Practice 10

Mayra Cristina Berrones Reyes

April 3, 2020

1 Introduction

For this practice we continue to explore some more data structures related to graphs. The subject is topological ordering or sorting. In this case it is necessary to refresh other concepts first.

1.1 Directed Acyclic Graphs

First of, a graph is a structure of several nodes that are connected by edges. In this case, this edges are depicted by arrows which represents the single directional flow from one node to the other. And acyclic means that there are no loops or cycles in the graph. A directed acyclic graph (DAG) is a type of graph in which, if you start in a certain node and follow the edges that connect this node to another, there is no path in the graph that can get you back to the original node.

Now, in a real world sense, we can see the DAG structure as a representation of a series of activities. The order in which this activities must be performed are depicted by the graph structure, each node representing an activity, some of them connected by lines that represent the flow from one activity to the other. Seeing it this way, there are many real world situation that we can model as a DAG, such as:

- School classes pre requisites
- Event scheduling
- Instructions for assembly.
- Program dependencies

An easy example for this can be seen in Figure 1 where each node is a class, and if, for example we want to take the class of the node 8, we need to take class 1, 2, 4, and 5 before we can take class 8.

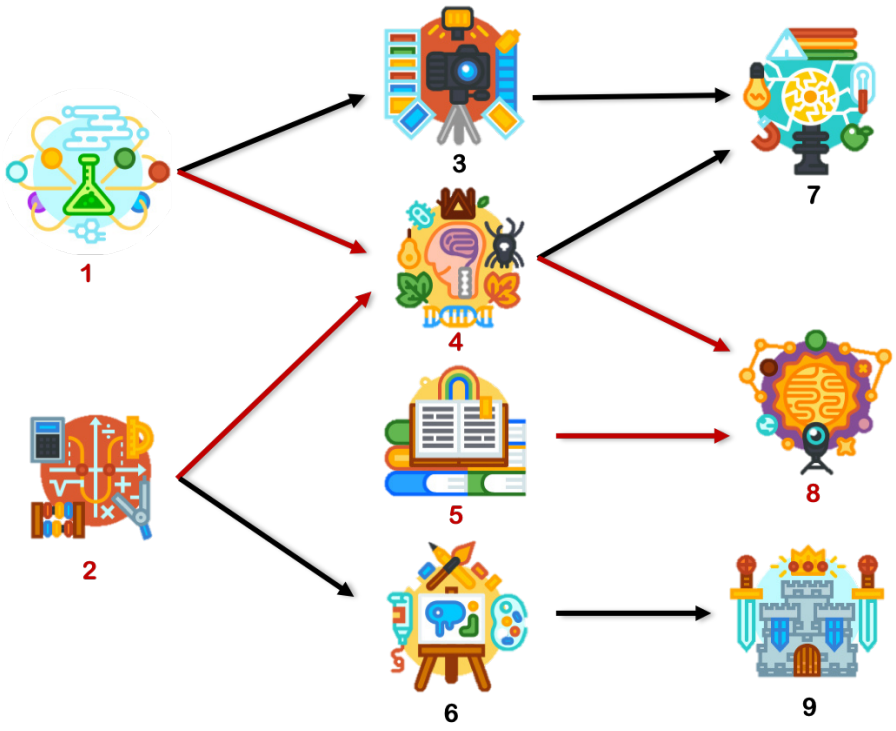

Figure 1: Example of the school class pre requisite.

2 Topological sorting

Topological sorting is an ordering of nodes. To explain it we can look again at Figure 1, and we can look at the nodes as a stack. A stack is a linear data structure that can store items in a last in/ first out (LIFO) or a First in/ last out (FILO) manner. The way a stack works is, when a new element is added, it goes on top of the last element added, and the only way to remove it, is by taking out all of the elements above this item. The insert and delete operations are commonly called push and pop.

There are two conditions in order to find the topological ordering or sorting of a graph:

- The graph must be a directed acyclic graph.
- The initial node in a topological graph should be a node with no incoming edges.

The first condition is explained above. The second condition is because, if the chosen node has incoming edges, then it can not be taken out of the stack. It is important to note that there is not a singular solution for this topological sorting. Now this is important because there are several ways to approach this problem.

2.1 Kahn's Algorithm

In this way we need use the adjacent list to select the order of the nodes. The steps for this are:

1. Identify the node that has no incoming edges. Select that node to start.
2. Delete the starting node and delete all the outgoing edges from the graph. Place the deleted node in the output list.
3. Repeat step 1 and 2 until the graph is empty.

As we can see in the steps, this algorithm works by starting with the nodes that have no incoming edges. In this case we need to choose the one node that has the same order as the final topological sort, so we need to pick the one with the most outgoing edges. The solution is an ordered list, that it is not necessarily unique.

In Figure 2 we see an example of a graph solved in this way.

