# COMP2521 24T3

## Graphs (V)
## Digraph Algorithms

Hao Xue

cs2521@cse.unsw.edu.au

digraph traversal
cycle checking
transitive closure

# Directed Graphs (Digraphs)



Reminder: directed graphs are graphs where...

- Each edge $(v, w)$ has a source $v$ and a destination $w$
- Unlike undirected graphs, $v \to w \neq w \to v$

# Digraph Traversal

Same as for undirected graphs:

```
bfs(G, src):
    initialise visited array
    mark src as visited
    enqueue src into Q
    while Q is not empty:
        v = 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, src):
    initialise visited array
    dfsRec(G, src, visited)

dfsRec(G, v, 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?

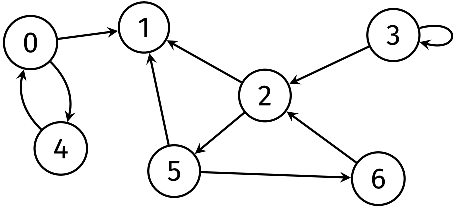
Note: we can't use a visited array, as we don't know how many webpages there are. Instead, use a visited set.

## 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
```

# Cycle Checking

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

# Cycle Checking

**Recall:** Cycle checking for undirected graphs:

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

    return false

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

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

    return false
```

Does this work for directed graphs?

# Cycle Checking

**Recall:** Cycle checking for undirected graphs:

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

    return false

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

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

    return false
```

Does this work for directed graphs?

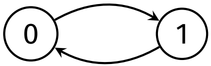
No

# Cycle Checking



## 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

```
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?

# Cycle Checking

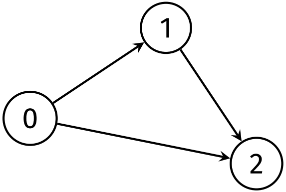
```
hasCycle(G):
    initialise visited array to false
    for each vertex v in G:
        if visited[v] = false:
            if dfsHasCycle(G, v, visited):
                return true

    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

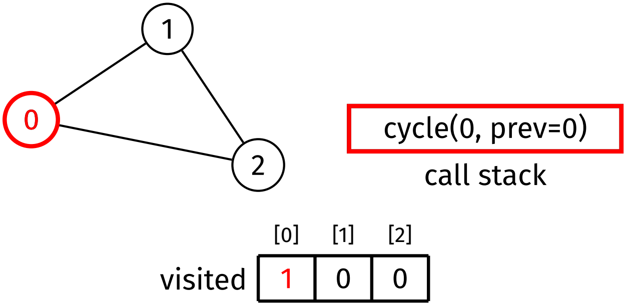
Algorithm can detect cycles when there is none, for example:




Algorithm starts at 0, recurses into 1 and 2, backtracks to 0, sees that 2 has been visited, and concludes there is a cycle

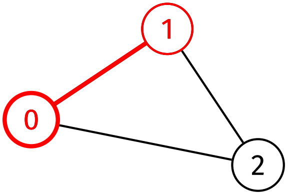
# Cycle Checking

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



# Cycle Checking

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




cycle(0, prev=0)

