You have **2** free member-only stories left this month. Sign up for Medium and get an extra one

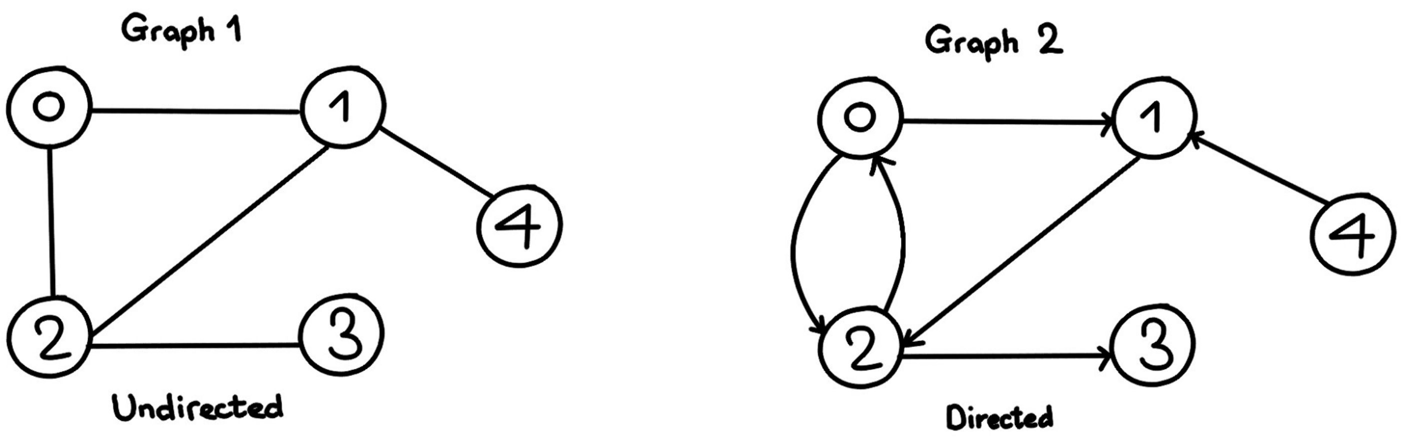
Graph data structure cheat sheet for coding interviews.

Tuan Nhu Dinh Apr 10, 2020 · 6 min read ★

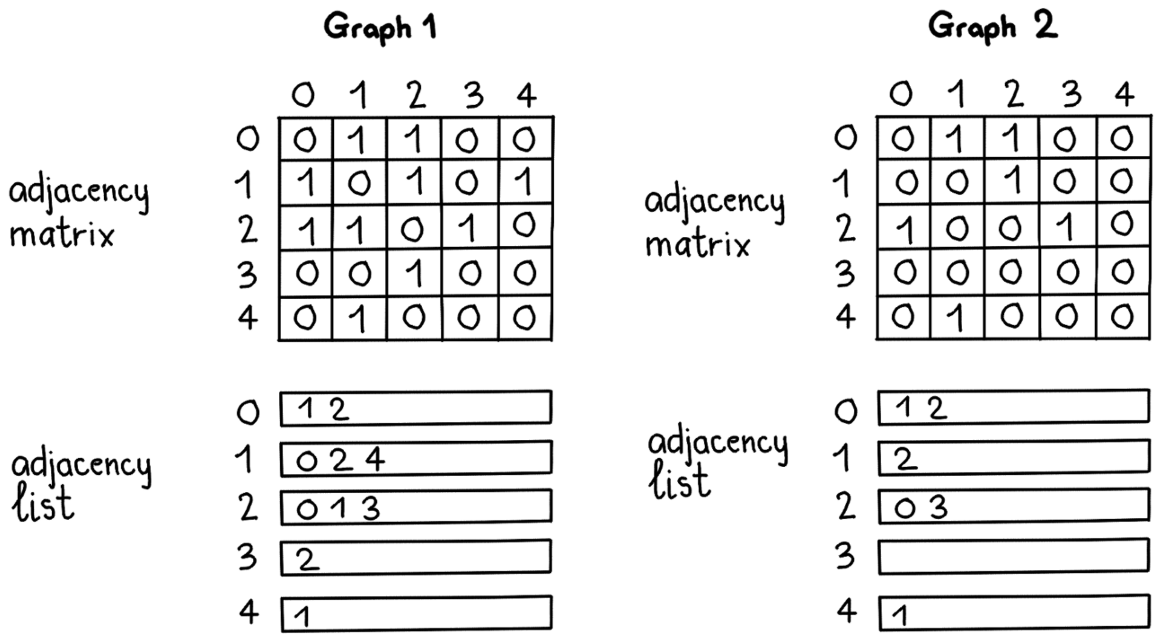
This blog is a part of my "15 days cheat sheet for hacking technical interviews at big tech companies". In this blog, I won't go into detail about graph data structure, but I will summarise must-to-know graph algorithms to solve coding interview questions.

