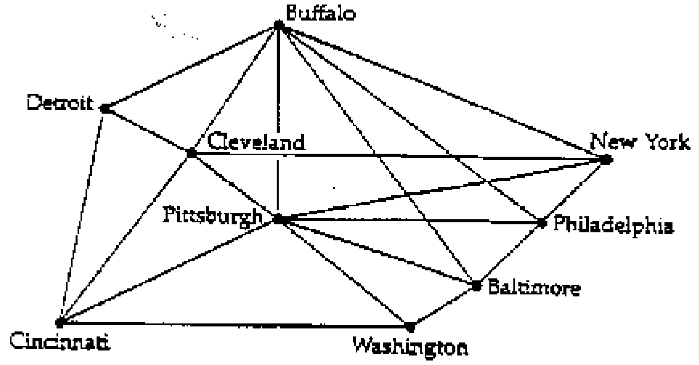
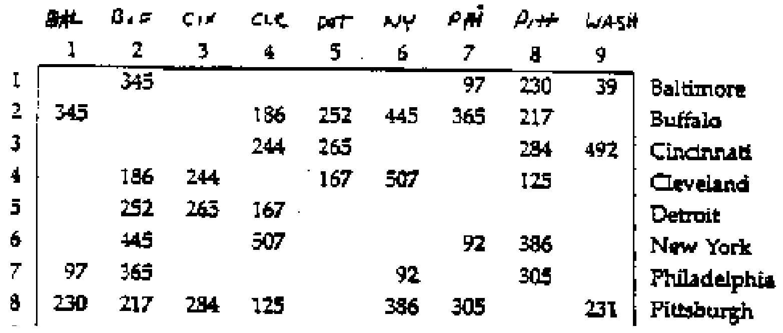
CLASS NOTES, CS W3137

1 Finding Shortest Paths: Dijkstra's Algorithm



(a) The graph



	BAL 1	BUF 2	CIN 3	CLE 4	DET 5	NY 6	PHI 7	PITT 8	WASH 9	
1		345					97	230	39	Baltimore
2	345			186	252	445	365	217		Buffalo
3				244	265			284	492	Cincinnati
4		186	244		167	507		125		Cleveland
5		252	263	167						Detroit
6		445		507			92	386		New York
7	97	365				92		305		Philadelphia
8	230	217	284	125		386	305		231	Pittsburgh

EXAMPLE OF DIJKSTRA'S ALGORITHM

			DISTANCES								
			1	2	3	4	5	6	7	8	9
Iteration	Settled	Selected	Bal	Buff	Cinc	Clev	Det	NYC	Phl	Pitt	Wash
Initial			0	345	inf	inf	inf	inf	97	230	39
1	1	9	0	345	531	inf	inf	inf	97	230	39
2	1,9	7	0	345	531	inf	inf	189	97	230	39
3	1,9,7	6	0	345	531	696	inf	189	97	230	39
4	1,9,7,6	8	0	345	514	355	inf	189	97	230	39
5	1,9,7,6,8,	2	0	345	514	355	597	189	97	230	39
6	1,9,7,6,8,2	4	0	345	514	355	522	189	97	230	39
7	1,9,7,6,8,2,4	3	0	345	514	355	522	189	97	230	39
8	1,9,7,6,8,2,4,3		0	345	514	355	522	189	97	230	39

1. We want to compute the shorterst path distance from a source node S to all other nodes. We associate lengths or costs on edges and find the shortest path.

2. We can't use edges with a negative cost. If graph has cycles, we can take endless loops to reduce the cost by continuing to travel along the negative cost edge.

3. Finding a path from vertex S to vertex T has the same cost as finding a path from vertex S to all other vertices in the graph (within a constant factor).

4. If all edge lengths are equal, then the Shortest Path algorithm is equivalent to the breadth-first search algorithm. Breadth first search will expand the nodes of a graph in the minimum cost order from a specified starting vertex (assuming equal edge weights everywhere in the graph).

5. **Dijkstra's Algorithm**: This is a greedy algorithm to find the minimum distance from a node to all other nodes. At each iteration of the algorithm, we choose the minimum distance vertex from all unvisited vertices in the graph,

   - There are two kinds of nodes: **settled** or closed nodes are nodes whose minimum distance from the source node S is known. **Unsettled** or open nodes are nodes where we don't know the minimum distance from S.
   - At each iteration we choose the unsetteld node V of minimum distance from the source S. This settles (closes) the node since we know its distance from S. All we have to do now is to update the distance to any unsettled node reachable by an arc from V. At each iteration of the algorithm, we close off aother node, and eventually we have all the minimum distances from source node S.

```
void dijkstra( Vertex S){
    for each Vertex v {
        v.dist = INFINITY;
        v.known = false;
    }
    s.dist = 0;
    for( ; ; )  {
        Vertex v = smallest unknown distance vertex;
        if( v == NOT_A_VERTEX )
            break;
        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
                if( v.dist + cvw < w.dist )  {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
    }
}
```

6. Cost: using the table on page 1, we need to find the minimum distance out of V cities (vertices) ($O(V)$) and we need to do this $V - 1$ times, yielding an $O(V^2)$ algorithm. We also may have to do an update on each of E edges to reduce cost, so the total running time is $O(V^2 + E) = O(V^2)$. We can reduce this cost by using a priority queue that holds the cities and their distances. Since we are seeking the smallest distance each time, this is a $find - min$ operation on the priority queue. As distances are updated, we add them to the priority queue, and the new, smaller distance is always deleted first. We do have to check that the distance we get via a $find - min$ operation is not for a closed, settled vertex. This brings the complexity to $O(V + E\,log\,V)$. The V term is the cost of building the initial heap, and the $E\,log\,V$ term is the update to insert a new distance into the queue as an edge updates a distance.

