4

Shortest Paths in Graphs

In this chapter, we look at another basic algorithmic construct: given a graph where edges have weights, find the shortest path between two specified vertices in it. Here the weight of a path is the sum of the weights of the edges in it. Or given a source vertex, find shortest paths to all other vertices. Or find shortest paths between all pairs of vertices in the graph. Of course, each harder problem can be solved by multiple calls of the easier ones, but can we do better?

Let us give some notation. The input is a graph \( G = (V, E) \), with each edge \( e = uv \) having a weight/length \( w_{uv} \in \mathbb{R} \). For most of this chapter, the graphs will be directed: in this case we use the terms *edges* and *arcs* interchangeably, and an edge \( uv \) is imagined as being directed from \( u \) to \( v \) (i.e., from left to right). Given a *source* vertex \( s \), the *single-source shortest paths (SSSP)* asks for the weights of shortest paths from \( s \) to each other vertex in \( V \). The *all-pairs shortest paths (APSP)* problem asks for shortest paths between each pair of vertices in \( V \). (In the worst-case, algorithms for the \( s-t \)-shortest-path problem also solve the SSSP, so we do not consider these any further.) We will consider both these variants, and give multiple algorithms for both.

There is another source of complexity: whether the edge-weights are all non-negative, or if they are allowed to take on negative values. In the latter case, we disallow cycles of negative weight, else the shortest-path is not well-defined, since such a cycle allows for ever-smaller shortest paths as we can just run around the cycle to reduce the total weight arbitrarily.

4.1 Single-Source Shortest Path Algorithms

The *single-source shortest path problem* (SSSP) is to find a shortest path from a single source vertex \( s \) to every other vertex in the graph. The output of this algorithm can either be the \( n - 1 \) numbers giving the weights of the \( n - 1 \) shortest paths, or (some compact representa-
tion of) these paths. We first consider Dijkstra’s algorithm for the case of non-negative edge-weights, and give the Bellman-Ford algorithm that handles negative weights as well.

4.1.1 Dijkstra’s Algorithm for Non-negative Weights

Dijkstra’s algorithm keeps an estimate $\text{dist}$ of the distance from $s$ to every other vertex. Initially the estimate of the distance from $s$ to itself is set to 0 (which is correct), and is set to $\infty$ for all other vertices (which is typically an over-estimate). All vertices are unmarked. Then repeatedly, the algorithm finds an unmarked vertex $u$ with the smallest current estimate, marks this vertex (thereby indicating that this estimate is correct), and then updates the estimates for all vertices $v$ reachable by arcs $uv$ thus:

$$\text{dist}(v) \leftarrow \min\{\text{dist}(v), \text{dist}(u) + w_{uv}\}$$

We keep all the vertices that are not marked and their estimated distances in a priority queue, and extract the minimum in each iteration.

**Algorithm 2: Dijkstra’s Algorithm**

- **Input:** Digraph $G = (V, E)$ with edge-weights $w_e \geq 0$ and source vertex $s \in G$
- **Output:** The shortest-path distances from each vertex to $s$

1. add $s$ to heap with key 0
2. for $v \in V \setminus \{s\}$ do
   3. add $v$ to heap with key $\infty$
4. while heap not empty do
   5. $u \leftarrow \text{deletemin}$
   6. for $v$ a neighbor of $u$ do
      7. key($v$) $\leftarrow \min\{\text{key}(v), \text{key}(u) + w_{uv}\}$ // relax $uv$

To prove the correctness of the algorithm, it suffices to show that each time we extract a vertex $u$ with the minimum estimated distance from the priority queue, the estimate for that vertex $u$ is indeed the distance from $s$ to $u$. This can be proved by induction on the number of marked vertices, and left as an exercise. Also left as an exercise are the modifications to return the shortest-path tree from node $s$.

