Interview Skills

Graph Algorithms

Richard Morrill

Fordham University CS Society

Wednesday, January 9th 2019

What is a Graph?

What is a Graph?

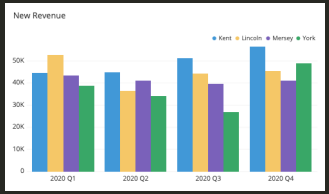


Figure 1: Not This Kind of Graph!

CS SOCIETY
FORDHAM UNIVERSITY

What is a Graph?

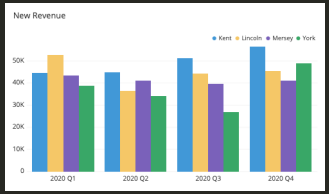


Figure 1: Not This Kind of Graph!

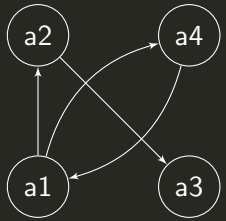


Figure 2: This Kind of Graph!

CS SOCIETY
FORDHAM UNIVERSITY

Characteristics of a Graph

- Set of Nodes S
- Set of Edges $E \subset S^{2*}$

*This just means every possible ordered pair of nodes.

Characteristics of a Graph

- Set of Nodes S
- Set of Edges $E \subset S^{2*}$
- Can represent a wide variety of real-world problems (such as?)

*This just means every possible ordered pair of nodes.

CS SOCIETY
FORDHAM UNIVERSITY

Characteristics of a Graph

- Set of Nodes S
- Set of Edges $E \subset S^{2*}$
- Can represent a wide variety of real-world problems (such as?)
  - Islands & Bridges
  - Network Connections†
  - Intersections and Streets

*This just means every possible ordered pair of nodes.

†You'll see this come up when you take a networking class.

CS SOCIETY
FORDHAM UNIVERSITY

Characteristics of a Graph

- Set of Nodes S
- Set of Edges $E \subset S^{2*}$
- Can represent a wide variety of real-world problems (such as?)
  - Islands & Bridges
  - Network Connections†
  - Intersections and Streets
- Position of nodes is for communication only
- Edges may have a "weight" assigned to them, which may represent distance in some cases

*This just means every possible ordered pair of nodes.

†You'll see this come up when you take a networking class.

CS SOCIETY
FORDHAM UNIVERSITY

Adjacency Matrix

- Very useful for mathematical proofs

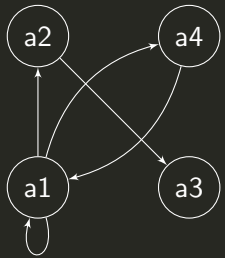


Figure 3: Visual Representation of a Graph

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Figure 4: The Same Graph as 3, in an Adjacency Matrix

CS SOCIETY
FORDHAM UNIVERSITY

Adjacency Matrix

- Very useful for mathematical proofs

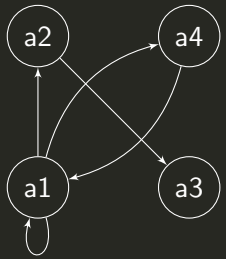


Figure 3: Visual Representation of a Graph

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Figure 4: The Same Graph as 3, in an Adjacency Matrix

- Memory usage causes issues when used in programs.

Representations of Graphs in Code

- A graph is an **abstract data type (ADT)**

Representations of Graphs in Code

- A graph is an **abstract data type (ADT)**
  - The operations you can perform on a graph are consistently defined.
  - The actual way data is stored may vary from implementation to implementation.

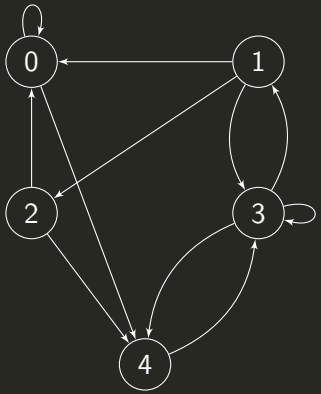
Representations of Graphs in Code

- A graph is an **abstract data type (ADT)**
  - The operations you can perform on a graph are consistently defined.
  - The actual way data is stored may vary from implementation to implementation.
- For higher level problems, you might use a graph library that provides a consistent interface to access and manipulate graphs.
- For now, though, it's important to show you know how to actually work with low-level implementations.

