In this lecture, we will study algorothms for the *single-source shortest paths* (SSSP) and *all-pairs shortest paths* (APSP) problems. For SSSP we explain two algorithms:

1. Dijkstra's Algorithm for the case of non-negative edge weights, and
2. Bellman-Ford Algorithm when the graph has negative weights.

For APSP we explain the following algorithms:

1. Johnson's Algorithm,
2. the Floyd-Warshall Algorithm,
3. Matrix Multiplication using min-sum product, and
4. Seidel's Algorithm

Among the above algorithms for APSP, Seidel's Algorithm is the fastest one, but it has the assumption that the graph is unweighted and undirected.

1 Single-Source Shortest Path Algorithms

Given a directed graph $G = (V, A)$ with edge weights $w_e \in \mathbb{R}$, the *single-source shortest path problem* (SSSP) is to find the shortest path from a single vertex s (the source) to every other vertex in the graph. Since in general the edge weights are allowed to be negative, we should disallow negative-weight cycles (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). Hence, given negative edge-weight, a correct SSSP algorithm must either return a shortest path from s to all other vertices, or a report that there is a negative cycle in the graph.

1.1 Dijkstra's Algorithm

Dijkstra's algorithm is designed to work on graphs with non-negative edge weights. It keeps an estimate of the distance from s to every other vertex. It initialize the estimates of the distances to be 0 for s itself and ∞ for all other vertices. Then repeatedly, it finds the vertex u with the smallest distance; then it commits that distance for u and "relax" the arcs out of it, which means it will update the estimates for all the vertices v that there exists w_{uv} for them as the following.

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

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

To prove the correctness of the algorithm it is enough to show that each time we extract the vertex with the minimum distance from the priority queue, the estimated distance for that vertex is the shortest distance form s. This can be proved by induction; you may find this proof in [CLRS09].

The time complexity of the algorithm depends on the heap structure used for the priority queue; if we use binary heap, we will achieve the running time of $O(m \log n)$ where n is the number of vertices and m is the number of edges. We can do better with a Fibonacci Heap [FT87] (see, e.g., notes here) which supports inserts, find-mins, decrease-keys, and melds in $O(1)$ time, and delete-mins in $O(\log n)$ time. Since Dijkstra's algorithm uses n inserts, n delete-mins, and m decrease-keys, this better heap improves the running time to $O(m + n \log n)$. Later, [AMOT90] introduced two other implementation that achieves $O(m \log \log C)$ and $O(m + n\sqrt{\log C})$ where C is the maximum weight of the graph. Later, [Tho04] 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.

Algorithm 1: 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

```
1.1  add s to heap with key 0;
1.2  for v ∈ V \ {s} do
1.3  |  add v to heap with key ∞
1.4  end
1.5  while heap not empty do
1.6  |  u ← deletemin;
1.7  |  for v a neighbor of u do
1.8  |  |  /* relax the edge (u,v) */
1.9  |  |  key(v) ← min{ key(v), key(u) + w_{u,v} }
1.10 |  end
1.11 end
```

As we mentioned, Dijkstra's does not work on the samples with negative edge. In Figure 4.1 you may find a simple example in which Dijkstra will fail. In this example, Dijstra's algorithm will find the distance 2 for the vertex a; however, the shortest distance to a is via b, and is $3 - 2 = 1$.

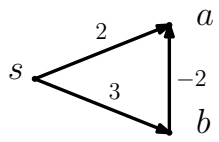

Figure 4.1: Example in which Dijkstra's algoirthm does not work on

1.2 The Bellman-Ford Algorithm

For the graphs with negative edge-weights, we cannot use Dijkstra's algorithm. The Bellman-Ford algorithm is one of the well-known algorithms for this case. [1] 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

[1] Jeff Erickson points out that 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.

following. (A visualization can be found [Vis].)

