DATA MINING ON SOCIAL NETWORKS

Laboratory Lectures Notes

Dionisios N. Sotiropoulos, Ph.D

June 26, 2018

Department of Computer Science
University of Piraeus

Co-Authorship Network: Dataset

Data Files:

- authors.mat: [1155x3] cell array **authors** storing column-wise
  - `author_id`
  - `author_surname`
  - `author_firstname`
- ICMB_2002.mat … ICM_2013.mat: [1155×1155] matrices **array_2002** … **array_2013**

Temporal Adjacency Matrices

Each $\mathsf{W_t} \in M_{1155\times1155}$ with $t \in \{2002 \ldots 2013\}$ is a symmetric adjacency matrix where the $\mathsf{W_t}(i,j)$ elements quantify the number of papers that have been co-authored between authors i and j.

Specifically, $\mathsf{W_t} = [\mathsf{W_t}(i,j)]$ such that:

$$W_t(i,j) = \begin{cases} \text{\# papers co-authored between authors i and j at time t,} i \neq j; \\ \text{\# of papers authored by author i at time t,} i = j; \end{cases}$$

* **Information along the diagonal elements will be discarded**.

Co-Authorship Network: Overall Adjacency Matrix

The final co-authorship network weight matrix $\mathbf{W_0} \in M_{1155\times1155}$ to be constructed will be of the following form:
$\mathbf{W_0} = [\mathbf{W_0}(i,j)]$ where:

$$W_0(i,j) = \begin{cases} 1, i \neq j \text{ when authors } i \text{ and } j \text{ have at least one paper in common;} \\ 0, i = j. \end{cases}$$

The previous equation may be equivalently expressed as:

$$W_0(i,j) = \begin{cases} 1, \sum_t W_t(i,j) > 1 \; i \neq j; \\ 0, i = j. \end{cases}$$

Co-Authorship Network: Code Requirements

Matlab Routines to be Implemented:

1. Load separate weight-matrices and construct overall network weight matrix.
2. Compute Degree Centrality Measure.
3. Construct the Degree Centrality distribution graph.
4. Report top N authors ranked by Degree Centrality (or any other centrality measure).
5. Implement algorithm for extracting connected components (**Breadth First Search Algorithm**).
6. Report top N connected components (ranked by size).
7. Implement Shortest Path extraction algorithm from predecessor matrix (**Floyd-Warshall Algorithm**).

CoAuthorshipNetworkManipulation.m I

```
clc
clear all
% Set the period of years.
Years = [2002:1:2013];
YearsNum = length(Years);
% Load weight matrices for each year.
for year = Years
    filename = strcat(['ICMB-' num2str(year) '.mat']);
    load(filename);
end;
% Load authors' names.
```

```
load('authors.mat');
% Set a container storing the weight matrices for
% all years.
ICMB = cell(1,numel(Years));
% Populate cell array
for y = 1:YearsNum
    ICMB{y} = eval(genvarname(strcat(['array_'
        num2str(Years(y))])));
end;
% Get the number of nodes N.
N = size(ICMB{1},1);
% Construct the overall graph weight matrix.
W = zeros(N,N);
```

CoAuthorshipNetworkManipulation.m III

```
for y = 1:1:YearsNum
    W = W + ICMB{y};
end;
% Set up a vector of indices pointing to the
% diagonal elements of the weight matrix W.
Idiag = [1:N+1:N*N];

% Re–initialize the overall weight matrix W so that
% fundamental social network analysis tasks can be
% performed. W should be a binary adjacency matrix
% so that W[i,j] = 1 indicates the presence of an
% edge between authors i and j. Moreover, the
```

CoAuthorshipNetworkManipulation.m IV

```
% diagonal elements of W should also be set to zero.
Wo = W;
Wo(Wo>1) = 1;
Wo(Idiag) = 0;
% Extract Degree Centrality measure for each author.
Degrees = sum(Wo,2);
```

DegreeCentralityDistribution.m I

```
function [H] = DegreeCentralityDistribution(Degrees)
% This function computes and displays the Degree
% Centrality Distribution for a given vector of
% degree  centralities.
min_degree = min(Degrees);
max_degree = max(Degrees);
degrees_range = [min_degree:max_degree];
H = hist(Degrees,degrees_range);
figure('Name','Degree Centrality Distribution');
bar(degrees_range,H);
axis([min_degree-1 max_degree+1 min(H) max(H)+5]);
```

DegreeCentralityDistribution.m II

```
xlabel('Degrees');
ylabel('Absolute Frequency');
grid on
end
```

