Solution 10.1  Path Planning in Labyrinths.

a) We define a directed graph with a vertex for each possible state of the system. A state is completely determined by three parameters:

1) the position where the robot is located,

2) the direction in which the robot is facing, and

3) whether the robot is standing or moving.

For every position where the robot can be located we then have 8 possible states (4 directions multiplied by 2 possibilities whether the robot is standing or moving). We construct our graph accordingly with 8 vertices per square in the labyrinth. In addition, we need a vertex for the target state, representing the state in which the robot escaped. We add an edge for each movement, rotation and stopping operation, with weights according to the given duration of the corresponding operations. A shortest path in the graph from the starting state of the robot to the target state corresponds to a sequence of operations of the robot that allows it to escape as quickly as possible.

The following picture illustrates this construction on a small section of a labyrinth. The labels of the vertices indicate the direction that the robot is facing; double arrows indicate that the robot is moving. Of course, we could remove the states that are not reachable.

b) Since the graph that we constructed does not have negative edge weights, a shortest path can be found using Dijkstra’s algorithm.

c) For each position in the labyrinth, we used a constant number of vertices (precisely 8) and a constant number of edges (at most 20). If \( n \) is the number of squares in the labyrinth then we have \( |V| \in \mathcal{O}(n) \) and \( |E| \in \mathcal{O}(n) \), thus the running time of Dijkstra's algorithm is bounded by \( \mathcal{O}(n \log n) \).
Solution 10.2  Variants of Shortest Path Problems.

a) We can modify the algorithm of Bellman and Ford to solve this problem. For every vertex \( v \in V \) and every \( i \in \mathbb{N} \) let \( d_{i,v} \) be the length of a shortest path from \( s \) to \( v \) that contains at most \( i \) edges. Furthermore let \( \pi_{i,v} \) be the predecessor of \( v \) on a shortest \( s \)-\( v \) path with at most \( i \) edges. We obtain the following pseudocode:

**Bellman-Ford**\((V, E, w, s)\)

**Input:** Directed, weighted graph \((V, E, w)\), starting vertex \( s \in V \)

**Output:** Preceding vertices \( \pi_{i,v} \) for all \( v \in V \) and all \( i \in \{0, \ldots, k\} \)

1. for each \( v \in V \) do \( d_{0,v} \leftarrow \infty; \pi_{0,v} \leftarrow \text{null} \)
2. \( d_{0,s} \leftarrow 0 \)
3. for \( i \leftarrow 1, \ldots, k + 1 \) do
   4. for each \( v \in V \) do \( d_{i,v} \leftarrow d_{i-1,v}; \pi_{i,v} \leftarrow \pi_{i-1,v} \)
   5. for each \((u, v) \in E\) do
      6. if \( d_{i-1,u} + w((u, v)) < d_{i,v} \) then
         7. \( d_{i,v} \leftarrow d_{i-1,u} + w((u, v)) \)
         8. \( \pi_{i,v} \leftarrow u \)

After \( i \) iterations of the loop in the third step we considered every path from the starting vertex that consists of at most \( i \) edges. After \( k + 1 \) iterations we therefore considered every path that consists of at most \( k + 1 \) edges, i.e. that contains at most \( k \) intermediate vertices (the normal algorithm would repeat the loop \(|V| - 1\) times to consider all possible shortest paths). A shortest \( s \)-\( t \) path with at most \( k + 1 \) many edges can be reconstructed by initially setting \( v \leftarrow t \) and following the corresponding preceding vertex until a vertex without a predecessor (i.e., the starting vertex \( s \)) is reached. This task can be solved as follows:

**Reconstruct-Shortest-Path**\((\pi_{i,v}, s, t)\)

**Input:** Preceding vertices \( \pi_{i,v} \) for all \( v \in V \) and all \( i \in \{0, \ldots, k\} \), vertices \( s \) and \( t \)

**Output:** A shortest \( s \)-\( t \) path \( P = \langle v_0, v_1, \ldots, v_l \rangle \) where \( v_0 = s \), \( v_l = t \) and \( l \leq k + 1 \)

1. if \( d_{k+1,t} = \infty \) then Report that there exists no \( s \)-\( t \) path with at most \( k + 1 \) edges.
2. \( v \leftarrow t ; \ P \leftarrow \langle \rangle ; \ i \leftarrow k + 1 \)
3. while \( v \neq \text{null} \) do \( P \leftarrow v \oplus P ; \ v \leftarrow \pi_{i,v} ; \ i \leftarrow i - 1 \)
4. return \( P \)

In the above algorithm, \( P \) is implemented as a sorted list. The operator \( \oplus \) refers to the concatenation of two lists, and \( v \oplus P \) is the list that we obtain when \( v \) is inserted at the beginning of \( P \).

In the first algorithm every iteration of the loop takes time \( \mathcal{O}(|V| + |E|) \). Additionally a shortest path can be reconstructed by the second algorithm in time \( \mathcal{O}(|V|) \). The overall running time of our solution therefore is \( \mathcal{O}(k \cdot (|V| + |E|)) \), i.e., \( \mathcal{O}(k \cdot |E|) \) if the graph is connected.

b) We first compute a shortest path from \( s \) to \( t \). Let \( l \) be the number of edges used by this path.