Graph Data Structure

A graph is a non-linear data structure consisting of vertices (V) and edges (E).



lists represents the list of vertices adjacent to each vertex).



In the following sections, let's take a look must-to-know algorithms related to graph data structure. For simplicity, adjacency list representation is used in all the implementation.

1. Breadth First Search (BFS)

```
Input (graph 1): graph = [[1,2], [0,2,4], [0,1,3], [2], [1]], s = 0
Output: 0 1 2 4 3
```

Breadth First Search for a graph is similar to Breadth First Traversal of a tree. However, graphs may contain cycles, so we may visit the same vertex again and again. To avoid that, we can use boolean visited array to mark the visited vertices.

Time complexity is O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.

2. Depth First Search (DFS)

```
Input (graph 1): graph = [[1,2], [0,2,4], [0,1,3], [2], [1]], s = 0
Output: 0 1 2 3 4 (or 0 2 3 1 4)
```

Similar to BFS, we also need to use a boolean array to mark the visited vertices for DFS.

Recursive implementation:

```python
# graph is represented by adjacency list: List[List[int]]
# s: start vertex
def dfs(graph, s):
    # set is used to mark visited vertices
    visited = set()

    def recur(current_vertex):
        print(current_vertex)

        # mark it visited
        visited.add(current_vertex)

        # Recur for all the vertices adjacent to current_vertex
        for v in graph[current_vertex]:
            if v not in visited:
                recur(v)

    recur(s)
```

dfs_recursive.py hosted with ❤ by GitHub view raw

Iterative implementation using stack. Please note that the stack may *contain the same vertex twice,* so we need to check the visited set before printing.

```
    # set is used to mark visited vertices
    visited = set()

    # create a stack for DFS
    stack = []

    # Push vertex s to the stack
    stack.append(s)

    while stack:
        current_vertex = stack.pop()

        # Stack may contain same vertex twice. So
        # we need to print the popped item only
        # if it is not visited.
        if current_vertex not in visited:
            print(current_vertex)
            visited.add(current_vertex)

        # Get all adjacent vertices of current_vertex
        # If an adjacent has not been visited, then push it to stack
        for v in graph[current_vertex]:
            if v not in visited:
                stack.append(v)
```

dfs_iterative.py hosted with ❤️ by GitHub view raw

For both implementations, all the vertices may not be reachable from a given vertex (example Disconnected graph). To do complete DFS traversal, we need to call DFS for every vertex.

Time complexity is O(V+E) where V is the number of vertices in the graph and E is number of edges in the graph.

3. Detect cycle in directed graph

> Given a **directed** graph, return true if the given graph contains at least one cycle, else return false.

