prima

February 14, 2024

Sofiia Popeniuk, Lushpak Victoriia

Problem Statement and Experiment Setup: To evaluate the performance of Prima's algorithm, we have implemented our own version and compared its execution time with a built-in algorithm. The objective is to assess the efficiency and effectiveness of our implementation in finding minimum spanning trees compared to established algorithms.

[219]:
```
!pip install networkx
!pip install matplotlib
!pip install tqdm
```

```
Requirement already satisfied: networkx in
./algorithms_lab/venv/lib/python3.12/site-packages (3.2.1)

[notice] A new release of pip is
available: 23.2.1 -> 24.0
[notice] To update, run:
pip install --upgrade pip
Requirement already satisfied: matplotlib in
./algorithms_lab/venv/lib/python3.12/site-packages (3.8.2)
Requirement already satisfied: contourpy>=1.0.1 in
./algorithms_lab/venv/lib/python3.12/site-packages (from matplotlib) (1.2.0)
Requirement already satisfied: cycler>=0.10 in
./algorithms_lab/venv/lib/python3.12/site-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
./algorithms_lab/venv/lib/python3.12/site-packages (from matplotlib) (4.48.1)
Requirement already satisfied: kiwisolver>=1.3.1 in
./algorithms_lab/venv/lib/python3.12/site-packages (from matplotlib) (1.4.5)
Requirement already satisfied: numpy<2,>=1.21 in
./algorithms_lab/venv/lib/python3.12/site-packages (from matplotlib) (1.26.4)
Requirement already satisfied: packaging>=20.0 in
./algorithms_lab/venv/lib/python3.12/site-packages (from matplotlib) (23.2)
Requirement already satisfied: pillow>=8 in
./algorithms_lab/venv/lib/python3.12/site-packages (from matplotlib) (10.2.0)
Requirement already satisfied: pyparsing>=2.3.1 in
./algorithms_lab/venv/lib/python3.12/site-packages (from matplotlib) (3.1.1)
Requirement already satisfied: python-dateutil>=2.7 in
./algorithms_lab/venv/lib/python3.12/site-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in
```

```
./algorithms_lab/venv/lib/python3.12/site-packages (from python-
dateutil>=2.7->matplotlib) (1.16.0)

[notice] A new release of pip is
available: 23.2.1 -> 24.0
[notice] To update, run:
pip install --upgrade pip
Requirement already satisfied: tqdm in
./algorithms_lab/venv/lib/python3.12/site-packages (4.66.2)

[notice] A new release of pip is
available: 23.2.1 -> 24.0
[notice] To update, run:
pip install --upgrade pip
```

[8]:
```
import random
import networkx as nx
import matplotlib.pyplot as plt
from itertools import combinations, groupby
from copy import deepcopy
```

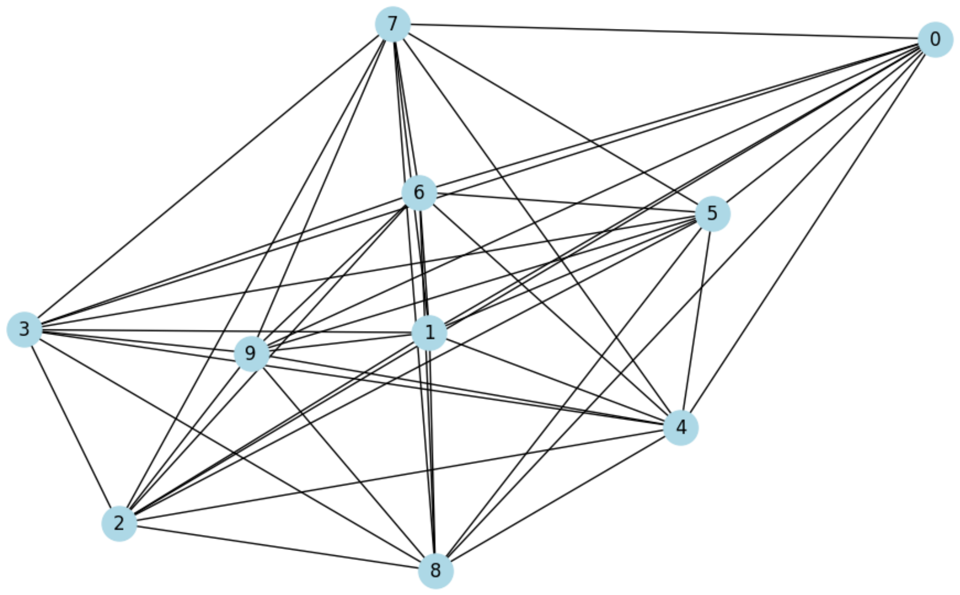
0.1 Generating graph

[6]:
```
# 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
```

The function weight_build_in_algorithm calculates the weight of the obtained framework built using the built-in algorithm. It iterates through each edge and adds its weight to the output.

