Performance comparison of Kruskal's and Prim's MST algos

Authors: Pavlo Kryven and Danylo Hotskivsky

Task and experiment description

This experiment's purpose is to demonstrate work of Prim's and Kruskal's algorithms, show one of their implementations and measure their performance depending on number of nodes and edges in graph.

The experiment measures mean time of 1000 finding MST of graphs in different configurations.

For each algo was performed 1000 tries for each combination of (10, 20, 50, 100, 200, 500) nodes and (0.1, 0.3, 0.5, 0.7, 0.9) probabilites of certain edge being in graph.

Then performance of every graph configuration (num of nodes and probability) is compared between our observed algorithms. Based on this information the conclusion about performance and scenarios of usage are made.

Computer Specifiction

- processor: AMD Ryzen 7 5700U

- number of cores: 8

- frequency: 1.8 - 4.3 GHz

- memory: 15.08 GiB

- os: Arch Linux

- kernel: Linux 5.15.7-arch1-1

Kruskal's Algorithm

In [ ]:
```
def kruskal(graph):
    edges = graph[1]
    nodes = graph[0]
    components = [set(node) for node in list(map(lambda x: [x],nodes))]
    tree = []
    edges.sort(key=lambda x: x[2])
    component1 = 0
    component2 = 1
    for edge in edges:
        component1 = component2 = set()
        for component in components:
            if edge[0] in component:
                component1 = component
            elif edge[1] in component:
                component2 = component
        if component1 != component2 and component1!=set() and component2!=set(
            component1.update(component2)
            components.remove(component2)
            tree.append(edge)
        if len(tree) == len(nodes) - 1:
            break
    return tree
```

Prim's Algorithm

In [1]:
```
from heapq import heappop, heappush


def prim_heaps(graph):
    mst = []
    used_verts = set()
    num_verts = len(graph[0])
    edges = [[] for _ in range(num_verts)]
    for edge in graph[1]:
        if edge[0] == edge[1]: continue
        heappush(edges[edge[0]], (edge[2], edge[1]))
        heappush(edges[edge[1]], (edge[2], edge[0]))

    cost, dest = 0, 1
    while len(used_verts) < num_verts:
        smallest_edge_vert = 0
        for vert in used_verts:
            while(len(edges[vert])) > 0 and edges[vert][0][dest] in used_verts
                heappop(edges[vert])

            if len(edges[vert]) == 0: continue

            if len(edges[smallest_edge_vert]) == 0 or edges[vert][0][cost] < ed
                smallest_edge_vert = vert
        edge = heappop(edges[smallest_edge_vert])
        mst.append((smallest_edge_vert, edge[dest], edge[cost]))
        used_verts.add(smallest_edge_vert)
        used_verts.add(edge[dest])

    return mst
```

Primitive implementation of prim's algorithm

In [ ]:
```python
def prim(graph):
    edges = graph[1]
    nodes = graph[0]
    tree = []
    connected = {edges[0][0]}
    for i in range(len(nodes) - 1):
        min_edges = []
        for edge in edges:
            if (edge[0] in connected) and (edge[1] not in connected):
                min_edges.append(edge)
            elif (edge[1] in connected) and (edge[0] not in connected):
                min_edges.append(edge)
        min_edge = sorted(min_edges, key = lambda x: x[2])[0]
        tree.append(min_edge)
        connected.add(min_edge[2])
    return tree
```

Experiments environment

In [6]:
```python
import random
import networkx as nx
import matplotlib.pyplot as plt
import time

from itertools import combinations, groupby
from tqdm import tqdm

def gnp_random_connected_graph(num_of_nodes: int,
                               completeness: int,
                               draw: bool = False) -> list[tuple[int, int]]:
    """
    Generates a random undirected graph, similarly to an Erdős-Rényi
    graph, but enforcing that the resulting graph is conneted
    """

    edges = combinations(range(num_of_nodes), 2)
    G = nx.Graph()
    G.add_nodes_from(range(num_of_nodes))

    for _, node_edges in groupby(edges, key=lambda x: x[0]):
        node_edges = list(node_edges)
        random_edge = random.choice(node_edges)
        G.add_edge(*random_edge)
        for e in node_edges:
            if random.random() < completeness:
                G.add_edge(*e)

    for (u, v, w) in G.edges(data=True):
        w['weight'] = random.randint(0, 10)

    if draw:
        plt.figure(figsize=(10, 6))
        nx.draw(G, node_color='lightblue',
        with_labels=True,
                node_size=500)

    return G
```

