Published in Towards Data Science

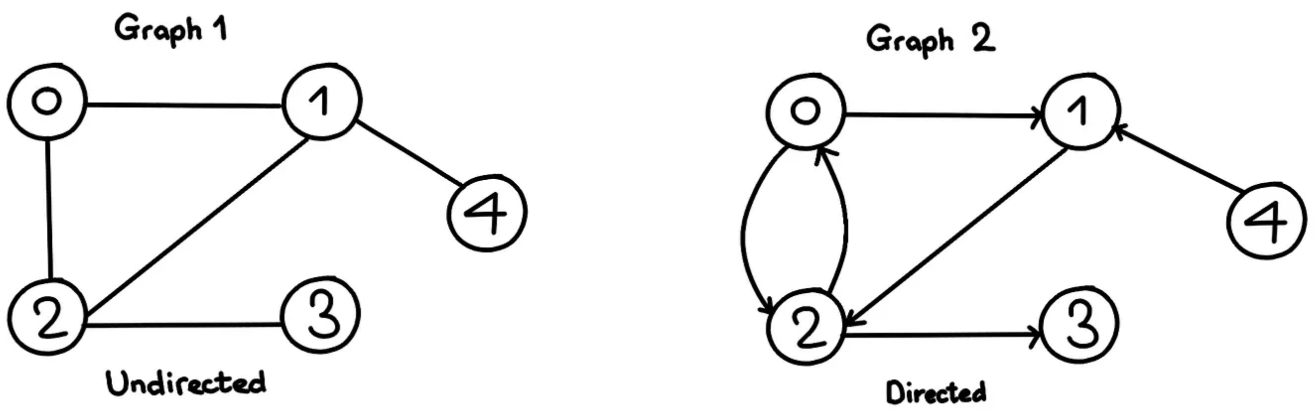
Tuan Nhu Dinh Follow
Apr 10, 2020 · 6 min read · · Listen

Graph data structure cheat sheet for coding interviews.

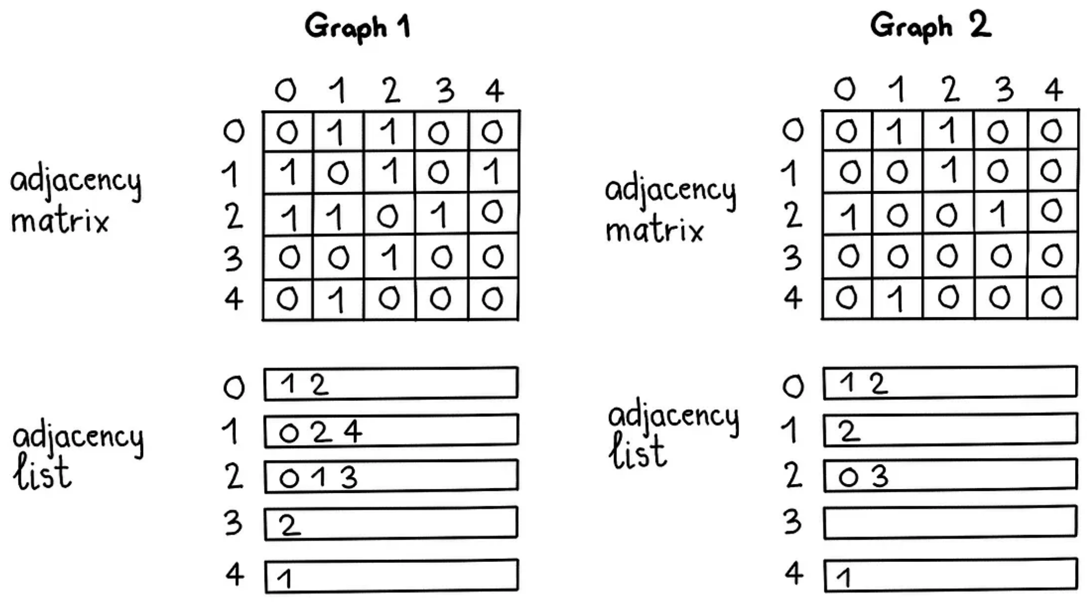
This blog is a part of my "15 days cheat sheet for hacking technical interviews at big tech companies". In this blog, I won't go into detail about graph data structure, but I will summarise must-to-know graph algorithms to solve coding interview questions.

Graph Data Structure

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



The most commonly used representations of a graph are adjacency matrix (a 2D array of size V x V where V is the number of vertices in a graph) and adjacency list (an array of lists represents the list of vertices adjacent to each vertex).



