MST-algorithms

February 14, 2024

[2]:
```
!pip install networkx
!pip install matplotlib
!pip install tqdm
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: networkx in
/home/zhukowych/.local/lib/python3.10/site-packages (3.2.1)

[notice] A new release of pip is
available: 23.3.2 -> 24.0
[notice] To update, run:
python3 -m pip install --upgrade pip
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: matplotlib in
/home/zhukowych/.local/lib/python3.10/site-packages (3.8.2)
Requirement already satisfied: contourpy>=1.0.1 in
/home/zhukowych/.local/lib/python3.10/site-packages (from matplotlib) (1.2.0)
Requirement already satisfied: cycler>=0.10 in
/home/zhukowych/.local/lib/python3.10/site-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/home/zhukowych/.local/lib/python3.10/site-packages (from matplotlib) (4.47.2)
Requirement already satisfied: kiwisolver>=1.3.1 in
/home/zhukowych/.local/lib/python3.10/site-packages (from matplotlib) (1.4.5)
Requirement already satisfied: numpy<2,>=1.21 in
/home/zhukowych/.local/lib/python3.10/site-packages (from matplotlib) (1.26.3)
Requirement already satisfied: packaging>=20.0 in
/home/zhukowych/.local/lib/python3.10/site-packages (from matplotlib) (23.2)
Requirement already satisfied: pillow>=8 in /usr/lib/python3/dist-packages (from
matplotlib) (9.0.1)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/lib/python3/dist-
packages (from matplotlib) (2.4.7)
Requirement already satisfied: python-dateutil>=2.7 in
/home/zhukowych/.local/lib/python3.10/site-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in
/home/zhukowych/.local/lib/python3.10/site-packages (from python-
dateutil>=2.7->matplotlib) (1.16.0)

[notice] A new release of pip is
```

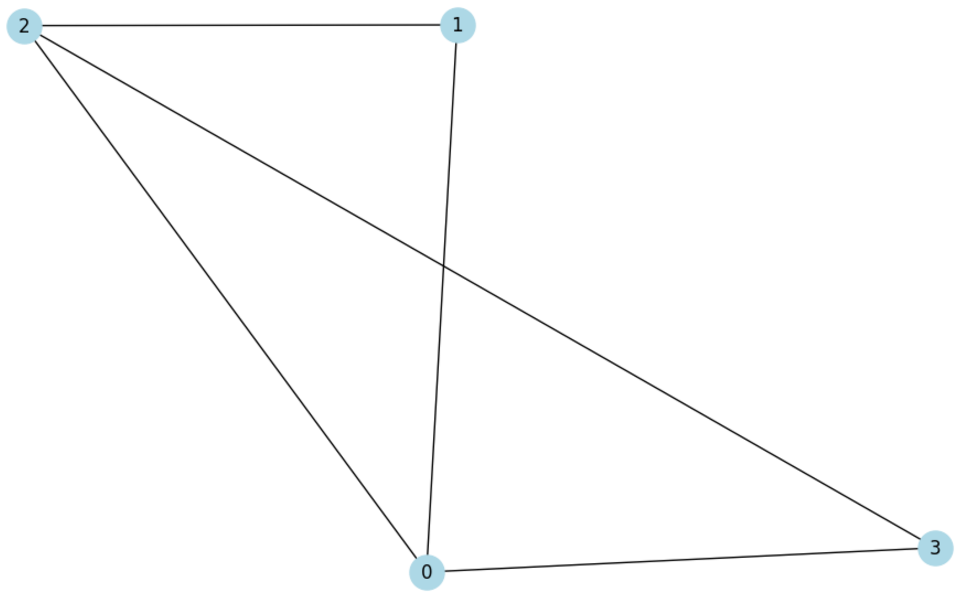
```
available: 23.3.2 -> 24.0
[notice] To update, run:
python3 -m pip install --upgrade pip
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: tqdm in
/home/zhukowych/.local/lib/python3.10/site-packages (4.66.1)

[notice] A new release of pip is
available: 23.3.2 -> 24.0
[notice] To update, run:
python3 -m pip install --upgrade pip
```

[3]:
```python
import random
import networkx as nx
import matplotlib.pyplot as plt
from itertools import combinations, groupby
```

