COMP2521 24T1
Graphs (V)
Digraph Algorithms

Kevin Luxa
cs2521@cse.unsw.edu.au

digraph traversal
cycle checking
transitive closure
Reminder: **directed graphs** are graphs where...

- Each edge $(v, w)$ has a **source** $v$ and a **destination** $w$
- Unlike undirected graphs, $v \rightarrow w \neq w \rightarrow v$
<table>
<thead>
<tr>
<th></th>
<th>Domain</th>
<th>Vertex is...</th>
<th>Edge is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>WWW</td>
<td>article</td>
<td>hyperlink</td>
</tr>
<tr>
<td>target</td>
<td>program</td>
<td>function</td>
<td>board</td>
</tr>
<tr>
<td>dependency</td>
<td>scheduling</td>
<td>call</td>
<td>state</td>
</tr>
<tr>
<td></td>
<td>chess</td>
<td>task</td>
<td>legal</td>
</tr>
<tr>
<td></td>
<td>journals</td>
<td>function</td>
<td>move</td>
</tr>
<tr>
<td></td>
<td>scheduling</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Same as for undirected graphs:

**bfs**\((G, \text{src})\):

- initialise visited array
- mark \text{src} as visited
- enqueue \text{src} into \(Q\)
- while \(Q\) is not empty:
  - \(v = \text{dequeue from } Q\)
  - for each edge \((v, w)\) in \(G\):
    - if \(w\) has not been visited:
      - mark \(w\) as visited
      - enqueue \(w\) into \(Q\)

**dfs**\((G, \text{src})\):

- initialise visited array
- dfsRec\((G, \text{src, visited})\)

**dfsRec**\((G, v, \text{visited})\):

- mark \(v\) as visited
- for each edge \((v, w)\) in \(G\):
  - if \(w\) has not been visited:
    - dfsRec\((G, w, visited)\)
Web crawling
Visit a subset of the web...
...to index
...to cache locally

Which traversal method? BFS or DFS?

Note: we can’t use a visited array, as we don’t know how many webpages there are. Instead, use a visited set.
Web crawling algorithm:

```
webCrawl(startingUrl, maxPagesToVisit):
    create visited set
    add startingUrl to visited set
    enqueue startingUrl into Q

    numPagesVisited = 0
    while Q is not empty and numPagesVisited < maxPagesToVisit:
        currPage = dequeue from Q

        visit currPage
        numPagesVisited = numPagesVisited + 1

        for each hyperlink on currPage:
            if hyperlink not in visited set:
                add hyperlink to visited set
                enqueue hyperlink into Q
```
In directed graphs, a **cycle** is a directed path where the start vertex = end vertex

This graph has three distinct cycles:
0-4-0, 2-5-6-2, 3-3
Recall: Cycle checking for undirected graphs:

\[
\text{hasCycle}(G): \\
\quad \text{initialise visited array to false} \\
\quad \text{for each vertex } v \text{ in } G: \\
\quad \quad \text{if visited}[v] = \text{false}: \\
\quad \quad \quad \text{if dfsHasCycle}(G, v, v, \text{visited}): \\
\quad \quad \quad \quad \quad \quad \text{return} \; \text{true} \\
\quad \quad \text{return} \; \text{false} \\
\]

\[
\text{dfsHasCycle}(G, v, \text{prev}, \text{visited}): \\
\quad \text{visited}[v] = \text{true} \\
\quad \text{for each edge } (v, w) \text{ in } G: \\
\quad \quad \text{if } w = \text{prev}: \\
\quad \quad \quad \text{continue} \\
\quad \quad \quad \text{if visited}[w] = \text{true}: \\
\quad \quad \quad \quad \quad \quad \text{return} \; \text{true} \\
\quad \quad \text{else if } \text{dfsHasCycle}(G, w, v, \text{visited}): \\
\quad \quad \quad \text{return} \; \text{true} \\
\quad \text{return} \; \text{false} \\
\]