[13]:
```
def weight_build_in_algorithm(mstp: object) -> int:
    """
    Weight of build in algorithm.
    """
    output_weight = 0
    conections = dict(mstp.adjacency())
    edges = mstp.edges()
    for edge in edges:
        for node_num, conection in conections.items():
            if node_num == edge[0]:
                for connected_node, weight in conection.items():
                    if connected_node == edge[1]:
                        output_weight += weight['weight']
                        break
    return output_weight
```

[9]:
```
G = gnp_random_connected_graph(10, 1, False, True)
```

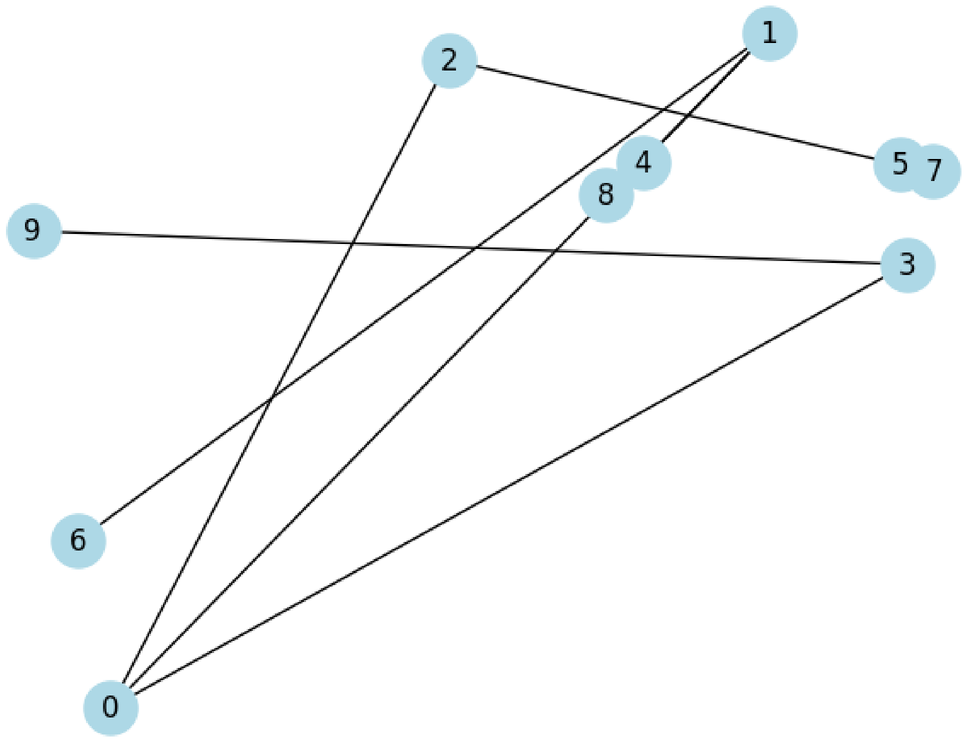


[3]:
```
from networkx.algorithms import tree
```

0.2 Prim's algorithm

[10]:
```
mstp = tree.minimum_spanning_tree(G, algorithm="prim")
```

[209]:
```
nx.draw(mstp, node_color='lightblue',
        with_labels=True,
        node_size=500)
```



[11]:
```
mstp.edges(), len(mstp.edges())
```

[11]:
```
(EdgeView([(0, 2), (0, 9), (0, 3), (1, 5), (2, 5), (3, 4), (4, 7), (6, 9), (7,
8)]),
 9)
```

The functions has_cycle and has_cycle_dfs, which search for cycles using Depth-First Search (DFS), operate by visiting each node of the graph and checking if it has already been visited during the current traversal. During DFS traversal, nodes of the graph are visited in depth until a terminal node is reached or a cycle is found.

This code defines a function called main that implements Prim's algorithm for finding the minimum spanning tree (MST) of a graph. It initializes a heap data structure with edge weights, then iteratively selects the edge with the smallest weight, checks if adding it forms a cycle in the current MST, and if not, adds it to the MST. The function continues this process until the MST contains one less edge than the total number of nodes in the graph, and finally returns the total weight of the MST.

