Graphs: Minimum Spanning Trees

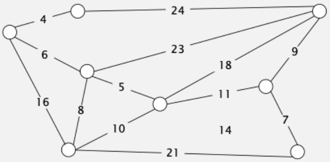
Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 4.3), Prof. Janicki's course slides and https://www.cs.princeton.edu/~rs/AlgsDS07/14MST.pdf

Edge weighted Graph

- An edge-weighted graph is an undirected graph model where we associate weights or costs with each edge.



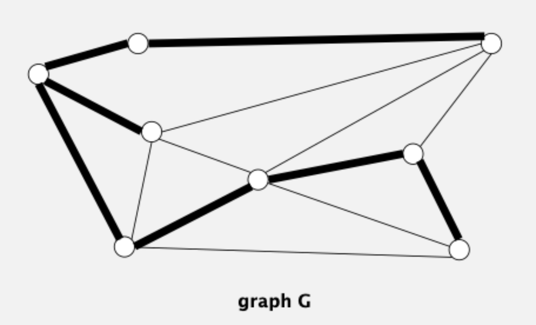
edge-weighted graph G

- Given. Undirected graph G with positive edge weights (connected).
- Goal. Find a min weight set of edges that connects all of the vertices.

Spanning tree

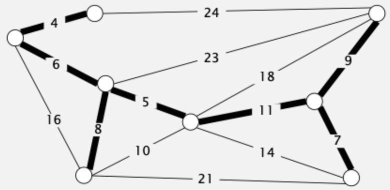
A spanning tree of a graph G is a subgraph T that is:

- Connected.
- Acyclic.
- Includes all of the vertices.



Minimum spanning tree

A **minimum spanning tree (MST)** of an edge-weighted undirected graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree.



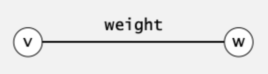
minimum spanning tree T
(cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7)

Brute force. Try all spanning trees? No!
Solution. Use greedy approach.
Simplifying assumption. All edge weights we are distinct.

Weighted edge API

`public class Edge implements Comparable<Edge>`

	`Edge(int v, int w, double weight)`	*create a weighted edge v-w*
`int`	`either()`	*either endpoint*
`int`	`other(int v)`	*the endpoint that's not v*
`int`	`compareTo(Edge that)`	*compare this edge to that edge*
`double`	`weight()`	*the weight*
`String`	`toString()`	*string representation*



Idiom for processing an edge e: int $v = e.either(),\ w = e.other(v);$

Weighted edge: Java implementation

```
public class Edge implements Comparable<Edge>
{
   private final int v, w;
   private final double weight;

   public Edge(int v, int w, double weight)
   {
      this.v = v;
      this.w = w;
      this.weight = weight;
   }

   public int either()
   {  return v;  }

   public int other(int vertex)
   {
      if (vertex == v) return w;
      else return v;
   }

   public int compareTo(Edge that)
   {
      if      (this.weight < that.weight) return -1;
      else if (this.weight > that.weight) return +1;
      else                                return  0;
   }
}
```

constructor

either endpoint

other endpoint

compare edges by weight

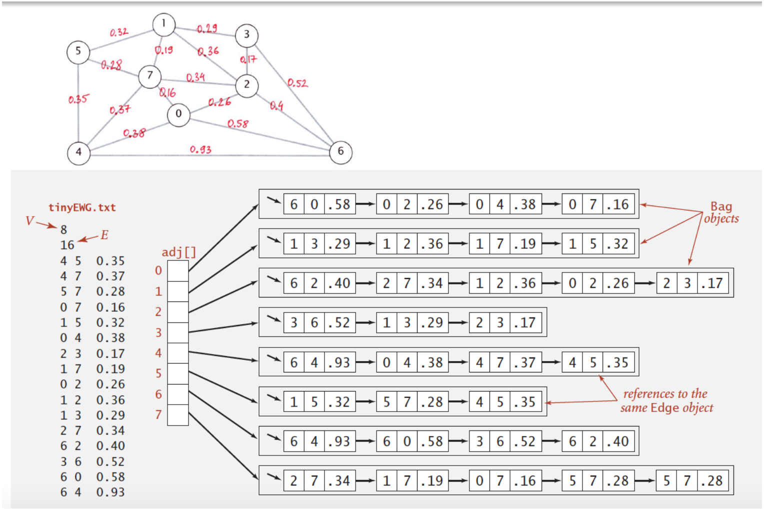
Edge-weighted graph API

```
public class EdgeWeightedGraph
```

