

16 Shortest Paths: Part 2

In this chapter we will look at how to compute shortest paths in graphs where edge weights may be negative. In particular, we will see an algorithm to solve the single source shortest path problem which is efficient even when edges can have arbitrary weights. Then we will look at an even more general shortest-path problem, where we want to find the shortest path between *every* pair of vertices and give a clever dynamic programming algorithm to solve it.

16.1 Negative Weights

Negative weights introduce some new complexities to the concept of a shortest path. Recall that a path in a graph cannot repeat vertices. In many practical applications of shortest path algorithms this is an unnatural constraint. If going in a circle somehow allowed you to get from point A to point B faster, people would be willing to do it. That is, often what we are actually interested in computing is often a shortest *walk*. In a graph with non-negative edge weights, this difference doesn't matter. Repeating a vertex can never make our walk shorter, so for any s and t , there will be a shortest $s \rightarrow t$ walk which is a path (there could theoretically also be shortest $s \rightarrow t$ walks which are not paths, if the graph has a zero-weight cycle).

With negative weights, however, the notion of a shortest walk sometimes stops making sense. In particular, if our graph contains a *cycle* of negative weight which is reachable from s , then we can obtain arbitrarily short walks from s by walking to the cycle, walking around it until we feel like our walk length is sufficiently negative, and then walking to our destination. Thus, even when dealing with negative weights, we prefer to focus on graphs without negative-weight cycles (in these graphs it remains the case that some shortest $s \rightarrow t$ walk is always a path). It's not obvious how to determine whether a graph has a negative-weight cycle, but we'll see an algorithm that can do this (in addition to efficiently solving single source shortest paths with negative weight edges) in the next section.

Although Theorem 72 only stated the correctness of Dijkstra's algorithm for non-negative edge weights, Dijkstra's algorithm does correctly compute a shortest path tree even for graphs with negative edge weights, as long as the graph does not have a negative-weight cycle. The runtime, however, can be extremely bad, $\Theta(2^n)$ in the worst case (See Section 8.6 of Erickson for an example where Dijkstra's runs very slowly). With negative edges, we no longer have a guarantee that each vertex is extracted from our priority queue at most once, so we can end up relaxing the same edges over and over and over again. Note that if G has a negative-weight cycle, then Dijkstra's algorithm can run forever, continually relaxing edges round and round the cycle. We need a better approach if we want to quickly find shortest paths in graphs with negative edge weights.

16.2 Negative Edge Weights: Bellman-Ford

Perhaps surprisingly, the algorithm for handling negative edge weights is simpler than Dijkstra's algorithm. Indeed the algorithm we will describe in this section was actually discovered before Dijkstra's algorithm. It was first described by Alfonso Shimbil in 1955, but was popularized by Richard Bellman and Lester Ford Jr. who described it in later papers and is today known as the Bellman-Ford algorithm.

Dijkstra's algorithm can be viewed as a greedy algorithm, focusing on relaxing the edges of whichever vertex currently has the lowest distance upper bound. The Bellman-Ford algorithm, on the other hand, is a brute force algorithm. Rather than strategically selecting edges to relax, the algorithm simply relaxes all edges in the graph, and then repeats this process until there are no more tense edges. As we will prove below, this takes at most $n - 1$ iterations, so if there are still tense edges after $n - 1$ rounds of relaxation our algorithm can correctly report that there is a negative cycle.

```

BELLMAN-FORD( $G = (V, E), s$ )
   $D, pred \leftarrow$  length  $n$  arrays
  For each  $v \in V$ :
     $D[v] \leftarrow \infty$ 
     $pred[v] \leftarrow \text{Null}$ 
   $D[s] \leftarrow 0$ 
  Repeat  $n - 1$  times:
    For each edge  $(u, v) \in E$ :
      If  $(u, v)$  is tense:
        RELAX( $u, v$ )
  If  $G$  contains a tense edge:
    Return "Negative Cycle!"
  Return  $D, pred$ 

```

Theorem 74. For every weighted directed graph $G = (V, E)$ and every vertex $s \in V$:

- If G contains a negative-weight cycle reachable from s , BELLMAN-FORD(G, s) correctly reports this.
- Otherwise, BELLMAN-FORD(G, s) returns arrays D and $pred$ such that for every $v \in V$, $D[v] = \text{dist}(v)$ and (if $\text{dist}(v) \neq \infty$ and $v \neq s$) a shortest $s \rightarrow v$ path ends with the edge $(pred[v], v)$.

Furthermore, BELLMAN-FORD(G, s) runs in time $\Theta(mn)$

Proof. We first note that if G has a negative-weight cycle reachable from s , then no matter how much we relax edges, G will always have a tense edge. (every time we relax an edge in the cycle, the next one becomes tense). Thus, if G has a negative-weight cycle reachable from s , BELLMAN-FORD will correctly report this.

