CSE326 Winter 2004
Dry 5 Solutions

Problem 1.
Following the exact notation presented in lectures, here is the table of distances $\lambda(v)$ at all stages of Dijsktra's algorithm on the graph represented in the given adjacency matrix M:

	init	u=S	u=B	u=A	u=E	u=C	u=D
S	0	0*	0*	0*	0*	0*	0*
A	∞	3	3	3*	3*	3*	3*
B	∞	1	1*	1*	1*	1*	1*
C	∞	∞	6	5	5	5*	5*
D	∞	8	8	8	8	7	7*
E	∞	∞	3	3	3*	3*	3*

* non-T vertices.

Problem 2.

a. Dijkstra's algorithm only finds one shortest path, not all. A naïve approach to this problem (although one that would work, albeit too inefficiently) is to find all shortest paths from *s* to *t*, perhaps after sifting through all possible paths between these two vertices, and checking if each shortest path contains the given edge *e*. The exhaustive search this requires makes this idea impractical for all graphs with more than a handful of vertices. A faster solution compares the shortest *s*-*t* path found in *G* with the path found with *e* removed, *G*-*e*:

run Djikstra(G,s); let d = (s,t) distance
run Djikstra(G-e,s); let d' = (s,t) distance
return "e is in all shortest paths" iff d' ≠ d

If there is a shortest *s*-*t* path in *G* not containing *e*, then removing *e* will not change the *s*-*t* distance. Conversely, if removing *e* does not change the length of the shortest *s*-*t* path, then there exists some path not containing *e* that is a shortest path.

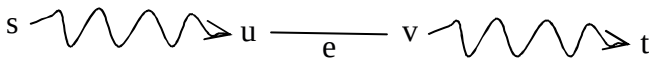
The running time for this algorithm is twice that of running Djikstra's, plus the time it takes to compute *G*-*e*. For any reasonable representation of a graph, computing *G*-*e* is faster than running Djikstra's, so the overall running time is $O((n+m)\log n)$.

A similar idea is to run Djikstra's on G, then increase the weight of e, re-run Djikstra's, and see if the shortest path length changes.

b. We can use a similar idea to see if *e* is in some shortest *s*-*t* path. Because we know no shortest path in a graph with nonnegative weights will contain cycles, we'll look at shortest paths from the endpoints of *e*, in *G*-*e*. Note that we have to compute *G*-*e* to be sure that *e* doesn't creep into the paths somehow and spoil the result (for example, if *e* has weight 0).

Let e = (u,v)
run Djikstra(G,s), let d = (s,t) distance
run Djikstra(G-e,s), let a = (s,u) distance and a' = (s,v) distance
run Djikstra(G-e,v), let b = (v,t) distance
run Djikstra(G-e,u), let b'=(u,t) distance
return "e is in some shortest path" iff a+b+w(e) = d or a'+b'+w(e) = d

The gist of the algorithm is to compare look at the following path:



e is in some shortest *s*-*t* path iff the shortest paths from *s*-*u* and *v*-*t* and the weight of *e* sum up to the shortest *s*-*t* path length. However, we don't know if *u* or *v* comes first in this picture, so we have to try both ways.

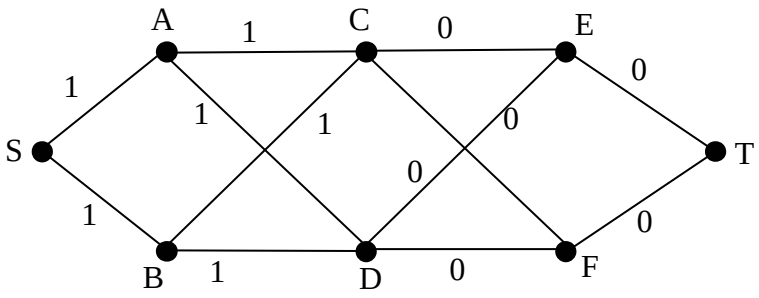
Note that, unlike part **a**, we can't reduce the weight of *e*, and see if the shortest path length changes, because if *w(e)*=0 to begin with, we'll make the weight negative, and Djikstra's won't produce the correct answer.

Comments:
The grading for this question was as follows. A correct, fast algorithm received the full 15 points. A correct, slow algorithm received 10 points. An incorrect algorithm received less than 10 points---for example, the variation below received 8.

Many people tried a variation on Djikstra's where a flag, "saw-e" is associated with each vertex. When a vertex is updated along an edge to a previous vertex, that vertex's "saw-e" flag is passed on to the vertex:

```
…. u has lam(u) minimum
For each v adjacent to u
If (lam(v) > lam(u) + w(u,v))
  lam(v) = lam(u)+w(u,v)
  saw-e(v) = saw-e(u) or (u,v) == e
```

etc, where the "or" is replaced by "and" depending on part (a) or (b). The problem with algorithms of this type is shown in the following graph.



