★ Учасники:

Святослав Стегній, Савчук Оксана

★ Опис постановки задачі та експерименту:

В цій роботі нам треба було дослідити ефективність роботи алгоритмів.

Після вибору алгоритмів і розподілення їх між собою, ми приступили до написання коди, застосувавши знання з дискретної математики.

Після написання коду ми створили графіки порівняння алгоритмів Краскала і Прима, а також алгоритму Флойда-Воршала, реалізованого вручну, та реалізованого з вбудованим алгоритмом.

★ Програмний код алгоритму(-тмів):

❖ Алгоритм Прима:

```
def prim_algorithm(graph, weights):


    start_edges = {i:weights[i] for i in graph.edges() if i[0] == 0}
    first_edge = list(min(start_edges, key=start_edges.get))
    min_edges = [(first_edge[0], first_edge[1], {'weight': weights[tuple(first_edge)]})]

    weights.pop((first_edge[0], first_edge[1]))

    used = [min_edges[0][0], min_edges[0][1]]
    unused = [i for i in range(1, len(graph.nodes()))]
    unused.remove(min_edges[0][1])

    sorted_weights_by_value = sorted(weights.items(), key=lambda x: x[1])
    sorted_weights_by_value = [i[0] for i in sorted_weights_by_value]
    index = 0

    while len(min_edges) < len(graph.nodes()) - 1:
        minimum_weight = sorted_weights_by_value[index]

        if minimum_weight[0] in used and minimum_weight[1] in unused:
            to_append = (minimum_weight[0], minimum_weight[1], {'weight': weights[minimum_weight]})
            min_edges.append(to_append)
            used.append(minimum_weight[1])
            used.append(minimum_weight[0])
            unused.remove(minimum_weight[1])

            minimum_weight_index = sorted_weights_by_value.index(minimum_weight)
            sorted_weights_by_value.pop(minimum_weight_index)
            index = 0

        elif minimum_weight[1] in used and minimum_weight[0] in unused:
            to_append = (minimum_weight[0], minimum_weight[1], {'weight': weights[minimum_weight]})
            min_edges.append(to_append)
            used.append(minimum_weight[0])
            used.append(minimum_weight[1])
            unused.remove(minimum_weight[0])
```

```
            minimum_weight_index = sorted_weights_by_value.index(minimum_weight)
            sorted_weights_by_value.pop(minimum_weight_index)
            index = 0

        else:
            index += 1
            if len(sorted_weights_by_value) == 0:
                break

    return min_edges
```

❖ Алгоритм Крускала:

```
def kruskal_algorithm(graph, weights):
    sorted_edges = sorted(weights.items(), key=lambda x: x[1])
    num_of_nodes = len(graph.nodes())
    sets = [{i} for i in range(num_of_nodes)]
    min_edges = []
    i = 0
    while len(min_edges) != num_of_nodes - 1:
        element = sorted_edges[i]
        edge = set(element[0])
        is_cycle = [edge.issubset(s) for s in sets]
        if True not in is_cycle:
            min_edges.append((element[0][0], element[0][1], {'weight': element[1]}))

            sets = new_sets(sets, edge)
            sorted_edges.pop(i)
            i=0
        else:
            i+=1

    return min_edges
```

❖ Алгоритм Флойда-Воршала:

```
def floyd_warshall_algorithm(graph):

    _, dist = floyd_warshall_predecessor_and_distance(graph)

    graph = []
    for _, v in dist.items():
        graph.append(dict(v))

    sorted_graph = []
    for i in graph:
        sorted_graph.append({k: i[k] for k in sorted(i.keys())})

    matrix = [[i for i in dicti.values()] for dicti in sorted_graph]
    n = len(matrix)

    n = len(matrix)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j])
```

```
    has_negative_cycle = any(matrix[i][i] < 0 for i in range(n))
    if has_negative_cycle:
        return "Negative cost cycle exists"

    return matrix
```

★ Програмний код проведення експериментів:

- ❖ Програмний код, який створює графік для порівняння алгоритмів Прима та Крускала:

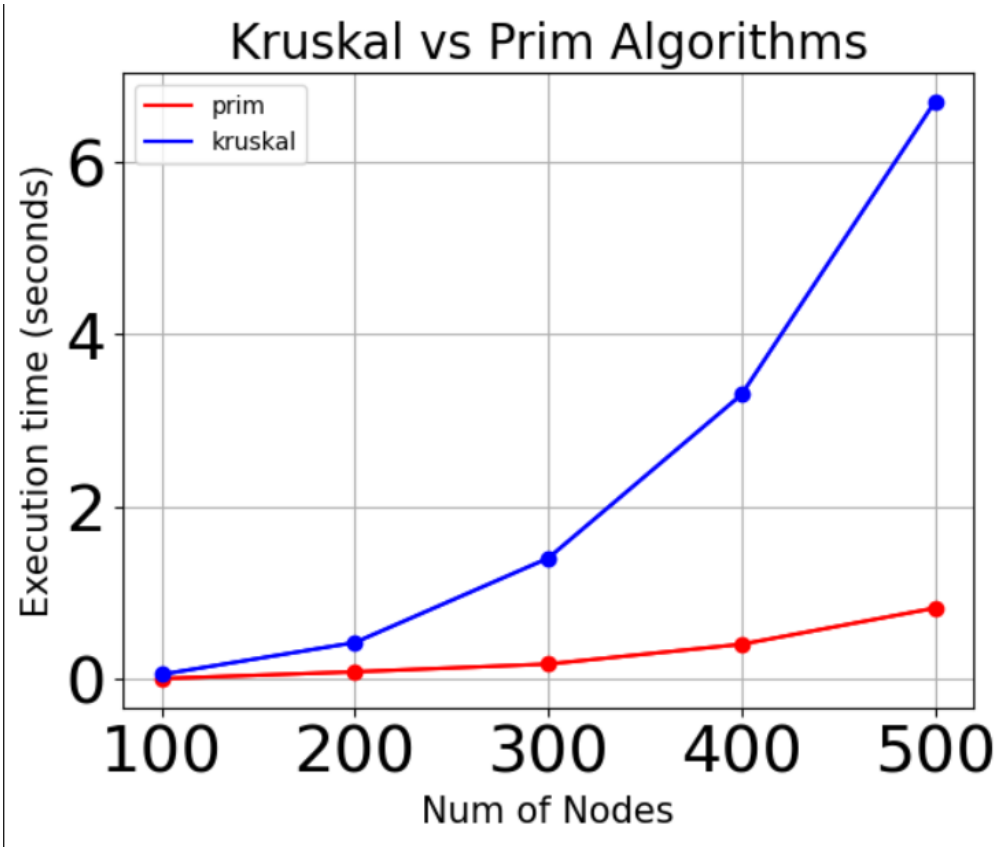
```
data_prim = {'size': ['100', '200', '300', '400', '500'],
        'time': [0.001, 0.08, 0.17, 0.4, 0.82]
       }
data_kruskal = {'size': ['100', '200', '300', '400', '500'],
        'time': [0.05, 0.42, 1.4, 3.3, 6.7]
       }

df = pd.DataFrame(data_prim)
df_it = pd.DataFrame(data_kruskal)
plt.rcParams.update({'font.size': 27})
plt.plot(df['size'], df['time'], "-r", label="prim")
plt.plot(df_it['size'], df_it['time'], "-b", label="kruskal")
plt.legend(loc="upper left", prop={'size': 10})

plt.plot(df['size'], df['time'], color='red', marker='o', )
plt.plot(df_it['size'], df_it['time'], color='blue', marker='o',)
plt.title('Kruskal vs Prim Algorithms', fontsize=20)
plt.xlabel('Num of Nodes', fontsize=15)
plt.ylabel('Execution time (seconds)', fontsize=15)

plt.grid(True)
```

- ❖ Програмний код, який створює графік для порівняння алгоритмів Флойда-Воршала реалізованого вручну та алгоритму Флойда-Воршала, реалізованого з вбудованим алгоритмом.

