Lecturer: Anupam Gupta Scribe: Nicholas Sieger, Adam Kavka

In this lecture, we did a review of single source shortest paths and all pair shortest paths. This included reviews of Dijkstra's algorithm, the Bellman-Ford algorithm, Johnson's algorithm, and the Floyd-Warshall algorithm. We also discussed the possibility of using min-sum products to find all pairs shortest path solutions. Lastly, we did an in depth discussion of Seidel's algorithm, which finds the all pairs shortest path solution for an undirected graph with all edge weights equal to 1.

1 Single Source Shortest Path Algorithms

For a weighted directed graph, the shortest path problem finds the path with the lowest total weight between two vertices. The single source shortest path problem (SSSP) finds the shortest path from a single vertex (the source) to every other vertex in the graph. However, what happens if there is a cycle with negative total weight? Such a cycle allows for ever-smaller shortest paths as we can just run around the cycle to reduce the total weight of a path. Hence, a correct SSSP algorithm must either return the short paths between all vertices or a report that there is a negative cycle in the graph. As the following classic algorithms show, this extra requirement tends to increase the running time.

1.1 Dijkstra's Algorithm

Dijkstra's Algorithm avoids the issue of negative cycles by assuming that all the edge weights are positive. This allows the algorithm to use a heap where the key of a vertex is the length of a shortest path to v. At a high level, we start with an initial guess of ∞ for the distance to each node. Then in each step, we look at the vertex in our priority queue with the shortest distance from the source. We see if we can go from that node to each of its neighbors and get a shorter path than that neighbor's current estimate. If so, we "relax" that estimate; that's to say, we update it with the new lower value. We also add the vertex to our priority. We repeat until the priority queue is empty. Here it is in pseudocode:

Algorithm 1: Dijkstra's Algorithm

Input: A directed graph G with nonnegative edge weights and a source vertex $s \in G$

Output: The shortest paths from each vertex to s

```
1.1  Heap ← new heap;
1.2  DistMap ← new mapping from vertices to distances;
1.3  for v ∈ V do
1.4      /* set all initial distances to ∞ */
1.5      DistMap(v) ← ∞;
1.6      add v to Heap
1.7  end
1.8  while Heap not empty do
1.9      u ← deletemin(Heap);
1.10     for v a neighbor of u do
1.11         /* the relax step */
1.12         alt ← DistMap(v) + weight((u, v));
1.13         if alt < DistMap(v) then
1.14             DistMap(v) = alt
1.15         end
1.16     end
1.17 end
1.18 return DistMap
```

Dijkstra's algorithm has a few advantages. Being a greedy algorithm, it's fairly intuitive. It's also fast. When implemented with a simple binary heap, (1) takes $O(m \log(n))$ time. However, we can do better with a Fibonacci Heap [FT87]. Fibonacci heaps are an incredible data structure which supports insert, find-min, decrease-key, and meld in $O(1)$ time, and delete-min in $O(\log(n))$ time. Since (1) uses n inserts, n deletes, and m decrease-keys, this better heap improves the running time to $O(m + n \log(n))$. Modern scholars have actually found a slightly better implementation of Dijkstra's algorithm that runs in $O(m + n \log \log(n))$ time. If you want to learn more about Dijkstra's Algorithm, here are a few good sources to check.

- Wikipedia this has a very useful animation of the execution of the algorithm on a graph
- Dijkstra with Fibonacci Heap Danny Sleator's notes on Fibonacci heaps with specific application to Dijkstra's algorithm

1.2 Bellman-Ford Algorithm

However, what if we want to have negative edge weights? Then we can look at the Bellman-Ford-Moore Algorithm

The algorithm starts with an overestimate of the shortest path to each vertex, just like Dijkstra's algorithm. At each step of the algorithm, each vertex looks at whether its current shortest path length plus its edge weight results in a lower estimate for its neighbor (it does this once for each adjacent edge). If so, we relax that edge's estimate. We repeat n times. In pseudocode

Algorithm 2: Bellman-Ford Algorithm

Input: A weighted digraph D and a source vertex $s \in D$

Output: A list of the shortest path from each vertex to s or a report that a negative weight cycle exists

```
2.1  /* Step 1:  initialize the graph */
2.2  Dist ← a new mapping from vertices to distances;
2.3  for v ∈ V do
2.4  |  Dist(v) ← ∞;
2.5  end
2.6  Dist(s) = 0 /* the source has distance 0 */
2.7  /* Step 2:  relax repeatedly */
2.8  for V iterations do
2.9  |  for edge e = (u, v)inE do
2.10 |  |  if Dist(v) > Dist(u) + weight(e) then
2.11 |  |  |  Dist(V) = Dist(u) + weight(e);
2.12 |  |  end
2.13 |  end
2.14 end
2.15 /* Step 3:  find any negative weight cycles */
2.16 for e = (u, v) ∈ E do
2.17 |  if Dist(V) > Dist(u) + weight(e) then
2.18 |  |  return "negative weight cycle";
2.19 |  end
2.20 end
2.21 return Dist
```

Wikipedia gives a nice lemma that makes the correctness clear.