ReportTopNAuthors.m I

```
function ReportTopNAuthors(MeasureValues,MeasureName,N,authors)
% This function reports the top N authors ranked by
% the measure identified by the input parameter
% MeasureName. The corresponding measure values are
% stored within the vector MeasureValues. The
% number of N and the complete list of authors'
% names are also given as input to the function.
[SortedValues,SortedIndices] = sort(MeasureValues,'descend');
TopNSortedValues = SortedValues(1:N);
TopNSortedIndices = SortedIndices(1:N);
```

ReportTopNAuthors.m II

```
TopNAuthorsFirstNames = authors(TopNSortedIndices,3);
TopNAuthorsSurNames = authors(TopNSortedIndices,2);
% Report Top N Authors' List.
fprintf('Top %d Authors according to %s\n',N, MeasureName);
for k = 1:1:N
    fprintf('%s %s: %d\n',TopNAuthorsSurNames{k}, TopNAuthorsFirstNames{k},TopNSortedValues(k));
end;
end
```

CoAuthorshipNetworkManipulation.m I

```
% Compute and display the degree centrality
% distribution.
H = DegreeCentralityDistribution(Degrees);
% Report top 10 authors according to
% Degree Centrality.
No = 10;
MeasureName = 'Degree Centrality';
MeasureValues = Degrees;
ReportTopNAuthors(MeasureValues,MeasureName,No,
    authors);
```

Connected Components I

In graph theory, a **connected component** (or just a component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths.

The connected components of a graph may be alternatively defined through the equivalence classes (induced subgraphs) of an equivalence relation.

Let $G = (V, E)$ where V = $\{v_1, v_2, ..., v_n\}$ and $E \subset V \times V$.

Let $R \subset V \times V$, defined as:

$$(u,v) \in R \Leftrightarrow uRv, \text{ if vertex (v) is reachable from (u)}$$

R is an equivalence relation because it has the following properties:

Connected Components II

1. **reflexivity**:

$$\forall u \in V,\ uRu.$$

(it holds since each vertex is reachable through the trivial path of zero length connecting each vertex to itself.)

2. **symmetricity**:

$$\forall (u, v) \in V^2,\ u \neq v:\ uRV \Rightarrow vRu.$$

(it holds since within an undirected graph the same path from (u) to (v) can be traversed backwards).

3. **transitivity**:

$$\forall (u, v, z) \in V^3,\ u \neq v \neq z : uRv \wedge vRz \Rightarrow uRz$$

(it holds since the path from (u) to (z) can be constructed through the concatenation of paths from (u) to (v) and from (v) to (z).

Breadth First Search: Algorithm

Input:

- A graph $G = (V, E)$
- A vertex $v \in V$.
- A set **Visited** of already Visited nodes initialized to the empty set ($Visited = \{\emptyset\}$).

Output:

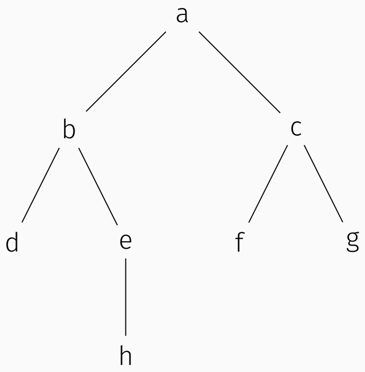
- A set **Reachable** of vertices that are reachable from vertex $v \in V$.

Breadth First Search: Algorithm Pseudo Code

```
Procedure BFS(G,v,Visited):
Reachable = {};
Q = []; %Let Q be an empty queue.
Q.enqueue(v);
Visited = Visited U {v};
while Q is not empty:
    v = Q.dequeue()
    Reachable <— Reachable U {V}
    % N(v) denotes the neighborhood of v.
    for all w in N(v):
        if w not in Visited:
            Q.enqueue(w)
            Visited = Visited U {w}
```

Breadth First Search Example I

Apply the BFS algorithm on the following graph:



NEIGHBOURS'LIST STRUCTURE:

NL(a)={b,c}

NL(b)={a,d,e}

NL(c)={a,f,g}

NL(d)={b}

NL(e)={b,h}

NL(f)={c}

NL(g)={c}

NL(h)={e}

Breadth First Search Example II

1. Calling Procedure BFS: **BFS**$(G, a, \{\emptyset\})$
2. Initialization:
    - $v = a$
    - $Visited = \{\emptyset\}$
    - $Reachable = \{\emptyset\}$
    - $Q = []$
3. Enqueue Operation: $(Q = [a], Visited = \{a\})$
4. While Loop Execution:

