Intermediate Software Engineering

Intermediate-Software-Engineering-Summer-2022 (@ Intermediate)
Personal Member ID#: 52751

Graphs UMPIRE Cheat Sheet

Definition

Graphs are one of the most prevalent data structures in computer science. It's a powerful data structure that's utilized to represent relationships between different types of data. In a graph, each data point is stored as a node and each relationship between those data points is represented by an edge. For example, a social network is considered a graph in which each person is considered a node and a friendship between two people is considered an edge.

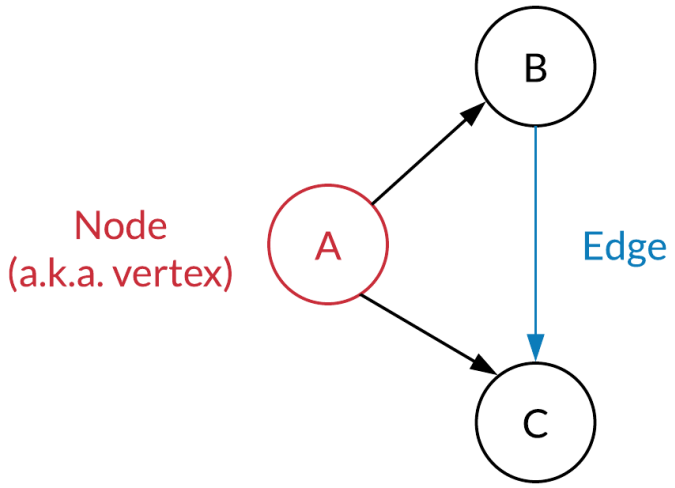
Graphs are best utilized for problems in which there are binary relationships between objects. Once a problem can be represented as a graph, the problem can generally be solved based off of one of the key graph algorithms. For interviews, it is vital to know how to implement a graph, basic graph traversals (BFS, DFS) and how to sort topologically the graph.

Graph Terminology

Graph components

Graphs consist of a set of..

- **vertices**, which are also referred to as nodes
    - Nodes that are directly connected by an edge are commonly referred to as **neighbors**.
- **edges**, connections between pairs of vertices



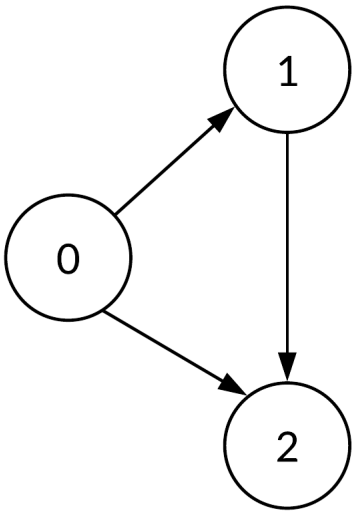
Graph types

Directed & undirected graphs

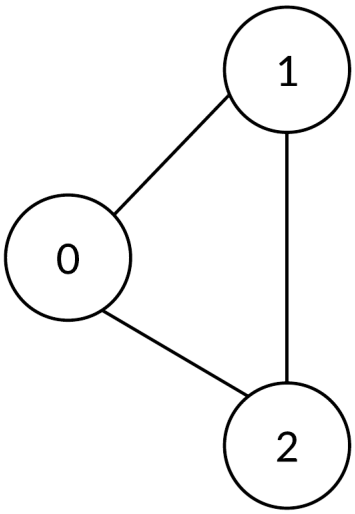
A **directed** graph is a graph that in which all edges are associated with a direction. An example of a directed edge would be a one way street.

An **undirected** graph is a graph in which all edges do not have a direction. An example of this would be a friendship!

Directed Graph



Undirected Graph



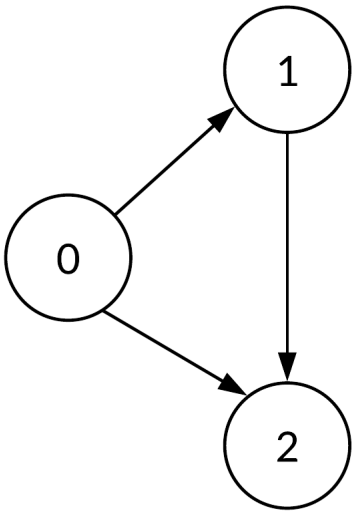
Cyclic & acyclic graphs

Before going over the what cyclic and acyclic graphs are, there are two key terms to cover: **path** and **cycle**. A **path** is a sequence of vertices connected by edges and a **cycle** a path whose first and last vertices are the same.