Lemma 4.1. *After i iterations of (2.8), the following holds for every $v \in V$*

- *$Dist(v) = \infty$ if v is not reachable in i steps from s*
- *$Dist(v)$ is at most the length of the shortest path from s to v with at most i edges*

Proof. We show the lemma by induction on i. The base case, $i = 0$, is trivial as all vertices other than the source are not reachable in 0 steps. For the induction, if v is not reachable in i steps, none of its neighbors are reachable in $i - 1$ steps. Therefore, $Dist(v) = \infty$ by (2.10) and induction. If v is reachable in i steps, then some of its neighbors are reachable in $i - 1$ steps. Then by (2.10) and induction, the algorithm chooses the path with the least distance. □

Since no path can be longer that n vertices, the algorithm is guaranteed to be correct after n iterations. The caveat for this is that Bellman-Ford does not give the correct answer if the graph contains a negative cycle. However, Bellman Ford can detect these negative cycles. After n iterations of the algorithm, if we do an additional iteration and one of the estimates changes, then there must be a negative cycle.

Each step of Bellman-Ford looks at each edge once, and there are n steps, so the runtime is $O(m*n)$, which is worse than Dijkstra's algorithm (1). To learn more about Bellman-Ford, see the following links

- Wikipedia
- A visualization of SSSP

2 All Pairs Shortest Paths (APSP)

Like the name implies, APSP is the problem of finding the shortest path between any combination of two pairs you choose. The most obvious way to do this is to run SSSP on every vertex in the graph. The runtime will be n multiplied by the runtime of your $SSSP$ algorithm. Thus for Bellman-Ford it is $O(mn^2)$, and for Dijkstra it is the slightly better $O(mn + n^2 \log(n))$. However, we must remember the limitations of Dijkstra's algorithm: no negative edge weights allowed. Fortunately, Johnson's algorithm provides a clever trick to bypass this issue and still get $O(mn + n^2 \log(n))$.

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 edge weights so that they are nonnegative yet preserve shortest paths
2. Run n instances of Dijkstra's SSSP algorithm

Of course, there is the obvious question of how to perform the re-weighting. This introduced the concept of a *feasible potential*, which give each vertex a carefully chosen weight. More formally,

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

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

Note that no such function can exist if D contains a negative cycle.

Given a feasible potential, we can set each weight w_{uv} to $\hat{w}_{uv} := w_{uv} + \phi(u) - \phi(v)$. Then, if we have a path p from u to v of total weight w_p, then the weight of p under the re-weighting is $\phi(u) + w_p - \phi(v)$. Since this holds for any path from u to v, the shortest path cannot change! So all we have to do is find a feasible potential.
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 will set $\phi(v) = \text{dist}(s, v)$.

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.

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 $A' \leftarrow E \cup \{(s, v, 0) | v \in V/\}$;
3.3 $Dist \leftarrow$ 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 $\quad L +=$ 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

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 `/* NOTE THE LOOP ORDER */`
4.2 **for** $z \in V$ **do**
4.3 $\quad$ **for** $x, y \in V$ **do**
4.4 $\quad\quad d(x, y) \leftarrow \min\{d(x, y), d(x, z) + d(z, y)\}$
4.5 $\quad$ **end**
4.6 **end**

We can actually use the same proof for correctness, almost line-for-line, as we did with Bellman-Ford (2). The basic idea is that we check to see if a different vertex z produces a shorter path from x to y. This has a runtime of $O(n^3)$...definitely worse 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. Lastly, it is 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 we don't know the exact exponent for the optimal matrix multiplication solution, but we write it as $O(n^\omega)$. 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 $O\left(\frac{n^3}{\log(n)}\right)$ [Fre76]. Then in 2014 a CMU alumni, 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. 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

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

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

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, vpath 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 $\forall w \in N_G(v)$ $D_{uw} \geq D_{uv}$ Note that we take the neighbors in G not in G^2.*

Proof. Assume not, and let z be such a vertex. Then the shortest uz path and the edge zv form a u, v-path of length at most $2D_{uz} + 1 < 2D_{uv} - 1 \leq d_{uv}$, a contradiction. □

Lemma 4.6. *If $d_{uv} = 2D_{uv} - 1$, then $\forall w \in N_G(v)$ $D_{uw} \leq D_{uv}$ and $\exists z \in N_G(v)$ $D_{uz} < D_{uv}$ 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 $2D_{uv} < 2D_{uy} - 1 \leq d_{uy}$.
For the second claim, consider the vertex z on a shortest path from u to v. Clearly, $D_{uz} < D_{uv}$ □

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. Then $(D * A)_{uv} = \sum_{z \in N(v)} D_{uz}$. 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 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

```
6.1 if A = J then
6.2 |   /* If A is the all-ones matrix, we are done */
6.3 |   return A;
6.4 else
6.5 |   A' ← A * A + A/* note these are boolean operations */
6.6 |   D ← Seidel(A');
6.7 |   D' ← 2D − 1((DA)_uv < deg(v)D_uv);
6.8 |   return D';
6.9 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 Lower Bounds

5.1 Fredman's Trick

After all the progress on algorithmic advances, one may wonder about lower bounds for the problem. There is the obvious $\Omega(n^2)$ lower bound from the time required to write down the answer. However, thanks to a result of Fredman [Fre75], we know that progress can still be made. More 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^T_i, 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. □

References

[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. 5.1

[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. 1.18

[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

[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