    4.1 $Q = [a] \neq []$:
    - Dequeue Operation: $(v = a, Q = [])$
    - $Reachable = \{a\}$
    - $NL(a) = \{b, c\}$
    - Enqueue Operation: $(Q = [b], Visited = \{a, b\})$
    - Enqueue Operation: $(Q = [b, c], Visited = \{a, b, c\})$

Breadth First Search Example III

4.2 $Q = [b, c] \neq []$:

- Dequeue Operation: ($v = b$, $Q = [c]$)
- $Reachable = \{a, b\}$
- $NL(b) = \{a, d, e\}$
- No Enqueue Operation: a is already visited
- Enqueue Operation: ($Q = [c, d]$, $Visited = \{a, b, c, d\}$)
- Enqueue Operation: ($Q = [c, d, e]$, $Visited = \{a, b, c, d, e\}$)

4.3 $Q = [c, d, e] \neq []$:

- Dequeue Operation: ($v = c$, $Q = [d, e]$)
- $Reachable = \{a, b, c\}$
- $NL(c) = \{a, f, g\}$
- No Enqueue Operation: a is already visited
- Enqueue Operation: ($Q = [d, e, f]$, $Visited = \{a, b, c, d, e, f\}$)
- Enqueue Operation: ($Q = [d, e, f, g]$, $Visited = \{a, b, c, d, e, f, g\}$)

4.4 $Q = [d, e, f, g] \neq []$:

- Dequeue Operation: ($v = d$, $Q = [e, f, g]$)

Breadth First Search Example IV

- $Reachable = \{a, b, c, d\}$
- $NL(d) = \{b\}$
- No Enqueue Operation: b is already visited

4.5 $Q = [e, f, g] \neq []$:

- Dequeue Operation: $(v = e, Q = [f, g])$
- $Reachable = \{a, b, c, d, e\}$
- $NL(e) = \{b, h\}$
- No Enqueue Operation: b is already visited
- Enqueue Operation: $(Q = [f, g, h], Visited = \{a, b, c, d, e, f, g, h\})$

4.6 $Q = [f, g, h] \neq []$:

- Dequeue Operation: $(v = f, Q = [g, h])$
- $Reachable = \{a, b, c, d, e, f\}$
- $NL(f) = \{c\}$
- No Enqueue Operation: c is already visited

4.7 $Q = [g, h] \neq []$:

Breadth First Search Example V

- Dequeue Operation: $(v = g, Q = [h])$
- $Reachable = \{a, b, c, d, e, f, g\}$
- $NL(g) = \{c\}$
- No Enqueue Operation: c is already visited

4.8 $Q = [h] \neq []$:

- Dequeue Operation: $(v = h, Q = [])$
- $Reachable = \{a, b, c, d, e, f, g, h\}$
- $NL(h) = \{e\}$
- No Enqueue Operation: e is already visited

4.9 $Q = []$ END OF WHILE LOOP

```
function [NL] = NeighboursList(W)
% This function extracts the neighbors' list
% corresponding to the weight matrix W
% which is assumed to be the binary matrix
% indicating the presence or absence of an edge
% between a given pair of nodes. Diagonal
% elements of matrix W are also assumed to be zero.
% NL is a cell array of vectors such that
% the element NL{u} stores the indices of
% nodes that are reachable from node u.
nodes_num = size(W,1);
NL = cell(1,nodes_num);
```

NeighboursList.m II

```
for v = 1:1:nodes_num
    NL{v} = find(W(v,:)==1);
end
end
```

ConnectedComponents.m I

```
function [C] = ConnectedComponents(NL)
% This function extracts the connected components
% of a given undirected graph whose neighbors' list
% NL is given as input. C is a cell array of
% vectors so that each vector stores the indices of
% each connected component.

% Initialize the cell array C storing the connected
% components of the graph.
C = cell(1,0);
% Get the number of graph nodes.
nodes_num = length(NL);
```

```
% Mark all nodes as unvisited.
 visited = false * ones(1,nodes_num);
% Initialize the number of connected components
% found so far.
components_num = 0;
for v = 1:1:nodes_num
    % If v is not visited yet, it's the start of a
    % newly discovered component containing v.

    % Process the component containing v.
    if(~visited(v))
        components_num = components_num + 1;
        % Initialize component container.
```

