COMP2521 23T3
Digraph Algorithms

Kevin Luxa
cs2521@cse.unsw.edu.au

digraph traversal
cycle checking
warshall’s algorithm
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>domain</th>
<th>vertex is...</th>
<th>edge is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>WWW</td>
<td>web page</td>
<td>hyperlink</td>
</tr>
<tr>
<td>chess</td>
<td>board state</td>
<td>legal move</td>
</tr>
<tr>
<td>scheduling</td>
<td>task</td>
<td>precedence</td>
</tr>
<tr>
<td>program</td>
<td>function</td>
<td>function call</td>
</tr>
<tr>
<td>journals</td>
<td>article</td>
<td>citation</td>
</tr>
<tr>
<td>make</td>
<td>target</td>
<td>dependency</td>
</tr>
</tbody>
</table>
Same as for undirected graphs:

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

\[
\text{dfs}(G, \ src): \\
\text{initialise visited array} \\
\text{dfsRec}(G, \ src, \ visited) \\
\text{dfsRec}(G, \ v, \ visited): \\
\quad \text{mark} \ v \ \text{as visited} \\
\quad \text{for each edge} \ (v, w) \ \text{in} \ G: \\
\quad \quad \text{if} \ w \ \text{has not been visited}: \\
\quad \quad \quad \text{dfsRec}(G, \ w, \ visited) \\
\]
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 $Q$
Wiki Game

Given two Wikipedia articles, navigate from the first article to the second in as few clicks as possible.
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:

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?
Recall: Cycle checking for undirected graphs:

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

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

Does this work for directed graphs? No
Problem #1

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

0 → 1

Simple fix: Don’t ignore edge to previous vertex
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?
Pseudocode

Example

Transitive Closure

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)**
- **call stack**
- **visited**

```
[0] [1] [2]
```

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

```
0
/|
/ |
1 - 2
```

- call stack: `cycle(0, prev=0)`
- visited array: `[1, 0, 0]`
Consider a cycle check on this graph (starting at 0):

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

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

```
call stack
```

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

```
visited
```

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Consider a cycle check on this graph (starting at 0):

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

visited: 1 1 1

cycle(0, prev=0)
cycle(1, prev=0)
cycle(2, prev=1)
call stack
Consider a cycle check on this graph (starting at 0):

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

![Call stack diagram]

```
visited  | 0 | 1 | 2 |
---------|---|---|---|
         | 1 | 1 | 1 |
```
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 (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

Traversals
Cycle Checking
Transitive Closure
Warshall's algorithm

adjacency matrix

reachability matrix
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
Main idea:

- 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
Warshall’s Algorithm

Main idea:
• 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
Main idea:
- 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 any of the other vertices
Warshall’s Algorithm

Main idea:
- 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

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
\end{array}
\]
Main idea:

- 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
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 $k - 1$)
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($A$):

**Inputs:** $n \times n$ adjacency matrix $A$

**Output:** $n \times n$ reachability matrix

create tc matrix, initialised to false

for each vertex $s$ in $G$:
    for each vertex $t$ in $G$:
        if $A[s][t]$:
            tc[$s$][$t$] = true

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
Find transitive closure of this graph

Warshall's Algorithm
Example

Initialise tc with edges of original graph

First iteration:
k = 0
There is a path 1 → 0 and a path 0 → 2
So there is a path 1 → 2
Done

Second iteration:
k = 1
There is a path 3 → 1 and a path 1 → 0
There is a path 3 → 1 and a path 1 → 2
There is a path 3 → 1 and a path 1 → 3
So there is a path 3 → 3
Done

Third iteration:
k = 2
No pairs (s, t) such that there are paths s → 2 and 2 → t
Done

Fourth iteration:
k = 3
There is a path 1 → 3 and a path 3 → 1
So there is a path 1 → 1
Done

Finished
Warshall’s Algorithm

Example

Initialise tc with edges of original graph

```
Initialise tc with edges of original 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>
```
First iteration: \( k = 0 \)
First iteration: $k = 0$
There is a path $1 \rightarrow 0$ and a path $0 \rightarrow 2$
Warshall’s Algorithm

Example

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

Example

First iteration: $k = 0$

Done

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>1</td>
<td>0</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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Warshall's Algorithm

Example

Second iteration: \( k = 1 \)

- 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

Second iteration: \( k = 1 \)
There is a path \( 3 \rightarrow 1 \) and a path \( 1 \rightarrow 0 \)
Traversals, Cycles, and Checking Transitive Closure

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 \( 3 \to 0 \)

![Graph diagram and Warshall's Algorithm 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>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 $1 \rightarrow 2$
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 $3 \to 1$ and a path $1 \to 3$
Second iteration:  $k = 1$
There is a path $3 \to 1$ and a path $1 \to 3$
So there is a path $3 \to 3$
Second iteration: $k = 1$
Done
Warshall’s Algorithm

Example

Third iteration: $k = 2$
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

Warshall’s Algorithm

Example

Find transitive closure of this graph

Initialise tc with edges of original graph

First iteration:
\( k = 0 \)

There is a path \( 1 \to 0 \) and a path \( 0 \to 2 \)

So there is a path \( 1 \to 2 \)

Done

Second iteration:
\( k = 1 \)

There is a path \( 3 \to 1 \) and a path \( 1 \to 0 \)

So there is a path \( 3 \to 0 \)

There is a path \( 3 \to 1 \) and a path \( 1 \to 2 \)

So there is a path \( 3 \to 2 \)

There is a path \( 3 \to 1 \) and a path \( 1 \to 3 \)

So there is a path \( 3 \to 3 \)

Done

Third iteration:
\( k = 2 \)

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

Done

Fourth iteration:
\( k = 3 \)

There is a path \( 1 \to 3 \) and a path \( 3 \to 1 \)

So there is a path \( 1 \to 1 \)

Done

Finished
Traversing Cycle Checking

Transitive Closure
Warshall’s algorithm
Pseudocode
Example

Analysis

Warshall’s Algorithm

Example

Fourth iteration: \( k = 3 \)

![Diagram of a graph with nodes 0, 1, 2, 3 and a transition 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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Fourth iteration: \( k = 3 \)
There is a path \( 1 \rightarrow 3 \) and a path \( 3 \rightarrow 1 \)
Fourth iteration: $k = 3$

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

So there is a path $1 \rightarrow 1$
Warshall’s Algorithm

Example

Fourth iteration: \( k = 3 \)
Done

![Graph](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>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>
Warshall’s Algorithm Example

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
Analysis:

- **Time complexity:** $O(V^3)$
  - Three nested loops iterating over all vertices
- **Space complexity:** $O(V^2)$
  - Reachability matrix is $V \times V$
- **Benefit:** checking reachability between vertices is now $O(1)$
  - Makes up for slow setup ($O(V^3)$) if reachability is a very frequent operation
https://forms.office.com/r/aPF09YHZ3X