0.1 Generating graph

[4]:
```python
# You can use this function to generate a random graph with 'num_of_nodes' nodes
# and 'completeness' probability of an edge between any two nodes
# If 'directed' is True, the graph will be directed
# If 'draw' is True, the graph will be drawn
def gnp_random_connected_graph(num_of_nodes: int,
                               completeness: int,
                               directed: bool = False,
                               draw: bool = False):
    """
    Generates a random graph, similarly to an Erdős-Rényi
    graph, but enforcing that the resulting graph is conneted (in case of␣
 ↪undirected graphs)
    """


    if directed:
        G = nx.DiGraph()
    else:
        G = nx.Graph()
    edges = combinations(range(num_of_nodes), 2)
    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)
        if random.random() < 0.5:
            random_edge = random_edge[::-1]
        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(-5, 20)

    if draw:
        plt.figure(figsize=(10,6))
        if directed:
            # draw with edge weights
            pos = nx.arf_layout(G)
            nx.draw(G,pos, node_color='lightblue',
                    with_labels=True,
                    node_size=500,
                    arrowsize=20,
                    arrows=True)
            labels = nx.get_edge_attributes(G,'weight')
            nx.draw_networkx_edge_labels(G, pos,edge_labels=labels)

        else:
            nx.draw(G, node_color='lightblue',
                with_labels=True,
                node_size=500)

    return G
```

[5]:
```
G = gnp_random_connected_graph(4, 0.5, False, True)
```



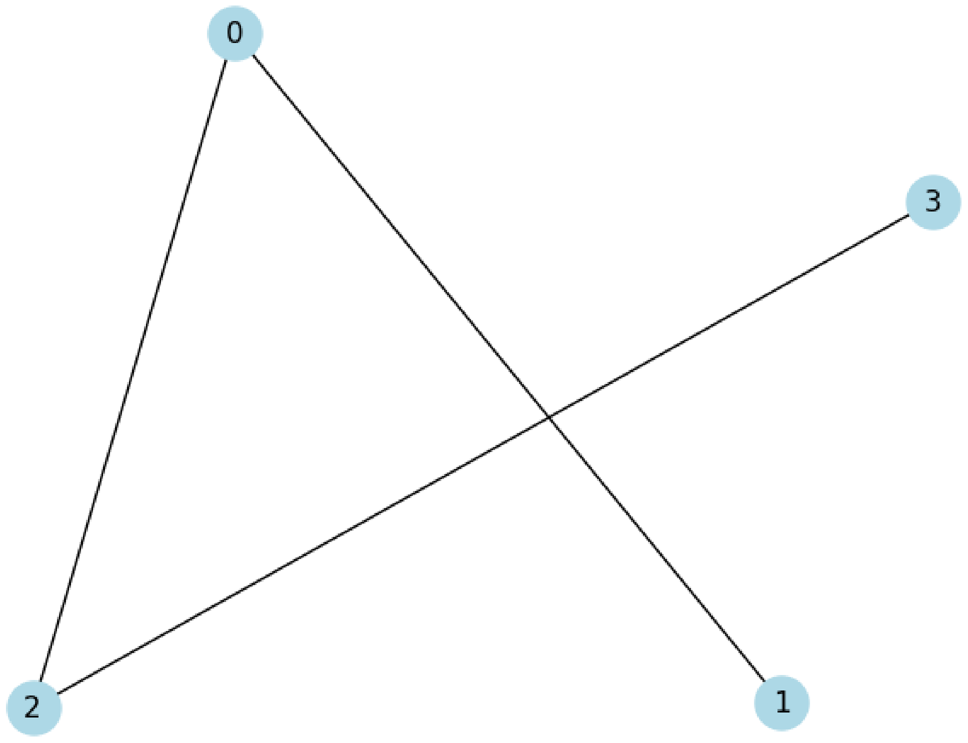
1 Minimum spanning trees algorithms

1.1 Kruskal's algorithm

[6]:
```python
from networkx.algorithms import tree
```

[7]:
```python
mstk = tree.minimum_spanning_tree(G, algorithm="kruskal")
```

[8]:
```python
nx.draw(mstk, node_color='lightblue',
        with_labels=True,
        node_size=500)
```