Now we will assume that G has no negative-weight cycles reachable from s . We will prove by induction on i that after i iterations of the “Repeat $n - 1$ times” loop, $D[v] = \text{dist}(v)$ for every vertex v such that some shortest $s \rightarrow v$ path (i.e. the one of minimum total weight) uses at most i edges.

- Base Case: The only vertex whose shortest path from s uses 0 edges is s itself. Since we set $D[s]$ to 0 before the loop, after 0 iterations of the loop we have that $D[v] = \text{dist}(v)$ for every vertex v for which the shortest $s \rightarrow v$ path has at most 0 edges.
- Induction Hypothesis: Suppose that after $i - 1$ iterations of the loop, $D[v] = \text{dist}(v)$ for every vertex v such that some shortest $s \rightarrow v$ path uses at most $i - 1$ edges.
- Suppose $i > 0$. By our IH, at the start of iteration i of the loop, every vertex v for which some shortest $s \rightarrow v$ has fewer than i edges already has $D[v] = \text{dist}(v)$. We note that such a vertex cannot have any tense incoming edges, since this would imply that there is a walk to v shorter than $\text{dist}(v)$ (which would correspond to a path of length less than $\text{dist}(v)$ since G has no negative cycles). Thus, all these vertices will still have $D[v] = \text{dist}(v)$ after this loop iteration.

It remains to consider the vertices v for which the shortest $s \rightarrow v$ path with a minimum number of edges has exactly i edges. Let v be such a vertex and let $v_1, v_2, \dots, v_{i+1}$ be a shortest $s \rightarrow v$ path (so $v_1 = s$ and $v_{i+1} = v$).

We note that by the optimal substructure of shortest paths, $v_1, v_2, \dots, v_i$ is a shortest $s \rightarrow v_i$ path, and uses only $i - 1$ edges. Thus, by our IH, $D[v_i] = \text{dist}(v_i)$. Also, since $v_1, \dots, v_i, v_{i+1}$ is a shortest $s \rightarrow v$ path, $\text{dist}(v) = \text{dist}(v_i) + w(v_i, v) = D[v_i] + w(v_i, v)$. Thus, if $D[v] > \text{dist}(v)$, this iteration of the loop will relax the edge (v_i, v) and $D[v]$ will become $\text{dist}(v)$.

Since every vertex reachable from s must be reachable on a path of at most $n - 1$ edges, we have proven that after all $n - 1$ iterations of the for loop, all of our distance upper bounds in D will be tight, and therefore our shortest path tree will be correct.

Finally, we analyze the runtime. The “repeat $n - 1$ times” loop runs $\Theta(n)$ times and the “for each” loop runs $\Theta(m)$ times. Checking if an edge is tense and relaxing it takes constant time, so the total work of the “repeat” loop is $\Theta(mn)$. The work of the algorithm outside of the loop can also be done in $O(mn)$ time, so the overall runtime is $\Theta(mn)$. $\square$

16.3 All Pairs Shortest Paths: Floyd-Warshall

In this section we consider a generalization of the shortest paths problem, all pairs shortest paths (APSP), where the goal is to compute the shortest paths between every pair of vertices in a graph. We can represent the solutions to all pairs shortest paths using n shortest path trees (one rooted at each vertex) represented as predecessor arrays as we've done for single source shortest paths. To simplify the algorithm and its analysis, however, in this section we will only focus on simply computing the *length* of all shortest paths (although the algorithm we give could be extended to create a shortest path tree for each vertex). That is, our goal is to compute a 2-D array D where $D[i, j]$ is the length of the shortest path between vertex i and vertex j .

The obvious way to compute all pairs shortest paths is to simply run a single source shortest path algorithm n times, starting once from each vertex in G . If our graph is unweighted, we can use BFS and this takes $O(n(m + n)) = O(mn + n^2)$ time. If it has non-negative edge weights we can use Dijkstra's and this takes $O(n(m + n \log n)) = O(mn + n^2 \log n)$ time. If our graph has arbitrary edge weights, then we can use Bellman-Ford and this takes $O(n(mn)) = O(mn^2) \leq O(n^4)$ time. However, for graphs with arbitrary edge weights, there is a faster approach.

We can solve APSP faster using dynamic programming. As usual, the key is coming up with the right recurrence. While there are several ways that define a recurrence that we can memoize into a polynomial time algorithm, the clever choice that allows us to obtain a DP algorithm faster than repeatedly running Bellman-Ford is probably not the first one that you would think of. Assume again that our vertex set is $\{1, \dots, n\}$. We define $APSP(i, j, k)$ as the length of the shortest $i \rightarrow j$ path which uses only vertices from $\{1, \dots, k\} \cup \{i, j\}$.

