6.890 Lecture 7 Paths, Successors and Witnesses
Scribe: Jaehyun Park, Yinzhan Xu **Date:** September 30, 2021

So far, we studied how to compute the shortest path distances under various matrix products, but never showed how one can compute the paths corresponding to these shortest distances. In this lecture, we focus on how to find the actual paths, by modifying the algorithms we previously studied.

We use $\odot : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ in this lecture to represent a binary operation between two integers. For a graph $G = (V, E)$ with weights $w : E \to \mathbb{Z}$, we define the weight of a path by

$$w_{\odot}(i_1 \to i_2 \to \cdots \to i_t) = ((w(i_1, i_2) \odot w(i_2, i_3)) \odot \cdots) \odot w(i_{t-1}, i_t).$$

Using this notation of weight, we can define the shortest $\odot$ path between two vertices as

$$d_{\odot}(i, j) = \min_{\text{paths } P \text{ from } i \text{ to } j} w_{\odot}(P).$$

(For some problems, the best path could be a maximum, not a minimum, such as for bottleneck paths, but in those cases one can often define an operation $\odot'$ where $a \odot' b = -((-a) \odot (-b))$, where now the max paths are now min paths under $\odot'$.)

In this lecture, we assume that the weight function $\odot$ is such that the best path is simple, so the distance is well defined.

We can also use $\odot$ to define the $(\min, \odot)$-product of $n \times n$ integer matrices A, B. It is defined as $(A \odot B)[i, j] = \min_k\{A[i, k] \odot B[k, j]\}$. Note that many problems we considered in previous lectures, such as $(\min, +)$-product, $(\max, \min)$-product, and $(\min, \leq)$ product fall under this framework. For BMM, it can actually be viewed as a $(\max, \times)$-product restricted to $\{0, 1\}$ entries, where $\times$ is the ordinary multiplication; by negating the entries of one of the matrices, BMM can also be represented as a $(\min, \odot)$ product.

Let us assume that our operation $\odot$ has the following monotonicity property: For all $a, b, c \in \mathbb{Z}$ where $a \leq b$, we must have $a \odot c \leq b \odot c$.

If this property holds and the best paths are wlog simple, then the shortest $\odot$ paths for all pairs of vertices can be computed by defining an adjacency matrix and then successively right-multiplying $n - 1$ times.

In this lecture, we will use the notation $i \rightsquigarrow j$ to denote a path from i to j, and whether it represents the shortest path or not should be clear from the context.

To motivate the discussion, suppose that we want to compute shortest paths between *all* pairs of nodes. However, in general, the size of the output can be $\Omega(n^3)$, and as a result, many related problems become uninteresting. Thus, we will look for the *predecessor matrix* or *successor matrix* instead, which essentially have the same representational power as the list of all shortest paths, but use only $\tilde{O}(n^2)$ memory.

Definition 0.1. *An $n \times n$ matrix P is a predecessor matrix for all pairs $(\min, \odot)$ paths if for every pair of i, j, $d_{\odot}(i, j) = d_{\odot}(i, P(i, j)) \odot w(P(i, j)), j)$.*

This definition immediately gives a procedure to retrieve a shortest path between any given pair of nodes using the predecessor matrix, assuming that the path is simple[1]. Start from a pair (i, j), we can find the previous node in the shortest path by replacing j with $P(i, j)$. We can repeat this process until we reach vertex i. This shows that finding a predecessor matrix is basically as useful as having the entire list of shortest paths.

Different from predecessor matrix, the successor matrix is only defined if $\odot$ is associative.

Definition 0.2. *An $n \times n$ matrix S is a successor matrix for all pairs $(\min, \odot)$ paths if for every pair of i, j, $d_{\odot}(i, j) = w(i, S(i, j)) \odot d_{\odot}(S(i, j), j)$.*

When $\odot$ is associative, we can use the successor matrix to recover the whole path as well.

