# MATH 308

## Mohamed Kasem

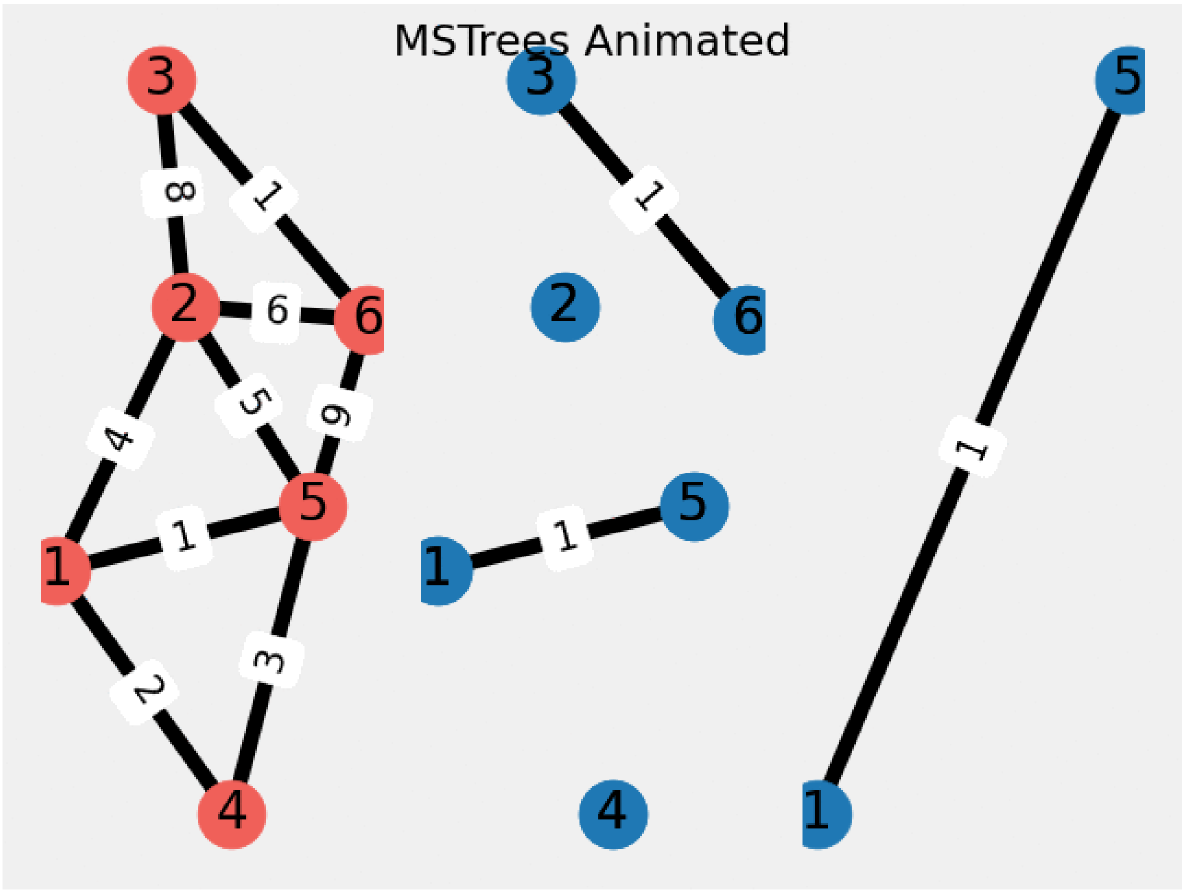
**ID: 201601144**

**01127161185**

**s-mohamed.kasem@zewailcity.edu.eg**

# Minimum Spanning Trees

### NOTE: !!!!! For this animation, open report.html instead of report.pdf !!!!!



## To run the code

### Install required packages

Inside the folder containing this, you will find `requirements.txt` file. This file contains all the required packages along with their version number.

#### On Linux

```
pip3 install -r requirements.txt
```

**OR**

```
pip3 install -r requirements.txt --no-index --find-links file:///tmp/packages
```

#### On Windows

**YOU MUST HAVE CONDA**

```
conda install --file requirements.txt
```

There are 2 options to run the code, either it reads data from a **CSV file** (excel file) or data is entered through the **CLI**. The graph in the GIF above is already available and the data is filled in; however, the position of the nodes on the screen is randomized every time the program runs so it might look different on your computer.

## Launching the code and simulation

### To read data from a .csv file

```
python3 MSTrees.py -f <CSV_FILE_NAME>
```

#### Example

```
python3 MSTrees.py -f sample.csv
```

This will execute the python script which will load data from "sample.csv" which contains the graph animated above.