Graph 1 — adjacency matrix

	0	1	2	3	4
0	0	1	1	0	0
1	1	0	1	0	1
2	1	1	0	1	0
3	0	0	1	0	0
4	0	1	0	0	0

Graph 1 — adjacency list

0	1 2
1	0 2 4
2	0 1 3
3	2
4	1

Graph 2 — adjacency matrix

	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	0	0
2	1	0	0	1	0
3	0	0	0	0	0
4	0	1	0	0	0

Graph 2 — adjacency list

0	1 2
1	2
2	0 3
3	
4	1

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

```
from collections import deque

```

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

    # create a queue for BFS
    queue = deque()

    # Mark the start vertex as visited and enqueue it
    visited.add(s)
    queue.appendleft(s)

    while queue:
        current_vertex = queue.pop()
        print(current_vertex)

        # Get all adjacent vertices of current_vertex
        # If a adjacent has not been visited, then mark it
        # visited and enqueue it
        for v in graph[current_vertex]:
            if v not in visited:
                visited.add(v)
                queue.appendleft(v)
```

bfs.py hosted with ❤ by GitHub view raw

The above code traverses only the vertices reachable from a given source vertex. To do complete BFS traversal for disconnected graphs, we need to call BFS for every vertex.

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 = [[1 430 0,1,3], [2], [1]], s = 0
Output: 0 1 2 3 4 (or 0 2 3
```

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

Recursive implementation:

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.

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.

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

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.

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
```

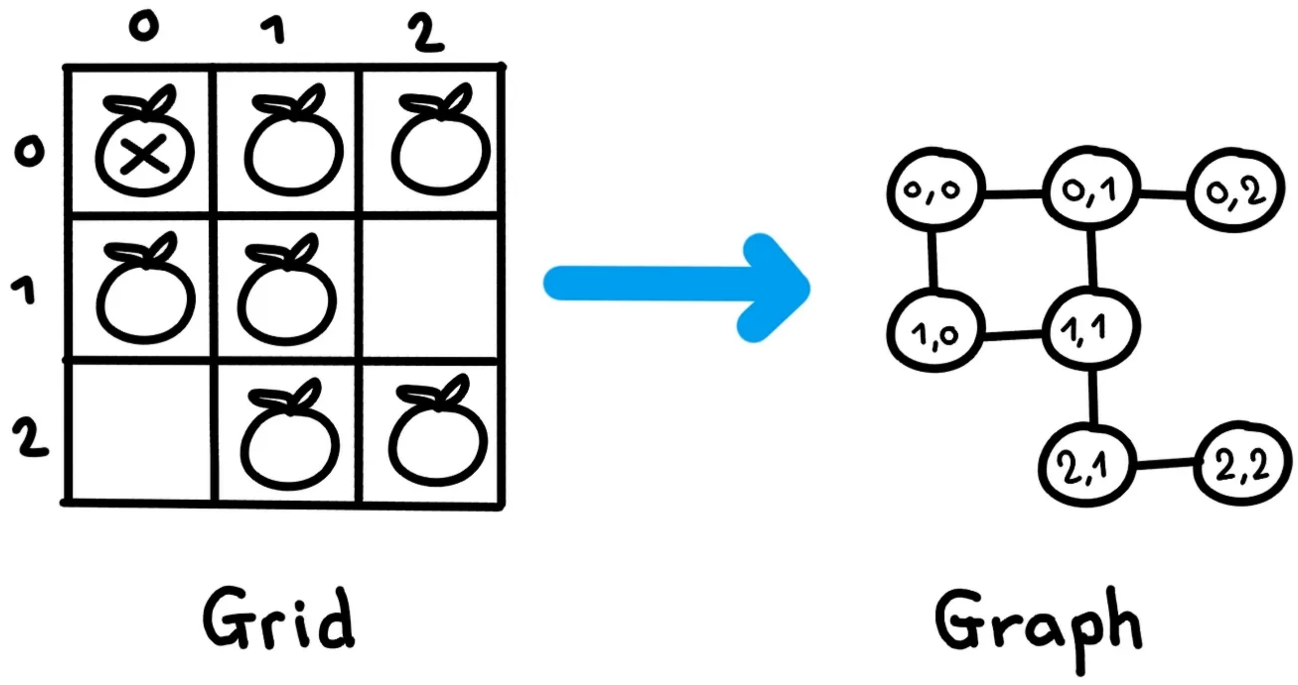
For undirected graph, we don’t need to keep track of the whole stack tree (compared to directed graph cases). For every vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not a parent of v, then there is a cycle in the graph.

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/
```

