Shortest Paths with Negative Weights

In the previous lecture, we studied Dijkstra's algorithm for finding shortest paths in graphs with non-negative edge weights. However, its greedy strategy depends crucially on this restriction. What if our graph contains negative-weight edges? Can we still find shortest paths efficiently?

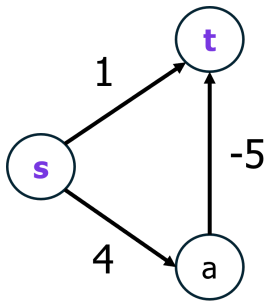
In this lecture, we will develop several algorithms that support negative edge weights, using ***dynamic programming*** and the **subpaths property** of shortest paths. We will begin with directed acyclic graphs, where shortest paths can be computed in linear work. We will then generalize to graphs with cycles using the **Bellman-Ford algorithm**, and finally consider the all-pairs shortest-path problem using the **Floyd-Warshall algorithm**. Along the way, we will also learn how to detect negative-weight cycles, which can make shortest paths undefined.

1 Shortest Paths in the Presence of Negative Weights

1.1 Why the Problem is Harder

Why does Dijkstra's algorithm require non-negative edge weights? Recall that its greedy strategy relies on the fact that extending a path cannot decrease its weight. When negative edges are allowed, this property no longer holds, as the following example demonstrates.

Consider the following graph:



Starting from s, Dijkstra's algorithm would initially discover paths to t and a with weights 1 and 4, respectively. It therefore greedily finalizes t with distance 1. But the actual shortest path to t is $s \to a \to t$, with weight $4 + (-5) = -1$! Dijkstra's algorithm misses this path because it finalizes t before exploring a, because it assumes that the weight of a path can not decrease.

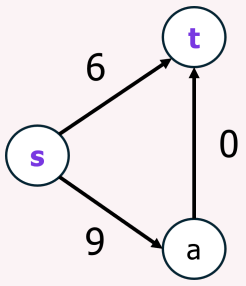
The problem is that a path with a larger initial weight can become cheaper after traversing a negative-weight edge. Thus, the greedy property that makes Dijkstra's algorithm correct no longer holds, so we need a different algorithmic approach.

Remark: Can we remove negative weights?

One tempting idea is to add a sufficiently large constant c to every edge weight, making all weights non-negative, and then run Dijkstra's algorithm. Unfortunately, this does not preserve shortest paths. Adding c to every edge increases the weight of a path P by c times its length:

$$w'(P) = w(P) + c \cdot \text{length}(P).$$

Consequently, paths with more edges become disproportionately expensive, potentially changing which path is shortest. For example, adding 5 to every edge in our graph gives:

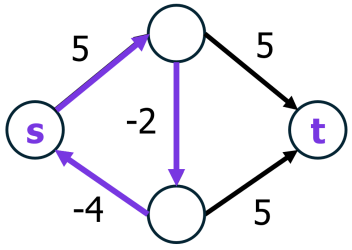


Now the direct path $s \to a$ has weight 6, while $s \to a \to t$ has weight $9+0=9$. The ordering of the two paths has reversed! Thus, simply adding a constant to the edge weights does not solve the problem.

1.2 Negative-Weight Cycles

Recall that a **negative-weight cycle** is a cycle whose edge weights sum to a negative value. Importantly, not every edge in the cycle needs to have negative weight, only their sum.

As we discussed in the previous lecture, negative-weight cycles can make shortest paths undefined. If a negative-weight cycle is reachable from s, and we can reach t from that cycle, then we can construct an s-t path that traverses the cycle arbitrarily many times. Each traversal decreases the path's weight, so there is no minimum-weight path from s to t.



Notice that a negative-weight cycle does not necessarily invalidate *every* shortest path in the graph. It only affects pairs of vertices for which the cycle can be reached from the source and the destination can be reached from the cycle.

Fortunately, *negative edges alone are not a problem.* As long as there are no negative-weight

cycles, shortest paths are well-defined between all reachable pairs of vertices. We will develop algorithms that compute these shortest paths efficiently, and learn how to detect negative-weight cycles when they exist.

2 Shortest Paths and Dynamic Programming

Recall the **subpaths property** from the previous lecture:

Theorem: Subpaths Property

Any contiguous subpath of a shortest path is itself a shortest path between its endpoints.

