Theory: Prim's algorithm

25 minutes 12 / 12 problems solved

Start practicing

438 users solved this topic. Latest completion was 1 day ago.

Prim's algorithm allows one to find a **minimum spanning tree** in an undirected connected weighted graph. The algorithm starts constructing a tree with a single node. At each step, it considers all edges $\{x, y\}$ such that x is a node of the current tree and y is not. Among all such edges, the one with the smallest weight is added to the current tree. So, the algorithm is **greedy**: at each step, an edge giving the best possible advantage is selected.

Current topic:

✓ Prim's algorithm

Topic depends on:

✓ Spanning trees

§1. Example

Let's consider how the algorithm works for the following graph:

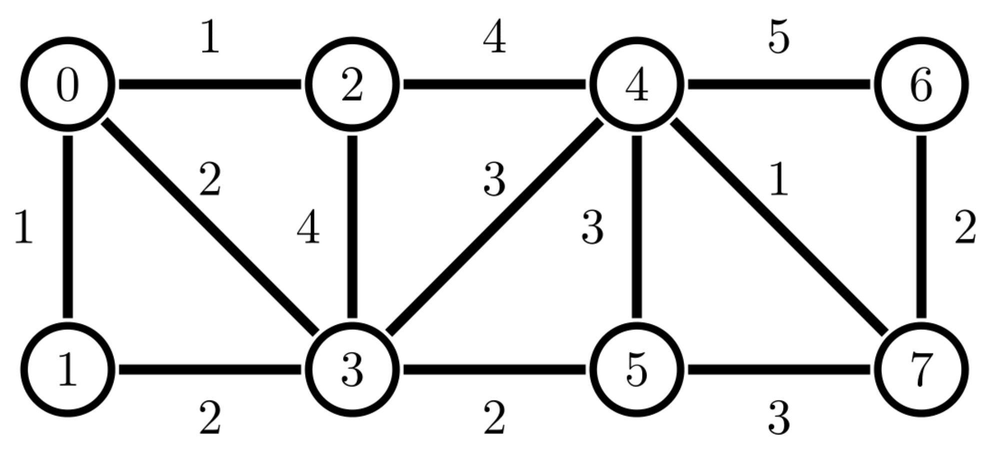
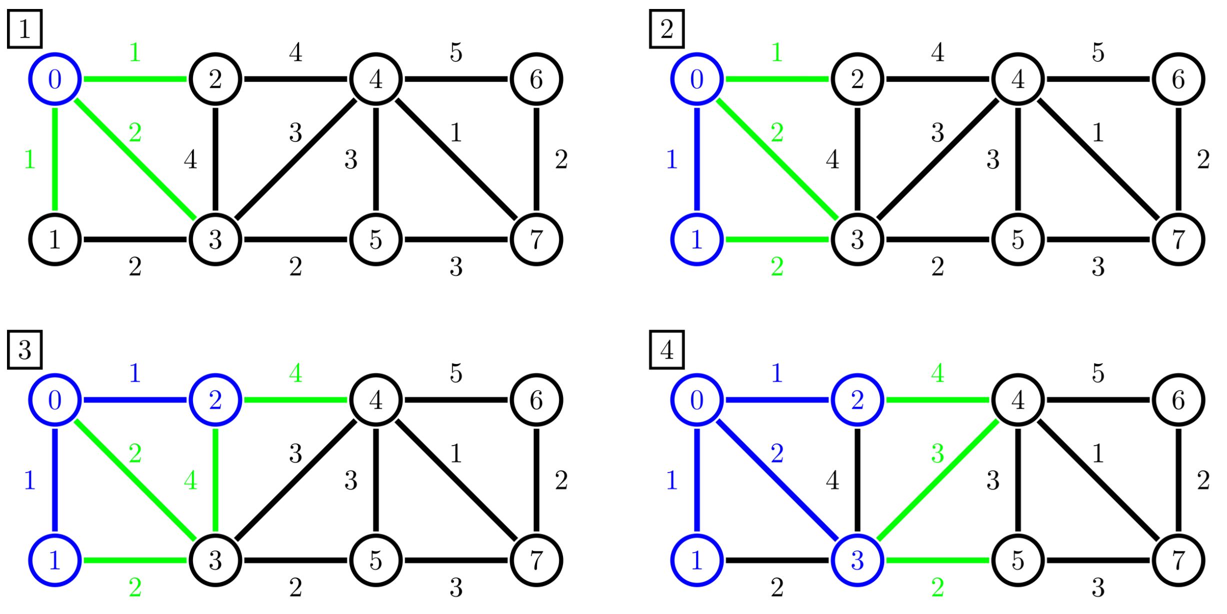


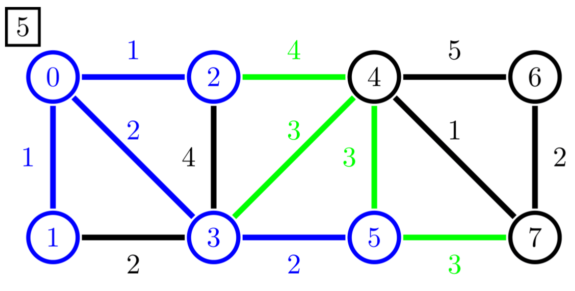
Table of contents:

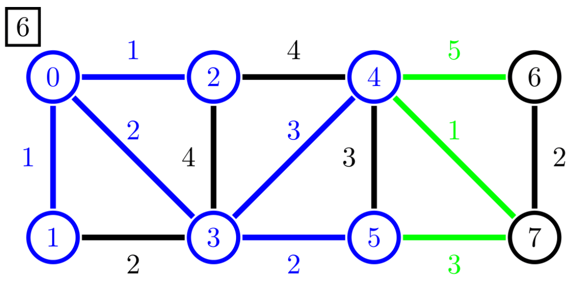
In the figures below, the nodes and the edges of the current tree are shown in blue, the edges which are candidates to be added to the current tree will be shown in green.