ConnectedComponents.m III

```
        component = [];
        % Initialize queue for implementing
        % breadth-first search.
        Q = [];
        % Start the traversal from node v.
        Q = enqueue(Q,v);
        visited(v) = true;
        while(~isempty(Q))
            [Q,w] = dequeue(Q);
            % w is a node in this component.
            component = [component,w];
            % Get all nodes neighboring w.
            node_neighbours = NL{w};
```

ConnectedComponents.m IV

```
% Traverse each unvisited node
% neighboring w.
for node_index = 1:1:length(
    node_neighbours)
     node = node_neighbours(node_index);
     if(~visited(node))
          % Another node within the
          % current component has been
          % found.
          visited(node) = true;
          Q = enqueue(Q,node);
     end
end
```

```
            end
            C{components_num} = component;
        end
    end
        function [Q] = enqueue(Q,element)
        % This is a sub-function implementing the
        % enqueue operation within a queue
        % which is realized as a vector of elements
        Q = [Q,element];
        end
        function [Q,element] = dequeue(Q)
        % This is a sub-function implementing the
        % dequeue operation within a queue
```

```
    % which is realized as a vector of elements.
    element = Q(1);
    Q = Q(2:end);
    end
end
```

ReportTopNConnectedComponents.m I

```
function [TopNComponentsSizes,TopNComponentsIndices] = ReportTopNConnectedComponents(C,N,authors)

% This function reports the top N
% (measured by size) connected components
% of the co-authorship network that are stored in
% cell array C. The number of top N components and
% the initial authors' list are passed as
% input arguments to the function.

% Get the number of connected components.
components_num = length(C);
```

```
% Get the size of each connected component.
components_sizes = zeros(1,components_num);
for k = 1:1:components_num
    components_sizes(k) = length(C{k});
end
% Sort connected components sizes in descending
% order.
[SortedComponentsSizes,SortedComponentsIndices] =
    sort(components_sizes,'descend');
% Get the top N connected components sizes and
% corresponding indices.
TopNComponentsSizes = SortedComponentsSizes(1:N);
```

ReportTopNConnectedComponents.m III

```
TopNComponentsIndices = SortedComponentsIndices(1:N);

% Report Connected Components.
% Cycle through the top N connected components:
for n = 1:1:N
    component_index = TopNComponentsIndices(n);
    component_size = TopNComponentsSizes(n);
    component = C{component_index};
    fprintf('Component %d of size %d\n', component_index,component_size);
    % Cycle through the authors of each connected
    % component:
```

```
    for m = 1:1:component_size
        author_index = component(m);
        author_firstname = authors(author_index,3);
        author_lastname = authors(author_index,2);
        fprintf('%d: %s %s\n',m,cell2mat(
            author_lastname),cell2mat(
            author_firstname));
    end
end

end
```

```
% Extract connected components of co-authorship
% network.

% Initially set the corresponding
% NeighboursList.
NL = NeighboursList(Wo);
C = ConnectedComponents(NL);

% Report top 6 connected components of the
% co-authorship network.
No = 6;
[TopNComponentsSizes,TopNComponentsIndices] =
    ReportTopNConnectedComponents(C,No,authors);
```

Floyd-Warshall: Problem Definition

Find the shortest path between every pair (v_i, v_j) of vertices on a graph $G = (V, E)$ where $V = \{v_1, \ldots, v_n\}$ and $E \subset V \times V$.

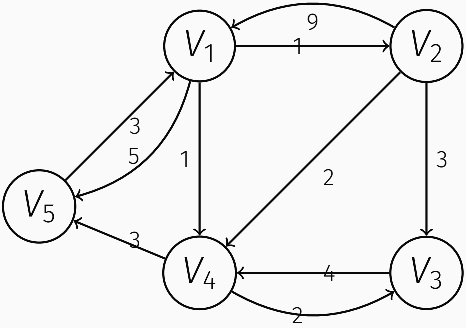
The graph may contain negative edges but not cycles with cumulative weight negative.

Weight-Matrix Representation:

- W(i,j)= 0, if i=j
- W(i,j)= ∞, if there is no edge between i and j with $i \neq j$.
- W(i,j) = ”actual weight” of the edge (i,j) with $i \neq j$.

Floyd-Warshall: Example Graph Representation I

- Example Graph:



Floyd-Warshall: Example Graph Representation II

- Weight Matrix:

$$W = \begin{bmatrix} 0 & 1 & \infty & 1 & 5 \\ 9 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

Floyd-Warshall: Smaller Problems

How can be define the shortest distance d_{ij} between nodes v_i and v_j in terms of "smaller" problems?