The time complexity of the algorithm depends on the priority queue data structure. E.g., if we use binary heap, which incurs $O(\log n)$ for decrease-key as well as extract-min operations, we incur a running time of $O(m \log n)$. But just like for spanning trees, we can do better with Fibonacci heaps, which implement the decrease-key operation in constant amortized time, and extract-min in $O(\log n)$ time. Since Dijkstra’s algorithm uses $n$ inserts, $n$ delete-mins, and $m$ decrease-keys, this improves the running time to $O(m + n \log n)$. This update step is often said to relax the edges out of $u$, which has a nice physical interpretation. Indeed, any edge $uv$ for which the $\text{dist}(v)$ is strictly bigger than $\text{dist}(u) + w_{uv}$ can be imagined to be over-stretched, which this update fixes.
There have been many other improvements since Dijkstra’s original work. If the edge-weights are integers in \(\{0, \ldots, C\}\), a clever priority queue data structure of Peter van Emde Boas\(^1\) can be used instead; this implements all operations in time \(O(\log \log C)\). Carefully using it can give us runtimes of \(O(m \log \log C)\) and \(O(m + n \sqrt{\log C})\) (see Ahuja et al.\(^2\)). Later,\(^3\) showed a faster implementation for the case that the weights are integer, which has the running time of \(O(m + n \log \log(n))\) time. Currently, Give latest results.

4.1.2 The Bellman-Ford Algorithm

Dijkstra’s algorithm does not work on instances with negative edge weights. For example, it will return a distance of 2 for the vertex \(a\), whereas the correct shortest-path from \(s\) to \(a\) goes via \(b\), and has weight \(3 - 2 = 1\). For such instances, a correct SSSP algorithm must either return the distances from \(s\) to all other vertices, or else find a negative-weight cycle in the graph.

The most well-known algorithm for this case is the Shimbel-Bellman-Ford algorithm.\(^4\) Just like Dijkstra’s algorithm, this algorithm also starts with an overestimate of the shortest path to each vertex. However, instead of relaxing the out-arcs from each vertex once (in a careful order), this algorithm relaxes the out-arcs of all the vertices \(n - 1\) times, in round-robin fashion. Formally, the algorithm is the following. (A visualization can be found \(^5\).)

\begin{algorithm}
\begin{algorithmic}[1]
\State **Input:** A digraph \(G = (V, E)\) with edge weights \(w_e \in \mathbb{R}\), and source vertex \(s \in V\)
\State **Output:** The shortest-path distances from each vertex to \(s\), or report that a negative-weight cycle exists
\State \(\text{dist}(s) = 0\) // the source has distance 0
\For {\(v \in V\)}
\State \(\text{dist}(v) \leftarrow \infty\)
\EndFor
\For {\(|V|\) iterations}
\For {edge \(e = (u,v) \in E\)}
\State \(\text{dist}(v) \leftarrow \min\{\text{dist}(v), \text{dist}(u) + \text{weight}(e)\}\)
\EndFor
\EndFor
\If {any distances changed in the last \(n^{th}\) iteration, output “\(G\) has a negative weight cycle”}
\EndIf
\end{algorithmic}
\end{algorithm}

The proof relies on the following lemma, which is easily proved by induction on \(i\).

Lemma 4.1. After \(i\) iterations of the algorithm, \(\text{dist}(v)\) equals the weight of the shortest-path from \(s\) to \(v\) containing at most \(i\) edges. (This is defined to be \(\infty\) if there are no such paths.)

If there is no negative-weight cycle, then the shortest-paths are

\(^1\) This algorithm also has a complicated history. The algorithm was first stated by Shimbel in 1954, then Moore in ’57, Woodbury and Dantzig in ’57, and finally by Bellman in ’58. Since it used Ford’s idea of relaxing edges, the algorithm “naturally” came to be known as Bellman-Ford.
well-defined and simple, so a shortest-path contains at most $n - 1$
edges. Now the algorithm is guaranteed to be correct after $n - 1$
iterations by Lemma 4.4; moreover, none of the distances will change
in the $n^{th}$ iteration.

However, suppose the graph contains a negative cycle that is
reachable from the source. Then the labels $\text{dist}(u)$ for vertices on
this cycle continue to decrease in each subsequent iteration, because
we may reach to any point on this cycle and by moving in that cy-
kle we can accumulate negative distance; therefore, the distance will
get smaller and smaller in each iteration. Specifically, they will de-
crease in the $n^{th}$ iteration, and this decrease signals the existence of
a negative-weight cycle reachable from $s$. (Note that if none of the
negative-weight cycles $C$ are reachable from $s$, the algorithm out-
puts a correct solution despite $C$’s existence, and it will produce the
distance of $\infty$ for all the vertices in that cycle.)

