Course 141

- Reasoning about Programs -

Warshall's Algorithm

- Start with some mathematical insight
- Clever choice of invariant and variant converts this to a clever algorithm
- Without going through this conversion the algorithm is incomprehensible.

THE PROBLEM

Given a directed graph, find which nodes (vertices) are connected by paths through the graph. E.g.:

```
  a ---- c ---- d
   \    /     / \\
    b   c     d
```

Paths: a to d, but not a to b.

Equivalently: given a relation R, compute its transitive closure. i.e. add in all arcs that would be necessary to make R transitive.

Applications:
- Given direct air flights, what journeys are possible without leaving planes or airports?
- Knowing which procedures call which others, which are potentially recursive?
- Can give edges a cost and use to find minimum cost paths.

OUTLINE

Problem is to find which nodes in a graph are connected by a path

We'll look at 3 algorithms, each an improvement on the previous one

The best is called Warshall’s Algorithm

We'll apply the algorithm to

Shortest Path problem
Best Path problem

All the algorithms will compute the transitive closure of a relation

REVIEWS OF MATHEMATICS

Each of the following carries the same information for a set of SIZE nodes and directed edges:

- a directed graph
- a binary relation on the set of nodes
- a SIZE × SIZE edge incidence matrix with Boolean entries:
  true = edge, false = no edge.

This relation tells us where the edges are.

Its transitive closure is another relation, telling us where there are paths. It too has an incidence matrix, the path incidence matrix.

Computing paths in a graph ⇔ computing the transitive closure of the relation represented by the graph ⇔ what we want.
EXAMPLE (SIZE = 4)

Directed Graph:

```

```

Edge Incidence Matrix:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>b</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>c</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>d</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

A graph relation edge(x,y) is transitive if, for any nodes x,y,z, edge(x,y) ∧ edge(y,z) ⇒ edge(x,z)

COMPUTER REPRESENTATION

We use incidence matrices to represent graphs.

```java
final int SIZE = ... // number of nodes

void transClos(boolean[][] edge, boolean[][] path) {
// Pre: edge and path are matrices of SIZE x SIZE ∧ SIZE≥1
// Post: edge=edge0 ∧ path.length=SIZE ∧
//      path represents transitive closure of edge
}
```

You can wrap the method inside a class called Graphs and can give the nodes names and provide proper print methods, etc.

You could alternatively declare transClos to return boolean [][] and set the result to the transitive closure of edge.

EXAMPLE (CONT)

Exercise: For initial directed graph below add arcs in stages to obtain transitive closure:

```

```

The transitive closure of a graph relation edge is the smallest superset of edge that is transitive.

Transitive Closure Incidence Matrix:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>b</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>c</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>d</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

This incidence matrix is what we want to compute.

REPRESENTATION (CONTINUED)

```java
edge[i][j] = true
⇒ graph has an edge from node i to node j
⇒ i related to j
```

On return, path[i][j] = true
⇒ graph has a path from node i to node j
⇒ i related to j in transitive closure

The length of a path in the graph is the number of edges in the path.

We’ll also use the following notation:

```
i →* j means there is a path from i to j, and
i →* n j means there is a path of length n from i to j
```

Postcondition is then

```
path is SIZE x SIZE matrix ∧ edge=edge0 ∧
∀i,j:[0..SIZE-1](path[i][j] = true ↔ i →* j)
```

(We’ll assume path is SIZE x SIZE matrix throughout)
PROOF IDEA 1 (NOT YET WARSHALL’S ALGORITHM!)

- The matrix \( \text{edge} \) records paths of length 1 (called edges).
- Build up matrix \( \text{path} \) by starting with \( \text{edge} \), and recording longer and longer paths — say up to length \( n \) at any given stage.

**Loop Invariant (informal):** "\( 1 \leq n \) \& all paths of length \( \leq n \) are recorded in \( \text{path} \) \& only paths are recorded in \( \text{path} \) \& \( \text{edge} = \text{edge0} \)"

- To make progress, must consider and record paths of length \( n+1 \). Every one is a shorter path (of length \( n \)) followed by an edge:

\[
i \overset{k}{\longrightarrow} j
\]

That a path from \( i \) to \( k \) exists is recorded in \( \text{path} \) \( \text{path}[i][k] = \text{true} \) and the edge from \( k \) to \( j \) is in \( \text{edge} \) \( \text{edge}[k][j] = \text{true} \)

- Therefore, for each \( i \) and \( j \), look at all possible \( k \); can improve \( \text{path}[i][j] \) to \text{true} if for some \( k \) we find \( \text{path}[i][k] = \text{edge}[k][j] = \text{true} \).

We’ve now indicated how to make progress and increase \( n \) to \( n+1 \).

MORE ABOUT THE INVARIANT

**Loop Invariant (first attempt):**
"\( 1 \leq n \) and all paths of length \( \leq n \) have been recorded in \( \text{path} \) and only paths are recorded in \( \text{path} \) and \( \text{edge} = \text{edge0} \)"

We won’t change \( \text{edge} \) anywhere so \( \text{edge} = \text{edge0} \) is true and we’ll ignore that part in what follows.

The third conjunct is necessary to be sure that \( \text{path} \) has not been set to true incorrectly.

We can write this Invariant more formally as:

(1) \( 1 \leq n \)

(12) \( \forall \text{i},\text{j}:[0..\text{SIZE}-1] \) (if \( \exists \text{m}( \text{i} \rightarrow \text{m} \land \text{m} \leq \text{n}) \) then \( \text{path}[\text{i}][\text{j}] = \text{true} \) \( \text{for each i, j} \)

(13) \( \forall \text{i},\text{j}:[0..\text{SIZE}-1] \) (if \( \text{path}[\text{i}][\text{j}] = \text{true} \) then \( \text{i} \rightarrow \text{j} \))

MORE ABOUT THE VARIANT

**Question:** When should we stop increasing \( n \)?

Whenever we stop the outer while loop we know from (12) that if any path of length \( \leq n \) exists between \( i \) and \( j \) then \( \text{path}[i][j] = \text{true} \)

(12) \( \forall \text{i},\text{j}:[0..\text{SIZE}-1] \) (if \( \exists \text{m}( \text{i} \rightarrow \text{m} \land \text{m} \leq \text{n}) \) then \( \text{path}[\text{i}][\text{j}] = \text{true} \) \( \text{for each} \text{ i, j} \)

What we would like is that \( \text{path}[\text{i}][\text{j}] = \text{true} \) if any path at all exists between \( i \) and \( j \), not just paths of length \( \leq n \); ie

(Final) \( \forall \text{i},\text{j}:[0..\text{SIZE}-1] \) (if \( \exists \text{m}( \text{i} \rightarrow \text{m} \land \text{m} \leq \text{n}) \) then \( \text{path}[\text{i}][\text{j}] = \text{true} \))

Is there a value for \( n \) that will allow us to deduce (Final) from (12)?

There is! We can stop when \( n = \text{SIZE} \).
MORE ABOUT THE INVARIANT AND VARIANT

Observation: If there's a path of length $L > \text{SIZE}$ between $i$ and $j$ it must have a loop, which we can cut out leaving a shorter path.

