ЗВІТ

Підготували: Штогрин Олег та Валігурський Антон

Опис задачі: написати алгоритми: Прима та Краскла- для знаходження найменшого каркасу; Флойда- Ворашала для знаходження найкоротших відстаней між усіма вершинами та Белмана- Форда для знаходження найкоротшої відстані від заданої точки.

Опис експерименту: дослідити ефективність написаних нами алгоритмів відносно вбудованих.

За яким принципом проводиться експеримент: ми будемо порівнювати наші та вбудовані алгоритми на вершинах: 5, 10, 20, 50, 100, 250, 500. Було використано бібліотеку time для занотування результатів ефективності алгоритмів. Графік роботи буде представлено за допомою бібліотеки matplotlib.pylot.

Хід роботи:

Експеримент 1:

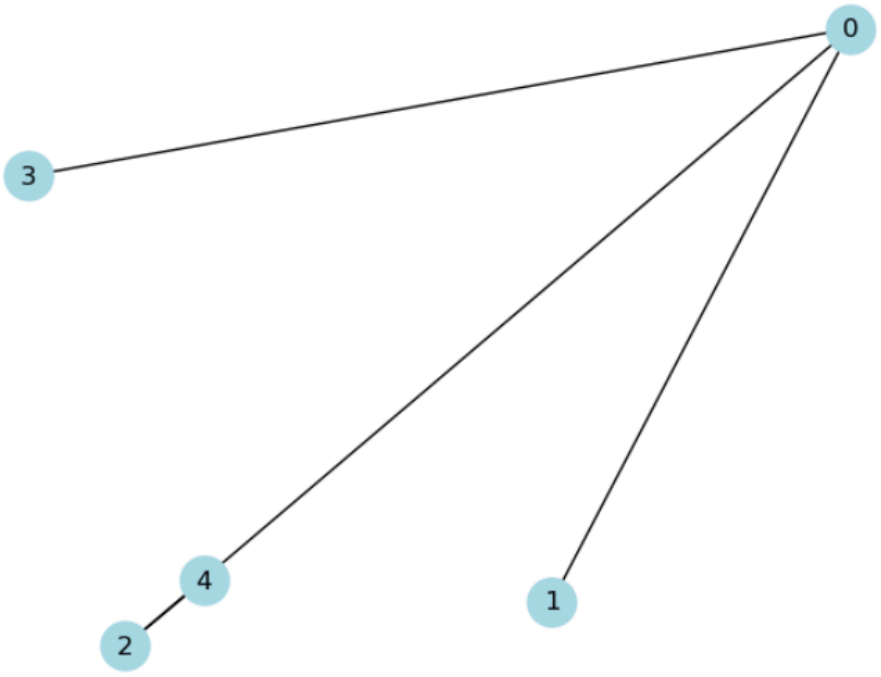
Алгоритм Прима:

Код:

```python
def prims_algorithm(graph, starting_node: int):

    frame = nx.Graph()
    frame.add_node(starting_node)

    num_of_nodes = len(graph)
    accessed_notes = [starting_node]

    while len(accessed_notes) < num_of_nodes:

        min_weight, new_min_node = inf, (-1, -1)

        for node in accessed_notes:
            for neighbor, info in graph[node].items():
                if info["weight"] < min_weight and neighbor not in accessed_notes:
                    min_weight = info["weight"]
                    new_min_node = node, neighbor

        accessed_notes.append(new_min_node[1])
        frame.add_node(new_min_node[1])
        frame.add_edge(*new_min_node, weight = min_weight)

    return frame

self_done = prims_algorithm(G, 1)

nx.draw(self_done, node_color='lightblue',
        with_labels=True,
        node_size=500)

for edge in self_done.edges:
    print(self_done.get_edge_data(*edge))
print(sum(self_done.get_edge_data(*edge)["weight"] for edge in self_done.edges))
```