The runtime is $O(mn)$, since each iteration of Bellman-Ford looks
at each edge once, and there are $n$ iterations. This is still the fastest
algorithm known for SSSP with general edge-weights, even though
faster algorithms are known for some special cases (e.g., when the
graph is planar, or has some special structure, or when the edge
weights are “well-behaved”). E.g., for the case where all edge weights
are integers in the range $[-C, \infty)$, we can compute SSSP in time
$O(m\sqrt{n}\log C)$, using an idea we may discuss in Homework #1.

### 4.2 The All-Pairs Shortest Paths Problem (APSP)

The obvious way to do this is to run an algorithm for SSSP $n$ times,
each time with a different vertex being the source. This gives an
$O(mn + n^2 \log n)$ runtime for non-negative edge weights (using $n$
runs of Dijkstra), and $O(mn^2)$ for general edge weights (using $n$
runs of Bellman-Ford). Fortunately, there is a clever trick to bypass this ex-
tra loss, and still get a runtime of $O(mn + n^2 \log n)$ with general edge
weights. This is known as Johnson’s algorithm, which we discuss
next.

#### 4.2.1 Johnson’s Algorithm and Feasible Potentials

The idea behind this algorithm is to (a) re-weight the edges so that
they are nonnegative yet preserve shortest paths, and then (b) run $n$
instances of Dijkstra’s algorithm to get all the shortest-path distances.
A simple-minded hope (based on our idea for MSTs) would be to add
a positive number to all the weights to make them positive. Although
this preserves MSTs, it doesn’t preserve shortest paths. For instance,
the example on the right has a single negative-weight edge. Adding
1 to all edge weights makes them all have non-negative weights, but the shortest path from \( s \) to \( d \) is changed.

Don Johnson gave an algorithm that does the edge re-weighting in a slightly cleverer way, using the idea of feasible potentials. Loosely, it runs the Bellman-Ford algorithm once, then uses the information gathered to do the re-weighting. At first glance, the concept of a feasible potential does not seem very useful. It is just an assignment of weights \( \phi_v \) to each vertex \( v \) of the graph, with some conditions:

**Definition 4.2.** For a weighted digraph \( G = (V, A) \), a function \( \phi : V \to \mathbb{R} \) is a feasible potential if for all edges \( e = uv \in A \),

\[
\phi(u) + w_{uv} - \phi(v) \geq 0.
\]

Given a feasible potential, we can transform the edge-weights of the graph from \( w_{uv} \) to

\[
\hat{w}_{uv} := w_{uv} + \phi(u) - \phi(v).
\]

Observe the following facts:

1. The new weights \( \hat{w} \) are all positive. This comes from the definition of the feasible potential.

2. Let \( P_{ab} \) be a path from \( a \) to \( b \). Let \( \hat{\ell}(P_{ab}) \) be the length of \( P_{ab} \) when we use the weights \( \hat{w} \), and \( \ell(P_{ab}) \) be its length when we use the weights \( \hat{w} \). Then

\[
\hat{\ell}(P_{ab}) = \ell(P_{ab}) + \phi_a - \phi_b.
\]

The change in the path length is \( \phi_a - \phi_b \), which is independent of the path. So the new weights \( \hat{w} \) preserve the shortest \( a \)-to-\( b \) paths, only changing the length by \( \phi_a - \phi_b \).

This means that if we find a feasible potential, we can compute the new weights \( \hat{w} \), and then run Dijkstra’s algorithm on the remaining graph. But how can we find feasible potentials? Here’s the short answer: Bellman-Ford. Indeed, suppose there some source vertex \( s \in V \) such that every vertex in \( V \) is reachable from \( s \). Then, set \( \phi(v) = \text{dist}(s, v) \).

**Lemma 4.3.** Given a digraph \( G = (V, A) \) with vertex \( s \) such that all vertices are reachable from \( s \), \( \phi(v) = \text{dist}(s, v) \) is a feasible potential for \( G \).