DFS can be used to detect a cycle in a Graph. There is a cycle in a graph only if there is a back edge that is from a vertex to itself (self-loop) or to one of its ancestor in DFS stack tree.

```python
# graph is represented by adjacency list: List[List[int]]
# DFS to detect cyclic
def is_cyclic_directed_graph(graph):
    # set is used to mark visited vertices
    visited = set()
    # set is used to keep track the ancestor vertices in recursive stack.
    ancestors = set()

    def is_cyclic_recur(current_vertex):
        # mark it visited
        visited.add(current_vertex)
        # add it to ancestor vertices
        ancestors.add(current_vertex)

        # Recur for all the vertices adjacent to current_vertex
        for v in graph[current_vertex]:
            # If the vertex is not visited then recurse on it
            if v not in visited:
                if is_cyclic_recur(v):
                    return True
            elif v in ancestors:
                # found a back edge, so there is a cycle
                return True

        # remove from the ancestor vertices
        ancestors.remove(current_vertex)
        return False

    # call recur for all vertices
    for u in range(len(graph)):
        # Don't recur for u if it is already visited
        if u not in visited:
            if is_cyclic_recur(u):
                return True
    return False
```

Time complexity is the same as the normal DFS, which is O(V+E).

4. Detect cycle in undirected graph

```
Given an **undirected** graph, return true if the given graph contains
at least one cycle, else return false.

Input (graph 1): graph = [[1,2], [0,2,4], [0,1,3], [2], [1]]
Output: True
```

For undirected graph, we don’t need to keep track of the whole stack tree (compared to directed graph cases). For every vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not a parent of v, then there is a cycle in the graph.

```python
# graph is represented by adjacency list: List[List[int]]
# DFS to detect cyclic
def is_cyclic_undirected_graph(graph):
    # set is used to mark visited vertices
    visited = set()

    def is_cyclic_recur(current_vertex, parent):
        # mark it visited
        visited.add(current_vertex)

        # Recur for all the vertices adjacent to current_vertex
        for v in graph[current_vertex]:
            # If the vertex is not visited then recurse on it
            if v not in visited:
                if is_cyclic_recur(v, current_vertex):
                    return True
            elif v != parent:
                # found a cycle
                return True

        return False

    # call recur for all vertices
    for u in range(len(graph)):
        # Don't recur for u if it is already visited
        return False
```

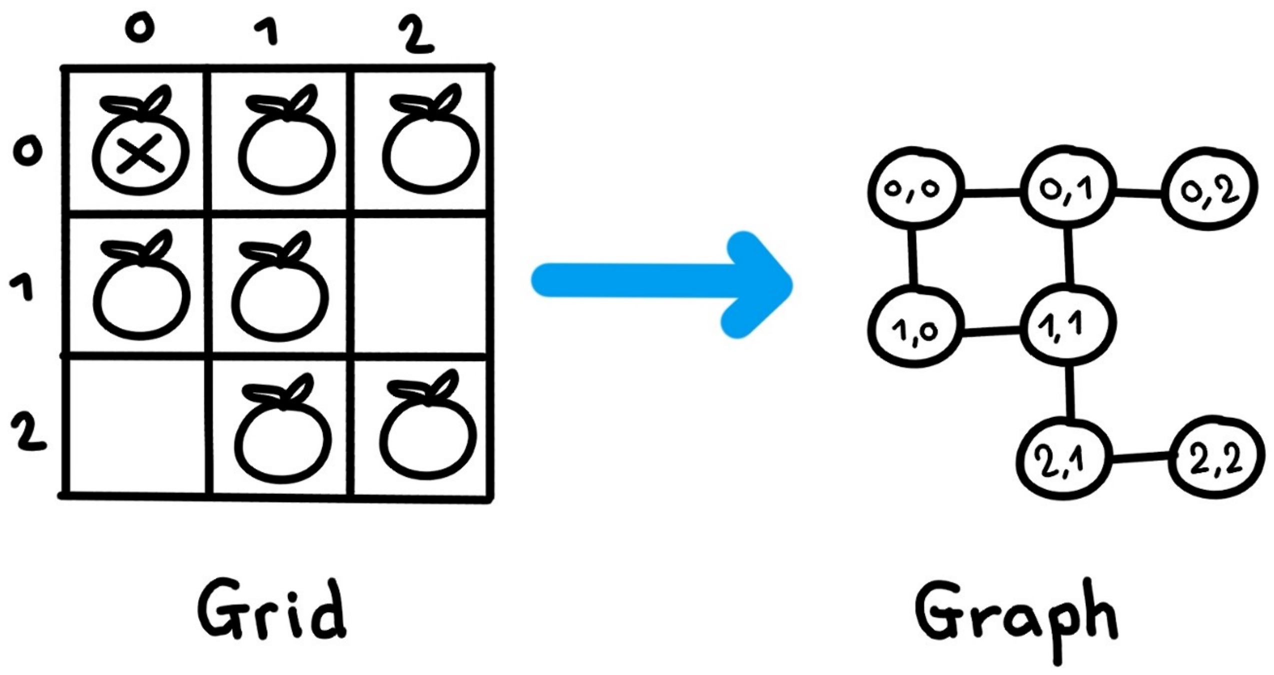
is_cyclic_undirected_graph.py hosted with ❤ by GitHub view raw

