Problem 2

Clearly in a DAG the simple path with the most edges must be between some source and some sink. For vertex v let $d(v)$ be the the maximal number of edges on a path from some source to v. For a source vertex s we have $d(s) = 0$. Since there are no cycles in a DAG for every vertex v we have

$$d(v) = 1 + \max\{u \in \mathrm{In}(v) \mid d(u)\} \qquad (*)$$

Compute all the $d(v)$'s by first initializing $d(v) = 0$ for each vertex v and then going through all the vertices in topological order and applying formula (*).
This takes time proportional to 1 plus the sum of the indegrees, which is $O(n+m)$, with n being the number of vertices, and m the number of edges.

Problem 3

(Solution by A. Lohr.)

a. The test is whether the root has more than one child, in which case it is an articulation point.
First suppose the root r of G_π is an articulation point. Then the removal of r from G would cause the graph to disconnect, so r has at least 2 children in G. If r has only one child v in G_π then it must be the case that there is a path from v to each of r's other children. Since removing r disconnects the graph, there must exist vertices u and w such that the only paths from u to w contain r. To reach r from u, the path must first reach one of r's children. This child is connect to v via a path which doesn't contain r. To reach w, the path must also leave r through one of its children, which is also reachable by v. This implies that there is a path from u to w which doesn't contain r, a contradiction.

Now suppose r has at least two children u and v in G_π. Then there is no path from u to v in G which doesn't go through r, since otherwise u would be an ancestor of v. Thus, removing r disconnects the component containing u and the component containing v, so r is an articulation point.

b. Suppose that v is a nonroot vertex of G_π and that v has a child s such that neither s nor any of s's descendants have back edges to a proper ancestor of v. Let r be an ancestor of v, and remove v from G. Since we are in the undirected case, the only edges in the graph are tree edges or back edges, which means that every edge incident with s takes us to a descendant of s, and no descendants have back edges, so at no point can we move up the tree by taking edges. Therefore r is unreachable from s, so the graph is disconnected and v is an articulation point.

Now suppose that for every child of v there exists a descendant of that child which has a back edge to a proper ancestor of v. Remove v from G. Every subtree of v is a connected component. Within a given subtree, find the vertex which has a back edge to a proper ancestor of v. Since the set T of vertices which aren't descendants of v form a connected component, we have that every subtree of v is connected to T. Thus, the graph remains connected after the deletion of v so v is not an articulation point.

c. Since v is discovered before all of its descendants, the only back edges which could affect $v.low$ are ones which go from a descendant of v to a proper ancestor of v. If we know $u.low$ for every child u of v, then we can compute $v.low$ easily since all the information is coded in its descendants. Thus, we can write the algorithm recursively: If v is a leaf in G_π then $v.low$ is the minimum of $v.d$ and $w.d$ where (v, w) is a back edge. If v is not a leaf, v is the minimum of $v.d$, $w.d$ where w is a back edge, and $u.low$, where u is a child of v. Computing $v.low$ for a vertex is linear in its degree. The sum of the vertices' degrees gives twice the number of edges, so the total runtime is $O(E)$.

d. First apply the algorithm of part (c) in $O(E)$ to compute $v.low$ for all $v \in V$. If $v.low = v.d$ if and only if no descendant of v has a back edge to a proper ancestor of v, if and only if v is not an articulation point. Thus, we need only check $v.low$ versus $v.d$ to decide in constant time whether or not v is an articulation point, so the runtime is $O(E)$.

e. An edge (u, v) lies on a simple cycle if and only if there exists at least one path from u to v which doesn't contain the edge (u, v), if and only if removing (u, v) doesn't disconnect the graph, if and only if (u, v) is not a bridge.

f. A edge (u, v) lies on a simple cycle in an undirected graph if and only if either both of its endpoints are articulation points, or one of its endpoints is an articulation point and the other is a vertex of degree 1. Since we can compute all articulation points in $O(E)$ and we can decide whether or not a vertex has degree 1 in constant time, we can run the algorithm in part d and then decide whether each edge is a bridge in constant time, so we can find all bridges in$O(E)$ time.

