Лабораторна робота №1

Івасюк Михайло, Тишнюк Роман

ЗВІТ

ЗАВДАННЯ 1

Алгоритм Краскала

Алгоритм виконаний у парадигмі ООП. В основі алгоритму Краскала я імплементував псевдокод, основна ідея та перевага якого, в тому, що він при ітерації по ребрам графа, шукає в яких множинах знаходяться початок та кінець даного ребра, відповідно якщо вони в різних множинах - тоді між ними не існує циклу, а отже ребро можна додавати в каркас та з'єднувати ці 2 множини.

```python
class kruskal_alghoritm:
    def __init__(self, graph) -> None:
        self.nodes = [[i] for i in graph.nodes]
        self.edges = self.kruskal(list(graph.edges(data = True)), self.nodes)
    @staticmethod
    def find(u, nodes):
        for mn in nodes:
            if u in mn:
                return mn
    def kruskal(self, edges, nodes):
        edges = sorted(edges, key= lambda x : x[2]['weight'])
        edges_tree = []
        for edge in edges:
            if len(nodes) == 1:
                return edges_tree
            vertex1 = kruskal_alghoritm.find(edge[0], nodes)
            vertex2 = kruskal_alghoritm.find(edge[1], nodes)
            if vertex1 != vertex2:
                edges_tree.append((edge[0], edge[1]))
                nodes[nodes.index(vertex2)] += (vertex1)
                nodes.pop(nodes.index(vertex1))
        return edges_tree
```

Також він не застосовує ніяких додаткових ітерацій та порівнянь для знаходження тих самих множин вершин ребер в основній множині, а просто використовує індексацію: **nodes[nodes.index(vertex2)] +=(vertex1)** та **nodes.pop(nodes.index(vertex1)).**

Ось той пошук множини:

```
@staticmethod
def find(u, nodes):
    for mn in nodes:
        if u in mn:
            return mn
```

Та відповідно саме порівняння:

```
vertex1 = kruskal_alghoritm.find(edge[0], nodes)
vertex2 = kruskal_alghoritm.find(edge[1], nodes)
if vertex1 != vertex2:
```

Порівняння з алгоритмом модуля networkx

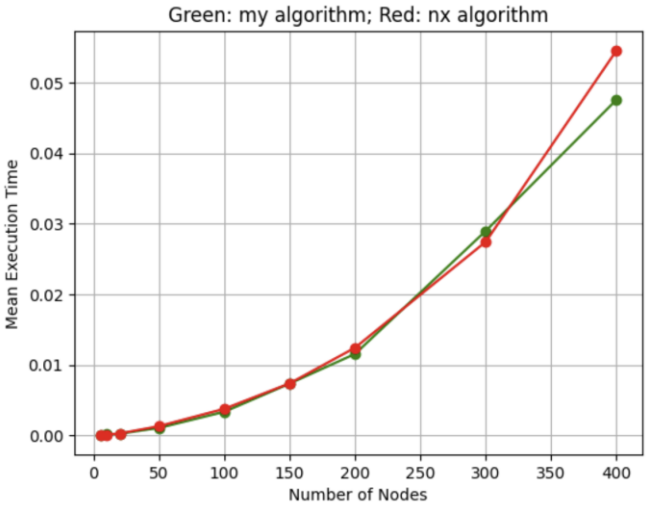
Порівняння було виконано шляхом зіставлення функції графіка створеного алгоритму та алгоритму networkx.