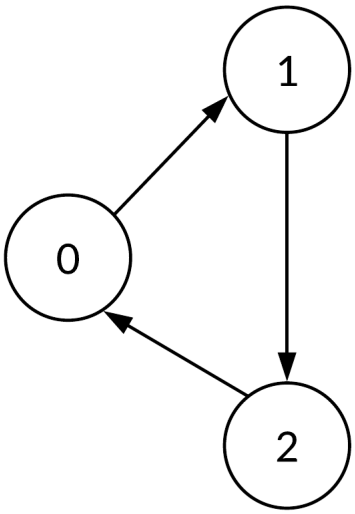
A **cyclic** graph means that there contains a least one cycle within the graph.

An **acyclic** graph has no cycles within it.

Acyclic Graph



Cyclic Graph

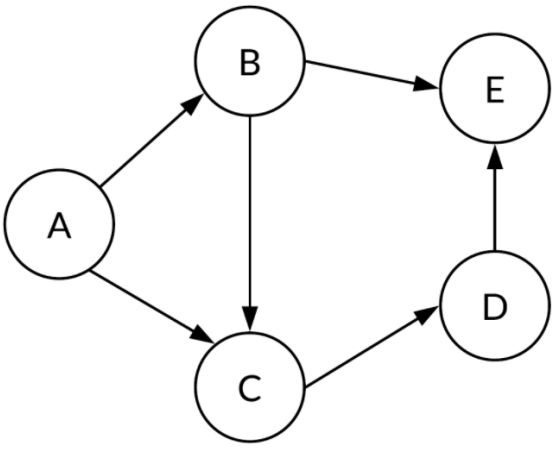


A commonly used phrase when referring to graphs is a **directed acylic graph (DAG)**, which is a directed graph in which there are *no* cycles. In a DAG, these two terms are commonly used to denote nodes with special properties:

- **Sink** nodes have no outgoing edges, only incoming edges
- **Source** nodes only have outgoing edges, no incoming edges

Graph representations

In this section we discuss two potential ways graphs are represented in text. For both representations we will use the following graph example.



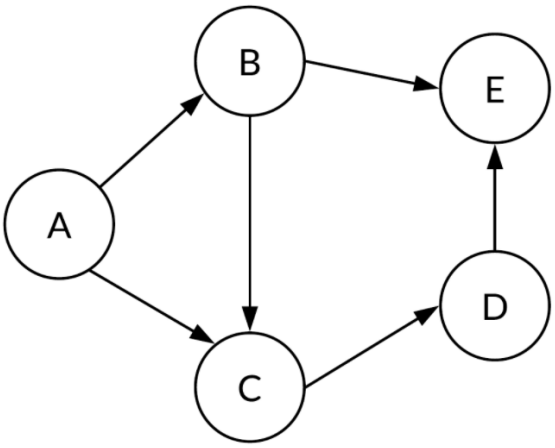
The graph above have the following qualities:

- Total of 5 nodes with the data values A, B, C, D, and E.
- Node A points towards two nodes: Node B and Node C
- Node B points towards two nodes: Node C and Node E

- Node C points towards one node: Node D
- Node D points towards one node: Node E
- Node E points towards zero nodes.

Adjacency lists

Adjacency list is the most common way to represent graphs. With this approach of representing a graph, each node stores a list of its adjacent vertices. For undirected graphs, each edge from *u* to *v* would be stored twice: once in *u*'s list of neighbors and once in *v*'s list of neighbors.