7. Sketch of Proof that Dijkstra's Algorithm Produces Min Cost Path

   (a) At each stage of the algorithm, we settle a new node V and that will be the minimum distance from the source node S to V. To prove this, assume the algorithm *does not* report the minimum distance to a node, and let V be the first such node reported as settled yet whose distance reported by Dijkstra, $Dist(V)$, is not a minimum.

   (b) If $Dist(V)$ is not the minimum cost, then there must be an unsettled node X such that $Dist(X)+Edge(X,V) < Dist(V)$. However, this implies that $Dist(X) < Dist(V)$, and if this were so, Dijkstra's algorithm would have chosen to settle node X before we settled node V since it has a smaller distance value from S. Therefore, $Dist(X)$ cannot be $< Dist(V)$, and $Dist(V)$ is the minium cost path from S to V.

2 Computing Transitive Closure of a Graph

1. Given Graph G below, Can we find the Transitive Closure of this Graph? Essentially this computes if there is **any** path that exists between two nodes - can we get from A to B in a graph.

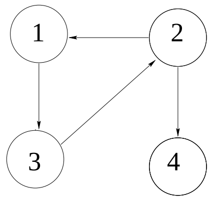


Figure 1: Graph used in Transitive Closure example

2. We can think of an Adjacency Matrix A as a Boolean Matrix where a 1 means an edge exists between 2 vertices, and a 0 means there is no edge. Assume each edge has length 1. The Adjacency Matrix A encodes the information about paths of length 1 - what edges exist in the graph. Then $A * A$ = A^2= all paths of length 2 between specified vertices. $A * A * A$ =A^3= all paths of length 3 between specified vertices. In this analysis, we multiply matrices using Boolean AND for Multiplication and Boolean OR for Addition:

$$A^1 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, A^2 = A{*}A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} A^3 = A{*}A^2 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, A^4 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{1}$$

$$PATH = A^1 \vee A^2 \vee A^3 \vee A^4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2}$$

   Note: $\vee$ is the logical OR operator.

3. The **transitive closure** of a graph is the logical OR of all these matrices A^i, i=1 to N: $PATH[i][j] = A^1[i][j] \vee A^2[i][j] \vee \ldots \vee A^N([i][j]$. We are simply stating that $PATH[i][j]$ is true if there is a path of length 1 from i to j, OR a path of length 2 from i to j,..., OR a path of length N from i to j.

4. Finding out if there is a path between two vertices in a directed graph: Consider the logical expression:
   $A^1[i][k]$ AND $A^1[k][j]$

   This is TRUE (TRUE=1) if and only if an edge exists from i to k AND an edge exists from k to j. $A^2[i][j]$ will be 1 if there is a path of length 2 from i to j. Similarly for $A^3[i][j]$, and so on. We need to OR together all N matrices to find if any path exists, because the longest simple path in graph of N nodes has length N - and this is a simple **cycle**. Adding paths longer than N will give us no new information.

5. The method above is **slow** for computing transitive closure of a graph of N nodes: The total cost to create $N-1$ A^i matrices is $(N-1)*N^3 = O(N^4)$ Plus we have to also OR together $N-1$ matrices, where each matrix has N^2 elements, yielding $O(N^4 + (N-1)*N^2)$ operations!

6. Can we speed up this algorithm? Yes, thanks to Warshall's algorithm (a variant is known as Floyd's algorithm). Also, reference Weiss text, section 10.3.4, page 472.

```
for(k=1; k<=N; k++)
   for(i=1; i<=N; i++)
         for(j=1; j<=N; j++)
             A[i][j] = A[i][j] OR ( A[i][k] AND A[k][j])
```

   This algorithm is $O(N^3)$. We use a dynamic programming approach above where we simply use an intermediate node k each time as a way station between nodes i and j.

7. Define $A_k[i][j]$ as TRUE if there exist a path between nodes i and j that **does not** go through an intermediate node numbered higher than k.

8. $A_0[i][j]$is just the initial adjacency matrix. On a graph with N nodes, $A_N[i][j]$ is the transitive closure of the graph, since it encodes all paths between nodes i and j that do not go through any nodes numbered higher than N - which is in fact all possible paths.

