COP 3530: Computer Science III
Summer 2005

Graphs – Part 2

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CSB 242, 823-2790

http://www.cs.ucf.edu/courses/cop3530/summer05

School of Computer Science
University of Central Florida

Shortest Path Problems

- In the shortest path problem, the edges of the graph are assigned certain weights. The meaning of the weights will vary from application to application, but common representations are: distance between two cities indicated by the vertices, cost of transmission across this link, amounts of some substance moved across the network., etc.
- When determining the shortest path from vertex *v* to vertex *u*, information about the distances between intermediate vertices *w* must be recorded. This information can be recorded as a label associated with these vertices, where the label is only the distance from *v* to *w* or the distance along with the predecessor of *w* in this path.
- The methods of finding the shortest path rely on these labels. Depending upon how many times these labels are updated, the methods solving the shortest path problem are divided into two classes: label-setting algorithms and label-correcting algorithms.

Shortest Path Problems (cont.)

- For label-setting algorithms, in each pass through the vertices still to be processed, one vertex is set to a value which remains unchanged to the end of the execution.
  - This, however, limits such methods to processing graphs with only positive weights.
- The label-correcting algorithms will allow for the changing of *any* label during the execution of the algorithm.
- Most of the label-setting and label-correcting algorithms can be subsumed to the same form which will allow finding the shortest path from one vertex to all other vertices in the graph.

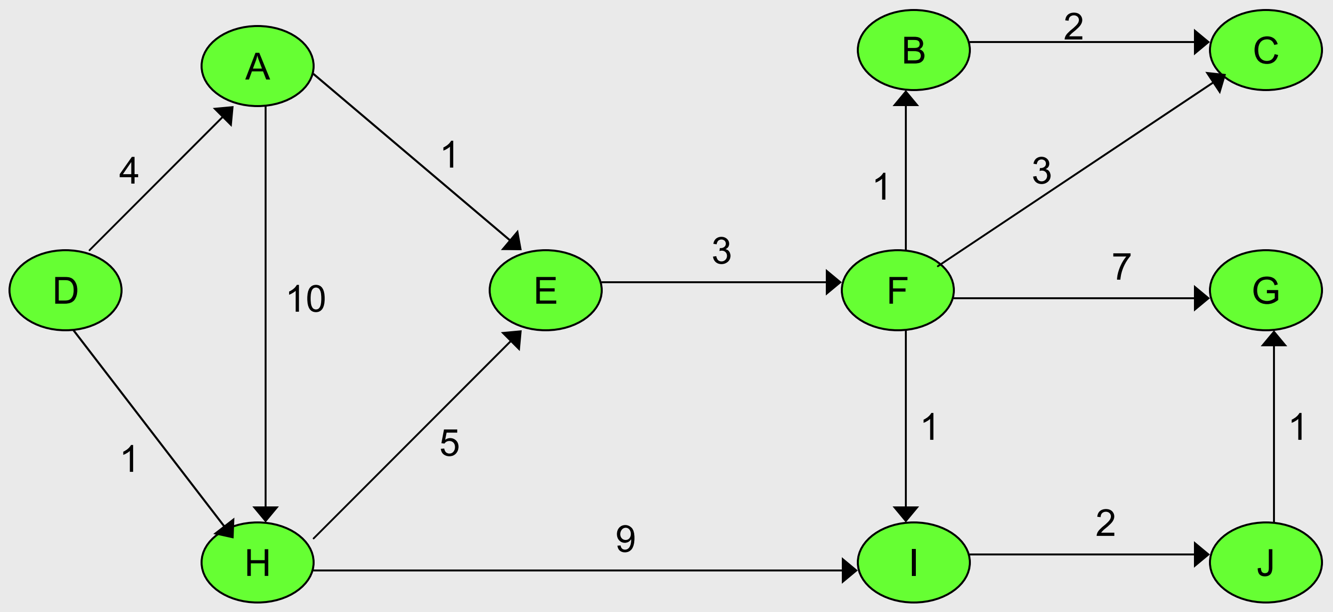
Dijkstra's Label Setting Algorithm

- Dijkstra was one of the first to develop a label-setting algorithm for finding the shortest path in a graph.
- In this algorithm (shown on the next slide) a number of paths p_1, p_2, …, p_n from a vertex v are tried, and each time, the shortest path among them is tried, which may mean that the same path p_i can be continued by adding one more edge to it.
  - If p_i turns out to be longer than any other path that can be tried, p_i is abandoned and this other path is tried by resuming from where it was left and by adding one more edge to it.
- Since paths can lead to vertices with more than one outgoing edge, new paths for possible exploration are added for each outgoing edge. Each vertex is tried once, all paths leading from it are opened, and the vertex itself is put away and not used anymore. After all vertices are visited, the algorithm terminates.