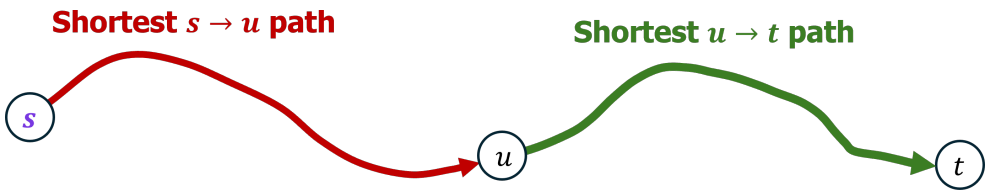
In particular, if u lies on a shortest path from s to t, then

$$\delta(s,t) = \delta(s,u) + \delta(u,t).$$

This property holds regardless of whether the graph has negative edge weights, provided the shortest path exists. The subpaths property is extremely useful because it allows us to express shortest paths in terms of smaller shortest paths. This suggests that we might be able to solve shortest-path problems using **dynamic programming**.

Splitting a shortest path. Consider a shortest path from vertex s to vertex t. The subpaths property says that for any vertex u on a shortest path, the resulting paths if we split at vertex u are themselves shortest, i.e.,

$$\delta(s,t) = \delta(s,u) + \delta(u,t).$$



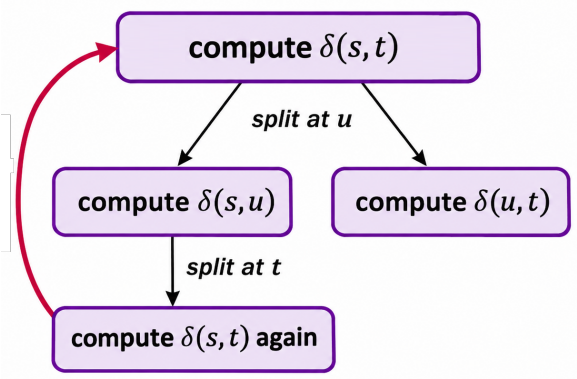
Of course, we do not know in advance *which* vertices lie on a shortest s-t path. However, there must be *some* vertices on the path (at least s and t themselves[1]), so in general it is true that

$$\boxed{\delta(s,t) = \min_{u \in V} \big(\delta(s,u) + \delta(u,t)\big).}$$

This is a general recurrence for shortest paths, and it holds even in the presence of negative-weight edges! However, it is not immediately useful as an algorithm.

For example, say we are trying to evaluate $\delta(s,t)$ and considering splitting the path at vertex u, so we evaluate $\delta(s,u)$ and $\delta(u,t)$. The subproblem $\delta(s,u)$ may then try to split the path at t and evaluate $\delta(s,t)$, which then leads to infinite recursion!

[1] If the shortest s-t path happens to be a single edge (s,t), note that the equation is still trivially true for $u = t$ because $\delta(s,t) = \delta(s,t) + \delta(t,t)$, although it would not be useful as a recurrence—-this would be a base case.

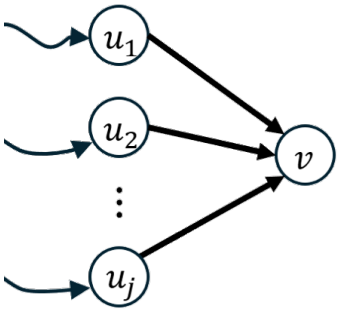


To turn these relationships into efficient dynamic programs, we need to ensure that the dependencies between subproblems are acyclic. We will see three different ways to accomplish this, leading to three shortest-path algorithms.

3 Shortest Paths on DAGs

Our general recurrence from the previous section considers all possible ways to split a shortest path into two shorter paths. Let us now consider a more specific natural way of splitting: *splitting the path into a path with one fewer edge and its final edge.*

Suppose we are solving the single-source shortest-path problem from a source s. For any destination $v \neq s$, the last edge of a shortest path must be (u, v) for some in-neighbor u of v.



By the subpaths property, the portion from s to u must itself be shortest and have weight $\delta(s,u)$, while the remaining portion is just the single edge (u,v), with weight $\delta(u,v) = w(u,v)$.

Thus, specializing our general recurrence to this particular way of splitting gives

$$\delta(s,v) = \min_{u \in N^-(v)} \big(\delta(s,u) + w(u,v)\big).$$

where $N^-(v)$ denotes the set of in-neighbours of v. As usual, the minimum over an empty set is defined to be ∞, so a vertex with no in-neighbours other than s itself is unreachable from s.

On a general directed graph, evaluating the recurrence recursively might lead to cyclic dependencies. However, on a **directed acyclic graph (DAG)**, the dependencies are guaranteed to be acyclic. We can therefore solve the single-source shortest-path problem by evaluating the following dynamic program.

