Lecture 30: Graphs: Shortest Paths
10:00 AM, Apr 10, 2018

Contents

1 Recording Paths Taken 1
2 A Weighty Matter 2
3 Dijkstra's Algorithm 3
4 Dijkstra's Algorithm: Complexity 4
5 Dijkstra's Algorithm: Correctness 5
6 Proof of Correctness 5

Objectives

By the end of this lecture, you will be able to:

- record the paths taken by breadth-first traversal; and
- use Dijkstra’s algorithm to find shortest paths in a graph

Note: These notes were written by Professor Amy Greenwald

1 Recording Paths Taken

So far, we’ve seen how to use breadth-first traversal to explore graphs. But what if we want to do more than just discover nodes? For instance, suppose we’re searching a street map for a route from the CIT to Kabob and Curry. If our search only tells us that Kabob is reachable from the CIT, it won’t help us actually find our way there. Fortunately, saving paths involves only a simple modification to our BST function. Here is the pseudocode:

```java
BFT(G, v) {
    Q = new Queue()
    visited = new Set()
    parents = new Map[Node, Node]()
    Q.add(v)
    visited.add(v) // don't re-add start to worklist
```
while(!Q.empty()) {
    v2 = Q.getFirst()
    Q.removeFirst()
    for(w in G.adjs(v2)) {
        if(!visited.contains(w)) {
            Q.add(w)
            visited.add(w) // added to worklist
            parents.put(w, v2) // first time seen
        }
    }
}

When the algorithm finishes, the parents map contains all the paths from reachable nodes back to the starting node v. To extract the path, just follow the entries! The visited set now protects entries in the parents map from ever being overwritten: nodes are added to visited as soon as they are noticed and added to the work list.

One virtue of breadth-first traversal is that the paths taken will always be “shortest” paths, in that they always use a minimal number of edges to go from the source to any given node.

2 A Weighty Matter

Suppose that you’re booking airline tickets. You enter a source airport and a destination, and the system computes potential itineraries. We just saw how the system might use a breadth-first search to find travel plans with the least number of layovers (since, in this toy example, edges correspond to airplane flights). But while minimizing the number of stops is nice, you may have other concerns as well: minimizing distance traveled, perhaps, and certainly cost!

To represent the idea that taking edges may incur a cost, we’ll add weights to our graphs. We won’t worry about whether these represent dollars or hours or some other metric; the point is that now we can speak not just of least-edge paths, but also least-cost paths. The graph below shows an example weighted graph.

Breadth-first traversal won’t produce least-cost paths on this example graph. If we start in Boston, BFT will visit Hartford and Providence before any other cities. The direct hop from Boston to Hartford may be minimal in terms of the number of edges taken, but it costs 300—which is greater than the route via Providence (which costs 30 + 50 = 80). We need an alternative algorithm that takes costs into account.

Make sure that you keep this distinction clear in your mind: sometimes we want to find paths that minimize the number of edges taken, and other times we want to find paths that minimize the cost of edges taken. When people say “shortest path algorithm” outside of this course, they usually mean “minimal-cost path algorithm”.

Dijkstra’s Algorithm

Dijkstra’s algorithm always produces lowest-cost paths. Its structure is heavily inspired by breadth-first traversal. The key idea is that, instead of using a queue to control the order in which nodes get explored, we’ll use a priority queue. The priority of a given node will be our best estimate for the lowest cost to reach it from the starting node. Every time we remove a top-priority node \( v \), we’ll conclude that we’ve found a lowest-cost path to \( v \) from the starting node. We then use that knowledge to refine estimates for the nodes still on the priority queue. Here’s Dijkstra’s algorithm in full:

\[
\text{Dijkstra}(G, v) \{
\begin{align*}
PQ &= \text{new MinPriorityQueue}() \\
\text{dist} &= \text{new Map[Node, Int]}() \\
\text{parents} &= \text{new Map[Node, Node]}()
\end{align*}
\]

\[
\text{for}(w \text{ in } G.\text{nodes}) \{ \\
\quad \text{if}(w == v) \text{ dist}[w] = 0 \\
\quad \text{else} \quad \text{dist}[w] = \text{infinity} \\
\quad \text{PQ.add}(w, \text{dist}[w])
\}
\]

\[
\text{while}(!\text{PQ}.\text{empty}()) \{ \\
\quad v2 = \text{PQ}.\text{getAndRemoveFirst}() \\
\quad \text{for}(w \text{ in } G.\text{adjs}(v2)) \{ \\
\quad \quad \text{newEstimate} = \text{dist}[v2] + G.\text{weight}(v2, w) \\
\quad \quad \text{if}((\text{newEstimate} < \text{dist}[w])) \{ \\
\} \\
\}
\]
dist[w] = newEstimate
PQ.adjustPriority(w, newEstimate)
parents.put(w, v2)
}
}
}

The first portion just sets up the data structures used. PQ is the priority queue. The dist map (short for distance) stores our best estimate for reaching each node. Like in BFT, the parents map will store edges of a shortest-path tree so that we can reconstruct the paths discovered.

