Single-Source Shortest Paths: General Lengths

Our argument of correctness for Dijkstra’s algorithm was based on the “time metaphor”: the most imminent prospective event (arrival of an explorer) must take place, exactly because it is the most imminent. This however would not work if we had negative edges. See for example a counter-example in Figure 5.1. Obviously, with negative lengths we need a different algorithm.

![Graph](image_url)

Figure 5.1: Dijkstra’s algorithm would incorrectly calculate the shortest path from A to D as being of length 3 in this graph, whereas the true shortest path has length 2.

We will now develop the Bellman-Ford algorithm which solves this problem. It is what is known as a dynamic programming algorithm — we will spend more time on such algorithms later in the course. We first explain an approach based on a technique called memoization, which is a cousin to dynamic programming. The analogy to keep in mind is dynamic programming is to iteration as memoization is to recursion.

We are given a directed graph $G$, and $s$ is our source vertex that we are finding shortest paths from. Define a function $f(u,k)$ which is the length of the shortest path from $s$ to $u$ using at most $k$ edges. First, let us suppose that we were promised that $G$ has no negative weight cycles. In this case, the length of the shortest path from $s$ to any $u$ would be $f(u,n−1)$, since taking $k > n − 1$ would imply a cycle, which cannot provide any benefit if there are no negative weight cycles. Of course if there are cycles with cumulative negative weight, shortest paths to some vertices may not be defined. For example, imagine a 3-cycle $s \rightarrow A \rightarrow B \rightarrow s$. If all edge lengths are $−1$, then there is no shortest path from $s$ to $B$, since one could loop around the cycle indefinitely to make the total length be arbitrarily negative (i.e. arbitrarily small).

So, how can we calculate $f(u,n−1)$? Recursively!
\[
f(u,k) = \begin{cases} 
0, & \text{if } u = s, k = 0 \\
\infty, & \text{if } u \neq s, k = 0 \\
\min \{f(u,k-1), \min_{v \in V : (v,u) \in E} f(v,k-1) + \text{length}(v,u)\}, & \text{otherwise}
\end{cases}
\]

In words, the base case is \(k = 0\) (a path with zero edges), in which the length of the shortest path is either 0 or \(\infty\) depending on whether \(u = s\) (we also use \(\infty\) as the shortest path distance to represent the lack of a path). If \(k > 0\), then the shortest path from \(s\) to \(u\) of length at most \(k\) is, by the law of the excluded middle, either of length less than \(k\) or exactly \(k\). Thus we simply take the minimum of the two possibilities. The \(f(u,k-1)\) term represents the first possibility. The other term in the minimum, which takes a min over \(v \in V\), represents the other possibility: a path of length \(k\) is simply a path of length \(k - 1\), ending at some node \(v\), followed by the edge \((v,u)\). Shortest paths have the property that all subpaths are themselves shortest paths (justify this to yourself as an exercise!), and thus the path of length \(k - 1\) to \(v\) should itself be a shortest path. Note that above we actually take \(f(v,k-1) + w(v,u)\), and \(f(v,k-1)\) isn’t actually a path of length exactly \(k - 1\), but rather simply at most \(k - 1\). This is OK though, since this only makes the minimum smaller and \(f(v,k-1) + w(v,u)\) is still a valid upper bound on \(f(u,k)\).

As an exercise, you may want to show that implementing this recursive algorithm to obtain \(f(u,n-1)\) would take exponential time in the size of \(G\). But then we do something called memoizing. What is memoizing? We maintain two global arrays \(dp[][]\) and \(seen[][]\). \(seen[u][k]\) is True if we have ever called \(f(\cdot,\cdot)\) with the arguments \(u,k\), and it is False otherwise. If we have computed \(f(u,k)\) somewhere before in the recursion tree, we store the result in \(dp[u][k]\). That way, if we ever come across the same arguments to \(f\) in the future, rather than recompute we can immediately return the answer from the \(dp\) array. This so-called “memoization” approach eliminates entire branches of the recursion tree, speeding up our algorithm (at the expense of extra memory to store the \(dp[][]\) and \(seen[][]\) arrays). See Figure 5.2 for a vanilla recursive implementation of the approach, and Figure 5.3 for the memoized version.

```
Algorithm f(u,k):
1. if k = 0 and u = s: return 0
2. else if k = 0 and u ≠ s: return ∞
3. else:
   ans ← f(u,k-1)
   for (v,u) ∈ E:
      ans ← min(ans, f(v,k-1) + length(v,u))
   return ans
```

Figure 5.2: Basic recursive implementation of the Bellman-Ford algorithm, without memoization. This code’s running time is exponential in the size of the graph.