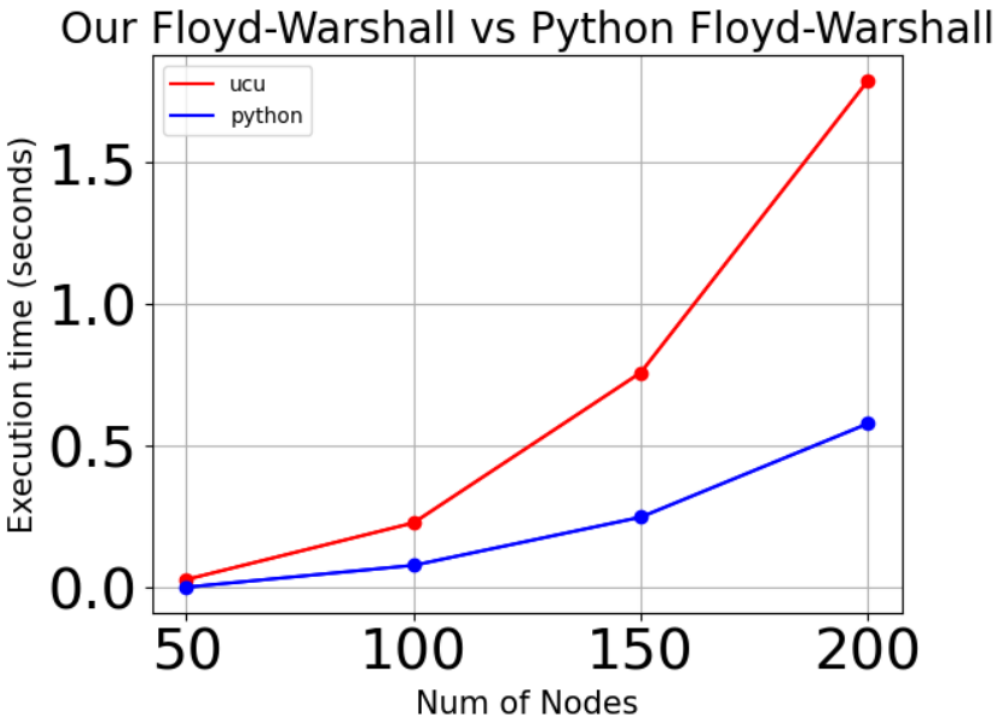
```
data_floyd_warshall_ucu = {'size': ['50', '100', '150', '200'],
        'time': [0.03, 0.23, 0.76, 1.79]
       }
data_floyd_warshall = {'size': ['50', '100', '150', '200'],
        'time': [0.003, 0.08, 0.25, 0.58]
       }

df = pd.DataFrame(data_floyd_warshall_ucu)
df_it = pd.DataFrame(data_floyd_warshall)
plt.rcParams.update({'font.size': 27})
plt.plot(df['size'], df['time'], "-r", label="ucu")
plt.plot(df_it['size'], df_it['time'], "-b", label="python")
plt.legend(loc="upper left", prop={'size': 10})
plt.plot(df['size'], df['time'], color='red', marker='o', )
plt.plot(df_it['size'], df_it['time'], color='blue', marker='o',)
plt.title('Our Floyd-Warshall vs Python Floyd-Warshall', fontsize=20)
plt.xlabel('Num of Nodes', fontsize=15)
plt.ylabel('Execution time (seconds)', fontsize=15)
plt.grid(True)
```

★ Графіки часу роботи:

- ❖ Ми порівнювали ефективність роботи алгоритмів Прима та Крускала. На зображені ми бачимо, що ефективність алгоритму Прима набагато краща за алгоритм Крускала.



- Порівнявши ефективність роботи алгоритму Флойда-Воршала, реалізованого вручну, та алгоритму Флойда-Воршала, реалізованого з вбудованим алгоритмом, ми бачимо, що вбудований алгоритм працює швидше, коли вершин є більше ніж 50.



★ Висновки:

- ❖ Дослідивши роботу алгоритмів Прима та Крускала, ми переконалися, що алгоритм Прима є більш ефективним і при збільшенні кількості вершин працював набагато швидше ніж алгоритм Крускала.
- ❖ В експерименті було проведено порівняння алгоритму Флойда-Воршала, реалізованого вручну, та алгоритму Флойда-Воршала, реалізованого з вбудованим алгоритмом. Алгоритм вбудованої бібліотеки є більш ефективним ніж той, який був написаний вручну. Також коли вершин у графі є більше ніж 50, то вбудований алгоритм працює швидше ніж той, який реалізований вручну.