In [ ]:
```
NUM_OF_ITERATIONS = 1000

def performance_test(num_vert, probability, method):
    time_taken = 0
    for i in tqdm(range(NUM_OF_ITERATIONS)):

        # note that we should not measure time of graph creation
        G = gnp_random_connected_graph(num_vert, probability, False)
        edges = list(map(lambda x: (x[0], x[1], x[2]['weight']), G.edges.data()
        nodes = list(G.nodes)
        my_graph = (nodes, edges)
        if method == 'kruskal':
            f = kruskal
        elif method == 'prim':
            f = prim
        elif method == 'prim_heaps':
            f = prim_heaps
        start = time.time()
        f(my_graph)
        end = time.time()

        time_taken += end - start

    time_taken / NUM_OF_ITERATIONS
    return time_taken
```

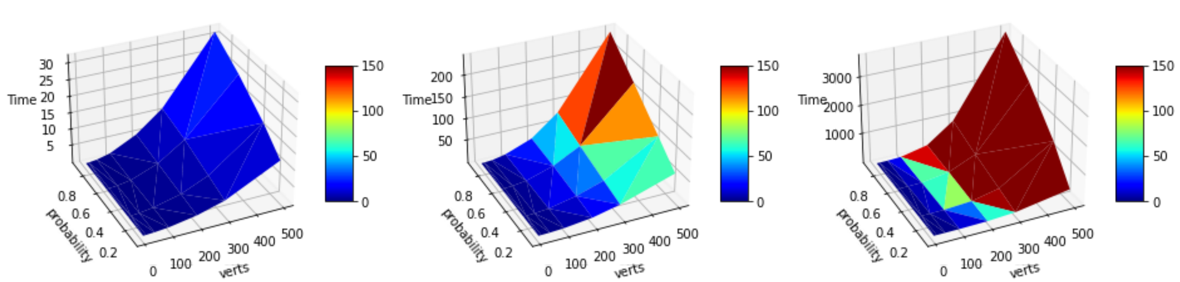
In [ ]:
```
verts = [10, 20, 50, 100, 200, 300, 500]
probs = [0.1, 0.3, 0.6, 0.9]
methods = ['kruskal', 'prim', 'prim_heaps']

output = []

for method in methods:
    for prob in probs:
        for vrt in verts:
            t = performance_test(vrt, prob, method)
            output.append(f'{vrt},{prob},{method},{t}\n')

with open('results.csv', 'w') as file:
    file.write('Number of verts,Probability,Method,Time taken\n')
    file.writelines(output)
```

General plot for performnace depending on graphs' size

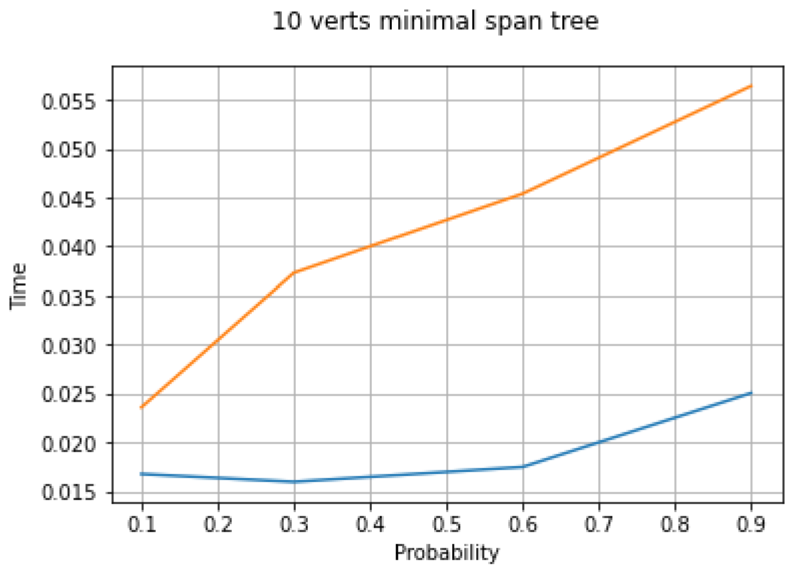


In the left we see Kruskal's algorithm. In the center - Prim's algorithm using heaps. In the right - primitive prim's Algorithm.

