Lecture 30: Graphs: Optimal-Cost Paths

11:00 AM, Mar 31, 2021

Contents

Objectives

By the end of this lecture, you will be know:

- how to analyze the running time for Dijkstra's algorithm
- the difference between depth-first and breadth-first search

1 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 $O(n^2logn)$.

1.1 The cost of Adjusting Priorities

Wait – how does `adjustPriority` take only $O(logn)$? Don't we have to both find the item for a specific node in the priority queue ($O(n)$), change the priority (constant), then re-sift the node into its proper place ($O(logn)$)? This sounds like $O(n + logn)$ which is $O(n)$, not $O(logn)$.

This is where details of your data structures and your specific implementation matter in computing runtime. Let's say you store Nodes in a Priority Queue that was defined in a library, and the library doesn't give you access to the internal Priority Queue organization. Then you have to ask the Priority Queue implementation to search for the node whose priority you want to adjust. But instead, what if you built the priority queue manually, and had a way to get from the Nodes you were storing to objects or array indices within the heap in constant time (such as adding an extra hashmap with this info, or storing the heap element objects along with the nodes in your Priority Queue entries. Then you could, in principle, find the items in the heap in constant time.

When you look up explanations of Dijkstra's algorithm online, even from well-regarded sources, you will typically find the $O(logn)$ time reported rather than $O(nlogn)$. You should simply be aware of the assumptions about the data structures that this depends on.

1.2 Refining Runtime with the Number of Edges

Returning to the runtime, we said that the inner loop happens n times, because each node could have an edge to every other node. While this is possible, it isn't common. Usually graphs don't have edges between every pairs of nodes (such as a edge from Providence to every other city, without going through other cities). Indeed, the number of edges is often much smaller than n^2.

As a result, the complexity analyses of graph algorithms often account for nodes and edges separately. Let's use e for the number of edges. Since the number of neighbors across all nodes is equal to the number of edges, the inner loop will only execute a total of e times. Thus, the true complexity of Dijkstra's algorithm[1] is $O((n + e)logn)$—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 e queue adjustments, yielding the somewhat worse complexity of $O(n^2 + elogn)$.)

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".

2 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:

2.1 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.

[1]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.

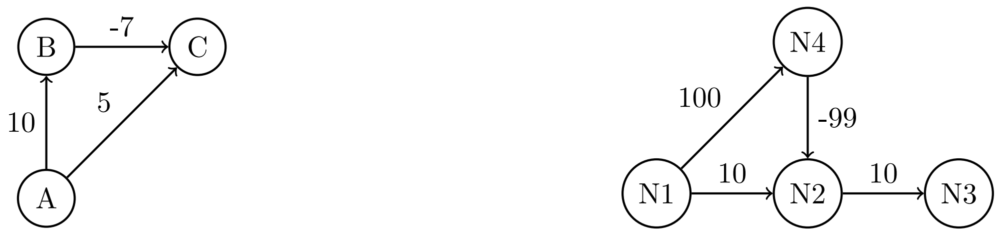
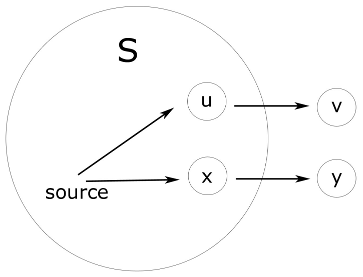


Figure 1: Two graphs on which Dijkstra's algorithm fails (start at node A and node 1 respectively). On the left graph, the algorithm will try to update a weight for a node that has already been removed from the priority queue. This might seem OK at first—the new estimate is correct, after all. But the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

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 $v \in S$, $\mathit{distance}(v) = \mathit{distance}^*(v)$. 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., $distance(v) \leq distance(y)$, for all y on the priority queue.

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

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. $\diamond$

3 Breadth-First vs Depth-First Search

3.1 Depth-First Search

Our `getRouteCheckList` function (from last class) implements a classic graph algorithm known as *depth-first search* (DFS). With depth first search, when a node has more than one outbound edge, we explore all of the paths out of the first edge (in our example, Providence) before we explore paths out of the other edges (Worcester).

Where in the code does this "deep" decision get made? It's in the expression where we augment the `check` list. By putting the nodes reachable from the current node at the *front* of the `check` list, we guarantee we will visit them before we visit nodes that are already pending in the *check* list.

3.2 Breadth-First Search

So what if we made a different decision there, and put the new nodes at the end of the *check* list instead of at the front? In other words, what if our code looked like:

```
for (next <- curr.getsTo) {
   if (!visited.contains(next)) {
      check = check ::: List(next) // the change is here
      cameFrom += (next -> curr)
    }
}
```

Now which order would we visit the nodes in when searching for a Boston-Hartford route?

We'd still start at Boston, putting Providence and Worcester in the *check* list (in that order). When processing Providence, we would put any next-nodes that we hadn't visited *after* Worcester in the list (in this case, there aren't any Providence next-nodes we haven't visited, but that's besides the point). In this way, we wouldn't get stuck in a deep infinite cycle without also making progress on the other paths.

This version is called *breadth-first search* (BFS), because it explores the breadth of next nodes before moving onto their next nodes. This implies that you first check all nodes reachable in one step from the starting point, then all nodes reachable in two steps from the starting point, then all nodes reachable in three steps, and so on.

If we were using breadth-first search, would we still need a visited list? Wouldn't we eventually find the route, before getting stuck in an infinite loop? If there is a route, you would find it without going into an infinite loop. But what if there is no route (as with Hartford to Boston)? Then, you would get stuck in an infinite loop if you didn't have a visited list.

Furthermore, the visited list is still useful for time-efficiency. When your graph has multiple paths of different length to the same node from your starting point, you'd end up exploring that same node for each path, rather than only once.

3.3 Differences between Depth-First and Breadth-First Search

At core, the difference between DFS and BFS is that the former maintains the check list as a *stack*, while the latter maintains it as a *queue*. As we have seen, in code this difference amounts to using `addFirst` versus `addLast` when adding nodes to the list.

Which algorithm (BFS or DFS) should you use in practice? It depends on context.

- If your goal is to find the shortest path length, use BFS. Since BFS checks all nodes at each distance from the starting node before looking at any node at distance + 1, if there are two paths of different lengths to the same node, BFS detects the shortest one first.
- If your goal is detect cycles, use DFS. As soon as DFS tries to process a node that is already in the visited list, you know you have a cycle in the graph. Cycle-detection ends up being a key component of some advanced computing applications – we will study one of these in the next lecture.
- Also, given the shape of your particular graph and characteristics of your route queries, one of BFS or DFS might perform better in practice.

We'll see a couple of interesting applications of these algorithms before the semester ends.

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`.