CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 23

May 1, 2014

Positive-Weighted, Shortest-path Problem

- Many times the edges have weights – indicating that the relations between nodes are not always equal.
- Example – travel distance between cities.
- In this case the algorithm from last class wouldn't work.
- Goal: Find the shortest path (measured by total cost) from a designated vertex S to every vertex in a weighted graph.
- The weights (costs) are assigned to edges.
- All edge costs are positive.

Dijkstra's Algorithm

- A greedy technique that actually works well
- Starting from a certain vertex v_0, we set up a Map to hold all the running min costs D_i from v_0 to various destinations v.
- Like the unweighted case, we have an "eyeball" that indicates what node we are visiting.
- We move the eyeball to a new node v, the one with the least cost from v_0 that has not been visited by the eyeball.

Dijkstra's Algorithm

- We compute the min cost D_w from v_0 to various destinations w following any path through the visited node v
  - $D_w = \min\{D_w, D_v + c_{vw}\}$
  - c_{vw} = cost for edge $v \rightarrow w$ (these are all constant positive numbers)
- Repeat by moving the eyeball to another node which has the lowest cost and has not been visited.

Adjust D_w

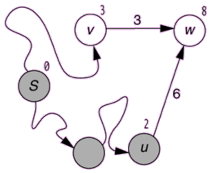
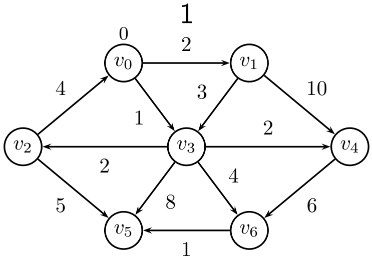


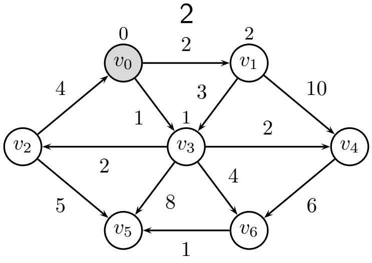
figure 14.23

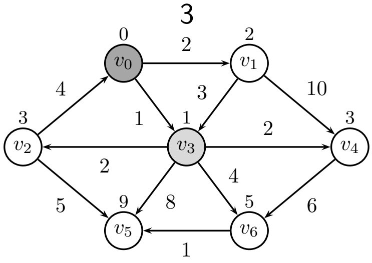
The eyeball is at v and w is adjacent, so D_w should be lowered to 6.

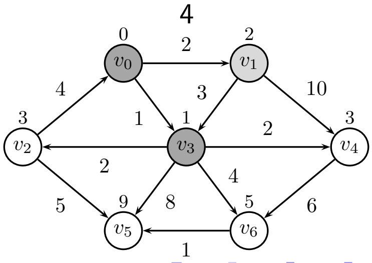
The eyeball is at v and w is adjacent to v, so D_w should be lowered from 8 to 6.

Running Example, Starting From V_0

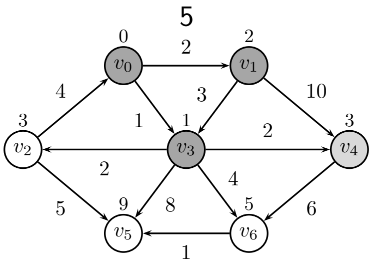


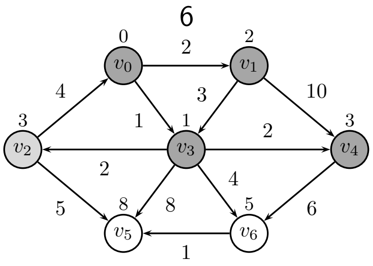


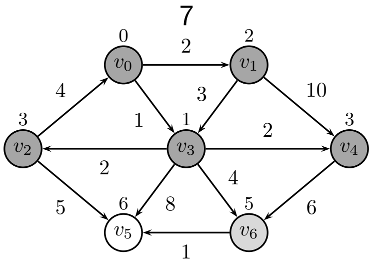


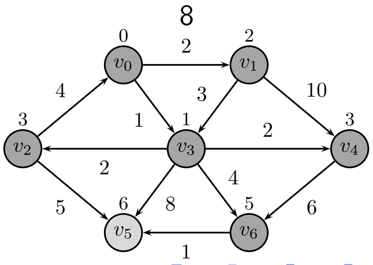


Running Example, Starting From V_0

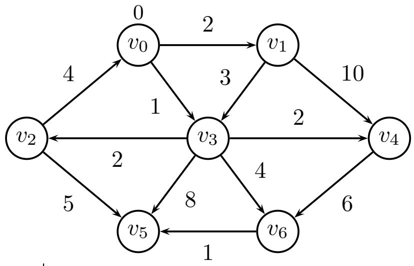








Path with Minimum Weights from V_0



eyeball	D_0	D_1	D_2	D_3	D_4	D_5	D_6
–	0	∞	∞	∞	∞	∞	∞
V_0	—	2	∞	1	∞	∞	∞
V_3	—	2	3	—	3	9	5
V_1	—	—	3	—	3	9	5
V_4	—	—	3	—	—	9	5
V_2	—	—	—	—	—	8	5
V_6	—	—	—	—	—	6	—
V_5	—	—	—	—	—	—	—
final:	0	2	3	1	3	6	5

