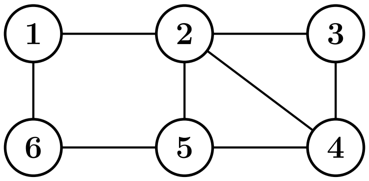
University of Tokyo: Advanced Algorithms **Summer 2010**

Lecture 8 — 10 June

Lecturer: Christian Sommer *Scribe: Florian Wagner*

8.1 Graph Representation in Memory

There are several possibilities to represent a graph $G = (V, E)$ in memory. Let the set of nodes be $V = \{1, 2, \ldots, n\}$ with edges $E \subseteq V \times V$, and let $m := |E|$.



1. **Adjacency matrix**

   The adjacency matrix of a graph $G = (V, E)$ is a $|V| \times |V|$ matrix A, where each entry a_{ij} is equal to 1 if there exists an edge $e = (v_i, v_j) \in E$ and 0 otherwise. In case a weight function $w : E \to \mathbb{R}$ is given, then $a_{ij} = w\big(v_i, v_j\big)$.

   The adjacency matrix of the graph depicted above is

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

   Note that, since the graph in the example is undirected, the resulting matrix is symmetric. Also, since there are no self loops, each entry in the main diagonal is zero.

2. **Edge List**

   In an edge list L, every edge e is stored as a tuple (v_i, v_j) in a sorted list. The edge list of the graph depicted above is:

$$L = \big\{(1,2), (1,6), (2,3), (2,4), (2,5), (3,4), (4,5), (5,6)\big\}$$

   While being a space-efficient way to store the graph, the access times of an edge list are slower than the access times of an adjacency matrix (for most edges).

3. **Adjacency List**

   An adjacency list is similar to an edge list: for each node a sorted list of adjacent nodes (also termed *neighbors*) is stored. Likewise, this is an efficient way to represent a graph but the access times are slower compared to a matrix ($\mathcal{O}(\lg n)$ compared to $\mathcal{O}(1)$). One way to solve this issue is to use a hash table for each node, containing all its neighbors.

   The adjacency list for the example graph is:

Node	Neighbors
1	{2, 6}
2	{1, 3, 4, 5}
3	{2, 4}
4	{2, 3, 5}
5	{2, 4, 6}
6	{1, 5}

Remark. The optimal representation depends on the type of the graph. For instance, a matrix wastes a lot of memory if a sparse graph is given, but it has optimal access times for any type of graph.

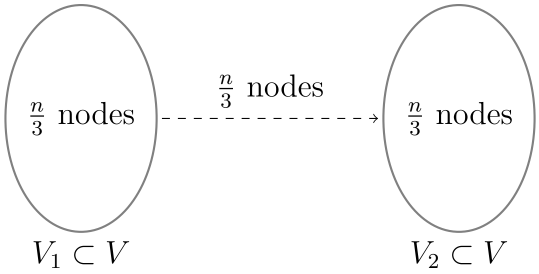
8.2 Shortest Path Problems

In this section the problems of finding paths $p = \langle v_1, v_2, \ldots \rangle$ and distances $d(v_{start}, v_{end})$ between vertex pairs will be discussed. The problems can be divided into three categories:

1. **Single source shortest path problem**: Find the shortest path from a given source $s \in V$ to all other nodes $v \in V$. For unweighted graphs, this problem can be solved using breadth-first-search in time $\mathcal{O}(m)$.

2. **All pairs shortest path problem**: Find the shortest path for *all* pairs $(u, v) \in V \times V$. Since an all-pairs shortest paths algorithm must output all the pair-wise distances, the running time is at least $\Omega(n^2)$. If the actual paths need to be output explicitly, the running time increases to $\Omega(n^3)$.

   The second lower bound follows from a graph on n nodes as in the picture below. We first divide the set of nodes into two sets with $\frac{n}{3}$ nodes each and a path with another $\frac{n}{3}$ nodes. For at least $\frac{n^2}{9}$ pairs, any shortest path uses at least $\frac{n}{3}$ edges.