**при створенні функцій використовувався масив, що складався з середнього значення роботи функції для кожної кількості вершин.*

Отже, графік роботи алгоритмів:

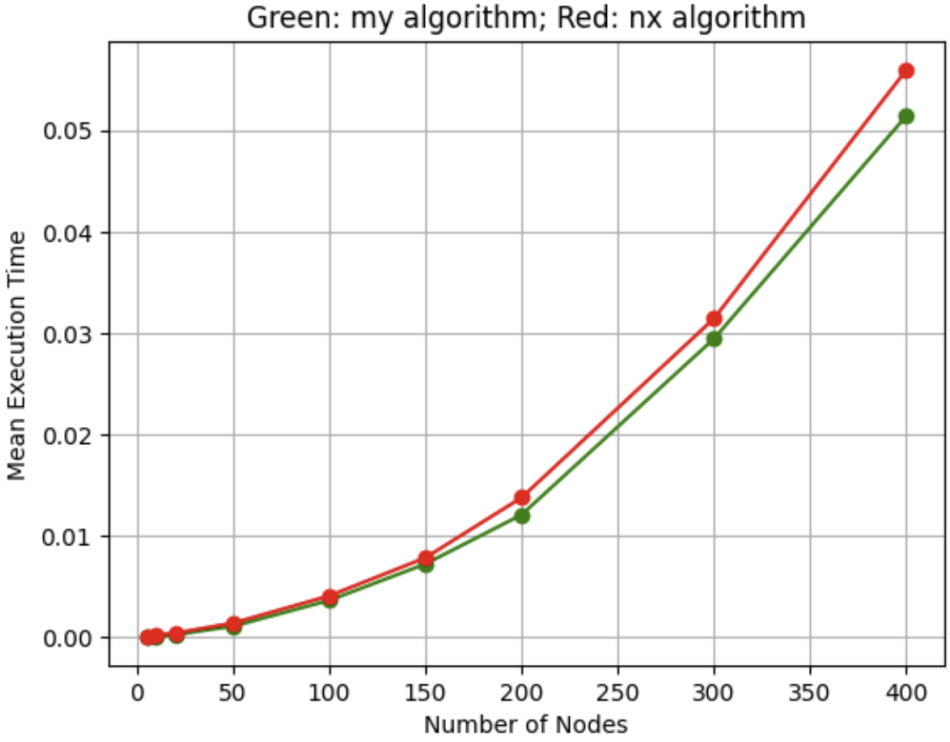
1)

- Кількість вершин: [5, 10, 20, 50, 100, 150, 200, 300, 400].
- Кількість ітерацій: 50
- Ймовірність з’єднання вершинами ребер - 0.1



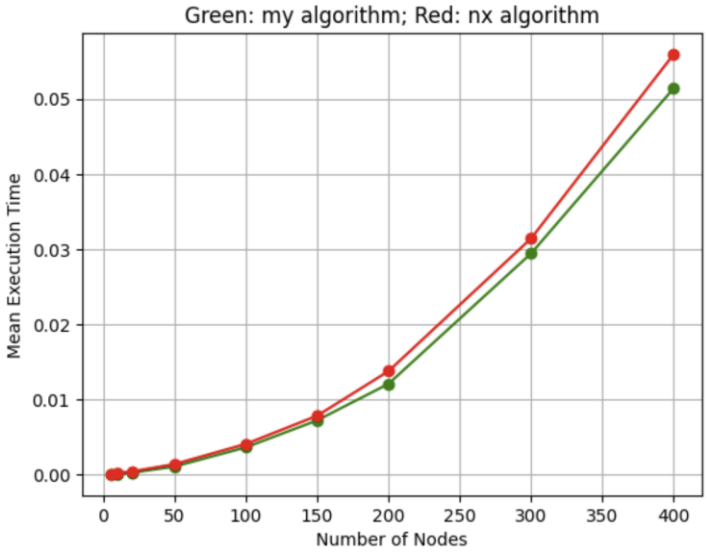
2)

- Кількість вершин: [5, 10, 20, 50, 100, 150, 200, 300, 400].
- Кількість ітерацій: 100
- Ймовірність з’єднання вершинами ребер - 0.1



3)

- Кількість вершин: [5, 10, 20, 50, 100, 150, 200, 300, 400].
- Кількість ітерацій: 10
- Ймовірність з’єднання вершинами ребер - 0.1



Висновок: як бачимо по графікам функцій, їхній вигляд практично ідентичний, що свідчить про вдалу імплементацію та відповідну складність алгоритму. Загалом в деяких моментах вдалося навіть обігнати алгоритм *networkx* по середній швидкості виконання.