Pseudocode for Dijkstra's Algorithm

```
 Given a directed graph G and source node S.
Set up storage for a distance for each node, $D_i$:
    For each node i, set $D_i = c_{S,i}$
    (cost of the edge from S to i)
Set up a data structure P for unvisited nodes and put all
nodes except S in it.
Loop while P is not empty
    Choose a node v in P with minimum $D_i$
    Delete v from P
    (it's now visited, and now the ``eyeball'' node)
    Loop though nodes w adjacent to v and in P
        $D_w = min(D_w, D_v + c_{vw})$
```

Implementation of Dijkstra

- Problem - the D's change during the running of the algorithm, so a simple priority queue would seem to be useless.
- Weiss has a trick to allow us to use a simple priority queue even so.
- Finding the path can be done with the same trick as the simpler case.
- Every time you find a new better final edge for a path, you save the node it came from as "prev".

PQ for Elements That Change Priorities

- A trick that works in general for objects whose priority changes occasionally.
- Given an example of support messages with priorities. Now we add a valid field in the objects.
- PQ = {(Joe, pri 3, valid), (Sue, pri 4, valid), (Tom, pri 5, valid)}
- A manager has a ref to Sues message, wants to change it to pri 1, to be processed next. Just make a copy of it, change priority in the copy and set valid to invalid in the old one:
- PQ = {(Sue, pri 1, valid), (Joe, pri 3, valid), (Sue, pri 4, invalid), (Tom, pri 5, valid)}

PQ for Elements That Change Priorities

- This is perfectly proper use of the ordinary priority queue.
- We can't change the priority of an element in a simple PQ, but nothing is stopping us from changing other fields.
- When we deleteMin a message that's marked invalid, we just throw it away and deleteMin again.

PQ for Elements That Change Priorities

- Of course we are carrying around extra baggage in the PQ, so this is only a good solution for occasional changes in working priority.
- It works particularly well for changes that lower the priority number, because the new valid entry then always precedes the newly-invalid entry, so the invalid ones aren't a surprise when they come off the PQ.
- This is the case in Dijkstra's algorithm.

Trace of Dijkstra's algorithm With PQ Shown

eyeball	D_0	D_1	D_2	D_3	D_4	D_5	D_6	PQ
–	0	∞	∞	∞	∞	∞	∞	$\{(V_0, 0)\}$
V_0	-	2	∞	1	∞	∞	∞	$\{(V_3, 1), (V_1, 2)\}$
V_3	-	2	3	-	3	9	5	$\{(V_1, 2), (V_4, 3), (V_2, 3), (V_6, 5), (V_5, 9)\}$
V_1	-	-	3	-	3	9	5	$\{(V_4, 3), (V_2, 3), (V_6, 5), (V_5, 9)\}$
V_4	-	-	3	-	-	9	5	$\{(V_2, 3), (V_6, 5), (V_5, 9)\}$
V_2	-	-	-	-	-	8	5	$\{(V_6, 5), (V_5, 8), (V_5, 9, \mathit{invalid})\}$
V_6	-	-	-	-	-	6	-	$\{(V_5, 6), (V_5, 8, \mathit{invalid}), (V_5, 9, \mathit{invalid})\}$
V_5	-	-	-	-	-	-	-	$\{(V_5, 8, \mathit{invalid}), (V_5, 9, \mathit{invalid})\}$
final:	0	2	3	1	3	6	5	

Trace of Dijkstra's algorithm

- We see that the valid entries always have lower priority numbers than the corresponding invalid ones, so they are always deleteMin'ed out ahead of the invalid ones.
- However, it is possible to have some invalid ones preceding another node's valid entry, like this:

```
{(V5,6),(V5,8,invalid),(V5,9,invalid),(V7,11,valid),(V7,13,invalid)}
<---V5 group, valid one first-----> <---V7 group, valid one first--->
```

