Some Extra Information on Graph Search

David Kempe
December 5, 2013

We have given basic definitions of graphs earlier in class, and talked about how to implement them. We had also already mentioned paths, and nodes reaching each other. Indeed, one of the questions most frequently asked about graphs is some variant of the following:

- Is there a path from $u$ to $v$?
- What is the shortest path from $u$ to $v$? (Either in terms of number of hops, or distances or costs which may be stored on edges.)
- What is the distance from $u$ to every other node in the graph (or every node that can be reached from $u$)?

All of these problems fall under the broad area of graph search or computation of shortest paths (or just paths). These problems have many important applications:

1. Finding fast or cheap ways to go from one place to another, by utilizing road networks (optimizing distance or time), or airline or train networks (optimizing time or cost).

2. Finding social distance in a social network. One prominent example of this is the Kevin Bacon Game, in which the players try to construct short sequences of movies that connect a given actor or actress to the actor Kevin Bacon. There is an edge between two actors iff they co-starred in at least one movie.¹

3. In a game (usually a puzzle or logic game), is there a way to manipulate the game according to the rules (sliding boxes, squares, changing switches, moving around, etc.) to get from a given initial state to a given target state?

4. The idea of planning to get from one state to another is more pervasive, and extends beyond games. Robots need to plan sequences of moves to get them from one configuration to another. Many systems need to plan sequences of actions to execute.

One can come up with many more examples: a huge number of real-world problems are extremely naturally modeled as shortest path searches in suitably defined graphs. Therefore, graph search algorithms are one of the most central concepts for a computer scientist to be familiar with.

As we briefly mentioned above, there are a few parameters along which concrete instances can differ:

1. Do we just want to know whether a path exists, or do we need to find it?
2. If we need to find a path, do we want the shortest path, or will any path do?
3. If we don’t need to find it, do we need the exact distance between nodes, or just whether a path exists?
4. Do edges have lengths/costs associated with them, or are we just interested in the number of hops.

¹The mathematicians’ version of this is the Erdős Number, where two mathematicians are connected iff they co-authored at least one paper, and one is interested in the number of hops to the legendary Hungarian mathematician Paul Erdős.
5. If edges have costs associated with them, can they be negative?

For the first three questions, it turns out that generally, the problem doesn’t get much easier by asking just whether a path exists, so the algorithms we will investigate will actually find a path; and the most “natural” algorithm (BFS) actually finds a shortest path. On the other hand, edge lengths/costs make the problem a bit more challenging, and we will see later how to deal with them. Negative edge costs are not terribly difficult, but are a little bit beyond this class — you will likely learn about them in CSCI 270.

1 Breadth-First Search (BFS)

Breadth-First Search (BFS) is a fairly simple algorithm that finds the shortest path (in terms of number of hops) from a given node \( u \) to all nodes \( v \). The idea of BFS is to “explore” the graph in layers.

- Layer 0 is the node \( u \) itself.
- Layer 1 consists of all nodes \( v \) with a direct edge \((u, v) \in E\).
- Layer 2 consists of all nodes \( w \) with a direct edge \((v, w) \in E\) from some node \( v \) in layer 1. However, it excludes nodes that were already in Layer 1 themselves.
- More generally, Layer \( k \) consists of all nodes \( w \) with a direct edge from some node \( v \) in Layer \( k-1 \), but such that \( w \) itself was not already in an earlier layer.

![Figure 1: A directed graph. Here, layer 0 is just node \( u \) (the source). The nodes \( v_1, v_2, v_3 \) form layer 1. The nodes \( w_1, w_2, w_3 \) are layer 2. Notice that node \( v_2 \) was already in layer 1, so it’s not also in layer 2. Layer 3 contains just node \( x \). Notice that node \( y \) is not reachable from \( u \), and is thus at infinite distance.](image)

The recursive definition of layers is quite nice. It’s not too difficult to prove with this definition that the nodes in layer \( k \) are exactly at distance \( k \) from \( u \). However, to implement this algorithm, we will have to process nodes one by one, as processing an entire layer at once is not standard programming functionality.

