Topological Sort

Outline

- Definition **+ Use case**
  - Course Scheduler
- **Topological Sort Template**
- **Talk is cheap. Show me the code.**
  1. **Course Scheduler**
  2. **Alien Dictionary**
  3. **Longest Increasing Path in a Matrix**

Topological Sort | Definition

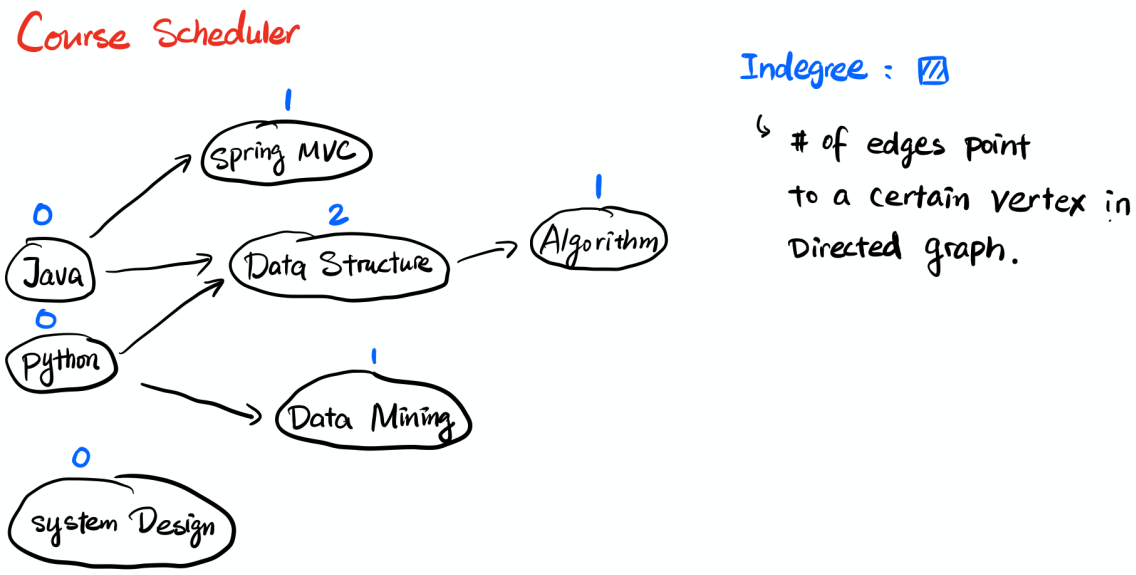
Topological sort or topological ordering of a `directed graph` is a linear ordering of its `vertices` such that for every directed edge uv from vertex u to vertex v, u `comes before` v in the ordering.

Key Concept

`Indegree`

Number of edges point to a certain vertex in `Directed Graph`

▼ Graph



Thought Process:

Initialization

- Calculate **Indegree for all vertices**
- **If Vertice's Indgree ==** `0` **, EnQueue Vertices as starting points**

BFS

- **Pop** `node1` **from Queue, Update all adjacent vertices's indegree**
- **Enqueue condition: When discover new vertices indegree == 0**

▼ 1. Topological Sorting

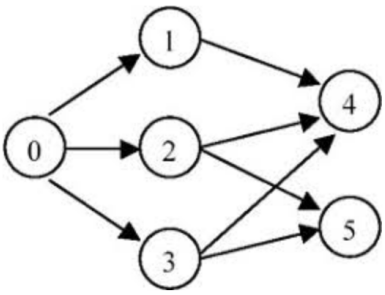
Given an directed graph, a topological order of the graph nodes is defined as follow:

1. For each directed edge A → B in graph, A must before B in the order list.
2. The first node in the order can be any node in the graph with no nodes direct to it.

Find any topological order for the given graph.
You can assume that there is **at least one topological order** in the graph.

Example

For graph as follow:



The topological order can be:

```
[0, 1, 2, 3, 4, 5]
[0, 2, 3, 1, 5, 4]
...
```

▼ (Final)

Time Complexity: `O(n)` | Space Complexity: `O(n)`

```
"""
Definition for a Directed graph node
class DirectedGraphNode:
    def __init__(self, x):
        self.label = x
        self.neighbors = []
"""

from collections import deque
class Solution:
    """
    @param graph: A list of Directed graph node
    @return: Any topological order for the given graph.
    """
    def topSort(self, graph):
        # Initialization
        indeg = self.getIndeg(graph)
```

```
        start_nodes = [node for node in graph if indeg[node] == 0]
        res = []
        queue = deque(start_nodes)

        # BFS
        while queue:
            node = queue.popleft()
            res.append(node)
            for neighbor in node.neighbors:
                indeg[neighbor] -= 1
                if indeg[neighbor] == 0:
                    queue.append(neighbor)
        return res

    def getIndeg(self, graph):
        indeg = {node : 0 for node in graph}

        for node in graph:
            for neighbor in node.neighbors:
                indeg[neighbor] += 1

        return indeg
```

算法模板

Initialization

- Calculate **Indegree for all vertices**
- **If Vertice's Indgree == `0`，EnQueue Vertices as starting points**

BFS

- **Pop `node1` from Queue，Update all adjacent vertices's indegree**

```
def topSort(graph):
    # Initialization
    indeg = getIndeg(graph)
    start = [n for n in graph if indeg[n] == 0]
    res = []
    queue = collections.deque(start)


    # BFS
    while queue:
        node = queue.popleft()
        res.append(node)
        for neighbor in node.neighbors:
            indeg[neighbor] -= 1
            if indeg[neighbor] == 0:
                queue.append(neighbor)
    return res

def getIndeg(graph):
```

