Lecture 29: Graphs: Shortest Paths (Part 2)

10:00 AM, Apr 9, 2018

Contents

1 The Bellman-Ford Algorithm 1

2 Negative Cycles and Optimal Substructure 2

Objectives

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

- use the Bellman-Ford algorithm to find shortest paths in graphs with negative edge weights;
  and
- use the Bellman-Ford algorithm to detect negative-weight cycles.

1 The Bellman-Ford Algorithm

In the last lecture, we saw that Dijkstra’s algorithm can produce incorrect results if the input graph contains negative edge weights. While negative weights may not make sense if the graph represents something like roads between cities, they can be quite useful for representing refunds, pay, recovery, and so on. So we’d like an algorithm that can find shortest paths in this more general situation. Recall this example graph from last time:

![Example Graph](image)

Figure 1: An example graph (start at node 1)

Dijkstra’s algorithm failed because it finalized the estimate for node 3 before exploring node 4 (and discovering the negative edge). How can we do better? The trick lies in using an idea from earlier in the semester: dynamic programming!

The first step in designing a dynamic programming algorithm is to find a recurrence that defines what we consider a “subproblem” and serves as a template for what data to memoize (top-down) or calculate (bottom-up). The Bellman-Ford recurrence breaks the shortest-path problem down into
subproblems asking: “For an arbitrary destination node \( d \) and maximum path length \( l \), what is the total weight of the shortest path?”

How many subproblems are there? There are only \( n \) nodes to pick from. And any path of length greater than \( n - 1 \) must repeat some node. Indeed, if paths of length \( n \) improve on paths of length \( n - 1 \), we’ll know that the graph doesn’t just have negative edges—it contains a cycle with negative total weight. In the presence of such a cycle, optimal substructure fails (more on this later!) So there are only \( n^2 \) subproblems if we include paths of length 0; \( n \times (n + 1) \) if we want to detect negative cycles. Here is the Bellman-Ford recurrence for a fixed starting node \( s \):

\[
BF_s(d, l) = \begin{cases} 
0, & \text{if } s = d \\
\infty, & \text{if } s \neq d \text{ and } l = 0 \\
\min(BF_s(d, l - 1), \min_{(u,d) \in E} (BF_s(u, l - 1) + w(u, d))), & \text{otherwise}
\end{cases}
\]

When the maximum path length (\( l \)) is 0, we immediately know what the cheapest path is: either we’re trying to go where we already are (no cost) or we don’t yet know how to get there (infinite cost). This is more or less the same as the initialization in Dijkstra’s algorithm.

When the maximum path length (\( l \)) is greater than zero, we check, for every node \( u \) that has an edge to \( d \), if reaching \( u \) in \( l - 1 \) or fewer edges enables a better path to \( d \).

How would we implement this? Ordinary bottom-up dynamic programming, starting with the \( l = 0 \) column (or row, depending on your choice of how to organize the data) of the table first, and continuing until all values or \( l \) are filled; the order in which we visit values of \( d \) doesn’t matter.

**Is there a more space-efficient way?** Yes! It turns out that we only need to keep 1 row of history at a time, since the recurrence never reaches back more than 1 row.

## 2 Negative Cycles and Optimal Substructure

Remember that to apply dynamic programming, we need a problem to have optimal substructure. In the case of shortest path, this holds because if I find a shortest path \((s, v_1, ..., v_n, d)\) from \( s \) to \( d \), every subpath of that path had better also have minimal weight. If there were, for instance, a better route from \( s \) to \( v_n \) than \((s, v_1, ..., v_n)\), we could replace that subpath to improve on the overall answer! So shortest path is well-suited to dynamic programming.

But wait—is that argument actually true? Only if the graph has no cycles with negative total weight. If the graph does have a negative cycle, we need to make sure that the paths we find don’t repeatedly visit the same node (this is usually called “shortest simple path”). Otherwise, the answer diverges! Sadly, in a graph with negative cycles, shortest simple path lacks optimal substructure:

The shortest simple path from \( A \) to \( E \) is \((A, B, C, D, E)\). But the shortest simple path from \( A \) to \( B \) isn’t the direct edge; it’s \((A, F, D, C, B)\). In short, solutions to small subproblems don’t compose well into solutions to larger subproblems. Optimal substructure fails. Because of this, implementations of Bellman-Ford usually continue from \( l = 0 \) onward to \( l = n \) (rather than stopping at \( l - 1 \), the maximum path length without repetition). If there are any improvements vs. the \( l = n - 1 \) row, we can conclude that the graph has negative cycles.
Figure 2: A graph with a negative-weight cycle.

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: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)