A Brief Overview of Graphs, Graph Algorithms, and Special Graphs

Sahil Adhawade

December 3, 2021

Graphs

Consider the airplane routes from city to city, an interconnected web of friends on Facebook, or neurons in your brain. We can picture these examples as objects or pieces of information that are all connected in some way. In fact, each of these structures have two things in common: (1) a collection of objects, and (2) links between those objects. We can formalize these as **graphs**—mathematical structures for representing relationships. We can define a graph $G = (V, E)$ as an arbitrary non-empty finite set of *vertices*, V, and *edges*, E, consisting of pairs of vertices drawn from V. Let's take a look at the following two graphs.

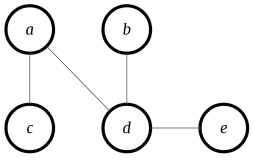

Figure 1: Undirected Graph

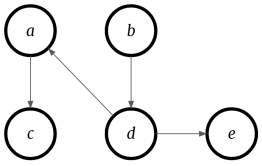

Figure 2: Directed Graph

In Figure 1, we present an undirected graph. This type of graph contains an *undordered* pair of vertices to represent edges. Therefore, we get the following vertex and edge sets:

$$V = \{a, b, c, d, e\}$$
$$E = \{\{a, c\}, \{a, d\}, \{d, b\}, \{d, e\}\}.$$

In Figure 2, we present a directed graph. This type of graph contains an *ordered* pair of vertices to represent edges. Therefore, we get the following vertex and edge sets:

$$V = \{a, b, c, d, e\}$$
$$E = \{\{a, c\}, \{d, a\}, \{b, d\}, \{d, e\}\}.$$

For the remainder of the paper, we will be referring to graphs that are undirected. Let's proceed to translate this formal definition of graphs in known data structures such as a map

`Map<v, Set<A>>`

that consists of vertices $v \in V$ as "keys" and a set of adjacent vertices, A, as "values".

In a table, a map representing the vertices and set of adjacent vertices for the undirected graph (Figure 1) should look like the following:

v	Set<A>
a	$\{c, d\}$
b	$\{d\}$
c	$\{a\}$
d	$\{a, b, e\}$
e	$\{d\}$

Now that we have translated the formal definition of a graph $G = (V, E)$ into a data structure, we can explore graph traversal algorithms such as the BFS (breadth-first search). Let's

Algorithm 1 BFS Implementation

Require: Map<V, Set[A]> G
Require: Queue<V> Q
Require: Vector<bool> B
  Pick any $v \in V$
  $B[v] \leftarrow true$
  $Q \leftarrow enqueue[v]$
  while $Q \neq \emptyset$ **do**
    $u \leftarrow dequeue$ from Q
    $A \leftarrow G[u]$
    for do $x \in A$
      if $!B[x]$ **then**
        $B[x] \leftarrow true$
        $Q \leftarrow enqueue[x]$
      end if
    end for
  end while

visualize this algorithmic implementation into the undirected graph shown in Figure 1.

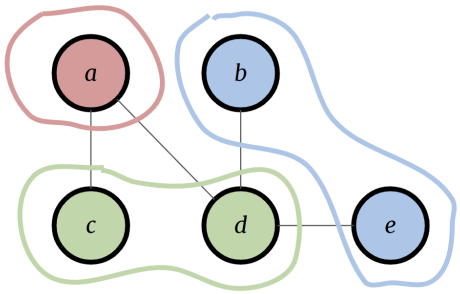

Figure 3: BFS on an Undirected Graph

Each time we visit a node in each BFS layer, we color the node. Different colors are

used to show the different BFS layers. The red node, a, is the first layer of the breadth-first search. It then expands out to its green neighbors, c and d, as the second layer of the BFS. Finally, since c has no non-visited adjacent nodes, we look at node d which has two unvisited adjacent nodes, b and e. The blue nodes, b and e, are comprised of our last layer of our BFS. We will revisit this BFS implementation when we look at bipartite graphs.

Vertex Covers and Independent Sets

Imagine a square park. Park designers are attempting to install lamp posts around the square park so that it is lighted at night. One obvious way to orient these lamp posts is by placing one at each vertex of the square.

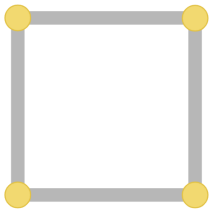

Figure 4: Lamp Post Orientation

However, they are on a budget and placing four lamp posts will be very expensive. How can they place the lamp posts so that no matter where a person stands they can have a lighted path? Consider the following orientation:

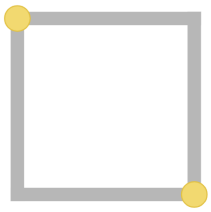

Figure 5: Optimized Lamp Post Orientation

As you can notice in Figure 5, no matter where you are along the square, you can see at least one lamp post. This is what we call a **vertex cover**. To formalize the definition of a vertex cover, consider a graph $G = (V, E)$. A *vertex cover* of graph G is a set $C \subseteq V$ such that the following statement is true:

$$\forall v \in V. \forall u \in V. (\{v, u\} \in E \implies (v \in C \vee u \in C)).$$

This essentially means that every edge in G has at least one endpoint in C.