	`EdgeWeightedGraph(int V)`	*create an empty graph with V vertices*
	`EdgeWeightedGraph(In in)`	*create a graph from input stream*
`void`	`addEdge(Edge e)`	*add weighted edge e to this graph*
`Iterable<Edge>`	`adj(int v)`	*edges incident to v*
`Iterable<Edge>`	`edges()`	*all edges in this graph*
`int`	`V()`	*number of vertices*
`int`	`E()`	*number of edges*
`String`	`toString()`	*string representation*

Conventions. Allow self-loops and parallel edges.

Edge-weighted graph: adjacency-lists representation

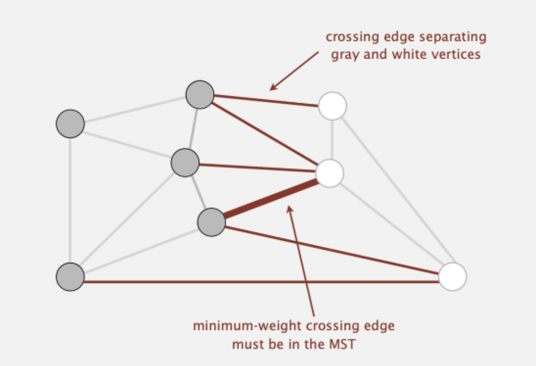


Cut Property

A **cut** in a graph is a partition of its vertices into two (nonempty) disjoint sets (for example: S and $V - S$).

A **crossing edge** connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.



Cut Property: correctness proof

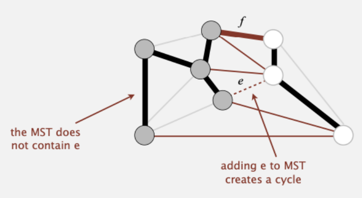
A **cut** in a graph is a partition of its vertices into two (nonempty) sets.

A **crossing edge** connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.

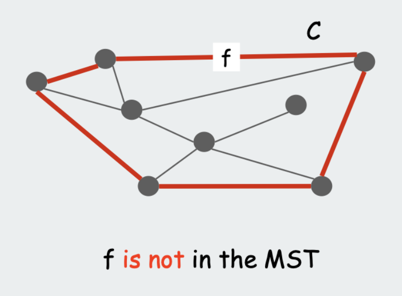
Pf. Suppose min-weight crossing edge e is not in the MST.

- Adding e to the MST creates a cycle.
- Some other edge f in cycle must be a crossing edge.
- Removing f and adding e is also a spanning tree.
- Since weight of e is less than the weight of f, that spanning tree is lower weight. f – a Contradiction.



Cycle Property

Cycle Property Let C be any cycle, and let f be the max. cost edge belonging to C. Then the MST does not contain f.

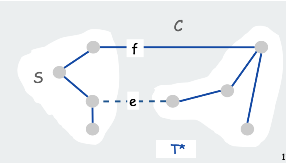


Cycle Property: correctness proof

Cycle Property Let C be any cycle, and let f be the max. cost edge belonging to C. Then the MST does not contain f.

Pf. [by contradiction]

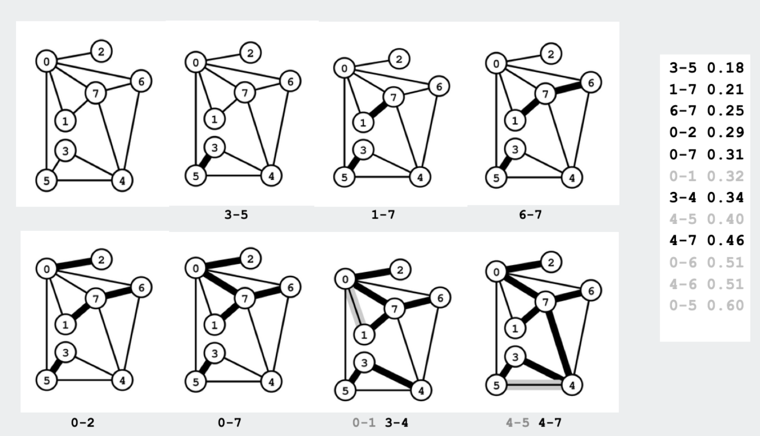
- Suppose f belongs to T^*. Let's see what happens.
- Deleting f from T^* disconnects T^*.
- Let S be one side of the cut.
- Some other edge in C, say e, has exactly one endpoint in S.
- $T = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, $cost(T) < cost(T^*)$, where c_e and c_f are the costs associated with the edges e, f.
- Contradicts minimality of T^*.



