Lecture 30: Graphs: Optimal-Cost Paths

11:00 AM, Mar 31, 2021

Contents

Objectives

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

- use Dijkstra's algorithm to find the lowest-cost paths in a graph

Note: These notes were written by Professor Amy Greenwald

1 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

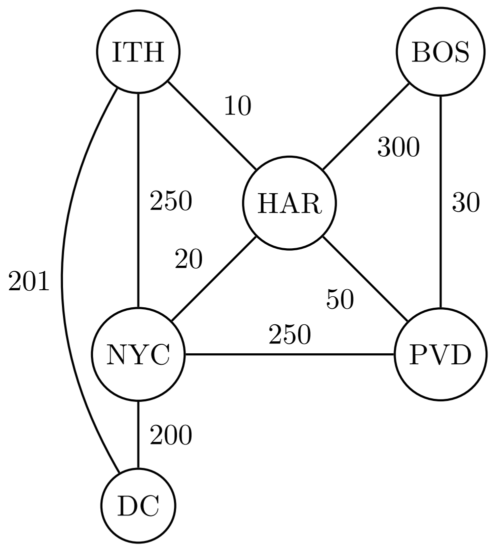


Figure 1: An example weighted graph. Edges are undirected; travel is possible in both directions at the same cost.

of edges taken. **When people say "shortest path algorithm" outside of this course, they usually mean "minimal-cost path algorithm".**

2 Dijkstra's Algorithm

Dijkstra's[1] algorithm always produces lowest-cost paths. Its structure is heavily inspired by breadth-first traveral. 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:

```
Dijkstra(G, v) {
  PQ = new MinPriorityQueue()
  dist = new Map[Node, Int]()
  cameFrom = new Map[Node, Node]()

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

  while(!PQ.empty()) {
```

[1]Named for Edsger Dijkstra, the computer-scientist who invented it.

```
    v2 = PQ.getAndRemoveFirst()
    for(w in G.adjs(v2)) {
      newEstimate = dist[v2] + G.weight(v2, w)
      if(newEstimate < dist[w]) {
        dist[w] = newEstimate
        PQ.adjustPriority(w, newEstimate)
        cameFrom.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 `cameFrom` 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 $v2$ is removed, we implicitly *finalize* the estimate—it really is the lowest path cost possible to $v2$. Then for each neighbor w of $v2$, we ask: "Can the current estimate for w be improved by visiting $v2$ first?" If so, we adjust the estimate for w and the corresponding priority queue entry to the sum of the (optimal!) cost to reach $v2$ plus the weight of the edge from $v2$ 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`):

```
[BOS:0, PVD:inf, HAR:inf, ITH:inf, NYC:inf, DC:inf]
```

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]`

3 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 $O(logn)$ time (since there are at most n items in the queue). This leads us to an initial complexity of $O(n^2logn)$.

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[2] is $O((n+m)logn)$—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 $O(n^2 + mlogn)$.)

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".

4 Dijkstra's Algorithm: Correctness (Optional)

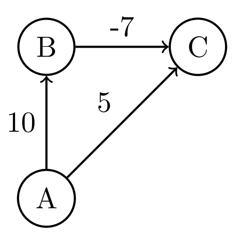
Dijkstra's algorithm is correct only if edges have non-negative weights. This is because the core idea is that once a node is visited (i.e., removed from the priority queue), the current estimate is the optimal estimate. This assumption is not always correct if weights can be negative—there might be 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:

5 Proof of Correctness

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

[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.



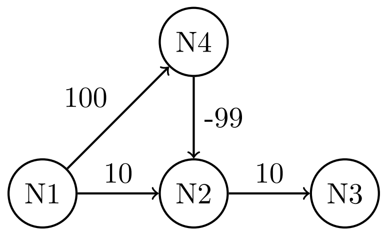
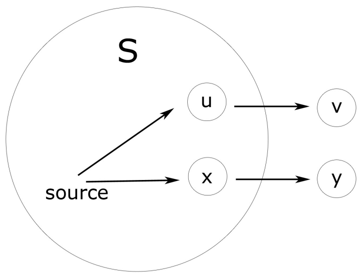


Figure 2: Two graphs on which Dijkstra's algorithm fails (start at node A and node 1 respectively). On the left graph, the algorithm will try to update a weight for a node that has already been removed from the priority queue. This might seem OK at first—the new estimate is correct, after all. But the 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.

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 $v \in S$, $\mathit{distance}(v) = \mathit{distance}^*(v)$. That is, no vertex v is present in S unless $\mathit{distance}(v) = \mathit{distance}^*(v)$.



Proof The proof is by induction on the size of S.

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

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

The proof proceeds by contradiction. Assume $\mathit{distance}(v) > \mathit{distance}^*(v)$. 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., $\mathit{distance}^*(y) = \mathit{distance}^*(x) + w(x, y)$. Now

$$\begin{aligned}
\mathit{distance}^*(v) &\geq \mathit{distance}^*(y) \\
&= \mathit{distance}^*(x) + w(x, y) \\
&= \mathit{distance}(x) + w(x, y) \\
&= \mathit{distance}(y) \\
&\geq \mathit{distance}(v)
\end{aligned}$$

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 induction hypothesis. The fourth line follows from the definition of *distance*. The fifth and last line follows from the fact that v is next on the priority q ueue: i.e., $\mathit{distance}(v) \leq \mathit{distance}(y)$, for all y on the priority queue.

We assumed $\mathit{distance}(v) > \mathit{distance}^*(v)$, and we proved $\mathit{distance}^*(v) \geq \mathit{distance}(v)$. This is a contradiction. So $\mathit{distance}(v) \leq \mathit{distance}^*(v)$. But, by definition, no distance is shorter than $\mathit{distance}^*(v)$. Therefore, $\mathit{distance}(v) = \mathit{distance}^*(v)$. $\diamond$

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. $\diamond$