**Proof.** Since every vertex is reachable from \( s \), \( \text{dist}(s, v) \) and therefore \( \phi(v) \) is well-defined. For an edge \( e = uv \in A \), taking the shortest path from \( s \) to \( u \), and adding on the arc \( uv \) gives a path from \( s \) to \( v \), whose length is \( \phi(u) + w_{uv} \). This length is at least \( \phi(v) \), the length of the shortest path from \( s \) to \( v \), and the lemma follows. \( \square \)
In summary, the algorithm is the following:

**Algorithm 4: Johnson’s Algorithm**

**Input:** A weighted digraph $G = (V, A)$

**Output:** A list of the all-pairs shortest paths for $G$

1. $V' \leftarrow V \cup \{s\}$ // add a new source vertex
2. $A' \leftarrow E \cup \{(s, v, 0) \mid v \in V\}$
3. $\text{dist} \leftarrow \text{BellmanFord}((V', A'))$ // set feasible potentials
4. for $e = (u, v) \in A$ do
   5. weight$(e) + = \text{dist}(u) - \text{dist}(v)$
4. for $v \in V$ do
   5. $L = []$ // the result
6. $L += \text{Dijkstra}(G, v)$
4. return $L$

We now bound the running time. Running Bellman-Ford once takes $O(mn)$ time, computing the “reduced” weights $\tilde{w}$ requires $O(m)$ time, and the $n$ Dijkstra calls take $O(n(m + n \log n))$, if we use Fibonacci heaps. Therefore, the overall running time is $O(mn + n^2 \log n)$—almost the same as one SSSP computation, except on very sparse graphs with $m = o(n \log n)$.

4.2.2 More on Feasible Potentials

Fix this! How did we decide to use the shortest-path distances from $s$ as our feasible potentials? Here’s some more observations, which give us a better sense of these potentials, and which lead us to the solution.

1. If all edge-weights are non-negative, then $\phi(v) = 0$ is a feasible potential.
2. Adding a constant to a feasible potential gives another feasible potential.
3. If there is a negative cycle in the graph, there can be no feasible potential. Indeed, the sum of the new weights along the cycle is the same as the sum of the original weights, due to the telescoping sum. But since the new weights are non-negative, so the old weight of the cycle must be, too.
4. If we set $\phi(s) = 0$ for some vertex $s$, then $\phi(v)$ for any other vertex $v$ is an underestimate of the $s$-to-$v$ distance. This is because for all the paths from $s$ to $v$ we have

   \[ 0 \leq \hat{\ell}(P_{sv}) = \ell(P_{sv}) - \phi_v + \phi_s = \ell(P_{sv}) - \phi_v, \]
giving \( \ell(P_v) \geq \phi_v \). Now if we try to set \( \phi(s) \) to zero and try to maximize summation of \( \phi(v) \) for other vertices subject to the feasible potential constraints we will get an LP that is the dual of the shortest path LP.

\[
\begin{align*}
\text{Maximize} & \quad \sum_{x \in V} \phi_x \\
\text{Subject to} & \quad \phi_s = 0 \\
& \quad w_{vu} + \phi_u - \phi_v \geq 0 \quad \forall (v,u) \in E
\end{align*}
\]

4.2.3 The Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is perhaps best introduced via its strikingly simple pseudocode. It first puts down estimates \( \text{dist}(u,v) \) for the distances thus:

\[
\text{dist}_{ij} = \begin{cases} 
  w_{ij}, & i, j \in E \\
  \infty, & i, j \notin E, i \neq j \\
  0, & i = j
\end{cases}
\]

Then it runs the following series of updates.

**Algorithm 5**: The Floyd-Warshall Algorithm

**Input**: A weighted digraph \( D = (V,A) \)

**Output**: A list of the all-pairs shortest paths for \( D \)

1. set \( d(x,y) \leftarrow w_{xy} \) if \( (x,y) \in E \), else \( d(x,y) \leftarrow \infty \)
2. for \( z \in V \) do
   3. for \( x, y \in V \) do
      4. set \( d(x,y) \leftarrow \min\{d(x,y), d(x,z) + d(z,y)\} \)

   Importantly, we run over the “inner” index \( z \) in the outermost loop. The proof of correctness is similar to, yet not that same as that of Algorithm 3, and is again left as a simple exercise in induction.

**Lemma 4.44.** After we have considered vertices \( V_k = \{z_1, \ldots, z_k\} \) in the outer loop, \( \text{dist}(u,v) \) equals the weight of the shortest \( x \)-\( y \) path that uses only the vertices from \( V_k \) as internal vertices. (This is \( \infty \) if there are no such paths.)

The running time of Floyd-Warshall is clearly \( O(n^3) \)—no better than Johnson’s algorithm. But it does have a few advantages: it is simple, and it is quick to implement with minimal errors. (The most common error is nesting the for-loops in reverse.) Another advantage is that Floyd-Warshall is also parallelizable, and very cache efficient. **More details, pointers?**
4.3 Min-Sum Products and APSPs

A conceptually different way to get shortest-path algorithms is via matrix products. These may not seem relevant, *a priori*, but they lead to deep insights about the APSP problem.