Dijkstra's Label Setting Algorithm

```
Dijkstra (weighted simple digraph, vertex first)
    for all vertices v
        currDist(v) = ∞;
    currDist(first) = 0;
    tobeChecked = all vertices;
    while tobeChecked is not empty
        v = a vertex in tobeChecked with minimal currDist(v);
        remove v from tobeChecked;
        for all vertices u adjacent to v and in tobeChecked
            if currDist(u) > currDist(v) + weight(edge(vu))
                currDist(u) = currDist(v) + weight(edge(vu));
                predecessor(u) = v;
```

Dijkstra's Shortest Path Algorithm



Graph for Djjkstra's Shortest Path Algorithm Example

Initial Table for Dijkstra's Algorithm

- Initially the *currDist(v)* for every vertex in the graph is set to ∞.

- Next the *currDist(start)* is set to 0, where *start* is the initial node for the path. In this example *start* = vertex *D*. The set *tobeChecked* is initialize to contain every vertex in the graph. Since *start* = *D* and *currDist(D)= 0* this vertex will have the minimum *currDist( )* value and thus vertex D will be the first vertex removed from the set *tobeChecked*.

- In the sequence of tables shown on the following slides, the set *tobeChecked* is indicated by the leftmost column with the current members of the set indicated by shading the cells for current members in light blue.

- After this initialization stage the table will look like the one on the next slide.

Dijkstra's Shortest Path Algorithm

iteration →	initial	1	2	3	4	5	6	7	8	9	10
active vertex ↓	■										
A	∞										
B	∞										
C	∞										
D	0										
E	∞										
F	∞										
G	∞										
H	∞										
I	∞										
J	∞										

First Iteration of Dijkstra's Algorithm

- The first iteration of the algorithm will remove the vertex with the minimum *currDist( )* which will be vertex D and then set the *currDist( )* for every vertex which is both adjacent to D and in *tobeChecked*.
- In this case, only vertices A and H are both adjacent to D and in *tobeChecked*.
  - The value of *currDist(A)* = *currDist(D)* + *weight(edge(DA))* = *0* + *4* = *4*.
  - The value of *currDist(H)* = *currDist(D)* + *weight(edge(DH))* = *0* + *1* = *1*.
- After the first iteration the table will look like the table shown in the next slide:

Dijkstra's Shortest Path Algorithm

iteration →	initial	1	2	3	4	5	6	7	8	9	10
active vertex ↓	■	D									
A	∞	4									
B	∞	∞									
C	∞	∞									
D	0										
E	∞	∞									
F	∞	∞									
G	∞	∞									
H	∞	1									
I	∞	∞									
J	∞	∞									

Second Iteration of Dijkstra's Algorithm

- Notice that when a vertex is removed from the set *tobeChecked* it is no longer participating in setting the values in the table so its row is unused after its removal from the set.

- The second iteration will again selected the minimum value of *currDist( )* from the vertices in *tobeChecked*. In this case the vertex with this minimum value is vertex H since *currDist(H) = 1* and *currDist(A) = 4*. So vertex H is removed from the set *tobeChecked* and the active vertex is set to H. Vertices which are both adjacent to H and in *tobeChecked* are vertices E and I.
  - The value of *currDist(E) = currDist(H) + weight(edge(HE)) = 1 + 5 = 6*.
  - The value of *currDist(I) = currDist(H) + weight(edge(HI)) = 1 + 9 = 10*.

- After the second iteration the table looks like the one in the next slide.

