Prim's algorithm

Outline

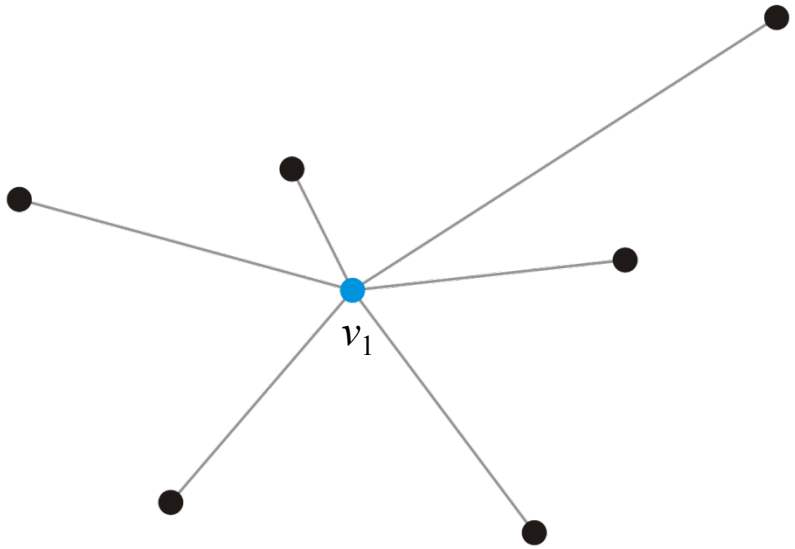
This topic covers Prim's algorithm:

- Finding a minimum spanning tree
- The idea and the algorithm
- An example

Strategy

Suppose we take a vertex

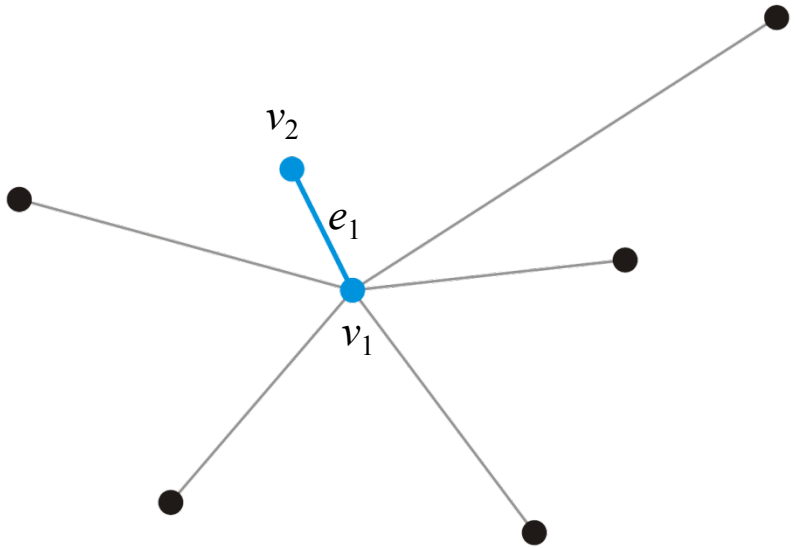
– Given a single vertex e_1, it forms a minimum spanning tree on one vertex



Strategy

Add that adjacent vertex v_2 that has a connecting edge e_1 of minimum weight

- This forms a minimum spanning tree on our two vertices and e_1 must be in any minimum spanning tree containing the vertices v_1 and v_2



Prim's Algorithm

Initialization:

- Select a root node and set its distance as 0
- Set the distance to all other vertices as ∞
- Set all vertices to being unvisited
- Set the parent pointer of all vertices to 0

Prim's Algorithm

Iterate while there exists an unvisited vertex with distance $< \infty$

- Select that unvisited vertex with minimum distance
- Mark that vertex as having been visited
- For each adjacent vertex, if the weight of the connecting edge is less than the current distance to that vertex:
  - Update the distance to equal the weight of the edge
  - Set the current vertex as the parent of the adjacent vertex

Prim's Algorithm

Halting Conditions:

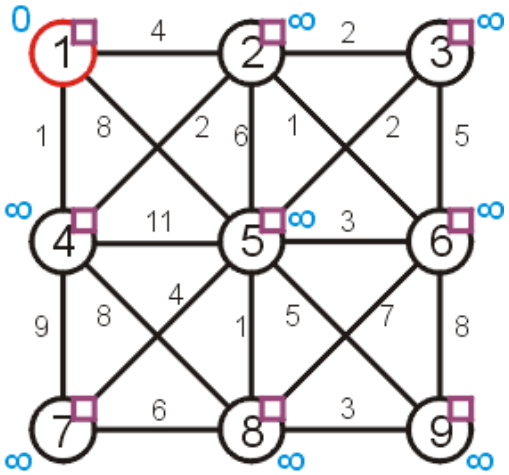
- There are no unvisited vertices which have a distance < ∞

If all vertices have been visited, we have a spanning tree of the entire graph

If there are vertices with distance ∞, then the graph is not connected and we only have a minimum spanning tree of the connected sub-graph containing the root

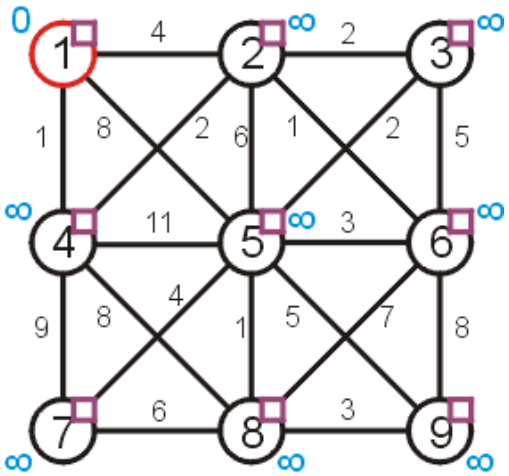
Prim's Algorithm

Let us find the minimum spanning tree for the following undirected weighted graph



Prim's Algorithm

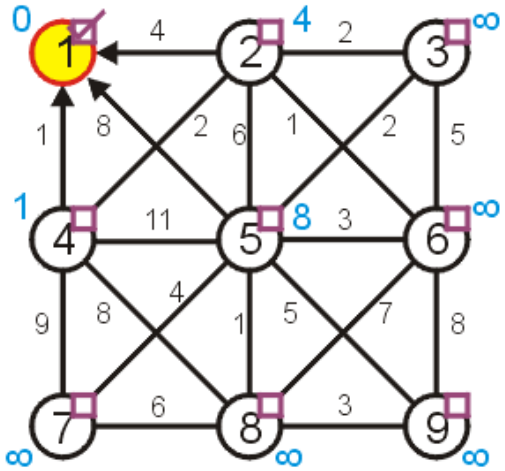
First we set up the appropriate table and initialize it



		Distance	Parent
1	F	0	0
2	F	∞	0
3	F	∞	0
4	F	∞	0
5	F	∞	0
6	F	∞	0
7	F	∞	0
8	F	∞	0
9	F	∞	0

Prim's Algorithm

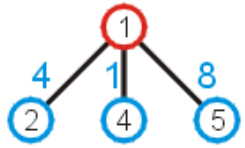
Visiting vertex 1, we update vertices 2, 4, and 5



		Distance	Parent
1	T	0	0
2	F	4	1
3	F	∞	0
4	F	1	1
5	F	8	1
6	F	∞	0
7	F	∞	0
8	F	∞	0
9	F	∞	0

Prim's Algorithm

What these numbers really mean is that at this point, we could extend the trivial tree containing just the root node by one of the three possible children:

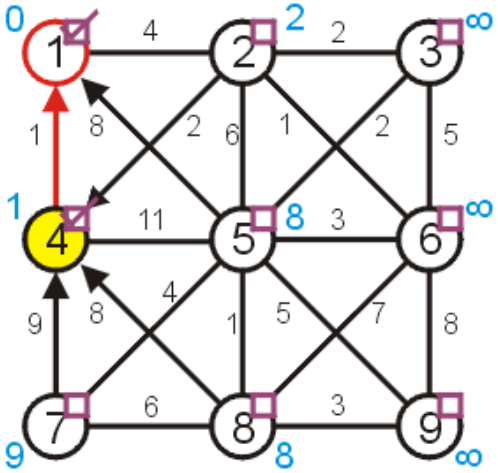


As we wish to find a *minimum* spanning tree, it makes sense we add that vertex with a connecting edge with least weight