3. **All pairs shortest distances**: Compute the distances d (instead of the whole path) for all pairs $(u, v) \in V \times V$. In many scenarios a weight function (such as distance, time, or cost) is provided: $w : E \rightarrow \mathbb{R}$. In the following, we mainly consider *unweighted* graphs (all weights are 1). For these graphs and for a single source, breadth-first search (BFS) can be applied. The running time is $\mathcal{O}(m)$ for SSSP and $\mathcal{O}(mn)$ for APSP.

   For sparse graphs ($m = \mathcal{O}(n)$ edges), this is an optimal solution. In the following, methods for dense graphs will be discussed.

Definition The *weight* of a path $p = \langle v_0, v_1, ..., v_k \rangle$ is defined as

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

First, the well-known Floyd-Warshall algorithm is presented. Second, a more advanced algorithm by Seidel [Sei95] is explained.

8.3 Floyd-Warshall Algorithm

This algorithm can handle negative weights where negative cycles are not allowed. Below, the matrix A is an adjacency matrix A with entries a_{ij} and matrix D is a distance matrix with entries d_{ij} that contain the distances of the shortest path between each pair of nodes i and j.

Observation. We define matrices $D^{(k)}$ (for integers k) as intermediate results. Each entry $d_{ij}^{(k)}$ contains the weight of the shortest path from i to j, such that all intermediate nodes are in the set $\{1, 2, \ldots, k\}$. The entries of $D^{(0)}$ are initialized as follows:

$$d_{ij}^{(0)} = \begin{cases} a_{ij} & \text{if } a_{ij} > 0 \qquad \textit{i.e. } (v_i, v_j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

Starting with this matrix, the algorithm computes the matrices $D^{(0)}$ for $k = 1, 2, \ldots, n$ as follows:

$$d_{ij}^{(k)} = \min\left(\underbrace{d_{ij}^{(k-1)}}_{\substack{\text{not using} \\ \text{node } k}}, \underbrace{d_{ik}^{(k-1)} + d_{kj}^{(k-1)}}_{\text{using node } k} \right)$$

Either by adding the node k to the set of possible intermediate nodes the distances do not change $\left(d_{ij}^{(k)} = d_{ij}^{(k-1)}\right)$ or a shorter path from i through k to j exists.

Algorithm 1: Floyd-Warshall algorithm

```
Input: Adjacency matrix $A$ (=: $D^{(0)}$)
Output: $D^{(n)}$
for $k \leftarrow 1$ to $n$ do
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $n$ do
            $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
        end
    end
end
```

The running time of the Floyd-Warshall algorithm (Algorithm 1) is $\Theta(n^3)$, due to its three nested loops. An important observation is that the structure of the algorithm is similar to matrix multiplication. In the following, the problem of computing the APSP problem will be reduced to multiplying two integer matrices.

8.4 Floyd-Warshall vs. Matrix Multiplication

Two matrices A and B are multiplied, in this particular case square matrices $n \times n$, by multiplying each row vector of the first matrix with each column vector of the second matrix:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \\ b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\ b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \\ b_{51} & b_{52} & b_{53} & b_{54} & b_{55} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} \\ c_{51} & c_{52} & c_{53} & c_{54} & c_{55} \end{pmatrix}$$

The resulting matrix C with entries c_{ij} is calculated as follows:

$$C = AB$$

$$\Rightarrow c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

The formula for the product of two boolean matrices A and B is the following:

$$C = A \cdot B$$

$$\Rightarrow c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj}$$

Finally, as seen in the Floyd-Warshall algorithm, for calculating the distance matrix, a "distance product" is used:

$$C = A \otimes B$$

$$\Rightarrow c_{ij} = \min_{k \in \{1..n\}} \Big(a_{ik} + b_{kj} \Big)$$

These similarities indicate that the all-pairs shortest paths problem and the integer matrix multiplication problem are somewhat related. In the following, we use fast integer matrix multiplication to obtain a faster algorithm for the all-pairs shortest paths problem. Matrix multiplication implemented with three nested loops runs in time $\mathcal{O}(n^3)$. The more sophisticated algorithm by *Strassen* [Str69] runs in time $\mathcal{O}(n^{\log_2 7})$. Currently, the fastest algorithm (by *Coppersmith and Winograd* [CW87]) runs in time $\mathcal{O}(n^{\omega})$ for $\omega = 2.376$. Note that any algorithm that multiplies two $n \times n$ integer matrices must have $\omega \geq 2$.