Kruskal's algorithm (see demo)

Consider edges in ascending order of weight.

- Add next edge to tree T unless doing so would create a cycle.

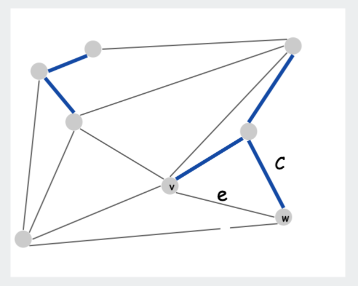


Kruskal's algorithm: correctness proof

Proposition. Kruskal's algorithm computes the MST.

Pf. [case 1] Suppose that adding e to T creates a cycle C

- e is the max weight edge in C (weights come in increasing order)
- e is not in the MST (cycle property)

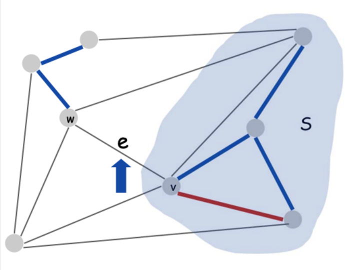


Kruskal's algorithm: correctness proof

Proposition. Kruskal's algorithm computes the MST.

Pf. [case 2] Suppose that adding $e = (v, w)$ to T does not create a cycle

- Let S be the vertices in v's connected component
- w is not in S
- e is the min. weight edge with exactly one endpoint in S
- e is in the MST (cut property)



Krushkal's algorithm implementation challenge I

Q. How to check if adding an edge $v - w$ to T would create a cycle?

A1. Naive solution: use DFS from w on T to check if v is reachable

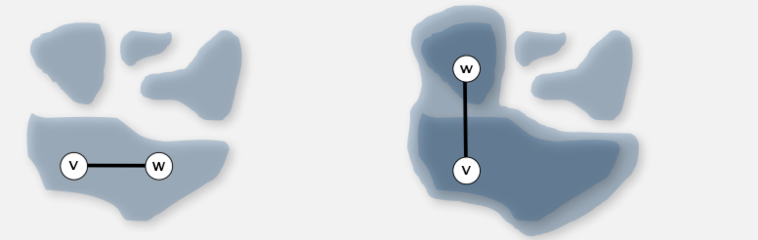
- $O(|V|)$ time per cycle check, as T has at most $|V| - 1$ edges and $|V|$ vertices.
- $O(|E||V|)$ time overall.

Krushkal's algorithm implementation challenge II

Q. How to check if adding an edge to T would create a cycle?

Efficient Solution Use the union-find data structure

- Maintain a set for each connected component.
- If v and w are in same component, then adding $v - w$ creates a cycle.
- To add $v - w$ to T, merge sets containing v and w.



Case 1: adding v–w creates a cycle   Case 2: add v–w to T and merge sets containing v and w

Krushkal's algorithm implementation

- Use Min. priority queue data structure to store all $|E|$ edges.
- Perform delete min. operation to examine the edges.
- While examining an edge e, check for cycle (using the union-find data structure).
- If e does not create a cycle in the minimum spanning tree T, then add e to T
- Continue until $|V| - 1$ edges are added to T

Krushkal's algorithm: time complexity

Proposititon: Kruskal's algorithm can compute Minimum Spanning Tree in $O(|E| \log |E|)$ time.