Then we can omit the section between ‘a’ and ‘a’ or between ‘i’ and ‘i’ and get a shorter path from $i$ to $j$:

Imagine we continue to do this until we get a path of length $\leq \text{SIZE}$, which we know about by (I2). Therefore we can take

**Loop Variant** = $\text{SIZE} - n$, which decreases as $n$ increases.

**Loop Invariant (final version):**

(I1) $1 \leq n \leq \text{SIZE}$

(I2) $\forall i, j [0..\text{SIZE} - 1]$ (if $\exists m (i \rightarrow^n j \land 1 \leq m \leq n)$ then $\text{path}[i][j] = \text{true}$)

(I3) $\forall i, j [0..\text{SIZE} - 1]$ (if $\text{path}[i][j] = \text{true}$ then $i \rightarrow j$)

PROOF THAT THE VARIANT IS "GOOD"

- We show: $\text{variant} = 0 \rightarrow \forall i, j [0..\text{SIZE} - 1] (i \rightarrow j \rightarrow \text{path}[i][j] = \text{true})$

- Assume that the $\text{variant} = 0$. By (I2)

  $\forall i, j [0..\text{SIZE} - 1]((E \exists m (i \rightarrow^n j \land 1 \leq m \leq n) \rightarrow \text{path}[i][j] = \text{true})$

- Suppose for contradiction that for some $i$ and $j$ a path exists between $i$ and $j$ but $\text{path}[i][j] = \text{false}$. Consider the shortest such path:

  - If the path has length $\leq \text{SIZE}$ it contradicts (*)

  - If the path's length $\geq \text{SIZE} + 1$ ($\geq \text{SIZE} + 2$ nodes) it visits a node twice:

    Then we can omit the section between ‘a’ and ‘a’ or between ‘i’ and ‘i’ and get a shorter path from $i$ to $j$:

- This contradicts the assumption as there is a shorter path between $i$ and $j$ than the one initially considered.

- So we conclude that by considering paths of length $\leq \text{SIZE}$ all paths are recorded. Therefore, can stop when $\text{SIZE} - n = 0$.

FINALISATION AND THE VARIANT

We suppose the loop stops when $n = \text{SIZE}$

By the invariant (I2), $\text{path}[i][j]$ will record whether a path of length $\leq \text{SIZE}$ exists between nodes $i$ and $j$:

$\forall i, j [0..\text{SIZE} - 1]$ (if $\exists m (1 \leq m \leq \text{SIZE} \land i \rightarrow^n j)$ then $\text{path}[i][j] = \text{true}$)

We argued informally that this implies

$\forall i, j [0..\text{SIZE} - 1]$ (if $\exists m (1 \leq m \leq \text{SIZE} \land i \rightarrow^n j)$ then $\text{path}[i][j] = \text{true}$)

(i.e. if there is a path of any length then $\text{path}[i][j] = \text{true}$)

By (I3) of Inv. already know

$\forall i, j [0..\text{SIZE} - 1] (\text{path}[i][j] = \text{true} \rightarrow i \rightarrow j)$

If also $\forall i, j [0..\text{SIZE} - 1] (i \rightarrow j \rightarrow \text{path}[i][j] = \text{true})$

Then $\forall i, j [0..\text{SIZE} - 1] (i \rightarrow j \rightarrow \text{path}[i][j] = \text{true})$

and finalisation only needs to return path as result.

GOODNESS OF THE VARIANT

As an example of the argument on slide 12, let $\text{SIZE} = 3$. Then the shortest path between $i$ and $j$ that might not be recorded in $\text{path}[i][j]$ must have length $\geq 4$, i.e. has $\geq 4$ arcs. Suppose it does have 4 arcs.

How many nodes must it have?

Such a path looks like:

\[
\begin{array}{ccccccc}
  & n_2 & n_3 & n_1 & \ldots & j \\
1 & n_1 & n_2 & n_3 & \ldots & j
\end{array}
\]

Now, there are 5 nodes in this path. Even if $i = j$, there are still 4 nodes. But $\text{SIZE} = 3$, so we only have 3 different nodes to choose from. Therefore, either $n_1$, $n_2$ and $n_3$ are all different and $i$ and $j$ are equal to one of $n_1$, $n_2$ or $n_3$, or $n_1$, $n_2$ and $n_3$ are not all different.

Either way, at least one node is duplicated (as shown on slide 13) and a shorter path can be found.

In the case when $\text{SIZE} = 3$ and $n_1$, $n_2$ and $n_3$ are all different, suppose $i = j = n_2$, then a shorter path is $n_2 \rightarrow n_3 \rightarrow j$.

If it is that $n_1$, $n_2$ and $n_3$ are not all different (let's suppose that $n_2 = n_3$), then the shorter path is $i \rightarrow n_1 \rightarrow n_2 \rightarrow j$. 

March 6, 2009

Warshall and Floyd Algorithms page 16
INITIALISATION

When \( n = 1 \), the Loop Invariant says: “\( 1 \leq i \leq \text{SIZE} \land \) all paths of length 1 are recorded in \( \text{path} \land \) only paths are recorded”

So we can initially establish the Invariant by copying \( \text{edge} \) to \( \text{path} \):
(all paths of length 1 (ie. all \( \text{edges} \) are in \( \text{path} \land \) each edge is a path)

In Java:

```java
for (int i = 0; i < \text{SIZE}; i++)
    for (int j = 0; j < \text{SIZE}; j++) \text{path}[i][j] = \text{edge}[i][j];
```

or

```java
boolean [][] \text{path} = new boolean [\text{SIZE}][\text{SIZE}]
    for (int i = 0; i < \text{SIZE}; i++) \text{path}[i] = \text{edge}[i].clone();
```

//makes a copy of each row of edge

Only useful in case \( \text{transClo} \) \( \text{returns} \) \text{path} \ as the result. Why?

RE-ESTABLISHING INVARIANT

A Subcontract:

// Invariant: \( 1 \leq n \leq \text{SIZE} \land \)
// \( \forall i,j:1..\text{SIZE} \) (if \( 3m(i \rightarrow i \wedge 1 \leq m \leq n) \) then \( \text{path}[i][j] = \text{true} \))
// \( \land \forall i,j:1..\text{SIZE} \) (if \( \text{path}[i][j] = \text{true} \) then \( i \rightarrow j \))
// Continuing while loop: \( n < \text{SIZE} \)
// \( n++ \);
//can increment n here or after the for loops

for (int i = 0; i < \text{SIZE}; i++)
    for (int j = 0; j < \text{SIZE}; j++)
        // for each i,j
        for (int k = 0; k < \text{SIZE}; k++)
            // if path from i to j, record it
            \text{path}[i][j] = \text{true} \land (\text{path}[i][k] \land \text{edge}[k][j]);

// Invariant reestablished:
// \( 1 \leq n \leq \text{SIZE} \) and all paths of length \( n \) or less recorded in \text{Path}
// and only paths recorded in \text{path}

Proof Idea explains why this works.

PROGRAM SO FAR

```java
void \text{transClo}(boolean[][], boolean[][], path[][]) {
// Pre: \( \text{edge} \) and \( \text{path} \) are matrices of \( \text{SIZE} \times \text{SIZE} \land \text{SIZE} = 1 \)
// Post: \( \text{edge} = \text{edge0} \land \text{path} \) is \( \text{SIZE} \times \text{SIZE} \) matrix \land \)
// \( \text{path} \) represents trans. closure of \( \text{edge} \)
// \( \text{edge} \) doesn't change and assume dimension of \( \text{path} \) is \( \text{unchanged} \) so ignore 1st and 2nd conjuncts from now on
    int n = 1;
    for (int i = 0; i < \text{SIZE}; i++)
        for (int j = 0; j < \text{SIZE}; j++) \text{path}[i][j] = \text{edge}[i][j];

    while
        //Loop Inv.: all paths of length \( \leq n \) recorded in \text{path}
        // \( \land \) only paths recorded \( \land 1 \leq n \leq \text{SIZE} \)
        // Variant = \text{SIZE} - n
        (n < \text{SIZE}) {n++;
            // decrease Variant
            }
        // ... re-establish Invariant …
    // Invariant \( \land n \geq \text{SIZE} \), so \text{path} = transitive closure of \text{edge}
}
```

SHOW THE INVARIANT IS REESTABLISHED

There are 3 parts to the invariant. We'll look here at part (I3):

\[ \forall i,j:0..\text{SIZE}-1(\text{if } \text{path}[i][j] = \text{true} \text{ then } i \rightarrow^* j) \]