Вивід:

```
{'weight': -4}
{'weight': -4}
{'weight': 0}
{'weight': 11}
3
```



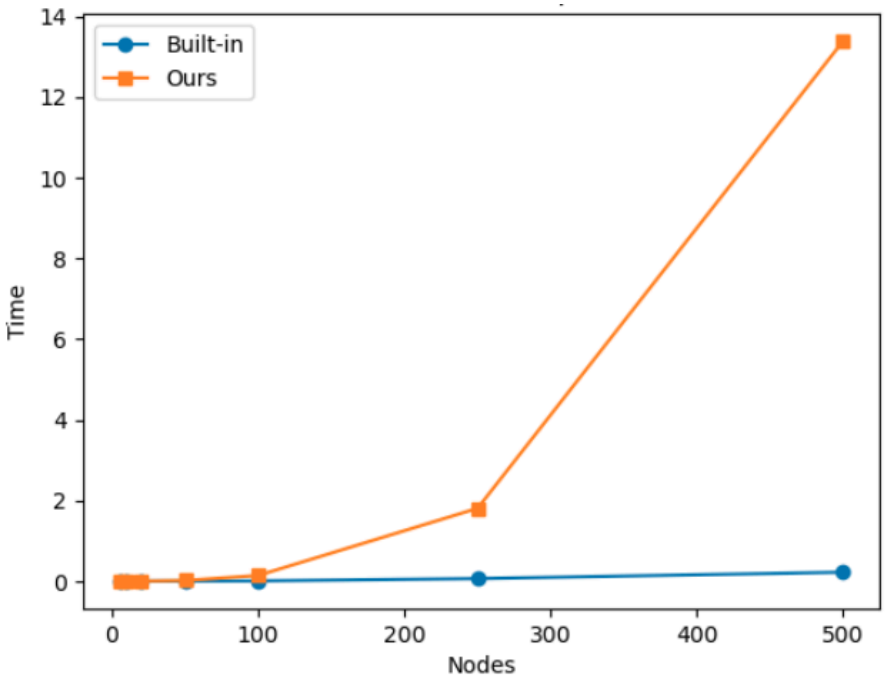
Код проведення експерименту:

```
import matplotlib.pyplot as plt
from time import time
vertices_num = [5,10,20,50,100,250,500]
y1 = []
y2 = []

for i in vertices_num:

    time_sum = 0
    time_sum_own = 0
    for _ in range(4):

        G_ = gnp_random_connected_graph(i, 1, False, False)

        print(i, "nodes")

        start = time()
        mstp_ = tree.minimum_spanning_tree(G_, algorithm="prim")
        stop = time() - start
        print(f"Built-in algorithm worked in: {stop:.3f}. With total weight of: {sum(mstp_.get_edge_data(*edge)['weight']
        time_sum += stop

        start = time()
        mstp_own = prims_algorithm(G_, 1)
        stop = time() - start
        print(f"Our algorithm worked in: {stop:.3f}. With total weight of: {sum(mstp_own.get_edge_data(*edge)['weight'] f
        time_sum_own += stop

    y1.append(float(f"{time_sum/4:.3f}"))
    y2.append(float(f"{time_sum_own/4:.3f}"))

plt.plot(vertices_num, y1, label='Built-in', marker='o')

# Plotting the second dataset
plt.plot(vertices_num, y2, label='Ours', marker='s')

# Adding labels and title
plt.xlabel('Nodes')
plt.ylabel('Time')
plt.title('Built-in kraskal vs our implementation')

# Adding a legend to differentiate between datasets
plt.legend()

# Display the plot
plt.show()
```

Приклад виводу:

```
5 nodes
Built-in algorithm worked in: 0.000. With total weight of: 12
Our algorithm worked in: 0.000. With total weight of: 12

5 nodes
Built-in algorithm worked in: 0.001. With total weight of: 5
Our algorithm worked in: 0.000. With total weight of: 5


500 nodes
Built-in algorithm worked in: 0.194. With total weight of: -2495
Our algorithm worked in: 13.442. With total weight of: -2495

500 nodes
Built-in algorithm worked in: 0.205. With total weight of: -2495
Our algorithm worked in: 13.295. With total weight of: -2495
```

Графік роботи вбудованого та нашого алгоритму:



Підсумок: вбудований алгоритм працює краще нашого, хоча на невеликій кількості вершин це не є помітним. Наш алгоритм працює гірше через те, що у вбудованому є додаткові умови(if, else), які не дають робити ‘зайві’ кроки алгоритму, тобто пропускаючи непотрібні ітерації.