Dijkstra’s Shortest Path Algorithm

iteration →	initial	1	2	3	4	5	6	7	8	9	10
active vertex ↓		D	H								
A	∞	4	4								
B	∞	∞	∞								
C	∞	∞	∞								
D	0										
E	∞	∞	6								
F	∞	∞	∞								
G	∞	∞	∞								
H	∞	1									
I	∞	∞	10								
J	∞	∞	∞								

Third and Fourth Iterations

- The third iteration will select vertex A as it has the minimum weight for all of the vertices in *tobeChecked( )*. So the next active vertex becomes vertex A. See slide 14.

- Notice in the third iteration with active vertex A, the only vertex adjacent to A which has not been visited previously is vertex E. The value of *currDist(E)* is set to 5 during this iteration. See slide 15.

- The fourth iteration (see slide 15) will select vertex E to be the active vertex and remove it from the set *tobeChecked*. The only vertex adjacent to vertex E which has not yet been visited is vertex F.

Dijkstra's Shortest Path Algorithm

iteration →	initial	1	2	3	4	5	6	7	8	9	10
active vertex ↓		D	H	A							
A	∞	4	4								
B	∞	∞	∞	∞							
C	∞	∞	∞	∞							
D	0										
E	∞	∞	6	5							
F	∞	∞	∞	∞							
G	∞	∞	∞	∞							
H	∞	1									
I	∞	∞	10	10							
J	∞	∞	∞	∞							

Dijkstra’s Shortest Path Algorithm

iteration →	initial	1	2	3	4	5	6	7	8	9	10
active vertex ↓		D	H	A	E						
A	∞	4	4								
B	∞	∞	∞	∞	∞						
C	∞	∞	∞	∞	∞						
D	0										
E	∞	∞	6	5							
F	∞	∞	∞	∞	8						
G	∞	∞	∞	∞	∞						
H	∞	1									
I	∞	∞	10	10	10						
J	∞	∞	∞	∞	∞						

Fifth Iteration

- The fifth iteration will select vertex F with minimum value of 8.
- The fifth iteration is shown in the next slide.

Dijkstra’s Shortest Path Algorithm

iteration →	initial	1	2	3	4	5	6	7	8	9	10
active vertex ↓		D	H	A	E	F					
A	∞	4	4								
B	∞	∞	∞	∞	∞	9					
C	∞	∞	∞	∞	∞	11					
D	0										
E	∞	∞	6	5							
F	∞	∞	∞	∞	8						
G	∞	∞	∞	∞	∞	15					
H	∞	1									
I	∞	∞	10	10	10	9					
J	∞	∞	∞	∞	∞	∞					

Sixth Iteration

- The sixth iteration will find two vertices with equal values as the minimum *currDist( )* (both vertex B and I have values of 9).
- Which vertex is selected as the active vertex in this case is arbitrary.
- In this example, Vertex B has been selected as the next active vertex.
- Only vertex *C* is adjacent to vertex B and unvisited. Only the *currDist(c)* will change during the sixth iteration.
- Upon completion of the sixth iteration the only unvisited vertices are C, G, I, and J.
- The sixth iteration is shown in the next slide.

Dijkstra's Shortest Path Algorithm

iteration →	initial	1	2	3	4	5	6	7	8	9	10
active vertex ↓		D	H	A	E	F	B				
A	∞	4	4								
B	∞	∞	∞	∞	∞	9					
C	∞	∞	∞	∞	∞	11	11				
D	0										
E	∞	∞	6	5							
F	∞	∞	∞	∞	8						
G	∞	∞	∞	∞	∞	15	15				
H	∞	1									
I	∞	∞	10	10	10	9	9				
J	∞	∞	∞	∞	∞	∞					

Seventh and Eighth Iterations

- The seventh iteration will select vertex I as the active vertex. Only vertex J is adjacent to vertex I.
- Iteration seven is illustrated in slide 21.
- The eighth iteration will select vertex C or vertex J arbitrarily, for this example vertex *C* has been selected.
- The eighth iteration is shown in slide 22.
- Notice in the eighth iteration that vertex C has no adjacent vertices and thus no values in the table are set, however, vertex C is removed from the set *tobeChecked*.