• Let \( \text{path1}[i][j] \) be value of \( \text{path} \) at start of an iteration for all \( i,j \).
  By Inv(3) \( \forall i,j:0..\text{SIZE}-1(\text{if } \text{path1}[i][j] = \text{true} \text{ then } i \rightarrow^* j) \) (**)

• Consider an arbitrary pair \( (i,j) \); assume for all \( (i,j) <_{\text{in}} (I,J) \) it holds that
  if \( \text{path1}[i][j] = \text{true} \text{ then } i \rightarrow^* j \) (*).
  Then if \( \text{path1}[I][J] = \text{true} \):
  
  **Case 1:** \( \text{path1}[I][J] = \text{true} \implies I \rightarrow^* J \) by (**)

  **Case 2:** \( \text{path1}[I][J] = \text{true} \) \&\& \( \text{edge}[K][J] = \text{true} \implies \text{path1}[I][K] = \text{true} \)
  
  either \( \text{path1}[I][K] \) has been reset and by (*) \( \implies I \rightarrow^* K \implies I \rightarrow^* J \),
  
  or \( \text{path1}[I][K] \) has not yet been reset and
  
  \( \text{path1}[I][K] = \text{true} \implies I \rightarrow^* K \) by (**)
  \( \implies I \rightarrow^* J \);
  
  (remember \( \text{edge}[K][J] = \text{true} \))

Either way \( I \rightarrow^* J \) and by induction we conclude  \( \text{path1}[I][J] = \text{true} \implies I \rightarrow^* J \)
Just for the record here are the proofs for the various checks that need to be made. (See end of these notes for some additional comments about reasoning with FOR loops.)

1. Check that the invariant is set up initially. \( n = 1 \), \( \text{path}[i][j] = \text{edge}[i][j] \). As \( \text{edge} \) is assigned to \( \text{path} \), if \( \text{path}[i][j] = \text{true} \) then \( \text{edge}[i][j] = \text{true} \) and \( i \rightarrow j \). Also if \( i = j \) then \( \text{edge}[i][j] = \text{true} \) and hence \( \text{path}[i][j] = \text{true} \). Since \( \text{SIZE} \geq 1 \) (by assumption) \( 1 \leq n \leq \text{SIZE} \).

2. Check that loop terminates. That’s easy: \( n \) increases so \( \text{SIZE} \cdot n \) decreases in each iteration. Since \( \text{SIZE} \cdot n \geq 0 \) by the invariant the loop cannot go on for ever.

3. Check that the postcondition is implied by false-while-condition and the invariant. Must show that all paths are recorded in \( \text{path} \), not just those of length \( \text{SIZE} \) or less. The argument on slide 15 showed this.

4. Check array accesses all valid. Since only for loops involved this is clearly the case.

5. Check the invariant is maintained. Within the loop \( n \) is increased by 1. Let’s call the value of \( n \) before the increment \( n_1 \). Since the while condition is true, \( 1 \leq n_1 \leq \text{SIZE} \) so \( 1 \leq n_1 + 1 \leq \text{SIZE} \), making \( (11) \) true again. The argument for \( (13) \) was shown on slide 20. For \( (12) \), we must show that, for each \( m \), \( 1 \leq m \leq n_1 + 1 \), if \( i \rightarrow j \) then \( \text{path}[i][j] = \text{true} \), for each \( i \) and \( J \). The invariant for \( n_1 \) tells us that for each \( m \), \( 1 \leq m_1 \leq n_1 \), if \( i \rightarrow j \) then \( \text{path}[i][j] = \text{true} \), and so \( \text{path}[i][j] = \text{true} \) as it never changes from true to false. Now suppose \( i \rightarrow j \). We know the path is made up of a path of length \( n_1 \) from \( i \) to \( K \) and an edge from \( K \) to \( J \), for some \( K \). Again by the invariant for \( n_1 \), \( \text{path}[i][K] = \text{path}[K][J] = \text{true} \). The for loop iteration for \( i = 1 \), \( j = 1 \) and \( k = K \) will set \( \text{path}[i][j] = \text{true} \) and it stays true.

### FIRST IMPROVEMENT

**(BUT STILL NOT WARSHALL’S ALGORITHM)**

Suppose all paths of length \( n \) or less are recorded in \( \text{path} \). Then any path of length \( 2^n \) or less can be decomposed into two parts, each of length \( n \) or less:

\[
\begin{array}{c}
& i & \rightarrow & k & \rightarrow & j \\
\end{array}
\]

Hence, for some node \( k \) we have already recorded that \( \text{path}[i][k] = \text{path}[k][j] = \text{true} \).

So instead of using the innermost statement

\[
\text{path}[i][j] = \text{path}[i][j] \lor (\text{path}[i][k] \land \text{edge}[k][j])
\]

we can **double** \( n \) at each stage by using the innermost statement

\[
\text{path}[i][j] = \text{path}[i][j] \lor (\text{path}[i][k] \land \text{path}[k][j])
\]

and replace the increment (by 1) of \( n \) by \( n = 2^n \).

### INEFFICIENCY

- There are four nested loops, controlled by \( n \), \( i \), \( j \) and \( k \). Each is executed \( \text{SIZE} \) times.
- Therefore, total number of iterations = \( \text{SIZE}^4 \).
- This measures the **complexity** of the algorithm: \( \text{SIZE} \) measures how big the problem is.
- The execution time increases roughly as the fourth power of the size of the problem.
- Thus big problems (lots of nodes) will take really quite a long time.
- **Approximately how long** for \( n=64 \)? \( n=256 \)? \( n=1024 \)?

Assume 16 iterations per microsecond:

- \( n=64 \): approx. \( 2^20 (2^24/2^4) \) approx. \( 10^6 \) \( \implies \) 1 sec
- \( n=256 \): approx. \( 2^28 \) approx. \( 10^9/4 \) \( \implies \approx 250 \) secs \( \implies \approx 4 \) mins
- \( n=1024 \): approx. \( 2^36 \) approx \( 64 \times 10^9 \) \( \implies \approx \) approx. 1000 mins = approx 16 hours!

This page is deliberately left blank
**IMPROVEMENT**

- Still four nested loops controlled by n, i, j, k.
- The outermost (n) loop is now executed approx. \( \log_2 \text{SIZE} \) times.
- The total number of iterations is of the order of \( \log_2 \text{SIZE} \times \text{SIZE}^3 \).
- \( \log_2 \text{SIZE} \) increases much more slowly than \( \text{SIZE} \).
- (Approximately) **how long** for \( n=64 \)? \( n=256 \)? \( n=1024 \)?

Assume 16 iterations per microsecond:

\( n=64 \): approx. 6 \times 2^{14} approx. 100 \times 1000 \implies 0.1 \text{ secs} \\
\( n=256 \): approx. 8 \times 2^{20} approx. \implies 8 \text{ secs} \\
\( n=1024 \): approx. 10 \times 2^6 \implies 640 \text{ secs}

- Some real timings on (a very old!) PC:

<table>
<thead>
<tr>
<th></th>
<th>n=32</th>
<th>n=64</th>
<th>n=128</th>
<th>n=256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>0.1s</td>
<td>1.5s</td>
<td>17s</td>
<td>?</td>
</tr>
<tr>
<td>Double</td>
<td>0s</td>
<td>0.1s</td>
<td>1.25s</td>
<td>9s</td>
</tr>
</tbody>
</table>

- Can we do better? **YES – Warshall’s Algorithm**

**PROOF IDEA**

Suppose Warshall’s invariant \( I_W \) holds:

we have recorded all paths whose transit nodes have ordinal \( < n \) and only those paths.