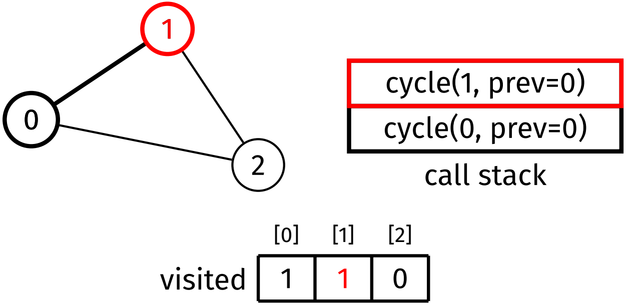
call stack

visited

| [0] | [1] | [2] |
|---|---|---|
| 1 | 0 | 0 |

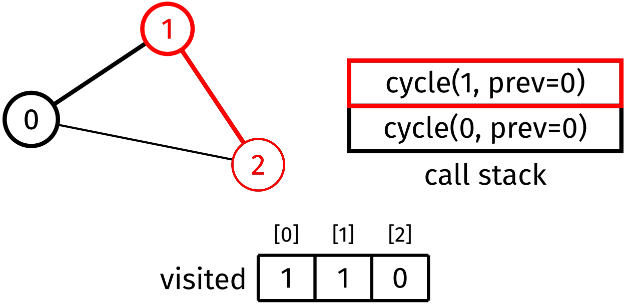
# Cycle Checking

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



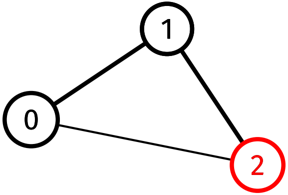
# Cycle Checking

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



# Cycle Checking

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



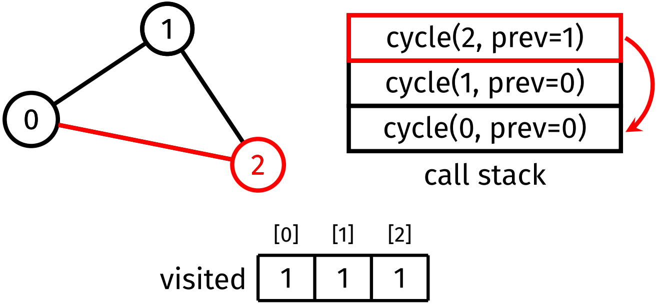

| call stack |
| --- |
| cycle(2, prev=1) |
| cycle(1, prev=0) |
| cycle(0, prev=0) |

call stack

| | [0] | [1] | [2] |
| --- | --- | --- | --- |
| visited | 1 | 1 | 1 |

# Cycle Checking

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



# Cycle Checking

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!

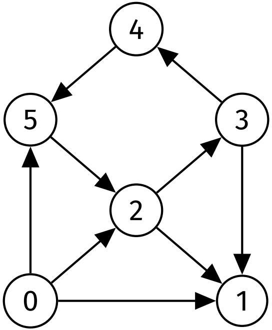
Need to use separate array to
keep track of when a vertex is on the call stack

# Cycle Checking

## Pseudocode

```
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:



# Transitive Closure

Problem: computing reachability

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

- Is vertex $t$ reachable from vertex $s$?

# Transitive Closure

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?

# Transitive Closure

## Idea

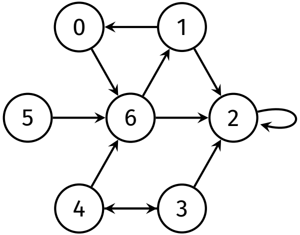
Construct a $V \times V$ matrix
that tells us whether there is a path (not edge)
from $s$ to $t$, for $s, t \in V$

This matrix is called the transitive closure (tc) matrix
(or reachability matrix)

`tc`$[s][t]$ is true if there is a path from $s$ to $t$, false otherwise

# Transitive Closure






| | [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|---|
| [0] | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| [1] | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| [2] | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| [3] | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| [4] | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| [5] | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| [6] | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

adjacency matrix

