Breadth-First Search and Shortest Paths

Suhas Arehalli

COMP221 - Spring 2024

1 Introductions

- When you finally get free of sorting, the next most common Algorithms topic is Graphs!
- Why? Because, as you saw in COMP128, graphs are really good at representing a variety of problems, and the solutions for those problems can often be found by analyzing the corresponding graph using a standard graph algorithm.
- **Note:** Skiena does a very in-depth job introducing you to the various ways that graph traversals (today's topic) can be useful for a variety of problems. But that's just one pillar of the course! I'll spend today's notes preparing you for one other pillar: Analyzing graph algorithms for correctness!
- Since BFS/DFS are algorithms you've seen and talked about before, I'll be light on the exposition and dive into the trickiness of the working with graphs formally.

2 Shortest Paths in an Undirected Graph

- Our goal for today is to prove formally that BFS finds us the shortest path between the start node and a target node. Pay attention to how the proof is structured, and where the intutions come from, because working with graphs is a bit tricker than arrays!
- A Graph G is a pair (V, E), where V is a set consisting of vertices and E is a set of edges, where each edge is a pair $(v_1, v_2) \in E$, where $v_1, v_2 \in V$.
  - Today we'll consider *undirected* graphs, where $(v_1, v_2) \in E$ implies $(v_2, v_1) \in E$. That is, the order of vertices in an edges doesn't matter!
  - We often visually represent them as circles connected by lines, but to analyze algorithms over graphs, it's helpful to build an intuition for the correspondence between our visual representations of data structures, their implementation (in adjacency lists or adjacency matrices!), and their mathematical structure (A pair (V, E)).
- A path from $v \in V$ to $v' \in V$ is (for our purposes) a sequence of edges

  $$(v, v_1), (v_1, v_2), \ldots, (v_{k-2}, v_{k-1}), (v_{k-1}, v') \in E$$

  We call the number of edges in the path (here k) it's length.

- With that terminology out of the way, we want to prove that BFS will find the *shortest path* from a start vertex $s \in V$ to target vertex $t \in V$.
- For simplicity, the pseudocode for BFS in Alg. 1 does not explicitly construct the path from s to t, but instead returns the *length* of the shortest path. Maintaining the path is not hard (just maintain a representation of the path, either in the graph's data structure or in the tuple in the queue!), but will clutter our pseudocode — consider it an exercise for the reader!

Algorithm 1 A Breadth First Search that returns the length of the shortest path from s to t.

```
function BFS($G = (V, E)$, s, t)
    Let $Q$ be a Queue.
    DISCOVER($s$)
    $Q.enqueue((s, 0))$
    while $Q$ is nonempty do
        $v, d \leftarrow Q.dequeue()$
        for $(v, v') \in E$ do
            if $v'$ is undiscovered then
                if $v' == t$ then
                    return d + 1
                end if
                DISCOVER($v'$)
                $Q.enqueue(v', d + 1)$
            end if
        end for
    end while
end function
```

2.1 Proof of Correctness

- We'll proceed through a few lemmas (as a proof is normally presented!), but hopefully you'll look back once you see why the lemmas are crucial and work to identify how you could have constructed the lemma yourself. They'll look weird at first, especially if you haven't run through the algorithm on a few graphs a few times!
- I've since become frustrated with the presentation I gave you in class (I think I tried to be too clever working a proof of correctness for BFS-based algorithms to find connected components into the lemma structure), so this proof is modified a bit for clarity.

Lemma 1: The distances in the queue are monotonic increasing (i.e., sorted!) and differ by at most 1. Formally, for a Queue $Q = (v_1, d_1) \dots (v_k, d_k)$, we want that

$$d_1 \leq d_2 \leq \cdots \leq d_k \leq d_1 + 1$$

Convince yourself that the words and the math align!

Proof: Let's proceed to prove this as a *loop invariant*: We'll show it's true before the while loop starts, and at the end of every iteration.

First, our **Base Case**: Consider that before our loop, the only thing in Q is $(s, 0)$. With one element, it's by definition monotonic increasing and it can't differ from itself by more than 1. So we're done!

For our **Inductive Step**, we will assume, by our inductive hypothesis, that we begin the while loop with $Q = (v_1, d_1) \dots (v_k, d_k)$ such that

$$d_1 \leq d_2 \leq \dots \leq d_k \leq d_1 + 1$$

What do we do each iteration to manipulate Q? Two things!

1. First we remove (v_1, d_1) from the queue.
2. Then, for each $(v_1, v') \in E$, we add $(v', d_1 + 1)$ to the queue. For notational convenience, let's say there are l of them and call those entries $(v'_i, d_1 + 1)$ for $1 \leq i \leq l$.

Thus after all of our manipulations, our queue will look like

$$Q = (v_2, d_2), \dots, (v_k, d_k), (v'_1, d_{k+1} = d_1 + 1), \dots, (v'_l, d_{k+l} = d_1 + 1)$$

Now observe that our array must be sorted/monotonic increasing, because our new elements all have distance $d + 1$. By our IH,

$$d_1 \leq d_2 \leq \dots \leq d_k \leq d_1 + 1$$

So considering this for our new elements, we can realize that $d_{k+1} \dots d_{k+l} = d_1 + 1$, so $d_{k+1} \geq d_k$ (the last inequality in the IH!), and $d_{k+1} \leq d_{k+2} \leq \dots d_{k+l} \leq d_1 + 1$ just because weak inequalities are weaker than strict equality! All that's left is observing that since, by our IH, $d_1 \leq d_2$, $d_1 + 1 \leq d_2 + 1$, and thus we get that for Q as above, we have

$$d_2 \leq d_3 \leq \dots \leq d_k \leq d_{k+1} \leq \dots d_{k+l} \leq d_2 + 1$$

As desired. This conclude our inductive step and our proof!