Time complexity is the same as the normal DFS, which is O(V+E).

5. BFS with multiple sources

In some problems, you need to start BFS for multiple vertices as well as calculate the travel depth. Let's take a look at a typical problem.

```
Rotting Oranges: https://leetcode.com/problems/rotting-oranges/
```



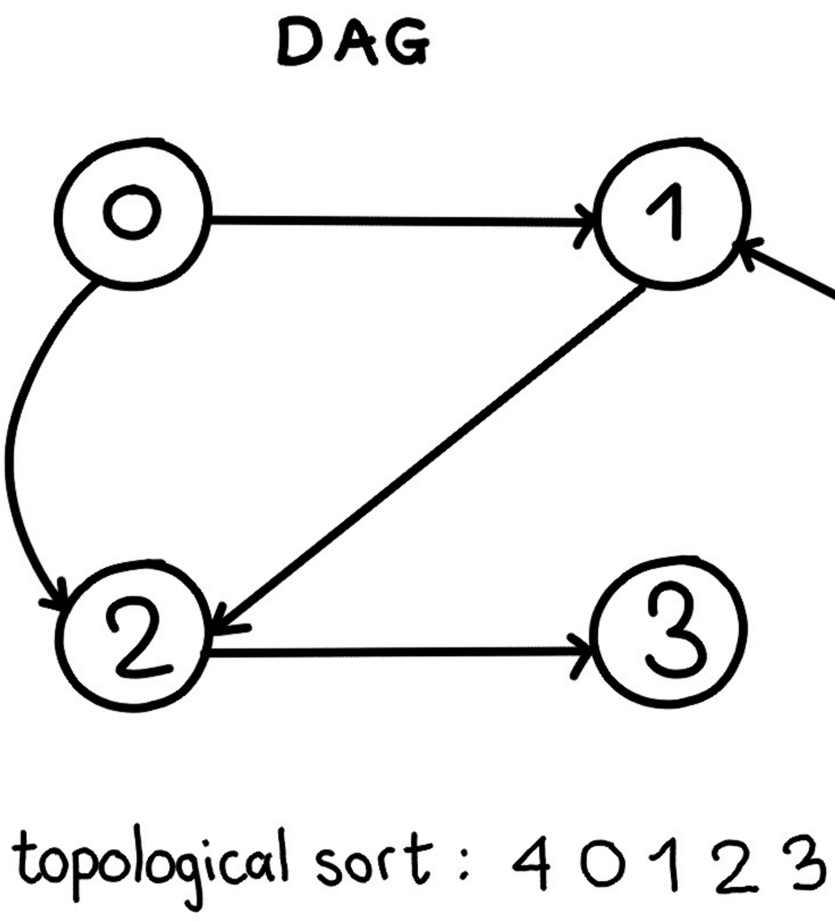
In this question, we use a BFS to model the process. The grid is considered as a graph (an orange cell is a vertex and there are edges to the cell's neighbours). Starting BFS from all rotten oranges with 'depth' 0, and then travel to their neighbours which have 1 more depth. At the end, if we still have unvisited oranges, return -1 as this is impossible to rotten all oranges.

```python
    def orangesRotting(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """

        # corner cases
        if not grid or not grid[0]:
            return 0

        n = len(grid)
        m = len(grid[0])

        queue = deque()
        visited = set()

        # init multiple starting vertices
        for i in range(n):
            for j in range(m):
                if grid[i][j] == 2:
                    # start with depth = 0
                    queue.appendleft((i, j, 0))
                    visited.add((i, j))

        # BFS
        d = 0
        while queue:
            r, c, d = queue.pop()
            for i, j in [[r + 1, c], [r, c + 1], [r - 1, c], [r, c - 1]]:
                # valid neighbours
                if 0 <= i < n and 0 <= j < m:
                    if grid[i][j] == 1 and (i, j) not in visited:
                        queue.appendleft((i, j, d + 1))
                        visited.add((i, j))

        # check if there are still fresh oranges
        for i in range(n):
            for j in range(m):
                if grid[i][j] == 1 and (i, j) not in visited:
                    return -1

        return d
```