One way is to restrict the paths to include vertices exclusively from a restricted subset V^*.

Subset V^* is initially empty ($V^*_{(0)} = \emptyset$).

Finally, subset V^* will contain all possible intermediate nodes ($V^*_{(n)} = V$).

Let $D^{(k)}[i,j]$ to denote the weight of the shortest path from v_i to v_j using only the vertices from the set $V^*_{(k)} = \{v_1, v_2, .., v_k\}$ as intermediate vertices in the path.

- $D^{(0)}$ = W
- $D^{(n)}$ = D (which is the goal matrix)

Floyd-Warshall: Recursive Definition I

How do we compute $D^{(k)}$ from $D^{(k-1)}$?

During the execution of the k-th step of the Floyd-Warshall algorithm, matrix $D^{(k-1)}$ has been computed based on the subset of intermediate nodes: $V^*_{(k-1)} = \{v_1, v_2, .., v_{k-1}\}$.

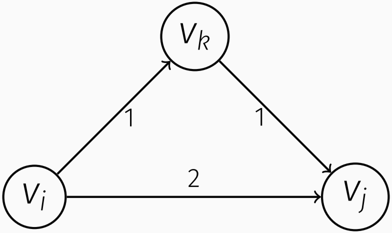
Case 1: The shortest path from v_i to v_j is composed by utilizing nodes from the set of intermediate vertices $V^*_{(k)}$ such that $v_k \notin V^*_{(k)}$. Then,

$$D^{(k)}[i, j] = D^{(k-1)}[i, j]$$

Case 2: The shortest path from v_i to v_j is composed by utilizing nodes from the set of intermediate vertices $V^*_{(k)}$ such that $v_k \in V^*_{(k)}$. Then,

$$D^{(k)}[i, j] = D^{(k-1)}[i, k] + D^{(k-1)}[k, j]$$

Floyd-Warshall: Graphical Representation



1: shortest path using intermediate vertices $\{v_1, v_2, ..., v_k\}$

2: shortest path using intermediate vertices $\{v_1, v_2, ..., v_{k-1}\}$

Floyd-Warshall: Recursive Definition II

Since,

$$D^{(k)} = \begin{cases} D^{(k-1)}[i,j], \text{if node } v_k \text{ is not included;} \\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j], \text{if node } v_k \text{ is included} \end{cases}$$

we may conclude that:

$$D^{(k)}[i,j] = \min\{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$

Floyd-Warshall: Predecessor Matrix P

P is an index matrix that can be used for extracting the full sequence of nodes that compose the shortest path between any given pair of vertices.

1. Matrix P is initialized with zeros ($\mathsf{P} = \mathbf{0}_{n \times n}$).
2. Each time the shortest path between vertices v_i and v_j is being updated by including node v_k (i.e. when $D^{(k-1)}[i,k] + D^{(k-1)}[k,j] < D^{(k)}[i,j]$) the (i,j)-th element of P is set to k (i.e. $\mathsf{P}[i,j] = k$)
3. Therefore, $\mathsf{P}[i,j] = k$ indicates that v_k is the last vertex that has to be traversed along the shortest path connecting nodes v_i and v_j.

Floyd-Warshall: Pseudo Code I

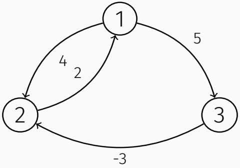
```
Floyd−Warshall(W):
D = W;
P = zeros(n,n);
for k = 1:1:n
    for i = 1:1:n
        for j = 1:1:n
            if (D[i,j]>D[i,k]+D[k,j]
               D[i,j] = D[i,k] + D[k,j];
               P[i,j] = k;
            end
        end
    end
end
```

Floyd-Warshall: Pseudo Code II

```
Path(index q,r):
% Extract intermediate nodes within the shortest
% path from vertex index (q) to vertex index (r).
if(P[q,r] != 0)
     Path(q,P[q,r]);
     println("V"+P[q,r]);
     Path(P[q,r],r);
     return;
else
     % No intermediate nodes
     return;
end
```

Floyd-Warshall Example I

Apply Floyd-Warshall Algorithm on the following graph.