We begin by initializing our estimates. Since we're already at the starting node, we estimate it will take 0 to get there. Because we haven't explored the graph yet, we initialize all other estimates to positive infinity. Finally, we place each node on the priority queue according to the current estimate. (Note that, because the starting node has estimate 0, it will be removed first!)

Now we keep removing nodes from the priority queue until it is empty. When a node \( v_2 \) is removed, we implicitly finalize the estimate—it really is the lowest path cost possible to \( v_2 \). Then for each neighbor \( w \) of \( v_2 \), we ask: “Can the current estimate for \( w \) be improved by visiting \( v_2 \) first?” If so, we adjust the estimate for \( w \) and the corresponding priority queue entry to the sum of the (optimal!) cost to reach \( v_2 \) plus the weight of the edge from \( v_2 \) to \( w \).

Let's step through Dijkstra's algorithm on the example graph. As before, we'll start in Boston. After initialization, we should see the following estimates in dist (with corresponding priorities in PQ):

\[
\]

The algorithm removes BOS from the priority queue; our estimate of 0 for reaching Boston is optimal (we mark this with [X]). Now, for each of Boston's neighbors (HAR and PVD) we check to see if we can do better than the current estimate. Since the current estimates are both infinity, and Boston has finite-cost edges to both cities, we can improve both estimates.

1. \([BOS:0 \ [X], PVD:30, HAR:300, ITH:inf, NYC:inf, DC:inf]\)

Now the algorithm removes PVD: 30 is the optimal path length from Boston to Providence. We can improve estimates for 2 of Providence's neighbors: Hartford and New York City. Both now equal the optimal cost to reach Providence (30) plus the length of the direct edge from Providence (50 and 250 respectively):

2. \([BOS:0 \ [X], PVD:30 \ [X], HAR:80, ITH:inf, NYC:280, DC:inf]\)

The algorithm continues until it runs out of nodes on the priority queue, updating estimates as follows:

3. \([BOS:0 \ [X], PVD:30 \ [X], HAR:80 \ [X], ITH:90, NYC:100, DC:inf]\)
4. \([BOS:0 \ [X], PVD:30 \ [X], HAR:80 \ [X], ITH:90 \ [X], NYC:100, DC:291]\)
5. \([BOS:0 \ [X], PVD:30 \ [X], HAR:80 \ [X], ITH:90 \ [X], NYC:100 \ [X], DC:291]\)
6. \([BOS:0 \ [X], PVD:30 \ [X], HAR:80 \ [X], ITH:90 \ [X], NYC:100 \ [X], DC:291[X]\)
4 Dijkstra’s Algorithm: Complexity

What is the performance of Dijkstra’s algorithm? The initialization phase takes $O(n)$, where $n$ is the number of nodes in the graph. The outer loop executes $O(n)$ times, since each node must be removed from the priority queue, and no node is returned to the queue once it is removed. Each node certainly has no more than $n$ neighbors, and the `adjustPriority` operation takes $O(\log n)$ time (since there are at most $n$ items in the queue). This leads us to an initial complexity of $O(n^2 \log n)$.

However, when the number of edges ($m$) in the graph is low, this is a significant over-estimate! The number of neighbors across all nodes is equal to the number of edges. So the inner loop will only execute a total of $m$ times, total. Thus, the true complexity of Dijkstra’s algorithm is $O((n + m) \log n)$—assuming that the graph is represented using an adjacency list. (Using an adjacency matrix would mean iterating over all possible neighbors, but still only performing $m$ queue adjustments, yielding the somewhat worse complexity of $O(n^2 + m \log n)$.)

A note on terminology You may hear people refer to a node of the graph as a “vertex” and vice versa. These terms are equivalent. The same applies to “edge” vs. “arc”.