[1]The shortest paths problems we consider all have the property that there is always a simple shortest path.

Proposition 0.1. *Given a predecessor/successor matrix S, for any i, j, a shortest $i \rightsquigarrow j$ path can be found in time linear in its length.*

Another concept that is closely related to predecessor matrix is the witness matrix.

Definition 0.3. *A witness for $(A \odot B)[i, j]$ is the index k such that $A(i, k) \odot B(k, j) = (A \cdot B)[i, j]$. A witness matrix W has $W(i, j)$ as a witness for $(A \odot B)[i, j]$.*

In the last section, we will prove the following theorem, which essentially shows that finding the witness matrix is equivalent to computing the $(\min, \odot)$-product, up to poly-logarithmic factors.

Theorem 1. *If there is an algorithm for $(\min, \odot)$-product in $T(n)$ time, then there is a randomized algorithm computing W w.h.p. running in $O(T(n) \log^3 n)$ time.*

1 Applications of the Witness Matrix

In many cases, witness matrices can be used to construct the successor matrix for the shortest paths.

1.1 All-pairs shortest paths for unweighted undirected graphs

We presented a nice algorithm by Seidel for APSP in unweighted undirected graphs. In this section, we will show that if witnesses for BMM can be computed in $T(n)$ time (thus we can also compute BMM in $T(n)$ time), then we can compute predecessor matrix in undirected unweighted graphs in $\tilde{O}(T(n))$ time. In the Pset, you will show how to compute APSP in unweighted undirected graphs in $O(T(n))$ time, assuming BMM can be computed in $T(n)$ time. Thus, it remains to show how to find the predecessor matrix if we are given the distance matrix d.

One of the main ideas in Seidel's algorithm was that for all i, j, and a neighbor k of j, $d[i, j]$ and $d[i, k]$ differ by at most 1. Moreover, any k that satisfies $d[i, k] = d[i, j] - 1$ is a valid predecessor of j in path $i \rightsquigarrow j$. Now, because $d[i, k]$ can only take values from $\{d[i, j] - 1, d[i, j], d[i, j] + 1\}$, we know that $d[i, k] \equiv d[i, j] - 1 \pmod 3$ implies that k is a predecessor. This fact can be used to compute the predecessor matrix, given the distance matrix d, and the adjacency matrix A.

For $s = 0, 1, 2$, construct matrices $D^{(s)}$ as below:

$$D^{(s)}[i, k] = \begin{cases} 1 & \text{if } d[i, k] \equiv s - 1 \pmod 3, \\ 0 & \text{otherwise.} \end{cases}$$

Then, we compute the Boolean matrix product $D^{(s)}A$, and the corresponding witness matrix $W^{(s)}$. Now, for each i and j, set $P[i, j] = W^{(s_{ij})}[i, j]$, where $s_{ij} = D[i, j] \mod 3$. To see why this is correct, fix i and j, and consider the (i, j)-th entry of the product $D^{(s_{ij})} \cdot A$. The index k that contributes to this entry satisfies $D^{(s_{ij})}[i, k] = A[k, j] = 1$. By construction, this means that $d[i, k] \equiv d[i, j] - 1 \pmod 3$, and from the previous observation, we can conclude that k is a predecessor of j in the shortest path $i \rightsquigarrow j$.

1.2 Transitive closure via successive squaring

Consider computing the transitive closure of a graph G by successively squaring the adjacency matrix A, using the Boolean matrix product. If there is a path from i to j, then the (i, j)-th entry of the successor matrix S should give the node next to i in some simple path $i \rightsquigarrow j$. If there is no path, then $S[i, j]$ can be arbitrary (this is consistent with the definition of successor matrices). We show that the successor matrix can be computed along with the transitive closure, using Algorithm **??**.