When we process nodes sequentially, we would like to make sure that all nodes from layer \( k \) have been processed before we get to the nodes in layer \( k+1 \). Because we “discovered” (with the algorithm) the nodes in layer \( k \) before those in layer \( k+1 \), this means we should process nodes in the order they were discovered. We have seen a data structure supporting exactly that: a FIFO queue. Indeed, the FIFO queue is the central part of the Breadth-First Search algorithm.

For the description and the analysis, we will let \( \hat{d}(v) \) denote the actual distance from \( u \) to \( v \), and \( d[v] \) the distance the algorithm computes. Of course, we hope that those will be the same, but we need to actually prove that. We will assume that nodes are numbered with integers \( 0, 1, \ldots, n-1 \), so that we can simply store values associated with nodes in arrays of size \( n \). In addition to the array of distances \( d[v] \), the algorithm
will also compute an array of *predecessors* \( p[v] \): for each node \( v \) (other than \( u \)), this is a node in the previous layer which has an edge to \( v \). For instance, in Figure 1, \( p[w_3] = v_3 \), and \( p[x] = w_1 \). On the other hand, \( p[w_1] \) could be either \( v_1 \) or \( v_2 \), depending on the order in which those two nodes were explored. We will consider the arrays as global variables, just to keep notation simple.

```cpp
int d[n]; // will store distances from u
int p[n]; // will store predecessors of nodes on shortest paths from u

void BFS(int u) {
    bool visited[n] = {false}; // no node is visited yet.
    Queue<int> q(); // empty queue to start.
    visited[u] = true; d[u] = 0;
    q.enqueue (u);
    while (!q.empty()) {
        int v = q.peekfront();
        q.dequeue();
        for (all nodes w with an edge (v,w) in E) // use some iterator
            if (!visited[w]) {
                visited[w] = true;
                d[w] = d[v] + 1;
                p[w] = v;
                q.enqueue(w);
            }
    }
}
```

The idea of the algorithm is as follows: we start with just the seed node \( u \). Its distance from itself is obviously 0. Then, the for loop adds all of \( u \)'s neighbors to the queue.

Whenever we process a node \( v \) (dequeuing it), we look at all its outgoing edges. The edge could lead to a node we have already seen, in which case we should not process it again — if we did, we might accidentally increase the distance label. If we have not seen the node \( w \) before, then we mark it as “visited” (so it isn’t processed again\(^2\)), and set its distance and predecessor correctly. The predecessor is the node \( v \) from which the edge \((v,w)\) allowed us to discover \( w \), and the fastest way to go from \( u \) to \( w \) is to first go to \( v \), then take one more hop, so the distance is one more than that of \( v \). Finally, we add \( w \) to the queue, so it’s processed later.

While we won’t do a formal correctness proof of BFS in this class, we want to roughly think about how BFS ensures correctness. Clearly, the FIFO queue is essential, and we should think what exactly it guarantees. A loop invariant that works would be the following. At each point in the while loop, the following are true:

1. For all nodes \( v \) with \( visited[v] == true \), the values of \( d[v] \) and \( p[v] \) are correct. That is, \( d[v] = \hat{d}(v) \), and \( p[v] \) is a last hop node on a shortest path from \( u \) to \( v \).
2. The distances of nodes in the queue are sorted. Let \( v_1, v_2, \ldots, v_k \) be the nodes in the queue, sorted by when they were inserted. (\( v_1 \) has been in the queue longest, and \( v_k \) was added most recently.) Then, \( d[v_1] \leq d[v_2] \leq \ldots \leq d[v_k] \).
3. The nodes in the queue are all from just two adjacent layers. That is, \( d[v_k] \leq d[v_1] + 1 \).

While this may seem a little convoluted, if you think about it carefully, you’ll hopefully see how that captures exactly our intuition that the FIFO queue gives a close emulation of processing nodes in layers. A correctness proof would now establish the loop invariant by induction over iterations of the while loop.