Lemma 2: If s and $v \in V$ are connected, BFS will add a pair (v, d) corresponding to a shortest path from s to v.

Proof: First observe that if v is connected to s, there exists at least one path $(s, v_1) \dots (v_{k-1}, v) \in E$ from s to v, where k is the length of that path. Let's proceed by induction over the length of the *shortest* of these paths!

Our **Base Case** is when the path is of length 0. There is only one such vertex (s!) that is connected by a path of length 0. We enqueue $(s, 0)$ before we start our loop, so we're done.

Our inductive hypothesis is that all vertices that are connected by a (shortest) path of length $k - 1$ will be considered. We must show that all vertices connected by a path of length k will be considered.

To see this, we first consider an arbitrary vertex connected by a shortest path of length k.[1] That path is some sequence of edges

$$(s, v_1), \dots, (v_{k-2}, v_{k-1}), (v_{k-1}, v) \in E$$

[1]People will sometimes say this is done *without loss of generality*, or WLOG, to indicate that we are assuming nothing new about these vertices. If we prove something for an *arbitrary* vertex with these properties, we prove that for *all* vertices with those properties

Now we make a key observation: A shortest path of length k is a shortest path of length $k-1$ *plus one edge*! To see why the path of length $k-1$ must be the shortest path to v_{k-1}, we simply observe that if there were a shorter path to v_{k-1}, that path plus (v_{k-1}, v) would be a shorter path to v (a mini-proof by contradiction!).

So we know the path from s to v_{k-1} is a shortest path of length $k-1$, and thus by our IH, we add a tuple corresponding to the shortest path to v_{k-1} to Q!

Of course, we then note that the while loop only terminates once Q is empty. This means that we won't stop until v_{k-1} is dequeued as well! But, once v_{k-1} is dequeued, we consider all edges from v_{k-1} in the for loop, one of which we know (by the existence of the path!) is (v_{k-1}, v)!

Now here's our trick: At this point, one of two things happens. Either v is undiscovered, we enter the if-statement, and add a tuple corresponding to it to Q and we're done, or v is discovered already. But v begins undiscovered, and is marked discovered only when a tuple corresponding to it is added to the queue! Thus a tuple corresponding to v has already been added to Q!

Then (changing our structure from class), we can explicitly work in Lemma 1: If a tuple (v, d') has been added to our queue previously, it must be be earlier in the queue, and thus by Lemma 1, $d' \leq d$. If $d' < d$, we reach a contradiction (thankfully, I can still work those in!): d is the length of the shortest path to v by assumption! Thus, we know that that $d' = d$. Since we just need a tuple corresponding to *a* shortest path (there can be multiple paths of the same length!), this completes our proof: If it's already been added to the queue, we've added a tuple corresponding to a shortest path from s to v, which is exactly what we want!

- Here, it's useful to note that the last proof is *still* essentially the proof of correctness for a BFS-based algorithm to find connected components! The proof for something like this for DFS will necessarily look a little different (Lemma 1 certainly can't apply to DFS!), but we can generalize to saying that *a* path from s to any connected vertex will be (implicitly) added to our (runtime) stack (through a recursive call!).

Statement: $(v, d) \in Q$ iff d is the length of the shortest path from s to v.

Proof: Given how we've restructured our proof relative to lecture, most of our work here has been offloaded into Lemma 1. However, there is still one thing left to prove: We've shown that a tuple corresponding to the shortest path is always added to Q, but we haven't shown that *every* tuple added to Q represents a shortest path (note that this is an iff!). This still lets us get our nice contradiction example!

Assume for contradiction that we enqueue some (v, d) when there exists a path of length $k < d$ from s to v. That path takes the form $(s, v_1) \dots (v_{k-1}, v) \in E$. Because it's a shortest path, by Lemma 2 there was, at some point, a $(v, k) \in Q$. But Lemma 1 says the distances in the queue are always increasing, and thus since $k < d$, that means that (v, k) was enqueued before (v, d) was enqueued. Now we use our secret trick — DISCOVER!. When (v, k) was enqueued, DISCOVER(v) was called (this occurs right before both enqueues in our pseudocode!). This means that v was discovered. But we assumed that (v, d) was being enqueued, which can only be *within* the if statement that checks that v is *undiscovered*! We find a contradiction!

Thus, we cannot enqueue any (v, d) where d is not the length of the *shortest* path from s to v.

- You should take note at how the proof relies on the details of the tricky part of implementing BFS: Remembering to mark discovered nodes as discovered and making sure you don't add them to your queue! This correspondence between tricky pieces in proving code correct and tricky pieces of code should point toward the value of working through formal proofs of correctness. If you *didn't* know if your BFS implementation was correct, getting to this point of the proof would provide the shape of the hole your fix should fill!