Dijkstra's Shortest Path Algorithm

iteration →	initial	1	2	3	4	5	6	7	8	9	10
active vertex ↓		D	H	A	E	F	B				
A	∞	4	4								
B	∞	∞	∞	∞	∞	9					
C	∞	∞	∞	∞	∞	11	11				
D	0										
E	∞	∞	6	5							
F	∞	∞	∞	∞	8						
G	∞	∞	∞	∞	∞	15	15				
H	∞	1									
I	∞	∞	10	10	10	9	9				
J	∞	∞	∞	∞	∞	∞					

Dijkstra’s Shortest Path Algorithm

iteration →	initial	1	2	3	4	5	6	7	8	9	10
active vertex ↓		D	H	A	E	F	B	I			
A	∞	4	4								
B	∞	∞	∞	∞	∞	9					
C	∞	∞	∞	∞	∞	11	11	11			
D	0										
E	∞	∞	6	5							
F	∞	∞	∞	∞	8						
G	∞	∞	∞	∞	∞	15	15	15			
H	∞	1									
I	∞	∞	10	10	10	9	9				
J	∞	∞	∞	∞	∞	∞		11			

Dijkstra's Shortest Path Algorithm

iteration →	initial	1	2	3	4	5	6	7	8	9	10
active vertex ↓		D	H	A	E	F	B	I			
A	∞	4	4								
B	∞	∞	∞	∞	∞	9					
C	∞	∞	∞	∞	∞	11	11	11			
D	0										
E	∞	∞	6	5							
F	∞	∞	∞	∞	8						
G	∞	∞	∞	∞	∞	15	15	15			
H	∞	1									
I	∞	∞	10	10	10	9	9				
J	∞	∞	∞	∞	∞	∞	∞	11			

Dijkstra’s Shortest Path Algorithm

iteration →	initial	1	2	3	4	5	6	7	8	9	10
active vertex ↓		D	H	A	E	F	B	I	C		
A	∞	4	4								
B	∞	∞	∞	∞	∞	9					
C	∞	∞	∞	∞	∞	11	11	11			
D	0										
E	∞	∞	6	5							
F	∞	∞	∞	∞	8						
G	∞	∞	∞	∞	∞	15	15	15	15		
H	∞	1									
I	∞	∞	10	10	10	9	9				
J	∞	∞	∞	∞	∞	∞	∞	11	11		

Ninth Iteration

- The ninth iteration will select vertex J as the active vertex. Only vertex G is both adjacent to vertex J and unvisited (i.e., still in the set *tobeChecked*).
- The ninth iteration is illustrated in slide 26.

Dijkstra's Shortest Path Algorithm

iteration →	initial	1	2	3	4	5	6	7	8	9	10
active vertex ↓		D	H	A	E	F	B	I	C	J	
A	∞	4	4								
B	∞	∞	∞	∞	∞	9					
C	∞	∞	∞	∞	∞	11	11	11			
D	0										
E	∞	∞	6	5							
F	∞	∞	∞	∞	8						
G	∞	∞	∞	∞	∞	15	15	15	15	12	
H	∞	1									
I	∞	∞	10	10	10	9	9				
J	∞	∞	∞	∞	∞	∞	∞	11	11		

Tenth and Final Iteration

- The tenth and final iteration (there are only ten vertices in the original graph) serves only to remove the vertex G from the set *tobeChecked.*
- The final table is exactly the same as the previous table expect that the set *tobeChecked* is now empty and thus the algorithm will terminate.
- The final iteration is shown in the next slide.

