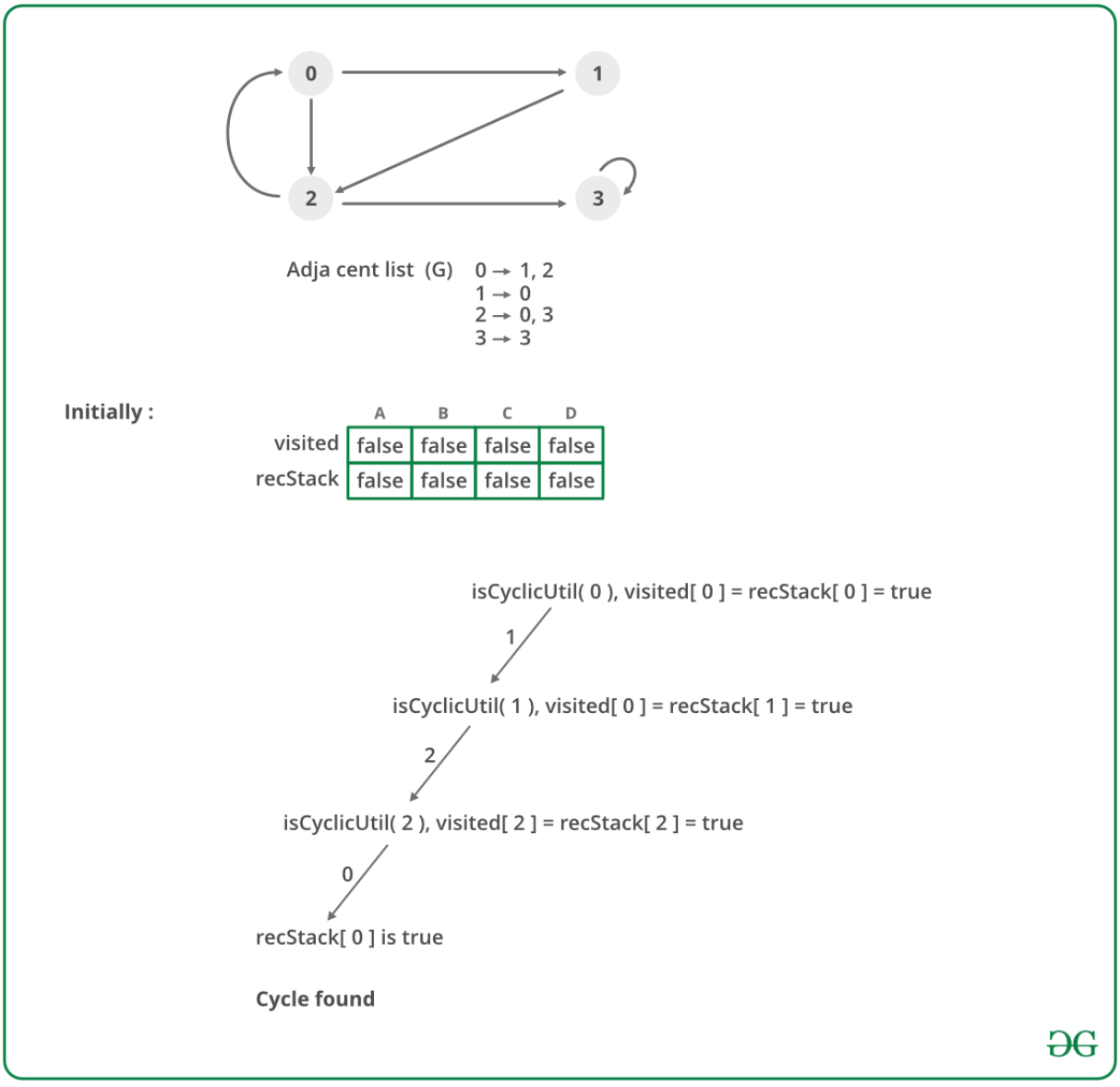
Hypothesis

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them.

Methodology

Algorithm:

- Create the graph using the given number of edges and vertices.
- Create a recursive function that initializes the current index or vertex, visited, and recursion stack.
- Mark the current node as visited and also mark the index in the recursion stack.
- Find all the vertices which are not visited and are adjacent to the current node. Recursively call the function for those vertices, If the recursive function returns true, return true.
- If the adjacent vertices are already marked in the recursion stack then return true.
- Create a wrapper class that calls the recursive function for all the vertices and if any function returns true return true. Else if for all vertices the function returns false return false.