Performance of algorithms for graphs with 10 nodes

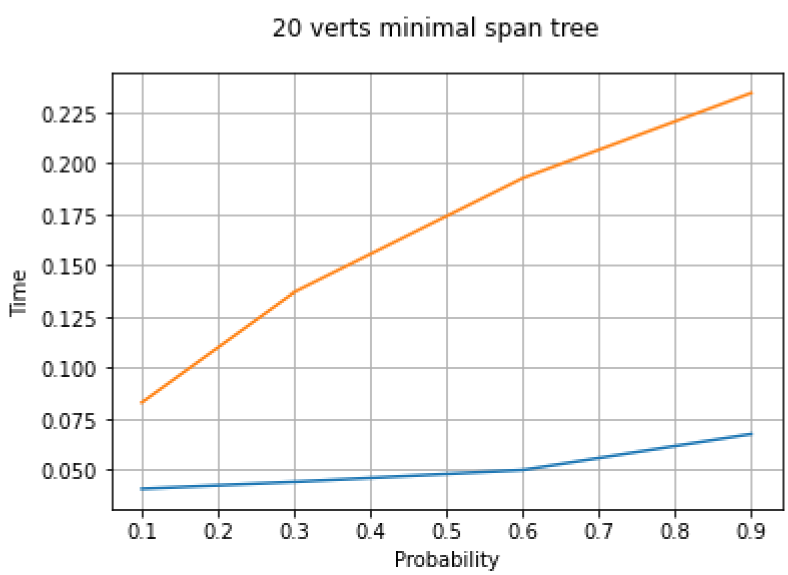
On each plot orange line is the performance of Kruskal's algo, and the blue one is Prim's algo

---plot for 2 algos where x - probability and y is time



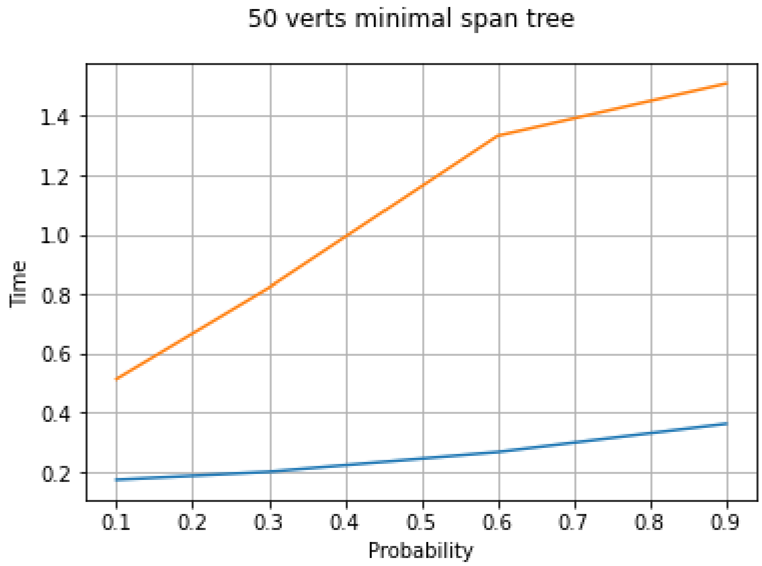
Performance of algorithms for graphs with 20 nodes

---plot for 2 algos where x - probability and y is time



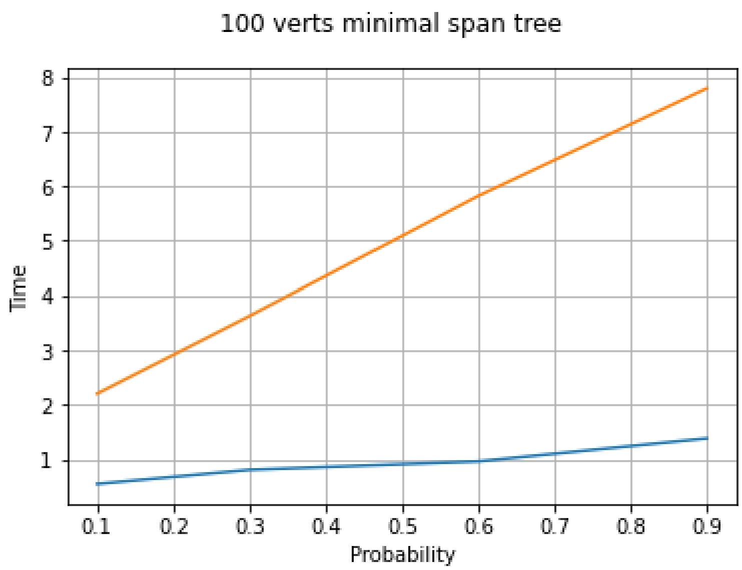
Performance of algorithms for graphs with 50 nodes