Initialize Distance and Predecessor matrices D and P.

$$D^{(0)} = \begin{bmatrix} 0 & 4 & 5 \\ 2 & 0 & \infty \\ \infty & -3 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Floyd-Warshall Example II

- Step 1:
  - Determine all possible pairs for which vertex $k = 1$ can act as an intermediate node: $\{(2,3),(3,2)\}$
  - For the first pair $(2,3)$, evaluate $D^{(1)}[2,3]$ as:
    $D^{(1)}[2,3] = \min\{D^{(0)}[2,3], D^{(0)}[2,1] + D^{(0)}[1,3]\} = \min\{\infty, 2+5\} = 7$
  - Assign $P[2,3] = 1$
  - For the second pair $(3,2)$, evaluate $D^{(1)}[3,2]$ as:
    $D^{(1)}[3,2] = \min\{D^{(0)}[3,2], D^{(0)}[3,1] + D^{(0)}[1,2]\} = \min\{-3, \infty + 4\} = -3$
  - Thus, we have no change for the second pair $(3,2)$

$$D^{(1)} = \begin{bmatrix} 0 & 4 & 5 \\ 2 & 0 & 7 \\ \infty & -3 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Floyd-Warshall Example III

- **Step 2**:
  - Determine all possible pairs for which vertex $k = 2$ can act as an intermediate node: $\{(1,3),(3,1)\}$
  - For the first pair $(1,3)$, evaluate $\mathsf{D}^{(2)}[1,3]$ as: $\mathsf{D}^{(2)}[1,3] = \min\{\mathsf{D}^{(1)}[1,3], \mathsf{D}^{(1)}[1,2] + \mathsf{D}^{(1)}[2,3]\} = \min\{5, 4+7\} = 5$
  - Thus, we have no change for the second pair $(1,3)$
  - For the second pair $(3,1)$, evaluate $\mathsf{D}^{(2)}[3,1]$ as: $\mathsf{D}^{(2)}[3,1] = \min\{\mathsf{D}^{(1)}[3,1], \mathsf{D}^{(1)}[3,2] + \mathsf{D}^{(1)}[2,1]\} = \min\{\infty, -3+2\} = -1$
  - Assign $\mathsf{P}[3,1] = 2$

$$\mathsf{D}^{(2)} = \begin{bmatrix} 0 & 4 & 5 \\ 2 & 0 & 7 \\ -1 & -3 & 0 \end{bmatrix} \qquad \mathsf{P} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 2 & 0 & 0 \end{bmatrix}$$

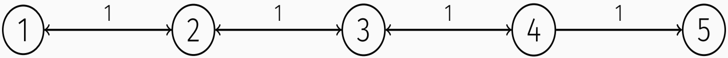
Floyd-Warshall Example IV

- **Step 3**:
  - Determine all possible pairs for which vertex $k = 3$ can act as an intermediate node: $\{(1,2),(2,1)\}$
  - For the first pair $(1,2)$, evaluate $\mathsf{D}^{(3)}[1,2]$ as: $\mathsf{D}^{(3)}[1,2] = \min\{\mathsf{D}^{(2)}[1,2], \mathsf{D}^{(2)}[1,3] + \mathsf{D}^{(2)}[3,2]\} = \min\{4, 5 + (-3)\} = 2$
  - Assign $\mathsf{P}[1,2] = 3$
  - For the second pair $(2,1)$, evaluate $\mathsf{D}^{(3)}[2,1]$ as: $\mathsf{D}^{(3)}[2,1] = \min\{\mathsf{D}^{(2)}[2,1], \mathsf{D}^{(2)}[2,3] + \mathsf{D}^{(2)}[3,2]\} = \min\{2, 7 + (-1)\} = 2$
  - Thus, we have no change for the second pair $(1,2)$

$$\mathsf{D}^{(3)} = \begin{bmatrix} 0 & 2 & 5 \\ 2 & 0 & 7 \\ -1 & -3 & 0 \end{bmatrix} \qquad \mathsf{P} = \begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 1 \\ 2 & 0 & 0 \end{bmatrix}$$

Recursive Path Reconstruction Example I

For the following linear graph:

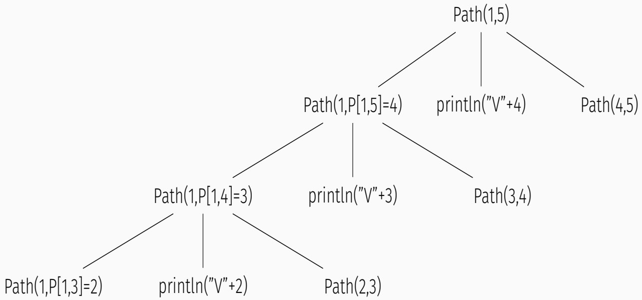


it is easy to deduce that:

$$D^{(5)} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 0 & 1 & 2 & 3 \\ 2 & 1 & 0 & 1 & 2 \\ 3 & 2 & 1 & 0 & 1 \\ 4 & 3 & 2 & 1 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & 2 & 3 & 4 \\ 0 & 0 & 0 & 3 & 4 \\ 2 & 0 & 0 & 0 & 4 \\ 2 & 3 & 0 & 0 & 0 \\ 2 & 3 & 4 & 0 & 0 \end{bmatrix}$$