Getting the results

After the algorithm finds the visited neighbor, it traverses back to the stack each time visiting the nodes in reverse order. This means that at some point, it finds the current node equals the final, which was found during the initial travel. Saving this path to a list, or stack and reverse will produce the matching cycle path or loop, so the path can be traversed again.

The output array looks like this:

[4, 8, 2, 4]

Complexity

To calculate a complexity the Big O notation is used

Time Complexity: O(n+m) — it's need to recur check all vertices and edges

Space Complexity: O(n) — in order to store the stack visited stack O(n) space is required

Examples

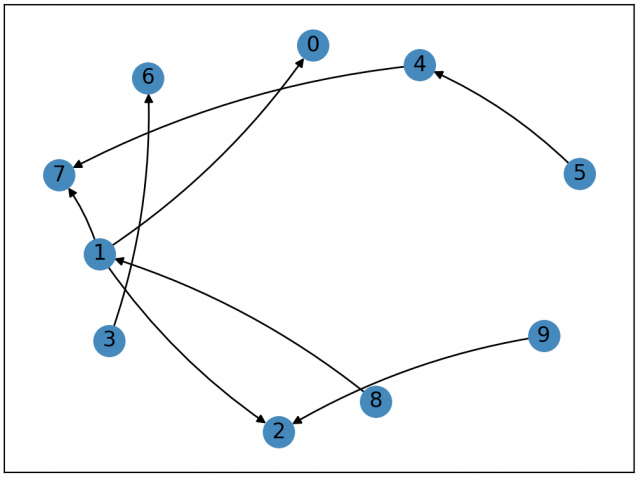


fig. 1 - no cycle

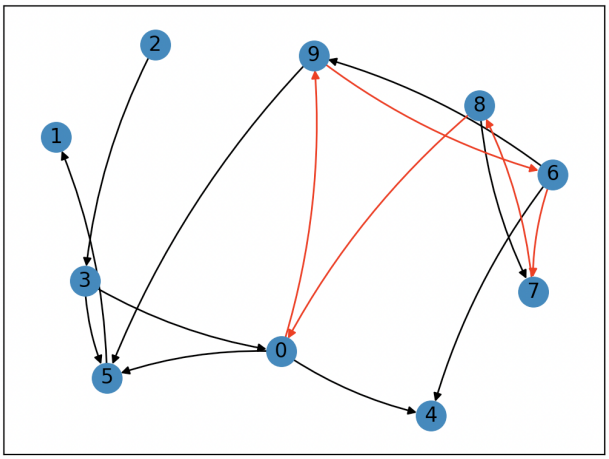
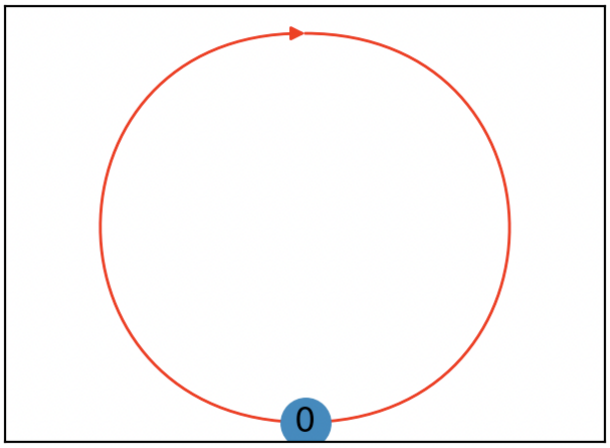
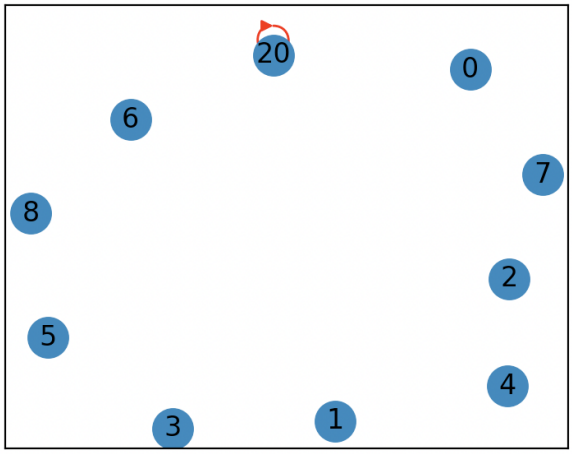