Algorithm 2: The Bellman-Ford-Moore Algorithm

Input: A digraph $G = (V, E)$ with edge weights $w_e \in \mathbb{R}$, and source vertex $s \in V$

Output: The shortest-path distances from each vertex to s, or report that a negative-weight cycle exists

2.1 $Dist(s) = 0$ `/* the source has distance 0 */`
2.2 **for** $v \in V$ **do**
2.3 | $Dist(v) \leftarrow \infty$;
2.4 **end**
2.5 **for** $|V|$ *iterations* **do**
2.6 | **for** *edge* $e = (u, v) \in E$ **do**
2.7 | | $Dist(v) \leftarrow \min\{ Dist(v), Dist(u) + weight(e)\}$
2.8 | **end**
2.9 **end**
2.10 If any distances changed in the last (n^{th}) iteration, output "G has a negative weight cycle".

The proof relies on the following fact that is easily proved by induction on i.

Lemma 4.1. *After i iterations of the algorithm, $Dist(v)$ equals the length of the shortest-path from s to v containing at most i edges. (This is defined to be ∞ if there are no such paths.)*

Suppose there is no negative-weight cycle, then the shortest-paths are well-defined and simple, and hence contain at most $n-1$ edges, because any path with more than $n-1$ edges has a cycle in it. By Lemma 4.1, the algorithm is guaranteed to be correct after $n-1$ iterations (and hence 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 $Dist(u)$ for vertices on this cycle will continue to decrease in each subsequent iteration, because we may reach to any point on this cycle and by moving in that cycle we can accumulate negative distance; therefore, the distance will get smaller and smaller in each iteration. Specifically, they will decrease 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 outputs a correct solution despite C's existence, and it will produce the distance of ∞ for all the vertices in that cycle)

For the runtime, each iteration of Bellman-Ford looks at each edge once, and there are n iterations, so the runtime is $O(mn)$. 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 $[-M, \infty)$, we can compute SSSP in time $O(m\sqrt{n}\log M)$, using an idea we discuss in Homework #1.

2 The All-Pairs Shortest Paths Problem (APSP)

As the name implies, APSP is the problem of finding the shortest-paths between all pairs of vertices. The obvious way to do this is to run SSSP n times, each time with a different vertex in the graph being the source vertex. Hence the runtime for such an algorithm is n times the runtime of the underlying $SSSP$ algorithm. If we have non-negative edge weights we can use Dijkstra and get $O(mn + n^2 \log n)$ time. If we have negative edge weights, we would naively use Bellman-Ford and

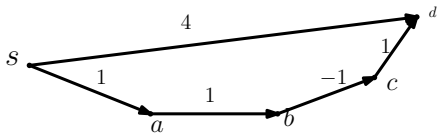

Figure 4.2: A graph with negative edges in which adding positive constant to all the edges will change the shortest paths

get a runtime of $O(mn^2)$. Fortunately, there is a clever trick to bypass this extra loss, and still get a runtime of $O(mn + n^2 \log n)$ — this is known as Johnson's algorithm, which we discuss next.

2.1 Johnson's Algorithm and Feasible Potentials

One way to avoid the problem of negative weights would be to do something like the following

1. Re-weight the edges so that they are nonnegative, yet preserve shortest paths.
2. Run n instances of Dijkstra's SSSP algorithm.

A simple hope (based on what we did 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, Figure 4.2 shows a graph with one negative edge-weight. If we add 1 to all of the weights, we won't have any negative-weight edges, but the shortest path from s to d will be changed. However, there is a better way of rewriting weights using *feasible potential*.

Let us introduce and formalize the concept of a *feasible potential*, which gives each vertex a carefully chosen weight ϕ_v. Formally,

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

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

Given a feasible potential, we can set each weight w_{uv} to $\hat{w}_{uv} := w_{uv} + \phi(u) - \phi(v)$. Now, let us make some remarks about the new weights and the feasible potentials:

1. The new weights are all positive. This comes from the definition of the feasible potential.
2. If for all the edges $w_{uv} > 0$, then $\phi(v) = 0$ is a feasible potential.
3. Adding a constant to a feasible potential makes another feasible potential.
4. If there is a negative cycle in the graph, there will be no feasible potential. (Because in a cycle sum of the new weights will be the same as the sum of the original weights due to the telescoping sum)
5. Let P_{ab} be a path from a to b and let $l(P_{ab})$ be the length of P_{ab} when we use the weights W and $\hat{l}(P_{ab})$ be the length of P_{ab} when we use the weights $\hat{W}$. Then

$$\hat{l}(P_{ab}) = l(P_{ab}) + \phi_a - \phi_b$$

   Therefore, we can conclude the new weights will preserve the shortest paths because the length of all the paths from s to a vertex v will be increased by a constant. $(\phi_s - \phi_v)$