Recall the classic definition of matrix multiplication, for two real-valued matrices $A, B \in \mathbb{R}^{n \times n}$

$$(AB)_{ij} = \sum_{k=0}^{n} (A_{ik} \ast B_{kj}).$$

Hence, each entry of the product $AB$ is a sum of products, both being the familiar operations over the field $(\mathbb{R}, +, \ast)$. But now, what if we change the constituent operations, to replace the sum with the $\min$ operation, and the product with a sum? We get the *Min-Sum Product* (MSP): given matrices $A, B \in \mathbb{R}^{n \times n}$, the new product is

$$(A \circledast B)_{ij} = \min_{k} \{A_{ik} + B_{kj}\}.$$ 

This is the usual matrix multiplication, but over the *semiring* $(\mathbb{R}, \min, +)$.

It turns out that computing Min-Sum Products is precisely the operation needed for the APSP problem. Indeed, initialize a matrix $D$ exactly as in the Floyd-Warshall algorithm:

$$D_{ij} = \begin{cases} w_{ij}, & i, j \in E \\ \infty, & i, j \notin E, i \neq j \\ 0, & i = j \end{cases}.$$ 

Now $(D \circledast D)_{ij}$ represents the cheapest $i$-$j$ path using at most 2 hops! (It’s as though we made the outer-most loop of Floyd-Warshall into the inner-most loop.) Similarly, we can compute

$$D^{\circledast k} := D \circledast D \circledast D \cdots \circledast D,$$

whose entries give the shortest $i$-$j$ paths using at most $k$ hops (or at most $k - 1$ intermediate nodes). Since the shortest paths would have at most $n - 1$ hops, we can compute $D^{\circledast n-1}$.

How much time would this take? The very definition of MSP shows how to implement it in $O(n^3)$ time. But performing it $n - 1$ times would be $O(n)$ worse than all other approaches! But here’s a classical trick, which probably goes back to the Babylonians: for any integer $k$,

$$D^{\circledast 2k} = D^{\circledast k} \circledast D^{\circledast k}.$$ 

(Here we use that the underlying operations are associative.) Now it is a simple exercise to compute $D^{\circledast n-1}$ using at most $2 \log_2 n$ MSPs.

A *semiring* has a notion of addition and one of multiplication. However, neither the addition nor the multiplication operations are required to have inverses.
This a runtime of $O(MSP(n) \log n)$, where $MSP(n)$ is the time it takes to compute the min-sum product of two $n \times n$ matrices. Now using the naive implementation of MSP gives a total runtime of $O(n^3 \log n)$, which is almost in the right ballpark! The natural question is: can we implement MSPs faster?

### 4.3.1 Faster Algorithms for Matrix Multiplication

Can we get algorithms for MSP that run in time $O(n^{3-\varepsilon})$ for some constant $\varepsilon > 0$? To answer this question, we can first consider the more common case, that of matrix multiplication over the reals (or over some field)? Here, the answer is yes, and this has been known for now over 50 years. In 1969, Volker Strassen showed that one could multiply $n \times n$ matrices over any field $F$, using $O(n^{\log_2 7}) = O(n^{2.81})$ additions and multiplications. (One can allow divisions as well, but Strassen showed that divisions do not help asymptotically.)

If we define the exponent of matrix multiplication $\omega > 0$ to be smallest real such that two $n \times n$ matrices over any field $F$ can be multiplied in time $O(n^\omega)$, then Strassen’s result can be phrased as saying:

$$\omega \leq \log_2 7.$$

This value, and Strassen’s idea, has been refined over the years, to its current value of 2.3728 due to François Le Gall (2014). (See this survey by Virginia for a discussion of algorithmic progress until 2013.) There has been a flurry of work on lower bounds as well, e.g., by Josh Alman and Virginia Vassilevska Williams showing limitations for all known approaches. see here.