Алгоритм Пріма

Реалізований в парадигмі ООП. Кожну ітерацію він підбирає ребро з найменшою вагою, яке можна долучити до списку ребер каркаса. Не завжди додає ті самі ребра, що і алгоритм від networkx, це пов’язано з тим, що деякі графи мають ребра однакової ваги, тому мінімальний каркас залишається незмінним.

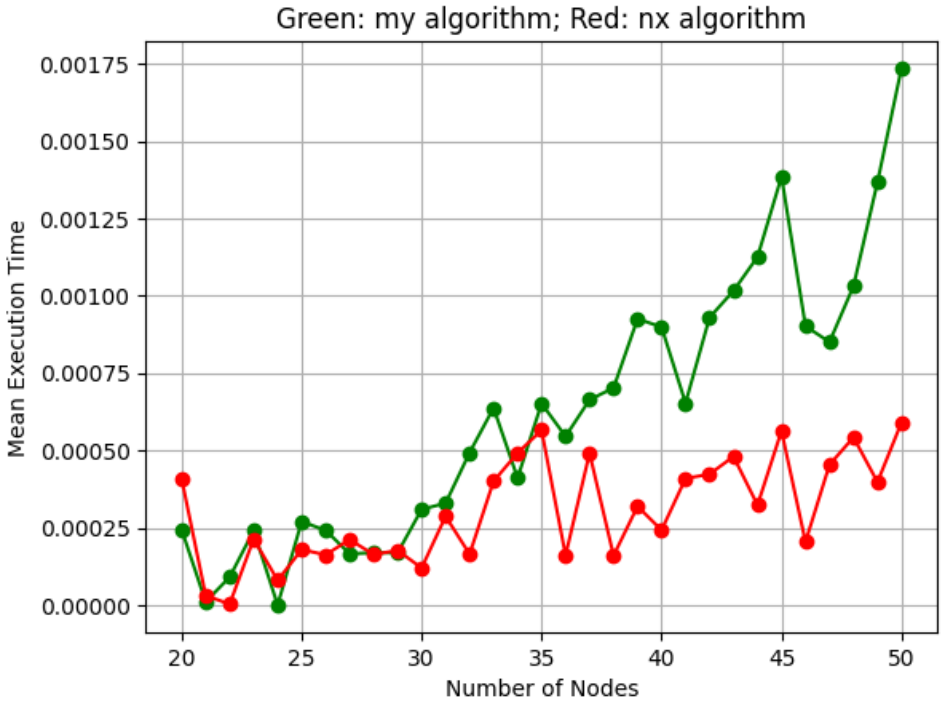
```
class PrimAlgo:
    def __init__(self, graph):
        self.edges = list(graph.edges(data=True))
    def prim(self, graph, start=0):
        visited_nodes = set()
        visited_nodes.add(start)
        res = []
        while len(visited_nodes) < len(graph.nodes):
                possible_edges = []
                for edge in self.edges:
                    if (edge[0] in visited_nodes and edge[1] not in visited_nodes) or\
                        (edge[0] not in visited_nodes and edge[1] in visited_nodes):
                        possible_edges.append(edge)
                possible_edges = sorted(possible_edges, key = lambda x: x[2]['weight'])
                res.append(possible_edges[0])
                visited_nodes.add(possible_edges[0][0])
                visited_nodes.add(possible_edges[0][1])
        return res
a = PrimAlgo(G)
```

Ось порівняння часу роботи мого алгоритму Пріма та алгоритму від networkx:

1)К-сть ітерацій:100

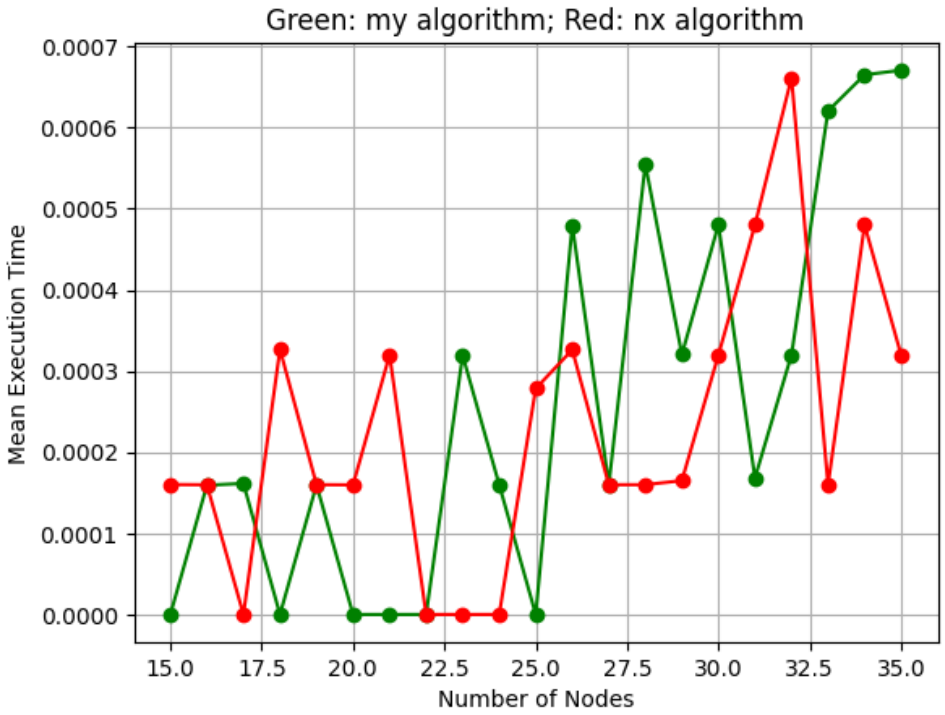
К-сть вершин 20 - 50

Імовірність з’єднання вершин: 0.1



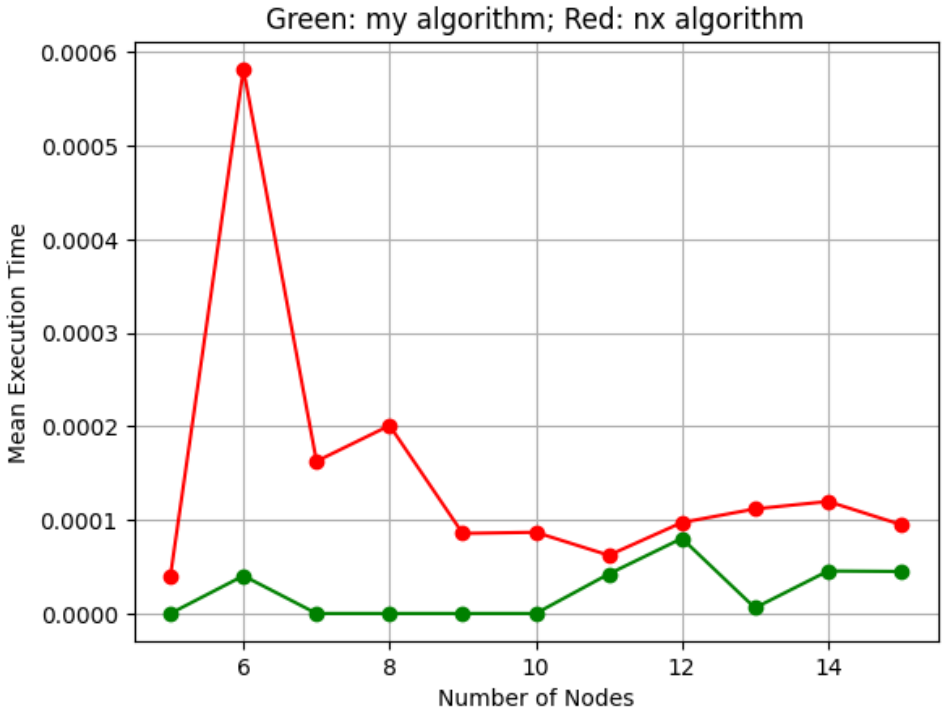
2) К-сть ітерацій - 50

К-сть вершин 15 – 35



3) К-сть ітерацій: 200

К - сть вершин: 5 – 15



Висновок: на більшій кількості вершин мій алгоритм працює дещо повільніше, але добре показує себе на меншій.

ЗАВДАННЯ 2

Алгоритм Беллмана-Форда

Алгоритм виконаний у парадигмі ООП. Алгоритм спочатку створює словник з відповідними шляхами від заданої початкової точки, до всіх інших. Кількість ітерацій становить |V| - 1. Основний цикл ітерується по ребрах вигляду (v, w, weight) та перевіряє чи: v + weight < w, відповідно тоді створюється нова *“мітка”* в словнику distance, та цикл продовжується.

```python
class Berman_Ford_alghrotim:
    def __init__(self, orientated_graph, starting_node) -> None:
        self.starting_node = starting_node
        self.edges = list(orientated_graph.edges(data = True))
        self.nodes = orientated_graph.nodes
