CSE 2320 Notes 14: Shortest Paths

(Last updated 11/6/06 5:48 PM)

CLRS, 24.3, 25.2

CONCEPTS

Input:

Directed graph with *non-negative* edge weights (stored as adj. matrix for Floyd-Warshall)
Dijkstra – source vertex

Output:

Dijkstra – tree that gives a shortest path from source to each vertex
Floyd-Warshall – shortest path between each pair of vertices ("all-pairs") as matrix

DIJKSTRA'S ALGORITHM – three versions

Similar to Prim's MST:

S = vertices whose shortest path is known (initially just the source)

Length of path
Predecessor (vertex) on path (AKA shortest path tree)

T = vertices whose shortest path is not known

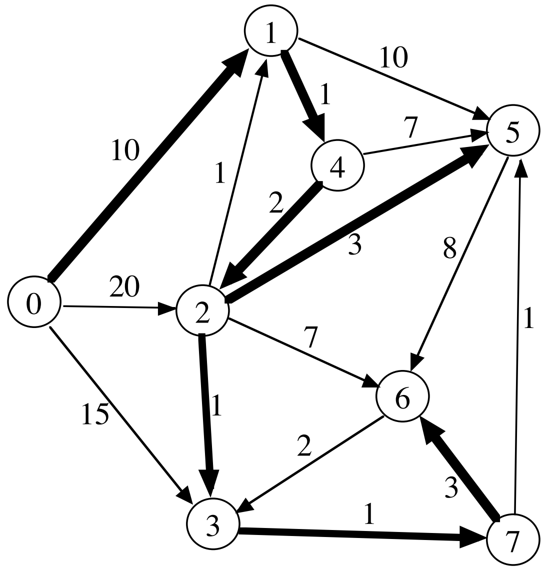
Each phase moves a T vertex to S by virtue of that vertex having the shortest path among all T vertices.

Third version may be viewed as being BFS with the FIFO queue replaced by a priority queue.

1. "Memoryless" – Only saves shortest path tree and current partition. (`dijkstraMemoryless.c`)

```
Place desired source vertex x ∈ V in S
T = V – {x}
x.distance = 0
x.pred = (-1)
while T ≠ ∅
        Find the edge (s, t) over all t ∈ T and all s ∈ S with minimum value for s.distance + weight(s, t)
                (i.e. scan adj. list for each s)
        t.distance = s.distance + weight(s, t)
        t.pred = s
        T = T – {t}
        S = S ∪ {t}
```

Since no substantial data structures are used, this takes $\Theta(EV)$ time.



0	1	2	3	4	5	6	7
0(-)	∞	∞	∞	∞	∞	∞	∞
*	10(0)	20(0)	15(0)				
	*			11(1)	20(1)		
		13(4)		*	18(4)		
		*	14(2)		16(2)	20(2)	
			*				15(3)
						18(7)	*
					*		
						*	

2. Maintains T-table that provides the predecessor vertex in S for each vertex t ∈ T to give the shortest possible path through S to t. (`dijkstraTable.c`)

Eliminates scanning all S adjacency lists in every phase, but still scans the list of the last vertex moved from T to S.

```
Place desired source vertex x ∈ V in S
T = V – {x}
x.distance = 0
x.pred = (-1)
for each t ∈ T
        Initialize t.distance with weight of (x, t) (or ∞ if non-existent) and t.pred = x
while T ≠ ∅
        Scan T entries to find vertex t with minimum value for t.distance
        T = T – {t}
        S = S ∪ {t}
        for each vertex x in adjacency list of t (i.e. (t, x))
                if x ∈ T and t.distance + weight(t, x) < x.distance
                        x.distance = t.distance + weight(t, x)
                        x.pred = t
```

Analysis:

Initializing the T-table takes $\Theta(V)$.
Scans of T-table entries contribute $\Theta(V^2)$.
Traversals of adjacency lists contribute $\Theta(E)$.
$\Theta(V^2 + E)$ overall worst-case.