Експеримент 2:

Алгоритм Краскла:

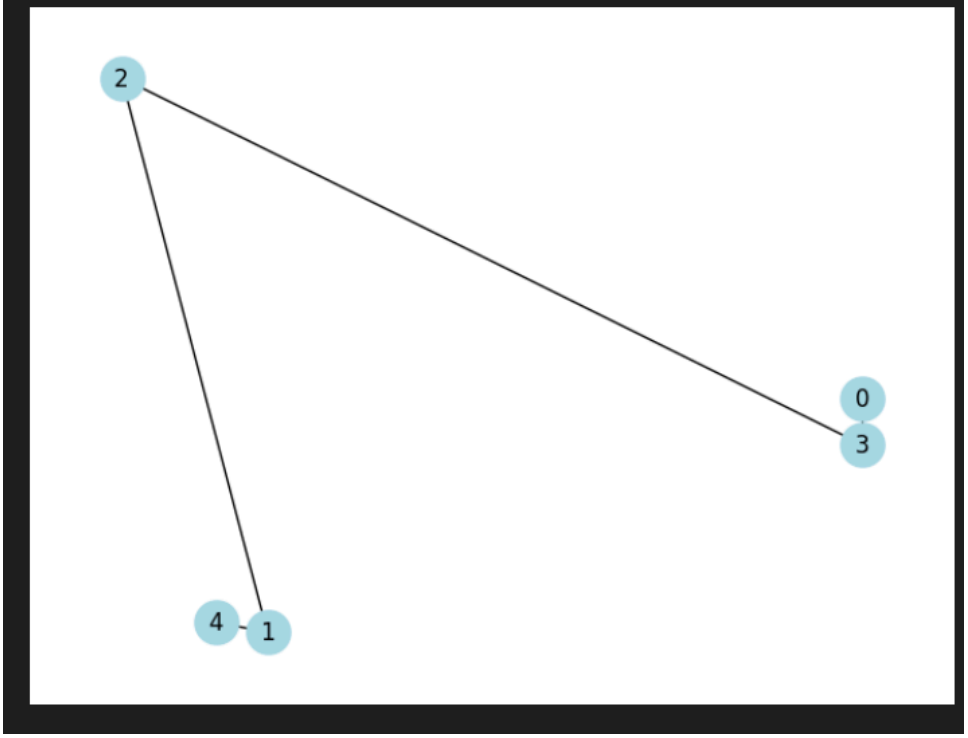
Код:

```python
from copy import deepcopy

def make_stek(connected_lst, edge: list) -> list | bool:
    """
    Creates a stek
    """
    stek1 = None
    stek2 = None
    copy_connected = deepcopy(connected_lst)
    for _, stek in enumerate(connected_lst):

        if edge[0] in stek and edge[1] in stek:
            return False

        elif edge[0] in stek:
            stek1 = stek
            copy_connected.remove(stek)

        elif edge[1] in stek:
            stek2 = stek
            copy_connected.remove(stek)

    stek1.update(stek2)
    copy_connected.append(stek1)
    return copy_connected
```

```python
def kruskal(g):
    """
    Make a Kruskal algorythm
    """
    frame_krusk = nx.Graph()
    edges = {(key, key1): value1['weight'] for key, value in g.adj.items() for key1, value1 in value.items()}

    sorted_edges = dict(sorted(edges.items(), key = lambda x: x[1]))

    stek_nodes = list({nd} for nd in g.nodes)
    lst_nodes = []
    for edge, weigh in sorted_edges.items():

        if len(stek_nodes) == 1:
            return frame_krusk

        copy_stek = deepcopy(stek_nodes)

        if make_stek(copy_stek, edge):
            stek_nodes = make_stek(stek_nodes, edge)
            lst_nodes.append(edge)
            frame_krusk.add_edge(*edge, weight = weigh)

    return frame_krusk

frame_krsk = kruskal(G)
```