[211]:
```
import heapq
def has_cycle_dfs(edges: list, current: int, visited: int, parent: int) -> bool:
    """
    Depth-first search.
    """
    visited.add(current)
    for edge in edges:
        node, neighbor = edge
        if node == current:
            if neighbor not in visited:
                if has_cycle_dfs(edges, neighbor, visited, current):
                    return True
            elif neighbor != parent:
                return True
        elif neighbor == current:
            if node not in visited:
                if has_cycle_dfs(edges, node, visited, current):
                    return True
            elif node != parent:
                return True
    return False

def has_cycle(edges: list) -> bool:
    """
    Checks if the graph contains any cycles.
    >>> has_cycle([(1, 2), (2, 3), (1, 3)])
    True
    """
    visited = set()
    for edge in edges:
        node, neighbor = edge
        if node not in visited:
            if has_cycle_dfs(edges, node, visited, None):
                return True
        if neighbor not in visited:
            if has_cycle_dfs(edges, neighbor, visited, None):
                return True
        visited.add(node)
        visited.add(neighbor)
    return False

def main(G):
    """
    >>> main(G) == weight_build_in_algorithm(mstp)
    True
    """
    connections = dict(G.adjacency())
    heap = [(data['weight'], node, neighbor) for node, edges in connections.
 ↪items() for neighbor, data in edges.items()]
    heapq.heapify(heap)
    visited = set([0])
    prima = []
    output_weight = 0
    while len(prima) != len(connections) - 1:
        weight, node, neighbor = heapq.heappop(heap)
        if (has_cycle(prima + [(node, neighbor)]) is False):
            visited.add(node)
            visited.add(neighbor)
            prima.append((node, neighbor))
            output_weight += weight
    return output_weight
if __name__ == '__main__':
    import doctest
    print(doctest.testmod())
```

```
TestResults(failed=0, attempted=2)
```

0.3 Example on time measuring

Read more on this: https://realpython.com/python-timer/

Recall that you should measure times for 5, 10, 20, 50, 100, 200, 500 nodes 1000 times (and take mean of time taken for each node amount).

Then you should build the plot for two algorithms (x - data size, y - mean time of execution).

[214]:
```
import time
from tqdm import tqdm
```

[215]:
```
NUM_OF_ITERATIONS = 1000
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(100, 0.4, False)

    start = time.time()
    tree.minimum_spanning_tree(G, algorithm="prim")
    end = time.time()

    time_taken += end - start

time_taken / NUM_OF_ITERATIONS
```

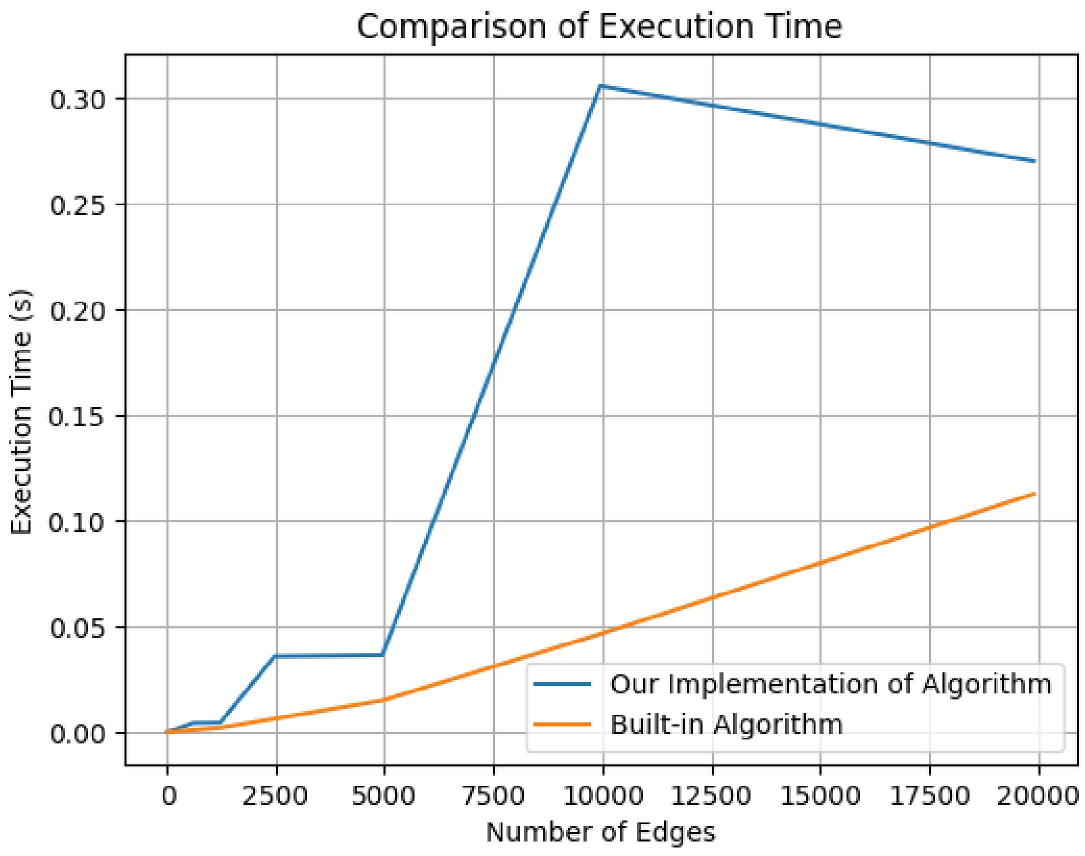
```
100%|      | 1000/1000 [00:03<00:00, 328.74it/s]
```