9. The trick is to compute $A_k[i][j]$ from $A_{k-1}[i][j]$. $A_k[i][j]$ is TRUE if a path exists between nodes i and j that does not go through any node labeled higher than k. This will be TRUE if:

   (a) There is a direct edge between i and j. This is encoded in $A_0[i][j]$.

   (b) There is a path from i to k, and a path from k to j. The two components of this path are paths that must go through nodes labeled no higher than $k-1$, and this information is contained in $A_{k-1}[i][k]$ and $A_{k-1}[k][j]$. So we can use the contents of the already computed matrices to generate the next level of matrix:

$$A_k[i][j] = A_{k-1}[i][j]\ OR\ (A_{k-1}[i][k]\ AND\, A_{k-1}[k][j]) \tag{3}$$

   Since we can reuse the same A matrices, we can drop the subscripts and use a single matrix $A[i][j]$ that we fill up as we iterate over each intermediate node k that our paths go through.

10. Note: Matrix Multiplication is defined as:

$$c_{ij} = \sum_k a_{ik}b_{kj} \ , \quad A * B = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \tag{4}$$

$$= \begin{bmatrix} a_{11}*b_{11}+a_{12}*b_{21}+a_{13}*b_{31}+a_{14}*b_{41} & \dots & \dots & a_{11}*b_{14}+a_{12}*b_{24}+a_{13}*b_{34}+a_{14}*b_{44} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ a_{41}*b_{11}+a_{42}*b_{21}+a_{43}*b_{31}+a_{44}*b_{41} & \dots & \dots & a_{41}*b_{14}+a_{42}*b_{24}+a_{43}*b_{34}+a_{44}*b_{44} \end{bmatrix} \tag{5}$$

```
// Compute transitive closure of a graph
public class transitiveclosure {

public static void main( String [ ] args ) {

// Graph is stored as an adjacency matrix: 1=edge exists, 0= no edge

int i,j,k;
int [ ][ ] a = { { 0,0,1,0 },
                     { 1,0,0,1 },
                     { 0,1,0,0 },
                     { 0,0,0,0} };

     System.out.println("Original Adjacency Matrix:");
     for(i = 0; i < 4; i++ ){
        for(  j = 0; j < 4; j++ )
          System.out.print( a[ i ][ j ] + "    " );
        System.out.println( );
     }
     System.out.println( );
// see if vertices i,j reachable through vertex k

   for(  k = 0; k < 4; k++ )
     for( i = 0; i < 4; i++ )
       for(  j = 0; j < 4; j++ )
          a[i][j] = a[i][j] | (a[i][k] & a[k][j]);

   System.out.println();
// print out the reachability matrix
   System.out.println("Reachability Matrix:");
   for(  i = 0; i < 4; i++ ) {
      for(  j = 0; j < 4; j++ )
           System.out.print( a[ i ][ j ] + "    " );
      System.out.println( );

   }
}
}
-----------------------------
Original Adjacency Matrix:
0    0    1    0
1    0    0    1
0    1    0    0
0    0    0    0

Reachability Matrix:
1    1    1    1
1    1    1    1
1    1    1    1
0    0    0    0
```

If we use the actual distances on the edges instead of just a binary adjacency, Warshall’s algorithm allows us to create an all pairs distance array. We simply have to change the update from a binary reachability update to a distance update.

```
old update:  a[i][j] = a[i][j] | (a[i][k] & a[k][j]);

new update: if (d[i][j] >  d[i][k] + d[k][j])
                   d[i][j] = d[i][k] + d[k][j];
```

here is the main loop of the program:

```
for(  k = 0; k < Dimension; k++ ) {
   for( i = 0; i < Dimension; i++ )
      for(  j = 0; j < Dimension; j++ ){
             if (d[i][j] >  d[i][k] + d[k][j]){
                 d[i][j] = d[i][k] + d[k][j];
      }
   }
}
```

Here is an example (999 is infinite distance in original array)

```
Original distance array
        BAL     BUF     CIN     CLE     DET     NYC     PHI     PIT     WAS
-------------------------------------------
BAL     000     345     999     999     999     999     097     230     039
BUF     345     000     999     186     252     445     365     217     999
CIN     999     999     000     244     265     999     999     284     492
CLE     999     186     244     000     167     507     999     125     999
DET     999     252     265     167     000     999     999     999     999
NYC     999     445     999     507     999     000     092     386     999
PHI     097     365     999     999     999     092     000     305     999
PIT     230     217     284     125     999     386     305     000     231
WAS     039     999     492     999     999     999     999     231     000
```

```
All Pairs Shortest Path array
     BAL    BUF    CIN    CLE    DET    NYC    PHI    PIT    WAS
-------------------------------------------
BAL  000    345    514    355    522    189    097    230    039
BUF  345    000    430    186    252    445    365    217    384
CIN  514    430    000    244    265    670    589    284    492
CLE  355    186    244    000    167    507    430    125    356
DET  522    252    265    167    000    674    597    292    523
NYC  189    445    670    507    674    000    092    386    228
PHI  097    365    589    430    597    092    000    305    136
PIT  230    217    284    125    292    386    305    000    231
WAS  039    384    492    356    523    228    136    231    000
```