3. Replace T-table by a heap. (`dijkstraHeap.cpp`, `minHeap.cpp`)

The time for updating distances and predecessors increases, but the time for selection of the next vertex to move from T to S improves.

```
Place desired source vertex x ∈ V in S
T = V – {x}
x.distance = 0
x.pred = (-1)
for each t ∈ T
        Initialize T-heap with weight (as the priority) of (x, t) (or ∞ if non-existent) and t.pred = x
BUILD-MIN-HEAP(T-heap)
while T ≠ ∅
        Use HEAP-EXTRACT-MIN to obtain T-heap entry with minimum t.distance
        T = T – {t}
        S = S ∪ {t}
        for each vertex x in adjacency list of t (i.e. (t, x))
                if x ∈ T and t.distance + weight(t, x) < x.distance
                        x.distance = t.distance + weight(t, x)
                        x.pred = t
                        MIN-HEAP-DECREASE-KEY(T-heap)
```

Analysis:

Initializing the T-heap takes $\Theta(V)$.
Total cost for HEAP-EXTRACT-MINs is $\Theta(V \log V)$.
Traversals of adjacency lists and MIN-HEAP-DECREASE-KEYs contribute $\Theta(E \log V)$.
$\Theta(E \log V)$ overall worst-case, since $E > V$.

Which version is the fastest?

		Sparse $(E = O(V))$	Dense $(E = \Omega(V^2))$
1.	$\Theta(EV)$	$\Theta(V^2)$	$\Theta(V^3)$
2.	$\Theta(V^2 + E)$	$\Theta(V^2)$	$\Theta(V^2)$
3.	$\Theta(E \log V)$	$\Theta(V \log V)$	$\Theta(V^2 \log V)$

FLOYD-WARSHALL ALGORITHM

Based on adjacency matrices. Will examine three versions:

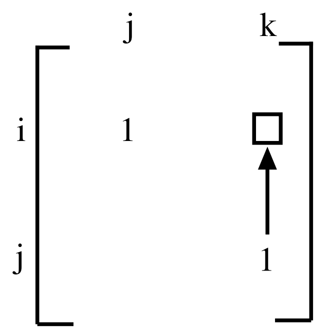
Warshall's Algorithm – After $\Theta(V^3)$ preprocessing, processes each path *existence* query in $\Theta(1)$ time.

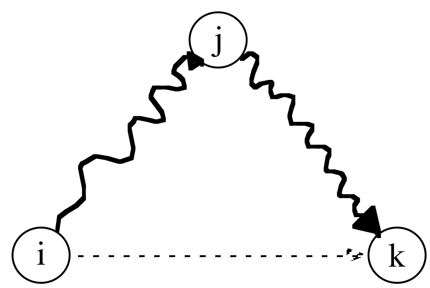
Warshall's Algorithm with Successors - After $\Theta(V^3)$ preprocessing, provides a path in response to a path existence query in $O(V)$ time.

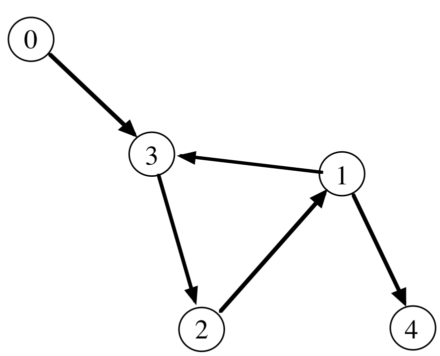
Floyd-Warshall Algorithm (with Successors) - After $\Theta(V^3)$ preprocessing, provides each *shortest* path in $O(V)$ time.

Warshall's Algorithm:

```
for (j=0; j<V; j++)
     for (i=0; i<V; i++)
          if (A[i][j])
               for (k=0; k<V; k++)
                    if (A[j][k])
                         A[i][k]=1;
```







	0	1	2	3	4
0				1	
1				1	1
2		1			
3			1		
4					

If zero-edge paths are useful for an application (i.e. reflexive, self-loops), the diagonal may be all ones.

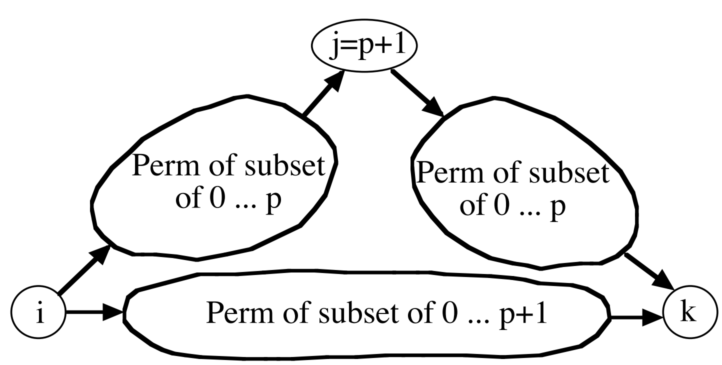
Why does it work?

a. *Correct* in use of transitivity.

b. Is it *complete*?