[215]:
```
0.0008434839248657227
```

We ran the algorithm for graphs with 10, 50, 100 and 200 vertices and densities of 0.5 and 1. For each parameter, we executed the algorithm 1000 times and calculated the average value for both our algorithm and the built-in one. We plotted the graph based on the time taken against the number of edges. We chose to analyze the number of edges because for the same number of vertices, we executed the algorithm twice with different densities, making it the most representative graph in our opinion.

[228]:
```
import time
import matplotlib.pyplot as plt
from tqdm import tqdm

NUM_OF_ITERATIONS = 1000
graph_sizes = [10, 50, 100, 200]
fullness_levels = [0.5, 1]

main_execution_times = []
built_in_algorithm_execution_times = []
num_edges = []

for size in graph_sizes:
    for fullness in fullness_levels:
        my_time = 0
        built_in_time = 0
        for i in tqdm(range(NUM_OF_ITERATIONS)):
            G = gnp_random_connected_graph(size, fullness, False, False)

            start_main_execution = time.time()
            main_execution_time = main(G)
            end_main_execution = time.time()
            my_time += end_main_execution - start_main_execution

            start_built_in_execution = time.time()
            build_in_algorithm_execution_time = weight_build_in_algorithm(G)
            end_built_in_execution = time.time()
            built_in_time += end_built_in_execution - start_built_in_execution

        main_execution_times.append(my_time / 1000)
        built_in_algorithm_execution_times.append(built_in_time / 1000)
        num_edges.append(size * (size - 1) / 2 * fullness)

# Plot the results
plt.plot(num_edges, main_execution_times, label='Our Implementation of␣
 ↪Algorithm')
plt.plot(num_edges, built_in_algorithm_execution_times, label='Built-in␣
 ↪Algorithm')
plt.xlabel('Number of Edges')
plt.ylabel('Execution Time (s)')
plt.title('Comparison of Execution Time')
plt.legend()
plt.grid(True)
plt.show()
```

```
100%|      | 1000/1000 [00:00<00:00, 9283.69it/s]
100%|      | 1000/1000 [00:00<00:00, 8023.75it/s]
100%|      | 1000/1000 [00:05<00:00, 172.63it/s]
100%|      | 1000/1000 [00:07<00:00, 136.27it/s]
100%|      | 1000/1000 [00:44<00:00, 22.42it/s]
100%|      | 1000/1000 [00:55<00:00, 18.01it/s]
100%|      | 1000/1000 [06:03<00:00,  2.75it/s]
100%|      | 1000/1000 [06:41<00:00,  2.49it/s]
```



On small graphs (25 and 45 edges): our algorithm operates efficiently, demonstrating similar performance to the built-in Prim's algorithm.

On medium-sized graphs (615 and 1225 edges): A slight time difference was observed, but it is quite small (0.01 seconds). This may be due to certain peculiarities in the implementation of our algorithm.

On large graphs (2475, 4950, 9950, and 19900 edges): An increase in execution time of our algorithm compared to the built-in one has been noted. This may indicate that our algorithm may be less optimized for large graphs or may require optimization to improve performance.