Prim's Algorithm

The next unvisited vertex with minimum distance is vertex 4

- Update vertices 2, 7, 8
- Don't update vertex 5



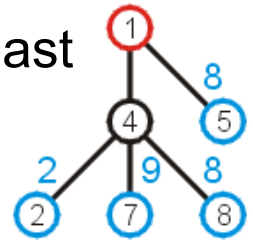
		Distance	Parent
1	T	0	0
2	F	2	4
3	F	∞	0
4	T	1	1
5	F	8	1
6	F	∞	0
7	F	9	4
8	F	8	4
9	F	∞	0

Prim's Algorithm

Now that we have updated all vertices adjacent to vertex 4, we can extend the tree by adding one of the edges

(1, 5), (4, 2), (4, 7), or (4, 8)

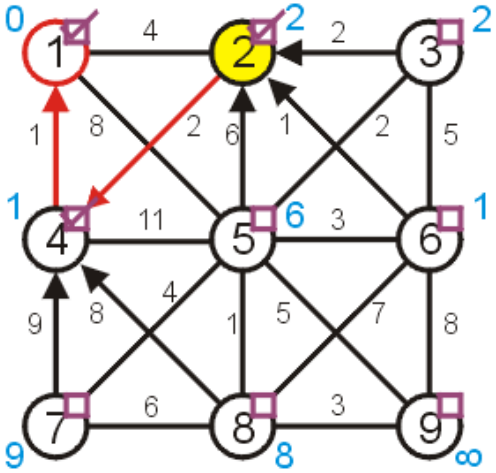
We add that edge with the least weight: (4, 2)



Prim's Algorithm

Next visit vertex 2

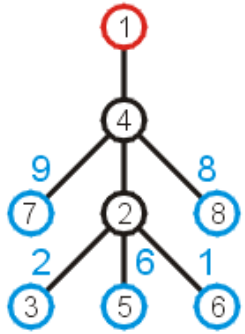
– Update 3, 5, and 6



		Distance	Parent
1	T	0	0
2	T	2	4
3	F	2	2
4	T	1	1
5	F	6	2
6	F	1	2
7	F	9	4
8	F	8	4
9	F	∞	0

Prim's Algorithm

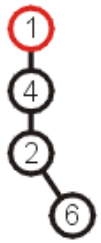
Again looking at the shortest edges to each of the vertices adjacent to the current tree, we note that we can add (2, 6) with the least increase in weight

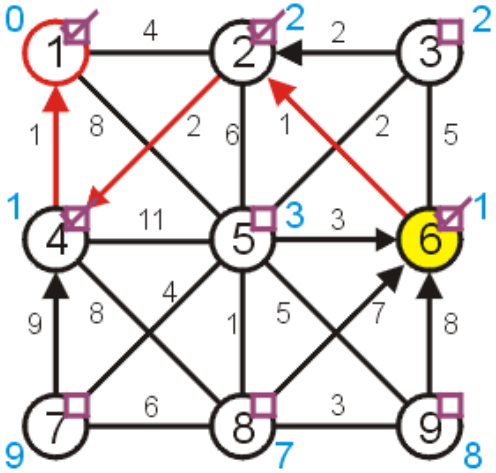


Prim's Algorithm

Next, we visit vertex 6:

- update vertices 5, 8, and 9



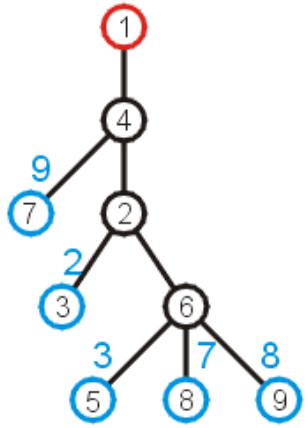


		Distance	Parent
1	T	0	0
2	T	2	4
3	F	2	2
4	T	1	1
5	F	3	6
6	T	1	2
7	F	9	4
8	F	7	6
9	F	8	6