### To input data manually

```
python3 MSTrees.py
```

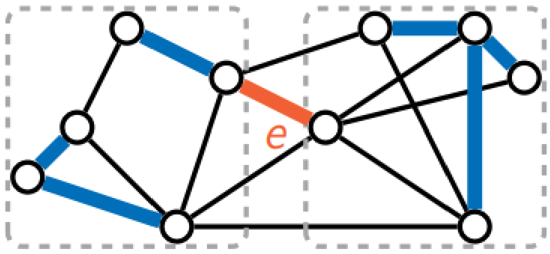
launching the script without any parameters will execute it in default mode and you will be prompted to input the data accordingly.

## Cut property

Before we delve into the algorithms, we must first go through a very important property that both algorithms will depend on, namely, cut property.

## Cut property

Let $X \subseteq E$ be a part of a MST of $G(V, E)$, $S \subseteq V$ be such that no edge of $X$ crosses between $S$ and $V - S$, and $e \in E$ be a lightest edge across this partition. Then $X + \{e\}$ is a part of some MST.

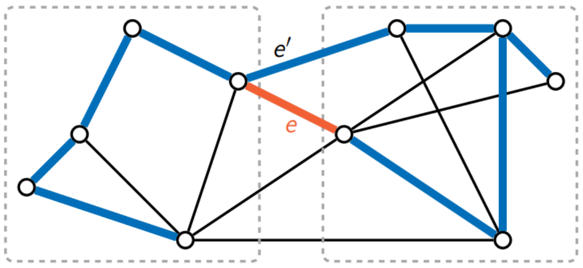


cut property states that $X + \{e\}$ is also a part of some MST

### Proof

we need to find $X$; $X \subseteq E$ of some MST $T$.

1. partition of $V$ into $S$ and $V - S$. i. e. 2 disjoint sets of vertices.
2. choose $e$ the lightest edge.
   - since $X$ belongs to some MST $T$ what we need to prove is that $X + e$ is also a part of a (possibly different) MST
3. Assuming $e$ doesn't already belong to $T$, in which case we'd be done.
   - Adding $e$ to $T$ creates a cycle; let $e'$ be an edge of this cycle that crosses $S$ and $V - S$



4. Then $T' = T - e' + e$ is an MST containing X + {e}: it is a tree, and $w(T') \leq w(T)$ since $w(e) \leq w(e')$

## Kruskal's algorithm

This algorithm operates on edges instead of nodes. The algorithm goes through the following steps:

1. **sort edges by weight in non-decreasing order.**

2. **for each edge check if it doesn't produce a cycle.**
3. **if the edge doesn't produce a cycle add it and repeat.**

For checking if an edge will produce a cycle, the easiest and fastest way is to start with $V$ disjoint sets. Upon adding a new edge, we merge the sets of the nodes on both ends of the edge into 1 set. An edge will produce a cycle if it's nodes lie in the same set, i.e. in the same connected component. In other words, for edge $\{u, z\}$ `if find(u) == find(z)` then the edge produces a cycle. This same approach is used in the code.

### Running Time Analysis

**Sorting edges:**

$$O(|E| log |E|) = O(|E| log |V|^2) = O(2|E| log |V|) = O(|E| log |V|)$$

**Processing edges:**

$$2|E| \cdot T(Find) + |V| \cdot T(Union) = O((|E| + |V|) log |V|) = O(|E| log |V|)$$

**Total running time:**

$$O(|E| log |V|)$$

# Prim’s algorithm

This algorithm operates on nodes as opposed to edges. Prim's approach to this problem is very similar to Dijkstra's algorithm for finding the shortest path. In fact, this problem could be viewed as the shortest path between all nodes and that the tree itself is the shortest path that connects all the vertices on the graph.

The algorithms goes as follows:

1. Initiate all nodes with $nil$ parent and $\infty$ cost
2. pick any initial node and set its cost to $0$.
3. Make a Priority Queue (e. g. min heap structure) out of $V$
4. Repeatedly add nodes that are not in the tree whose edge weight is less than their cost. This can be done as follows:
    - while Priority Queue is not empty.
    - pop the node $u$ with the least cost (i. e. top of the min heap)
    - relax all the edges $\{u, z\}$ coming out of $u$.
    - if cost of $z > W(u, z)$ i. e. the weight of the edge between $u, z$
        - update the weight and parent of $z$ t as $W(u, z)$ and $u$ accordingly.
        - update the Priority Queue i. e. (re-heap) with newly calculated cost of $z$