Візуалізація:

```
nx.draw(frame_krsk, node_color='lightblue',
        with_labels=True,
        node_size=500)
```



Код проведення експерименту:

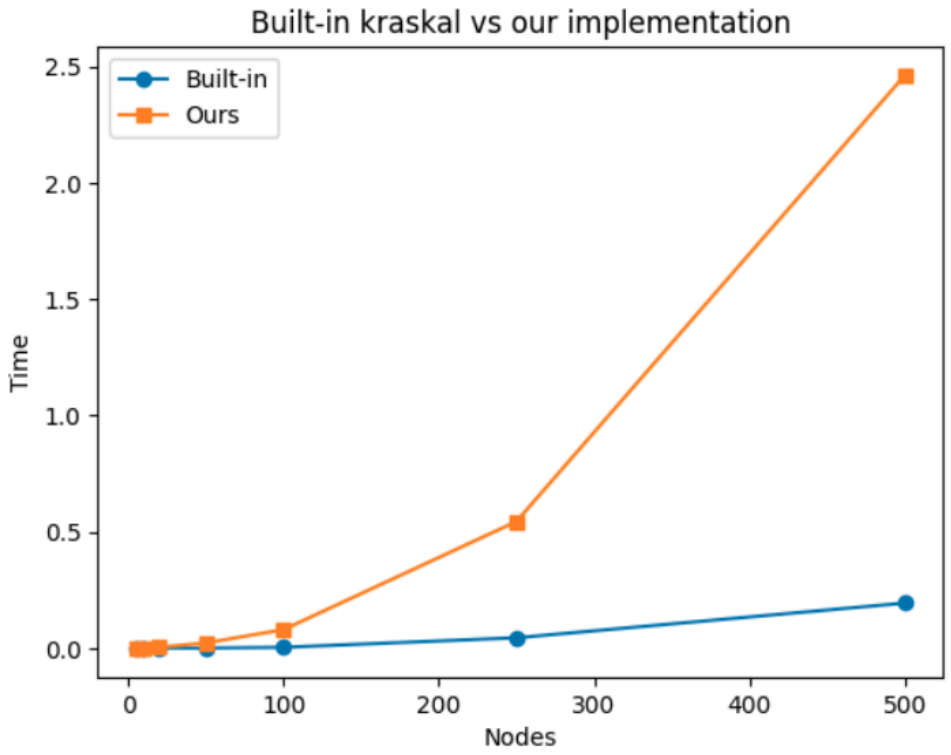
```python
import matplotlib.pyplot as plt
from time import time
vertices_num = [5,10,20,50,100,250,500]
y1 = []
y2 = []

for i in vertices_num:

    time_sum = 0
    time_sum_own = 0
    for _ in range(4):

        G_ = gnp_random_connected_graph(i, 1, False, False)

        print(i, "nodes")

        start = time()
        mstp_ = tree.minimum_spanning_tree(G_, algorithm="kruskal")
        stop = time() - start
        print(f"Built-in algorithm worked in: {stop:.3f}. With total weight of: {sum(mstp_.get_edge_data(*edge)['weight'] for edge in mstp_.edges)}")
        time_sum += stop

        start = time()
        mstp_own = kruskal(G_)
        stop = time() - start
        print(f"Our algorithm worked in: {stop:.3f}. With total weight of: {sum(mstp_own.get_edge_data(*edge)['weight'] for edge in mstp_own.edges)}\n")
        time_sum_own += stop

    y1.append(float(f"{time_sum/4:.3f}"))
    y2.append(float(f"{time_sum_own/4:.3f}"))

plt.plot(vertices_num, y1, label='Built-in', marker='o')

# Plotting the second dataset
plt.plot(vertices_num, y2, label='Ours', marker='s')

# Adding labels and title
plt.xlabel('Nodes')
plt.ylabel('Time')
plt.title('Built-in kraskal vs our implementation')

# Adding a legend to differentiate between datasets
plt.legend()

# Display the plot
plt.show()
```