Now let us analyze running time and space. First, in pre-processing, we form the reverse graph \(G_{rev}\) by reversing all edges in \(E\) then store it in adjacency list representation; this will help us loop over \(\min_{v \in V : (v,u) \in E} \) easily (it is just
global \( dp[1..n][0..(n-1)] \)

global \( seen[1..n][0..(n-1)] \), init to all False

Algorithm \( f(u,k) \):

1. \( \text{if } k = 0 \text{ and } u = s; \text{ return } 0 \)
2. \( \text{else if } k = 0 \text{ and } u \neq s; \text{ return } \infty \)
3. \( \text{else: if } seen[u][k]; \text{ return } dp[u][k] \)
   \( \text{else: } \)
   \( \quad seen[u][k] \leftarrow \text{True} \)
   \( \quad ans \leftarrow f(u,k-1) \)
   \( \quad \text{for } (v,u) \in E; \)
   \( \quad \quad ans \leftarrow \min(ans,f(v,k-1)+\text{length}(v,u)) \)
   \( \quad dp[u][k] \leftarrow ans \)
   \( \text{return ans} \)

Figure 5.3: Memoized recursive implementation of the Bellman-Ford algorithm.

The neighborhood of \( u \) in \( G_{rev} \). Now, fix some \( 0 \leq k \leq n - 1 \) and let us compute the total work to compute all \( f(\cdot,k) \) values. For a fixed \( u \), the total work through the loop to compute \( f(u,k) \) is \( O(1 + \deg_{G_{rev}}(u)) \), where \( \deg_H \) is the out degree in graph \( H \). Thus the total work done across all \( u \) is at most \( \sum_{u \in V} C(1 + \deg_{G_{rev}}(u)) = C(n+m) = O(m + n) \). Thus the total work done across all \( k \) is \( O((m+n)n) \). If the graph has no isolated vertices then \( m \geq n/2 \), which would simplify the running time expression to \( O(mn) \) (and note that we could always filter out isolated vertices in a pre-processing step).

Algorithm BellmanFord(\( G = (V,E),s \)):

\( \quad // dp[1][1] \) stores \( f(u,k-1) \), where initially “\( k-1 \)” is 0
\( \quad // dp[1][2] \) stores \( f(u,k) \)
1. array \( dp[1..n][1] \)
2. array \( dp[1..n][2] \)
3. for \( u \in V; \)
   \( \quad dp[u][1] \leftarrow \infty \)
4. \( dp[s][1] \leftarrow 0 \)
5. for \( k = 1..n-1; \)
6. \( \quad \text{for } u \in V; \)
   \( \quad \quad dp[u][2] \leftarrow dp[u][1] \)
   \( \quad \quad \text{for } (v,u) \in E; \)
   \( \quad \quad \quad dp[u][2] \leftarrow \min(dp[u][2],dp[v][1]+\text{length}(v,u)) \)
   \( \quad \quad \text{for } u \in V; \)
   \( \quad \quad \quad dp[u][1] \leftarrow dp[u][2] \)
7. \( \text{return } dp[1..n][2] \)

Figure 5.4: Bottom-up dynamic programming implementation of the Bellman-Ford algorithm.

What about the space complexity? Storing the “dynamic programming” (DP) table \( dp[u][k] \) would take \( O(n^2) \) space across all \( u,k \), since \( 1 \leq u \leq n \) and \( 0 \leq k < n \). Observe though that \( f(\cdot,k) \) values only depend on \( f(\cdot,k-1) \) values, so an iterative approach of filling in the \( dp[][] \) would at any point in time only need to keep track of two size-\( n \) arrays, by only keeping track of \( dp[][k] \) values when filling out the \( dp[][k+1] \) values. See for example Figure 5.4. This “bottom-up” variant of the algorithm is what is typically referred to as dynamic
programming (again, remember the analogy mentioned above: *dynamic programming is to iteration as memoization is to recursion*). When including the $O(m + n)$ space to store the graph, the bottom-up DP approach uses space $O(m + n)$ instead of $O(n^2)$. It is also possible to implement a slight twist on the DP approach using only a single array instead of two, which always uses time $O(mn)$ instead of $O((m + n)n)$, even in cases when $m \ll n$. See for example Figure 5.5.