Algorithm **??** returns the transitive closure T, along with the successor matrix S. To see its correctness, we argue that $S^{(k)}$ is a correct successor matrix for all pairs of nodes i, j such that j is reachable from i via a path of length $\leq 2^k$. The base case $k = 0$ is obtained directly from the adjacency matrix. When $k > 0$, if there is a path $i \rightsquigarrow j$ of length at most 2^k, then the corresponding entry of $W^{(k)}$ gives a midpoint node w

Algorithm 1: TransitiveClosure(G, A)

$A^{(0)} \leftarrow A + I$;
For all i and j, $S^{(0)}[i,j] \leftarrow j$ if $A[i,j] = 1$;
for $k = 1, \ldots, \log n$ **do**
  $A^{(k)} \leftarrow (A^{(k-1)})^2$;
  $W^{(k)} \leftarrow$ witness matrix of the product above;
  for *every pair* (i,j) **do**
    if $A^{(k-1)}[i,j] = 1$ **then**
      $S^{(k)}[i,j] = S^{(k-1)}[i,j]$
    else
      $S^{(k)}[i,j] = S^{(k-1)}\left[i, W^k[i,j]\right]$
  For all i and j, $S^{(k)}[i,j] \leftarrow S^{(k-1)}[i, W^{(k)}[i,j]]$;
$T \leftarrow A^{(\log n)}$;
$S \leftarrow S^{(\log n)}$;
return (T, S);

so that there are paths $i \rightsquigarrow w$ and $w \rightsquigarrow j$, both of which have lengths at most 2^{k-1}. The successor of path $i \rightsquigarrow j$ can then be retrieved from the successor of path $i \rightsquigarrow w$, which is already computed in $S^{(k-1)}$. This shows that $S^{(k)}$ is indeed a correct successor matrix.

1.3 All-pairs shortest paths for weighted graphs via Zwick's algorithm

Assume we can get the witnesses for $(\min, +)$-product of matrices A and B with entries in $\{-M, \ldots, M\} \cup \{\infty\}$ in $\tilde{O}(Mn^\omega)$ time. Then the idea in the previous example can readily be applied to Zwick's algorithm. Recall, from lecture 4, that Zwick's algorithm does a matrix multiplication in each of its $\log_{3/2} n$ iterations, computing the distances between nodes connected via paths of length $\leq (3/2)^\ell$ in iteration ℓ. In iteration ℓ, we need to compute the $(\min, +)$-product of an $n \times O(\frac{n}{(3/2)^\ell})$ matrix and an $O(\frac{n}{(3/2)^\ell}) \times n$ matrix, such that the entries of both matrices are bounded by $M(3/2)^\ell$. For each of these matrix multiplications, we can find the associated witness matrices by splitting each matrix to square matrices, and find the witnesses of the $(\min, +)$-product between each pair of square matrices. The witness matrix for iteration ℓ finds a midpoint of each shortest path of length in $[(3/2)^{\ell-1}, (3/2)^\ell)$. The midpoint w of any such $i \rightsquigarrow j$ path lies in the "middle third" of the path, and hence similar to Algorithm **??**, we can find the successor of i on the path by setting it to the successor of i on the $i \rightsquigarrow w$ path computed in the previous iteration.

2 Computing Witness Matrices

In this section, we show an algorithm for computing witness matrices associated with the matrix product $A \odot B$, so that the successor matrix can be computed as described in the previous part.

Before handling the general case, we first focus on an easier variant of the problem, in which the witnesses are unique.

Definition 2.1. *A matrix U is a unqiue witness matrix for the $(\min, \odot)$-product of A and B if $U[i,j] = k$ whenever k is the unique minimizer of $A[i,k] \odot B[k,j]$.*

Note by the above definition, when (i,j) has more than one witnesses, $U[i,j]$ can have arbitrary values.

In the case of Boolean matrix multiplication, computing a unique witnesses matrix is easy; let A' be a matrix defined by $A'[i,k] = kA[i,k]$. Then, the integer matrix product $A'B$ directly gives the unique witness

matrix. Even under general matrix products, computing a unique witness matrix isn't much harder than computing the matrix product itself, as showed by the following lemma.