Representations of Graphs in Code

- A graph is an **abstract data type (ADT)**
  - The operations you can perform on a graph are consistently defined.
  - The actual way data is stored may vary from implementation to implementation.
- For higher level problems, you might use a graph library that provides a consistent interface to access and manipulate graphs.
- For now, though, it's important to show you know how to actually work with low-level implementations.
- Graphs are most often represented as:
  - Node Lists
  - Edge Lists

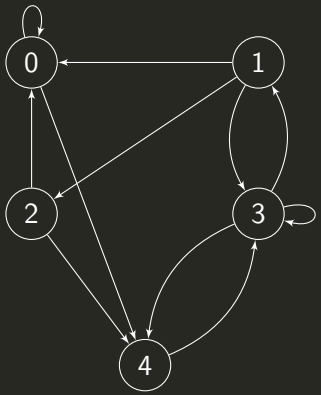
Node List




```
1  vector<vector<int>> graph = {
2      {0, 4},
3      {2, 3, 0},
4      {0, 4},
5      {3, 1, 4},
6      {3}
7  };
```

CS SOCIETY
FORDHAM UNIVERSITY

Edge List




```
vector<vector<int>> graph = {
    {0, 0},
    {0, 4},
    {1, 2},
    {1, 3},
    {1, 0},
    {2, 0},
    {2, 4},
    {3, 3},
    {3, 1},
    {3, 4},
    {4, 3}
};
```

CS SOCIETY
FORDHAM UNIVERSITY

Graph Operations

- Part of the graph ADT
- Theoretical functions you can perform on nodes

Graph Operations

- Part of the graph ADT
- Theoretical functions you can perform on nodes
- View Operations
  - adjacent(x,y): bool
  - neighbors(x): list of edges
  - get_vertex_value(x): value type
  - get_edge_value(...): value type[‡]

[‡]Argument is either edge key or the two vertices it connects, depending on representation.

CS SOCIETY
FORDHAM UNIVERSITY

Graph Operations

- Part of the graph ADT
- Theoretical functions you can perform on nodes
- View Operations
  - adjacent(x,y): bool
  - neighbors(x): list of edges
  - get_vertex_value(x): value type
  - get_edge_value(...): value type[‡]
- Modify Operations
  - set_vertex_value(x, v): void
  - set_edge_value(..., v): void
  - add / remove edges / vertices

[‡]Argument is either edge key or the two vertices it connects, depending on representation.

CS SOCIETY
FORDHAM UNIVERSITY

Properties of a Graph

- This is information you might be given as part of an interview prompt.
- If you misinterpret it, you're basically guaranteed to fail.

Properties of a Graph

- This is information you might be given as part of an interview prompt.
- If you misinterpret it, you're basically guaranteed to fail.
- Cyclic / Acyclic
  - It it possible to end up back where you started without re-using edges?
  - Most of the time problems that involve acyclic graphs are phrased as tree problems, so if you aren't told otherwise, assume you have to deal with cycles in your code.

Properties of a Graph

- This is information you might be given as part of an interview prompt.
- If you misinterpret it, you're basically guaranteed to fail.
- Cyclic / Acyclic
  - It it possible to end up back where you started without re-using edges?
  - Most of the time problems that involve acyclic graphs are phrased as tree problems, so if you aren't told otherwise, assume you have to deal with cycles in your code.
- Directed / Undirected
  - Are the edges directional?
  - All the graphs I showed thus far were directed, but by removing information you could easily make them undirected.

Properties of a Graph

- This is information you might be given as part of an interview prompt.
- If you misinterpret it, you're basically guaranteed to fail.
- Cyclic / Acyclic
  - It it possible to end up back where you started without re-using edges?
  - Most of the time problems that involve acyclic graphs are phrased as tree problems, so if you aren't told otherwise, assume you have to deal with cycles in your code.
- Directed / Undirected
  - Are the edges directional?
  - All the graphs I showed thus far were directed, but by removing information you could easily make them undirected.
- Values on Edges / Nodes
  - Sometimes all that really matters is the connections, sometimes you have values attached.
  - For e.g. calculating the travel time between two cities on a graph representing a map.
  - Representation in code will get more complex, but luckily you'll usually be given starter code.