Recursive Path Reconstruction Example II

Determining the intermediate nodes on the shortest path between nodes 1 and 5 results in the following recursive calling sequence of the *Path*() function:



Floyd-Warshall: Analysis I

It is easy to deduce that the Runtime Complexity of the Floyd-Warshall algorithm is $\Theta(n^3)$

Its Space Complexity is $\Theta(n^2)$

However, it is not obvious why the Floyd-Warshall algorithm can be implemented through the utilization of a single $n \times n$ distance matrix D.

- $\mathsf{D}[i, j]$ depends only on the elements in the k-th column and row.
- It is easy to show that the k-th row and the k-th column of the distance matrix remain unchanged when $\mathsf{D}^{(k)}$ is being computed.

Floyd-Warshall: Analysis II

Before showing that the k-th row and column of the distance matrix $\mathbf{D}$ remain unchanged, we will show that the elements along the main diagonal remain 0.

- $\mathbf{D}^{(\mathbf{k})}[j,j] = \min\{D^{(k-1)}[j,j], D^{(k-1)}[j,k] + D^{(k-1)}[k,j]\} \Leftrightarrow$
  $\mathbf{D}^{(\mathbf{k})}[j,j] = \min\{0, D^{(k-1)}[j,k] + D^{(k-1)}[k,j]\} \Leftrightarrow$
  $\mathbf{D}^{(\mathbf{k})}[j,j] = 0$
- This is true since we have assumed that there may exist negative edges but not cycles with cumulative weight negative.

Floyd-Warshall: Analysis III

The k-th column of $\mathbf{D^{(k)}}$ is equal to the k-th column of $\mathbf{D^{(k-1)}}$.

Intuitively true since a path from v_i to v_k will not become shorter by adding v_k to the allowed subset of intermediate nodes.

- $\forall i,\ \mathbf{D^{(k)}}[i,k] = \min\{D^{(k-1)}[i,k], D^{(k-1)}[i,k] + D^{(k-1)}[k,k]\} \Leftrightarrow$
  $\mathbf{D^{(k)}}[i,k] = \min\{D^{(k-1)}[i,k], D^{(k-1)}[i,k] + 0\} \Leftrightarrow$
  $\mathbf{D^{(k)}}[i,k] = D^{(k-1)}[i,k]$

Floyd-Warshall: Analysis IV

The k-th row of $\mathsf{D}^{(\mathsf{k})}$ is equal to the k-th row of $\mathsf{D}^{(\mathsf{k}-1)}$.

- $\forall j,\ \mathsf{D}^{(\mathsf{k})}[k,j] = \min\{D^{(k-1)}[k,j], D^{(k-1)}[k,k] + D^{(k-1)}[k,j]\} \Leftrightarrow$
  $\mathsf{D}^{(\mathsf{k})}[k,j] = \min\{D^{(k-1)}[k,j], 0 + D^{(k-1)}[k,j]\} \Leftrightarrow$
  $\mathsf{D}^{(\mathsf{k})}[k,j] = D^{(k-1)}[k,j]$

FloydWarshall.m I

```
function [D,P] = FloydWarshall(W)

% This function computes shortest paths' distances
% for each pair of nodes within the graph whose
% initial weight (adjacency) matrix is stored in
% matrix W. Matrix W is assumed to be properly
% initialized. Element D[i,j] of matrix D stores
% the shortest path distance from node i to node j.
% Matrix P is the corresponding predecessor matrix
% so that element P[i,j] stores the last vertex
% traversed within the shortest path connecting
% nodes i and j.
```

FloydWarshall.m II

```matlab

% Get the number of nodes pertaining to the graph.
nodes_num = size(W,1);

% Initialize internal matrix D.
D = W;

% Initialize internal matrix P.
P = zeros(nodes_num,nodes_num);

% Main Algorithm.
for k = 1:1:nodes_num
    for i = 1:1:nodes_num
```