Algorithm: Shortest Paths on a DAG (Dynamic Programming)

Subproblems: For each vertex $v \in V$, define

$$\text{Dist}(v) = \text{the shortest-path weight from } s \text{ to } v.$$

If v is unreachable from s, then $\text{Dist}(v) = \infty$.

Recurrence:

$$\text{Dist}(v) = \begin{cases} 0, & \text{if } v = s, \\ \min\limits_{u \in N^-(v)} \big(\text{Dist}(u) + w(u,v)\big), & \text{otherwise.} \end{cases}$$

Because the graph is acyclic, the recurrence is well defined and the recursion always terminates. Memoization ensures that each subproblem is evaluated only once.

Theorem: Shortest Paths on DAGs

Single-source shortest paths on a DAG with n vertices and m edges can be computed in $O(n+m)$ work, even with negative edge weights.

Proof. There are n subproblems, one for each vertex. Evaluating $\text{Dist}(v)$ requires considering all of its in-neighbors, taking $O(1+|N^-(v)|)$ work. Since memoization ensures that each subproblem is evaluated only once, the total work is

$$O\left(\sum_{v \in V} (1 + |N^-(v)|)\right) = O(n+m),$$

because the sum of all in-degrees is m. □

Notice that this algorithm makes no assumptions about the signs of the edge weights. The absence of cycles is sufficient to make the dynamic programming recurrence viable.

Remark: Topological Ordering

Another way to evaluate the same recurrence is to process the vertices in **topological order**. Recall that in a DAG, a topological ordering places every vertex u before all vertices v for which (u,v) is an edge.

Thus, when we reach a vertex v, the distances to all of its in-neighbors have already been computed. We can evaluate each subproblem directly, without recursion or memoization. This is a bottom-up implementation of exactly the same dynamic program, with the same $O(n+m)$ work bound.

4 k-Hop Shortest Paths and Bellman-Ford

4.1 k-Hop Shortest Paths

In the previous section, we saw that shortest paths on DAGs can be computed efficiently using dynamic programming. However, our recurrence cannot be evaluated directly on a general directed graph, since cycles in the graph can lead to cyclic dependencies between subproblems.

Can we modify our dynamic programming formulation to avoid this problem, without requiring the graph itself to be acyclic? Recall how we derived our shortest-path recurrence: we split a path into a shorter path followed by its last edge. In particular, if a path has k edges, removing its last edge leaves a path with $k-1$ edges.

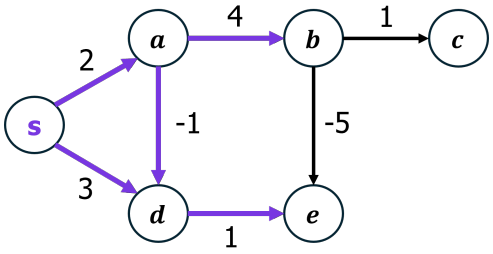
This suggests a way to *strengthen our subproblems*: rather than considering arbitrary shortest paths, let us restrict the number of edges they may contain.

Definition: k-Hop Shortest Paths

A **k-hop shortest path** from u to v is a minimum-weight path from u to v among all paths containing *at most k edges*. Its weight is called the **k-hop distance**. If no such path exists, the k-hop distance is ∞.

Unlike ordinary shortest paths, k-hop shortest paths are well-defined even in the presence of negative-weight cycles, since the number of edges is bounded. Note that we do not require these paths to be simple: they may visit the same vertex multiple times.

Consider the following graph, and suppose we want to find the 2-hop shortest-path distances from s to every vertex.



The 2-hop shortest paths from s are:

- **To s:** The empty path, with weight 0.
- **To a:** $s \to a$, with weight 2.
- **To b:** $s \to a \to b$, with weight $2+4=6$.
- **To c:** No path with at most two edges exists, so the 2-hop distance is ∞.
- **To d:** $s \to a \to d$, with weight $2+(-1)=1$, which is better than the direct edge of weight 3.
- **To e:** $s \to d \to e$, with weight $3+1=4$. Although $s \to a \to d \to e$ is cheaper, it has three edges.

Remark: Other Uses for k-Hop Shortest Paths

k-hop shortest paths are also useful when restricting the number of edges in a path is part of the problem. For example, suppose we want to find the cheapest flight itinerary from one city to another, but are willing to take at most two flights. We would then want to find a 2-hop shortest path.

From k-hop distances to shortest paths. How does solving the k-hop shortest-path problem help us solve the unrestricted shortest-path problem? Recall that if a shortest path exists, there is always a simple shortest path, which contains at most $n-1$ edges. This immediately gives us:

Corollary: Enough Hops

In a graph with no negative-weight cycles, the shortest-path distances are exactly the $(n-1)$-hop distances.