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|---|
| [0] | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| [1] | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| [2] | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| [4] | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| [5] | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| [6] | 1 | 1 | 1 | 0 | 0 | 0 | 1 |

reachability matrix

# Transitive Closure

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

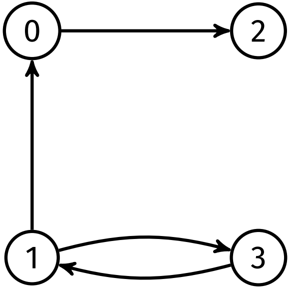
# Warshall's Algorithm

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:

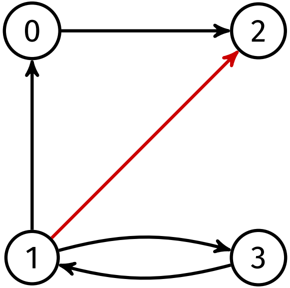
- There is a path from $s$ to $t$ if:
  - There is an edge from $s$ to $t$, or



# Warshall's Algorithm

Example:

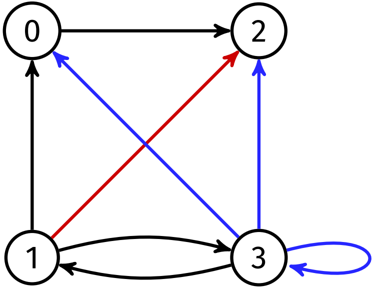
- There is a path from $s$ to $t$ if:
  - There is an edge from $s$ to $t$, or
  - There is a path from $s$ to $t$ via vertex 0, or



# Warshall's Algorithm

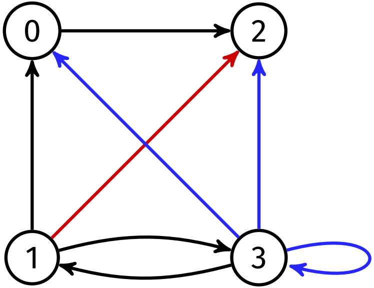
Example:

- 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



Example:

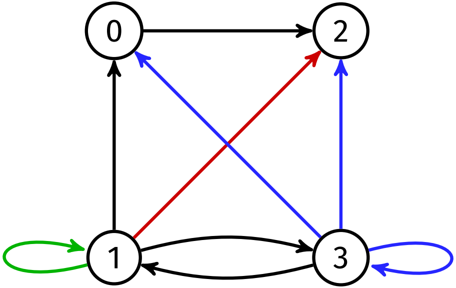
- There is a path from $s$ to $t$ if:
  - There is an edge from $s$ to $t$, or
  - There is a path from $s$ to $t$ via vertex 0, or
  - There is a path from $s$ to $t$ via vertex 0 and/or 1, or
  - There is a path from $s$ to $t$ via vertex 0, 1 and/or 2, or



# Warshall's Algorithm

Example:

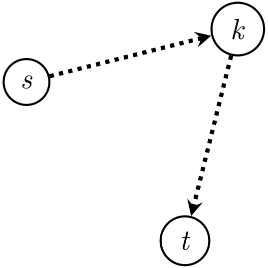
- There is a path from $s$ to $t$ if:
  - There is an edge from $s$ to $t$, or
  - There is a path from $s$ to $t$ via vertex 0, or
  - There is a path from $s$ to $t$ via vertex 0 and/or 1, or
  - There is a path from $s$ to $t$ via vertex 0, 1 and/or 2, or
  - There is a path from $s$ to $t$ via vertex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



# Warshall's Algorithm



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



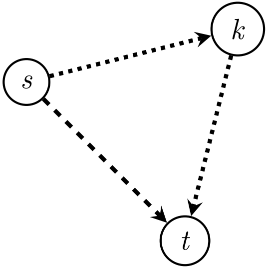

On the $k$-th iteration

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, ... , $k$ 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 $k$