FloydWarshall.m III

```
            for j = 1:1:nodes_num
                if( D(i,j) > D(i,k) + D(k,j))
                    D(i,j) = D(i,k) + D(k,j);
                    P(i,j) = k;
                end
            end
        end
end

end
```

ExtractShortestPaths.m I

```
function [Dtop,Ptop] = ExtractShortestPaths(Wo,Ctop
    )

% This function extracts the pairwise shortest
% paths matrices and corresponding path
% reconstruction indices matrices for each one of
% the top No connected components of the
% co-authorship network.

% Wo: is the initial binary connectivity matrix.
% Ctop: is a cell array storing the indices of
% the top No connected components stored in
```

ExtractShortestPaths.m II

```
% decreasing order of magnitude.

% Get the number of top connected components
% stored in Ctop.
Ntop = length(Ctop);

% Initialize cell array containers for variables
% Dtop and Ptop.
% Each element of Dtop stores the matrix of
% pairwise shortest distances.
% Each element of Ptop stores the
% corresponding matrix of predecessor
% indices that can be utilized to reconstruct the
```

ExtractShortestPaths.m III

```
% sequence of nodes in each shortest path.

% Loop through the various connected components
% stored in Ctop.
for component_index = 1:1:Ntop
    % Get the current component.
    component = Ctop{component_index};
    % Get the number of nodes pertaining to the
    % current component.
    Nc = length(component);
    % Get the corresponding sub-weight matrix for
    % the current component.
    Wc = Wo(component,component);
```

ExtractShortestPaths.m IV

```
% Set the diagonal indices for the current
% sub—weight matrix.
Idiag = [1:Nc+1:Nc*Nc];
% Get the indices of all zero elements of the
% current sub—weight matrix.
Izero = find(Wc==0);
% Get the indices of all zero elements in the
% current sub—weight matrix
% that do not reside on its main diagonal.
Izero_non_diagonal = setdiff(Izero,Idiag);
% Set all non—diagonal zero entries of the
% current sub—weight matrix to Inf.
Wc(Izero_non_diagonal) = Inf;
```

ExtractShortestPaths.m V

```
    % Run the Floyd-Warshall algorithm of the
    % current connected component.
    [Dc,Pc] = FloydWarshall(Wc);

    % The following line of code should be
    % uncommented if:
    % the predecessor indices stored in Pc should
    % be synchronized with the original author
    % indices in Wo excluding the zero values of Pc.
    % Pc(Pc~=0) = component(Pc(Pc~=0));

```

ExtractShortestPaths.m VI

```
    % Set the correspoding entries of Dtop and Ptop
        .
    Dtop{component_index} = Dc;
    Ptop{component_index} = Pc;
end

end
```

```
function [Path] = ReconstructPath(P,source_node,
    target_node)

% This function reconstructs the path between the
% given pair of (source_node,target_node)
% based on the predecessor matrix P.
% The predecessor matrix is assumed to be
% associated with a connected component of an
% underlying graph.

% Initially, construct the intermediate path
% between the source_node and the target_node.
```

ReconstructPath.m II

```
% (Mind that an intermediate path may not exist in
% case the source and target nodes are
% immediately connected).

% Initialize the intermediate path.
intermediate_path = [];

% Get the last intermediate node between the source
% and target nodes.
intermediate_node = P(source_node,target_node);

% While the intermediate node index is not (0)
% retrieve the full set of intermediate
```

```
% nodes in reverse order.
while(intermediate_node~=0)
    intermediate_path = [intermediate_path,
        intermediate_node];
    intermediate_node = P(source_node,
        intermediate_node);
end

% If the intermediate path is not empty reverse it.
if(~isempty(intermediate_path))
    intermediate_path = intermediate_path(end:-1:1)
        ;
end
```

ReconstructPath.m IV

```
% Construct the full path.
Path = [source_node,intermediate_path,target_node];

end
```

```
% Isolate the top No connected components.
Ctop = C(TopNComponentsIndices);

% Extract the pair-wise shortest paths and
% predecessor indices for each connected component.
[Dtop,Ptop] = ExtractShortestPaths(Wo,Ctop);

% Example: Report the shortest distance and
% corresponding path between nodes (1),(50) and
% (1),(97) in component 1.
component = Ctop{1};
P = Ptop{1};
```

```
D = Dtop{1};
source_node = 1
target_node = 50
sortest_distance = D(source_node,target_node)
Path = ReconstructPath(P,source_node,target_node)
source_node = 1
target_node = 97
sortest_distance = D(source_node,target_node)
Path = ReconstructPath(P,source_node,target_node)
```