1. An algorithm for directed graphs is more general. It can be applied to an undirected graph by splitting each undirected edge into two directed edges: one either way.

**Remark**: Algorithms that are limited to undirected graphs often use some acknowledgement scheme for messages, which can only be implemented efficiently if edges are undirected.

2. Initially $\text{dist}(A) = 0$ and all other nodes have $\text{dist}$ value $\infty$; all nodes have $\text{next}$ value $\bot$.

- A is removed from the heap: $\text{dist}(B) \leftarrow 13$, $\text{dist}(C) \leftarrow 6$ and $\text{dist}(D) \leftarrow 4$. Moreover, the $\text{next}$ value of these nodes becomes A.
- D is removed from the heap: $\text{dist}(B) \leftarrow 10$, $\text{dist}(C) \leftarrow 5$, $\text{dist}(F) \leftarrow 7$ and $\text{dist}(G) \leftarrow 24$. Moreover, the $\text{next}$ value of these nodes becomes D.
- C is removed from the heap.
- F is removed from the heap.
- B is removed from the heap: $\text{dist}(E) \leftarrow 12$ and $\text{next}(E) \leftarrow B$.
- E is removed from the heap: $\text{dist}(G) \leftarrow 18$ and $\text{next}(G) \leftarrow E$.
- G is removed from the heap.

3. Yes. When the heap consists of a single node, the $\text{dist}$ and $\text{next}$ values of all other nodes have been fixed, because they have already been removed from the heap. Removing the last node from the heap will therefore not give rise to any further improvement of $\text{dist}$ and $\text{next}$ values.

4. Suppose the cycle consists of edges $uv$, $vw$ and $wu$, and we compute shortest paths to $u$.

- Initially $\text{dist}(u) = 0$ and the other two nodes have $\text{dist}$ value $\infty$; all nodes have $\text{next}$ value $\bot$.
- In the first cycle of the for loop, edges $uv$ and $vw$ don’t give rise to an improved $\text{dist}$ value, while $wu$ gives rise to $\text{dist}(w) \leftarrow -1$ and $\text{next}(w) \leftarrow u$.
- In the second and last cycle of the for loop, edge $uv$ doesn’t give rise to an improved $\text{dist}$ value, $vw$ gives rise to $\text{dist}(v) \leftarrow -2$ and $\text{next}(v) \leftarrow w$, and edge $wu$ doesn’t give rise to an improved $\text{dist}$ value.
Finally, edge \( uv \) gives rise to \( \text{dist}(u) \leftarrow -3 \) and \( \text{next}(u) \leftarrow v \). So a cycle is detected.

5. (a) Initially \( \text{value}(s) = 0 \) and all other nodes have \( \text{value} \infty \); all nodes have \( \text{next} \perp \).

\( s \) is removed from the heap: \( \text{value}(t) \leftarrow 7 \), \( \text{value}(u) \leftarrow 4 \), \( \text{value}(v) \leftarrow 2 \), \( \text{value}(w) \leftarrow 9 \) and \( \text{value}(x) \leftarrow 8 \). Moreover, the \( \text{next} \) of these nodes becomes \( s \).

\( v \) is removed from the heap: \( \text{value}(u) \leftarrow 3 \) and \( \text{value}(w) \leftarrow 7 \). Moreover, the \( \text{next} \) of these nodes becomes \( v \).

\( u \) is removed from the heap: \( \text{value}(t) \leftarrow 6 \) and \( \text{next}(t) \leftarrow u \).

\( t \) is removed from the heap. \( \text{value}(x) \leftarrow 7 \) and \( \text{next}(x) \leftarrow t \).

\( w \) is removed from the heap. \( \text{value}(x) \leftarrow 1 \) and \( \text{next}(x) \leftarrow w \).

\( x \) is removed from the heap.

(b) Initially each node forms a single-element set and the minimum spanning tree is empty; the edges are placed in a list, ordered by weight.

Edge \( wx \) is removed from the set, and becomes part of the spanning tree; fragments \( \{w\} \) and \( \{x\} \) are joined.

Edge \( sv \) is removed from the set, and becomes part of the spanning tree; fragments \( \{s\} \) and \( \{v\} \) are joined.

Edge \( uw \) is removed from the set, and becomes part of the spanning tree; fragments \( \{u\} \) and \( \{s,v\} \) are joined.