```
if tc[s][k] and tc[k][t]:
    tc[s][t] = true
```

# Warshall's Algorithm

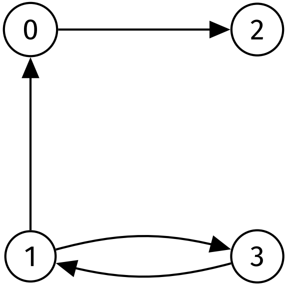
## Pseudocode

```
warshall(A):
    Input:  n × n adjacency matrix A
    Output: n × n reachability matrix

    create tc matrix which is a copy of A

    for each vertex k in G: // from 0 to n - 1
        for each vertex s in G:
            for each vertex t in G:
                if tc[s][k] and tc[k][t]:
                    tc[s][t] = true

    return tc
```

# Warshall's Algorithm

## Example

Find transitive closure of this graph



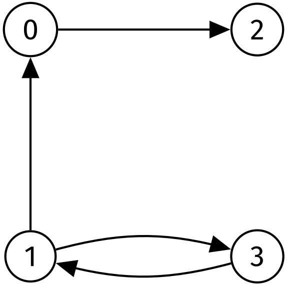

| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 0 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 0 | 1 | 0 | 0 |

# Warshall's Algorithm

## Example



Initialise tc with edges of original graph

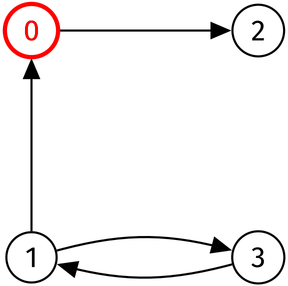



| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 0 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 0 | 1 | 0 | 0 |

# Warshall's Algorithm

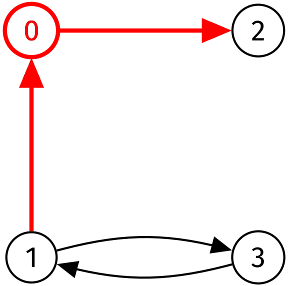
## Example

First iteration: $k = 0$




| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 0 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 0 | 1 | 0 | 0 |

First iteration: $k = 0$
There is a path $1 \rightarrow 0$ and a path $0 \rightarrow 2$

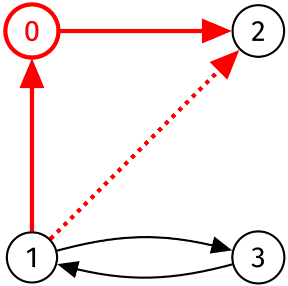



| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 0 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 0 | 1 | 0 | 0 |

First iteration: $k = 0$

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

So there is a path $1 \to 2$

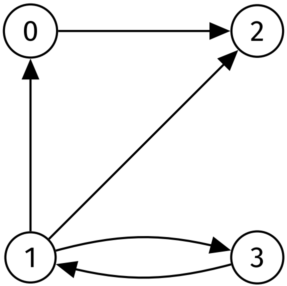



| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 0 | 1 | 0 | 0 |

# Warshall's Algorithm

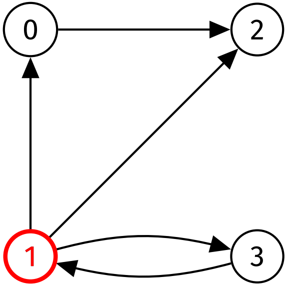
## Example

First iteration: $k = 0$
Done



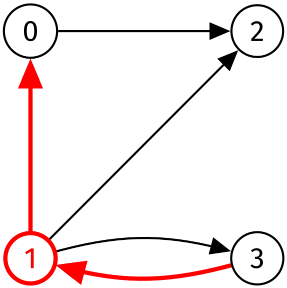

| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 0 | 1 | 0 | 0 |

Second iteration: $k = 1$




| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 0 | 1 | 0 | 0 |

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




| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 0 | 1 | 0 | 0 |

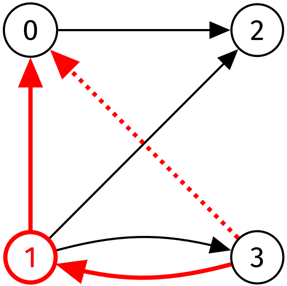
# Warshall's Algorithm

## Example

Second iteration: $k = 1$

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

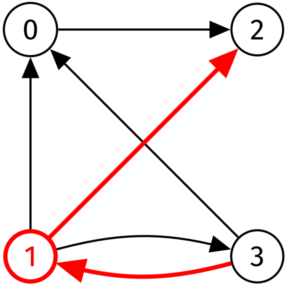
So there is a path $3 \to 0$




| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 0 | 0 |

Second iteration: $k = 1$

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




| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 0 | 0 |

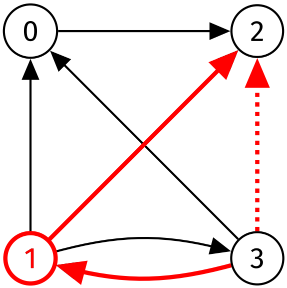