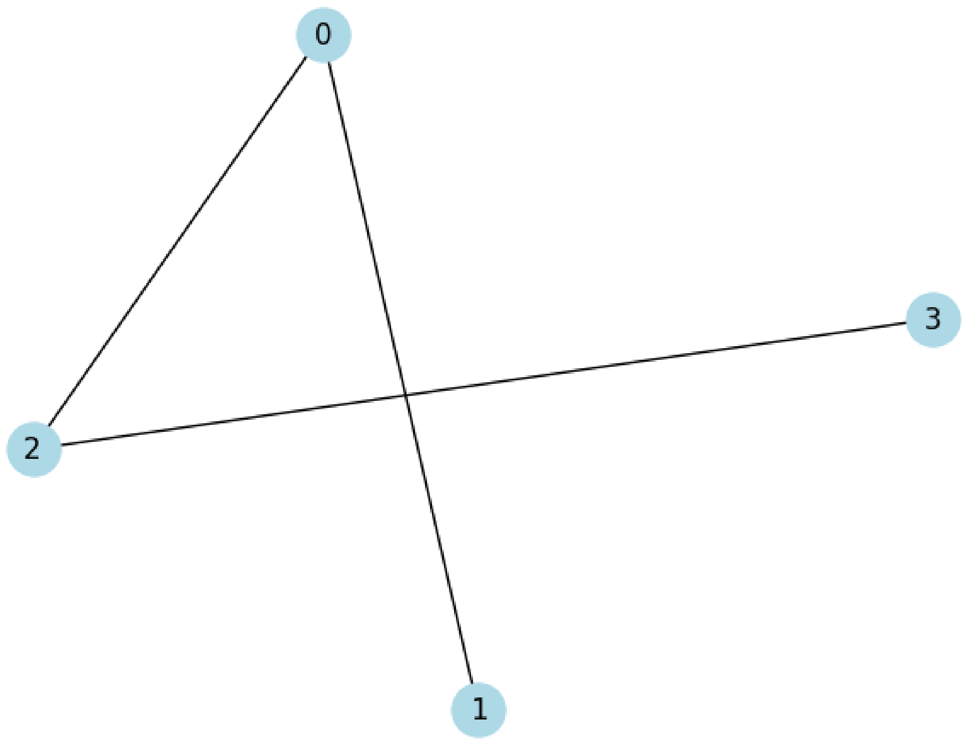
```
[9]: mstk.edges(), len(mstk.edges())
```

```
[9]: (EdgeView([(0, 1), (0, 2), (2, 3)]), 3)
```

```
[10]: def kruskal(G):
          """
          Realisation of Kruskal's algorithm to find minimal spanning
          tree of graph G
          """
          edges = [ ((edge[0], edge[1]), edge[2]['weight']) for edge in G.
       ↪edges(data=True)]
          edges.sort(key=lambda x: x[1])

          mst_edges = []
          mst_nodes = set()
          partitioning = { node: {node} for node in G.nodes }

          for edge in edges:
              if len(mst_edges) == len(G) - 1:
                  break
```

```
        (u, v), weight = edge
        u_set = partitioning[u]
        v_set = partitioning[v]
        if v_set != u_set:
            mst_edges.append((u, v, weight))
            mst_nodes.add(u)
            mst_nodes.add(v)
            u_set |= v_set
            for w in u_set:
                partitioning[w] = u_set


    mst = nx.Graph()
    mst.add_weighted_edges_from(mst_edges)

    return mst
```