6. If we set $\phi(s) = 0$ for some vertex s, then $\phi(v)$ for all other vertices will be an underestimate of the distance from s that is because for all the paths from s to v we have $\hat{l}(P_{sv}) = l(P_{sv}) - \phi_v \geq 0$, therefore $l(P_{sv}) \geq \phi_b$. 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.

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

Based on the mentioned fact all we need is finding a feasible potential and then we can rewrite the weights and use Dijkstra's algorithm.
Let us say for the moment that there is a vertex s (a source you might say) such that every vertex is reachable from s. Then, we may set $\phi(v) = \text{dist}(s, v)$ and in the following lemma we will show that it is a valid feasible potential.

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

Proof. Since every vertex is reachable from s, $\phi(v)$ is well-defined. Therefore, consider $e = (u, v) \in A$. we must show that $\phi(u) + w_{uv} \geq \phi(v)$. Note that the shortest path from s to u and the edge (u, v) from a path from s to v. The length of this path is at least the length of the shortest path from s to v, and the lemma follows. □

So, if we have a source vertex, we can just run the Bellman-Ford algorithm (2) from the source and take the length of the paths it returns as our feasible potential. But wait, what happens if D has no source vertex? Then we just add an extra vertex with 0-weight edges to every vertex. Combining these ideas gives us Johnson's Algorithm. In fact, it is better to add a new vertex, even if there is already a source vertex because it may take more time to find that source vertex than just adding a new vertex.

Algorithm 3: Johnson's Algorithm

Input: A weighted digraph $D = (V, A)$
Output: A list of the all-pairs shortest paths for D

3.1 $V' \leftarrow V \cup \{s\}$ `/* add a new source vertex */`
3.2 $A' \leftarrow E \cup \{(s, v, 0) \mid v \in V\}$;
3.3 $Dist \leftarrow \text{BellmanFord}((V', A'))$;
3.4 `/* set feasible potentials */`
3.5 **for** $e = (u, v) \in A$ **do**
3.6 $\quad weight(e) += Dist(u) - Dist(v)$;
3.7 **end**
3.8 $L = []$ `/* the result */`
3.9 **for** $v \in V$ **do**
3.10 $\quad L += \text{Dijkstra}(D, v)$;
3.11 **end**
3.12 **return** L;

By (4.3) and the correctness of Bellman-Ford (2) and Dijkstra (1) this algorithm is correct. For the running time, Bellman-Ford requires $O(mn)$ time, setting the potentials requires $O(m)$ time, and the n Dijkstra calls require $O(n(m + n \log n))$ using Fibonacci heaps. Therefore, the overall running time is $O(mn + n^2 \log n)$.

2.2 Floyd-Warshall's Algorithm

The Floyd-Warshall algorithm is perhaps best understood through its strikingly simple pseudocode. It first initializes the distance matrix using the edge weights (and 0 for the distance from one edge to itself and ∞ if there is no edge between two vertices).

Algorithm 4: Floyd Warshall's Algorithm

Input: A weighted digraph $D = (V, A)$
Output: A list of the all-pairs shortest paths for D

```
4.1 set d(x, y) ← w_xy if (x, y) ∈ E, else d(x, y) ← ∞
4.2 for z ∈ V do
4.3 |   for x, y ∈ V do
4.4 |   |   d(x, y) ← min{d(x, y), d(x, z) + d(z, y)}
4.5 |   end
4.6 end
```

We can actually use almost the same proof for correctness as we did with Bellman-Ford (2). The difference is in the induction: here the induction hypothesis is that after we've considered vertices $V_k = \{z_1, \ldots, z_k\}$ in the outer loop, $d(x, y)$ contains the shortest x-y path using only the vertices from V_k as internal vertices. This has a runtime of $O(n^3)$ — 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 parellelizable, and very cache efficient.