# Warshall's Algorithm

## Example

Second iteration: $k = 1$

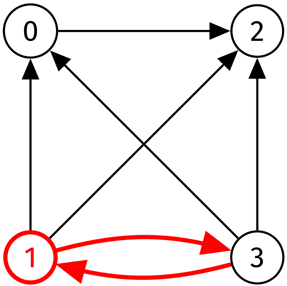
There is a path $3 \to 1$ and a path $1 \to 2$

So there is a path $3 \to 2$




| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 1 | 0 |

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




| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 1 | 0 |

# Warshall's Algorithm

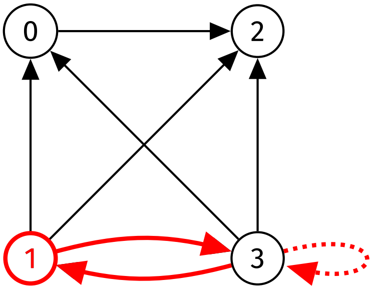
## Example



Second iteration: $k = 1$

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

So there is a path $3 \to 3$



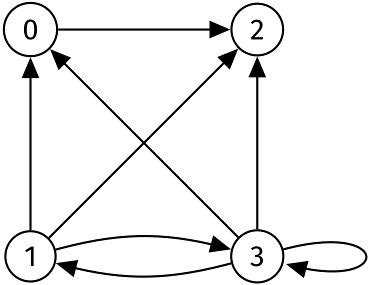

| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 1 | 1 |

# Warshall's Algorithm

## Example

Second iteration: $k = 1$

Done



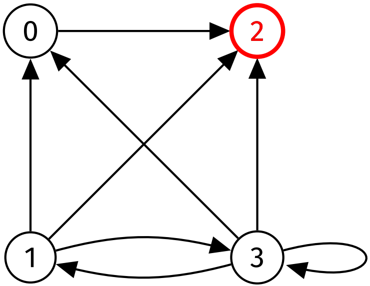

| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 1 | 1 |

# Warshall's Algorithm

## Example



Third iteration: $k = 2$

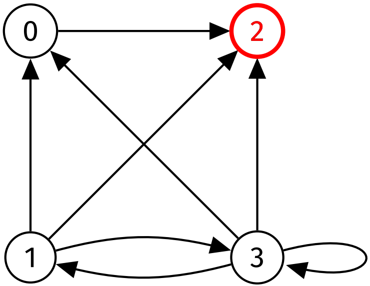



| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 1 | 1 |

# Warshall's Algorithm
## Example



Third iteration: $k = 2$
No pairs $(s, t)$ such that there are paths $s \to 2$ and $2 \to t$

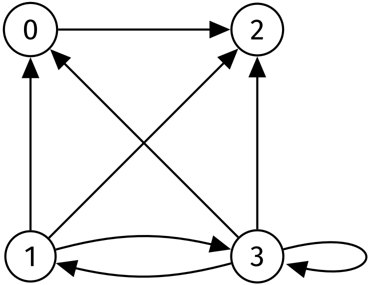



| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 1 | 1 |

# Warshall's Algorithm

## Example

Third iteration: $k = 2$
Done

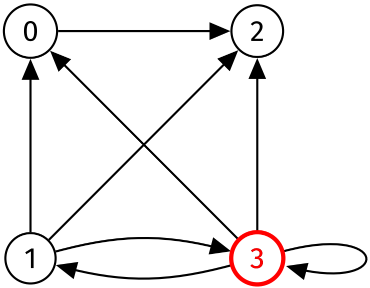



| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 1 | 1 |

# Warshall's Algorithm

## Example

Fourth iteration: $k = 3$




| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 1 | 1 |

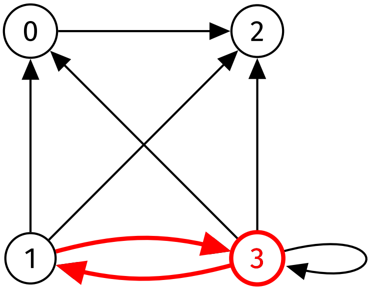
# Warshall's Algorithm

## Example



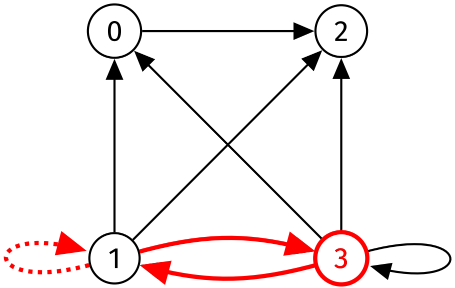
Fourth iteration: $k = 3$

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




| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 1 | 1 |

Fourth iteration: $k = 3$
There is a path $1 \to 3$ and a path $3 \to 1$
So there is a path $1 \to 1$




| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 1 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 1 | 1 |

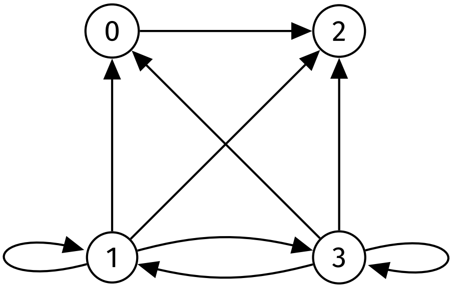
# Warshall's Algorithm

## Example



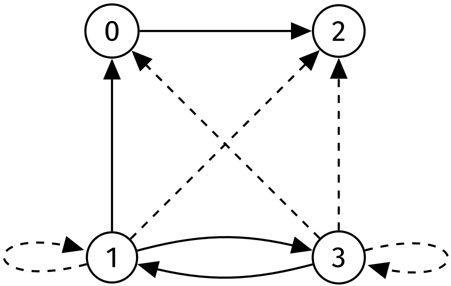
Fourth iteration: $k = 3$

Done




| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 1 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 1 | 1 |

Finished




| | [0] | [1] | [2] | [3] |
|---|---|---|---|---|
| [0] | 0 | 0 | 1 | 0 |
| [1] | 1 | 1 | 1 | 1 |
| [2] | 0 | 0 | 0 | 0 |
| [3] | 1 | 1 | 1 | 1 |

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

# Other Algorithms

Strongly connected components:

- Kosaraju's algorithm
- Tarjan's algorithm

# Feedback

https://forms.office.com/r/zEqxUXvmLR