When	Paths That Can Be Detected
Before j=0	x → y
After j=0	x → 0 → y
After j=1	x → 1 → y x → 0 → 1 → y x → 1 → 0 → y
After j=2	x → 2 → y x → 0 → 2 → y x → 1 → 2 → y x → 2 → 0 → y x → 2 → 1 → y x → 0 → 1 → 2 → y x → 0 → 2 → 1 → y x → 1 → 0 → 2 → y x → 1 → 2 → 0 → y x → 2 → 0 → 1 → y x → 2 → 1 → 0 → y
. . .	
After j=p	x → Permutation of *subset* of 0 … p → y
After j=V-1	ALL PATHS

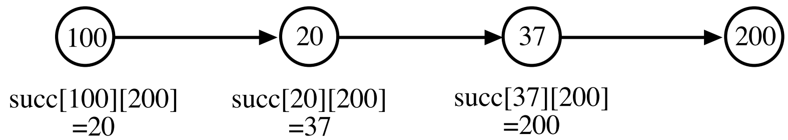
Math. Induction:



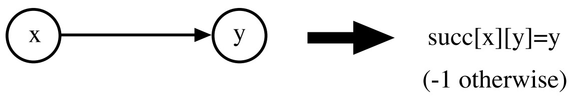
Warshall's Algorithm with Successors

Successor Matrix (CLRS uses predecessor)

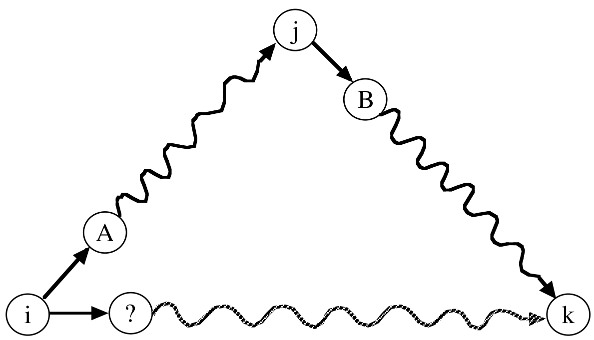
7-11 directions:



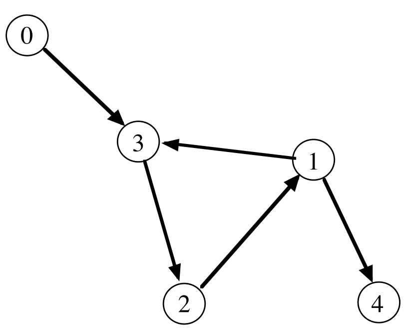
Initialize:



Warshall Matrix Update:



succ[i][j] = A   succ[j][k] = B   succ[i][k] = ?



	0	1	2	3	4
0				3	
1				3	4
2		1			
3			2		
4					

```
for (j=0; j<V; j++)
     for (i=0; i<V; i++)
          if (s[i][j] != (-1))
               for (k=0; k<V; k++)
                    if (succ[i][k]==(-1) && succ[j][k]!=(-1))
                         succ[i][k] = succ[i][j];
```

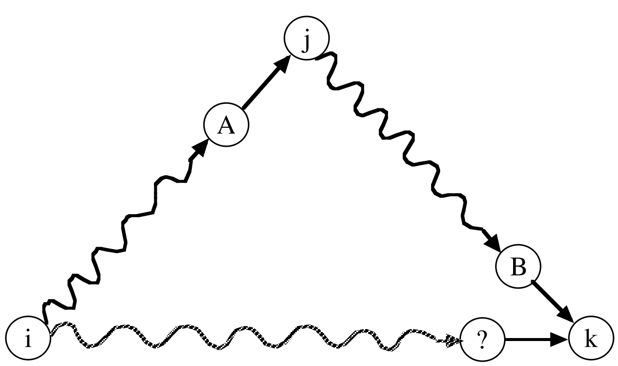
Complete Example (`warshall.c`) saving paths using successors:

```
-1  -1  -1   3  -1
-1  -1  -1   3   4
-1   1  -1  -1  -1
-1  -1   2  -1  -1
-1  -1  -1  -1  -1
-------------------
-1  -1  -1   3  -1
-1  -1  -1   3   4
-1   1  -1  -1  -1
-1  -1   2  -1  -1
-1  -1  -1  -1  -1
-------------------
-1  -1  -1   3  -1
-1  -1  -1   3   4
-1   1  -1   1   1
-1  -1   2  -1  -1
-1  -1  -1  -1  -1
-------------------
```