\(^2\)Forgetting this part is a commonly made mistake when coding BFS.
Next, let’s analyze the running time of the BFS algorithm. First, we notice that the initialization of the \textit{visited} array, while written as one line, actually takes \(\Theta(n)\). All other individual commands take time \(\Theta(1)\). So, as usual, we’ll analyze loops inside out. The \texttt{for} loop goes through all outgoing edges of node \(v\). If our iterator is implemented carefully (e.g., we used an adjacency list for our graph), then this takes \(\Theta(\text{out-degree}(v))\) for node \(v\). All the rest of the code inside the \texttt{while} loop takes time \(\Theta(1)\), so the total time inside one iteration of the loop is \(\Theta(1 + \text{out-degree}(v))\). (Normally, we would drop the 1. The reason we are keeping it here for the moment is that it is possible that \text{out-degree}(v) = 0, in which case our analysis would claim that the iteration takes 0 time, which is not true.) Now, as usual, we notice that the \texttt{while} loop may run over all nodes, so the running time is a sum

\[
\sum_v \Theta(1 + \text{out-degree}(v)) = \Theta(\sum_v 1 + \sum_v \text{out-degree}(v)) = \Theta(n + m).
\]

Together with the \(\Theta(n)\) for initialization, the total running time of BFS is \(\Theta(n + m)\), which is linear in the size of the graph (the time it takes just to read the graph as input). So BFS is a fast algorithm.

2 Depth-First Search (DFS)

Breadth-First Search processes the graph by layers of increasing distances from the start node \(u\). That is what FIFO queues are great for. The downside is that the search algorithm jumps all around over different areas of the graph. For instance, if you trace the execution on the graph in Figure 1, you see that after processing \(u, v_1, v_2, v_3\), rather than staying with \(v_3\) and its neighbors, the algorithm next jumps to \(w_1\). When you want just shortest paths, that’s of course not a problem, but sometimes, you want to explore a graph by staying local — this is sometimes needed as part of other algorithms. To explore a graph locally, we would want to first explore all the nodes “around \(v\)” before switching to other parts of the graph.

We can accomplish this by replacing the FIFO Queue in the BFS algorithm with a LIFO Stack instead. That way, the most recently added node will be the one explored next, which ensures that the algorithm stays in a neighborhood of a node. Apart from that one-word change, the algorithm is exactly the same. This algorithm is called Depth-First Search (DFS), because it first explores the neighborhood of a node going deeper into that area, before going for other parts of the graph.

Of course, DFS will not find shortest paths any more necessarily. For instance, in our Figure 1, it will first reach \(v_2\) from \(v_1\), assigning it a distance of 2, and never revisiting this choice. Notice that this is fine — DFS is not used when one wants shortest paths. For that, you would use BFS. But since the \(d[v]\) values don’t really capture anything particularly useful any more, we could simplify the algorithm a bit by removing those arrays altogether.

The running time of DFS is obviously also \(\Theta(n + m)\), since we just plugged one data structure in for another, and all operations take the same amount of time.

There is an alternative (perhaps even simpler) implementation of DFS using recursion instead of a stack. In fact, we already saw that algorithm for trees, when we looked at in-order (or post-order or pre-order) traversal of a tree. Those were all specific instantiations of DFS, and we can now formulate it somewhat more generally:

```cpp
bool visited[n] = {false};

void DFS (int u) {
    for (all nodes v with an edge (u,v) in E) // again use an iterator
        if (!visited[v]) {
            visited[v] = true;
            p[v] = u;
            DFS(v);
        }
}
```
In this implementation, we don’t have to explicitly worry about stacks — the recursion automatically takes care of the stack for us.

3 Graph Search with Costs — Dijkstra’s Algorithm

Now, let’s return to our goal of finding shortest paths, but let’s assume that each edge $e = (v, w)$ has a cost (or length) $c_e = c[v][w] \geq 0$. First, we might just try to run BFS. But that’s clearly not correct, as we can see from Figure 2.

![Figure 2: An example with edge costs in which BFS does not find the shortest path from $u$ to $w$.](image)

When we process $u$ (the start node), we add $v_1$ and $w$ to the queue. We next process $v_1$, and add $v_2$. But when we process $w$, we are done with it, and never revisit its distance label. So we think that it is at distance 10 from $u$, when it really is at distance 4.