---plot for 2 algos where x - probability and y is time



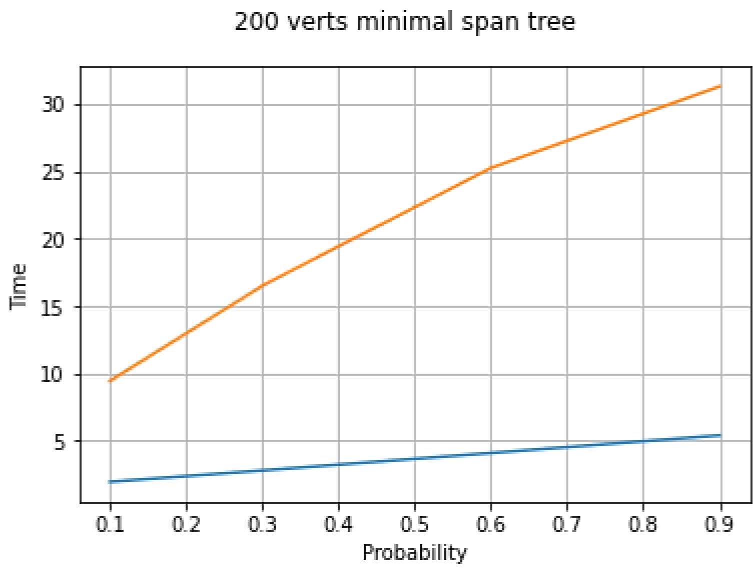
Performance of algorithms for graphs with 100 nodes

---plot for 2 algos where x - probability and y is time



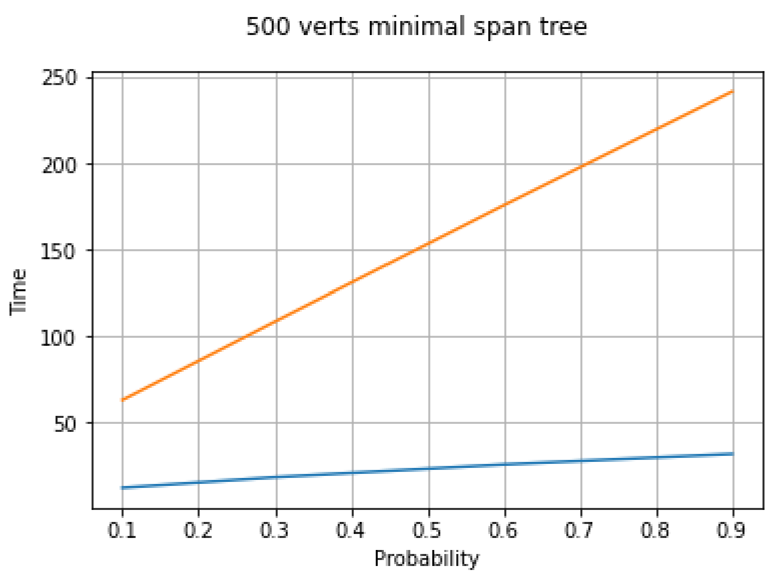
Performance of algorithms for graphs with 200 nodes

---plot for 2 algos where x - probability and y is time



Performance of algorithms for graph with 500 nodes

---plot for 2 algos where x - probability and y is time



Result

In general, it can be seen that each algorithm coped with the task in a reasonable amount of time, this is because the ammount of vertices was not huge, also Kruskal's algorithm did its job way faster, this is due to appropriate types of data storage and manipulation available in python, algorithm's realisation utilizes language's features. The same on the other hand can not be told about Prim's algorithm. In order for it to work properly we had to use structure called binary heaps. Without them time complexity of this algorithm scales abnormally and firstly it took us about 2 hours to run all tests for Prim's algorithm, with the heap implementation all tests took about 20 minutes.