Edge \( su \) is removed from the set.

Edge \( tu \) is removed from the set, and becomes part of the spanning tree; fragments \( \{t\} \) and \( \{s,u,v\} \) are joined.

Edge \( st \) is removed from the set.

Edge \( tx \) is removed from the set, and becomes part of the spanning tree; fragments \( \{s,t,u,v\} \) and \( \{w,x\} \) are joined.

Edge \( vw \) is removed from the set.

Edge \( sx \) is removed from the set.

Edge \( sw \) is removed from the set.

6. The initialization phase, where the selected node gets weight 0, all other nodes get weight \( \infty \), and the nodes are placed in a min-heap, clearly takes \( \Theta(n) \) time.

**Remark:** For the time analysis of the while loop ("while \( H \) is non-empty do"), it is important not to analyze the worst case of each run of the while loop separately. Because then we would end up with a worst-case time complexity of \( O(n^2) \): there are \( n \) runs of the while loop, and in case the root node \( v \) of the heap has \( n - 1 \) neighbors, the while loop takes \( O(n) \) time.

Instead, we analyze the part "For each neighbor \( w \in H \) of \( v \)" within the while loop separately. This check is in total executed \( 2m \) times: twice per edge (one time for either direction of an edge). Each check and the assignments that may result from it take \( \Theta(1) \); and restructuring the heap takes \( O(\log n) \). So in total these checks and assignments have a worst-case time complexity of \( O(m \cdot \log n) \).
Finally, in the while loop, swapping the root node of $H$ with the last element of $H$ takes $\Theta(1)$, and then restructuring the heap takes at most $O(\log n)$. So in total, doing this for all $n$ runs of the while loop takes at most $O(n \cdot \log n)$.

Of the three parts of the time complexity we have determined (initialization $\Theta(n)$, and within the while loop $O(m \cdot \log n)$ and $O(n \cdot \log n)$), the second part $O(m \cdot \log n)$ contributes most to the time complexity. So concluding, the worst-case time complexity of Prim’s algorithm is $O(m \cdot \log n)$.

7. Consider a scenario of Kruskal’s algorithm where after $i$ joins of fragments, for each $i = 0, \ldots, n-1$, there is one fragment $F$ of $i+1$ nodes and all other fragments have one node. Meaning that each join connects $F$ to a single node.

Suppose moreover that the union-find algorithm would allow the smaller set to subsume the larger set. Then in our scenario it could be the case that in each join, all nodes in $F$ must update their link to the single node to which $F$ is joined. So the first join gives rise to one such update, the second join to two such updates, et cetera, until the $(n-1)$th and last join gives rise to $n-1$ such updates. Adding up to $1+2+\cdots+(n-1) = \frac{1}{2}(n-1)n$ updates in total.

This implies that the worst-case time complexity with this changed (deteriorated) union-find algorithm would be $O(n^2 + m \log n)$ (where the summand $n^2$ is due to the updates of links in the deteriorated union-find algorithm and the summand $m \log n$ is due to sorting the list of edges by their weights).

**Remark:** Since $m$ can be between $n-1$ and $n^2$, the summands $n^2$ and $m \log n$ in this worst-case time complexity are incomparable.

8. Suppose, toward a contradiction, that there are two different minimum spanning trees $T_1$ and $T_2$ for the weighted graph. Let $e$ be the lowest-weight edge that is in one of the trees (say $T_1$) but not in the other ($T_2$). Clearly, $T_2 \cup \{e\}$ contains a cycle. Since $T_1$ is acyclic, at least one of the edges on this cycle, say $e'$, isn’t in $T_1$. By replacing $e'$ in $T_2$ by $e$, we clearly obtain another spanning tree $T_3$ of the weighted graph.

Since $e$ is the lowest-weight edge that is in one of the trees but not in the other, and $e' \notin T_1$ implies $e' \neq e$, the weight of $e'$ is larger than the weight of $e$. (Here we use that all edges in the weighted graph carry a different weight, so that $e$ and $e'$ can’t have the same weight.) Hence the sum of the weights of the edges in $T_3$ is smaller than in $T_2$. This contradicts the assumption that $T_2$ is a minimum spanning tree.

Concluding, there is a unique minimum spanning tree.