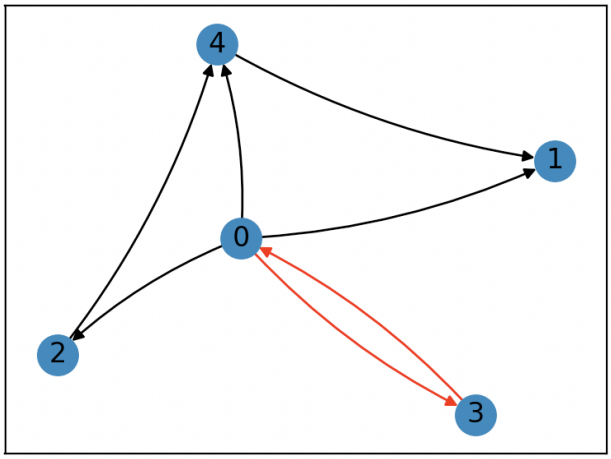
fig. 2 - cycle found: 0->9->6->7->8->0

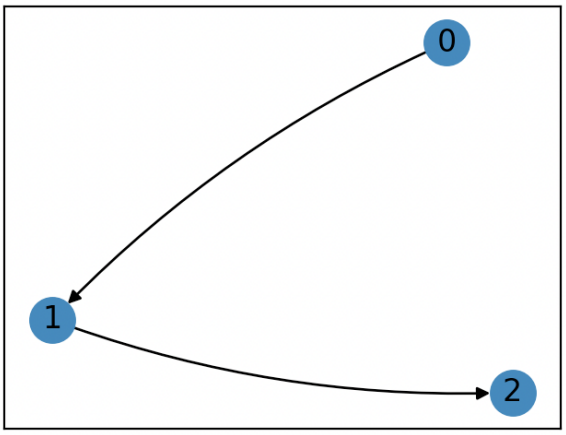
Single loop and Dangling nodes single loop



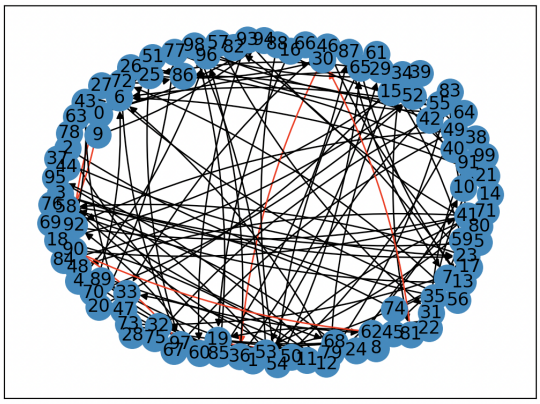


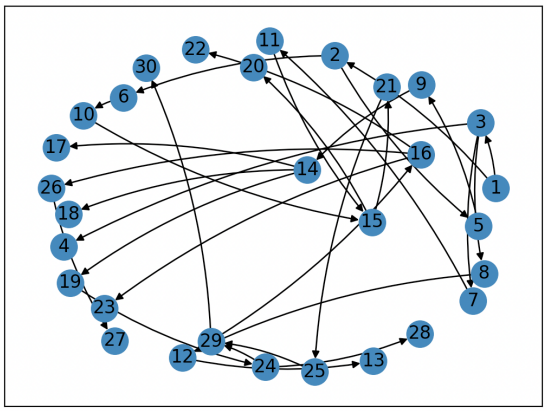
Bi-directed cycle and no cycle loop





One hundred nodes graph and a graph with many terminal vertices





Randomly Generating 100 different digraphs with possibility 13% of generating an edge

After the algorithm finds the visited neighbor, it traverses back to the stack each time visiting the nodes in reverse order. This means that at some point, it finds the current node equals the final, which was found during the initial travel. Saving this path to a list, or stack and reverse will produce the matching cycle path or loop, so the path can be traversed again.

https://drive.google.com/file/d/1LTH21qksZ0QZ1G3gYB1qM0KzHJjG85Fl/view?usp=sharing

Sources list

1. https://en.wikipedia.org/wiki/Cycle_(graph_theory)
2. https://en.wikipedia.org/wiki/Directed_acyclic_graph
3. https://en.wikipedia.org/wiki/Graph_theory
4. https://en.wikipedia.org/wiki/Depth-first_search
5. https://habr.com/ru/post/558806/