Weiss Implementation

```
public void dijkstra( String startName )
{
    PriorityQueue<Path> pq = new PriorityQueue<Path>( );

    Vertex start = vertexMap.get( startName );
    if( start == null )
        throw new NoSuchElementException( "Start vertex not found" );

    clearAll( );
    pq.add( new Path( start, 0 ) ); start.dist = 0;

    int nodesSeen = 0;
    while( !pq.isEmpty( ) && nodesSeen < vertexMap.size( ) )
    {
        Path vrec = pq.remove( );
        Vertex v = vrec.dest;
        if( v.scratch != 0 )  // already processed v
            continue;

        v.scratch = 1;
        nodesSeen++;

        for( Edge e : v.adj )
        {
            Vertex w = e.dest;
            double cvw = e.cost;

            if( cvw < 0 )
                throw new GraphException( "Graph has negative edges" );

            if( w.dist > v.dist + cvw )
            {
                w.dist = v.dist + cvw;
                w.prev = v;
                pq.add( new Path( w, w.dist ) );
            }
        }
    }
}
```

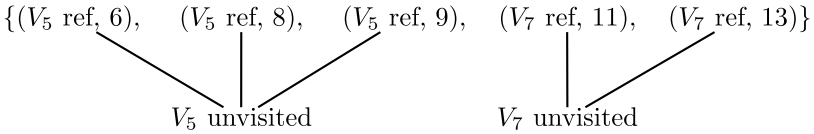
Invalid Entries

- Weiss's code simply counts the nodes as they are processed as the eyeball nodes, and terminates the loop when $|V|$ are seen this way.
- The code simply never gets to the invalid entries in the case traced above.
- What if there are invalid ones preceding another valid one, like the V_5, V_7 PQ shown above, in Weiss's code?
- He uses his per-Vertex "scratch" field to keep track of whether or not a node has been processed as an eyeball node.

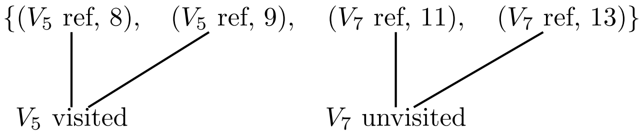
Invalid Entries

- Since valid entries precede invalid ones, scratch gets set to 1 for the node while processing the valid entry, and is tested to eliminate the invalid entries trailing behind the valid one in the PQ (the valid and invalid PQ entries both point to the same Vertex object, with its scratch field).
- Using this approach, the PQ entries themselves don't have to be marked invalid!

Invalid Entries – Example



deletemin V_5, and it becomes visited



Invalid Entries – Example

- Now the other V_5 entries are ignored when deletemin'd, based on scratch=1, i.e, the node has been visited.
- Note that the success of this trick depends on the fact that in Dijkstra's algorithm, priority values for the same key are only decreased, not increased.

Implementation of Dijkstra's

- How should we implement Dijkstra's algorithm with JGraphT graphs?
- We need a Map (something like the Map from V to Unweighted.DistInfo in cs310.Unweighted) to handle the information held in Weiss's "scratch" spot, v.scratch, as well as v.dist and v.prev.
- But don't use the name scratch. This is a flag for the vertex having been processed, and deserves a name that says that.

Implementation of Dijkstra's

- The Path class on pg. 550 is usable as is.
- We could consider adding a valid flag to the PQ entries, to show the full power of the invalidation trick.
- However, it's not that easy to locate the PQ entry when we want to invalidate it.
- We don't want to have to scan the whole PQ looking for it – that would ruin our performance.
- So we put a PQ entry ref in the Map value for the vertex, and this is a lot like having the status itself in the Map value, but more complicated.

Implementation of Dijkstra's

- Thus the simplest way is to follow Weiss's approach, use his Path class as is, and keep all the status information by vertex.
- After vertex is visited, the rest of its PQ entries are ignored when deletemin'd from the PQ.
- Then a lot of the code on pg. 551 can be used as is, but some details change.

Recovering the Paths

- As in the unweighted algorithm, we are recording the “prev” node as we go along.
- So we can work backwards from the various endpoints to the start node.
- Marking improvements to D, including infinity to finite: The * in a column marks the point where prev is last set for that vertex, recorded below.

Recovering the Paths

eyeball	D_0	D_1	D_2	D_3	D_4	D_5	D_6
–	0	∞	∞	∞	∞	∞	∞
V_0	—	2*	∞	1*	∞	∞	∞
V_3	—	2	3*	—	3	9	5*
V_1	—	—	3	—	3	9	5
V_4	—	—	3	—	—	9	5
V_2	—	—	—	—	—	8	5
V_6	—	—	—	—	—	6*	—
V_5	—	—	—	—	—	—	—
final:	0	2	3	1	3	6	5
prev:	–	V_0	V_3	V_0	V_3	V_6	V_3

- So the path for V_1 is $V_0 \to V_1$, for V_2 is $V_0 \to V_3 \to V_2$, and so on.
- In pa6, there is a call for each such path.