All the edges on the left have weight one, as marked; the edges on the right have weight zero. Suppose the edge we're interested in is (B,C). Djikstra's algorithm will label C and D with lamda=2; only C will have seen (B,C). If we now explore E,F,T from D and only then look at the neighbors of C, T will be marked as not having seen (B,C), and while E and F will be updated when C is explored, T will not be. But, there clearly are shortest paths to T involving (B,C).

For part (a), remove node A. After marking C and D, we could then explore from C, marking E, F and T having seen (B,C) on all paths. Then we explore from D, marking E and F as not having seen (B,C), but again T will not be updated.

This example hinges on two things: (i) edges of weight zero can do nonintuitive things, and (ii) when there are multiple minimums we don't have control over which gets used first. This technique might have been extended to work if each time we changed a node's saw-e flag, we explored all neighbors of that node; but then our running time would be greater than Djikstra's. This technique might also work if instead of just choosing the minimum lambda arbitrarily, we broke ties by counting the number of edge (without weighting) of nodes from S. But nobody (that I saw) tried that.

Some other people is thinking of reducing the weight of e to detect if e is in some of the shortest path. This is a feasible method, except there is a special case that there could exist negative cycle after reduction of the weight of e. So we must make sure the reduction amount should not cause negative cycle.
If c(e)>0, it is easy that reduce the c(e) but keep positive.
If c(e)=0, and if it is the only edge of weight zero, then you just reduce the weight by an amount less than the smallest weight in the graph without e so that the negative amount of e will not exceed the other weights.
If c(e)=0, and there are other edges with zero weights, then you need to do some preprocessing.

```
For  e(m,n) in E-{e(u,v)}  do
        If  c(e(m,n)) = 0 and (m,n) is not (u,v) then
                Merge m and n
        Else if c(e(m,n))=0 and (m,n) is (u,v) then
                Remove e(m,n) from E
        End if
End loop
```

So, basically if there is an edge with zero weight, we merge the two nodes into one single node, except that if the edge is connecting u and v, then we throw it away.
After preprocessing, the graph will not have zero edges except e(u,v) itself.

It is easy to see that the shortest path found in the new graph is equivalent to the ones in old graph. So then reduce the c(e(u,v)) by some amount less than the smallest weight in the new graph, and now we can detect if e(u,v) is on some of the shortest path.

In addition, for the graph used for D's algorithm, all weights should be non-negative.

One more thing is the upper bound of total shortest paths between s and t. Actually roughly calculating, the number of the total shortest paths is possible of O((n-1)!).
If there is only 1 vertex, then the shortest path is only 1, that is 0!=1.
Assume for a graph with n-1 vertex, we say the number of the total shortest paths between s and any other vertex is O((n-2)!).
Then adding another vertex into the graph, and the possible total shortest paths now is

$ST(n) <= ST(n-1)*(n-1)$,

Because there will be at most n new edges connecting to the new node (if there is no multiple edge between two nodes).

Then $ST(n) = O((n-1)!)$

So we say $ST(n) = O((n-1)!)$, n is the size of V in the worst case.
Hence, if you search the shortest paths to look for e, you probably will fall into a factorial worst case.

For a more tight bound, it could be $O(e^{|E|})$

Problem 3.

We proved that if there are no negative cycles in a graph, Bellman-Ford will halt after |V| sweeps. Thus if Bellman-Ford does not halt in |V| sweeps, there must be a negative cycle. To show this is necessary as well, consider that any two vertices on a negative cycle have an unbounded shortest path distance, as we can keep running around the cycle to reduce the path length. Hence each sweep will change at least the shortest path length between those two points, so Bellman-Ford will run > |V| sweeps.

> Run (|V|+1) sweeps of the Bellman-Ford algorithm.
> If the last sweep resulted in any improvement of any of the distances, then there is a negative cycle in the graph. Otherwise, there is no negative cycle.

The time complexity is $O(|E| * (|V|+1)) = O(|E|*|V|)$., since the number of sweeps is one more than that in the conventional Bellman-Ford algorithm.

Comments:

Several people used the vertex-queuing version of Bellman-Ford and argued that you know you have a negative cycle if a vertex is dequeued more than |V|+1 times. That's true, but you also may know that much sooner, as a vertex, even one in a negative cycle, may not be enqueued on every single sweep (consider a negative cycle that's far from S, for example). So people who answered like this demonstrated less-than-perfect understand of what's going on, and got a couple points taken off.

Problem 4.

I. Union(2,3), Union(1,4), Union(1,2), Union(5,6)

II. Not possible, as set (1) would have to be formed by taking the union of {1,2,3} with {4,5}, or {2,3} with {1,4,5}. For the first case to produce the final tree, we'd have to create a list 3 elements by unioning {1} with {2,3}, but union by size will instead make 2 the root. The other case is symmetric.

III. Union(2,3), Union(1,6), Union(1,2), Union(4,5), Union(1,4)