Does this work for directed graphs?
Recall: Cycle checking for undirected graphs:

\[ \text{hasCycle}(G): \]
initialise visited array to false
for each vertex \( v \) in \( G \):
  if \( \text{visited}[v] = \text{false} \):
    if \( \text{dfsHasCycle}(G, v, v, \text{visited}) \):
      return \text{true}
return \text{false}

\[ \text{dfsHasCycle}(G, v, prev, visited): \]
visited[\( v \)] = true
for each edge \((v, w)\) in \( G \):
  if \( w = \text{prev} \):
    continue
  if \( \text{visited}[w] = \text{true} \):
    return \text{true}
  else if \( \text{dfsHasCycle}(G, w, v, \text{visited}) \):
    return \text{true}
return \text{false}

Does this work for directed graphs? No
Problem #1

Algorithm ignores edge to previous vertex and therefore does not detect the following cycle:

Simple fix: Don’t ignore edge to previous vertex
Cycle Checking

Pseudocode

Example

Transitive Closure

Other Algorithms

Traversal

hasCycle(G):
    initialise visited array to false
    for each vertex v in G:
        if visited[v] = false:
            if dfsHasCycle(G, v, visited):
                return true

    return false

dfsHasCycle(G, v, visited):
    visited[v] = true

    for each edge (v, w) in G:
        if visited[w] = true:
            return true
        else if dfsHasCycle(G, w, visited):
            return true

    return false

Does this work for directed graphs?
hasCycle(\(G\)):
    initialise visited array to false
    for each vertex \(v\) in \(G\):
        if visited[\(v\)] = false:
            if dfsHasCycle(\(G\), \(v\), visited):
                return true
    return false

dfsHasCycle(\(G\), \(v\), visited):
    visited[\(v\)] = true
    for each edge \((v, w)\) in \(G\):
        if visited[\(w\)] = true:
            return true
        else if dfsHasCycle(\(G\), \(w\), visited):
            return true
    return false

Does this work for directed graphs?

No!
Problem #2

Algorithm can detect cycles when there is none, for example:

Algorithm starts at 0, recurses into 1 and 2, backtracks to 0, sees that 2 has been visited, and concludes there is a cycle
Consider a cycle check on this graph (starting at 0):

```
cycle(0, prev=0)
```

```
visited: [0] 1 0 0
```
Consider a cycle check on this graph (starting at 0):

- call stack
  - cycle(0, prev=0)

- visited
  - [0] [1] [2]
  - 1 0 0
Consider a cycle check on this graph (starting at 0):
Consider a cycle check on this graph (starting at 0):

Call stack

`cycle(1, prev=0)`

`cycle(0, prev=0)`

visited

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Cycle Checking
Consider a cycle check on this graph (starting at 0):

\[
\begin{array}{c}
0 \\
1 \\
2
\end{array}
\]

Cycle Checking

Pseudocode

Example

Transitive Closure

Other Algorithms

Call stack

\begin{array}{c}
\text{cycle(0, prev=0)} \\
\text{cycle(1, prev=0)} \\
\text{cycle(2, prev=1)}
\end{array}

Visited

\[
\begin{array}{c|c|c|c}
& 0 & 1 & 2 \\
\hline
0 & 1 & 1 & 1
\end{array}
\]
Consider a cycle check on this graph (starting at 0):

```
cycle(0, prev=0)
cycle(1, prev=0)
cycle(2, prev=1)
```

Call stack:
```
cycle(2, prev=1)
cycle(1, prev=0)
cycle(0, prev=0)
```
Idea:
To properly detect a cycle, check if neighbour is already on the call stack.

When the graph is undirected, this can be done by checking the visited array, but this doesn’t work for directed graphs!

Need to use separate array to keep track of when a vertex is on the call stack.
hasCycle($G$):
  create visited array, initialised to false
  create onStack array, initialised to false

  for each vertex $v$ in $G$:
    if visited[$v$] = false:
      if dfsHasCycle($G$, $v$, visited, onStack):
        return true

  return false

dfsHasCycle($G$, $v$, visited, onStack):
  visited[$v$] = true
  onStack[$v$] = true

  for each edge ($v$, $w$) in $G$:
    if onStack[$w$] = true:
      return true
    else if visited[$w$] = false:
      if dfsHasCycle($G$, $w$, visited, onStack):
        return true

  onStack[$v$] = false
  return false
Check if a cycle exists in this graph:
Problem: computing reachability