Prim's Algorithm

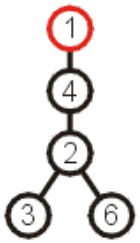
The edge with least weight is (2, 3)

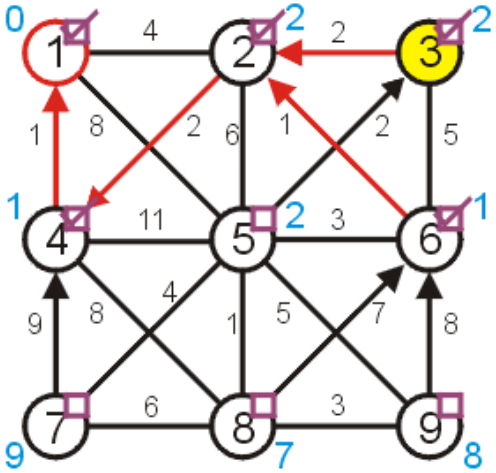
- This adds the weight of 2 to the weight minimum spanning tree



Prim's Algorithm

Next, we visit vertex 3 and update 5

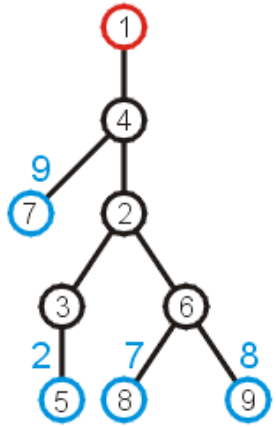




		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	F	2	3
6	T	1	2
7	F	9	4
8	F	7	6
9	F	8	6

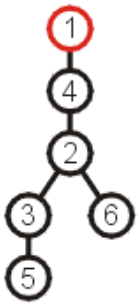
Prim's Algorithm

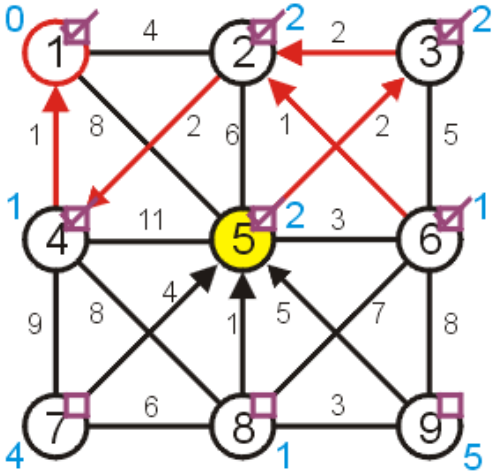
At this point, we can extend the tree by adding the edge (3, 5)



Prim's Algorithm

Visiting vertex 5, we update 7, 8, 9



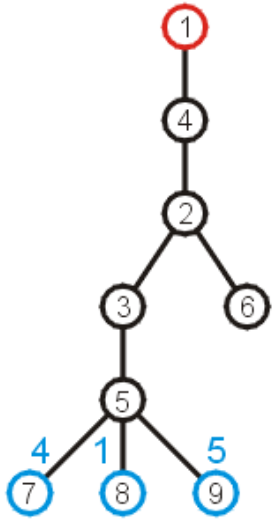


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	F	1	5
9	F	5	5

Prim's Algorithm

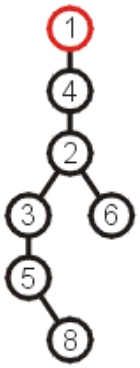
At this point, there are three possible edges which we could include which will extend the tree