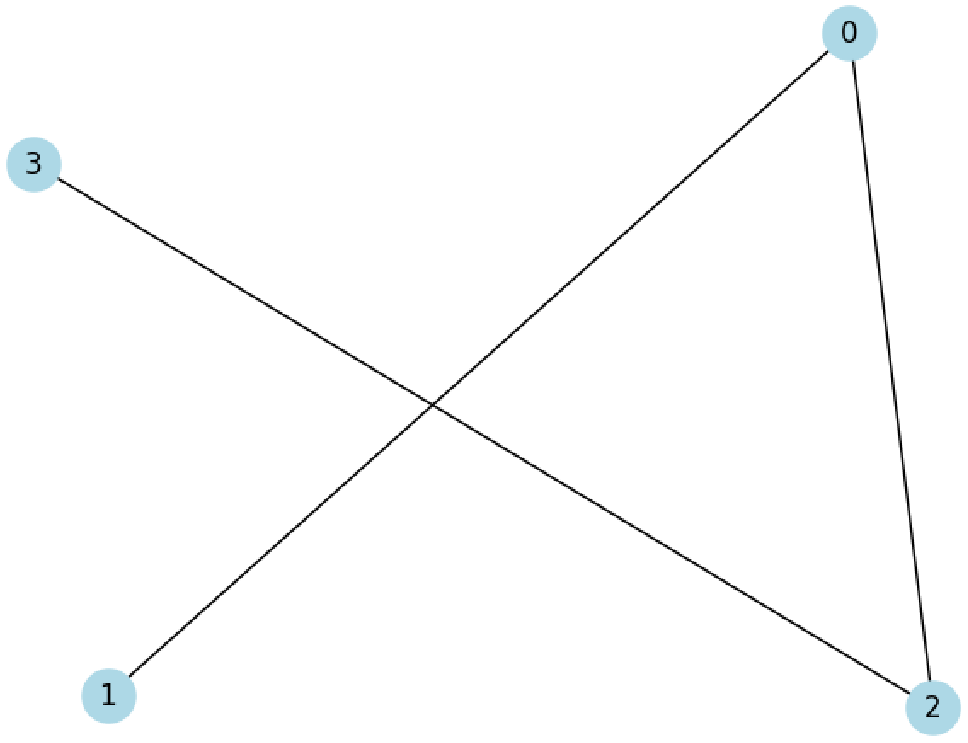
[11]:
```
mstk_r = kruskal(G)
nx.draw(mstk_r, node_color='lightblue',
        with_labels=True,
        node_size=500)
```



1.2 Prim's algorithm

[12]:
```
mstp = tree.minimum_spanning_tree(G, algorithm="prim")
```

[13]:
```
nx.draw(mstp, node_color='lightblue',
        with_labels=True,
        node_size=500)
```