In increasing \( n \) to \( n+1 \), the new paths to record are just those that definitely pass in transit through the node with ordinal \( n \) (i.e. the one called \( n \)) …

… and whose other transit nodes all have ordinal \( < n \) (i.e. any of 0,1,2,...,n-1).

```
  i  n  j
```

Can stop when \( n=\text{SIZE} \) as no nodes have ordinal \( \text{SIZE} \)

**WARSHALL’S ALGORITHM (IDEA)**

- Present invariant is too “global”; it refers to all paths between nodes.
- Instead of restricting attention to paths of length \( \leq n \), can restrict the nodes that any path can go through.
- A **transit node** in a path is a node other than one of the endpoints.
- Assume the nodes are labelled by their position in the enumeration of nodes, in the range \( 0..\text{SIZE}-1 \), also called the **ordinal** of the node (node at position 0 is called 0, node at position 1 is called 1, etc.).
- \( i \Rightarrow n \ j \) means there’s a path from \( i \) to \( j \) using only transit nodes of ordinal-cm.
- Warshall’s invariant \( I_W \) – (for all \( i, j \) in \( 0..\text{SIZE}-1 \))

"path records those paths whose transit nodes have ordinal \( < n \)"

\[
\text{edge} = \text{edge0} \land \text{if } i \Rightarrow^n j \text{ then } \text{path}[i][j] = \text{true} \land \text{if } \text{path}[i][j] = \text{true} \text{ then } i \Rightarrow^n j \land 0 \leq \text{size} \leq \text{SIZE}
\]

We’ll ignore edge=\text{edge0} from now on as edge will not be changed.

**TOWARDS CODE (WARSHALL)**

As before, can assume each transit node is visited only once.

So we record a new path from \( i \) to \( j \) exactly when:

- there’s a path from \( i \) to \( n \) with all transit nodes \( < n \) …
- … and there’s a path from \( n \) to \( j \) with all transit nodes \( < n \)

That is, when we already have

\[
\text{path}[i][n] = \text{path}[n][j] = \text{true}.
\]

So, for each \( i, j \), we set \( \text{path}[i][j] \) to true if

- it’s already true,
- or else \( \text{path}[i][n] = \text{path}[n][j] = \text{true} \).

We don’t need to check all possible transit nodes \( k \), but just the single node \( n \). This cuts out an entire loop!!

This is Warshall’s algorithm.
INITIALISATION AND FINALISATION

Loop initialisation: When n = 0, the Invariant says:
"path records those paths whose transit nodes have ordinal < 0"

But no nodes have ordinal < 0, so this means path should record the
paths with no transit nodes at all. These are just the edges.

Therefore to initialize, just copy edge to path as before.

Loop Finalisation:
- At the end of the loop n ≥ SIZE and n<SIZE by Inv. => n=SIZE
- if i => SIZE j then path[i][j]=true ∧ if path[i][j]=true then i => SIZE j
- Since i ↦ j ↔ i => SIZE j (every node has ordinal < SIZE),
we can stop and the Postcondition is already satisfied:
  path[i][j] = true if, and only if, i ↦ j

LOOP CODE: RE-ESTABLISHING THE INVARIANT(1)

We've a Subcontract to re-establish lw: (n1 is value of n at loopstart)

Given:  (lw at start of loop): SIZE≥n1≥0 ∧
  i=>n1 j → path[i][j] = true ∧ path[i][j] = true → i=>n1 j
  while-condition-true: so n1 < SIZE

Show:  (lw at end of loop): SIZE≥n2≥0 ∧
  i=>n2 j → path[i][j] = true ∧ path[i][j] = true → i=>n2 j
  for (int i = 0; i < SIZE; i++)
    for (int j = 0; j < SIZE; j++)
      {path[i][j] = path[i][j] ∨ (path[i][n] & path[n][j]);}
    n++;               //lw

• 0≤n1<SIZE ≤ n1+1<SIZE => 0≤n2<SIZE (since n=n1+1)
• Array accesses OK since n<SIZE within loop.

PROGRAM SO FAR
void wtransClos(boolean [][] edge, boolean [][] path) {
  // Pre: edge and path are matrices of SIZE x SIZE ∧ SIZE≥1
  // Post: edge=edge0 ∧ path represents trans. closure of edge
  int n = 0;
  for (int i = 0; i < SIZE; i++)
    for (int j = 0; j < SIZE; j++)
      path[i][j] = edge[i][j];
  while (n < SIZE) {
    // lw: (IW1) SIZE≥n≥0 ∧
    // (IW2) if i => n then path[i][j] = true ∧
    // (IW3) if path[i][j] = true then i => n
    // Variant: SIZE-n
    // ... re-establish lw for 'n+1' ...
    n++;          // decrease Variant
    // lw ∧ n<SIZE, so deduce path is transitive closure of edge
  }
}

WARRSHALL INVARIANT IS RE-ESTABLISHED (2)

Given: i=>n1 j ↔ path[i][j] = true         (*)
Show: i=>n2 j → path[i][j] = true    (first part)
  // path[i][j] and n1 are values at start of loop
  for (int i = 0; i < SIZE; i++)
    for (int j = 0; j < SIZE; j++)
      path[i][j] = path[i][j] ∨ (path[i][n] & path[n][j]);
  n++;               //n=n1+1

FACT 1: for i≠n1 and j≠n1 path[i][j] is updated once
FACT 2: path[i][n1]=path[i][n1] and path[n1][j]=path[n1][j]
eg path[i][n1]
  =path[i][n1] ∨ (path[i][n1] & & path[n1][n1]) = path[i][n1]
Hence path[i][j]=path[i][j] ∨ (path[i][n1] & & path[n1][n1])
If i=>n2 j then either i=>n2 j or (i=>n2 j and n=>n2 j)
Hence from (*) path[i][j] is updated to true (path1[i][n1] etc=true)

WARMshall INVARIANT IS RE-ESTABLISHED (3)

Given: i=>j ↔ path1[i][j] = true
Show: path1[i][j] = true → i=>j (second part)

// path1[i][j] and n1 are values at start of loop
for (int i = 0; i < SIZE; i++)
    for (int j = 0; j < SIZE; j++)
        path1[i][j] = path[i][j] ll (path[i][n] & path[n][j]);
// path[i][j]= path[i][j] ll (path1[i][n1] & path1[n1][j]) (by FACT 2)

n++; // n=n1+1
if path[i][j]==true then by the code
    either path1[i][j]=true or path1[i][n1] = path1[n1][j] = true

Hence from (**): i=>o1j or i=>o1n+ and n=>o1j.

Either way i=>j

---

**WARSHALL’S ALGORITHM**

There are now three nested loops (for n, i and j).
Each one is executed SIZE times.
Therefore, total number of iterations ~ size^3.

*This (Warshall’s) is the most efficient of our three algorithms.*

- **(Approximately) how long** for n=64? n=512? n=1024?
Assume 16 iterations per microsecond:
  - n=64: approx. 2^14 => 0.016 secs
  - n=512: approx. 2^20 => 1 secs
  - n=1024: approx. 2^26 => 64 secs

- **Some real timings on a PC:**

<table>
<thead>
<tr>
<th></th>
<th>n=64</th>
<th>n=128</th>
<th>n=256</th>
<th>n=512</th>
<th>n=1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>1.5s</td>
<td>17s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>0.1s</td>
<td>1.25s</td>
<td>9s</td>
<td>74s</td>
<td></td>
</tr>
<tr>
<td>Warshall</td>
<td>0.05s</td>
<td>0.2s</td>
<td>1.5s</td>
<td>9s</td>
<td>73s</td>
</tr>
</tbody>
</table>

---

**How to obtain timings in a Java program.**

Java provides a useful class called `Date` in java.util, which allows you to obtain accurate times in milliseconds for your programs. You can use the methods of this class to find out how your program will behave for different sizes of the data structures. The study of these things is called complexity.