The mistake we are making here is that processing $w$ is premature. The node $v_2$ is a much better candidate to explore next, because it is closer to $u$; the fact that it is closer means that it may lie on an actual shortest path from $u$ to $w$ (as it does here), so we shouldn’t conclude anything about the distance from $u$ to $w$ until we’ve made sure that there isn’t another, shorter, path.

What this suggests is that nodes shouldn’t just be stored in a queue. When a newly added node — like $v_2$ in our example — is closer to $u$, then it should be allowed to jump ahead of other nodes and be processed before them. But we’ve already seen a data structure built exactly for this purpose: a Priority Queue. So Dijkstra’s Algorithm is obtained pretty much exactly by replacing the Queue in the BFS algorithm with a Priority Queue. Beyond that change, we just need some small modifications, mostly to take care of the fact that the estimated distance of a node may be updated multiple times. For instance, in our example in Figure 2, when we explore $v_2$, we update our distance estimate for $w$ to 8 instead of 10, but we later update it to 4, which is the final value. The more formal algorithm we get is as follows:

```plaintext
int d[n]; // will store distances from u
int p[n]; // will store predecessors of nodes on shortest paths from u
int c[n][n]; // contains the costs of edges

void Dijkstra (int u) {
    bool visited[n] = {false}; // no node is visited yet.
    PriorityQueue<int> pq ();
    /* empty priority queue to start. We assume that the priority queue always returns the element v with smallest d[v] value, and can deal with updating the d[v] value for an element. */
```
visited[u] = true; d[u] = 0;
pq.add (u);
while (!pq.isEmpty()) {
    int v = pq.peek();
    pq.remove();
    for (all nodes w with an edge (v,w) in E)
        if (!visited[w] || d[v] + c[v][w] < d[w]) { // found a shorter path to w
            d[w] = d[v] + c[v][w];
            p[w] = v;
            if (!visited[w])
                { 
                    visited[w] = true;
                    pq.add(w);
                }
            else pq.update(w);
        }
}

Notice the strong similarity to the BFS algorithm. The main change is the we use a priority queue instead of a queue, and that we have an additional condition that allows us to process a node w: not only when the node hasn’t been visited at all, but also when we found a strictly shorter distance to get to w. In the latter case, the node will already be in the priority queue, so instead of adding it, we need to update the associated value. We haven’t talked in class about how to implement that operation, but if we are somewhat careful to keep track of nodes’ positions in the heap, this isn’t terribly difficult to implement.

The correctness proof would again use induction over the iterations of the while loop. It requires a bit more care in the formulation of a loop invariant, and you may see it in CSCI 270 (or CSCI 570 or 670, if you take it). We won’t go over it here, even though it’s not all that difficult.

The running time analysis gets a little more interesting than for BFS. The remove() and add() operations now both take $\Theta(\log n)$ time (while peek() is still $\Theta(1)$). So the content of the for loop takes time $\Theta(\log n)$, and since it executes $\Theta(\text{out-degree}(v))$ times, the total time of the for loop is $\Theta(\text{out-degree}(v) \log n)$. This in turn gives us that the stuff inside the while loop takes time $\Theta((1 + \text{out-degree}(v)) \cdot \log n)$, which we now have to sum up over all nodes $v$, as usual:

$$\sum_v \Theta(1 + \text{out-degree}(v)) \cdot \log n) = \Theta(\log n \cdot (\sum_v 1 + \sum_v \text{out-degree}(v))) = \Theta(\log n \cdot (n + m)).$$

For Dijkstra’s Algorithm, even though in principle, we could have the case $m < n$ just like we could for BFS, people usually ignore the $n$ part of $n + m$, and simply describe the running time as $\Theta(m \log n)$. Not quite linear time, but it’s still quite fast, and a pretty cool application of priority queues. In fact, Dijkstra’s Algorithm (and its variant, A* search) is really the main reason computer scientists need to care about priority queues.