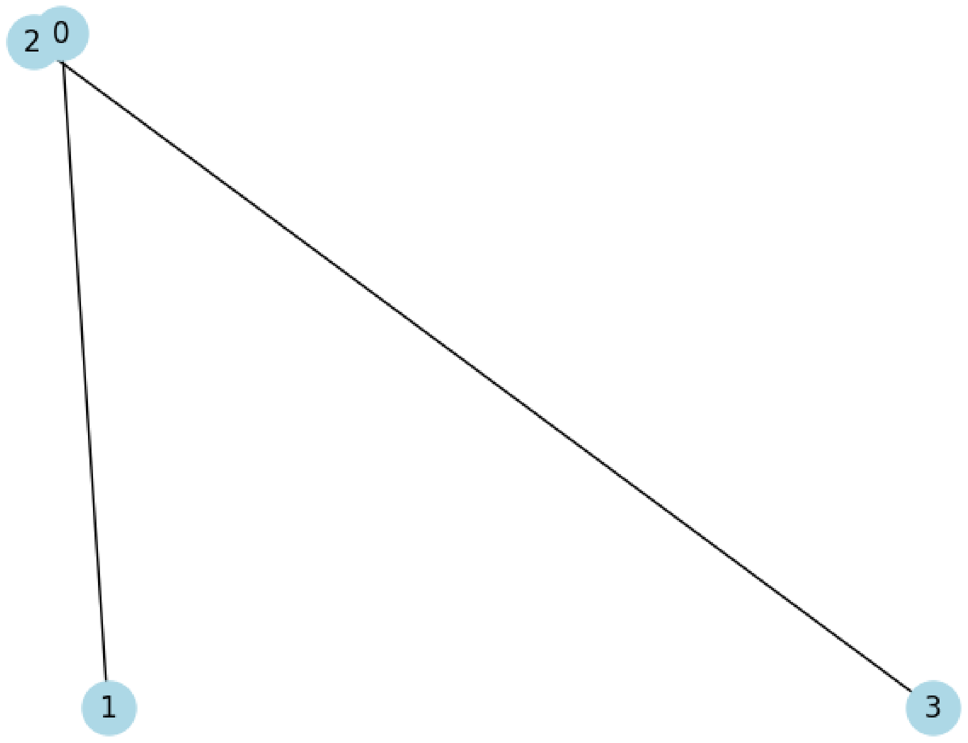


[14]:
```
mstp.edges(), len(mstp.edges())
mstp.size(weight="weight")
```

[14]:
```
-5.0
```

[15]:
```
from heapq import heappop, heappush, heapify

def prim_mst(G):
    """
    Find minimum spannint tree using Prim's algorithm.
    """
    fronteer = list( (w['weight'], (u, v))  for u, v, w in G.edges(0,␣
↪data=True))
    heapify(fronteer)
    nodes = {0}
    edges = set()

    edge_history = set()

    while fronteer:

        if len(nodes) == len(G):
            break

        w, (u, v) = heappop(fronteer)

        edge_history.add((u, v))

        if not v in nodes:
            nodes.add(v)
            edges.add((u, v, w))

        for v, w, weight in G.edges(v, data=True):
            if w not in nodes and (v, w) not in edge_history:
                heappush(fronteer, ((weight['weight'], (v, w))))

    mst = nx.Graph()
    mst.add_weighted_edges_from(edges)
    return mst
```

[16]:
```
mstp_r = prim_mst(G)
nx.draw(mstp_r, node_color='lightblue',
        with_labels=True,
        node_size=500)
```



```
[17]: mstp_r.edges(), len(mstp_r.edges())
      mstp_r.size(weight="weight")
```

```
[17]: -5.0
```

1.3 Testing

1.3.1 1. Testing correctness

Firstly, we must check if our algorithms work properly. To do so we must check: 1. our algorithms return a tree that contain all edges of the original graph 2. if this is tree but not forest 3. if weight of tree produced by implemented algorithm is less or equal to networkx implementation

Also we test on different graph sizes (form 2 to 1024) and vary completeness (from 0.2 to 1)

```
[18]: def test(networkx_algorithm_name: str, my_realization: callable):

          for n in range(1, 11):
              n = 2 ** n
              print(">>> Graph size:", n)
              for j in range(1, 5):
                  G = gnp_random_connected_graph(n, 1 / j, False, False)
```

```
            mst = nx.minimum_spanning_tree(G, algorithm=networkx_algorithm_name)
            mst_r = my_realization(G)
            assert nx.is_tree(mst_r)
            assert set(G.nodes) == set(mst.nodes)
            assert nx.number_connected_components(G) == 1
            assert mst.size(weight="weight") >= mst_r.size(weight="weight")
```

[19]:
```
print("Test Kruskal implementation")
test("kruskal", kruskal)
print("Test prim implementation")
test("prim", prim_mst)
```

```
Test Kruskal implementation
>>> Graph size: 2
>>> Graph size: 4
>>> Graph size: 8
>>> Graph size: 16
>>> Graph size: 32
>>> Graph size: 64
>>> Graph size: 128
>>> Graph size: 256
>>> Graph size: 512
>>> Graph size: 1024
Test prim implementation
>>> Graph size: 2
>>> Graph size: 4
>>> Graph size: 8
>>> Graph size: 16
>>> Graph size: 32
>>> Graph size: 64
>>> Graph size: 128
>>> Graph size: 256
>>> Graph size: 512
>>> Graph size: 1024
```