The constructor of the class `Date` measures and stores the current time in milliseconds. This time can be accessed via the `getTime()` method.

For example, in a test of Warshall’s algorithm, one can use a code fragment such as:

```java
long startTime = new Date().getTime(); // returns current time in ms
```

```java
if (path[i][j] == true) {
    startTime = new Date().getTime() - startTime;
} // gives runtime in ms
```

The entries in the table on slide 34 confirm some of the theoretical calculations we made about the complexity of the three algorithms, Warshall’s (C) and the two non-Warshall algorithms (A, B). For instance, if we write A(k) to mean algorithm A run on n=k, then you’d expect that the time for n=64 using the first algorithm ((2^26) x 4 iterations) would be similar to that for n=256 using Warshall ((2^8) x 3 iterations). And it is; i.e. A(64) = C(256). Similarly, you can check the following “equations” are also approximately true:

A(32) = B(64) = 2 x C(128); A(128) = 2 x C(512) = 2 x B(256); B(512) = C(1024), etc.

---

```java
void wtransClos(boolean [][] edge, boolean [] [] path) {
    // Pre: edge and path are matrices of SIZE x SIZE & SIZE=1
    // Post: edge=edge0 & path represents trans. closure of edge
    int n = 0;
    for (int i = 0; i < SIZE; i++)
        for (int j = 0; j < SIZE; j++)
            path[i][j] = edge[i][j];
    while (n < SIZE) {
        // Variant = SIZE-n
        for (int i = 0; i < SIZE; i++)
            for (int j = 0; j < SIZE; j++)
                path[i][j] = path[i][j] ll (path[i][n] & path[n][j]);
        n++; // make progress - decrease Variant
    }
    // lw and n=SIZE, so deduce path is transitive closure of edge;
}
```
SOME CURiosities

You may have wondered whether the invariant for the first algorithm, which was:
\[ \text{path}[i][j] = \text{true} \Rightarrow i \rightarrow j \land (i \not\rightarrow k) \land 1 \leq \text{size} \Rightarrow \text{path}[i][j] = \text{true} \land 1 \leq \text{size} \]

\[ \text{path}[i][j] = \text{true} \land 1 \leq \text{size} \Rightarrow \text{path}[i][j] = \text{true} \land 1 \leq \text{size} \].

Could be simplified to \( (i \rightarrow j) \land 1 \leq \text{size} \Rightarrow \text{path}[i][j] = \text{true} \land 1 \leq \text{size} \). For since we start with path recording paths of length 1, and gradually increase the length of the path on each iteration, it would appear that path records the existing paths of length \( \leq n \). But this is not quite so.

Consider the following example: Let SIZE=4 and edge[1][2] = edge[2][3] = edge[3][4] = true, with other values of edge[i][j]=false. Suppose n has just been updated to 2, then before the for loops path[1][3] and path[1][4] will be false. After the k = 2 iteration for i=1, j=3, path[1][3]=true and it stays true. For i=1, j=4, k=3, the assignment is path[1][4] = path[1][4] \[\land\] path[1][3] \&\& edge[3][4] = true. There is a path of length \( \leq 2 \) between 1 and 3, but there is no path of length \( \leq 2 \) between 1 and 4, although there is one of length 3. And yet path[1][4]=true. If we used the simplified invariant, it would not be re-established by the code. You could change the code so that it always used the value of path from the previous iteration, which would then re-establish the invariant, but this would require a lot of unnecessary copying of arrays.

On the other hand, for Warshall's algorithm, we can use the simpler invariant:

\[ (i \rightarrow j) \leftrightarrow \text{path}[i][j] = \text{true} \] and 0\leq\text{size}.

In this case, the values of path[n][i] and path[i][n] do not change in the n=n1-iteration and so the code for path[i][j] becomes path[i][j] = path[i][j] \[\land\] path[i][n1] \&\& path[i][n1]. (It doesn't matter whether path[i][n1] or path[n1][j] have been updated already in this iteration or not. 

Exercise: Show that path[i][n1] = path[i][n1] \&\& path[i][n1].

SUMMARY

- Defined Warshall's Algorithm by successive refinement.
- Without the Loop Invariant it would have been very difficult to say why the code should work.
- With the Loop Invariant, we can present the Proof Idea fairly easily. Then the code virtually writes itself.
- No detailed logical formalism was needed for either the Postcondition or the Loop Invariant or the Proof Idea. We used an informal language of edges, paths and diagrams. But this was quite enough to give us confidence in coding up that we could not have had otherwise.
- Use whatever language is most natural.

PATH FINDING ALGORITHMS – STORY SO FAR

Find which nodes are connected by a path in a graph
\[ \text{path}[i][j] = \text{true} \leftrightarrow i \rightarrow j \]

In Algorithm 1 paths of length=1 (edges), \( \leq 2 \), ..., were considered up to a maximum of \( \leq \text{SIZE} \).

Required SIZE’4 operations

In Algorithm 2 paths of length=1, \( \leq 2 \), \( \leq 4 \),... were considered up to a maximum of \( \leq \text{SIZE} \).

Required log, SIZE x SIZE’3 operations

Course 141
- Reasoning about Programs –

More on Graphs
Other Path Problems

Warshall's Algorithm considers paths according to transit nodes...

**Paths with no transit nodes, \( i \rightarrow j \) (edges)**

**Paths with transit nodes in \{0\}, \( i \rightarrow j \)**

**Paths with transit nodes in \{0,1\}, \( i \rightarrow j \)**...

**Paths with transit nodes in \{0,1,2,\ldots,\text{SIZE-1}\}, \( i \rightarrow \text{SIZE} j \)**

\[= \text{all paths.}\]

```
while (n < SIZE) {
  // lw: \( i \rightarrow j \) \rightarrow \text{path}[i][j] = true \land \text{path}[i][n] = true \rightarrow \text{i} \rightarrow \text{n} \rightarrow \text{j}
  // \land \text{SIZE} \geq n \geq 0
  for (int i = 0; i < SIZE; i++)
    for (int j = 0; j < SIZE; j++)
      path[i][j] = path[i][j] || (path[i][n] && path[n][j]);
  n++;
  // make progress - decrease Variant
  SIZE^3 operations
```

---

**THE BEST COST PATH PROBLEM**

We are given a directed graph in which each arc has a non-negative value (its cost).

The problem is to calculate the cost of the *least cost path* from every node to every other node, where the cost of a path is just the sum of the costs of the arcs on the path.

**Example:** Underground journeys (in either time or miles or £).

We assume that the routes go in both directions so all arrows can be omitted.

![Diagram of a graph with nodes and edges]

Best cost path from Vic to ChX = 6.5

---

**FLOYD'S ALGORITHM (VERY LIKE WARSHALL'S)**

- **Conventions:**
  SIZE = number of nodes
  Nodes labelled 0, 1, ..., SIZE-1
  Best cost of a path from a node to itself should have cost = 0

- **Loop variant:** SIZE-n.

- **Loop invariant \( I_f \):**
  "\( \text{SIZE} \geq n \geq 0 \), and
  \( \text{cost}[i][j] \) records the total cost of the minimal cost path from \( i \) to \( j \) *whose transit nodes are all \( \leq n \) (or \( \infty \) if there is no such path)"

These are analogous to Warshall.

