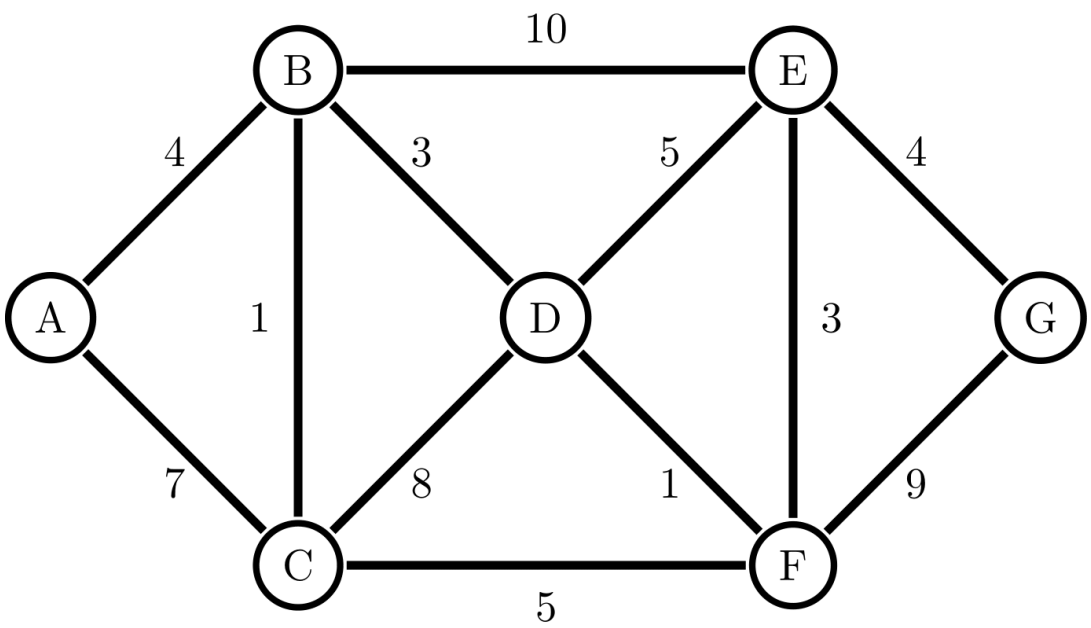
Theory: Dijkstra's algorithm

33 minutes 11 / 11 problems solved

Assume that a weighted graph below represents a map, where the nodes are cities, the edges are roads and the weight of each edge is the distance between two cities:



Suppose we are in the city A and want to take a trip to the city G. To get from A to G as fast as possible, we need to know **the shortest path** between the cities, that is, the path with the minimum total weight. How can this path be found?

One of the possible approaches is to use **Dijkstra's algorithm**. It allows to find the shortest paths from a given node to **all** other nodes of a graph. Let's consider the algorithm in more detail.

§1. Algorithm description

First, let's agree that we will use the following notations further:

- source — the node from which we want to find the shortest paths to all other nodes;
- $weight(u, v)$ — the weight of the edge that connects the nodes u and v;
- $dist(v)$ — the current distance from the source to the node v.

Now, Dijkstra's algorithm can be formulated as follows:

1. Set the current distance to the source to 0, for all other nodes assign it to ∞. Mark all nodes as unprocessed.
2. Find an unprocessed node u with the smallest $dist(u)$.
3. For all unprocessed neighbors v of the node u, check whether the distance from u to v is less than the current distance to v. If that is the case, that is $dist(u) + weight(u, v) < dist(v)$, update the current distance to v to a smaller value.
4. When all the neighbors of u are considered, mark u as processed.
5. If there are no unprocessed nodes, the algorithm is finished. Otherwise, go back to the 2nd step.

After all the nodes are processed, each of the current distances $dist(u)$ is the shortest distance from the source to the node u.