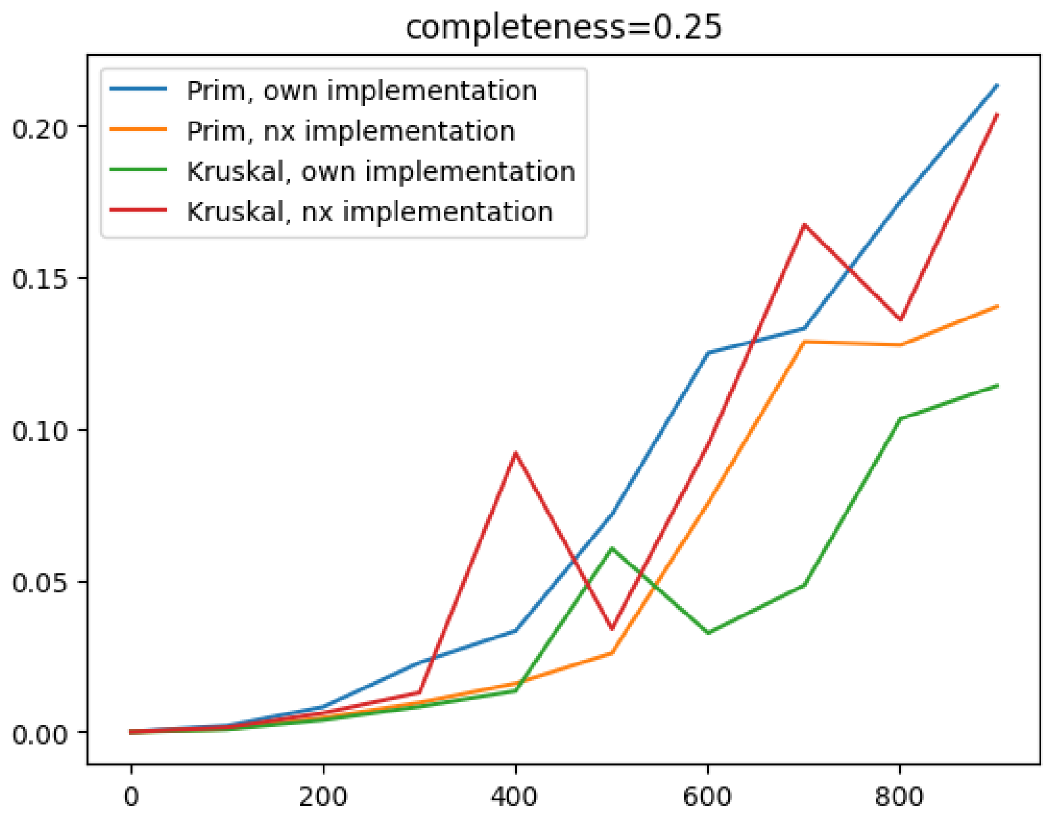
1.3.2 2. Testing performance

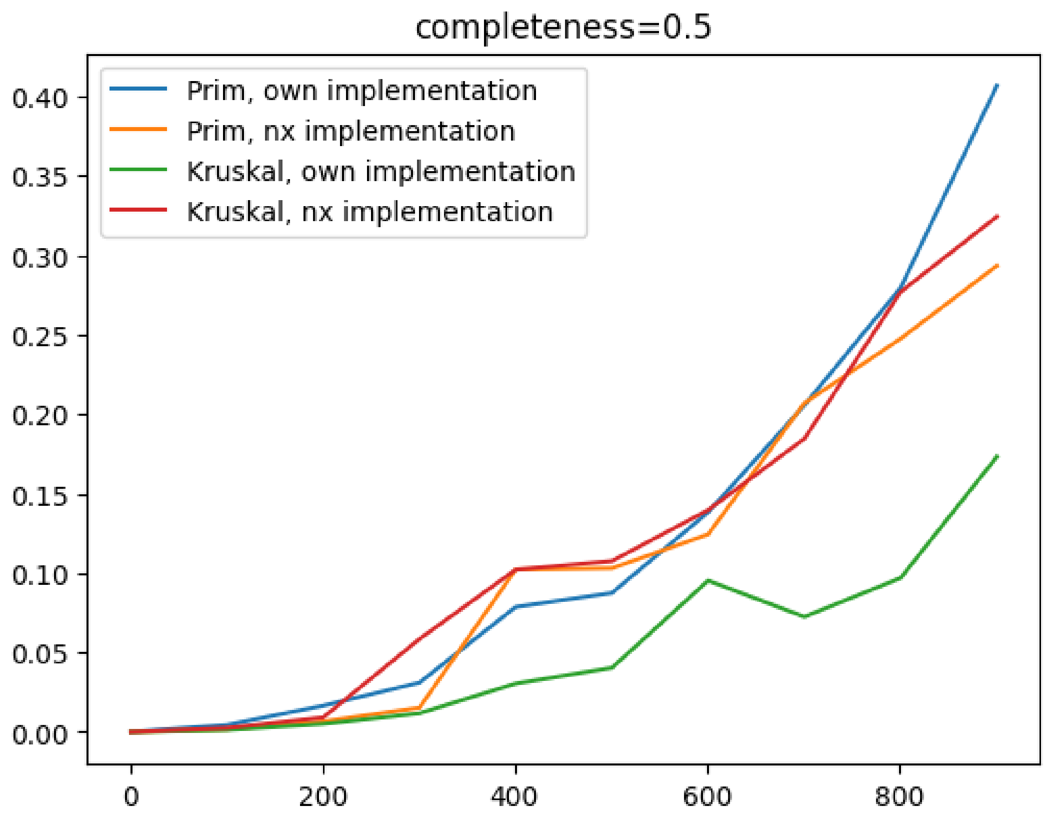
Now, we must check performace of impelemned algorithms and compare results with nx algorithms performance