---

**FLOYD'S ALGORITHM (INITIALISATION)**

- **Initialisation:**
  Initially, \( n = 0 \); i.e. no transit nodes at all. So at the start cost[i][j]
  should be the cost of the edge between \( i \) and \( j \)

  \[
  \text{edge}[i][i] = 0, \quad \text{for every} \; i, 0 \leq i < \text{SIZE} \\
  \text{edge}[i][j] = \text{cost of edge from} \; i \; \text{to} \; j \; \text{if it exists} \\
  \text{edge}[i][j] = \infty \; \text{if no edge exists} \\
  \text{(No edge means never reach} \; j \; \text{directly from} \; i \; \text{- ie infinite cost)}
  \]

- **Finalisation:**
  At the end, invariant and false while condition \( \Rightarrow \; n = \text{SIZE} \)
  cost[i][j] = cost of cheapest \( i \) - \( j \) path with all transit nodes \( < \) \( \text{SIZE} \)
  = cost of cheapest \( i \) - \( j \) path (for all \( i,j \)).
FLOYD'S ALGORITHM (CONTINUED)

• Continue loop while n < SIZE. (SIZE - n > 0)

• **Re-establishing the invariant:** At each step, and for each pair of nodes i,j, the least cost path will either be

  a) a cheapest i - j path with all transit nodes < n, or
  b) a cheapest i - n path with all transit nodes < n, concatenated with
     a cheapest n – j path with all transit nodes < n.

If (a), we leave cost[i][j] alone.
If (b), we set cost[i][j] to cost[i][n] + cost[n][j].

That is, to re-establish the invariant, it's enough if the code sets

\[
\text{cost}[i][j] = \min(\text{cost}[i][j], \text{cost}[i][n] + \text{cost}[n][j]).
\]

(Note that \(x + \infty = \infty\) and \(x + y < \infty\) if \(x < \infty\) and \(y < \infty\))

That is, for each pair of nodes i,j check if

\[
\text{cost}[i][n] + \text{cost}[n][j] < \text{cost}[i][j]
\]

and if it is reset the value of cost[i][j] to be this.

---

**EXAMPLE (UNDERGROUND JOURNEYS – EDGES)**

cost[i][j] represents the time for going from i to j.

```
<table>
<thead>
<tr>
<th></th>
<th>KX</th>
<th>LSq</th>
<th>BSt</th>
<th>ChX</th>
<th>E</th>
<th>SW</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>KX</td>
<td>0</td>
<td>7</td>
<td>10</td>
<td>∞</td>
<td>12</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>LSq</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>∞</td>
<td>6</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>BSt</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChX</td>
<td>12</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>6</td>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>6</td>
<td>0</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>∞</td>
<td>∞</td>
<td>10</td>
<td>∞</td>
<td>5.5</td>
<td>2.5</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Exercise:** Complete the cost matrix

---

**UNDERGROUND JOURNEYS – BEST COSTS**

```
<table>
<thead>
<tr>
<th></th>
<th>KX</th>
<th>LSq</th>
<th>BSt</th>
<th>ChX</th>
<th>E</th>
<th>SW</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>KX</td>
<td>0</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>12</td>
<td>17</td>
<td>17.5</td>
</tr>
<tr>
<td>LSq</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>11</td>
<td>11.5</td>
</tr>
<tr>
<td>BSt</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>ChX</td>
<td>10</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>6.5</td>
</tr>
<tr>
<td>E</td>
<td>12</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
</tr>
<tr>
<td>SW</td>
<td>17</td>
<td>11</td>
<td>11</td>
<td>6</td>
<td>5</td>
<td>2.5</td>
<td>2.5</td>
</tr>
<tr>
<td>V</td>
<td>17.5</td>
<td>11.5</td>
<td>10</td>
<td>6.5</td>
<td>5.5</td>
<td>2.5</td>
<td>0</td>
</tr>
</tbody>
</table>
```
**CODE FOR FLOYD'S ALGORITHM**

```java
ftransClos(double [] [] edge, double [] [] cost) {
    // Pre: edge is a matrix of SIZE x SIZE = SIZE + 1
    // Post: edge = edge0 & r gives least costs of paths using edge
    int n = 0;
    // Initialise cost to edge by two for loops
    while (n < SIZE) {
        // Variant: SIZE-n
        // I: cost[i][j] gives cost of cheapest path from i to j
        // with transit nodes of ordinal < n & SIZE = n + 0
        // ... re-establish I for incremented value of n ...
        for (int i = 0; i < SIZE; i++)
            for (int j = 0; j < SIZE; j++)
                cost[i][j] = Math.min(cost[i][j], (cost[i][n] + cost[n][j]));
        n++;
        // make progress - decrease Variant
    }
}
```

**FLOYD INVARIANT IS RE-ESTABLISHED**

If: cost1[i][j] = min cost of path from i to j with transit nodes < n

\[ \text{cost1[i][j] is value at start of loop} \]

for (int i = 0; i < SIZE; i++)
    for (int j = 0; j < SIZE; j++)
        cost[i][j] = Math.min(cost[i][j], (cost[i][n] + cost[n][j]));

FACT: cost[i][n1] = cost1[i][n1] and cost1[n1][j] = cost1[n1][j]

eg min(cost1[i][n1],(cost1[i][n1] + cost1[n1][n1])) = cost1[i][n1] WHY?

for i≠n1 and j≠n1 cost[i][j] is updated once

cost[i][j] = min(cost1[i][j],(cost1[i][n1] + cost1[n1][j]))

n++; // n=n1+1

Hence cost[i][j] satisfies If:

\[ \text{cost[i][j] = min cost of path from i to j with transit nodes < n} \]

**SOME CONSIDERATIONS**

The code requires manipulation and representation of \( \infty \).

Java provides a constant called Double.POSITIVE_INFINITY that is used to represent a number that is too big to represent.

That is,

- Double.POSITIVE_INFINITY represents any value greater than the largest representable number - about 2\(^{308}\) x 1.8;
- if \( x+y \) would result in a number too big to represent, it is replaced by the value Double.POSITIVE_INFINITY;

Note that, in the second of these Java does not throw an arithmetic overflow exception. As a user you must be aware of this - and still may need to check for it (although not here).

However, other languages may not be so convenient and so the programmer may have to provide some sort of implementation of \( \infty \) for this application. For example, see next slide.

**GENERALLY IMPLEMENTING \( \infty \)**

We want \( \infty \) to be very large - always larger than other values.

For this application we can simplify things a little if we define BIG as

```java
final double BIG = (Double.MAX_VALUE/2) - 1;
```

We must be able to make comparisons with BIG, which represents \( \infty \).

```java
private double bigMin(double x, double y) {
    // x >= BIG && y < BIG return y
    // y >= BIG && x < BIG return x
    // x >= BIG && y >= BIG return BIG
    // otherwise return Math.min(x,y)}
```