### Running Time Analysis

**Total running time:**

$$|V| \cdot T(ExtractMin) + |E| \cdot T(ChangePriority)$$

**ExtractMin, Change Priority:**

**For array-based implementation:**

$$O(|V|^2)$$

**For binary heap-based implementation**

$$O((|V| + |E|)log|V|) = O(|E|log|V|)$$

## References

1- **Coursera course:** Algorithms on Graphs

> by University of California San Diego & National Research University Higher School of Economics

## Code Appendex

### Core.py file containing the algorithms

```python
from collections import defaultdict
from functools import partial
from copy import deepcopy
import pandas as pd
import heapq
from math import inf
from operator import itemgetter


class Graph:
    def __init__(self, sample_csv=None):
        if not sample_csv:
            self.get_user_input()
        else:
            edges_df = pd.read_csv(sample_csv)
            self.num_vertices = edges_df.max().drop("weight").max()
            self.num_edges = len(edges_df)
            edges = set()
            for _, v1, weight, v2 in edges_df.itertuples():
                edges.add((v1, weight, v2))
            self.sorted_edges = sorted(edges, key=lambda x: x[1])

        self.disjoint_v_sets = [{i} for i in range(1, self.num_vertices + 1)]

    def get_user_input(self):
        # Obtain graph description
        self.num_vertices = int(input("Number of vertices: "))
        self.num_edges = int(input("Number of Edges: "))
        edges = set()
        print("====== Input edges like (v1, weight, v2) ======")
        while self.num_edges > 0:
            user_input = input(">>> ").split(",")
            try:
                v1, weight, v2 = tuple(map(int, user_input))
            except:
                print("Invalid input! Re-Enter edge")
                continue

            edges.add((v1, weight, v2))
            self.num_edges -= 1
        assert(len(edges) != self.num_edges)
        self.sorted_edges = sorted(edges, key=lambda x: x[1])


class Kruskal():
    def __init__(self, graph):
        self.graph = graph
        self.backup_disjoint_sets = deepcopy(self.graph.disjoint_v_sets)
        self.find_vertex = partial(find, self.graph.disjoint_v_sets)
        self.tree = set()

    def compute(self):
        for edge_indx, edge in enumerate(self.graph.sorted_edges):
            v1, _, v2 = edge
            v1_index, v2_index = self.find_vertex(v1), self.find_vertex(v2)
            if v1_index != v2_index:
                self.tree.add(edge)

 self.graph.disjoint_v_sets[v1_index].update(self.graph.disjoint_v_sets[v1_index
].union(
                    self.graph.disjoint_v_sets[v2_index]))

                del self.graph.disjoint_v_sets[v2_index]
        self.tree = sorted(self.tree, key=itemgetter(1))
        print("Output Tree: ", self.tree)
        self.animation_iterator = iter(self.tree)

    def compute_stepped(self):
        try:
            edge = next(self.animation_iterator)
        except StopIteration:
            print("Starting Over")
            self.animation_iterator = iter(self.tree)
            return None
        return edge

    def add_graph_nodes(self, nx_graph):
        # for component in self.backup_disjoint_sets:
        nx_graph.add_nodes_from(list(range(1, self.graph.num_vertices)))

    def add_graph_edge(self, nx_graph):
        edge = self.compute_stepped()
        if edge:
            nx_graph.add_edge(edge[0], edge[-1], weight=edge[1])
            return False
        else:
            nx_graph.remove_edges_from(nx_graph.edges)
            return True

    def add_all_graph_edges(self, nx_graph):
        # for edge in self.graph.sorted_edges:
        #     nx_graph.add_edge(edge[0], edge[-1], weight=edge[1])
        nx_graph.add_weighted_edges_from([(edge[0], edge[-1], edge[1])
```

```python
                                                  for edge in self.graph.sorted_edges])


class Prim(Graph):
    def __init__(self, sample_csv=None, graph=None):
        if graph:
            self.num_edges = graph.num_edges
            self.num_vertices = graph.num_vertices
            self.sorted_edges = graph.sorted_edges
        else:
            super().__init__(sample_csv=sample_csv)
        self.node_list = [[inf, i, None] for i in range(
            1, self.num_vertices + 1)]  # cost, name, parent
        self.adjacency_list = defaultdict(list)
        self.tree = list()
        for v1, w, v2 in self.sorted_edges:
            self.adjacency_list[v1].append([v2, w])
            self.adjacency_list[v2].append([v1, w])

    def compute(self):
        # setting cost of first node to zero
        self.node_list[0][0] = 0
        heapq.heapify(self.node_list)
        while len(self.node_list) > 0:
            self.tree.append(heapq.heappop(self.node_list))
            v = self.tree[-1][1]
            for neighbor in self.adjacency_list[v]:
                z, w = neighbor
                z_node = self.get_node(z)

                if z_node and z_node[0] > w:
                    z_node[0] = w
                    z_node[-1] = v
                    heapq.heapify(self.node_list)

        print("Output Tree: node(cost, name, parent) >> ", self.tree)
        self.animation_iterator = iter(self.tree)

    def compute_stepped(self):
        try:
            node = next(self.animation_iterator)
        except StopIteration:
            print("Starting Over")
            self.animation_iterator = iter(self.tree)
            return None
        return node

    def add_graph_node(self, nx_graph):
        # for component in self.backup_disjoint_sets:
        node = self.compute_stepped()
        if node:
            nx_graph.add_node(node[1])
            if node[-1]:
                nx_graph.add_edge(node[1], node[-1], weight=node[0])
            return False
        else:
            # nx_graph.remove_edges_from(nx_graph.edges)
            nx_graph.remove_nodes_from(deepcopy(nx_graph.nodes))
```

```
                return True

    def add_all_graph(self, nx_graph):
        nx_graph.add_weighted_edges_from([(edge[0], edge[-1], edge[1])
                                          for edge in self.sorted_edges])
        nx_graph.add_nodes_from(list(range(1, self.num_vertices)))

    def get_node(self, n):
        for node in self.node_list:
            if n == node[1]:
                return node


def find(iterable, key):
    for indx, element in enumerate(iterable):
        if key in element:
            return indx
```