But how about $MSP(n)$? Sadly, progress on this has been less impressive. Despite much effort, we don’t even know if it can be done in $O(n^{3-\varepsilon})$ time. In fact, most of the recent work has been on giving evidence that getting sub-cubic algorithms for MSP and APSP may not be possible. There is an interesting theory of hardness within $P$ developed around this problem, and related ones. For instance, it is now known that several problems are equivalent to APSP, and truly sub-cubic algorithms for one will lead to sub-cubic algorithms for all of them. Examples

Yet there is some interesting progress on the positive side, albeit qualitatively small. As far back as 1976, Fredman had shown an algorithm to compute MSP in $O(n^{3 \log \log n})$ time. He used the fact that the decision-tree complexity of APSP is sub-cubic (a result we will discuss in §4.5) in order to speed up computations over nearly-logarithmic-sized sub-instances; this gives the improvement above. More recently, another CMU alumnus Ryan Williams improved on...
this idea quite substantially to $O\left(\frac{n^3}{2^\sqrt{\log n}}\right)^6$, using very interesting ideas from circuit complexity. We will discuss this result in a later section, if we get a chance.

4.4 Undirected APSP Using Fast Matrix Multiplication

One case we have seen improvements in APSP algorithms is that of graphs with small integer edge-weights. Our focus will be on undirected, unweighted graphs: we present a beautiful algorithm of Raimund Seidel that runs in time $O(n^\omega \log n)$, assuming that $\omega > 2$.

To begin, the adjacency matrix $A$ for graph $G$ is the symmetric matrix

$$A_{ij} = \begin{cases} 1 & ij \in E \\ 0 & ij \notin E \end{cases}.$$

Now consider the graph $G^2$, the square of $G$, which has the same vertex set as $G$ but where an edge in $G^2$ corresponds to being at most two hops away in $G$—that is, $uv \in E(G^2) \iff d_G(u, v) \leq 2$. If we consider $A$ as a matrix over the finite field $(\mathbb{F}_2, +, \times)$, then the adjacency matrix of $G^2$ has a nice formulation:

$$A_{G^2} = A_G \times A_G + A_G.$$

This shows how to get the adjacency matrix of $G^2$ given one for $G$, having spent one Boolean matrix multiplication and one matrix addition. Suppose we recursively compute APSP on $G^2$: how can we translate this result back to $G$? The next lemma shows that the shortest-path distances in $G^2$ are nicely related to those in $G$.

**Lemma 4.5.** If $d_{xy}$ and $D_{xy}$ are the shortest-path distances between $x, y$ in $G$ and $G^2$ respectively, then

$$D_{xy} = \lceil \frac{d_{xy}}{2} \rceil.$$

**Proof.** Any $u$-$v$ path in $G$ can be written as

$$u, a_1, b_1, a_2, b_2, \ldots, a_k, b_k, v$$

if the path has odd length; an even-length path can be written as

$$u, a_1, b_1, a_2, b_2, \ldots, a_k, b_k, a_{k+1}, v.$$

In either case, $G^2$ has edges $ub_1, b_1b_2, \ldots, b_{k-1}b_k, b_kv$, and thus a $u$-$v$ path of length $\lceil \frac{d_{xy}}{2} \rceil$ in $G^2$. Therefore $D_{xy} \leq \lceil \frac{d_{xy}}{2} \rceil$.

To show equality, suppose there is a $u$-$v$ path of length $\ell < \lceil \frac{d_{xy}}{2} \rceil$ in $G^2$. Each of these $\ell$ edges corresponds to either an edge or a 2-edge path in $G$, so we can find a $u$-$v$ path of length at most $2\ell < d_{xy}$ in $G$, a contradiction. \qed
Lemma 4.5 implies that
\[ d_{uv} \in \{2D_{uv}, 2D_{uv} - 1\}. \]

But which one? The following lemmas give us simple rule to decide.
Let \( N_G(v) \) denote the set of neighbors of \( v \) in \( G \).

**Lemma 4.6.** If \( d_{uv} = 2D_{uv} \), then for all \( w \in N_G(v) \) we have \( D_{uw} \geq D_{uv} \).

*Proof.* Assume not, and let \( w \in N_G(v) \) be such that \( D_{uw} < D_{uv} \). Since both of them are integers, we have \( 2D_{uw} < 2D_{uv} - 1 \). Then the shortest \( u-w \) path in \( G \) along with the edge \( vw \) forms a \( u-v \)-path in \( G \) of length at most \( 2D_{uw} + 1 < 2D_{uv} = d_{uv} \), which is in contradiction with the assumption that \( d_{uv} \) is the shortest path in \( G \). \( \square \)

**Lemma 4.7.** If \( d_{uv} = 2D_{uv} - 1 \), then \( D_{uw} \leq D_{uv} \) for all \( w \in N_G(v) \); moreover, there exists \( z \in N_G(v) \) such that \( D_{uz} < D_{uv} \).