4.2 Dynamic Programming for k-Hop Distances

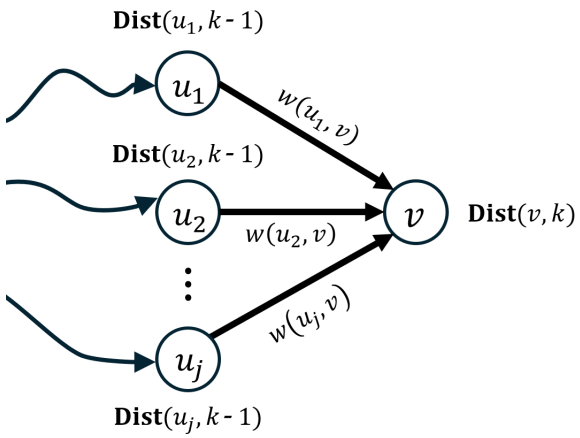
We can now use k-hop shortest paths to strengthen our recurrence from Section 3. Instead of defining one subproblem for each vertex, we define one subproblem per vertex *and hop limit*. The base cases are straightforward: with zero edges, we can only reach the source s itself, which is distance zero away from itself.

For $k \geq 1$, consider a k-hop shortest path from s to some vertex v. There are two possibilities:

- The path contains *at most* $k-1$ *edges*. In this case, its best possible weight is $\text{Dist}(v, k-1)$.
- The path contains *exactly* k *edges*. In this case, we can remove its last edge (u, v), leaving a path with $k-1$ edges to some in-neighbor u. We consider every possible choice of u, giving

$$\min_{u \in N^-(v)} \big(\text{Dist}(u, k-1) + w(u, v)\big).$$

Taking the minimum of these two possibilities gives us the recurrence.



Algorithm: Dynamic Programming for k-Hop Distances

Subproblems: For every vertex $v \in V$ and integer $k \geq 0$, define

$$\text{Dist}(v,k) = \text{the } k\text{-hop shortest-path weight from } s \text{ to } v.$$

Recurrence:

$$\text{Dist}(v,k) = \begin{cases} 0, & \text{if } k = 0 \text{ and } v = s, \\ \infty, & \text{if } k = 0 \text{ and } v \neq s, \\ \min\left\{ \begin{array}{l} \text{Dist}(v,k-1), \\ \min\limits_{u \in N^-(v)} \big(\text{Dist}(u,k-1) + w(u,v)\big) \end{array} \right\}, & \text{otherwise.} \end{cases}$$

Notice the crucial difference from our original recurrence: every subproblem $\text{Dist}(v,k)$ depends only on subproblems with a *smaller value of* k. Even if the graph contains cycles, the dependencies between these new subproblems cannot cycle, because k strictly decreases along every dependency. Thus, ordinary dynamic programming works on arbitrary directed graphs, including those with negative-weight edges or cycles!

Sequential implementation. We can implement the dynamic program by maintaining a 2D array `dist[v][k]` containing the k-hop distance to each vertex v. One subtlety is that the recurrence requires the **in-neighbors** of each vertex. We therefore use a **reverse adjacency list**, where `in_adj[v]` contains pairs (u, w) representing incoming edges (u, v) of weight w.

Algorithm: k-Hop Shortest-Path Distances

```
type RevAdjList = sequence<sequence<(int, real)>>

fun khop_dists(in_adj: RevAdjList, s: int, K: int) -> sequence<real>:
  n = |in_adj|
  dist = [[∞ for _ in 0...K] for _ in 0...n-1]
  dist[s][0] ← 0

  for k in 1...K:
    for v in 0...n-1:
      dist[v][k] ← dist[v][k-1]
      for u, w in in_adj[v]:
        dist[v][k] ← min(dist[v][k], dist[u][k-1] + w)

  return [dist[v][K] for v in 0...n-1]
```

Each iteration computes the next column of the dynamic programming subproblems using only the previous column. In particular, the updates for different vertices never depend on one another within the same iteration.

For a hop limit K, this algorithm takes $O(K(n+m))$ work and $O(n(K+1))$ space. In the next

sections, we will see how to use this dynamic program to obtain the Bellman-Ford algorithm, and how to parallelize its computation.

4.3 Detecting Negative-Weight Cycles

Recall that if a graph has no negative-weight cycles, the $(n-1)$-hop distances are exactly the shortest-path distances. However, what if the graph contains a negative-weight cycle? Our algorithm would still terminate and return some distances, but they would not necessarily be correct! Ideally, we would like our shortest-path algorithm to detect this situation and report that the input contains a negative-weight cycle.