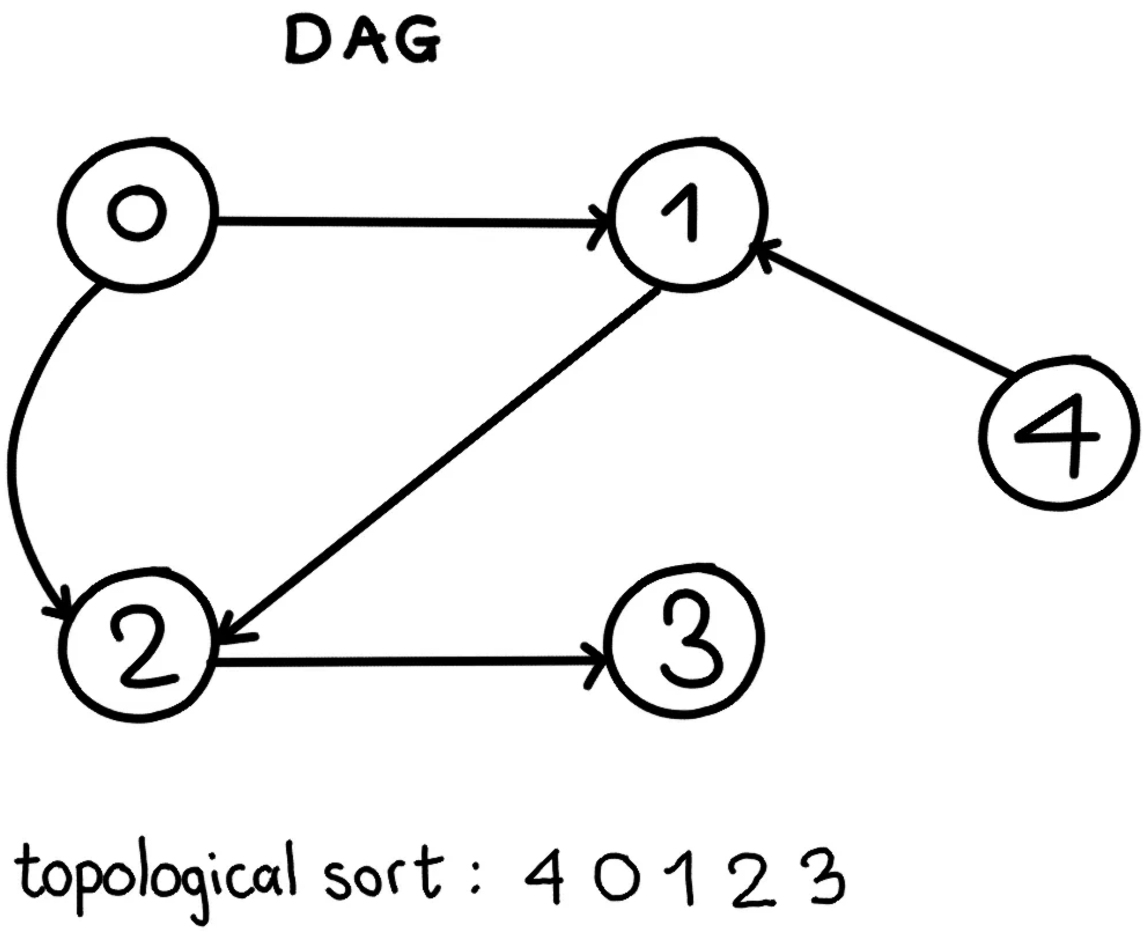


In this question, we use a BFS to model the process. The grid is considered as a graph (an orange cell is a vertex and there are edges to the cell's neighbours). Starting BFS from all rotten oranges with 'depth' 0, and then travel to their neighbours which have 1 more depth. At the end, if we still have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

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
```

In normal DFS, we print the vertex at the beginning of the recursive function. To find the topological sorting, we modify DFS to make it first recursively call for all its adjacent vertices, then push its value to a stack. At the end, we print out the stack.

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

7. Shortest path in an unweighted graph

```
Given a unweighted graph, a source and a destination, we need to find
shortest path from source to destination.