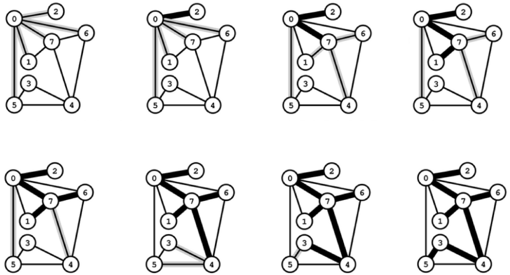
Pf outline.

- To build the Min. priority queue to store all $|E|$ edges - takes $O(|E|)$ time.
- Total time for all delete-min operations is at most $O(|E| \log |E|)$
- Checking for the presence of cycles using Union-Find takes $O(|E| + |V|)$

Note that, Union-Find with path compression takes $O(M + N)$ time starting from an empty data structure, where $N =$ total no.of objects, $M =$ total operations executed. In this case $N = |V|$ vertices, and $M = |E|$ find operations and $|V|$ union operations.

Prim's algorithm (see demo)

- Start with vertex 0 and greedily grow tree T
- Consider edges incident on the vertices in T, but disregard any edge with both end points in T, then
- Add to T the min weight edge with exactly one endpoint in T.
- Repeat until $V - 1$ edges.



0-1	0.32
0-2	0.29
0-5	0.60
0-6	0.51
0-7	0.31
1-7	0.21
3-4	0.34
3-5	0.18
4-5	0.40
4-6	0.51
4-7	0.46
6-7	0.25

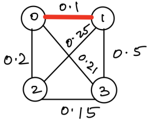
Prim's algorithm - Explanation of the example in previous slide

- Start with vertex 0, therefore $S = \{0\}$ and the MST T={}.
- Consider all the edges incident on 0, and add the min.weight edge to T. In the previous example, this edge is 0-2. Therefore, $S = \{0, 2\}$ and $T = \{0 - 2\}$.
- Now consider the edges incident on 0 and 2, disregarding any edge with two end points in S; that is, the edge $0 - 2$. We add the min.weight edge to T; that is the edge 0-7. Therefore, $S = \{0, 2, 7\}$ and $T = \{0 - 2, 0 - 7\}$.
- Repeat the above procedure until $V - 1$ edges are added to T.

Prim's algorithm: Another example I

When $S = \{0\}$, the minimum weighted edge $(0-1)$ out of all the edges incident to 0 is added to MST (highlighted in red).

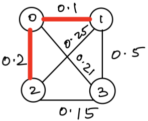
$S = \{0\}$



→ 0-1   0.1 ✓
→ 0-2   0.2
→ 0-3   0.21
1-2   0.25
1-3   0.5
2-3   0.15

Now $S = \{0, 1\}$. The minimum weighted edge $(0-2)$ out of all the edges incident to 0 and 1 (and which have only one end point in S) is added to MST (highlighted in red).

$S = \{0, 1\}$

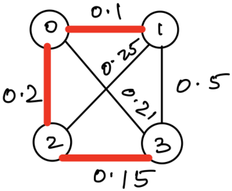


× 0-1   0.1
→ 0-2   0.2 ✓
→ 0-3   0.21
→ 1-2   0.25
→ 1-3   0.5
2-3   0.15

Prim's algorithm: Another example II

Now $S = \{0, 1, 2\}$. The minimum weighted edge $(2 - 3)$ out of all the edges incident to $0, 1$ and 2 (and which have only one end point in S) is added to MST (highlighted in red).

$S = \{0, 1, 2\}$



× 0-1 0·1
× 0-2 0·2
→ 0-3 0·21
× 1-2 0·25
→ 1-3 0·5
→ 2-3 0·15 ✓

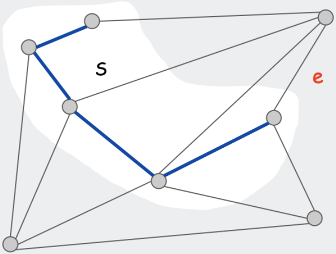
$S = \{0, 1, 2, 3\}$, and since we have added $|V| - 1 = 3$ edges to our MST, we are done.

Prim's algorithm: proof of correctness

Proposition. Prim's algorithm computes the MST.

Pf.

- Let S be the subset of vertices in current tree T.
- Prim adds the cheapest edge e with exactly one endpoint in S.
- e is in the MST (cut property) ■



Prim's algorithm implementation Challenge I

Q. How to find cheapest edge with exactly one endpoint in T?

A. Brute force: try all edges

- O(E) time per spanning tree edge.
- O(EV) time overall.