So, overall, it can be noted that our algorithm works effectively on small and medium-sized graphs, where it demonstrates similar performance to the built-in Prim's algorithm. However, when it comes to large graphs, an increase in execution time of our algorithm compared to the built-in one is observed. This may indicate the necessity for additional optimization of our algorithm for optimal operation on large datasets, where performance is of significant importance.

[15]:
```python
mstk = tree.minimum_spanning_tree(G, algorithm="kruskal")
def min_weight(conections: dict) -> tuple:
    """
    >>> min_weight({0: {1: {'weight': -4}, 2: {'weight': 9}}, 1: {0: {'weight':␣
  ↪-4}, \
2: {'weight': 0}}, 2: {0: {'weight': 9}, 1: {'weight': 0}}})
    ((0, 1), -4)
    """
    min_start_point = 0
    min_end_point = 0
    output_weight = float("inf")
    for node_num, conection in conections.items():
        for connected_node, weight in conection.items():
            if weight['weight'] < output_weight:
                output_weight = weight['weight']
                min_start_point = node_num
                min_end_point = connected_node
    return (min_start_point, min_end_point), output_weight

def has_cycle_dfs(edges: list, current: int, visited: int, parent: int) -> bool:
    """
    Depth-first search.
    """
    visited.add(current)
    for edge in edges:
        node, neighbor = edge
        if node == current:
            if neighbor not in visited:
                if has_cycle_dfs(edges, neighbor, visited, current):
                    return True
            elif neighbor != parent:
                return True
        elif neighbor == current:
            if node not in visited:
                if has_cycle_dfs(edges, node, visited, current):
                    return True
            elif node != parent:
                return True
    return False
```

```python
def has_cycle(edges: list) -> bool:
    """
    Checks if the graph contains any cycles.
    >>> has_cycle([(1, 2), (2, 3), (1, 3)])
    True
    """
    visited = set()
    for edge in edges:
        node, neighbor = edge
        if node not in visited:
            if has_cycle_dfs(edges, node, visited, None):
                return True
        if neighbor not in visited:
            if has_cycle_dfs(edges, neighbor, visited, None):
                return True
        visited.add(node)
        visited.add(neighbor)
    return False

def kruskala(G:object) -> int:
    """
    >>> kruskala(G) == weight_build_in_algorithm(mstk)
    True
    """
    graph = deepcopy(G)
    conections = dict(graph.adjacency())
    length = len(conections) - 1
    weight = 0
    visited = set()
    kruskal = []
    while len(kruskal) < length:
        try_conection, edge_weight = min_weight(conections)
        if (try_conection[1], try_conection[0]) not in kruskal:
            if has_cycle(kruskal + [try_conection]) is False:
                weight += edge_weight
                kruskal.append(try_conection)
                visited.add(try_conection[0])
                visited.add(try_conection[1])
        del conections[try_conection[0]][try_conection[1]]
    return weight
if __name__ == '__main__':
    import doctest
    print(doctest.testmod())
```