```java
private double bigPlus(double x, double y) {
    // x >= BIG || y >= BIG return BIG
    // x+y >= BIG return BIG
    // BIG was chosen so x+y should not cause overflow,
    // but implementation should catch it just in case ...
    // otherwise return x+y
```
**Some properties of bigMin and bigPlus**

The value of BIG as (MAX\_DOUBLE/2)-1 is chosen to ensure that there is no arithmetic overflow in any operation in the algorithm.

Where could overflow occur? Clearly, in the step to evaluate cost[i][n]+cost[n][j].

Suppose that all costs c in edge are initially in the range 0≤c<BIG/SIZE. The smallest cost path using SIZE arcs will be ≤BIG. In the update, at worst we add two of these values.

In bigPlus if the sum of two such costs is made, the result ≤ will be in the range 0≤BIG+BIG < MAX\_DOUBLE. Hence the result ≤ is representable.

More importantly, nowhere should the numbers ever be as large as MAX\_DOUBLE and so nowhere should there be arithmetic overflow.

**Re-establishing the Invariant**

Note that the updating code in the n−iteration for [i][n] is "cost[i][n]=min(cost[i][n], cost[i][n−1]+cost[n][n])"; since all costs are ≥0 cost will not change for this case, and similarly, nor will cost[n][n] change. So cost[i][n] only changes in the n−iteration for i≠n and j≠n, and then it can only decrease.

Before the for loops the Inv. says that cost[i][n] gives minimal cost of a path from i to j with transit nodes <1. After the for loops we want to show that cost[i][n] gives minimal cost of a path from i to j with transit nodes ≤1. For any i, j, such a cost = min(cost[i][j], cost[i][n]+cost[n][j]), and once updated will not change in the n−iteration. Since cost[i][n] and cost[n][j] do not change in the n−iteration the order of the updating for I and J doesn't matter.

---

**To compute the shortest length path from i to j**

**Choice of values in edge:** cost of an edge = its length = 1

- edge[i][i]=0
- edge[i][j]=1 if there is an edge from i to j (i≠j)
- edge[i][j]=∞ if no edge from i to j (i≠j) (could also use SIZE)

**Update:** (as in Floyd's algorithm)

\[
\text{cost[i][j]} = \text{Math.min(cost[i][j], cost[i][n]+cost[n][j])}
\]

**Finally:**

- if no path from i to j then cost[i][j]=∞, and
- if some path from i to j (i≠j) then cost[i][j]=length of shortest path from i to j. (If i=j then cost[i][j]=0.)

---

**SOME APPLICATIONS**

- The same method can be used for other applications too.
- How could you find the shortest length path between 2 nodes (ie the path with the fewest arcs)?
- Suppose nodes are coloured black and white. Can you adapt Warshall’s algorithm to find the "piebald" paths – those whose edges alternate between a black node and a white node?
- Suppose you are responsible for finding a route to transport a wide load. You have to find a route that is wide enough on all sections to accommodate the load (but no wider than needed). How could you adapt the Warshall-Floyd algorithm to deal with this application?

---

**Piebald graphs**

A Piebald graph has edges only between different colour nodes

In a path in a Piebald graph the node colours alternate

Can be no edge between b and d

\[
\begin{align*}
\text{edge[i][j]} = \text{true} & \iff \exists \text{ an edge from i to j and col(i) } \neq \text{ col(j)} \\
\text{Standard Update: path[i][j]} = \text{path[i][j]} \ll (\text{path[i][n]} \&\& \text{path[n][j]}) \\
\text{Invariant: path[i][j]} = \text{true} & \iff i \Rightarrow j \text{ & colours alternate on path}
\end{align*}
\]

Why will the invariant be maintained?
**Transport Example**

![Transport Example Diagram]

eg: cost of edge = width
    cost = 0 if no edge

**Update: Floyd**

\[
\text{cost}[i][j] = \min (\text{cost}[i][j], \text{cost}[i][n] + \text{cost}[n][j])
\]

**Transport**

\[
\text{cost}[i][j] = \max (\text{cost}[i][n], \text{cost}[n][j])
\]

**Invariant:** cost[i][j] = cost of path with max overall width

Check cost[i][n] does not change on the nth iteration, so the previous argument for maintaining the Invariant still works.

---

**A MORE ABSTRACT VIEW OF WARSHALL’S ALGORITHM**

In Warshall's algorithm there are two essential operators, one which we'll denote by ⊕ used to "combine" paths, and one which we'll denote by ⊗ to "compare" paths, together with an essential assignment of values to the edge matrix.

The main assignment in the code is then path[i][j] = path[i][j] ⊗ (path[i][n] ⊕ path[n][j]).

We can tabulate and consider these operators for various cases:

- **Warshall:** ⊗ = ||; ⊕ = &&; edge values are true (an edge exists) or false (no edge exists).
- **Floyd:** ⊗ = min; ⊕ = +; edge values are ≥0 (representing cost of an edge if it exists) or ∞ (no edge exists).
- **Fibahal:** ⊗ = ||; ⊕ = &&; edge values are true (an edge exists between a black and a white node) or false (no edge exists).
- **Safe transport:** ⊗ = max; ⊕ = min; edge values are ≥0 (an edge exists – e.g. could represent width of a road, or strength of a road) or 0 (no edge exists – e.g. also means useless path or bridge!). The overall cost of a path is its narrowest link. We want the best of such paths.
- **Shortest time for overall job completion (graph is acyclic):** ⊗ = min; ⊕ = max; edge values are ≥0 (an edge exists – e.g. time for a sub-job) or ∞ (no edge exists – e.g. sub-job can never be finished). The overall cost of a job (which requires all sub-jobs represented by edges to be completed in parallel) is the cost of the longest sub-job. We want shortest overall time.

---

**MORE DETAILS OF THE ABSTRACT VIEW**

The main assignment in the code is path[i][j] = path[i][j] ⊗ (path[i][n] ⊕ path[n][j]).

The argument that the algorithm is correct follows the same principle in all cases:

1. Show that path[i][n] = path[i][n] ⊕ (path[i][n] ⊕ path[n][n]) remains unchanged for the particular operators ⊕ and ⊗ for the iteration n. Do a similar thing for path[n][j].
2. Show the invariant is initially made true by the assignment to edge.
3. Show that the invariant is re-established after the 2 for loops.
4. Show that the postcondition follows from the invariant when n=SIZE.

eg - for the shortest path application:

Edge[i][i]=0; Edge[i][j]=1 (ij) for existing edges and = SIZE for non-existing edges. **Invariant:** path[i][j] gives the length of the shortest path from i to j using transit nodes in {0,1,...,n-1} and path[i][i]=0.

Why can we use SIZE for a non-edge? Any real path will have cost = length of the path, since cost of a path is obtained by adding edge costs, each of which is 1. Edge[i][i]=0 for all i and remains so from the update rule. For ij the shortest real path between i and j will use the other SIZE-2 nodes at most once leading to SIZE-1 edges. You could also use ∞ for the cost of a non-edge between i and j as in the standard Floyd.

Clearly, the postcondition follows from invariant when n=SIZE. The invariant is true when n=0 as there are no nodes in the set of allowable transit nodes. Re-establishing the invariant holds as the shortest path with transit nodes in {0,1,...,n-1} is either one using transit nodes in {0,1,...,n-1} or it is one that goes through n exactly once, as computed by the update.

---

**APPENDIX – FOR LOOPS**