## Graphing and visualization part using the Core.py file

```
from core import Graph, Kruskal, Prim
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import argparse


parser = argparse.ArgumentParser(
    """\n >>> MSTrees by Mohamed Kasem
    Perform Minimum spanning trees on a graph.""")
parser.add_argument("-f", "--file_name",
                    help="specify the input csv file containing the data for the
graph")
args = parser.parse_args()

plt.style.use('fivethirtyeight')
fig1, (ax1, ax2, ax3) = plt.subplots(1, 3)
fig1.suptitle('MSTrees Animated')


graph = Graph(args.file_name)

kruksal = Kruskal(graph)
kruksal.compute()

prim = Prim(sample_csv= args.file_name, graph = graph)
prim.compute()

G = nx.OrderedGraph()
G_animation_kruksal = nx.OrderedGraph()
G_animation_prim = nx.OrderedGraph()


prim.add_all_graph(G)
kruksal.add_graph_nodes(G_animation_kruksal)
pos = nx.spring_layout(G)
```

```
# Drawing main static graph
nx.draw_networkx_nodes(G, pos, node_size=700, ax=ax1, node_color='#f06059')
nx.draw_networkx_edges(G, pos, edgelist=G.edges(), width=6, ax=ax1)
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels, ax=ax1, font_size=15)
nx.draw_networkx_labels(G, pos, font_size=20, font_family="sans-serif", ax=ax1)
ax1.set_axis_off()

# Animation function that re-draws a snapshot of the algorithms periodically


def animate(i):
    if kruksal.add_graph_edge(G_animation_kruksal):
        ax2.clear()
    if prim.add_graph_node(G_animation_prim):
        ax3.clear()

    plt.cla()

    # Draw Kruksal's tree
    nx.draw_networkx_nodes(G_animation_kruksal, pos, node_size=700, ax=ax2)
    nx.draw_networkx_edges(
        G_animation_kruksal, pos, edgelist=G_animation_kruksal.edges(), width=6,
ax=ax2)
    labels = nx.get_edge_attributes(G_animation_kruksal, 'weight')
    nx.draw_networkx_edge_labels(
        G_animation_kruksal, pos, edge_labels=labels, ax=ax2, font_size=15)
    nx.draw_networkx_labels(G_animation_kruksal, pos, font_size=20,
                            font_family="sans-serif", ax=ax2)
    # Draw Prim's tree
    nx.draw_networkx_nodes(
        G_animation_prim, pos, nodelist=G_animation_prim.nodes, node_size=700,
ax=ax3)
    nx.draw_networkx_edges(G_animation_prim, pos,
                           edgelist=G_animation_prim.edges(), width=6, ax=ax3)
    labels = nx.get_edge_attributes(G_animation_prim, 'weight')
    nx.draw_networkx_edge_labels(
        G_animation_prim, pos, edge_labels=labels, ax=ax3, font_size=15)
    nx.draw_networkx_labels(G_animation_prim, pos,
                            font_size=20, font_family="sans-serif", ax=ax3)

    ax2.set_axis_off()
    ax3.set_axis_off()
    plt.tight_layout()


ani = FuncAnimation(plt.gcf(), animate, frames=100, interval=500)


plt.tight_layout()
# uncomment to save a gif of the figures
# ani.save("demo.gif",writer='imagemagick', fps=20)
plt.show()
```