3 Min-Sum Products and APSPs

Let us look at a seemingly unrelated topic, matrix products, which actually leads to a very deep insight about the APSP problem. There is of course the classic definition of matrix multiplication for two real-valued matrices $A, B \in \Re^{n \times n}$

$$(A * B)_{ij} = \sum_{k=0}^{n} (A_{ik} * B_{kj})$$

This is a sum of products, fundamentally an operation in the field $(\Re, +, *)$. Now let us define a similar operation, the *Min-Sum Product*(MSP), for the same matrices $A, B \in \Re^{n \times n}$

$$(A \circledcirc B)_{ij} = \min_k \{A_{ik} + B_{kj}\}$$

This is a minimum over sums, which lives in the semiring $(\Re, \min, +)$. So, how does this relate the to APSP problem? Consider what the Min-Sum Product does for a special kind of matrix A. We define the adjacency matrix A for a given graph to be:

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

Then $A \circledcirc A$ represents the best 1 or 2 hop path between *every* pair of vertices x, y! In other words, the Min-Sum product does the same thing as the inner loop in the Floyd-Warshall algorithm (4)! This gives us another algorithm to compute APSP

Algorithm 5: naive Min-Sum Product Algorithm

Input: A weighted digraph $D = (V, E)$ as an adjacency matrix A
Output: The distance matrix for D

5.1 **return** $A^{\circledcirc n}$`/* the` n`-fold Min-Sum Product of` A `with itself */`

Inductively, we can then see that $((A \circledcirc A) \circledcirc A)$ will be a matrix of shortest paths of 3 hops or fewer. We can continue up to $A \circledcirc A \circledcirc A \cdots \circledcirc A$ with n multiplications, which gives the shortest n-hop path between any two vertices. No shortest path can be more than n hops, so we can use n min-sum products on the adjacency matrix to get the APSP.

How long will this take? At first it may look like like $A^{\circledcirc n}$ will take n min-sum product calculations, but that is not true. By noting $A^{\circledcirc n} = A^{\circledcirc n/2} \circledcirc A^{\circledcirc n/2}$, we can get $A^{\circledcirc n}$ recursively in $\log n$ min-sum product steps. So our runtime is $O(\log n \times MSP(n))$, where $MSP(n)$ is the time it takes to take the min-sum product of two $n \times n$ matrices.

How long is $MSP(n)$? Naively, it takes n^3 time, just like matrix multiplication. But matrix multiplication can be sped up. Strassen's algorithm can do it in $O(n^{\log_2(7)})$ [Str69], and the Coppersmith-Winograd algoirthm does it in $O(n^{2.376\ldots})$ time. We don't know the best-possible exponent for the optimal matrix multiplication solution, but we write it as $O(n^\omega)$. The current best bound is $\omega \leq 2.373\ldots$ due to CMU alumna Virginia Vassilevska Williams. (See this survey by her). Can we do min-sum product in $O(n^\omega)$ time? Unfortunately scholars do not know. In fact, we don't even know if it can be done in $O(n^{3-\epsilon})$ time.

We have however made some improvement over the $O(n^3)$ time. Fredman showed an algorithm for doing the min-sum product in $O(n^3 \cdot \frac{\log\log n}{\log n})$ [Fre76]. Then in 2014, CMU alumnus Ryan Williams improved this to $O\left(\frac{n^3}{2^{\sqrt{\log n}}}\right)$ [Wil14].

4 Faster APSP Using Fast Matrix Multiplication

Unlike MSP, there are some great improvements in the complexity of matrix multiplication. In particular, there is the algorithm of Coppersmith and Winograd which improves ω to 2.376 [CW90]. We could hope that we could use similar techniques and create a MSP with a similar running time, but the Coppersmith-Winograd Algorithm depends crucially on the underlying algebraic structure being a commutative ring. Since $(\Re, \min, +)$ is only a semiring, we cannot hope to gain such an improvement naively. One can show solving MSP is equivalent of solving APSP more formally it is possible to show $APSP(n) = \theta(MSP(n))$. However, if our MSP algorithm relies on matrix multiplication, we can easily use Coppersmith's algorithm.