§2. Example

Let's apply the algorithm to find the shortest paths from the node A to all other nodes of the graph above. On the figures below, the node that is processed at the current step and the edges incident to the unprocessed neighbors of this node are shown in green. The nodes that are already processed and the edges that correspond to the current shortest paths are shown in blue. The current distances are shown in bold near the nodes.

Current topic:

- ✓ Dijkstra's algorithm

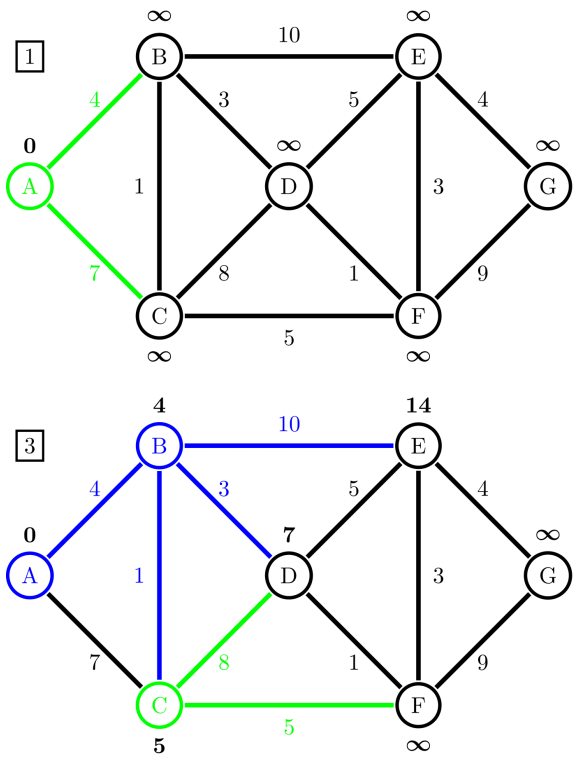
Topic depends on:

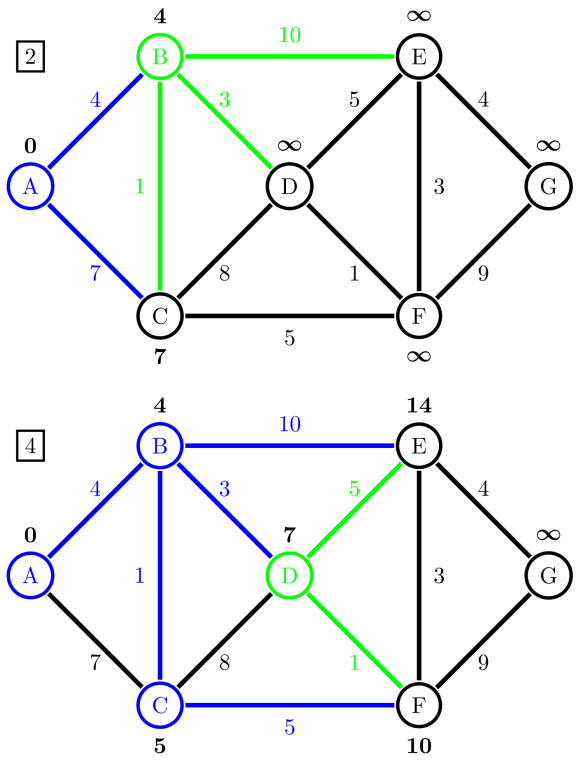
- ✓ Weighted graph
- ✓ Breadth-first search
- ✓ Priority queue

Table of contents:

At the first step, we consider the source node A, since it has the smallest current distance. The node has two neighbors: B and C. The distance from A to B is $dist(A) + weight(A, B) = 0 + 4 = 4$. Since 4 is less than ∞, the distance to B is set to 4. Similarly, the distance to C is set to 7. Then, A is marked as processed.