Given a digraph $G$ it is potentially useful to know:

- Is vertex $t$ reachable from vertex $s$?
One way to implement a reachability check:

- Use BFS or DFS starting at $s$
  - This is $O(V + E)$ in the worst case
  - Only feasible if reachability is an infrequent operation

What about applications that frequently need to check reachability?
Idea

Construct a $V \times V$ matrix that tells us whether there is a path \textit{(not edge)} from $s$ to $t$, for $s, t \in V$

This matrix is called the \textit{transitive closure (tc)} matrix (or reachability matrix)

\[ tc[s][t] \text{ is true if there is a path from } s \text{ to } t, \text{ false otherwise} \]
### Transitive Closure

#### Warshall's Algorithm

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Adjacency Matrix

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

#### Reachability Matrix

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```
One way to compute reachability matrix:

- Perform BFS/DFS from every vertex

Another way $\Rightarrow$ Warshall’s algorithm:

- Simple algorithm that does not require a graph traversal
Idea of Warshall’s algorithm:

- There is a path from \( s \) to \( t \) if:
  - There is an edge from \( s \) to \( t \), or
  - There is a path from \( s \) to \( t \) via vertex 0, or
  - There is a path from \( s \) to \( t \) via vertex 0 and/or 1, or
  - There is a path from \( s \) to \( t \) via vertex 0, 1 and/or 2, or
  - ...
  - There is a path from \( s \) to \( t \) via any of the other vertices
Example:

- There is a path from \( s \) to \( t \) if:
  - There is an edge from \( s \) to \( t \), or
  - There is a path from \( s \) to \( t \) via vertex 0, or
  - There is a path from \( s \) to \( t \) via vertex 0 and/or 1, or
  - There is a path from \( s \) to \( t \) via vertex 0, 1 and/or 2, or
  - There is a path from \( s \) to \( t \) via vertex 0, 1, 2 and/or 3
Example:
- There is a path from $s$ to $t$ if:
  - There is an edge from $s$ to $t$, or
  - There is a path from $s$ to $t$ via vertex 0, or
  - There is a path from $s$ to $t$ via vertex 0 and/or 1, or
  - There is a path from $s$ to $t$ via vertex 0, 1 and/or 2, or
  - There is a path from $s$ to $t$ via vertex 0, 1, 2 and/or 3

![Graph Diagram]
Example:

- There is a path from \( s \) to \( t \) if:
  - There is an edge from \( s \) to \( t \), or
  - There is a path from \( s \) to \( t \) via vertex 0, or
  - There is a path from \( s \) to \( t \) via vertex 0 and/or 1, or
  - There is a path from \( s \) to \( t \) via vertex 0, 1 and/or 2, or
  - There is a path from \( s \) to \( t \) via vertex 0, 1, 2 and/or 3
Example:

- There is a path from $s$ to $t$ if:
  - There is an edge from $s$ to $t$, or
  - There is a path from $s$ to $t$ via vertex 0, or
  - There is a path from $s$ to $t$ via vertex 0 and/or 1, or
  - There is a path from $s$ to $t$ via vertex 0, 1 and/or 2, or
  - There is a path from $s$ to $t$ via vertex 0, 1, 2 and/or 3
Warshall’s Algorithm

Example:

- There is a path from $s$ to $t$ if:
  - There is an edge from $s$ to $t$, or
  - There is a path from $s$ to $t$ via vertex 0, or
  - There is a path from $s$ to $t$ via vertex 0 and/or 1, or
  - There is a path from $s$ to $t$ via vertex 0, 1 and/or 2, or
  - There is a path from $s$ to $t$ via vertex 0, 1, 2 and/or 3
On the $k$-th iteration, the algorithm determines if a path exists between two vertices $s$ and $t$ using just $0, \ldots, k$ as intermediate vertices.

On the $k$-th iteration

If we have:
(1) a path from $s$ to $k$
(2) a path from $k$ to $t$
(using only vertices $0$ to $k - 1$)
Warshall’s Algorithm

On the \( k \)-th iteration, the algorithm determines if a path exists between two vertices \( s \) and \( t \) using just 0, \( \ldots \), \( k \) as intermediate vertices.