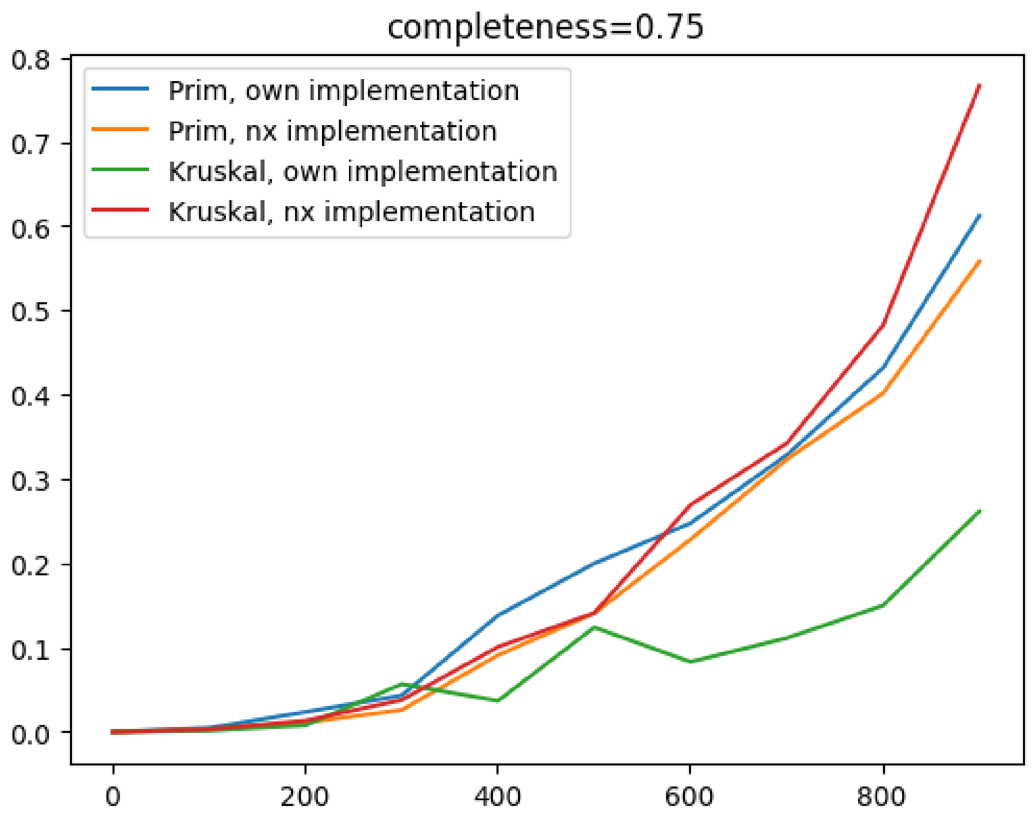
[20]:
```
import time
def plot_time_size(algorithms: dict[str, callable],
                   max_nodes: int,
                   completeness: float,
                   step: str):

    algorithms_performance = { algorithm: [] for algorithm in algorithms}
    x = list(range(1, max_nodes+1, step))

    for i in x:
        g = gnp_random_connected_graph(i, completeness, False, False)

        for algorithm, algorithm_function in algorithms.items():
            start = time.perf_counter()
            algorithm_function(g)
            end = time.perf_counter()
            algorithms_performance[algorithm].append(end-start)

    for algorithm, performance in algorithms_performance.items():
        plt.plot(x, performance, label=algorithm)

    plt.title(f"completeness={completeness}")
    plt.legend()
    plt.show()
```

