DS 320 Algorithms for Data Science **Lecture 5**
Spring 2022 TF Freddy Reiber

1 Breath First Search

Recall what we did last time! Given a graph $G = (V, E)$, we want to formally "explore" the graph. Specifically, we want to search the graph and sometimes order the verticies in a meaningful way. There are two commonly used types of graph traversal algorithms, the first being *depth first search*, which explores as far down as possible before backtracking and exploring a new node. The second, which we will study today is *breath first search* which instead explores all the nodes in order of their "depth" from our starting node.

1.1 Pseudo-Code

```
procedure BFS(G, s) is
      let Q be a queue
      label s as explored
      Q.enqueue(s)
      while Q is not empty do
          v := Q.dequeue()
          for all edges from v to w in G.adjacentEdges(v) do
              if w is not labeled as explored then
                  label w as explored
                  w.parent := v
                  Q.enqueue(w)
```

1.2 Intuition

Let's now give a quick English explanation for how this algorithm is working. We are given a graph $G = (V, E)$ and a starting node $s \in V$. We then "explore" every node that is adjacent to our starting node s, and add them to the queue. We then take the next element in the queue, and repeat this process. Notice that this explores the nodes in order of depth from s. Moreover, we first explore every node that has a depth of 1 from s, then every node that has a depth of 2 from s, and so on. Compare this to DFS, in which we explore one path all the way "down" before exploring the next path. One could also think of BFS as using a queue to determine which node to consider next, while DFS is basically using a stack.

1.3 Run time

We will know formally prove the runtime of BFS. Note that we will use the following notation, $n = |V|$ and $m = |E|$ First, lets give a quick argument for a weaker bound, specifically that this algorithm runs in $\mathcal{O}(n^2)$ time. To see this, notice that each node has at most n edges (this occurs when the graph is "complete"). Thus for each node which there are n of, when we check its neighbors we are most checking n elements. Each of those operations is constant, thus we get $\mathcal{O}(n^2)$ as an upper bound on the runtime. However, we can give a more accurate bound!

Theorem 1. *Breadth-First Search runs in $\mathcal{O}(m+n)$ time.*

To show this improved bound, we need to make a slightly more subtle observation. Specifically, notice that processing a single node might not need to take $\mathcal{O}(n)$ time. This occurs when the node has a few neighbors. Now, instead of simply upper bounding the number of possible edges, lets give a more accurate count by counting the degree of every node. We will use n_u to denote the degree of node u. Notice that the total amount of work we do inside the loop that considers adjacent nodes is simply the sum of the degrees. Formally this is $\sum_{u \in V} n_u$.

Lets now do a quick check to see what this sum evaluates to. Notice that each edge $e = (v, w)$ contributes exactly twice to this sum. Once in n_v and again in n_w. Thus we can conclude this sum is equal to $2m$. This also gives a very useful lemma called the "handshake" lemma. Something we will touch on if time permits.

Now that we know what this sum evaluates, we can give a tighter bound, since the total time spent looking at edges is now $\mathcal{O}(m)$. Combine this with the $\mathcal{O}(n)$ needed for array operations and such, we get a new and more accurate bound of $\mathcal{O}(m+n)$.

2 Useful Properties

Introduce the concept of layers or depth. Essentially each layer is the next level.

Theorem 2. *For each node with depth $i \geq 1$, the tree produced by BFS consist of all nodes at distance i from s. Moreover, there is a path from s to t if and only if t appears in some layer.*

Notice that if we take only the edges we used to explore, BFS produces a tree T, that is rooted at s. More specifically a connected a cyclic graph, or a graph in which any two pair of nodes has a unique path.

We can prove these two things pretty easily. Obviously the tree is spanning the connected component, thus if there is a path, we will find it.

Theorem 3. *Let T be a BFS tree and let x and y be nodes in T belonging to layers L_i and L_j, respectively, and let (x,y) be an edge of G. Then i and j differ by at most 1.*

3 Testing Bipartiteness

Let's first recall the definition of a bipartide graphs. A bipartide graph is a graph that can have its vertices divided into two sets X and Y, such that every edge is on the cut between X and Y. (Remember to define what a cut is!!).

Often times we will see bipartite graphs colored to make this more illustrative. An example can be seen in figure 1:

To help build some intuition lets try a few examples! First is the triangle graph bipartide? (The answer is no!) What about a "square" graph or C_4. (The answer is yes!). Try playing around with this a bit, and see if we can formulate some necessary property of bipartide graphs!

Theorem 4. *If a graph G is bipartite, then it cannot contain an odd cycle.*

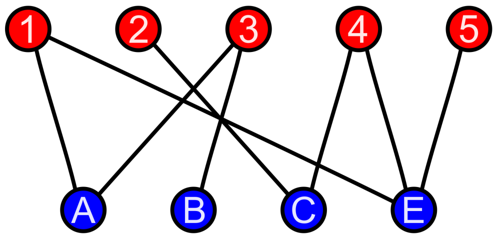

Figure 1: A simple bipartite graph

This proof is relatively straightforward. Notice that due to the bipartidness of the graph, we can only go from X to Y, and vice versa. Thus any cycle that starts in X, must be even! We can also show this by attempting to construct a bipartide partition. Assume we have some odd cycle, and lets try to split it up. We first, pick a node to start, and put it in X, then put on of the adjacent nodes in Y, and so on. By definition of an odd cycle, the graph will not be able to be bipartide.

Conjecture 1. *Is this relationship and equivalent relation. Is a graph bipartide if and only if it contains no odd cycles.*

3.1 The algorithm

First, lets assume that the graph is connected, otherwise we can simply run this algorithm on each individual component. Then pick a node and color it red. Then color all its neighbors blue, then those neighbors red and so on. Now, we either have a red/blue coloring of G, or there is a pair of adjacent nodes with the same color. Thus the graph is not bipartite.

Notice that this procedure is basically BFS, we started from s, and looked at all its neighbors. Then we looked at all of the neighbors of those neighbors. Essentially, we can think of this as just preforming a BFS and coloring when we enter the next "layer". Implementing this should be obvious, just adding a color to the nodes. As per are earlier analysis this is obviously still an $\mathcal{O}(m+n)$ algorithm.

3.2 Proving Correctness

We now finally prove that this algorithm correctly determines weather a graph is bipartite or not.

Theorem 5. *Let G be a connected graph, and let X and Y be the two sets defined by the coloring in our previous algorithm. Then if no adjacent edges share a color the graph is bipartite. If this not the case, then G contains an odd-length cycle and so it cannot be bipartite.*

Lets first proof the first case, where we know that no adjacent edges share a color. By our algorithm, we know that every edge of G joins nodes either in the same layer or in adjacent layers. But our assumption for this case, every edge must be in adjacent layer. Thus every edge joins two nodes in adjacent layers. Finally, since our coloring procedure gives noes in adjacent layers the opposite colors, we can conclude that G is bipartite.

Next, lets suppose we are in the second case. Why must G contain an odd cycle. We are told that G contains an edge joining two nodes in the same depth. We can then follow the path down to these two nodes, and then connect them with the edge that must be there. Since the path down is 2 times the layer level plus the one connected there must be an odd cycle.