The second shortest \( s \)-\( t \) path must differ from the shortest \( s \)-\( t \) path in at least one edge, otherwise they would be the same. Therefore we can successively remove exactly one of
the \( l \) edges of the shortest path from the graph and compute again the shortest path in the remaining graph. Thus we compute \( l \) shortest paths in a graph with exactly one edge (that differs in each step) less. Among these \( l \) shortest paths we take the one of shortest length and obtain the second shortest path. Notice that the length of the second shortest path does not necessarily differ from the length of a shortest path.

Obviously the running time is bounded by \( \mathcal{O}(l \cdot \text{Time to compute a shortest path}) \). In the worst case the shortest path visits every vertex and uses \( |V| - 1 \) edges. If Dijkstra’s algorithm is used to compute the shortest paths then the overall running time of the solution is \( \mathcal{O}(|V| \cdot |E| + |V|^2 \cdot \log(|V|)) \).

**Notice:** There exists an algorithm that computes the \( k \) best non-simple paths (where vertices/edges may be used multiple times) for a given \( k \in \mathbb{N} \) in time \( \mathcal{O}(|E| + |V| \log|V| + k) \) (details can be found in “Finding the \( k \) Shortest Paths”, D. Eppstein, FOCS, 1994).

c) We use Dijkstra’s algorithm, and for every vertex \( v \in V \) we do not only store the distance \( d_v \) from \( s \) to \( v \) but also the number \( N_v \) of paths having this distance. For every \( v \in V \setminus \{s\} \) we initially set \( d_v \leftarrow \infty \) and \( N_v \leftarrow 0 \). Additionally we set \( d_s \leftarrow 0 \) and \( N_s \leftarrow 1 \). The test whether it is worth to use a vertex \( u \) as a shortcut is adapted as follows: if \( d_u + w((u, v)) < d_v \) (i.e., the shortest \( s \)-\( v \) path that uses \( u \) is smaller than the current shortest path to \( v \)) then we set \( d_v \leftarrow d_u + w((u, v)) \) and \( N_v \leftarrow N_u \). If \( d_u + w((u, v)) = d_v \) (i.e., the \( s \)-\( t \) path that uses \( u \) has the same length than the current shortest path to \( v \)) then \( N_v \) is increased by \( N_u \) (since all shortest paths that use \( u \) have to be additionally counted).

The correctness of this algorithm follows from the fact that Dijkstra’s algorithm relaxes every edge only once. The running time is the same than the one of the usual algorithm because for every edge there are only constantly many additional operations.

**Solution 10.3**  
**Topological Sorting using DFS.**

a) “⇒”: Suppose that the depth first search finds a backward edge \((u, v)\). This means that the depth first search has already visited \( v \) and it is now at \( u \). In the DFS tree, \( v \) is an ancestor of \( u \). The path from \( v \) to \( u \) together with the edge \((u, v)\) is a cycle in \( G \).

“⇐”: Suppose that \( G \) contains a cycle. Let \( v \) be the first vertex in this cycle that is visited by the depth first search, and let \((u, v)\) be the (only) edge to \( v \) in the cycle. When \( u \) is visited by the depth first search, \( v \) is not left yet. Thus the edge \((u, v)\) leads to an ancestor of \( u \) in the DFS tree, and it is therefore a backward edge.

b) Let \( T(v) \) be the exit time of the vertex \( v \), i.e., the number of steps of the depth first search until \( v \) is left. We want to show that we obtain a topological sort if we arrange the vertices in descending order according to their exit times. This is the case if \( T(u) > T(v) \) for every edge \((u, v)\).

Since \( G \) is acyclic we know from a) that the depth first search does not encounter any backward edges. This means that every edge \((u, v)\) considered by the depth first search leads from \( u \) to an unvisited vertex or a vertex that has already been left earlier. If the vertex \( v \) was not left earlier, it is reached from \( u \). Thus the depth first search visits \( v \) and leaves it before leaving \( u \). Therefore, we have \( T(u) > T(v) \) for every edge \((u, v)\).

c) We could use a depth first search that remembers every exit time, and then sorts the vertices in descending order according to their exit times. However, it is much easier (and more efficient!) to add vertices to the solution as soon as they are left during the depth first search, without having to remember anything except the solution.
TopologicalSort($G, v$)

**Input:** Directed graph $G$, starting vertex $v$

**Output:** The vertices of $G$ in topological order, or a hint that $G$ contains a cycle

1. $S ← \langle \rangle$
2. Mark $v$ as “active”
3. **for each** $(v, w) ∈ E$ **do**
4.   **if** $w$ is marked as “active” **then**
5.     Report that $G$ contains a cycle and abort the search
6.   **if** $w$ is not marked as “visited” **then**
7.     $S ← \text{TopologicalSort}(G, w) ⊕ S$
8. Mark $v$ as “visited” (instead of “active”)
9. **return** $v ⊕ S$

In the above algorithm, $S$ is a sorted list and the operator $⊕$ is defined as in the previous exercise.

The algorithm is mainly a depth first search. Therefore its running time is $O(|V| + |E|)$.

**Notice:** In the implementation of the above pseudocode, you should avoid to copy partial solutions (i.e., the sorted subsequences) between recursive calls. Think how the running time would change if you copied partial solutions around, and how to avoid to copy them.