Приклад виводу:

```
5 nodes
Built-in algorithm worked in: 0.000. With total weight of: -4
Our algorithm worked in: 0.000. With total weight of: -4

5 nodes
Built-in algorithm worked in: 0.000. With total weight of: 3
Our algorithm worked in: 0.001. With total weight of: 3
```

```
500 nodes
Built-in algorithm worked in: 0.202. With total weight of: -2495
Our algorithm worked in: 2.337. With total weight of: -2495
```

Графік роботи вбудованого та нашого алгоритму:



Підсумок: як і з алгоритмом Прима: вбудований алгоритм працює краще нашого, хоча на невеликій кількості вершин це не є помітним. Це пов’язано з тим, що функція make_stek повторюється 2 рази , перед тим, як додати ребро, що ускладнює алгоритм, враховуючи, що в самій функції, виконуються операції copy та update, що дещо більше заповнює пам’ять та ускладнює алгоритм.

Експеримент 3:

Алгоритм Флойда- Воршала:

Код:

```
def floyd_warshall(graph):
    marks = [[] for _ in range(len(graph))]
    warshall = [[] for _ in range(len(graph))]


    for i in range(len(graph)):
        for j in range(len(graph)):
            if j in graph[i]:
                warshall[i].append(graph[i][j]["weight"])
            elif i == j:
                warshall[i].append(0)
            else:
                warshall[i].append(inf)
            if i==j:
                marks[i].append(-1)
            else:
                marks[i].append(i)
```

```
    for i, w_row in enumerate(warshall):
        new_warshall = []
        new_marks = []

        for j, row in enumerate(warshall):
            new_warshall.append([])
            new_marks.append([])
            for k, el in enumerate(row):
                if  warshall[i][k] + warshall[j][i]<el:
                    new_warshall[-1].append(warshall[i][k] + warshall[j][i])
                    new_marks[-1].append(marks[i][k])
                else:
                    new_warshall[-1].append(el)
                    new_marks[-1].append(marks[j][k])

        warshall = new_warshall
        marks = new_marks

    marks ={k:{i: v for i,v in enumerate(row)} for k, row in enumerate(marks)}
    warshall ={k:{i: v for i,v in enumerate(row)} for k, row in enumerate(warshall)}

    return marks, warshall

marks, matrix = floyd_warshall(G)


for i, row in marks.items():
    print(i, ":", row)
for i, row in matrix.items():
    print(i, ":", row)
```

Приклад виводу:

```
0 : {0: -1, 1: 0, 2: 0, 3: 0, 4: 1}
1 : {0: 4, 1: -1, 2: 1, 3: 1, 4: 1}
2 : {0: 4, 1: 0, 2: -1, 3: 2, 4: 2}
3 : {0: 4, 1: 0, 2: 0, 3: -1, 4: 3}
4 : {0: 4, 1: 0, 2: 0, 3: 4, 4: -1}
0 : {0: 0, 1: -5, 2: 5, 3: 3, 4: -9}
1 : {0: 12, 1: 0, 2: 16, 3: 13, 4: -4}
2 : {0: 35, 1: 30, 2: 0, 3: 18, 4: 19}
3 : {0: 28, 1: 23, 2: 33, 3: 0, 4: 12}
4 : {0: 16, 1: 11, 2: 21, 3: 18, 4: 0}
```