If we consider general real-valued weights, there are no known improvements for the APSP problem. if the graph is undirected and unweighted, we have the beautiful algorithm of Seidel [Sei92] which we present below. If the weights are restricted to lie in the interval $[0, W]$ for some $W \in \mathbb{R}$, we have the following results. For undirected graphs, [SZ99] provides an $\tilde{O}(Wn^\omega)$ algorithm, and [Zwi00] provides an $\tilde{O}\left(W^{\frac{1}{4-\omega}} n^{2+\frac{1}{4-\omega}}\right)$ time algorithm for directed graphs.

4.1 Seidel's Algorithm

Seidel's Algorithm can be used for undirected and unweighted graph to compute APSP. For this section, let a graph G be undirected and unweighted. Under this assumption, the adjacency matrix A of G is simply

$$A_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & (i,j) \notin E \end{cases}$$

We would like to look at the adjacency matrix of G^2 the graph on the same vertex set at G but where $(u,v) \in E(G^2) \iff d_G(u,v) \leq 2$. If we take A to be a matrix over $(\mathbb{F}_2, +, *)$, then the adjacency matrix of G^2 has a nice formulation as follow.

$$A_{G^2} = A_G * A_G + A_G$$

Note that the addition and multiplication are Boolean in the above equation. Moreover, we have the following lemma regarding G^2

Lemma 4.4. *Let d_{xy} be the length of the shortest path between $x, y \in G$, and likewise define D_{xy} in G^2. Then,*

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

Proof. Consider a u, v path in G, which we write as $u, a_1, b_1, a_2, b_2, \ldots, a_k, b_k, v$ if the path has even length and $u, a_1, b_1, a_2, b_2, \ldots, a_k, b_k, a_{k+1}, v$ if the path has odd length. Now consider the same set of vertices in G^2. By definition, $(u, b_1), (b_1, b_2), \ldots, (b_{k-1}, b_k), (b_k, v) \in E$ in either case.Therefore, there is a u, v-path of length $\lceil \frac{d_{xy}}{2} \rceil$ in G^2. Thus, $D_{xy} \leq \lceil \frac{d_{xy}}{2} \rceil$.

To show the other inequality, assume that there is a u, v-path of length $l < \lceil \frac{d_{xy}}{2} \rceil$ in G^2. Each edge in l is either a 1-edge or a 2-edge path in G. Thus, we can find a u, v-path of length at most $2l$ in G, a contradiction. The lemma follows. □

A simple consequence of this lemma is that if we know D_{uv}, then $d_{uv} \in \{2D_{uv}, 2D_{uv} - 1\}$. However, we have the following two lemmas to resolve this dilemma.

Lemma 4.5. *If $d_{uv} = 2D_{uv}$, then for all $w \in N_G(v)$ we have $D_{uw} \geq D_{uv}$. Note that we take the neighbors in G not in G^2.*

Proof. Assume not, and let z be such a vertex that $D_{uz} < D_{uv}$. Since both of them are integers we have $2D_{uz} < 2D_{uv} - 1$. Then the shortest uz path in G and the edge zv form a u, v-path in G of length at most $2D_{uz} + 1 < 2D_{uv} = d_{uv}$, which is in contradiction with the assumption that d_{uv} is the shortest path. □

Lemma 4.6. *If $d_{uv} = 2D_{uv} - 1$, then for all $w \in N_G(v)$ we have $D_{uw} \leq D_{uv}$ and there exists $z \in N_G(v)$ such that $D_{uz} < D_{uv}$. Note that we take the neighbors in G not in G^2.*

Proof. For the first claim, assume there exists a vertex y such that $D_{uy} > D_{uv}$. Then a shortest path from u to v in G and the wedge vy form a path of length $2D_{uv} < 2D_{uy} - 1 \leq d_{uy}$. Which is in contradiction with the assumption that d_{uy} is the length of the shortest path from u to y.
For the second claim, consider the vertex z that is the neighbor of v and it is on a shortest path from u to v. Then since $d_{uz} + 1 = d_{uv}$, we can conclude $D_{uz} < D_{uv}$ □