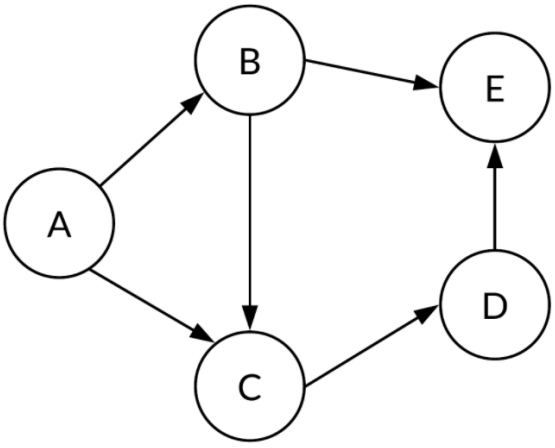



```
Adjacency list:

graph = {
  'A': ['B', 'C']
  'B': ['C', 'E']
  'C': ['D']
  'D': ['E']
}
```

Edge sets/ lists

An **edge set** simply represents a graph as a collection of all its edges.

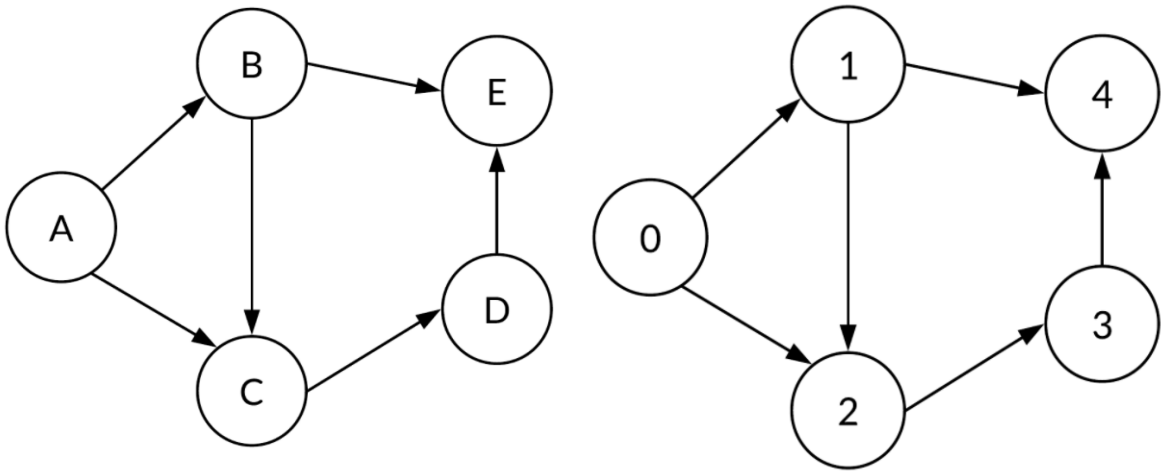


```
Edge List:

graph = [
  ('A', 'B'),
  ('B', 'C'),
  ('B', 'E'),
  ('C', 'D'),
  ('D', 'E')
]
```

Adjacency matrix

An **adjacency matrix** represents a graph with *n* nodes as a *n* by *n* boolean matrix, in which matrix[*u*][*v*] is set to true if an edge exists from node *u* to node *v*.



In this example, we assign each node an index. We will assign the following indices:

- A = 0
- B = 1
- C = 2
- D = 3
- E = 4

	0 (A)	**1 (B)**	**2 (C)**	**3 (D)**	**4 (D)**
0 (A)	0	1	1	0	0
1 (B)	0	0	1	0	1
2 (C)	0	0	0	1	0
3 (D)	0	0	0	0	1
4 (E)	0	0	0	0	0

```
Adjacency Matrix

graph = [
  [0, 1, 1, 0, 0],
  [0, 0, 1, 0, 1],
  [0, 0, 0, 1, 0],
  [0, 0, 0, 0, 1],
  [0, 0, 0, 0, 0]
]
```

This representation of a graph is efficient for checking if an edge exists between a pair of vertices. However, it may be less efficient for search algorithms because it requires iterating through all the nodes in the graph to identify a node's neighbors.

Runtime Analysis

Below is a chart of the most common graph operations and their runtimes for each of the graph representations. In the chart below, *V* represents the number of vertices in the graph and *E* represents the number of edges in the graph.