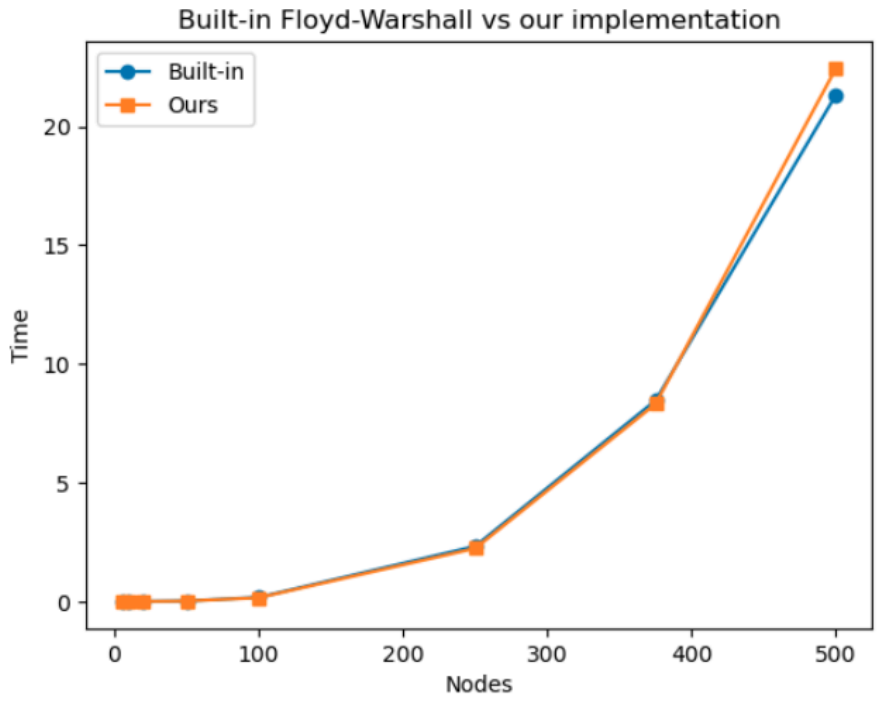
Код проведення експерименту:

```
import matplotlib.pyplot as plt
from time import time
vertices_num = [5,10,20,50,100,250,375, 500]
y1 = []
y2 = []

for i in vertices_num:
    time_sum = 0
    time_sum_own = 0

    G_ = gnp_random_connected_graph(i, 1, False, False)

    start = time()
    predecessors, dist = floyd_warshall_predecessor_and_distance(G_)
    stop = time() - start
    print(f"{i}nodes.\nBuilt-in algorithm worked in: {stop:.3f}")
    time_sum = stop

    start = time()
    marks, matrix = floyd_warshall(G_)
    stop = time() - start
    print(f"Our algorithm worked in: {stop:.3f}\n")
    time_sum_own = stop

    y1.append(float(f"{time_sum:.8f}"))
    y2.append(float(f"{time_sum_own:.8f}"))



plt.plot(vertices_num, y1, label='Built-in', marker='o')

# Plotting the second dataset
plt.plot(vertices_num, y2, label='Ours', marker='s')

# Adding labels and title
plt.xlabel('Nodes')
plt.ylabel('Time')
plt.title('Built-in Floyd-Warshall vs our implementation')

# Adding a legend to differentiate between datasets
plt.legend()

# Display the plot
plt.show()
```

Приклад виводу:

```
5nodes.
Built-in algorithm worked in: 0.000
Our algorithm worked in: 0.000

10nodes.
Built-in algorithm worked in: 0.000
Our algorithm worked in: 0.000

20nodes.
Built-in algorithm worked in: 0.002
Our algorithm worked in: 0.002

50nodes.
Built-in algorithm worked in: 0.020
Our algorithm worked in: 0.019

100nodes.
Built-in algorithm worked in: 0.174
Our algorithm worked in: 0.158

250nodes.
Built-in algorithm worked in: 2.325
Our algorithm worked in: 2.235

375nodes.
Built-in algorithm worked in: 8.465
Our algorithm worked in: 8.319

500nodes.
Built-in algorithm worked in: 21.338
Our algorithm worked in: 22.456
```

Графік роботи вбудованого та нашого алгоритму:



Підсумок: Наш алгоритм працює практично з такою самою ефективністю, як і вбудований. На малій кількості вершим він навіть дещо ефективніше. Невелика перевага, яку ми бачимо на 500 вершинах пов’язана з тим, що вбудований алгоритм має меншу складність: O(n^3).