    def shortest_path(self):
        distance = {}
        for v in self.nodes:
            distance[v] = float('inf')
        distance[self.starting_node] = 0
        for _ in range(1, len(self.nodes) - 1):
            for edge in self.edges:
                if distance[edge[0]] + edge[2]['weight'] < distance[edge[1]]:
                    distance[edge[1]] = distance[edge[0]] + edge[2]['weight']
        for edge in self.edges:
            if distance[edge[0]] + edge[2]['weight'] < distance[edge[1]]:
                return "Negative cycle detected"
        return {i : j for i, j in distance.items() if j != float('inf')}
berman_ford_result = Berman_Ford_alghrotim(G, 0)
try:
    dist = berman_ford_result.shortest_path()
    for u, w in dist.items():
        print(f"Distance to {u}:", w)
except ValueError:
    print("Negative cycle detected")
```

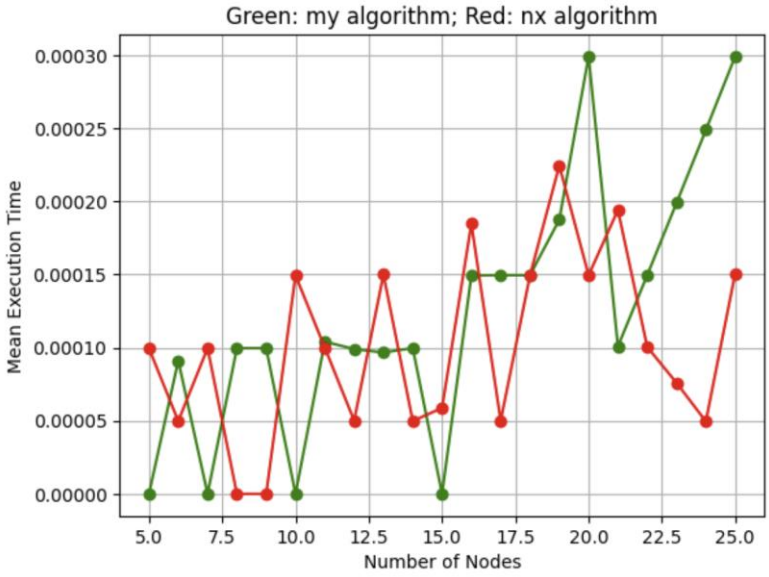
Порівняння з алгоритмом модуля networkx

Порівняння було виконано шляхом зіставлення функції графіка створеного алгоритму та алгоритму networkx.