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
```

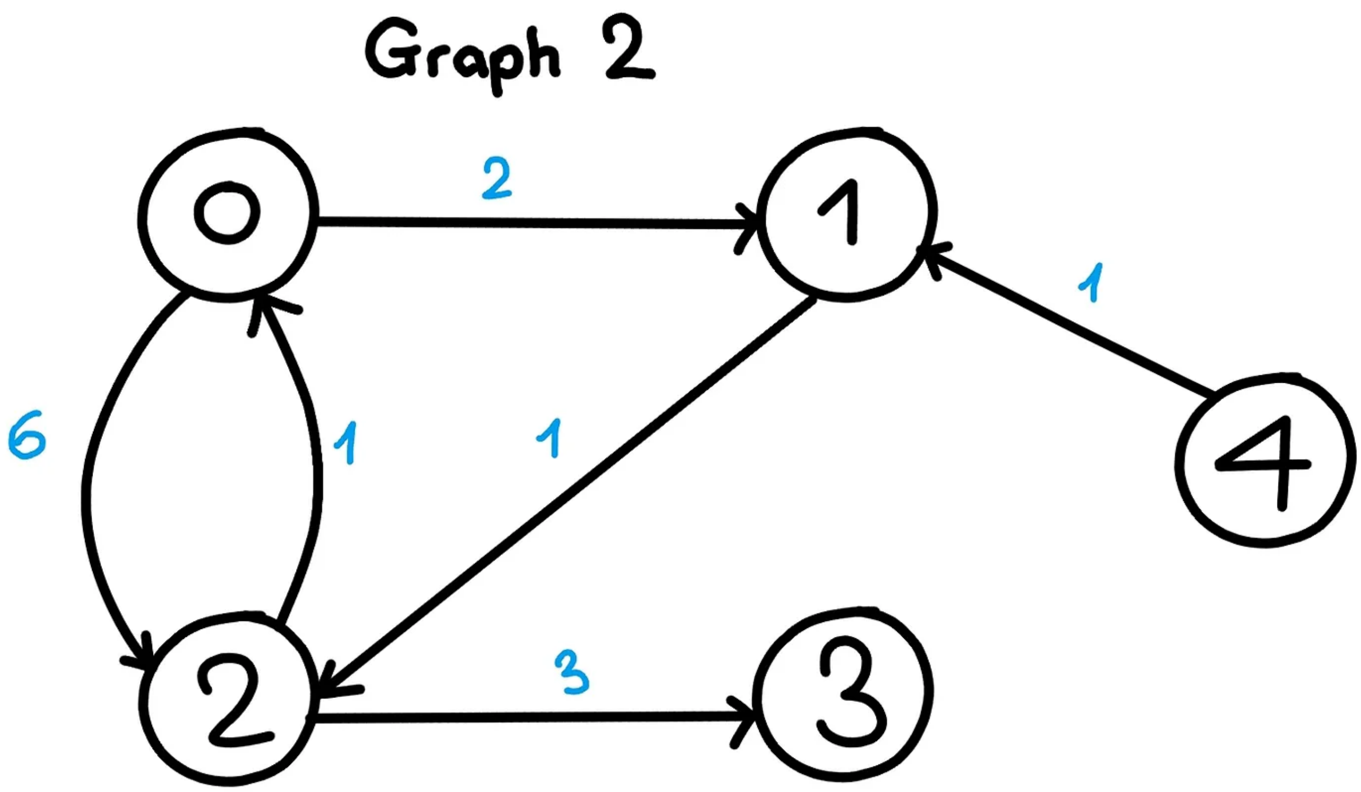
For this question, we use BFS and keep storing the predecessor of a given vertex while doing the breadth first search. At the end, we use the predecessor array to print the path.

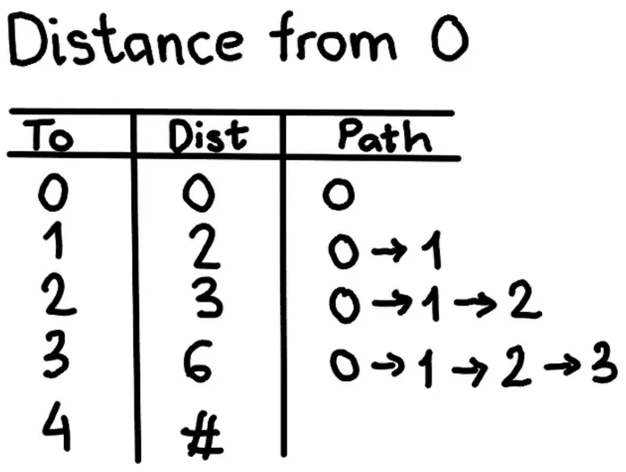
Time complexity is the same as normal BFS, which is O(V+E).

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]
```




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
5. Number of Islands
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)

Coding Interviews Graph Algorithms Computer Science Algorithms

Data Structures