Going back to the example of the lamp posts, consider the vertices with lamp posts to comprise the vertex cover set, $C = \{a, d\}$. In Figure 6, no matter which edge you choose, it

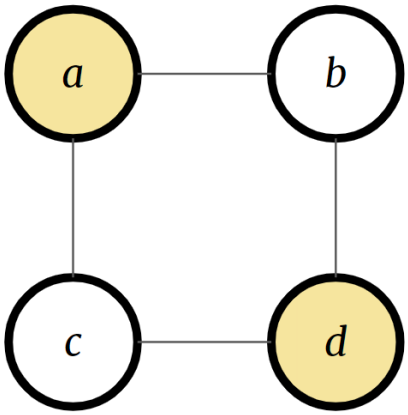

Figure 6: Graph Representation of Lamp Post Problem

will always have an endpoint in a node that is in C.

Now, let's look at the definition for an **independent set**. Consider an arbitrary graph $G = (V, E)$. An *independent set* $I \subseteq V$ exists such that:

$$\forall v \in I. \forall u \in I. \{v, u\} \notin E.$$

This means that there are no nodes in I that are adjacent. If we consider the graph pictured in Figure 6, we can note that the nodes a and d also make up an independent set I since there is no edge that connects any of the nodes in I.

Now that we have built a conceptual understanding of vertex covers and independent sets, let's take a look at an approximate algorithm for verifying vertex covers and an algorithm for independent sets in a graph.

Algorithm 2 (Approximate) Vertex Cover

Require: Vector<E> E
Require: Vector<V> C $\leftarrow \emptyset$
Ensure: C
  while $E \neq \emptyset$ **do**
    Pick any $\{u, v\} \in E$
    $C \leftarrow C \cup \{u, v\}$
    Delete all edges incident to either u or v
  end while
  return C

Algorithm 3 Independent Sets

```
Require: Map<V, Set[A]> G
Require: Vector<V> V
Require: State U, I, O (Undecided, in Independent Set, out of Independent Set)
  for v ∈ V do
    if v.state! = U then
      continue
    end if
    A ← G[v]
    for x ∈ A do
      if x.state == I then
        continue
      end if
      v.state ← I
      for y ∈ A do
        y.state ← O
      end for
    end for
  end for
```

Bipartite Graphs

An important family of graphs is known as **bipartite graphs**. Let's begin with a formal definition a bipartite graph. Consider an arbitrary undirected $G = (V, E)$. We consider a graph to be bipartite if there exists two sets $V_1, V_2 \in V$ where $V_1 \neq V_2$ such that

1. every node $v \in V$ belongs to exactly one of V_1 and V_2, and
2. every edge $e \in E$ has one endpoint in V_1 and the other in V_2.

To visualize this, consider the following bipartite graph representation. In Figure 7, we can

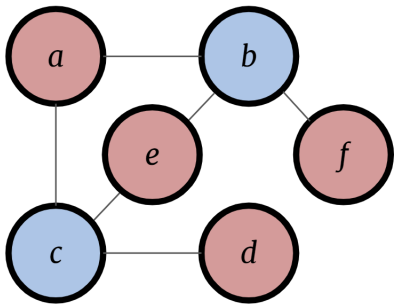

Figure 7: Bipartite Graph

note that there are two subsets of vertices in the bipartite graph. Let's assume that the red nodes are in v_1 and the blue nodes are in V_2. We can note that every node in the bipartite graph exists in one of those two bipartite classes, V_1 and V_2. Furthermore, we can confirm

that no matter which edge we pick, one endpoint of that edge is connected to a red node or a node in V_1 and the other end is connected to a blue node or a node in V_2. Note how we can apply the concept of independent sets in bipartite graphs. In bipartite graphs, V_1 and V_2 must be independent stets. Just how we wrote an algorithm that used flags on nodes to determine if they were in the independent set, in the algorithm to check bipartiteness of a graph, we will use node coloring. Furthermore, we will use BFS to traverse and color the graph. Consider the following algorithmic implementation to check if a graph is bipartite.