- **Enqueue condition: When discover new vertices indegree == 0**

```
    indeg = {node : 0 for node in graph}

    for node in graph:
        for neighbor in node.neighbors:
            indeg[neighbor] += 1
    return indeg
```

Topological Sort | Questions

▼ **Course Schedule**

There are a total of numCourses courses you have to take, labeled from 0 to numCourses-1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

Example 1:

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0. So it is possible.
```

Example 2:

```
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0, and to take course 0
you should
             also have finished course 1. So it is impossible.
```

▼ Solution **Final**

Time Complexity: `O(V + E)` | Space Complexity: `O(V + E)`

```
from collections import deque, defaultdict
class Solution(object):
    def canFinish(self, n, courses):
        graph, indeg = self.makeGraph(n, courses)
        return self.bfsHelper(n, graph, indeg)

    def makeGraph(self, n, courses):
        graph = defaultdict(list)
        indeg = {i : 0 for i in range(n)}

        for course , pre in courses:
            indeg[course] += 1
            graph[pre].append(course)
        return graph, indeg


    def bfsHelper(self, n, graph, indeg):
        queue = deque()
        for key, val in indeg.items():
            if val == 0:
                queue.append(key)

        taken = 0
        while queue:
            cur_course = queue.popleft()
            taken += 1
            for neighbor_course in graph[cur_course]:
                indeg[neighbor_course] -= 1
                if indeg[neighbor_course] == 0:
                    queue.append(neighbor_course)
        return taken == n
```

▼ Alien Dictionary

There is a new alien language which uses the latin alphabet. However, the order among letters are unknown to you. You receive a list of non-empty words from the dictionary, where words are `sorted lexicographically` by the rules of this new language. Derive the `order` of letters in this language.

Example 1:

```
Input:
[
  "wrt",
  "wrf",
  "er",
  "ett",
  "rftt"
]

Output: "wertf"
```

Example 2:

```
Input:
[
  "z",
  "x"
]

Output: "zx"
```

Example 3:

```
Input:
[
  "z",
  "x",
  "z"
]

Output: ""

Explanation: The order is invalid, so return "".
```

▼ Solution **(Final)**

 Time Complexity: `O(V + E)` | Space Complexity: `O(V + E)`

```
from collections import defaultdict, deque
class Solution:
    def alienOrder(self, words: List[str]) -> str:
        chars = set(char for word in words for char in word)
        indegree = {char : 0 for char in chars}
        graph, status = self.make_graph(words, indegree)
        if not status:
            return ""
        res = ""

       #bfs helper
```

```
        queue = deque([char for char in indegree if indegree[char] == 0])
        while queue:
            cur = queue.popleft()
            res += cur
            for neighbor in graph[cur]:
                indegree[neighbor] -= 1
                if indegree[neighbor] == 0:
                    queue.append(neighbor)
        if len(res) == len(chars):
            return res
        return ""

    def make_graph(self, words, indegree):
        graph = defaultdict(list)
        # graph = {char : [] for char in chars}
        for i in range(1, len(words)):
            # Check invalid cases like ["wrtkj","wrt"] and ["abc","ab"]
            n = len(words[i])
            if len(words[i - 1]) > len(words[i]) and words[i - 1][:n] == words[i]:
                return ({},False)

            for j in range(min(len(words[i - 1]), len(words[i]))):
                if words[i - 1][j] != words[i][j]:
                    graph[words[i - 1][j]].append(words[i][j])
                    indegree[words[i][j]] += 1
                    break
        return (graph, True)
```

▼ Longest Increasing Path in a Matrix

Given an integer matrix, find the length of the longest increasing path.

From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary (i.e. wrap-around is not allowed).

Example 1:

```
Input: nums =
[
  [9,9,4],
  [6,6,8],
  [2,1,1]
]
Output: 4
Explanation: The longest increasing path is [1, 2, 6, 9].
```

Example 2:

```
Input: nums =
[
  [3,4,5],
  [3,2,6],
  [2,2,1]
]
Output: 4
Explanation: The longest increasing path is [3, 4, 5, 6]. Moving diagonally
is not allowed.
```

▼ Solution **(Final)**

 Time Complexity: `O(V + E)` | Space Complexity: `O(V + E)`

```
from collections import defaultdict, deque

class Solution(object):
    def longestIncreasingPath(self, matrix):
        graph, indeg = self.makeGraph(matrix)
        start = [(i, j) for i in range(len(matrix)) for j in range(len(matrix[0])) if indeg[(i, j)] == 0]
        queue = deque(start)
        res = 0
        while queue:
            res += 1
            size = len(queue)
            for _ in range(size):
                x, y = queue.popleft()
                for neighbor in graph[(x , y)]:
                    indeg[neighbor] -= 1
                    if not indeg[neighbor]:
                        queue.append(neighbor)
        return res


    def makeGraph(self, matrix):
```

```
        graph = defaultdict(list)
        indeg = {(i, j) : 0 for i in range(len(matrix)) for j in range(len(matrix[0]))}
        for i in range(len(matrix)):
            for j in range(len(matrix[0])):
                for x, y in [(i + 1, j), (i - 1, j), (i, j + 1), (i, j - 1)]:
                    if 0 <= x < len(matrix) and 0 <= y < len(matrix[0]) and matrix[i][j] < matrix[x][y]:
                        graph[(i, j)].append((x, y))
                        indeg[(x, y)] += 1
        return graph, indeg
```