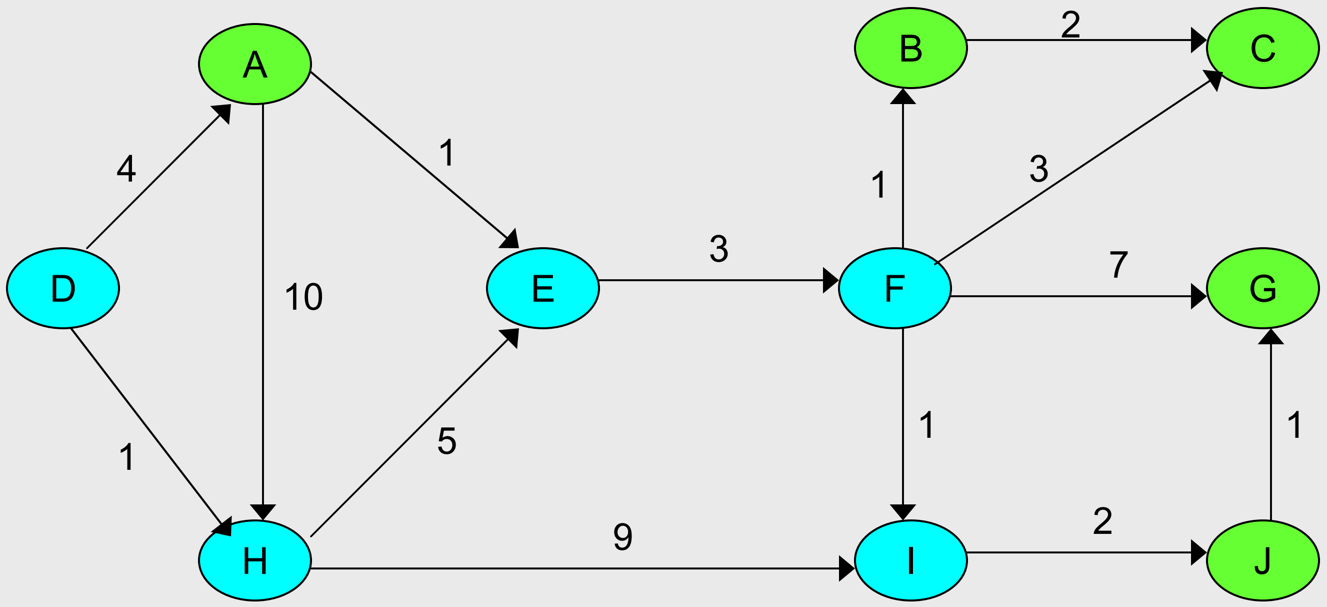
Dijkstra's Shortest Path Algorithm

iteration →	initial	1	2	3	4	5	6	7	8	9	10
active vertex ↓		D	H	A	E	F	B	I	C	J	G
A	∞	4	4								
B	∞	∞	∞	∞	∞	9					
C	∞	∞	∞	∞	∞	11	11	11			
D	0										
E	∞	∞	6	5							
F	∞	∞	∞	∞	8						
G	∞	∞	∞	∞	∞	15	15	15	15	12	
H	∞	1									
I	∞	∞	10	10	10	9	9				
J	∞	∞	∞	∞	∞	∞	∞	11	11		

Reading the Solution to Dijkstra's Shortest Path Example

- The results of Dijkstra's Shortest Path algorithm applied to our example are embedded in the table.
- The highlighted cells for each vertex represent the length of the shortest path from the start vertex D to the vertex identified by each row.
- Shortest Paths are:
  - D to A = 4 D to B = 9 D to C = 11
  - D to E = 8 D to F = 9 D to G = 12
  - D to H = 1 D to I = 11 D to J = 11

Dijkstra's Shortest Path Algorithm



Graph for Djjkstra's Shortest Path Algorithm Example

Shortest Path from D to I identified by blue nodes

Comments on Dijkstra’s Shortest Path Algorithm

- Although Dijkstra’s algorithm is quite efficient when dealing with graphs which contain only positive weights.
- Although many graphs contain only positive weights, it is also possible for them to contain negative weights.
- Shortest path algorithms for graphs containing negative weights are, in general, more robust and have less efficient execution (higher overhead for handling the negative weights) when dealing with graphs that contain only positive weights.
- Therefore, Dijkstra’s algorithm is very popular for positive weighted graphs, however, Dijkstra’s algorithm is not general enough, and will fail when negative weights are used in the graph.

Comments on Dijkstra’s Shortest Path Algorithm (cont.)

- To see why, change the weight of *edge*(*ah*) from 10 to –10.
- Note that the path *D*, *A*, *H*, *E* is now –1, whereas the path *D*, *A*, *E* as determined by the algorithm is 5.
- The reason for overlooking this less costly path is that the vertices with the current distance set from ∞ to a value are not checked anymore (remember it’s a label-setting algorithm): First successors of vertex D are checked and D is removed from *tobeChecked*, then vertex H is removed from *tobeChecked*, and only afterward is the vertex A considered to be a candidate to be included in the path from D to other vertices. But now, *edge*(AH) is not taken into consideration because the condition in the for loop prevents the algorithm from doing so. To overcome this limitation, a label-correcting algorithm is required.