Fortunately, we can detect negative-weight cycles using just one additional iteration of our dynamic program.

Theorem: Identifying Negative-Weight Cycles

A graph contains a negative-weight cycle reachable from the source s if and only if there exists a vertex v such that

$$\text{Dist}(v, n) < \text{Dist}(v, n-1).$$

Proof. First, suppose that for some vertex v, $\text{Dist}(v, n) < \text{Dist}(v, n-1)$. Then the n-hop shortest path to v must use exactly n edges. Since the graph has only n vertices, this path must repeat some vertex, and therefore contain a cycle.

If all cycles along this path had non-negative weight, we could remove them without increasing its weight, obtaining a path with at most $n-1$ edges. This would contradict the fact that the n-hop distance is strictly smaller. Hence, the path must contain a negative-weight cycle, which is reachable from s.

For the other direction, suppose a negative-weight cycle is reachable from s. Let x be any vertex on this cycle, and let the cycle have ℓ edges and total weight $c < 0$. Since x is reachable, there is a path from s to x using at most $n-1$ edges, so $\text{Dist}(x, n-1)$ is finite.

Now suppose, for the sake of contradiction, that the n-hop distances are identical to the $(n-1)$-hop distances for every vertex. Notice that our recurrence computes each layer entirely from the previous layer. Thus, if two consecutive layers are identical, all subsequent layers must also be identical! In particular,

$$\text{Dist}(x, k) = \text{Dist}(x, n-1) \qquad \text{for all } k \geq n-1.$$

But consider taking an $(n-1)$-hop shortest path to x and then traversing the negative-weight cycle once. This gives a path using at most $n-1+\ell$ edges, with weight

$$\text{Dist}(x, n-1) + c < \text{Dist}(x, n-1).$$

This contradicts our assumption that the distances never improve after $n-1$ hops. □

Thus, to detect a negative-weight cycle, we simply compute the n-hop distances and compare them against the $(n-1)$-hop distances. If any distance decreases, we report a negative-weight cycle. Otherwise, we return the $(n-1)$-hop distances, which are guaranteed to be correct.

Notice that this detects only negative-weight cycles *reachable from the source*. Cycles elsewhere in the graph do not affect our single-source shortest-path problem.

4.4 Sequential Bellman-Ford

We now have everything we need to solve single-source shortest paths on graphs with negative edge weights: compute the $(n-1)$-hop distances, then perform one additional iteration to detect negative-weight cycles. The standard textbook implementation of Bellman–Ford uses the same underlying idea, but makes two implementation choices:

- **Outgoing-edge relaxation:** Instead of computing each vertex's distance from its in-neighbors using a reverse adjacency list, we iterate over outgoing edges and update their destination vertices. Each update is an edge *relaxation*, as in Dijkstra's algorithm.
- **In-place updates:** Instead of keeping separate distance estimates for successive hop limits, we maintain a single array and update it in place. Consequently, iteration k may already discover paths with *more* than k edges.

In-place updates can only improve distance estimates earlier. After iteration k, every estimate is less than or equal to its k-hop distance, so $n-1$ iterations still suffice. This gives us the standard textbook implementation of the Bellman-Ford algorithm:

Algorithm: Sequential Bellman-Ford

```
type WeightedAdjList = sequence<sequence<(int, real)>>

fun bellmanford(adj: WeightedAdjList, s:int)->option<sequence<real>>:
  n = |adj|
  dist = [∞ for _ in 0...n-1]
  dist[s] ← 0

  for k in 1...n:
    for u in 0...n-1:
      for v, w in adj[u]:
        if dist[u] + w < dist[v]:
          if k == n: return NONE  // Negative cycle!
          dist[v] ← dist[u] + w

  return SOME(dist)
```

Each iteration iterates over all m edges, so the algorithm costs $O(nm)$ work. In-place updates can allow distance estimates to converge in fewer than $n-1$ iterations, depending on the order in which edges are processed. A common optimization is to terminate early if an entire iteration makes no updates. However, the worst-case cost remains $O(nm)$.

Remark: Historical note: What Is Bellman-Ford?

The name Bellman–Ford encompasses several closely related shortest-path algorithms developed in the 1950s[a]. Richard Bellman invented the dynamic-programming-based k-hop formulation, while Lester Ford developed an algorithm that relaxes any edge that can improve a distance estimate, repeating until no more improvements are possible.