Експеримент 4:

Алгоритм Белмана- Форда:

Код:

```python
#if we start from 0

def bell_ford(G__):
    dct_distance = {x: G__.adj[0][x]['weight'] if x in G__.adj[0] else float('inf') for x in G__.nodes}
    dct_distance[0] = 0
    nodes = list(G__.nodes)[1:]
    g = nx.Graph()

    for _ in range(len(dct_distance) - 2):
        for v in nodes:
            for u in G__.nodes:
                if u in G__.adj and v in G__.adj[u]:
                    dct_distance[v] = min(dct_distance[v], dct_distance[u] + G__.adj[u][v]['weight'])
                    g.add_edge(u, v, weight = min(dct_distance[v], dct_distance[u] + G__.adj[u][v]['weight']))

    for v1 in nodes:
        for u1 in G__.nodes:
            if  u1 in G__.adj and v1 in G__.adj[u1] and dct_distance[v1] > dct_distance[u1] + G__.adj[u1][v1]['weight']:
                return 'Negative cycle detected'

    return g

grph_bell_ford = bell_ford(G)
print(grph_bell_ford.edges)
```

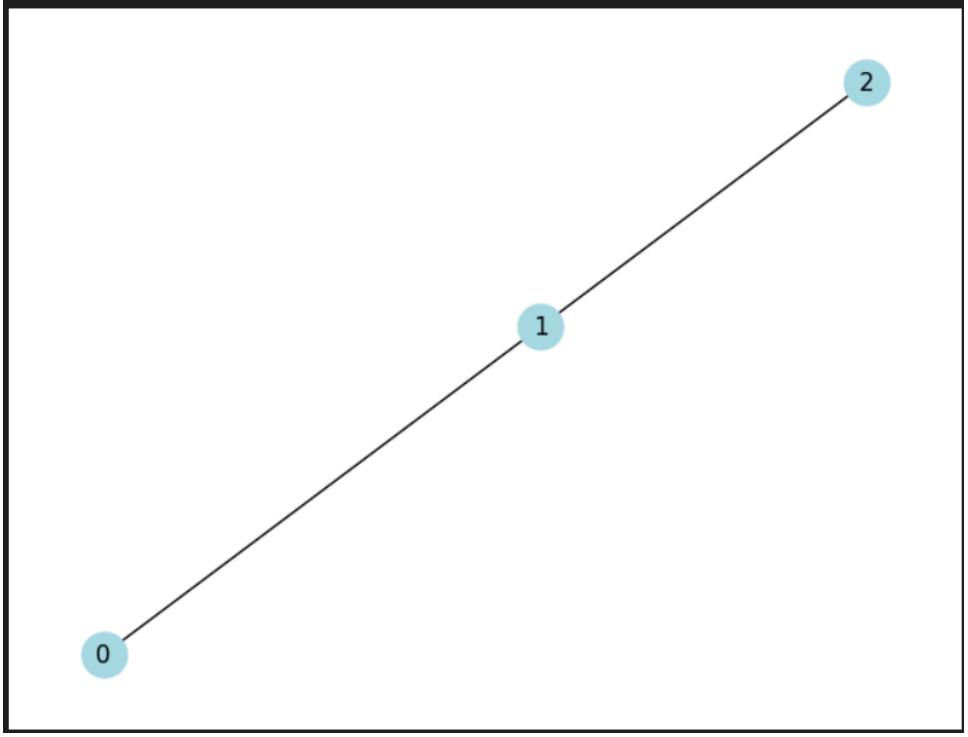
Приклад виводу:

```
[(0, 1), (1, 2)]
```