8.5 Seidel's Algorithm

Seidel's APSP algorithm [Sei95] is based on fast integer matrix multiplication. His algorithm can be used for unweighted, undirected graphs.

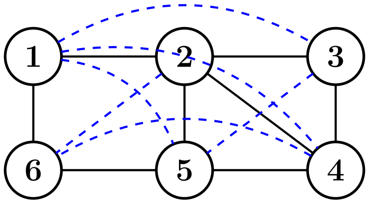
8.5.1 All Pairs Distances

We begin with an outline of Seidel's algorithm (see Algorithm 2).

Algorithm 2: Seidel's Algorithm (**A**ll **P**airs **D**istances)

```
1  function APD(A)
2  begin
3      $Z \leftarrow AA$
4      let $B$ be a boolean matrix with $b_{ij} \leftarrow \begin{cases} 1 & \text{if } i \neq j \textbf{ and } \big(a_{ij} = 1 \textbf{ or } z_{ij} > 0\big) \\ 0 & \text{otherwise} \end{cases}$
5      if $\forall i, j \;.\; i \neq j \Rightarrow b_{ij} = 1$ then
6          return $D \leftarrow 2B - A$
7      end
8      else
9          $T \leftarrow APD(B)$
10         $X \leftarrow TA$
11         return $D$ with $d_{ij} \leftarrow \begin{cases} 2t_{ij} & \text{if } x_{ij} \geq t_{ij} \cdot \deg(j) \\ 2t_{ij} - 1 & \text{otherwise} \end{cases}$
12     end
13 end
```

In the following, we explain how Algorithm 2 works using the example graph from above. At the beginning, the algorithm computes the *square graph*. Following one edge in the square graph amounts to following a path of length one or two in the original graph. In order to obtain the adjacency matrix of the square graph, first, the matrix A is squared (one matrix multiplication) and stored into Z in line 3. The graph and the matrix for the example are:



$$Z = AA = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 1 & 1 & 2 & 0 \\ 0 & 4 & 1 & 2 & 1 & 2 \\ 1 & 1 & 2 & 1 & 2 & 0 \\ 1 & 2 & 1 & 3 & 1 & 1 \\ 2 & 1 & 2 & 1 & 3 & 0 \\ 0 & 2 & 0 & 1 & 0 & 2 \end{pmatrix}$$

The value of z_{ij} indicates the number of distinct paths between i and j of length 2. Second, the matrix B (the adjacency matrix of the square graph) is calculated in line 4, having a 1 in each entry if the nodes are either directly connected by an edge or if there is at least one path of length two between these two nodes. For the example, B is:

$$B = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Apart from the entries of the main diagonal, only b_{36} and b_{63} are 0. Note that there is indeed no path of length one or two between nodes 3 and 6 of the graph.

Next, **if** all the entries of B (except for the diagonal) are 1, then all pairs of nodes in the graph defined by A are connected by a path of length at most 2. It remains to distinguish pairs for which the distance is 1 from pairs for which the distance is 2. B is multiplied by 2 and A is subtracted (line 6). This way, if the path has length 2, then the entry in A is 0 and the resulting value will be 2. Otherwise, the value of A will be subtracted, yielding 1 for this entry.

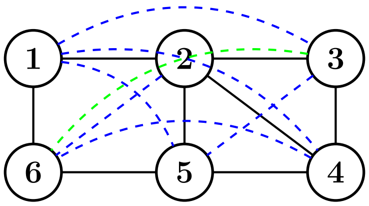
Otherwise, the algorithm is called recursively on matrix B in line 9. In the recursive call, the algorithm starts with the square graph, computes the square of the square graph, etc. Let us get back to our example. We use subscript 2 for the names of the matrices in the recursive call. The matrix $A_2 = B$ is squared:

$$Z_2 = A_2A_2 = BB = \begin{pmatrix} 5 & 4 & 3 & 4 & 4 & 3 \\ 4 & 5 & 3 & 4 & 4 & 3 \\ 3 & 3 & 4 & 3 & 3 & 4 \\ 4 & 4 & 3 & 5 & 4 & 3 \\ 4 & 4 & 3 & 4 & 5 & 3 \\ 3 & 3 & 4 & 3 & 3 & 4 \end{pmatrix}$$