The standard textbook implementation is a combination of both: repeatedly relax *every edge* in the graph (similar to Ford's algorithm) for exactly $n-1$ iterations (justified by the k-hop argument of Bellman's algorithm), followed by one addition iteration to detect negative-weight cycles.

[a] A similar algorithm was discovered by Alfonso Shimbel prior to Bellman's and Ford's algorithms, although as was common in the pre-internet area, the original inventor of an algorithm often did not become its namesake due to worse publicity.

4.5 Parallel Bellman-Ford

How can we parallelize Bellman-Ford? The in-place updates of the standard sequential implementation introduce dependencies between edge relaxations, making them difficult to parallelize. Instead, we can return to our original k-hop dynamic program from Section 4.2.

Computing the k-hop distances requires only the $(k-1)$-hop distances, thus the iterations over k must remain sequential, but *within* each iteration, all the k-hop distances can be computed in parallel. Computing the distance to each vertex requires taking a minimum over its in-neighbors, which we can implement using a parallel `reduce`. We can therefore compute each iteration using a parallel `tabulate` over the vertices and a `reduce` over each vertex's in-neighbours.

As with our original k-hop implementation, we use a reverse adjacency list to efficiently `map` over each vertex's in-neighbors to compute candidate k-hop distances.

Algorithm: Parallel Bellman-Ford

```
type RevAdjList = sequence<sequence<(int, real)>>

fun bellman_ford(in_adj: RevAdjList, s: int)->option<sequence<real>>:
  n = |in_adj|

  fun hop(prev: sequence<real>, k: int) -> sequence<real>:
    return tabulate(fn v =>
        min(prev[v],
          reduce(min, ∞, map(fn (u,w) => prev[u] + w, in_adj[v]))),
      n)

  init = tabulate(fn v => 0 if v == s else ∞, n)
  dist = fold_left(hop, init, 1...n-1)

  if hop(dist, n) != dist: return NONE  // Negative cycle!
  return SOME(dist)
```

The function `hop` takes the $(k-1)$-hop distances and computes the k-hop distances in parallel. It is a pure function of the $(k-1)$-hop distances; it creates a new sequence rather than modifying the previous one. As before, after $n-1$ iterations, we perform one additional iteration to check for negative-weight cycles.

Theorem: Cost of Parallel Bellman-Ford

Parallel Bellman-Ford costs $O(nm)$ work and $O(n \log n)$ span.

Proof. Each iteration computes the distances for all n vertices using a parallel `tabulate`. For each vertex v, we compute candidate distances through all its in-neighbors and take their minimum using a parallel `reduce`.

The work for vertex v is $O(|N^-(v)|)$. Summing over all vertices gives $O(m)$ work per iteration, since the sum of the in-degrees is m. Since each vertex has at most n in-neighbors, `reduce` costs $O(\log n)$ span. All vertices are processed in parallel, so the span of each iteration is $O(\log n)$.

The n iterations are performed sequentially, giving $O(nm)$ total work and $O(n \log n)$ span. □

5 All-Pairs Shortest Paths: Floyd-Warshall

We now turn to the **all-pairs shortest-path (APSP)** problem: given a weighted directed graph, we want to find the shortest-path distances between *every pair* of vertices. One straightforward approach is to run a single-source shortest-path algorithm n times, once from each source vertex. Since these computations are independent, we can also run them in parallel.

Theorem: All-Pairs Shortest Paths via SSSP

All-pairs shortest paths can be computed in $O(nm \log n)$ work and $O(m \log n)$ span for graphs with non-negative weights, or in $O(n^2 m)$ work and $O(n \log n)$ span for graphs with negative edge weights but no negative-weight cycles.

Proof. Run Dijkstra's algorithm or parallel Bellman-Ford n times, once from each source vertex. The work is multiplied by n, but since all n computations are independent, they can run in parallel without increasing the span. □

Can we do better, especially on dense graphs? We will develop another dynamic programming algorithm, **Floyd-Warshall**, that solves APSP in $O(n^3)$ work and is also highly parallelisable.

Recall the **subpaths property** from Section 2: any subpath of a shortest path is itself shortest. In our previous algorithms, we used this property by splitting off the *last edge* of a shortest path. However, we can also split a shortest path at *any intermediate vertex*.