Time complexity is O(V+E) = O(the number of cells in grid).

6. Topological sort

Topological sorting is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological sorting is only possible for Directed Acyclic Graph (DAG).



```
Given a DAG, return the topological sorting

Input: graph = [[1,2], [2], [3], [], [1]]
Output: 4 0 1 2 3
```

vertices, then push its value to a stack. At the end, we print out the stack.

```python
# graph is represented by adjacency list: List[List[int]]
# using DFS to find the topological sorting
def topological_sort(graph):
    # using a stack to keep topological sorting
    stack = []

    # set is used to mark visited vertices
    visited = set()

    def recur(current_vertex):
        # mark it visited
        visited.add(current_vertex)

        # Recur for all the vertices adjacent to current_vertex
        for v in graph[current_vertex]:
            if v not in visited:
                recur(v)

        # Push current vertex to stack after travelling to all neighbours
        stack.append(current_vertex)

    # call recur for all vertices
    for u in range(len(graph)):
        # Don't recur for u if it is already visited
        if u not in visited:
            recur(u)

    # print content of the stack
    print(stack[::-1])
```

topological_sort.py hosted with ❤ by **GitHub** view raw

Time complexity is the same as the normal DFS, which is O(V+E).

7. Shortest path in an unweighted graph

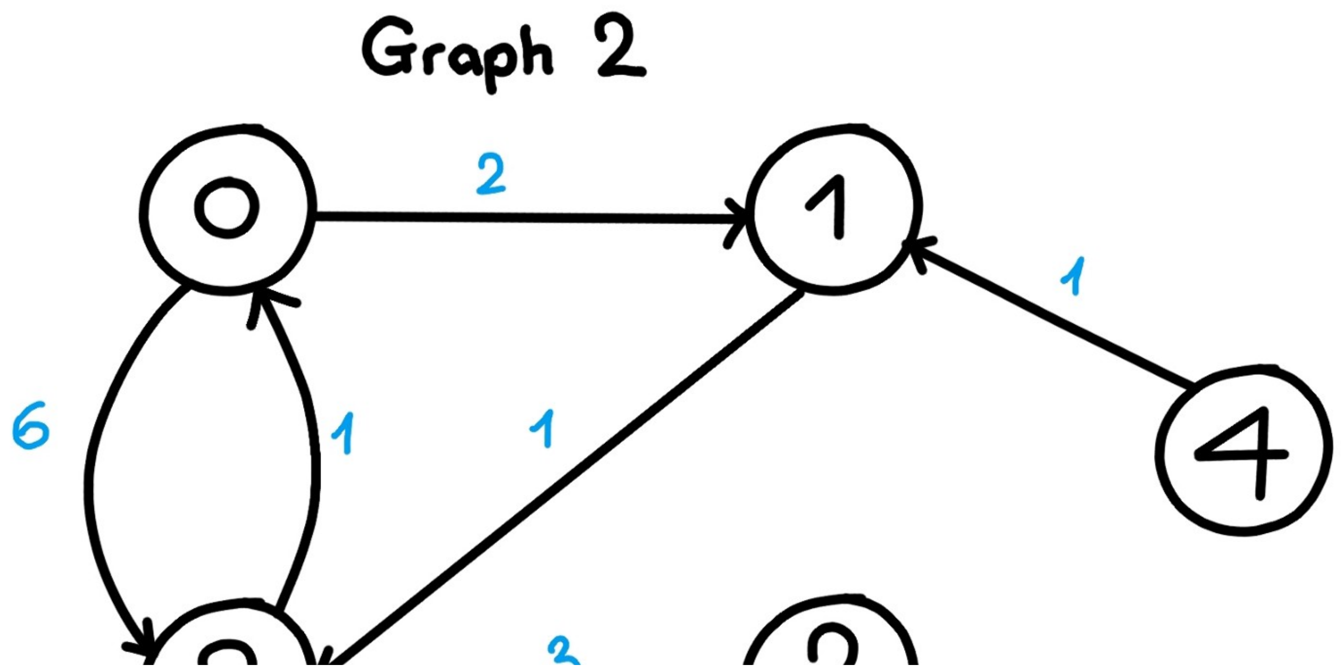
```
Input (graph 1): graph = [[1,2], [0,2,4], [0,1,3], [2], [1]], s=4,
d=0
Output: 4 1 0

Input (graph 2): graph = [[1,2], [2], [0, 3], [], [1]], s=1, d=0
Output: 1 2 0
```