Lemma 2.1. *If the* $(\min, \odot)$*-product takes* $T(n)$ *time, then we can compute a unique witness matrix in* $O(T(n) \log n)$ *time.*

Proof. We use a procedure that determines the witnesses bit by bit. For a subset S of $\{1, \ldots, n\}$, let's write $A[\cdot, S]$ to denote the submatrix of A, formed by taking the columns that are indexed by S. Define $B[S, \cdot]$ similarly. Now, for each $b = 1, \ldots, \log n$, let $S_b = \{k : b\text{th bit of } k \text{ is } 1\}$, and compute $C_b \leftarrow A[\cdot, S_b] \odot B[S_b, \cdot]$. For every i and j, if $C_b[i, j] = C[i, j]$, we set the bth bit of $W[i, j]$ to 1, and to 0 otherwise. This procedure calls the algorithm for $(\min, \odot)$ product $\log n$ times, so the total running time is $O(T(n) \log n)$.

It is also easy to see why this algorithm is correct. Fix a pair (i, j) that has a unique witness k. If the b-th bit of k is 1, then $k \in S_b$ and thus $C_b[i, j] = C[i, j]$; otherwise, $k \notin S_b$, so $C_b[i, j] < C[i, j]$. □

Finally, we show how to compute the more general witness matrix, given an algorithm for computing unique witnesses. To do this, we do random samplings.

Claim 2.1. *Assume that the pair* (i, j) *has* c *witnesses, where* $n/2^{s+1} \leq c < n/2^s$*. Suppose we pick* 2^s *elements from* $\{1, \ldots, n\}$ *uniformly at random with replacement, and let* S *be the set of elements picked. Then,* S *contains a unique witness for* i, j *with probability at least* $\frac{1}{2e}$*.*

Proof. Let W be the set of witnesses of i, j, and let W have size c. Then,

$$\mathbb{P}|S \cap W| = 1] \geq 2^s \cdot \frac{c}{n} \cdot \left(1 - \frac{c}{n}\right)^{2^s - 1} \geq 2^s \cdot \frac{1}{2^{s+1}} \cdot \left(1 - \frac{1}{2^s}\right)^{2^s - 1} \geq \frac{1}{2e}.$$

□

Now we are ready to prove Theorem **??**.

Proof of Theorem **??**. Claim **??** can be used to design a randomized algorithm that outputs a correct witness matrix. We do not know the number of witness for each individual (i, j) pair, but we know that it has to be contained in the intervals $[n/2^{s+1}, n/2^s)$ for some $s = 0, 1, \ldots, \log n$. The idea is to loop over all possible values of s, and determine the witnesses for all the (i, j) index pairs for which the number of witnesses fall into the right interval. Formally, the algorithm can be written as below.

Algorithm 2: WitnessMatrix(A, B)

```
W ← ∞;
C ← A ⊙ B;
repeat
    for s = 0, 1, ..., log n do
        S ← random subset of {1, ..., n} of size 2^s;
        W' ← unique witness matrix for A[·, S] ⊙ B[S, ·];
        for i, j = 1, ..., n do
            if A[i, W'[i, j]] ⊙ B[W'[i, j], j] = C[i, j] then
                W[i, j] ← W'[i, j];
until all elements of W are determined;
return W;
```

For every outermost iteration, each $W[i, j]$ is filled with a correct witness with probability at least $\frac{1}{2e}$ by Claim **??**, and once it is determined, it will never be changed to an incorrect value. If we repeat the outermost iteration $O(\log n)$ times, every entry of W will be determined with high probability.

Note that the outermost iteration gives a $\log n$ factor, the loop of s has a $\log n$ factor, and the unique witness algorithm has a $\log n$ factor. Thus, the overall runtime is $O(T(n) \log^3 n)$. □