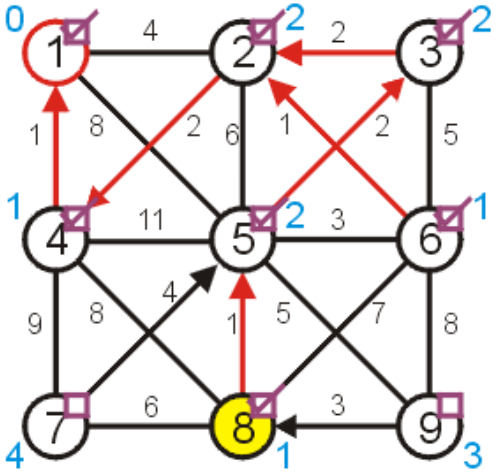
The edge to 8 has the least weight



Prim's Algorithm

Visiting vertex 8, we only update vertex 9

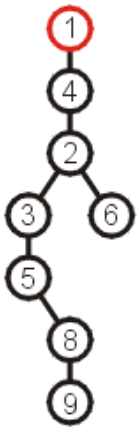


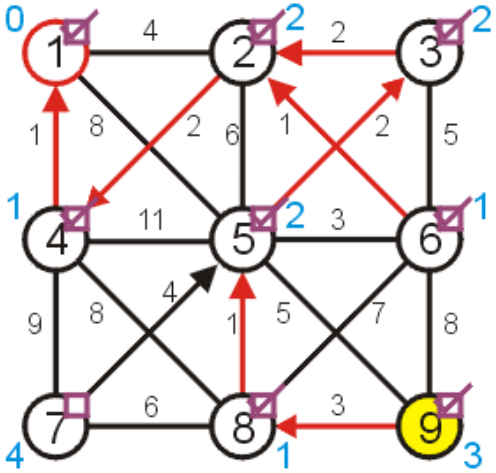


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	T	1	5
9	F	3	8

Prim's Algorithm

There are no other vertices to update while visiting vertex 9

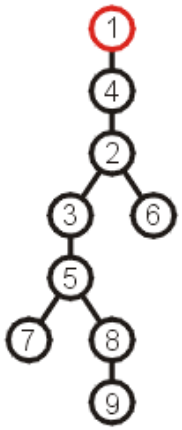


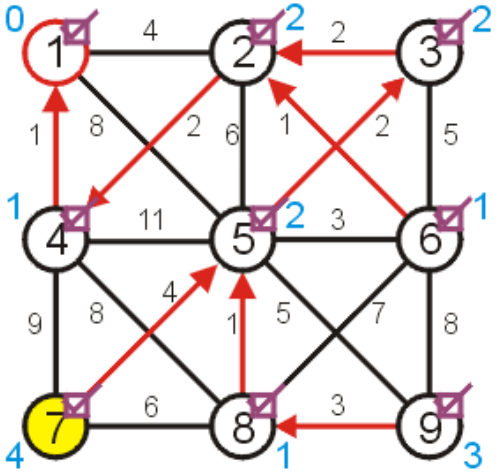


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	T	1	5
9	T	3	8

Prim's Algorithm

And neither are there any vertices to update when visiting vertex 7





		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	T	4	5
8	T	1	5
9	T	3	8

Prim's Algorithm

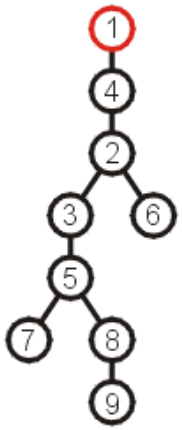
At this point, there are no more unvisited vertices, and therefore we are done

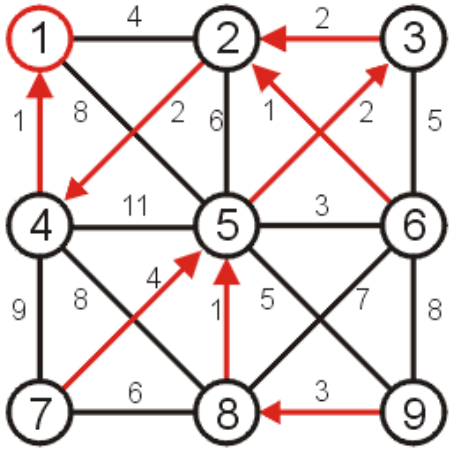
If at any point, all remaining vertices had a distance of ∞, this would indicate that the graph is not connected

- in this case, the minimum spanning tree would only span one connected sub-graph

Prim's Algorithm

Using the parent pointers, we can now construct the minimum spanning tree





		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	T	4	5
8	T	1	5
9	T	3	8

Prim's Algorithm

To summarize:

- we begin with a vertex which represents the root
- starting with this trivial tree and iteration, we find the shortest edge which we can add to this already existing tree to expand it

This is a reasonably efficient algorithm: the number of visits to vertices is kept to a minimum

Implementation and analysis

The initialization requires $\Theta(|V|)$ memory and run time

We iterate $|V| - 1$ times, each time finding the *closest* vertex

- Iterating through the table requires is $\Theta(|V|)$ time
- Each time we find a vertex, we must check all of its neighbors
- With an adjacency matrix, the run time is $\Theta(|V|(|V| + |V|)) = \Theta(|V|^2)$
- With an adjacency list, the run time is $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$ as $|E| = \mathrm{O}(|V|^2)$

Can we do better?