Next, B_2 is computed:

$$B_2 = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

This time, all the entries of B_2 are 1, except for the main diagonal. That is, the matrix describes a complete graph K_n:



The recursion ends, B_2 is multiplied by 2 and $A_2 = B$ is subtracted. The resulting matrix is returned to the first call of the function and stored in T:

$$T = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 2 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 2 & 1 & 1 & 0 \end{pmatrix}$$

Note that, according to T, the distance from node 3 to node 6 is 2, meaning that, since T corresponds to the distance in the square graph, the distance could be either 4 or also (like in this example) 3. It remains to distinguish two cases: whether the actual distance from node i to node j is $2t_{ij}$ or $2t_{ij} - 1$. In order to distinguish these two cases, the matrix T is multiplied with A and stored in X:

$$X = AT = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 2 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 2 & 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 2 & 3 & 3 & 1 \\ 1 & 4 & 1 & 2 & 2 & 2 \\ 3 & 3 & 2 & 2 & 4 & 2 \\ 2 & 3 & 1 & 3 & 2 & 2 \\ 2 & 3 & 2 & 2 & 3 & 1 \\ 1 & 5 & 2 & 4 & 2 & 2 \end{pmatrix}$$

Finally, in line 11 the resulting matrix D is returned, containing, for each of its entries, either the original value or the decrement of it. The following four propositions shall explain this formula.

Proposition 8.1. *If the distance d_{ij} is even, the resulting distance is $2t_{ij}$ and $2t_{ij} - 1$ otherwise.*

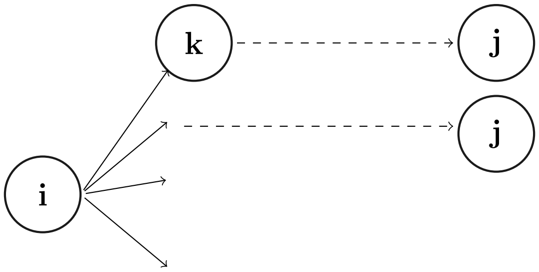
Proof: If d_{ij} is even, i.e. $d_{ij} = 2s$ and the shortest path is $\langle v_0, v_1, \ldots, v_{2s} \rangle$ then the path $\langle v_0, v_2, v_4, \ldots, v_{2s-2}, v_{2s} \rangle$ is the shortest path in the modified graph resulting from the square matrix. If the distance d_{ij} is odd, i.e. $d_{ij} = 2s - 1$, then the shortest path in the modified graph is $\langle v_0, v_2, v_4, \ldots, v_{2s-4}, v_{2s-2}, v_{2s-1} \rangle$. □

Next a trivial claim:

Proposition 8.2. *Let i and j be a pair of distinct vertices in G. For any neighbor k of j, we have*

$$d_{ij} - 1 \leq d_{ik} \leq d_{ij} + 1$$

Proof: By taking one step the distance can only change by 1 or it remains the same, depending on whether the neighbor k is the first node on a shortest path from i to j or not (see figure below).



□

Furthermore, there exists a neighbor k of i such that $d_{ik} = d_{ij} - 1$

Proposition 8.3.

$$\begin{aligned} d_{ij} \text{ is even} &\rightarrow t_{ik} \geq t_{ij} \text{ for all neighbors } k \text{ of } j \\ d_{ij} \text{ is odd} &\rightarrow t_{ik} \leq t_{ij} \text{ for all neighbors } k \text{ of } j\text{, and} \\ & t_{ik} < t_{ij} \text{ for at least one neighbor } k \text{ of } j \end{aligned}$$

Proof: If d_{ij} is even, i.e. $d_{ij} = 2l$, then $t_{ij} = l$. There also exists a k such that $d_{kj} \geq 2l - 1$.

$$t_{kj} \geq \frac{d_{kj}}{2} \geq l - \frac{1}{2}$$

Distances are integral, so $t_{kj} \geq t_{ij}$. The proof for an odd d_{ij} is analogous. □