On the \( k \)-th iteration

If we have:

1. a path from \( s \) to \( k \)
2. a path from \( k \) to \( t \)

(using only vertices 0 to \( k - 1 \))

Then we have a path from \( s \) to \( t \) using vertices from 0 to \( k \)

\[
\text{if tc}[s][k] \text{ and tc}[k][t]:
\]
\[
tc[s][t] = \text{true}
\]
Warshall’s Algorithm

Pseudocode

warshall\((A)\):

**Input:** \( n \times n \) adjacency matrix \( A \)

**Output:** \( n \times n \) reachability matrix

create tc matrix which is a copy of \( A \)

\[
\begin{align*}
\text{for each vertex } k \text{ in } G: & // \text{ from 0 to } n - 1 \\
\text{ for each vertex } s \text{ in } G: & \\
\text{ for each vertex } t \text{ in } G: & \\
\text{ if } tc[s][k] \text{ and } tc[k][t]: & \\
& \text{tc}[s][t] = \text{true}
\end{align*}
\]

return tc
Find transitive closure of this graph

```
<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>[1]</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>[2]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[3]</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Warshall's Algorithm

Example

**Initialise tc with edges of original graph**

![Graph with nodes and edges](image)

```
<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>[1]</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>[2]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[3]</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Warshall’s Algorithm

Example

First iteration: \( k = 0 \)

![Graph](image)

```
Initialise tc with edges of original graph

First iteration: \( k = 0 \)

There is a path \( 1 \rightarrow 0 \) and a path \( 0 \rightarrow 2 \) so there is a path \( 1 \rightarrow 2 \)

Done

Second iteration: \( k = 1 \)

There is a path \( 3 \rightarrow 1 \) and a path \( 1 \rightarrow 0 \) so there is a path \( 3 \rightarrow 0 \)

There is a path \( 3 \rightarrow 1 \) and a path \( 1 \rightarrow 2 \) so there is a path \( 3 \rightarrow 2 \)

There is a path \( 3 \rightarrow 1 \) and a path \( 1 \rightarrow 3 \) so there is a path \( 3 \rightarrow 3 \)

Done

Third iteration: \( k = 2 \)

No pairs \( (s, t) \) such that there are paths \( s \rightarrow 2 \) and \( 2 \rightarrow t \)

Done

Fourth iteration: \( k = 3 \)

There is a path \( 1 \rightarrow 3 \) and a path \( 3 \rightarrow 1 \) so there is a path \( 1 \rightarrow 1 \)

Done

Finished
```
First iteration: $k = 0$

There is a path $1 \to 0$ and a path $0 \to 2$
First iteration: $k = 0$
There is a path $1 \rightarrow 0$ and a path $0 \rightarrow 2$
So there is a path $1 \rightarrow 2$
First iteration: \( k = 0 \)
Done
Warshall’s Algorithm

Example

Second iteration: $k = 1$

**Graph:**

- Vertices: 0, 1, 2, 3
- Edges: (0, 1), (1, 2), (1, 3), (3, 1)

**Matrix:**

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>[1]</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[2]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[3]</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Warshall’s Algorithm
Example

Second iteration: $k = 1$
There is a path $3 \rightarrow 1$ and a path $1 \rightarrow 0$
Second iteration: $k = 1$

There is a path $3 \rightarrow 1$ and a path $1 \rightarrow 0$

So there is a path $3 \rightarrow 0$
Warshall’s Algorithm
Example

Second iteration: $k = 1$
There is a path $3 \rightarrow 1$ and a path $1 \rightarrow 2$

![Graph of a directed graph with nodes 0, 1, 2, 3 and edges 0→1, 1→2, 2→3, 3→1, 3→2, 1→0.]