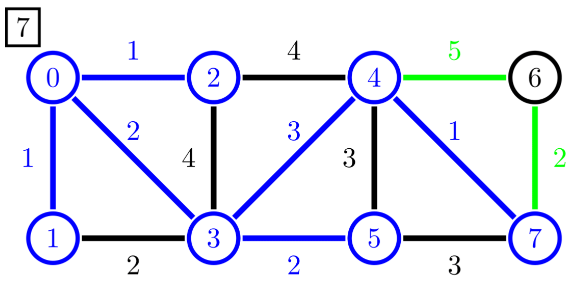
We start constructing a minimum spanning tree with the node 0. There are three edges that are incident to this node: $\{0, 1\}$, $\{0, 2\}$and $\{0, 3\}$. The edges $\{0, 1\}$ and $\{0, 2\}$ have the smallest weight. We choose the first edge and add to the current tree. At the next step, the only edge that can be added to the current tree is $\{0, 2\}$.

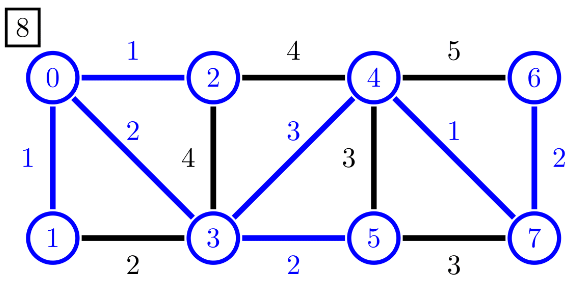


Then, we can choose either $\{0, 3\}$ or $\{1, 3\}$. We choose the first edge and add it to the tree. After that, the only edge that can be included in the tree is $\{3, 5\}$.

5


6


7


8


Continuing the same process we finally get a spanning tree shown in figure 8. The resulting minimum spanning tree consists of 7 edges and has a weight of $1+1+2+3+2+1+2=12$.

§2. Summary

To sum up, Prim's algorithm constructs a **minimum spanning tree** in an undirected connected weighted graph using a greedy approach. The steps of the algorithm are:

1. Select an arbitrary node in a graph as the first node of a spanning tree.
2. Consider all edges $\{x, y\}$ such that x is a node of the current tree and y is not.
3. Among all such edges, choose the one with the smallest weight and add it to the current tree.
4. Repeat steps 2-3 while there are edges that can be added to the current tree. Then, return the resulting tree as a final answer.

§3. Complexity analysis

Suppose that a graph G has n nodes and m edges. The running time of Prim's algorithm depends on the representation of the graph and the data structure used for finding an edge with the smallest weight.

For example, we may use an adjacency matrix to store the graph and at each step find an edge with the smallest weight by simply considering all the edges using this matrix. Since the size of the matrix is n^2 and the total number of steps is $n-1$, the running time in such a case is $O(n^3)$.

However, this running time can be decreased: for each node x of G that is not yet in the current tree, we may store an edge $\{x, y\}$ of the smallest weight where y is a node that is already in the tree. Then at each step, we can consider all such edges, choose the one with the smallest weight and add it to the current tree. If we use a list or an array to store the edges, finding an edge with the smallest weight will require $O(n)$ operations. Note that after adding a new edge $\{x, y\}$ to the tree, the nodes that are neighbors for y may need to be updated. This can be done in $O(n)$ by considering all the neighbors of y and checking whether the edge incident to y has a smaller weight than the previous edge with a smaller weight for the neighbor. Since the total number of steps is $n-1$, the overall running time is $O(n^2)$ which is better than in the previous case.

A possible modification of the previous approach is to use a priority queue instead of a list (or an array) to store a set of edges with the smallest weight. In this case, updating and finding an edge can be done in $O(logn)$ (assuming that a priority queue is implemented via a binary heap). Since each edge can be updated only once and the total number of times we find an edge with the smallest weight is no more than n, the overall running time is $O(mlogn + nlogn) = O(mlogn)$.

For sparse graphs $(m \approx n)$ this approach works in $O(nlogn)$, but for dense graphs $(m \approx n^2)$ it results in $O(n^2logn)$ which is slower than the previous algorithm. So, a priority queue is reasonable to use when you know that a graph is sparse. Note that in this case, it is better to use an adjacency list to store the graph since for sparse graphs it requires less additional memory.

Report a typo

29 users liked this theory. 5 didn't like it. What about you?

Start practicing

Comments (1) Hints (0) Useful links (0) Show discussion