Proposition 8.4. *Let $\Gamma(i)$ be the set of neighbors of node i.*

$$\begin{aligned} \sum_{k \in \Gamma(i)} t_{kj} \geq t_{ij} \deg(i) &\rightarrow d_{ij} \text{ is even} \\ \sum_{k \in \Gamma(i)} t_{kj} < t_{ij} \deg(i) &\rightarrow d_{ij} \text{ is odd} \end{aligned}$$

Thus, the resulting matrix D for this example is:

$$D = \begin{pmatrix} 0 & 1 & 2 & 2 & 2 & 1 \\ 1 & 0 & 1 & 1 & 1 & 2 \\ 2 & 1 & 0 & 1 & 2 & 3 \\ 2 & 1 & 1 & 0 & 1 & 2 \\ 2 & 1 & 2 & 1 & 0 & 1 \\ 1 & 2 & 3 & 2 & 1 & 0 \end{pmatrix}$$

In an unweighted, connected graph, the maximum distance is at most n. In each recursive call of Seidel's algorithm, distances are doubled. The recursion depth is thus at most logarithmic in n. Therefore, the total running time is the required time to multiply two integer matrices, multiplied by $\mathcal{O}(\log n)$.

8.5.2 All Pairs Shortest Paths

In the following we discuss how to retrieve the actual paths. While Algorithm 2 tells us the distance, we do not know which neighbor to "use" without potentially inspecting all of them. In the Floyd-Warshall algorithm, these intermediate nodes were computed as a side product. In Seidel's algorithm, we need to compute them. The intermediate nodes "witness" that the shortest path distance in the distance matrix is correct. In the following, we will compute these "witnesses."

Note that when the adjacency matrix A is multiplied with itself ($Z = AA$ as in line 3), then the entry $z_{ij} = \sum_k a_{ik} b_{kj}$ contains the number of paths from i to j with length 2. Instead, we would like to know one of these neighbors k between node i and node j. More precisely, we wish to compute a "witness" for each non-zero entry of the boolean product matrix C

$$c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj}.$$

We will now discuss a randomized algorithm to calculate a boolean product witness matrix. Again, instead of computing the boolean product matrix and the corresponding witnesses using the explicit formula (which requires cubic time), we wish to apply fast integer matrix multiplication.

Reminder:

$$AB = C \qquad c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \tag{8.1}$$

$$A \cdot B = C \qquad c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj} \tag{8.2}$$

For the boolean product matrix $C := A \cdot B$, if c_{ij} is 1, then there must be some values k for which $a_{ik} \wedge b_{kj}$ is 1 — the goal is to find one of these k as a *witness*. The values c_{ij} of the integer product matrix (equation 8.1) "count" the number of witnesses; if both a_{ik} and b_{kj} equal 1 then the value of c_{ij} increases by 1. By multiplying column k of matrix A by a factor k, the value c_{ij} of the integer product matrix increases by k if both a_{ik} and b_{kj} are 1:

$$\bar{c}_{ij} = \sum_{k=1}^{n} k \cdot a_{ik} \cdot b_{kj}$$

If there is just one witness k (meaning a pair (a_{ik}, b_{kj})), then $\bar{c}_{ij} = k$ and we have found a witness. However, since there may be more than one witness k for a path from i to j, $\bar{c}_{ij}$ is not necessarily equal to a witness k. At this point, we use randomization. Instead of summing over all values $k \in \{1, 2, \dots n\}$, we sum over random subsets of various sizes $k \in \{k_1, k_2, \dots k_d\}$:

$$\bar{c}_{ij} = \sum_{k \in \{k_1, k_2, \dots k_d\}}^{n} k \cdot a_{ik} \cdot b_{kj}$$

In some sense, the algorithm tries to “hit” exactly one witness. The sizes d of the random subsets $\{k_1, k_2, \dots k_d\}$ are in $\{1, 2, 4, 8, \dots\}$. If for an entry c_{ij} there are “many” witnesses k, then small values of d are likely to “produce” a witness. If there are only “few” witnesses, then large values of d are likely to “produce” a witness.