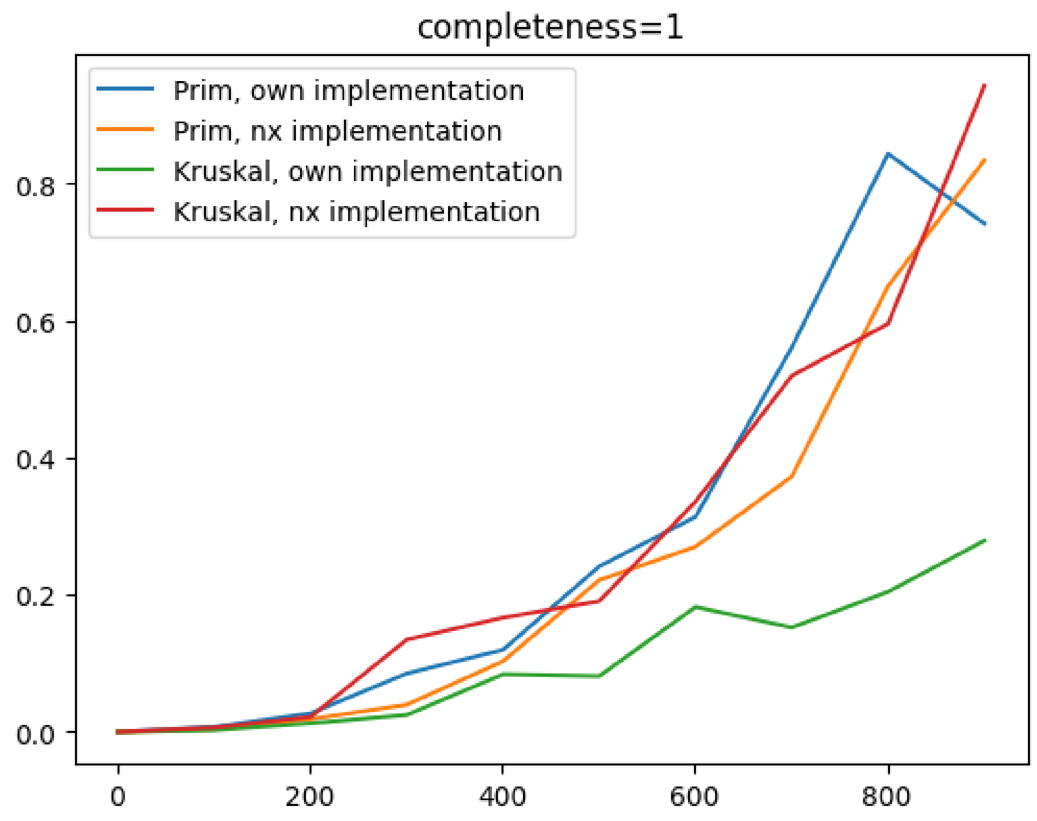
Here we test all algorithms (Prim and Kruskal) both nx and own. We varying sizes of graphs from 100 to 1000 with step 100 and completeness from 0.25 to 1

[21]:
```
algorithms = {
    'Prim, own implementation': prim_mst,
    "Prim, nx implementation": lambda g: tree.minimum_spanning_tree(g,␣
 ↪algorithm="prim"),
    "Kruskal, own implementation": kruskal,
    "Kruskal, nx implementation": lambda g: tree.minimum_spanning_tree(g,␣
 ↪algorithm="kruskal")
}

# 500 vertices
plot_time_size(algorithms, 1000, 0.25, 100)
plot_time_size(algorithms, 1000, 0.5, 100)
plot_time_size(algorithms, 1000, 0.75, 100)
plot_time_size(algorithms, 1000, 1, 100)
```


completeness=0.25
Prim, own implementation
Prim, nx implementation
Kruskal, own implementation
Kruskal, nx implementation
0.20
0.15
0.10
0.05
0.00
0
200
400
600
800


completeness=0.5
Prim, own implementation
Prim, nx implementation
Kruskal, own implementation
Kruskal, nx implementation
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
200
400
600
800


completeness=0.75
Prim, own implementation
Prim, nx implementation
Kruskal, own implementation
Kruskal, nx implementation
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
200
400
600
800



2 Conclusions

As can bee seen, we implemented Kruskal and Prim algorithms that return correct graphs (see **Tesing correctness** stage).

Prim algorithm analysis

Our realization of Prim algorithm work as fast as nx realization for all tested combinations, so can be used in both cases.

Kruskal algorithm analysis Our implementation of kruskall algorithm works pretty faster than nx one, because we use dictonaries instead of disjoint sets to handle vertices partition sets. This implementation can be used on bigger graphs (works 4x times faster). Although it may use a lot more RAM

Overall analysis As can be seen from graphs both Kruskal and Prim algorithms work as good as nx implementation. Kruskal is 4x faster, but uses more RAM. Also our algorithms return nx.Graph so that it can be used to apply other nx algorithms