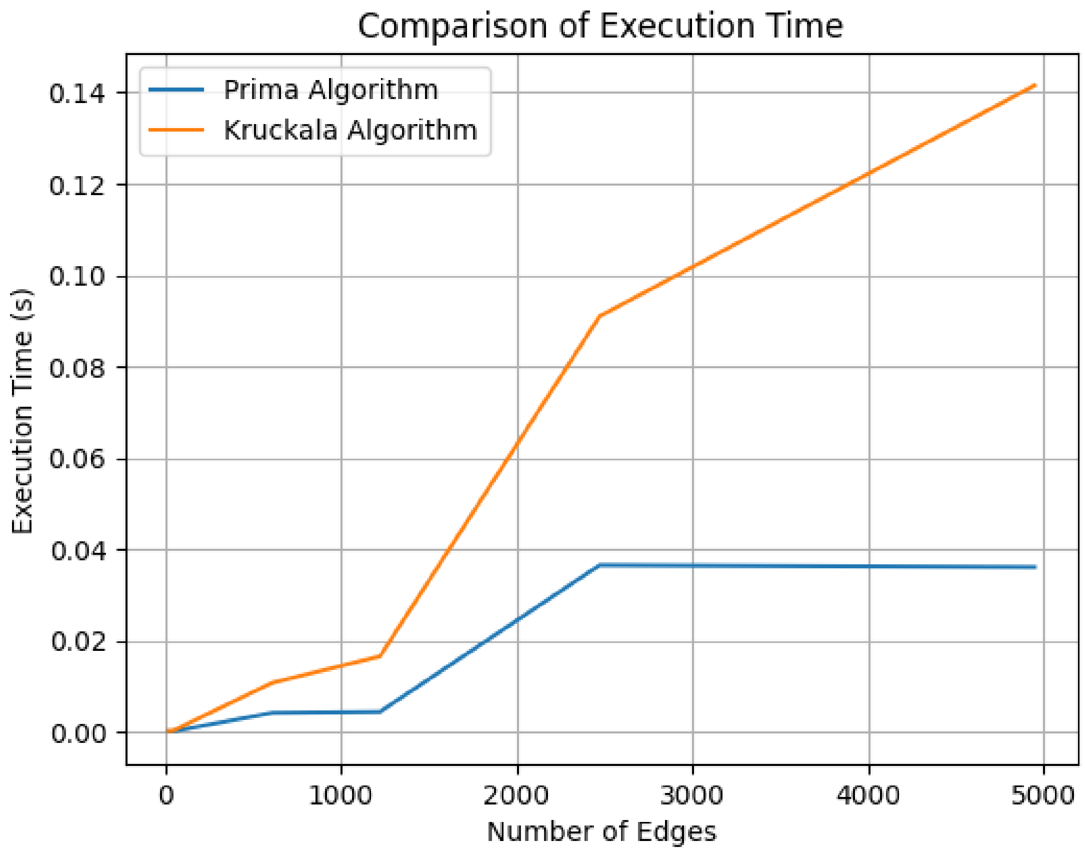
```
TestResults(failed=0, attempted=3)
```

We tested the algorithm on graphs with 10, 50, and 100 vertices, varying in densities of 0.5 and

1. For each configuration, we ran the algorithm 1000 times and computed the average execution time for both our implementation and the built-in one. Then, we generated a graph illustrating the relationship between the execution time and the number of edges. We focused on analyzing the number of edges because, for identical vertex counts, we ran the algorithm twice with distinct densities, considering it the most informative metric.

[232]:
```python
import time
import matplotlib.pyplot as plt
from tqdm import tqdm

NUM_OF_ITERATIONS = 1000
graph_sizes = [10, 50, 100]
fullness_levels = [0.5, 1]

main_execution_times = []
kruskala_execution_times = []
num_edges = []

for size in graph_sizes:
    for fullness in fullness_levels:
        my_time = 0
        kruskala_time = 0
        for i in tqdm(range(NUM_OF_ITERATIONS)):
            G = gnp_random_connected_graph(size, fullness, False, False)

            start_main_execution = time.time()
            main_execution_time = main(G)
            end_main_execution = time.time()
            my_time += end_main_execution - start_main_execution

            start_kruskala_execution = time.time()
            kruskala_execution_time = kruskala(G)
            end_kruskala_execution = time.time()
            kruskala_time += end_kruskala_execution - start_kruskala_execution

        main_execution_times.append(my_time / 1000)
        kruskala_execution_times.append(kruskala_time / 1000)
        num_edges.append(size * (size - 1) / 2 * fullness)

# Plot the results
plt.plot(num_edges, main_execution_times, label='Prima Algorithm')
plt.plot(num_edges, kruskala_execution_times, label='Kruckala Algorithm')
plt.xlabel('Number of Edges')
plt.ylabel('Execution Time (s)')
plt.title('Comparison of Execution Time')
plt.legend()
plt.grid(True)
plt.show()
```

```
100%|          | 1000/1000 [00:00<00:00, 4276.46it/s]
100%|          | 1000/1000 [00:00<00:00, 3706.83it/s]
100%|          | 1000/1000 [00:15<00:00, 64.45it/s]
100%|          | 1000/1000 [00:21<00:00, 45.73it/s]
100%|          | 1000/1000 [02:10<00:00,  7.69it/s]
100%|          | 1000/1000 [03:01<00:00,  5.50it/s]
```



Overall, our Prim's algorithm performs faster than the Kruskal's algorithm. This can be attributed to several factors. Firstly, in Prim's algorithm, we utilize the additional heapq module, which is optimized for heap operations and contributes to faster execution. Additionally, Prim's algorithm selects edges only from the incident edges of the current set, while Kruskal's algorithm considers edges from the entire graph. This difference in edge selection strategies can lead to Prim's algorithm being more efficient, especially in scenarios where the number of edges is significantly larger than the number of vertices, as it reduces the number of edges that need to be evaluated at each step.