Algorithm BellmanFord($G = (V,E), s$):

1. array dist[1..n]
2. for $u \in V$:
   dist[$u$] $\leftarrow \infty$
3. dist[$s$] $\leftarrow 0$
4. for $k = 1...n - 1$:
   for $(v,u) \in E$
     dist[$u$] $\leftarrow \min$ (dist[$u$], dist[$v$] $+$ length($v,u$))
   // return the array of shortest path distances from $s$
5. return dist[1..n]

Figure 5.5: Standard implementation of the Bellman-Ford algorithm found in textbooks, with only a single array.

To prove correctness of the pseudocode in Figure 5.5 (which is in fact the standard way Bellman-Ford algorithm is presented), one would prove the following lemma.

**Lemma 5.1** After completing iteration $k$ of the main *for* loop in Figure 5.5, dist[$u$] is equal to the length of some $s$ to $u$ path in $G$, and furthermore it is at most $f(u,k)$.

Lemma 5.1 can be proven by induction on $k$.

As with Dijkstra’s algorithm, it is of course possible to add a prev array to the Bellman-Ford implementation, so that prev[$u$] stores the vertex previous to $u$ in the shortest path from $s$ to $u$. This provides enough information to reconstruct the actual shortest paths themselves at the end of the main loop.

**Negative Cycles**

So far we have been assuming that the graph $G$ has no negative weight-cycles (or at least none that can be reached from $s$). This is important, since the shortest path problem is ill-posed in graphs with negative cycles. It makes no sense and deserves no answer. Our algorithm in the previous section works only in the absence of negative cycles. (Where did we assume no negative cycles in our correctness argument? Answer: When we asserted that a shortest path from $s$ to $a$ exists! Another answer: when we stated the shortest path has at most $n - 1$ edges, since otherwise we would have cycled and there’s no advantage to cycling.) But it would be useful if our algorithm were
able to detect whether there is a negative cycle in the graph, and thus to report reliably on the meaningfulness of the shortest path answers it provides.

This is easily done. To check whether there are negative weight cycles, we check to make sure that for all \( u \in V \), \( f(u, n) \geq f(u, n - 1) \). If there are no negative weight cycles this test will clearly hold, since a length \( n \) path must contain a cycle, and having no negative weight cycles means that this cannot be advantageous. For the other direction, suppose \( f(u, n) \geq f(u, n - 1) \) for all \( u \). This is the same as saying \( f(u, n - 1) \leq f(v, n - 1) + w(v, u) \) for all \( u, v \in V \). Consider a cycle \( v_1, v_2, \ldots, v_\ell \) in the graph. We just need to show this cycle has nonnegative total weight. For each \( i \) we have

\[
f(v_{i+1}, n - 1) \leq f(v_i, n - 1) + w(v_i, v_{i+1})
\]

where we treat \( \ell + 1 \) as 1 (wrapping around the cycle). Summing over all \( i \) from 1 to \( \ell \), we obtain the inequality

\[
\sum_{i=1}^{\ell} w(v_i, v_{i+1}) \geq 0,
\]

as desired.

### Shortest Paths on Directed Acyclic Graphs (DAG’s)

A “directed acyclic graph” is as the name suggests: a directed graph that does not contain any cycles. There are two subclasses of weighted graphs that automatically exclude the possibility of negative cycles: graphs with non-negative weights and DAG’s. We have already seen that there is a fast algorithm when the weights are non-negative. Here we will give a linear algorithm for single-source shortest paths in DAG’s.

Again this problem can be solved with memoization/dynamic programming. Fixing a source vertex \( s \), define \( f(u) \) to be the length of the shortest path from \( s \) to \( u \). Then we have the recurrence

\[
f(u) = \begin{cases} 
0, & \text{if } s = u \\
\min\{\infty, \min_{v \in V: (v, u) \in E} f(v) + w(v, u)\}, & \text{otherwise}
\end{cases}
\]

The running time and space with memoization are both \( O(m + n) \).

### All pairs shortest paths via dynamic programming

Suppose now we want to calculate the shortest paths between every pair of nodes. One way to do this is to run Dijkstra’s algorithm several times, once for each node (if edge weights are nonnegative). This would take \( O(mn + n^2 \log n) \) time using Fibonacci heaps. For graphs with negative edge weights, we could run Bellman-Ford from each source, taking time \( O(mn^2) \). An algorithm called Johnson’s algorithm can actually solve this problem in \( O(mn + n^2 \log n) \) by running Bellman-Ford once followed by \( n \) Dijkstra calls, though we will not cover it in class.
Here we develop a different solution, called the Floyd-Warshall algorithm. The running time will be \( O(n^3) \).