*Proof.* For any \( w \in N_G(v) \), considering the shortest \( u-v \) path in \( G \) along with the edge \( vw \) implies that \( d_{uw} \leq d_{uv} + 1 = (2D_{uv} - 1) + 1 \), so Lemma 4.5 gives that \( D_{uw} = \lfloor d_{uw}/2 \rfloor = D_{uv} \). For the second claim, consider a vertex \( z \in N_G(v) \) on a shortest path from \( u \) to \( v \). Then \( d_{uz} = d_{uv} - 1 \), and Lemma 4.5 gives \( D_{uz} < D_{uv} \). \( \square \)

These lemmas can be summarized thus:
\[ d_{uv} = 2D_{uv} \iff \frac{\sum_{w \in N(v)} D_{uw}}{\deg(v)} \geq D_{uv}, \quad (4.1) \]

where \( \deg(j) = |N_G(j)| \) is the degree of \( j \). Given \( D \), the criterion on the right can be checked for each \( uv \) in time \( \deg(v) \) by just computing the average, but that could be too slow—how can we do better?

Define the *normalized adjacency matrix* of \( G \) to be \( \hat{A} \) with
\[ \hat{A}_{uv} = 1_{uv \in E} \cdot \frac{1}{\deg(v)}. \]

Now if \( D \) is the distance matrix of \( G^2 \), then
\[ (D\hat{A})_{uv} = \sum_{w \in V} D_{uw} \hat{A}_{uw} = \frac{\sum_{w \in N_G(v)} D_{uw}}{\deg(v)}, \]
which is conveniently the expression in (4.1). Let \( 1_{(D\hat{A} < D)} \) be a matrix with the \( uv \)-entry being 1 if \( (D\hat{A})_{uv} < D_{uv} \), and zero otherwise. Then the distance matrix for \( G \) is
\[ 2D - 1_{(D\hat{A} < D)}. \]
This completes the algorithm, which we now summarize:

**Algorithm 6: Seidel’s Algorithm**

| Input: | Unweighted undirected graph $G = (V, E)$ with adjacency matrix $A$ |
| Output: | The distance matrix for $G$ |

1. If $A = J$ then
2. return $A$ // If $A$ is all-ones matrix, done!
3. else
4. $A' \leftarrow A \ast A + A$ // Boolean operations
5. $D \leftarrow \text{Seidel}(A')$
6. return $2D - I_{(D \preceq D)}$

Each call to the procedure above performs one Boolean matrix multiplication in step (6.4), one matrix multiplication with rational entries in step (6.6), plus $O(n^2)$ extra work. The diameter of the graph halves in each recursive call (by Lemma 4.5), and the algorithm hits the base case when the diameter is 1. Hence, the overall running time is $O(n^{\omega} \log n)$.

Ideas similar to these can be used to find shortest paths graphs with small integer weights on the edges: if the weights are integers in the interval $[0, W]$, someone and Uri Zwick give an $\tilde{O}(Wn^{\omega})$-time algorithm. In fact, Zwick also extends the ideas to directed graphs, and gives an algorithm with runtime $\tilde{O} \left( W^{\frac{1}{2}} n^{2 + \frac{1}{4}} \right)$.

### 4.4.1 Finding the Shortest Paths

How do we find the shortest paths themselves, and not just their lengths? For the previous algorithms, modifying the algorithms to output the paths is fairly simple. But for Seidel’s algorithm, things get tricky. Indeed, since the runtime of Seidel’s algorithm is strictly sub-cubic, how can we write down the shortest paths in $n^{\omega}$ time, since the total length of all these paths may be $\Omega(n^3)$? We don’t: we just write down the successor pointers. Indeed, for each pair $u, v$, define $S_v(u)$ to be the second node on a shortest $u$-$v$ path (the first node being $u$, and the last being $v$). Then to get the entire $u$-$v$ shortest path, we just follow these pointers:

$$u, S_v(u), S_v(S_v(u)), \ldots, v.$$  

So there is a representation of all shortest paths that uses at most $O(n^2 \log n)$ bits.