5.1 A Dynamic Programming Approach for APSP

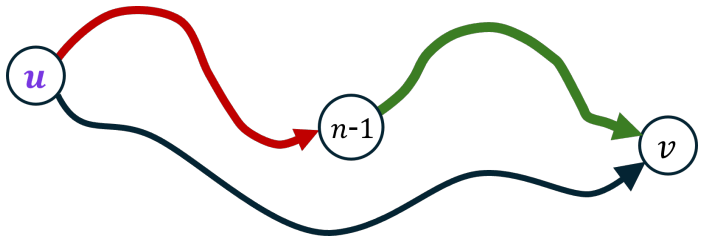
Recall our general recurrence:

$$\delta(u,v) = \min_{x \in V}\big(\delta(u,x) + \delta(x,v)\big).$$

As we discussed in Section 2, this recurrence cannot be evaluated directly using dynamic programming because its subproblems have cyclic dependencies. In Bellman-Ford, we solved this problem by restricting the number of edges in a path. Can we find a similar way to make the subproblems smaller here?

A different way to split paths. Suppose the vertices are numbered $0, 1, \ldots, n-1$, and we want to find a shortest path from u to v. Consider just one question: *Does the shortest path use vertex $n-1$ as an intermediate vertex?*

- **If not,** then we can find the shortest path without using vertex $n-1$ as an intermediate vertex.
- **If so,** then we can split the path at vertex $n-1$, into a path from u to $n-1$ and a path from $n-1$ to v. Crucially, since a shortest path never has to repeat vertices, neither of these two subpaths needs to use $n-1$ as an intermediate vertex!



Notice that *in either case*, we have reduced the set of allowed intermediate vertices by one. We can now repeat the same reasoning for vertex $n-2$, then $n-3$, and so on. This is much like the dynamic programming strategy for **Knapsack**: consider one element at a time, decide whether to use it, and recursively solve smaller subproblems.

This suggests *strengthening* our subproblems by keeping track of *which vertices are allowed as intermediate vertices* on the path. In particular, we can parameterize over $0 \le k \le n$ such that the path may only use the first k vertices as as intermediate vertices.

We can then apply the same general reasoning to derive a recurrence. Suppose we are to find the shortest path weight between vertex u and v, using only the first k vertices (i.e., the vertices $\{0, \ldots, k-1\}$) on the path between u and v. Such a path either:

- **Does not use vertex $k-1$:** Its best possible weight is $\text{Dist}(u, v, k-1)$.
- **Uses vertex $k-1$:** We split the path at $k-1$, giving two subpaths that do not need to use $k-1$ as an intermediate vertex. By the subpaths property, their minimum combined weight is

$$\text{Dist}(u, k-1, k-1) + \text{Dist}(k-1, v, k-1).$$

Taking the minimum of these two possibilities gives us a dynamic programming algorithm.

Algorithm: Dynamic Programming for All-Pairs Shortest Paths

Subproblems: For every pair of vertices $u, v \in V$ and integer $0 \le k \le n$, define:

$$\text{Dist}(u,v,k) := \begin{cases} \text{the shortest-path weight from } u \text{ to } v \text{ using} \\ \text{only intermediate vertices in } \{0,\dots,k-1\}. \end{cases}$$

Base cases:

$$\text{Dist}(u,v,0) = \begin{cases} 0, & \text{if } u = v, \\ w(u,v), & \text{if } (u,v) \in E, \\ \infty, & \text{otherwise.} \end{cases}$$

Recurrence: For $k \ge 1$,

$$\text{Dist}(u,v,k) = \min \left\{ \begin{array}{l} \text{Dist}(u,v,k-1), \\ \text{Dist}(u,k-1,k-1) + \text{Dist}(k-1,v,k-1) \end{array} \right\}.$$

Notice that every subproblem with parameter k depends only on subproblems with parameter $k-1$. Thus, the dependencies are acyclic and this is a valid dynamic programming algorithm.

Cost analysis. There are $O(n^3)$ subproblems: one for each pair of vertices u, v and each value of k. Each subproblem takes only $O(1)$ work to evaluate, so the total work is $O(n^3)$.

The key insight is that each subproblem requires only a *binary decision*: use vertex $k-1$, or do not use it. Unlike our original shortest-path recurrence, we do not need to try *every* possible vertex at which to split the path. This allows each subproblem to be evaluated in constant work, but even more importantly, makes the subproblems acyclic.

5.2 The Floyd-Warshall Algorithm

One downside of the above algorithm is that it uses a lot of space, $O(n^3)$, which is a factor n larger than the input graph. This is bad if the graph is large. Can we reduce this? There are two ways. First, notice that the subproblems for parameter k only depend on the subproblems with parameter $k-1$. So, we don't actually need to store all $O(n^3)$ subproblems, we can just keep the last set of subproblems and compute bottom-up in increasing order of k.

Here's an even simpler but more subtle way to optimize the algorithm. Similar to Bellman-Ford, we don't actually need to keep the intermediate results for all previous values of k. We can simply *update the distances in-place* as we go. This gives us the **Floyd-Warshall algorithm**.