Most Basic Graph Algorithm: Depth First Search

- Simple and straightforward (although not efficient) way to identify any path between two nodes.
- Path is not guaranteed to be optimal by any metric.

Most Basic Graph Algorithm: Depth First Search

- Simple and straightforward (although not efficient) way to identify any path between two nodes.
- Path is not guaranteed to be optimal by any metric.

1. Pick an arbitrary node.
2. Perform whatever you need to do, returning now if end condition is met.
3. For each node to which this node is connected, recurse with that node as the starting node.
4. Once function has been run on all child nodes, return result back up the stack.

Problem 1: Flower Garden

You have N gardens, labeled 1 to N - 1. In each garden, you want to plant one of 4 types of flowers (labeled 0 to 3). You are supplied with an array paths such that paths[i] = [x,y] means that there is a bidirectional path between nodes x and y. You may assume that no garden has more than 3 paths connecting to it.

Your task is to determine which type of flower to plant in each garden, such that no two connected gardens have the same flower. Your code should return an array of size N, where each element represents the flower planted in each garden.

Solution 1 (Inneficient but Easy to Understand)

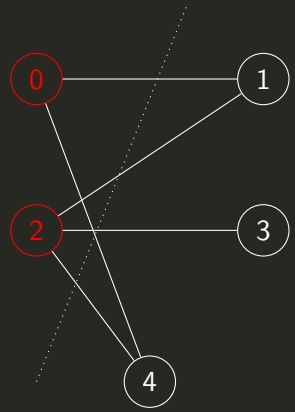
```cpp
vector<int> gardenNoAdj(int N, vector<vector<int>>& paths) {
    vector<int> result(N, -1);
    for (int i = 1; i <= N; ++i) {
        int color = 1;
        for (; color <= 4; ++color) {
            bool conflict = false;
            for (auto& edge : paths) {
                if (edge[0] == i) {
                    if (result[edge[1] - 1] == color) {
                        // Conflict found for this color
                        conflict = true;
                        break;
                    }
                }
                if (edge[1] == i) {
                    if (result[edge[0] - 1] == color) {
                        // Conflict found for this color
                        conflict = true;
                        break;
                    }
                }
            }
            if (!conflict) {
                break;
            }
        }
        result[i - 1] = color;
    }
    return result;
}
```

Problem 2: Bipartite

Given an undirected graph, determine whether it is bipartite. A bipartite graph is simply one that can be split into two sets of nodes such that no nodes in the same set are connected to each other.

In more graphical terms, this means you can draw a line that crosses every edge.

Your function should take input as a list of nodes and return a boolean. [§]

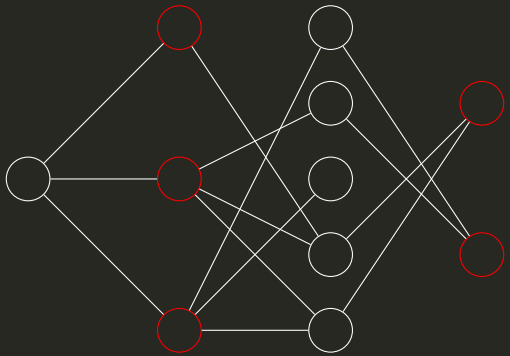


[§] `bool is_bipartite(const vector<vector<int>>& graph)`

CS SOCIETY
FORDHAM UNIVERSITY

Solution Problem 2

There about a million ways you could have coded this, so I'll explain the theory behind the solution.



You create a list that stores the "color" of each node: red, black, or uncolored.

CS SOCIETY
FORDHAM UNIVERSITY

Solution Problem 2 Continued

You run the following function for each node:

1. Color the node the opposite of the previous node visited (for 1st just choose arbitrarily).
2. Check each connecting node, if any match color of current node, return false.
3. If every node is either uncolored or a different color, recurse on each unvisited node (DFS).
4. If every node has been visited without conflict, return true.

Solution Problem 2 Continued

You run the following function for each node:

1. Color the node the opposite of the previous node visited (for 1st just choose arbitrarily).
2. Check each connecting node, if any match color of current node, return false.
3. If every node is either uncolored or a different color, recurse on each unvisited node (DFS).
4. If every node has been visited without conflict, return true.