Отже, графік роботи алгоритмів:

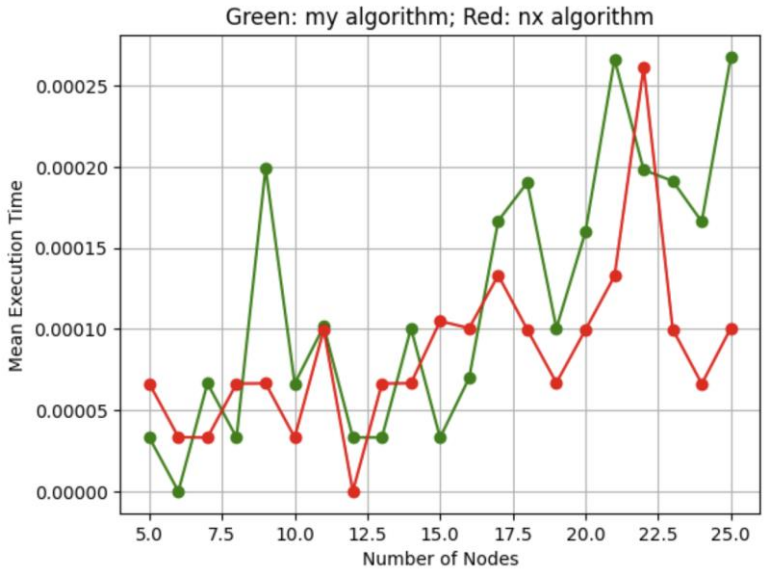
1)

- Кількість вершин: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
- Кількість ітерацій: 20
- Ймовірність з’єднання вершинами ребер - 0.01



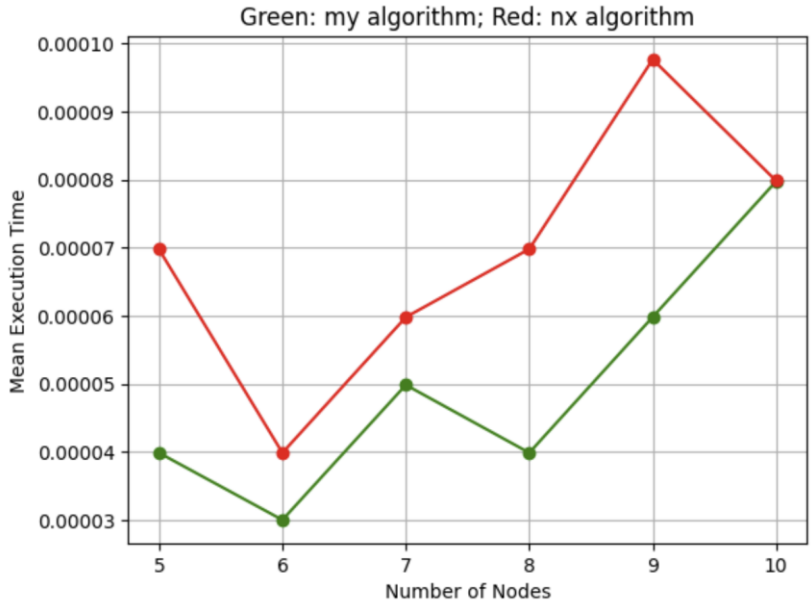
2)

- Кількість вершин: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
- Кількість ітерацій: 30



3)

- Кількість вершин: [5, 6, 7, 8, 9, 10]
- Кількість ітерацій: 100



Висновок: загалом графіки функцій по будові практично не відрізняються, але все ж алгоритм модуля *networkx* при більшій кількості вершин та ітерацій працює швидше.