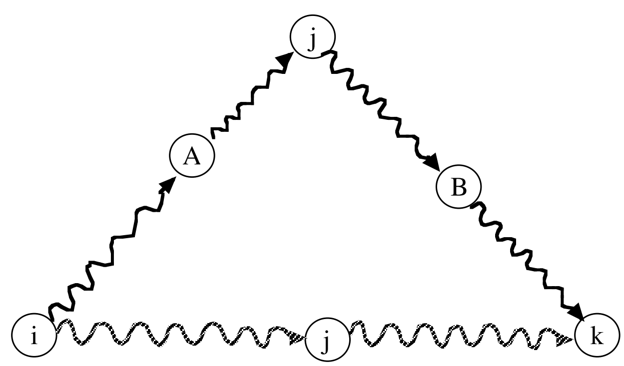
```
-1  -1  -1   3  -1
-1  -1  -1   3   4
-1   1  -1   1   1
-1   2   2   2   2
-1  -1  -1  -1  -1
-------------------
-1   3   3   3   3
-1   3   3   3   4
-1   1   1   1   1
-1   2   2   2   2
-1  -1  -1  -1  -1
-------------------
-1   3   3   3   3
-1   3   3   3   4
-1   1   1   1   1
-1   2   2   2   2
-1  -1  -1  -1  -1
-------------------
```

Other ways to save path information:

Predecessors (`warshallPred.c`)



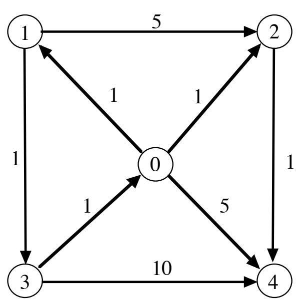
Transitive/Intermediate/Column (`warshallCol.c`)



Floyd-Warshall Algorithm (with Successors)

After j = p has been processed, the *shortest path* from each x to each y that uses *only* vertices in 0 . . . p as intermediate vertices is recorded in matrix.

```
for (j=0; j<V; j++)
     for (i=0; i<V; i++)
          if (dist[i][j] < 999)
               for (k=0; k<V; k++)
               {
                    newDist = dist[i][j] + dist[j][k];
                    if (newDist < dist[i][k])
                    {
                         dist[i][k] = newDist;
                         succ[i][k] = succ[i][j];
                    }
               }
```



	0	1	2	3	4
0		1	1		5
1			5	1	
2					1
3	1				10
4					

```
999 0   1 1   1 2 999 3   5 4 999 0   1 1   1 2   2 1   2 2
999 0 999 1   5 2   1 3 999 4 999 0 999 1   5 2   1 3   6 2
999 0 999 1 999 2 999 3   1 4 999 0 999 1 999 2 999 3   1 4
  1 0 999 1 999 2 999 3  10 4   1 0   2 0   2 0   3 0   3 0
999 0 999 1 999 2 999 3 999 4 999 0 999 1 999 2 999 3 999 4
----------------------------- -----------------------------
999 0   1 1   1 2 999 3   5 4   3 1   1 1   1 2   2 1   2 2
999 0 999 1   5 2   1 3 999 4   2 3   3 3   3 3   1 3   4 3
999 0 999 1 999 2 999 3   1 4 999 0 999 1 999 2 999 3   1 4
  1 0   2 0   2 0 999 3   6 0   1 0   2 0   2 0   3 0   3 0
999 0 999 1 999 2 999 3 999 4 999 0 999 1 999 2 999 3 999 4
----------------------------- -----------------------------
999 0   1 1   1 2   2 1   5 4   3 1   1 1   1 2   2 1   2 2
999 0 999 1   5 2   1 3 999 4   2 3   3 3   3 3   1 3   4 3
999 0 999 1 999 2 999 3   1 4 999 0 999 1 999 2 999 3   1 4
  1 0   2 0   2 0   3 0   6 0   1 0   2 0   2 0   3 0   3 0
999 0 999 1 999 2 999 3 999 4 999 0 999 1 999 2 999 3 999 4
----------------------------- -----------------------------
```

Note: In this example, zero-edge paths are not considered.