- Recall, we only need the shortest edge next
- How about a priority queue?
  - Assume we are using a binary heap
  - We will have to update the heap structure—this requires additional work

Implementation and analysis

The initialization still requires $\Theta(|V|)$ memory and run time

- The priority queue will also requires $O(|V|)$ memory

We iterate $|V| - 1$ times, each time finding the *closest* vertex

- Place the shortest distances into a priority queue
- The size of the priority queue is $O(|V|)$
- Thus, the work required for this is $O(|V| \ln(|V|))$

Is this all the work that is necessary?

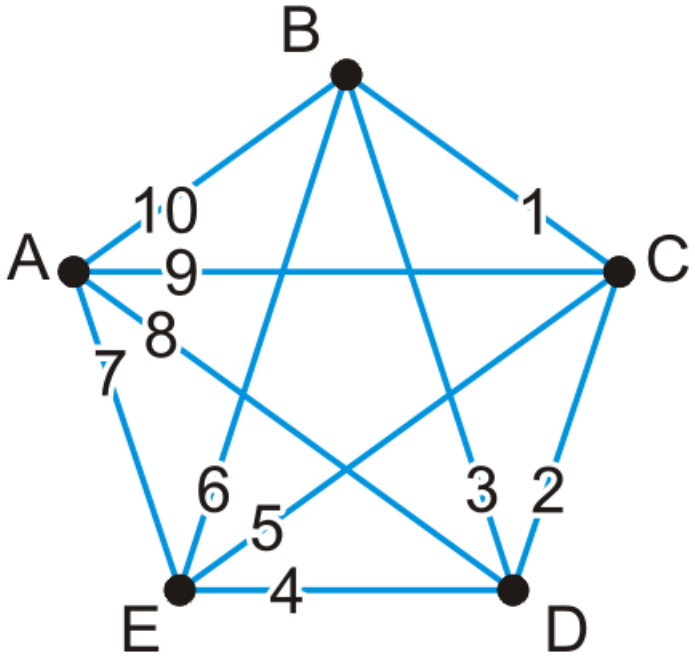
- Recall that each edge visited may result in a new edge being pushed to the very top of the heap
- Thus, the work required for this is $O(|E| \ln(|V|))$

Thus, the total run time is $O(|V| \ln(|V|) + |E| \ln(|V|)) = O(|E| \ln(|V|))$

Implementation and analysis

Here is a worst-case graph if we were to start with Vertex A

- Assume that the adjacency lists are in order
- Each time, the edge is percolated to the top of the heap



Implementation and analysis

We could use a different heap structure:

- A Fibonacci heap is a node-based heap
- Pop is still $O(\ln(|V|))$, but inserting and moving a key is $\Theta(1)$
- Thus, because we are only calling pop $|V| - 1$ times, work required reduces to $O(|E| + |V| \ln(|V|))$
- Thus, the overall run-time is $O(|E| + |V| \ln(|V|))$

Implementation and analysis

Thus, we have two run times when using

- A binary heap: $O(|E| \ln(|V|))$
- A Fibonacci heap: $O(|E| + |V| \ln(|V|))$

Questions: Which is faster if $|E| = \Theta(|V|)$? How about if $|E| = \Theta(|V|^2)$?

Summary

We have seen an algorithm for finding minimum spanning trees

- Start with a trivial minimum spanning tree and grow it
- An alternate algorithm, Kruskal's algorithm, uses a different approach

Prim's algorithm finds an edge with least weight which grows an already existing tree

- It solves the problem in $O(|E| \ln(|V|))$ time

References

Wikipedia, http://en.wikipedia.org/wiki/Minimum_spanning_tree
Wikipedia, http://en.wikipedia.org/wiki/Prim's_algorithm