11. Suppose we are given a value of $b > 1$. Since they claim that the Forward-Edge-Only Algorithm is guaranteed to find a flow of at least $1/b$ regardless of the graph, it suffices to construct a graph for which the algorithm fails to find a flow of this kind.

Let $G = (V, E)$ be a graph with $4b + 2$ vertices, with source s and sink t. There are $2b$ vertices $i_1, i_2, ..., i_{2b}$, each of which are connected to the source s. Similarly, there are $2b$ vertices $j_1, j_2, ..., j_{2b}$, each of which are connected to the sink t. Furthermore, i_1 is connected to j_1, i_2 is connected to j_2, and so on. In addition, we add an edge from j_1 to i_2, j_2 to i_3, and so on. Let the capacity of each edge be c.

Because they clalim that their algorithm works regardless of how it chooses its forward-edge paths, we can choose a forward-edge path for this graph. Note that the path $s \to i_1 \to j_1 \to i_2 \to ... \to j_{2b} \to t$ is a path that uses only the forward-edges, and in fact passes through each of the vertices. However, because this algorithm deletes the backwards edges, there is no longer a path from s to t, since all the edges between i and j are fully filled. Therefore, the algorithm terminates, with a maximum flow of c.

But there exists a maximum flow of $2bc$, since instead of following the edge (j_1, i_2), we could have simply taken the edge (j_1, t) and made it to the sink. Since there are $2b$ vertices, the maximum flow is $2bc$. Therefore the algorithm finds a flow of value $c/2bc = 1/2b < 1/b$ times the maximum, so their claim is false.

14. (a) In order to use the max-flow algorithm, we need a source node s' and a sink node t'. We create a new node s', and connect s' to each of the vertices $x \in X$. Similarly, we add a new node t' to the graph, and connect each of the vertices $s \in S$ to t'. Note that the problem is asking to find edge-disjoint paths from s' to t'. By Theorem 7.44, we can use Ford-Fulkerson to find a maximum set of edge-disjoint $s' - t'$ paths P in $O(nm^2)$ time. Since we want an escape route from every $x\imath X$ to some safe node, we can simply evaluate the cardinality of the set of edge-disjoint paths returned by Ford-Fulkerson. If $|P| < |X|$, then there is a vertex in X for which there is no escape route.

    Finding P requires $O(nm^2)$ time, and evaluating the cardinality of P takes linear time. Therefore, the algorithm runs in polynomial time.

    (b) Note that node-disjoint paths are also edge-disjoint, as two paths cannot share an edge (u, v) if they are not allowed to share u or v. Therefore, it suffices to find $|X|$ node-disjoint paths from s' to t'. For each node v in G, we split v into two nodes v_{in} and v_{out}, and add an edge of capacity 1 from v_{in} to v_{out}. We replace every edge (u, v) to be an edge from (u_{out}, v_{in}) with capacity 1 instead. We run the max-flow algorithm from s'_{out} to t'_{in}.

    Note that the division of each node from v to v_{in}, v_{out} causes the node-disjoint condition. To see this, consider two flows that pass through the same node in G before the node division occurs. Each of these flows has capacity 1. But after we divide the node, the edge from v_{in} to v_{out} only has capacity 1, meaning that both of these flows cannot pass through this same edge. At most one flow can go through each (v_{in}, v_{out}) pair. Therefore at the conclusion of the algorithm, the value of the maximum flow is the number of node-disjoint paths from s to t. If the maximum flow is fewer than the cardinality of X, then there is a vertex in X for which there is no escape route.

    Dividing each of the nodes takes $O(n)$ time, and running Ford-Fulkerson on the modified graph takes $O(nm^2)$. Therefore, the algorithm runs in polynomial time.

    As an example of a graph with a set of edge-disjoint escape routes but not node-disjoint escape routes, consider the following graph, with $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 3), (2, 3), (3, 4), (3, 5)\}$. Nodes 1 and 2 are the populated nodes, and nodes 4 and 5 are the safe nodes. Note that there is a edge-disjoint path from 1 to 4, and 1 to 5. However, as both must pass through node 3, these paths are not node-disjoint.

17. Because the maximum flow in the network before any edges are severed is k, we know that the min-cut has capacity k. Therefore, there exists some partition of vertices into A, B such that the sum of the flow leaving A minus the sum of the flow entering A is k. Consider each of the k edges that the attacker has destroyed. Since destroying these edges causes a minimum cut to appear (under the assumption that destroying k edges is the minimum number needed to separate s from t), our partition A, B is exactly the partition caused by a min-cut.

    Because the max-flow is k, we know that there are at least k edges leaving s with flow 1. Therefore, using Ford-Fulkerson, we can find a set of k edge-disjoint paths from s to t (assuming that the edges haven't been severed yet). For any given path $P = \{v_1, ..., v_i\}$, we can binary search with ping in order to find the edge that causes a failure. Because each path can only go through each node once, we know that $i \leq n$. Therefore, it takes $O(\log n)$ pings to binary search for a edge for any given path. Since we have k paths, we require $O(k \log n)$ pings in order to find all of the breaks.

29. Construct a directed graph $G = (V, E)$, with each of the $n - 1$ software applications $(2, ..., n)$ representing a node. We let node 1 be the source node, and let an arbitrary node t be the sink node. We split each application into two separate nodes i and i', with i representing moving the application to a new system, and i' representing not moving the application to a new system. We connect node 1 to all i, and connect node i' with t. Additionally, we connect j' to i for every $i \neq j$.

    For each edge $(1, i)$, we let the capacity of this edge be $b_i - x_{1i}$, since we already know that application 1 will be staying on the initial server. For each edge (i, t), we let the capacity of this edge be ∞, since we don't want the sink to become the bottleneck. For each edge $(i, j'), i \neq j$, we let the capacity of this edge be $b_i - x_{ij}$, representing the expense of taking i but not j. Finally, we run the Ford-Fulkerson on this graph to get the maximum flow. We follow the flow in the diagram, and if the flow passes through node i, we say that we port application i over. Otherwise, if it passes through node i', we say that we don't port application i over.

    Constructing the nodes in the graph takes $O(n)$ time, since each application must be split into two nodes. Constructing the edges in the graph takes $O(n^2)$ time, since there is an edge for each pair of applications (i, j'). Finally, running Ford-Fulkerson and tracing through the flow at the end takes $O(nm^2)$ time. Therefore, this algorithm runs in $O(nm^2)$.