Алгоритм Флойда-Воршелла

Алгоритм реалізовано в парадигмі ООП. Для зручності в реалізації граф перетворювався в матрицю, що повпливало на швидкість виконання алгоритму.

```python
class floyd_warshall:
    def __init__(self, graph):
        self.graph = graph
        self.matrix = self.graph_to_matr()
    def graph_to_matr(self):
        edges = list(self.graph.edges(data = True))
        n = len(self.graph.nodes())
        matrix = [[float('inf') for i in range(n)] for j in range(n)]
        for edge in edges:
            matrix[edge[0]][edge[1]] = edge[2]['weight']
        for i in range(n):
            matrix[i][i] = 0
        return matrix
    def floyd_warshall(self):
        length = len(self.matrix)
        res = self.matrix.copy()
        for k in range(length):
            for i in range(length):
                for j in range(length):
                    res[i][j] = min(res[i][j], res[i][k] + res[k][j])
        dict_res = {}
        for row_i, row in enumerate(res):
            dict_res[row_i] = {el_i:el for el_i, el in enumerate(row)}


        return dict_res
```

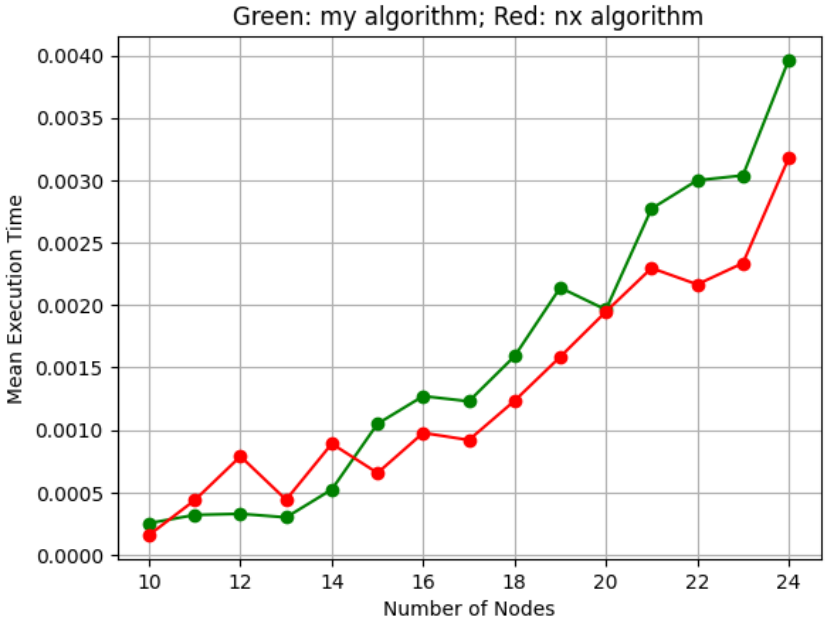
Функція graph_to_matr() перетворює граф із списку ребер в матрицю відстаней між вершинами, а далі головна функція вже змінює цю

матрицю, постійно порівнюючи існуючий шлях між вершинами і шлях через проміжну вершину. Далі швидкість було перевірено, порівнюючи мій алгоритм з уже існуючим алгоритмом від networkx. На основі даних про середній час виконання були побудовані графіки:

1) К-сть ітерацій:100

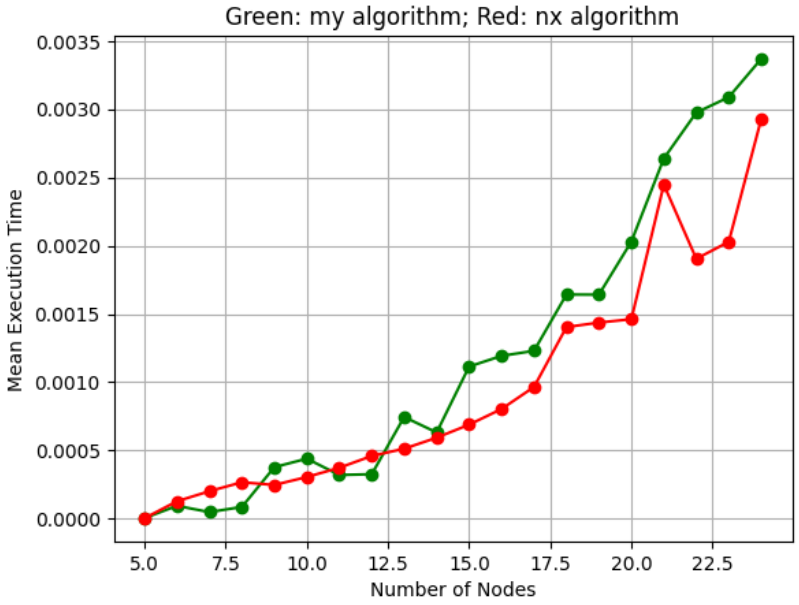
К-сть вершин:10 – 24

Імовірність з’єднання вершин: 0.01



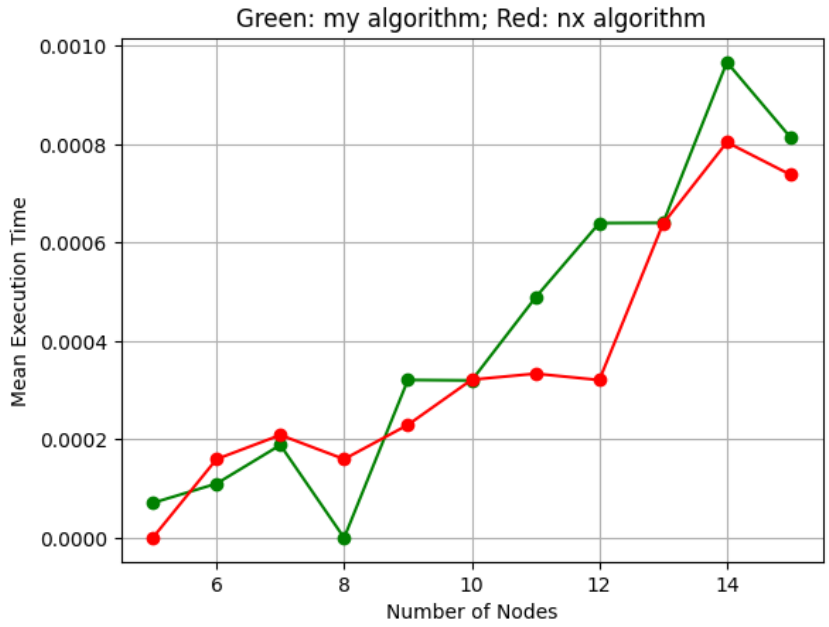
2) К-сть ітерацій:200

К-сть вершин: 5 – 24



3) К-сть ітерацій:50

К-сть вершин: 5-15



Висновок: Алгоритми працюють приблизно однаково по часу, але на більшій кількості вершин мій починає працювати повільніше(ймовірно, через більшу складність перетворення великого графа у матрицю)

Загальний підсумок

1. Алгоритм Краскала:

- Імплементація дозволяє швидко знаходити мінімальний каркас.
- Результати порівняння з алгоритмом модуля networkx свідчать про вдалу реалізацію та подібність швидкості виконання.

2. Алгоритм Пріма:

- Хороші результати, особливо на менших кількостях вершин.
- На більшій кількості вершин може працювати трохи повільніше.

3. Алгоритм Беллмана-Форда:

- Показав хороші результати, але алгоритм з модуля networkx працює трішки швидше, зокрема на більшій кількості вершин та ітерацій.

4. Алгоритм Флойда-Воршелла:

- Приблизно аналогічна швидкість виконання з алгоритмом з модуля networkx.
- На більшій кількості вершин може стати трохи повільнішим.

Імплементацію алгоритмів вважаємо вдалою.