5 Dijkstra’s Algorithm: Correctness

Dijkstra’s algorithm is correct only if edges have non-negative weights. This is because the core idea is that once a node is visited (i.e., removed from the priority queue), the current estimate is the optimal estimate. This assumption is not always correct if weights can be negative—there might be a later opportunity to recover most, or all, of a high initial cost that Dijkstra’s greedy approach will not detect. Here’s a small example that shows this happening:

![Diagram](image)

Figure 2: Two graphs on which Dijkstra’s algorithm fails (start at node A and node 1 respectively). On the left graph, the algorithm will try to update a weight for a node that has already been removed from the priority queue. This might seem OK at first—the new estimate is correct, after all. But the core problem is that this sort of “late update” can compound over multiple nodes. To see this, consider the right graph. The algorithm computes an incorrect estimate for node 3.

---

2 There are other versions of Dijkstra’s algorithm that are optimized for different situations, but these are beyond the scope of this course! If you take an algorithms class, you will likely see some alternatives that use more sophisticated data structures.
6 Proof of Correctness

**Theorem** Dijkstra’s algorithm is correct, assuming non-negative edge weights. That is, it terminates, and when it does so, \( \text{distance}(v) = \text{distance}^*(v) \), for all vertices \( v \in V \). Here, \( \text{distance} \) denotes the current estimate and \( \text{distance}^* \) denotes the true shortest-path distance.

We prove this theorem as two lemmas. The first states that whenever a node is dequeued (i.e., colored black), a shortest path to that node has been discovered. The second states that all nodes are eventually both enqueued and dequeued.

**Lemma 1** Assume non-negative weights: i.e., \( w(u,v) \geq 0 \) for all \( (u,v) \in E \). Let \( S \) be the set of finalized vertices (i.e., the vertices that have been removed from the priority queue). For all vertices \( v \in S \), \( \text{distance}(v) = \text{distance}^*(v) \). That is, no vertex \( v \) is present in \( S \) unless \( \text{distance}(v) = \text{distance}^*(v) \).

**Proof** The proof is by induction on the size of \( S \).

**Basis** Initially, \( S = \{s\} \), and \( \text{distance}(s) = 0 = \text{distance}^*(s) \).

**Step** Assume \( \text{distance}(u) = \text{distance}^*(u) \), for all vertices \( u \in S \).
Must show \( \text{distance}(v) = \text{distance}^*(v) \) for \( v \), the next item on the priority queue.

The proof proceeds by contradiction. Assume \( \text{distance}(v) > \text{distance}^*(v) \). Then there exists \( y \notin S \) but adjacent to \( S \) (and hence, on the priority queue) s.t. the shortest path to \( v \) goes through \( y \). Furthermore, there exists \( x \in S \) s.t. the shortest path to \( y \) goes through \( x \): i.e., \( \text{distance}^*(y) = \text{distance}^*(x) + w(x,y) \). Now
\[
\text{distance}^*(v) \geq \text{distance}^*(y) \\
= \text{distance}^*(x) + w(x,y) \\
= \text{distance}(x) + w(x,y) \\
= \text{distance}(y) \\
\geq \text{distance}(v)
\]

The first line follows from the fact that the shortest path to \( v \) goes through \( y \) and that weights are non-negative. The second line follows from the choice of \( x \) and \( y \). The third line follows from the definition of \( \text{distance} \). The fourth line follows from the induction hypothesis. The fourth line follows from the definition of \( \text{distance} \). The fifth and last line follows from the fact that \( v \) is next on the priority queue: i.e., \( \text{distance}(v) \leq \text{distance}(y) \), for all \( y \) on the priority queue.

We assumed \( \text{distance}(v) > \text{distance}^*(v) \), and we proved \( \text{distance}^*(v) \geq \text{distance}(v) \). This is a contradiction. So \( \text{distance}(v) \leq \text{distance}^*(v) \). But, by definition, no distance is shorter than \( \text{distance}^*(v) \). Therefore, \( \text{distance}(v) = \text{distance}^*(v) \). ✷
Lemma 2 Assuming non-negative weights, Dijkstra’s algorithm terminates.

Proof Sketch Dijkstra’s algorithm inserts all nodes into the priority queue during its initialization phase. Then, during the main loop of the algorithm, priorities only ever decrease. As above, since priorities are bounded below by 0, the priority queue eventually becomes empty. ◻

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: https://cs.brown.edu/courses/cs018/feedback