Representation	Getting all adjacent edges for a vertex	Traversing entire graph	hasEdge(u, v)	Space
Adjacency matrix	O(V)	$O(V^2)$	O(1)	$O(V^2)$
Edge Set	O(E)	O(E)	O(E)	O(E)
Adjacency List	O(1)	O(V + E)	O(max number of edges a vertex has)	O(E + V)

Credit: UC Berkeley data structures course

U-nderstand

What are some common questions we should ask our interviewer?

- Are there memory constraints?
- What's the required time complexity?
- What kind of data will the inputs be?
- Can I assume all the inputs will be valid?
- What if the input is empty?
- What should we return if there is no solution to the problem?
- What should we return if there are multiple solutions to the problem?

M-atch

Are there any special techniques that we can use to help make this easier?

In the Matching step of UMPIRE, you want to think about common patterns and tricks that could apply to this problem.

- **Edge List**
  - Many graph problems provide us an edge list as an input. If we are looking for the number of connected components, run a union-find on the edge list and we're done. If we need to do literally anything else, convert it to an adjacency list. Think about the tradeoff between representing the graph as an adjacency list (`O(V+E)` traversal time and space) vs. adjacency matrix (`O(V²)` time and space). If the graph is dense, the matrix is fine. Graphs are usually large and sparse so we opt for the adjacency list.
- **Union Find**
  - Identify if problems talks about finding groups or components.
- **Breadth First Search**
  - Start BFS from nodes from which shortest path is asked for.
- **Depth First Search**
  - Depth-first search is a graph traversal algorithm which explores as far as possible along each branch before backtracking. A stack is usually used to keep track of the nodes that are on the current search path. This can be done either by an implicit recursion stack, or an actual stack data structure.
- **Adjacency Matrix**
  - The last pattern is an adjacency matrix and is not that common, so I'll keep this section short.
  - The key here is to remember to iterate over the rows and the columns using Python's enumerate. The index of the row will represent node_1 and the index of the column will represent node_2. The value at the matrix[row][col] will tell us whether or not there is an edge connecting these two nodes.
- **Graph coloring/Bipartition**
  - Problems asks to check if its possible to divide the graph nodes into 2 groups
- **Topological Sort**
  - Check if its directed acyclic graph and we have to arrange the elements in an order in which we need to select the most independent node at first.
- **Adjacency List**
  - Once we've converted our edge list to an adjacency list, we arrive at pattern 2. There are a few sub patterns within the adjacency list pattern.
  - Is there a Path? BFS/DFS
  - Shortest Path w/ Unweighted Edges? BFS
  - Shortest Path w/ Weighted Edges? BFS w/ Max Heap
  - Ordering the nodes? Topological Sort
  - Detecting a cycle? Topological Sort or BFS/DFS
- **Dijkstra Algorithm**

    - Used only if weights are non-negative
    - Similar to BFS but has below difference
    - Used Priority Queue with Integer Array instead of Queue with Integer
    - Used Distance array instead of boolean visited array.
- **Prim's Algorithm**
    - Start with any vertex. Use Priority Queue to process the smallest edge.
    - Use visited array or distance array.
    - Difference between Prims and Dijkstra is “Don’t add current vertex distance to calculate neighbor distance”.

P-lan/Pseudocode

- Can you create any **magic** helper methods that would simplify the solution?
- Talk through different approaches you can take, and their tradeoffs
- Be able to verbally describe your approach and explain how an example input would produce the desired output

Tips:

- Try to avoid nested loops
    - This is usually a brute force solution and is $O(n^2)$ time complexity
- Look out for corner cases: Empty graph, Graph with one or two nodes, Disjoint graphs, Graph with cycles
- Common graph representations: Adjacency matrix, Adjacency list, Hash table of hash tables

E-valuate

Graph Algorithms Time Complexity

	Average Case	Worst Case
Dijkstra's Algorithm	`O(E log V)`	`O(V^2)`
Prim's Algorithm	`O(E log V)`	`O(V^2)`
Topological Sort	`O(V + E)`	`O(V + E)`
Bellman-Ford Algorithm	`O(V + E)`	`O(V + E)`
Floyd-Warshall Algorithm	`O(V^3)`	`O(V^3)`