A2. Maintain a priority queue. Two choices:

- Lazy implementation - PQ stores edges incident on the vertices in S - takes $O(|E| \log |E|)$
- Eager implementation - PQ stores vertices adjacent to the vertices in S - takes $O(|E| \log |V|)$

Prim's algorithm: Lazy implementation (see Demo)

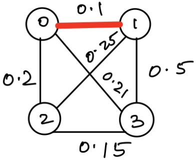
Lazy solution. Maintain a PQ of edges with (at least) one endpoint in T.

- Key = edge; priority = weight of edge.
- Delete-min edge $e = v - w$ from PQ to determine the next edge to add to T.
- Disregard if both endpoints v and w are marked (both in T).
- Otherwise, let w be the unmarked vertex (not in T):
    - add to PQ any edge incident to w (assuming other endpoint not in T)
    - add e to T and mark w

Prim's algorithm: Lazy implementation Example I

Initially, $s = \{0\}$ - add all edges incident to 0 to PQ. Then delete the min. weighted edge from PQ and add to MST (highlighted in red).

$S = \{0\}$



PQ

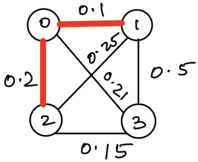
0-1
0-2
0-3

0-1	0·1
0-2	0·2
0-3	0·21
1-2	0·25
1-3	0·5
2-3	0·15

Prim's algorithm: Lazy implementation Example II

Then $S = \{0, 1\}$ - add all the edges incident to 1, such that both the edge vertices are not in S. Delete the min. weighted edge from PQ and add to MST (highlighted in red).

$S = \{0,1\}$



PQ

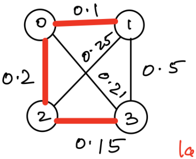
0-2
0-3
1-2
1-3

0-1	0.1
0-2	0.2
0-3	0.21
1-2	0.25
1-3	0.5
2-3	0.15

Prim's algorithm: Lazy implementation Example III

Then $S = \{0, 1, 2\}$ - add all the edges incident to 2, such that both the edge vertices are not in S. Delete the min. weighted edge from PQ and add to MST (highlighted in red).

$S = \{0, 1, 2, 3\}$



PQ

2-3

0-3

1-2

1-3

obsolete edge

lazy implementation leaves on PQ.

0-1	0.1
0-2	0.2
0-3	0.21
1-2	0.25
1-3	0.5
2-3	0.15

Lazy Prim's algorithm: complexity

Proposition. Lazy Prim's algorithm computes the MST in time proportional to $|E| \log |E|$.

Pf.

operation	frequency	binary heap
delete min	E	$\log E$
insert	E	$\log E$

Obsolete edges cause an increase in the running time.

Prim's algorithm: Eager implementation (see Demo)

Challenge. Find min weight edge with exactly one endpoint in T.

Eager solution. Maintain a PQ of vertices connected by an edge to T, where priority of vertex $v =$ min. weighted edge connecting v to T.

- Delete min vertex v, mark v to be in T.
- Update PQ by considering all edges $e = v - x$ incident to v
  - ignore if x is already in T
  - add x to PQ if not already on it
  - if already on PQ, then reduce priority of x if $v - x$ becomes the min. weighted edge connecting x to T

Prim's algorithm: Eager implementation Example I

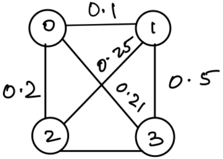
We begin with $S = \{0\}$, and add $0(0.0)$ to the PQ; that is the vertex 0, with weight 0.0. The corresponding $edgeTo[0]$ and $distTo[0]$ are left empty.

Then, we delete the min.weighted vertex (0) (marked in gray) form PQ, and add all the vertices adjacent to 0 along with the weight of the edges connecting them to 0.