Algorithm 3: Seidel’s Algorithm (**B**oolean **P**roduct **W**itness **M**atrix)

```
   Input: Matrices $A, B$
1  function BPWM($A, B$)
2  begin
3      let $W = -AB$ (witness matrix)
4      for $l \leftarrow 0$ to $\lg n$ do
5          $d \leftarrow 2^l$
6          repeat $\ln n$ times
7          begin
8              choose $d$ random numbers $k_1, k_2, \dots, k_d$ from $\{1, 2, \dots, n\}$
9              let $X$ be the $n \times d$ matrix with columns $k_i \cdot a_{k_i}$
10             let $Y$ be the $d \times n$ matrix with rows $b_{k_i}$
11             $C \leftarrow XY$
12             update $w_{ij} \leftarrow c_{ij}$ if $c_{ij}$ is a witness
13         end
14     end
15     foreach $(i, j)$ s.t. $w_{ij} < 0$ do
16         $w_{ij}$ := some witness $k$ for $(i, j)$, computed by trying each $k$
17     end
18     return $W$
19 end
```

In lines 9 and 10 the matrices X and Y are computed, where:

$$\underbrace{\begin{pmatrix} k_1 & k_2 & \cdots \\ 0 & 0 & \cdots \\ k_1 & 0 & \cdots \\ k_1 & k_2 & \cdots \\ 0 & 0 & \cdots \\ k_1 & 0 & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}}_{d} \left.\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 & \cdots \\ 1 & 0 & 0 & 1 & 0 & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}\right\} d = \underbrace{\begin{pmatrix} g_{11} & g_{12} & \cdots \\ g_{21} & g_{22} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}}_{n} \left.\vphantom{\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}}\right\} n$$

In line 15 the algorithm computes a witness for those pairs for which the randomized procedure was not successfull. We iterate over all pairs (i, j) and if no witness has been found so far, then all the values for $k \in \{1, \dots n\}$ are tested.

This algorithm is a Las Vegas algorithm with expected running time $\mathcal{O}\big(n^\omega \log^2 n\big)$ [Sei95, Theorem 3].

Example

Matrices A and B:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \qquad B = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

In the second iteration, we have $l = 1$ and therefore $d = 2$ (line 5). Next, let $k_1 = 3$ and let $k_2 = 4$. The matrices X and Y are:

$$X = \begin{pmatrix} & & 0 & 0 & & \\ & & 3 & 4 & & \\ & & 0 & 4 & & \\ & & 3 & 0 & & \\ & & 0 & 4 & & \\ & & 0 & 0 & & \end{pmatrix} \qquad Y = \begin{pmatrix} & & & & & \\ & & & & & \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ & & & & & \\ & & & & & \end{pmatrix}$$

The resulting matrix C is:

$$C = XY = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ \textcircled{7} & 0 & 0 & 0 & \textcircled{3} & 4 \\ 4 & 0 & 0 & 0 & 0 & 4 \\ 3 & 0 & 0 & 0 & 3 & 0 \\ 4 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

For entry c_{25} we found a witness. For entry c_{21} we were “unlucky”: two witnesses were “hit” and their sum does not tell us anything.

In the following, we give a lower bound on the probability that we find a witness for c_{ij}. Let i and j be fixed. A witness w_{ij} is found if there is *exactly one* witness amoung $k_1, k_2, \ldots, k_d$. Let $c = c_{ij}$. Look at the iterations for which d satisfies:

$$\frac{n}{2} \leq cd \leq n$$

Since there are c witnesses, the probability that k_i “hits” a witness k is c/n. The probability that exactly one witness is “hit” is at least

$$d\frac{c}{n}\left(1 - \frac{c}{n}\right)^{d-1} \geq \frac{1}{2}\left(1 - \frac{1}{d}\right)^{d-1} > \frac{1}{2e}.$$

For each value of d, the algorithm chooses $\ln n$ random subsets of size d. The probability that in $\ln n$ rounds no witness is found is proportional to $1/n$. The expected number of pairs (i, j) for which no witness is found is thus at most $\mathcal{O}(n)$. For each of these pairs, a brute-force linear-time algorithm eventually finds a witness. The expected total running time of this final step is thus bounded by $\mathcal{O}(n^2)$.

For the combination of the BPWM and APD algorithms we refer to [Sei95, Algorithm APSP, page 402].

Bibliography

[CW87] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, New York, NY, USA, 1987. ACM.

[Sei95] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51(3):400–403, 1995.

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