Let us first assume that there are no negative weight cycles. Like with Bellman-Ford above, we can come up with a recursive solution. Let \( f(u,v,k) \) be the length of the shortest path from \( u \) to \( v \) using only nodes \( 1 \ldots k \) as intermediate nodes. Of course when \( k \) equals the number of nodes in the graph, \( n \), we will have solved the original problem. Again we can compute \( f \) recursively:

\[
\begin{align*}
    f(u,v,k) &= \begin{cases} 
        w(u,v), & \text{if } k = 0 \\
        \min\{f(u,v,k-1), f(u,k,k-1) + f(k,v,k-1)\}, & \text{otherwise}
    \end{cases}
\end{align*}
\]

Again outside the base case we use the law of the excluded middle: either the shortest path when allowed to use intermediate nodes \( 1, \ldots, k \) uses node \( k \) or it doesn’t. We simply take the minimum of the two possibilities. There are \( n^3 \) states and we do \( O(1) \) work for each one (a minimum of two numbers) ignoring recursive calls. Thus with memoization the total running time would be \( O(n^3) \). The space naively would unfortunately also be \( \Theta(n^3) \), but a bottom-up DP implementation would only use \( O(n^2) \) space since \( f(\cdot,\cdot,k) \) values only depend on \( f(\cdot,\cdot,k-1) \) values.

A short implementation of bottom-up Floyd-Warshall follows. We assume that \( d_{ij} \) is set to \( \infty \) for all \( i, j \) such that \( (i,j) \notin E \).

\[
D = (d_{ij}), \text{ distance array, with weights from all } i \text{ to all } j
\]

for \( k = 1 \) to \( n \) do
    for \( i = 1 \) to \( n \) do
        for \( j = 1 \) to \( n \) do
            \[
            D[i,j] = \min(D[i,j], D[i,k] + D[k,j])
            \]

Note that again we can keep an auxiliary array to recall the actual paths. We simply keep track of the last intermediate node found on the path from \( i \) to \( j \). We reconstruct the path by successively reconstructing intermediate nodes, until we reach the ends.

What about if there are negative weight cycles?

**Exercise.** Show that there is a negative weight cycle in \( G \) if and only if for some \( 1 \leq i \leq n, D[i,i] < 0 \) at the end of running Floyd Warshall.

**Some Pointers on Point-to-Point Algorithms**
Naturally, one can use Dijkstra’s algorithm to solve the point-to-point shortest paths problem. Indeed, for non-negative distances, one can stop as soon as the other point is taken off the priority queue. But for real speed, one can optimize in various ways, or use alternative methods. Here are just a few ideas that have been used in algorithms for this problem.

- Bidirectional Dijkstra’s algorithm: start searches from \( s \) and \( t \) and alternate between them, and terminate when the searches meet in the middle. The stopping criterion is not trivial. The idea is that if the graph branches a lot, two “small” searches might require less exploration than one large one.

- Changing distances with potential functions: one can associated a potential \( \pi(x) \) with each vertex \( x \) and change each distance \( d(u,v) \) to \( d(u,v) + \pi(v) - \pi(u) \). If \( \pi(s) = 0 \) for the source vertex and the new distances are nonnegative, then the shortest paths remain the same (check why) and Dijkstra’s algorithm can still be used. Clever choices of potential functions can speed up the algorithm.

- Using landmarks and precomputation: by precomputing the distance between a small collection of landmark locations and nearby points, and precomputing all-pairs shortest paths between landmarks, one can quickly determine an upper bound on the shortest path between two points: add up the distance from the start point to its nearest landmark, the distance from the end point to its nearest landmark, and the distance between landmarks. Using this information, one can prune Djikstra’s algorithm when finding point-to-point shortest paths.

- Hierarchical methods: In real world networks, there are often natural partitions of the underlying space. For example, for a long trip, it may be clear that first you have to get on a highway, then take the highway, then get off the highway. To cross a body of water, you may have to go through one of a small number of bridges. Taking advantage of such hierarchical features can speed up calculations.

Sections 1-3 of the survey at \[\text{http://research.microsoft.com/pubs/207102/MSR-TR-2014-4.pdf}\] has a lot of good background on this problem. (The rest of the survey examines the problem of traveling via public transport, where timetables are involved.)