The main idea for computing the successor matrix for Seidel’s algorithm is to solve the **Boolean Product Matrix Witness** problem: given $n \times n$ Boolean matrices $A, B$, compute an $n \times n$ matrix $W$ such that $W_{uv} = k$ if $A_{ik} = B_{kj} = 1$, and $W_{ij} = 0$ if no such $k$ exists. We will
4.5 Optional: Fredman’s Decision-Tree Complexity Bound

Given the algorithmic advances, one may wonder about lower bounds for the APSP problem. There is the obvious $\Omega(n^2)$ lower bound from the time required to write down the answer. Maybe even the decision-tree complexity of the problem is $\Omega(n^3)$? Then no algorithm can do any faster, and we’d have shown the Floyd-Warshall and the Matrix-Multiplication methods are optimal. However, thanks to a result of Fredman, we know this is not the case. If we just care about the decision-tree complexity, we can get much better. Specifically, Fredman shows

\textbf{Theorem 4.8.} The Min-Sum Product of two $n \times n$ matrices $A, B$ can be deduced in $O(n^{2.5})$ additions and comparisons.

\textbf{Proof.} The proof idea is to split $A$ and $B$ into rectangular sub-matrices, and compute the MSP on the sub-matrices. Since these sub-matrices are rectangular, we can substantially reduce the number of comparisons needed for each one. Once we have these sub-MSPs, we can simply compute an element-wise minimum for find the final MSP.

Fix a parameter $W$ which we determine later. Then divide $A$ into $n/W \times W$ matrices $A_1, \ldots, A_{n/W}$, and divide $B$ into $n/W \times n$ submatrices $B_1, \ldots, B_{n/W}$. We will compute each $A_i \odot B_i$. Now consider $(A \odot B)_{ij} = \min_{k \in [W]} (A_{ik} + B_{kj}) = \min_{k \in [W]} (A_{ik} + B_{jk}^T)$ and let $k^*$ be the minimizer of this expression. Then we have the following:

\begin{align}
A_{ik^*} - B_{jk^*}^T &\leq A_{ik} - B_{jk}^T \forall k \tag{4.2} \\
A_{ik^*} - A_{ik} &\leq -(B_{jk^*}^T - B_{jk}^T) \forall k \tag{4.3}
\end{align}

Now for every pair of columns, $p, q$ from $A_i, B_i^T$, and sort the following $2n$ numbers

$$A_{1p} - A_{iq}, A_{2p} - A_{2q}, \ldots, A_{np} - A_{nq}, -(B_{1p} - B_{1q}), \ldots, -(B_{np} - B_{nq})$$

We claim that by sorting $W^2$ lists of numbers we can compute $A_i \odot B_i$. To see this, consider a particular entry $(A \odot B)_{ij}$ and find a $k^*$ such that for every $k \in [W], A_{ik^*} - A_{ik}$ precedes every $-(B_{jk^*}^T - B_{jk}^T)$ in their sorted list. By (4.3), such a $k^*$ is a minimizer. Then we can set $(A \odot B)_{ij} = A_{ik^*} + B_{k^*j}$.

This computes the MSP for $A_i, B_i$, but it is possible that another $A_j \odot B_j$ produces the actual minimum. So, we must take the elementwise minimum across all the $(A_i \odot B_i)$. This produces the MSP of $A, B$. Hopefully see (and solve) this problem in a homework.
Now for the number of comparisons. We have \( n/W \) smaller products to compute. Each sub-product has \( W^2 \) arrays to sort, each of which can be sorted in \( 2n \log n \) comparisons. Finding the minimizer requires \( W^2 n \) comparisons. So, computing the sub-products requires \( n/W \cdot 2W^2 n \log n = 2n^2 W \log n \) comparisons. Then, reconstructing the final MSP requires \( n^2 \) element-wise minimums between \( n/W - 1 \) elements, which requires \( n^3/W \) comparisons. Summing these bounds gives us \( n^3/W + 2n^2 W \log n \) comparisons. Optimizing over \( W \) gives us \( O(n^2/\sqrt{n \log n}) \) comparisons. \( \square \)

This result does not give us a fast algorithm, since it just counts the number of comparisons, and not the actual time to figure out which comparisons to make. Regardless, many of the algorithms that achieve \( n^3/\text{poly log } n \) time for APSP use Fredman’s result on tiny instances (say of size \( O(\text{poly log } n) \), so that we can find the best decision-tree using brute-force) to achieve their results.