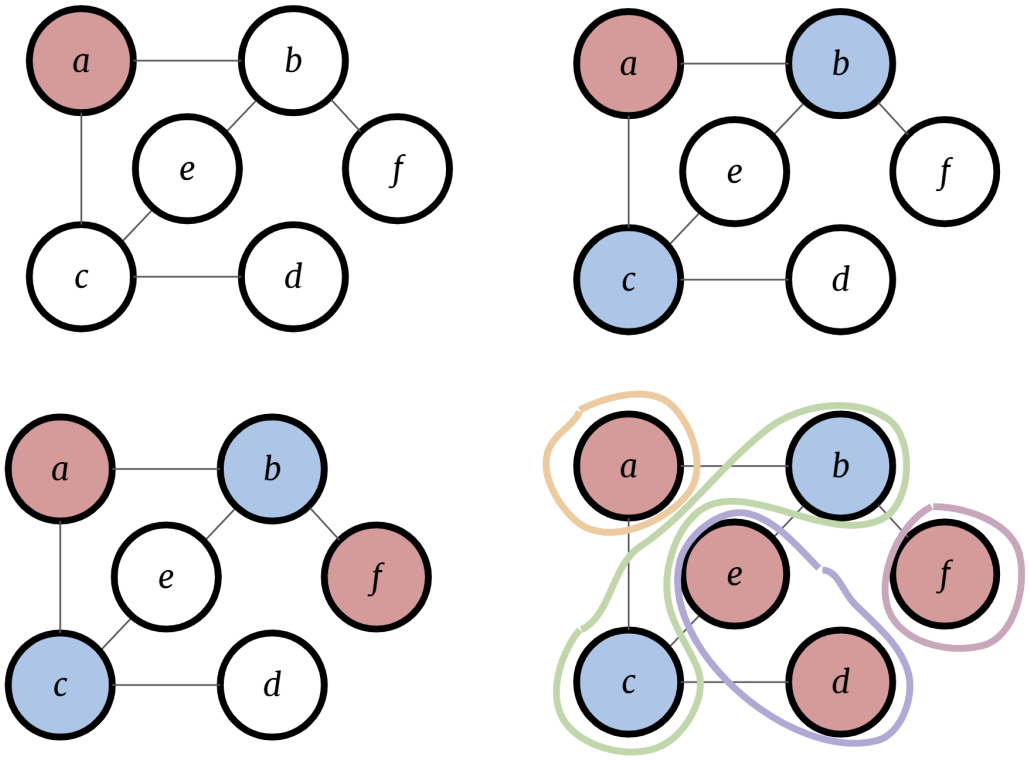

Figure 8: Bipartite Graph

In Figure 8, we can see the BFS-driven algorithm work to determine if the graph is bipartite. It colors the first node, a, red. It looks at all of its neighbors, b and c and colors them blue since a is red already. We then look at the neighbors of the current set of nodes, b and c. Since b is adjacent to f and it is blue, f is colored red. Next, we look at the neighbors of c which are d and e. Since c is red, it colors d and e red. Since the entire graph was colored successfully without any overlaps, it returns that the graph is "bipartite". Figure 8 also highlights the BFS progressions by marking the first layer of search orange, the second layer green, the third layer pink, and the final layer purple.

Algorithm 4 Bipartite Graph

Require: Map<V, Set[A]> G
Require: Queue<V> Q
Ensure: "Not Bipartite" or "Bipartite"
  Pick any $v \in V$
  $v.color \leftarrow RED$
  $Q \leftarrow enqueue[v]$
  while $Q \neq \emptyset$ **do**
    $u \leftarrow dequeue$ from Q
    $A \leftarrow G[u]$
    for $x \in A$ **do**
      if $x.color ==$ BLANK **then**
        $x.color \leftarrow (u.color == RED)?BLUE : RED$
        $Q \leftarrow enqueue[x]$
      else if $x.color == u.color$ **then**
        return "Not Bipartite"
      end if
    end for
  end while
  return "Bipartite"

Finally, let's discuss some interesting uses of bipartite graph and its algorithms in real-life use cases. Over quarantine, most of us watched a lot of Netflix. As we binge a move or TV-series, Netflix somehow always knows what we might want to watch next. This recommendation algorithm uses bipartite graphs as its foundation. The viewers as vertices in the set V and the movies and shows they watch can be seen as vertices in the set M. There is an edge from a node $v \in V$ to a node $m \in M$ if v viewed m. A cool nuance to this bipartite graph is that the edges are weighted towards the genres you watch more. Similarly, we can use this structure in NLP algorithms were we need to match up terms or wrods in a document. Suppose there are D documents and T terms. There exists an edge $\{d, t\}$ for some $d \in D$ and some $t \in T$ if the term t exists in document d. Bipartite graphs can be used to analyze the text and, for instance, cluster the documents. These are just a few applications of bipartite graphs.