Graph representation. Unlike all of our previous algorithms which work best with an adjacency list representation, the Floyd-Warshall algorithm works best with an *adjacency matrix*. This is because it isn't updating paths by extending them by a single edge anymore, so individual edges are less relevant. What it needs is the fastest way possible to look up the distance between any pair of vertices, which is what an adjacency matrix is optimized for.

Recall that when $k = 0$, no intermediate vertices are allowed, so our base cases are precisely the adjacency matrix: the shortest path weight between u and v using no intermediate vertices is just $w(u, v)$, with zeros on the diagonal and infinity for missing edges. We can therefore initialize the distance matrix to the weighted adjacency matrix and update it as we consider each successive intermediate vertex.

Algorithm: Floyd-Warshall

```
type WeightedAdjMatrix = sequence<sequence<real>>

fun floyd_warshall(D : WeightedAdjMatrix) -> WeightedAdjMatrix:
  n = |D|

  for k in 0...n-1:
    for u in 0...n-1:
      for v in 0...n-1:
        D[u][v] ← min(D[u][v], D[u][k] + D[k][v])

  return D
```

After iteration k, the matrix contains the shortest-path distances allowing intermediate vertices in $\{0, \ldots, k\}$. The algorithm takes $O(n^3)$ work, since it considers all n^2 pairs of vertices for each of the n iterations, and uses only $O(n^2)$ space for the distance matrix.

Remark: Why do in-place updates work?

At first glance, updating the matrix in place seems problematic: our recurrence requires distances from the *previous* iteration, but some entries may already have been updated in the current iteration.

However, there is a subtle observation that makes this work. During iteration k, the entire k^{th} row and column remain unchanged, i.e., $D[u][k]$ and $D[k][v]$ for all u, v *do not change* during iteration k.

To see why, recall that in a graph without negative-weight cycles, the shortest distance from any vertex to itself is zero, so $D[k][k] = 0$. Therefore, when $u = k$, the update is

$$D[k][v] \leftarrow \min(D[k][v], \underbrace{D[k][k]}_{0} + D[k][v]) = D[k][v].$$

Similarly, when $v = k$, the update is

$$D[u][k] \leftarrow \min(D[u][k], D[u][k] + \underbrace{D[k][k]}_{0}) = D[u][k].$$

Thus, the entries $D[u][k]$ and $D[k][v]$ used in every update are exactly the values from the previous iteration! Consequently, the in-place implementation computes exactly the same recurrence as our original dynamic program, and is therefore correct.

5.3 Parallel Floyd-Warshall

How can we parallelize Floyd-Warshall? As with Bellman-Ford, the iterations over k must remain sequential since each iteration depends on the distances computed by the previous one. However, within each iteration, the distances for all n^2 pairs of vertices are independent.

To make this parallelism explicit, we return to our original dynamic programming formulation, computing each iteration from the previous matrix rather than updating it in place. Each pair of vertices requires only a minimum of two values, so we can compute all n^2 distances in parallel using two nested `tabulate`s.

Algorithm: Parallel Floyd-Warshall

```
fun floyd_warshall(D: WeightedAdjMatrix) -> WeightedAdjMatrix:
  n = |D|

  fun next_k(prev: WeightedAdjMatrix, k: int) -> WeightedAdjMatrix:
    return tabulate(fn u => tabulate(fn v =>
        min(prev[u][v], prev[u][k] + prev[k][v]),
      n), n)

  return fold_left(next_k, D, 0...n-1)
```

The function `next_k` takes the distances allowing intermediate vertices in $\{0,\ldots,k-1\}$ and computes the distances additionally allowing vertex k. Like parallel Bellman-Ford, we use a sequential `fold_left` to combine the iterations, while each iteration is computed in parallel. Unlike Bellman-Ford, we do not need a parallel `reduce`, since each subproblem only requires taking the minimum of *two values*.

Theorem: Cost of Parallel Floyd-Warshall

Parallel Floyd-Warshall costs $O(n^3)$ work and $O(n)$ span.

Proof. Each iteration computes n^2 distances using nested parallel `tabulate`s. Each distance requires only $O(1)$ work and span, so an iteration costs $O(n^2)$ work and $O(1)$ span. The n iterations are performed sequentially by `fold_left`, giving $O(n^3)$ total work and $O(n)$ span. □

As an exercise, think about how to extend this algorithm to detect the presence of negative-weight cycles. Currently, it just returns a matrix of distances, some of which may be incorrect because of the existence of negative-weight cycles.