We give the following recurrence

$$APSP(i, j, k) = \begin{cases} 0 & \text{if } k = 0 \text{ and } i = j \\ w(i, j) & \text{if } k = 0 \text{ and } (i, j) \in E \\ \infty & \text{if } k = 0, i \neq j, \text{ and } (i, j) \notin E \\ \min \begin{cases} APSP(i, j, k-1) \\ APSP(i, k, k-1) + APSP(k, j, k-1) \end{cases} & \text{otherwise} \end{cases}$$

Here the recursive case considers the choice of whether or not to use vertex k in our $i \rightarrow j$ path.

We can memoize the function $APSP$ into a 3-D array $D[1, \dots, n][1, \dots, n][0, \dots, n]$. Each entry $D[i, j]$ depends only on entries with a smaller value of k $D[\cdot, \cdot, k-1]$, so we fill the array from smallest k to largest (we arbitrarily choose to, for a given k , fill in order of increasing values of i and j).

```

FLOYD-WARSHALL( $G = (V, E)$ )
   $D[1, \dots, n][1, \dots, n][0, \dots, n] \leftarrow$  an array of integers
  For  $i$  from 1 to  $n$ :
    For  $j$  from 1 to  $n$ :
       $D[i, j, 0] \leftarrow \infty$ 
      If  $i = j$ :
         $D[i, j, 0] \leftarrow 0$ 
      If  $(i, j) \in E$ :
         $D[i, j, 0] \leftarrow w(i, j)$ 
  For  $k$  from 1 to  $n$ :
    For  $i$  from 1 to  $n$ :
      For  $j$  from 1 to  $n$ :
         $D[i, j, k] \leftarrow \min\{D[i, j, k-1], D[i, k, k-1] + D[k, j, k-1]\}$ 
  Return  $D[\cdot, \cdot, n]$ 

```

This algorithm is known as the Floyd-Warshall algorithm, after Robert Floyd and Stephen Warshall who described it in 1962.

Theorem 75. *For every weighted directed graph $G = (V, E)$ which does not contain a negative-weight cycle, FLOYD-WARSHALL(G) returns an array D such that for every pair of vertices $s, t \in V$, $D[s, t] = \text{dist}(s, t)$. Furthermore, FLOYD-WARSHALL runs in $\Theta(n^3)$ time.*

Proof. The correctness of this algorithm follows from the correctness of the recurrence: As we noted above, the recursive case of our recurrence considers the choice of whether or not to use vertex k in our $i \rightarrow j$ path. If the shortest $i \rightarrow j$ path which uses only vertices from $\{1, \dots, k\} \cup \{i, j\}$ does not use vertex k , then it's the same as the shortest $i \rightarrow j$ path which uses only vertices from $\{1, \dots, k-1\} \cup \{i, j\}$, and its length is $APSP(i, j, k-1)$. Otherwise, the shortest $i \rightarrow j$ path which uses only vertices from $\{1, \dots, k\} \cup \{i, j\}$ uses vertex k , so it consists of taking a shortest $i \rightarrow k$ path followed by a shortest $k \rightarrow j$ path, and its length is $APSP(i, k, k-1) + APSP(k, j, k-1)$. A standard induction argument shows that $D[i][j][k]$ is the length of the shortest $i \rightarrow j$ path using only vertices from $\{1, \dots, k\} \cup \{i, j\}$ for all valid i, j, k .

We note that if $k = n$, $\{1, \dots, k\} \cup \{i, j\} = \{1, \dots, n\} = V$ for any i, j , so $D[i, j, n]$ is just the length of the shortest $i \rightarrow j$ path in G . Thus, the final 2-D array $D[:, \cdot, n]$ that we return contains the length of the shortest $i \rightarrow j$ path for every pair of vertices i and j .

Now we analyze the runtime. Our array has $\Theta(n^3)$ entries, and we can compute each one in $\Theta(1)$ time (each entry only requires look at/summing a constant number of other entries). Therefore the total runtime is $\Theta(n^3) \cdot \Theta(1) = \Theta(n^3)$.

Finally, we note that since our DP table has $\Theta(n^3)$ entries, the space complexity is $\Theta(n^3)$. $\square$

We note that although Theorem 75 assumed that the input graph has no negative cycles, it's not a big deal to handle negative cycles when computing APSP. Before running Floyd-Warshall we can check for negative cycles using BELLMAN-FORD in $O(mn) = O(n^2n) \leq O(n^3)$ time, which will not increase the overall runtime of the algorithm (Bellman-Ford only finds negative-weight cycles reachable from the starting vertex, but we can guarantee that all negative cycles will be reachable from our starting point by creating a new vertex s which has an out-edge to every vertex of G and running Bellman-Ford starting from this vertex. The runtime remains $O(n^3)$).