What about unconnected graphs?

Problem 3: Ratio Finder

You are provided with a list of symbolic equations of the form $\frac{a}{b} = c$, where a and b are symbols, and c is a real number. (I.e. you know what the ratio between all the symbols is, but not the values of the symbols.)

Given a list of challenges, consisting of pairs of symbols, return the ratio between each of them.

```
1 struct equation {
2     char numerator;
3     char denominator;
4     double ratio;
5 }
```

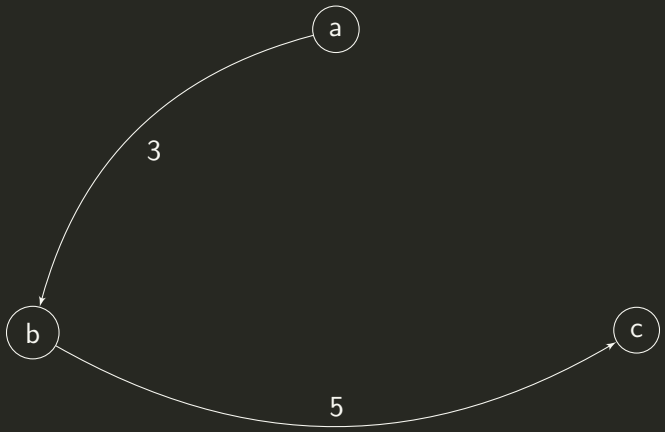
$$\frac{a}{b} = 3$$

$$\frac{b}{c} = 5$$

$$\frac{a}{c} = ?$$

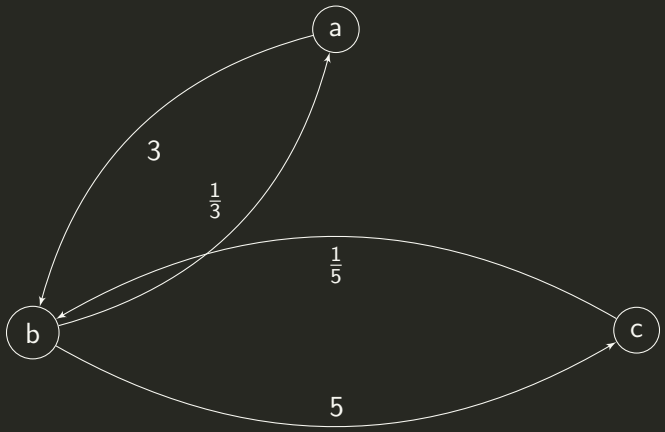
$$\frac{c}{b} = ?$$

Major Hint for Problem 3



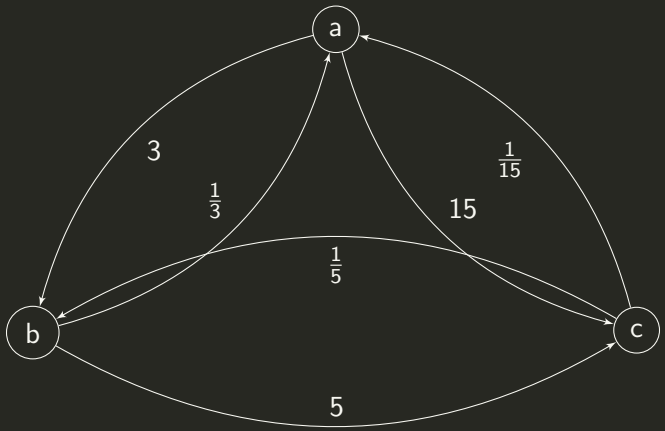
CS SOCIETY
FORDHAM UNIVERSITY

Major Hint for Problem 3



CS SOCIETY
FORDHAM UNIVERSITY

Major Hint for Problem 3



CS SOCIETY
FORDHAM UNIVERSITY

Solution Problem 3

1. Add inverse of every equation to graph to avoid duplicate code later on.
2. Iterate over every node that has challenge numerator as its numerator.
3. If any one of them has challenge denominator as its denominator return solution.
4. Else, recurse to 1 on every node, making new challenge numerator current node denominator.
5. When a match is found, multiply every node's ratio together back up stack.