Runtime for Dijkstra's

- Notation: for a set S, $|S|$ can be used to stand for the number of elements in S.
- A graph consists of a node set V and an edge set E, of sizes $|V|$ and $|E|$.
- As discussed on pg. 550, the performance of this algorithm is dominated by $|E|$ PQ insertions and deletions, each $O(log|E|)$, so a total of $O(|E|log|E|) = O(|E|log|V|)$, much better for sparse graphs than $O(|V|^2)$, and the nearly the same for dense graphs.

Shortest Paths With Negative Costs

- Dijkstra assumes no negative edge costs.
- In fact, it may not work with even a single negative edge. Why?
- A whole cycle that gives a negative cost makes it impossible to determine best paths for all cases.
- Why?
- It turns out a few negative edges can be tolerated and the shortest paths found using another algorithm known as the Bellman-Ford algorithm.

Bellman-Ford's algorithm

- Naturally it takes additional computation to compensate for the possibility of negative costs, so this algorithm does not perform as well as Dijkstra.
- But it is actually easier to implement, because it only uses a simple queue, not a priority queue.
- Every time a D is reduced, the target node is scheduled for future exploration by being put back on the queue, unless it's there already.

Bellman-Ford Pseudocode

```
 Given a directed graph G and source node S.
Set up a queue for a distance for each node, Di:
For each node i, make Di be INFINITY
Made Ds be 0
Enqueue S
Loop while queue is not empty
  Dequeue a node, v.
  Loop though nodes w adjacent to v.
      $D_w = min(D_w, D_v + c_{vw})$
      If v decreased w's weight and w is not in queue
          enqueue w.
```

Weiss' Implementation

The "scratch" field is used like this:

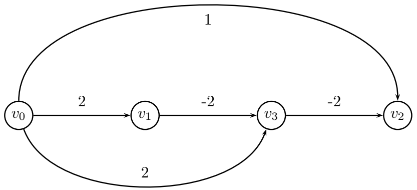
- scratch $=$ 0: not yet seen, not on q
- scratch $=$ 1: seen once as w or start node, on q
- scratch $=$ 2: visited once as eyeball, not on q
- scratch $=$ 3: visited once as eyeball, then later enqueued as w, on q
- scratch $=$ 4: visited twice as eyeball, not on q
- ...
- scratch $=$ odd number iff node is on q, and scratch/2 $=$ number of times it has been an eyeball.

The rule is that it should be done before being an eyeball $|V|$ times, so if this number gets too high, it means a negative cycle was found in the graph.

Example

C_{ij}:

$$\begin{array}{ccccc} & V_0 & V_1 & V_2 & V_3 \\ V_0 & \infty & 2 & 1 & 2 \\ V_1 & \infty & \infty & \infty & -2 \\ V_2 & \infty & \infty & \infty & \infty \\ V_3 & \infty & \infty & -2 & \infty \end{array}$$



- Although locally it looks good get to V_2 from V_0 at cost 1, there is a better path $V_0 \to V_1 \to V_3 \to V_2$ because of the negative cost links.
- The algorithm needs to keep going whenever it sees a D being reduced, revisiting that seen node.

Example

- Notation: $V_0(1,0)$: status=scratch=1, D=0
- Init: $q = [V_0(1,0)]$
- Dequeue V_0, now $V_0(2,0)$, look at neighbors, enqueue them: $q = [V_1(1,2), V_2(1,1), V_3(1,2)]$
- Dequeue V_1, now $V_1(2,2)$, look at neighbors, just $V_3(1,2)$, but now it has better D=0 via V_1,
- so would enqueue V_3 except that it's on the queue, as $V_3(1,0) : q = [V_2(1,1), V_3(1,0)]$

Example

- Dequeue V_2, now $V_2(2,1)$, look at neighbors, none, $q = [V_3(1,0)]$
- Dequeue V_3, now $V_3(2,0)$, look at neighbors, just V_2, has better D=-2 via V_3, enqueue V_2, $q = [V_2(3,-2)]$
- Dequeue V_2, now $V_2(4,-2)$, look at neighbors, none, q $=$ [] so DONE.

Example

- Result: look for last $V_i(x, D)$'s: $V_0(D = 0)$, $V_1(D = 2)$, $V_2(D = -2)$, $V_3(D = 0)$.
- An implementation with JGraphT would have a Map from V to DistInfo with status (instead of scratch), D, and prev for paths, as well as a Queue as in Weisss code on pg. 554.
- Runtime?