We also update the $edgeTo[1] = 0-1, edgeTo[2] = 0-2, edgeTo[3] = 0-3$ and $distTo[1] = 0.1, distTo[1] = 0.2, distTo[1] = 0.21$.

min. edge (weight for ordering)

$S = \{0\}$



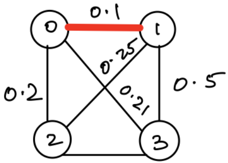
PQ (v)	EdgeTo[]	distTo[]
0(0)	—	—
→ 1(0.1)	0-1	0.1
2(0.2)	0-2	0.2
3(0.21)	0-3	0.21

→ 0-1	0.1
→ 0-2	0.2
→ 0-3	0.21
1-2	0.25
1-3	0.5
2-3	0.15

Prim's algorithm: Eager implementation Example II

- Then we delete the min. weighted vertex $1(0.1)$ (marked in gray) from PQ and add $edgeTo[1] = 0 - 1$ to the MST (highlighted in red), and update $S = \{0, 1\}$.
- We examine the vertices adjacent to 1; that is $\{0, 2, 3\}$. Since 0 is in T, we ignore it. Since $2, 3$ are on PQ and because the edge weights for edges $1 - 2$ and $1 - 3$ are more than the weights of edges in $edgeTo[]$ array, we ignore them.

$S = \{0, 1\}$



min. edge (weight for ordering)

PQ (v)	EdgeTo[]	distTo[]
0 (0)	—	—
1 (0.1)	0−1	0.1
→ 2 (0.2)	0−2	0.2
3 (0.21)	0−3	0.21

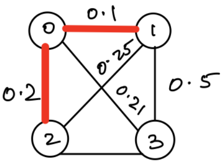
0−1	0.1
0−2	0.2
0−3	0.21
→ 1−2	0.25
→ 1−3	0.5
2−3	0.15

Prim's algorithm: Eager implementation Example III

- Then we delete the min. weighted vertex $2(0.2)$ (marked in gray) from PQ and add $edgeTo[2] = 0 - 2$ to the MST (highlighted in red), and update $S = \{0, 1, 2\}$.

- We examine the vertices adjacent to 2; that is $\{0, 1, 3\}$. Since $0, 1$ are in T, we ignore them. Since 3 is on PQ and because the edge weight of the edge $2 - 3 = 0.15 < 0.21$ the edge weight of $0 - 3$, we replace $edgeTo[3] = 2 - 3$ and $distTo[3] = 0.15$.

$S = \{0, 1, 2\}$



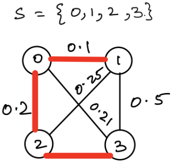
min. edge (weight for ordering)

PQ (v)	EdgeTo[]	distTo[]
0 (0)	—	—
1 (0.1)	0−1	0.1
2 (0.2)	0−2	0.2
→ 3 (~~0.21~~) (0.15)	~~0−3~~ 2−3	~~0.21~~ 0.15

0−1	0.1
0−2	0.2
0−3	0.21
→ 1−2	0.25
→ 1−3	0.5
2−3	0.15

Prim's algorithm: Eager implementation Example VI

- Then we delete the min. weighted vertex $3(0.15)$ (marked in gray) from PQ and add $edgeTo[3] = 2 - 3$ to the MST (highlighted in red), and update $S = \{0, 1, 2, 3\}$.
- We examine the vertices adjacent to 3; that is $\{0, 1, 2\}$. Since $0, 1, 2$ are in T, we ignore them.
- Since the PQ is empty and $V - 1$ edges are added to T we exit.



min. edge (weight for ordering)

PQ (v)	EdgeTo[]	distTo[]
0 (0)	—	—
1 (0.1)	0-1	0.1
2 (0.2)	0-2	0.2
3 (0.15)	2-3	0.15

PQ Empty exit

	0-1	0.1
	0-2	0.2
	0-3	0.21
→	1-2	0.25
→	1-3	0.5
	2-3	0.15

Eager Prim's algorithm: Time complexity

The eager version of Prim's algorithm uses extra space proportional to $|V|$ and time proportional to $|E| \log |V|$ (in the worst case) to compute the MST of a connected edge weighted graph with E edges and V vertices.

Kruskal's and Prim's Comparison

- Adds edges greedily to create a minimum spanning tree.
- Adds nodes greedily. Initially adds a source node to MST. Then greedily extends the MST by adding the least weighted edge extending it. Thus at each iteration it adds a new node to the MST.