Corollary 4.7. $d_{uv} = 2D_{uv} \iff \sum_{z \in N(v)} D_{uz} \geq \deg(v) D_{uv}$

Let us say that D is the distance matrix of G^2 and A is the adjacency matrix of G. Then $(DA)_{uv} = \sum_{z \in N(v)} D_{uz}$. Then if we let $\mathbf{1}((DA)_{uv} < \deg(v)D_{uv})$ be an indicator matrix where each entry is 1 if $(DA)_{uv} < \deg(v)D_{uv}$, the distance matrix of G is $2D - \mathbf{1}((DA)_{uv} < \deg(v)D_{uv})$! This key observation, along with recursion, gives us Seidel's Algorithm

Algorithm 6: Seidel's Algorithm

Input: An unweighted undirected graph $G = (V, E)$ as an adjacency matrix A
Output: The distance matrix for G

```
if A = J then
    /* If A is the all-ones matrix, we are done */
    return A;
else
    A' ← A * A + A/* note these are boolean operations */
    D ← Seidel(A');
    D' ← 2D − 1((DA)_uv < deg(v)D_uv);
    return D';
end
```

To show the running time, (6.5) can be performed in $O(n^\omega)$ time with matrix multiplication, as can (6.7). By (4.4), the diameter of the graph is halved at each recursive call, and the algorithm hits the base case when the diameter is 1. Hence, the overall running time is $O(n^\omega \log n)$

5 Fredman's Bound on the Decision-Tree Complexity (Optional)

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 [Fre75], we know this is not the case. If we just care about the decision-tree complexity, we can get much better. Specifically, Fredman shows

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.*

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

$$A_{ik^*} - B^T_{jk^*} \leq A_{ik} - B^T_{jk} \ \forall k \tag{4.1}$$

$$A_{ik^*} - A_{ik} \leq -(B^T_{jk^*} - B^T_{jk}) \ \forall k \tag{4.2}$$

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 $A_i \circledcirc B_i$. To see this, consider a particular entry $(A \circledcirc B)_{ij}$ and find a k^* such that for every $k \in [W]$, $A_{ik^*} - A_{ik}$ precedes every $-(B^T_{jk^*} - B^T_{jk})$ in their sorted list. By (4.2), such a k^* is a minimizer. Then we can set $(A \circledcirc B)_{ij} = A_{ik^*} + B_{k^*j}$.
This computes the MSP for A_i, B_i, but it is possible that another $A_j \circledcirc B_j$ produces the actual minimum. So, we must take the element-wise minimum across all the $(A_i \circledcirc B_i)$. This produces the MSP of A, B.

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 $n/W * 2W^2 n \log n = 2n^2 W \log n$ 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. □

Observe that the above idea 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 small 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.

Acknowledgments

These lecture notes were scribed by Alireza Samadianzakaria, based on previous scribe notes of Nicholas Sieger and Adam Kavka.

References

[AMOT90] Ravindra K. Ahuja, Kurt Mehlhorn, James Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *J. ACM*, 37(2):213–223, April 1990. 1.1

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. 1.1

[CW90] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, March 1990. 4

[Fre75] M. L. Fredman. On the decision tree complexity of the shortest path problems. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 98–99, Oct 1975. 5

[Fre76] Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976. 5.1

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987. 1.1

[Sei92] Raimund Seidel. On the all-pairs-shortest-path problem. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 745–749, New York, NY, USA, 1992. ACM. 4

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13(4):354–356, August 1969. 5.1

[SZ99] Avi Shoshan and Uri Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 605–, Washington, DC, USA, 1999. IEEE Computer Society. 4

[Tho04] Mikkel Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *J. Comput. Syst. Sci.*, 69(3):330–353, November 2004. 1.1

[Vis] A visualization for shortest path algorithms. https://visualgo.net/sssp. Accessed: 2018-09-11. 1.2

[Wil14] Ryan Williams. Faster all-pairs shortest paths via circuit complexity. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 664–673, New York, NY, USA, 2014. ACM. 5.1

[Zwi00] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *CoRR*, cs.DS/0008011, 2000. 4