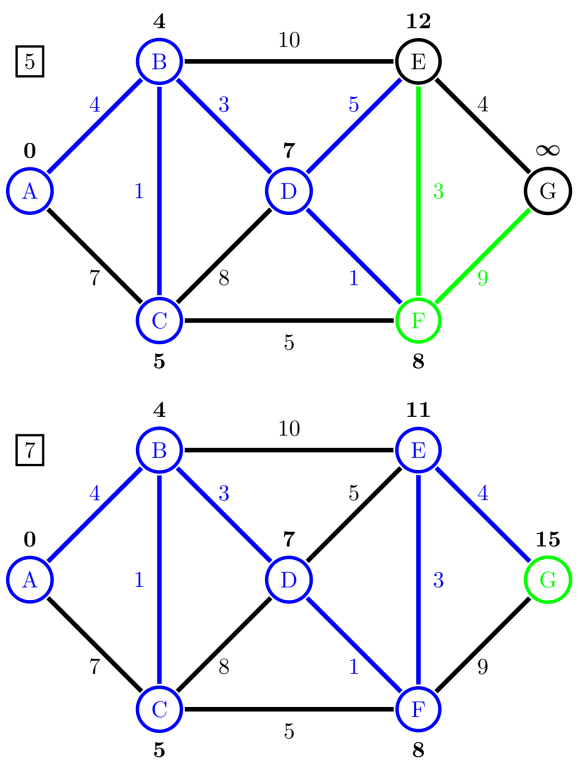
At the next step, the node B has the smallest distance. The distance from B to C is $dist(B) + weight(B, C) = 4 + 1 = 5$, which is less then 7. So, the distance to C is set to 5. Following the same rule, the distances to D and E are set to 7 and 14 respectively and then B is marked as processed.

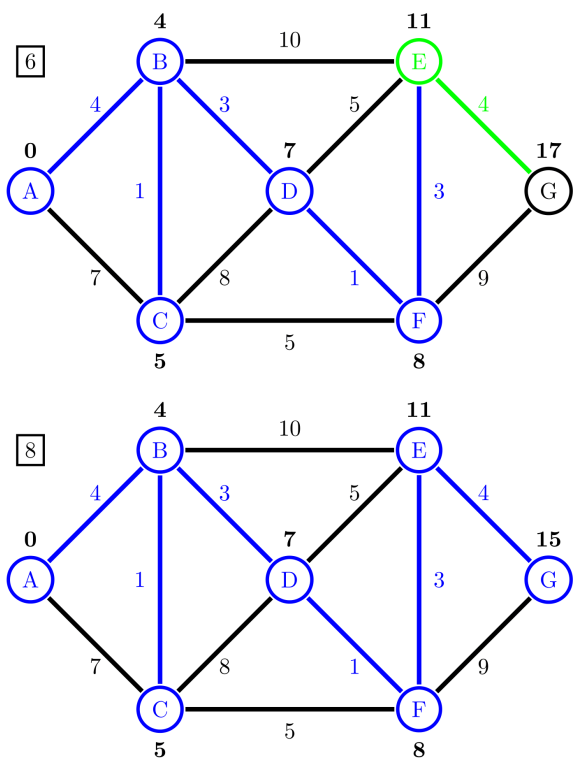




Next, the node C has the smallest distance. The distance from C to D is $dist(C) + weight(C, D) = 5 + 8 = 13$, which is greater than 7. So, the distance to D does not need to be updated. The distance from C to F is 10, which is less than ∞. So, it is set to 10.

At the 4th step, the node D has the smallest distance. The distances from D to E and F are less than the previous distances for these nodes, so they are set to 12 and 8 respectively.