```java
public class Matrix{
    private int [][] m, int size;
    //class invariant: size>0
    public Matrix(int n){
        m = new int [n][n]; size = n;
    }
    //public String toString() for printing out matrices
    public void zeromatrix(){
        for (int i=0; i<size; i++)
            for (int j=0; j<size; j++)
                m[i][j] = 0;
    }
    //What restriction would ensure the invariant is true initially?
}
```

---

March 6, 2009 Warshall and Floyd Algorithms page 57
March 6, 2009 Warshall and Floyd Algorithms page 58
March 6, 2009 Warshall and Floyd Algorithms page 59
March 6, 2009 Warshall and Floyd Algorithms page 60
FOR LOOP REASONING

**for** loops are typically used to do the same operation to all elements of an array. Different iterations of the loop do not interfere with each other and the fact they happen in some particular order is irrelevant.

(i) Sometimes the operations could be executed in parallel. Such **for** loops can be thought of as "do all these".

\[ \text{e.g. } \textbf{for} \ (\text{int} \ i = 0; \ i < a.\text{length}; \ i++) \ a[i]=0; \]

(ii) Sometimes this is not so, although the iterations could still be executed in any order. Such **for** loops can be thought of as "do this, then this,...".

\[ \text{e.g. } s = 0; \ \textbf{for} \ (\text{int} \ i = 1; \ i <= 5; \ i++) \ s+= i; \]

Why can’t the operations occur in parallel here?

It is safest to reserve **for** loops for independent operations as in (i); a **for** loop can be coded as a **while** loop and for reasoning purposes it is often simpler to reason about the corresponding **while** loop.

You may think that the second kind of **for** loop could be run in parallel. But consider again $s=0; \ \textbf{for} \ (\text{int} \ i =1; \ i<= 5; \ i++) \{s+=i;\}$

It does satisfy a kind of independence — it doesn't seem to matter in which order the steps are taken. To show the loop gives $s=15$, we could try to show that $s=(\text{Sum}(i=1 \ to \ 5)(i))=0$ ($=1+2+3+4+5=15$).

We could argue as follows: imagine $s$ is a location (a ‘bucket’ if you wish), initially with value 0. For an arbitrary $i$, the $i$th iteration adds $i$ into the location $s$ and nothing removes it. So at the end of the loop every value of $i$ has been added in and the value of $s$ is the total sum. However, it is only correct if we assume the additions are made at different times.

Imagine that we tried to make the additions for $i=2$ and $i=3$ at exactly the same time. We might argue as follows: "look in $s$ — the value is (say) 1 at the moment; "compute $i=1$ = 3" (i.e. $2+1$) — make sure the value of $s$ is now 3. But at the same time, the computation for $i=3$ would result in the conclusion "make sure the value of $s$ is now 4". So what is the value of $s$?

There are various calculi for reasoning about such parallel operations - see the second (and fourth) year courses in concurrency.

Of course, in practice, you will use **for** loops even when the operations are not independent. However, such loops are really masquerading as **while** loops and when reasoning about them you need to use the technique of invariants.

PROVING THE POSTCONDITION HOLDS

\[
\begin{align*}
\textbf{for} \ (\text{int} \ i=0; \ i<\text{size}; \ i++) \\
\textbf{for} \ (\text{int} \ j=0; \ j<\text{size}; \ j++) \ m[i][j] = 0;
\end{align*}
\]

Let $I$ and $J$ be integers $0 \leq IJ<\text{size}$. Show that, at the end, $m[i][j] = 0$.

**Proof:** there was an iteration of the **for** loops (namely with $i=I$ and $j=J$) in which $m[i][j]$ became 0; once that was done, none of the other iterations would ever undo it.

The pattern is quite general, and very easy. You reason that everything necessary was done, and then (because the iterations are independent) never undone. **for** loops should always terminate!

\[ s = 0; \ \textbf{for} \ (\text{int} \ i = 1; \ i <= 5; \ i++) \ s+= i; \]
- We must show that $s=(\text{Sum}(i=1 \ to \ 5)(i))$. Let $I$ be an arbitrary int.
- There is exactly one iteration which adds $I$ to $s$. Since this step is not undone, $s=(\text{Sum}(i=1 \ to \ 5)(i))+\text{initial value} =(\text{Sum}(i=1 \ to \ 5)(i)) + 0 = (\text{Sum}(i=1 \ to \ 5)(i))$.

A TYPICAL FOR LOOP?

\[
\textbf{boolean} \ \texttt{neg1} \ (\text{int} \ []) \ a \{
\quad \text{//post: r } \leftrightarrow \ a \ \text{contains a negative integer } \land \ a=a0
\}
\]

\[
\textbf{boolean} \ \texttt{isNeg} = \text{false};
\quad \textbf{for} \ (\text{int} \ i=0; \ i<\text{a.\text{length}}; \ i++) \ \texttt{isNeg} = \texttt{isNeg} \land \ (a[i]<0);
\quad \text{return} \ \texttt{isNeg};
\]

If $a[i]<0$ for every $i$ then $\texttt{isNeg} = \texttt{result} = \text{false}$, which is correct.

If $a[i]<0$ for some $i$, say $I$, then $\texttt{isNeg} = \text{true}$ after the $1^{st}$ iteration and stays true, and $\texttt{result} = \text{true}$, which is correct.

\[
\textbf{boolean} \ \texttt{neg2} \ (\text{int} \ []) \ a \{
\quad \text{for} \ (\text{int} \ i=0; \ i<\text{a.\text{length}}; \ i++) \ \text{if} \ (a[i]<0 \ \text{return} \ \text{true};
\quad \text{return} \ \text{false};
\}
\]

If $a[i]<0$ for every $i$ then $\texttt{neg2}$ returns from outside the for loop with $\texttt{result} = \text{false}$, which is correct. If $a[i]<0$ for some $i$, say $I$, then $\texttt{neg2}$ would return after the $I^{th}$ iteration with $\texttt{result} = \text{true}$, which is correct.
BUT ...

- The result of executing neg1 and neg2 would be the same even if the iterations of the for loop were executed in a different order.

- The reason is that although the answer could have been determined by any of the iterations, that answer would be the same in all cases.

- This is not the case for neg3.

```c
int neg3 (int [] a) {
    //post "returns the first value of i: a[i]<0 (if any)" \ a=a0
    //formally??
    for (int i=0; i<ca.length; i++) if a[i]<0 return i;
    return a.length;
}
```

- The answer depends on which for loop iteration causes the return.
- Use a while loop for this kind of for loop.

Let's apply the method to our earlier for loop:

```c
s=0; for (int i = 1; i<=5; i++) s=s+i;
```

As a while loop it becomes

```c
int i = 1; 
while i<=5 {   //inv true here and while condition true
    s = s+i; i=i+1; } //inv true here and while condition false
```

In order to show this loop adds together the first five positive integers we must find and include the correct mid-condition as invariant and show it is maintained. We must also find a variant and show the loop stops at the right time.

The **variant** in this case is 6-i (the loop will stop when it reaches 0). 6-i>0 <=5-1>0, so the loop test is equivalent to variant>0.

The **invariant** should represent the state when we are making progress. It should tell us what we have added to s so far. It is s=Sum(k)(k=1 to i-1) \ 1≤i≤6. We make the convention that Sum(k)=1 to 0 is 0. Now we show that the loop works and also that it stops. Note the second conjunct of the invariant – it’s important! It is equivalent to variant>0.

The **loop stops**: the variant decreases at each iteration since i increases. Within the loop (i.e. when the while condition is true) the variant is >0. Hence the loop must stop since the variant cannot continue decreasing and remain >0.

The **invariant is set up initially**: when i=1, s should be 0; it is. Also 1≤i≤6.

CONVERTING FOR LOOPS TO WHILE LOOPS

Generally, a for loop of the form

```c
for (<init> <test> <inc>) <code>
```

becomes the while loop

```c
<init>
while <test> {   //inv true here and <test> true
    <code>
    <inc> //variant decreased 
} //inv true here and <test> false
```

It’s up to you to find the right variant and invariant for the problem. The variant is often 0 when the loop stops – so test is variant>0. The invariant often includes the property variant≥0.

The **invariant is maintained**: call the values of i and s at the start of the loop il and si. The new requirement is s>=Sum(k)(k=1 to i(l-1)) = si-il. This is exactly what the loop code computes — first it adds il to si, then it increments il to il+1. Also 1≤il<i(l) as the true while condition gives il≤5. More formally, you have to show that 1≤il<i(l) \ 1≤il<i(l) = Sum(k)(k=1 to i(l-1)) (before the code)

```c
i=l+i \ s=si+il \ (effect of code)
```

```c
==l≤i≤5 \ s=Sum(k)(k=1 to i(l-1)) (after the code)
```

First, 1≤il<6 \ i+1 \ s=si+il \ s=si+il (effect of code) tells us i>5 and the invariant that i<6 \ s=Sum(k)(k=1 to i(l-1)). So i=6 and s=Sum(k)(k=1 to 5). Done!

Generally, a for loop of the form for (<init> <test> <inc>) <code> becomes the while loop

```c
<init>while <test> {   //inv true here and <test> true
    <code><inc> //variant decreased 
} //inv true here and <test> false
```

It is up to you to find the right variant and invariant for the problem. The variant is often 0 when the loop stops and the invariant often includes variant≥0. E.g. in the above example the variant was 6-i; it is 0 when i=6, which is when the loop will terminate. In addition i=5 is equivalent to 6-i, which is included in the invariant.

**Exercise**: Formalise neg3 as a while loop with suitable invariant.