g. It is clear that every nonbridge edge is in some biconnected component, so we need to show that if C_1 and C_2 are distinct biconnected components, then they contain no common edges. Suppose to the contrary that (u, v) is in both C_1 and C_2. Let (a, b) be any edge in C_1 and (c, d) be any edge in C_2. Then (a, b) lies on a simple cycle with (u, v), consisting of the path $a, b, p_1, \ldots, p_k, u, v, p_{k+1}, \ldots, p_n, a$. Similarly, (c, d) lies on a simple cycle with (u, v) consisting of the path $c, d, q_1, \ldots, q_m, u, v, q_{m+1}, \ldots, q_l, c$. This means $a, b, p_1, \ldots, p_k, u, q_m, \ldots, q_1, d, c, q_l, \ldots, q_{m+1}, v, p_{k+1}, \ldots, p_n, a$ is a simple cycle containing (a, b) and (c, d), a contradiction. Thus, the biconnected components form a partition.

h. Locate all bridge edges in $O(E)$ time using the algorithm described in part f. Remove each bridge from E. The biconnected components are now simply the edges in the connected components. Assuming this has been done, run the following algorithm, which clearly runs in $O(E)$ where E is the number of edges *originally* in G.

Problem 4

Note that if all edge weights are integers in the range 1 through K, then in Dikjstra's algorithm at any pont in time the set of keys in the priority queue is at most of size $K+2$, with all the at most $K+1$ finite integer keys lying between the current minimum x and $x+K$. When the algorithm starts there are only the key values 0 and ∞ and later iterations of the main loop of Dijkstra's algorithm can only create keys between x and $x+K$.
So the problem is to design a priority queue that can handle a set S of n elements with at most K distinct keys (the K here is the $K+2$ of the previous paragraph), so that all operations take $O(1)$ amortized time except for MinDelete and Delete, which are to take $O(\log K)$ time. Let us call this priority queuq *element queue*, EQ.
This element queue will make use of a priority queue PQ, with $O(1)$ time for all operations except for DeleteMin which takes logarithmic time (for the purposes of this question amortized bounds suffice, which subsumes worst case bound). Hollow heaps are a possible implementation for PQ.
PQ will store *items* which refer to non-empty sets of elements, i.e. for each item I in the PQ there is a non-empty set $U[I]$ of elements all having the same key key$[I]$. Every element of S occurs in exactly one $U[I]$.
We will require that an element x can test, whether its containing $U[I]$ contains other elements. If x is the only element, then it can determine the item I in constant time, otherwise it can be removed from $U[I]$ in constant time. We will also need that the union of two such (disjoint) sets $U[I]$ and $U[I']$ can be formed in constant time. This all can be achieved for instance by implementing $U[I]$ as a circular doubly linked list with a sentinel node.
Every x must of course know its key. We assume that sets of $O(K)$ keys can be stored in a dictionary W with constant insertion and lookup time. This can either be achieved with hashing, or in the application at hand by an array, that stores key k at position $k \bmod K$. We will also require that W can be enumerated in $O(K)$ time.
Now let us go through the operation for EQ.
To insert an element z into EQ, create a new item I with $U[I] = \{z\}$ and key$[I] = z$.key, and insert this item into PQ. This takes constant time, plus the insertion time into PQ, which is constant also.
To execute FindMin for EQ, do a FindMin on PQ which yields some item I, and then return one element of $U[I]$. This also takes constant time.
To do a DecreasKey(z, newkey) for EQ, first test whether the $U[I]$ that contains element z is a singleton. If yes, perform a DecreaseKey(I, newkey) operation on PQ. Otherwise remove z from $U[I]$, create a new item I' with $U[I'] = \{z\}$ and key$[I'] = z$.newkey, and insert this item into PQ. This takes constant time.
To perform a DeleteMin on EQ, take the element z returned by FindMin on EQ, and first test whether the $U[I]$ that contains element z is a singleton. If yes, perform a Delete(U) oepration on PQ. Otherwise simply remove z from $U[I]$. This takes constant time plus possibly the time to do a delete in PQ, which is logarithmic in L the number of items stored in PQ.
So it remains to make sure that $L = O(K)$. This is achieved as follows: Whenever PQ contains at least $2K$ items it is reconstructed. Using dictionary W a new set of items can be created so for every key there is exactly one new item that has associated with it all elements with that key. These new items are then inserted in a new PQ. This takes $O(K)$ time, which is constant in the amortized sense, since for such a reconstruction to happen, at least K insertions or decreasekey operations must have happend since the last reconstruction.

Problem 5

(a) Give the red edges a weight of -1 and the blue edges a weight of $+1$. Then the shortest path from u to v has negative weight iff there is a path with more red edges. This can be done by applying the Bellman-Ford algorithm and takes $O(mn)$ time.
(b) Give the red edges weight 1 and the blue edges weight 0, and apply Dijkstra's algorithm, which needs $O(m + n \log n)$ time.