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>[1]</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[2]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[3]</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Warshall’s Algorithm

Example

Second iteration: \( k = 1 \)
There is a path \( 3 \rightarrow 1 \) and a path \( 1 \rightarrow 2 \)
So there is a path \( 3 \rightarrow 2 \)
Second iteration: $k = 1$

There is a path $3 \rightarrow 1$ and a path $1 \rightarrow 3$
Warshall’s Algorithm

Example

Second iteration: $k = 1$

There is a path $3 \to 1$ and a path $1 \to 3$

So there is a path $3 \to 3$
Warshall’s Algorithm

Example

Second iteration: \( k = 1 \)
Done

```
[0] [1] [2] [3]
[0] 0 0 1 0
[1] 1 0 1 1
[2] 0 0 0 0
[3] 1 1 1 1
```
Find transitive closure of this graph

Initialise tc with edges of original graph

First iteration:

$k = 0$

There is a path $1 \rightarrow 0$ and a path $0 \rightarrow 2$

So there is a path $1 \rightarrow 2$

Done

Second iteration:

$k = 1$

There is a path $3 \rightarrow 1$ and a path $1 \rightarrow 0$

So there is a path $3 \rightarrow 0$

There is a path $3 \rightarrow 1$ and a path $1 \rightarrow 2$

So there is a path $3 \rightarrow 2$

There is a path $3 \rightarrow 1$ and a path $1 \rightarrow 3$

So there is a path $3 \rightarrow 3$

Done

Third iteration:

$k = 2$

No pairs $(s, t)$ such that there are paths $s \rightarrow 2$ and $2 \rightarrow t$

Done

Fourth iteration:

$k = 3$

There is a path $1 \rightarrow 3$ and a path $3 \rightarrow 1$

So there is a path $1 \rightarrow 1$

Done

Finished
Warshall’s Algorithm

Example

Third iteration: \( k = 2 \)
No pairs \((s, t)\) such that there are paths \(s \rightarrow 2\) and \(2 \rightarrow t\)
Warshall’s Algorithm

Example

Third iteration: \( k = 2 \)

Done
Fourth iteration: $k = 3$
Fourth iteration: \( k = 3 \)
There is a path \( 1 \rightarrow 3 \) and a path \( 3 \rightarrow 1 \)
**Warshall’s Algorithm**

**Example**

Fourth iteration: $k = 3$

There is a path $1 \rightarrow 3$ and a path $3 \rightarrow 1$

So there is a path $1 \rightarrow 1$

### Pseudocode

1. Initialise $tc$ with edges of original graph
2. For $k = 0, 1, 2, \ldots$:
   - For each pair $(s, t)$:
     - If there is a path $s \rightarrow t$ and a path $t \rightarrow s$
       - Update $tc[s][t] = 1$

### Example

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>[1]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[2]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[3]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Analysis

- Time complexity: $O(n^3)$
- Space complexity: $O(n^2)$
Fourth iteration: $k = 3$

Done
Find transitive closure of this graph

1. Initialise tc with edges of original graph
2. First iteration:
   - $k = 0$
   - There is a path $1 \rightarrow 0$ and a path $0 \rightarrow 2$
   - So there is a path $1 \rightarrow 2$
   - Done
3. Second iteration:
   - $k = 1$
   - There is a path $3 \rightarrow 1$ and a path $1 \rightarrow 0$
   - There is a path $3 \rightarrow 1$ and a path $1 \rightarrow 2$
   - So there is a path $3 \rightarrow 2$
   - There is a path $3 \rightarrow 1$ and a path $1 \rightarrow 3$
   - So there is a path $3 \rightarrow 3$
   - Done
4. Third iteration:
   - $k = 2$
   - No pairs $(s, t)$ such that there are paths $s \rightarrow 2$ and $2 \rightarrow t$
   - Done
5. Fourth iteration:
   - $k = 3$
   - There is a path $1 \rightarrow 3$ and a path $3 \rightarrow 1$
   - So there is a path $1 \rightarrow 1$
   - Done

Finished
Analysis:

- **Time complexity:** $O(V^3)$
  - Three nested loops iterating over all vertices
- **Space complexity:** $O(V^2)$
  - Can be $O(1)$ if overwriting the input matrix
- **Benefit:** checking reachability between vertices is now $O(1)$
  - Makes up for slow setup ($O(V^3)$) if reachability is a very frequent operation
Strongly connected components:

- Kosaraju’s algorithm
- Tarjan’s algorithm
https://forms.office.com/r/5c0fb4tvMb