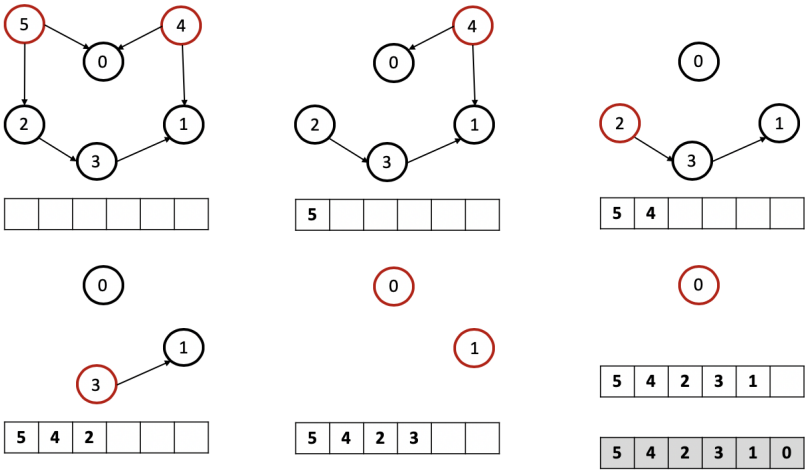

Figure 2: Example using the Kahn's algorithm.

2.2 Depth-first search

An alternative algorithm to solve this problem is based on depth-first search (DFS). In this other way the algorithm loops through each node of the graph in arbitrary order. From this node, it does a DFS and find the last node connected with a path to the starting node, save it on a stack, then delete that node. This step repeats until the graph is empty. In this case, in Figure 3 depicts the way it deletes the nodes. In the end, we print the elements of the stack. Since the only way to pop items from a stack is last ones first, we get the topological ordering. Since each edge and node are visited once, the algorithm runs in linear time.

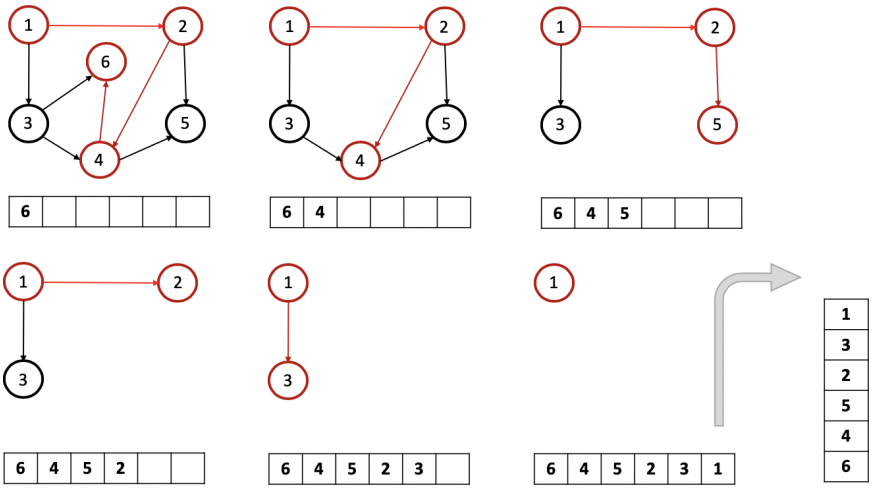

Figure 3: Example using the DFS algorithm

3 Code and terminal

```
from collections import defaultdict
class Graph:
  def __init__(self,vertices):
    self.graph = defaultdict(list) #adjacency List
    self.V = vertices #vertices

  def addEdge(self,u,v):
    self.graph[u].append(v)


  def topologicalSortUtil(self,v,visited,stack):
    visited[v] = True
    for i in self.graph[v]:
      if visited[i] == False:
        self.topologicalSortUtil(i,visited,stack)

    # Push
    stack.insert(0,v)

  def topologicalSort(self):
    visited = [False]*self.V
    stack =[]

    for i in range(self.V):
      if visited[i] == False:
        self.topologicalSortUtil(i,visited,stack)

    print stack

g= Graph(6)
g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);

print "Topological Sort"
g.topologicalSort()
```

```
(base) Mayras-MacBook-Pro:desktop mayraberrones$ python topo2.py
Topological Sort
[5, 4, 2, 3, 1, 0]
(base) Mayras-MacBook-Pro:desktop mayraberrones$
```

Figure 4: Result on terminal

4 Conclusions

Since there is not one concrete answer to this algorithm I struggled a little at the beginning because I wanted to do them by hand to understand them. Seeing the

problem as a DFS made it a lot easier to understand. Also the first example with the classes made it more relatable, since it is easier to understand the graph as a series of tasks that you have to complete in order to get to the one that you want.

Since I have worked with the Hamiltonian path in several other practices, the way the algorithm of DFS reminded me a little of finding paths in the graph. After reading some more on this, I found that there is a property on the topological sort that says if all pairs of consecutive nodes in the sorted order are connected by edges, then this form a Hamiltonian path. This property is called uniqueness, and it has been used to test in linear time if a Hamiltonian path exist despite its complexity. This could have been useful when I was studying the subject of reductions.

References

[1] Topological sort algorithms. Graph theory. `https://youtu.be/eL-KzMXSXXI`

[2] Stack in python `https://www.geeksforgeeks.org/stack-in-python/`

[3] DAG `https://golden.com/wiki/Directed_acyclic_graph-RYDG9`

[4] Topological Sorting in Python `https://www.codespeedy.com/topological-sorting-in-python/`