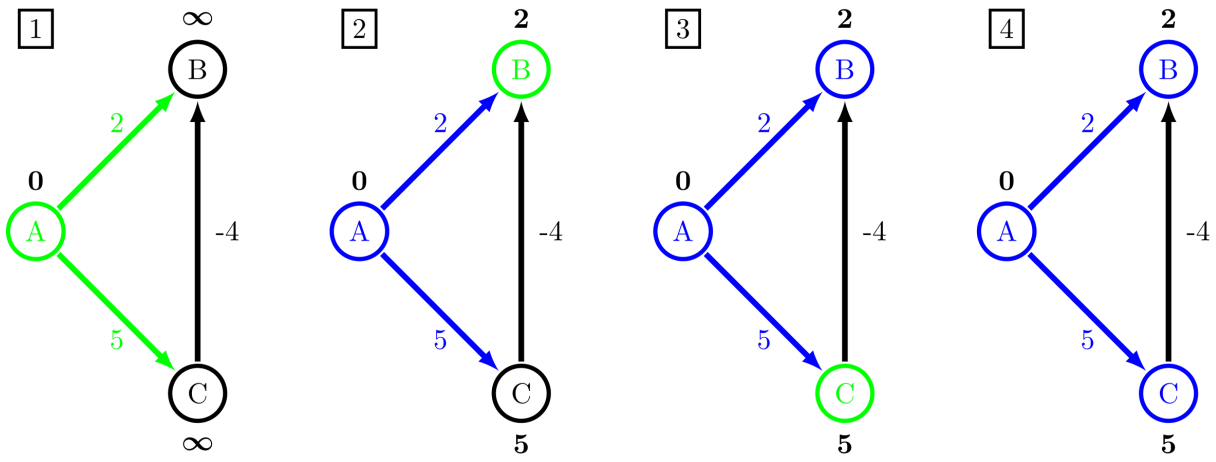
Continuing the same process for the remaining nodes, we finally get the graph shown in the figure 8.

The resulting tree (shown in blue) is called the **shortest path tree.** We can use it to reconstruct the shortest path from the source to any other node of the graph. To achieve this, each node should know the previous node on the shortest path from the source to it. The algorithm can be slightly modified to calculate this information.

In the beginning, we create another array $from$ and fill it with default values. Every time we stay at some vertex u and update the distance to its neighbour v, we should also update $from(v) = u$, since the current shortest path now comes to v from u.

To get the shortest path from the source to a node v, we start at v, then move to $from(v)$, then to $from(from(v))$, and so on, until we get to the source. This gives us the shortest path in the reversed order. For example, the shortest path from A to G is $A \to B \to D \to F \to E \to G$. The total weight of this path is $4 + 3 + 1 + 3 + 4 = 15$, which corresponds to $dist(G)$.

Note that although the graph above is undirected, the same algorithm works for directed graphs as well. Another important detail of Dijkstra's algorithm is that it does not work if a graph contains negative weight edges. The example below illustrates the statement:



We can see that the final distance from A to B is 2, although the path $A \to C \to B$ has a weight of $5 + (-4) = 1$. The problem is that when the node C was considered, the node B had already been processed and the weight of B could not be updated.

The last thing to mention is that there are a lot of resources that allow visualizing the steps of Dijkstra's algorithm. Those who are interested in a more interactive example of how the algorithm works may check out a visualization.

§3. Complexity analysis

Consider a graph G with n nodes and m edges. The running time of Dijkstra's algorithm for this graph depends on a data structure used for storing the current distances.

One of the possible implementations uses an array A of size n, where $A[i]$ stores the current distance to the node i. In this case, finding the smallest distance requires $O(n)$ operations, since we need to scan the whole array. The update operation can be done in $O(1)$. Since the total number of searches is n and the total number of updated is no more than m, the overall running time is $O(n^2 + m)$.

Another approach is to store the distances in a priority queue. If the queue is implemented via a binary heap, searching and updating can be done in $O(\log n)$. In this case, the running time is $n \cdot \log n + m \cdot \log n = O((n + m) \cdot \log n)$.

So, which of the implementations is preferable? If G is a dense graph (with lots of edges, $m \approx n^2$), then the priority queue based implementation results in $O(n^2 \cdot \log n)$. In this case, the array-based implementation is a better choice. For sparse graphs, (with few edges, $m \approx n$), the priority queue based implementation is more appropriate since for such graphs the running time of the algorithm is $O(n \log n)$.

Report a typo

29 users liked this theory. 3 didn't like it. What about you?

Start practicing

Comments (2) Hints (0) Useful links (0) Show discussion