For this question, we use BFS and keep storing the predecessor of a given vertex while doing the breadth first search. At the end, we use the predecessor array to print the path.

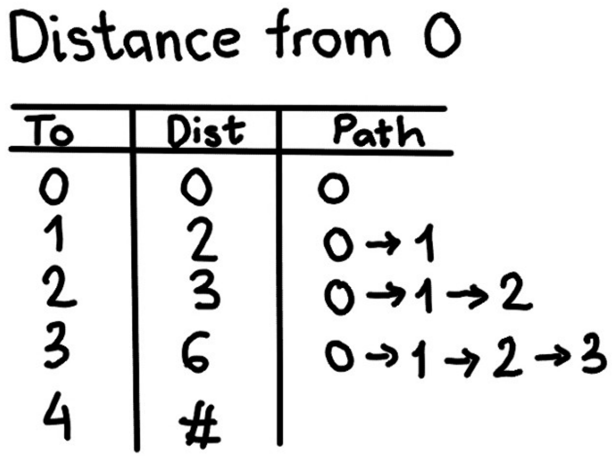
Time complexity is the same as normal BFS, which is O(V+E).

7. Shortest path in a weighted graph

```
Given a graph and a source vertex in graph, find shortest distances
from source to all vertices in the given graph.

Input: graph = [[[1, 2], [2, 6]], [[2, 1]], [[0, 1], [3, 3]], [],
[[1, 1]]], s=0
Output: [0, 2, 3, 6, inf]
```



Get started Open in app



Distance from 0

To	Dist	Path
0	0	0
1	2	0→1
2	3	0→1→2
3	6	0→1→2→3
4	#	

In this problem, we deal with weighted graph (a real number is associated with each edge of graph). The graph is represented as an adjacency list whose items are a pair of target vertex & weight. Dijkstra's algorithm is used to find the shortest path from a starting vertex to other vertices. The algorithm works for both directed or undirected graph as long as it *does not have negative weight on an edge.*

You can read more about Dijkstra's algorithm at here. The below is my implementation using priority queue.

The above implementation only returns the distances. You can add a predecessor array (similar to shortest path in an unweighted graph) to print out the path. Time complexity is O(V + VlogE), where V is number of vertices in the graph and E is the number of edges in the graph.

Recommended questions

You can practice the graph data structure with the following questions:

1. Is Graph Bipartite?

2. Clone Graph

3. Course Schedule

4. Course Schedule II

6. Number of Connected Components in an Undirected Graph

7. Graph Valid Tree

8. Reconstruct Itinerary

9. Cheapest Flights Within K Stops (hint: Dijkstra's algorithm)

10. Alien Dictionary (hints: topology sorting)

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

Get this newsletter

You'll need to sign in or create an account to receive this newsletter.

Coding Interviews Graph Algorithms Computer Science Algorithms Data Structures