Until recently it was commonly believed that APSP was unlikely to admit an $o(n^3)$ algorithm (i.e. that Floyd-Warshall was asymptotically the best algorithm possible). However, in October 2026, Alman and Williams published an $O(n^{2.9995})$ algorithm for APSP (based on an algorithm initially developed by an internal model at Anthropic), narrowly improving over Floyd-Warshall. Determining the actual best possible runtime for APSP remains an open area of research.

16.4 Comprehension Questions

1. True or false? If $G = (V, E)$ is a graph with negative edge weights, but no negative-weight cycle, and $s \in V$ then DIJKSTRA'S(G, s) will correctly compute $dist(s, v)$ for each vertex $v \in V$.
2. True or false? If $G = (V, E)$ is a directed graph where every cycle has positive-weight, and $s, t \in V$, then every shortest $s \rightarrow t$ walk is a path.
3. Let $G = (V, E)$ be a graph, $s \in V$, and $e \in E$. When running BELLMAN-FORD(G, s), what is the maximum number of times that e could be relaxed?
4. In our correctness proof for BELLMAN-FORD we showed that, assuming G has no negative cycles, after i iterations of the "repeat $n-1$ times" loop, BELLMAN-FORD will have $D[v] = dist(v)$ for which vertices v ?
5. Asymptotically, how many distinct subproblems does the Floyd-Warshall algorithm use?
6. In English, what does $APSP(i, j, k)$ represent (where $APSP$ is the recurrence used by FLOYD-WARSHALL)?

16.5 Exercises

1. Suppose that you are given a directed graph $G = (V, E)$ with n vertices and m edges where each edge e has an arbitrary integer edge weight $w(e) \in \mathbb{Z}$, as well as two vertices $s, t \in V$. Your goal is to find the shortest $s \rightarrow t$ path which uses an even number of edges (or correctly report that there is no $s \rightarrow t$ path with an even number of edges). You may assume that G

does not contain a negative-weight cycle. Give an efficient algorithm for this problem and analyze its runtime in terms of m and/or n .

2. Give an $O(m + n)$ time algorithm which takes as input a weighted directed acyclic graph $G = (V, E)$ and a vertex $s \in V$ and outputs an array giving the distance from s to t for each $t \in V$. (Note: the key property that will allow you to design a faster algorithm than BELLMAN-FORD is that the graph is a DAG).
3. Barterville is a town which has not invented money, so all purchases are made by exchanging one kind of fruit for another. Each merchant in Barterville sells exactly one kind of fruit, and has a list of other fruit they accept in payment, as well as how much of the other fruit they consider equivalent to one of their fruits. For example, Alvin the apple merchant will exchange 1 apple for either 3 strawberries, 20 blueberries, or 0.1 watermelons.

You are an enterprising but lazy young merchant in Barterville. You have done your market research by collecting a list of the n fruits currently being sold in the Barterville market. You've also collected a list of the m fruit pairs that can be exchanged, and for each pair (a, b) you've recorded a value $q(a, b)$, the quantity of fruit b which can be traded for 1 of fruit a .

Your goal is to find a way to make a profit without producing any fruit of your own. In particular, you want to find a way to start with some fruit and perform a sequence of trades using the prices set by the Barterville merchants which will leave you with more of your original fruit than you started with. For example, if you can trade 0.1 watermelons for 1 apple, 0.05 apples for 1 blueberry, and 100 blueberries for 1 watermelon, then you can make a profit by starting with a watermelon, trading it for 10 apples, trading those for 200 blueberries, and trading the blueberries for 2 watermelons.

Give an algorithm to determine whether or not there is a sequence of trades that will allow you to make a profit. Your algorithm should run in $O(mn)$ time.

4. Suppose that you are given a graph G with integer edge weights that can be positive, negative, or zero, but with no negative length cycle. Give an $O(mn)$ time algorithm to determine whether or not G contains a zero-length cycle (Hint: perturb the edge weights slightly then use an algorithm from this chapter). Make sure you justify why your algorithm always finds a zero-length cycle when one exists, but never reports a zero-length cycle when none exists.
5. Suppose that you are given a strongly connected directed weighted graph $G = (V, E)$, as well as a subset $A \subseteq E$ of the edges which are "awesome". You may assume that G has no negative-weight cycle. You want to find, for every pair of vertices $u, v \in V$, the shortest $u \rightarrow v$ walk which uses at least one awesome edge. Give an $O(n^3)$ time algorithm for this problem.