Приклад візуалізації:

```
nx.draw(grph_bell_ford , node_color='lightblue',
        with_labels=True,
        node_size=500)
```

0.0s



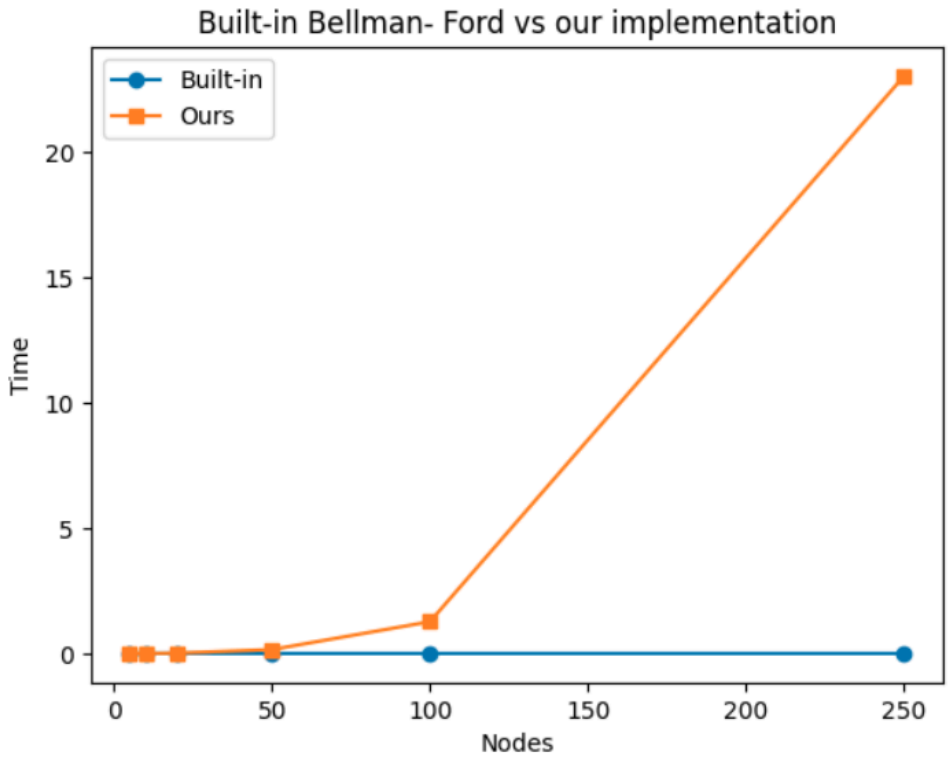
Код проведення експерименту:

```python
import matplotlib.pyplot as plt
from time import time
vertices_num = [5,10,20,50,100,250]
y1 = []
y2 = []

for i in vertices_num:
    time_sum = 0
    time_sum_own = 0

    G_ = gnp_random_connected_graph(i, 1, False, False)

    start = time()
    predecessors, dist = bellman_ford_predecessor_and_distance(G, 0)
    stop = time() - start
    print(f"{i}nodes.\nBuilt-in algorithm worked in: {stop:.3f}")
    time_sum = stop

    start = time()
    grph = bell_ford(G_)
    stop = time() - start
    print(f"Our algorithm worked in: {stop:.3f}\n")
    time_sum_own = stop

    y1.append(float(f"{time_sum:.8f}"))
    y2.append(float(f"{time_sum_own:.8f}"))



plt.plot(vertices_num, y1, label='Built-in', marker='o')

# Plotting the second dataset
plt.plot(vertices_num, y2, label='Ours', marker='s')

# Adding labels and title
plt.xlabel('Nodes')
plt.ylabel('Time')
plt.title('Built-in Bellman- Ford vs our implementation')

# Adding a legend to differentiate between datasets
plt.legend()

# Display the plot
plt.show()
```

Приклад виводу:

```
100nodes.
Built-in algorithm worked in: 0.000
Our algorithm worked in: 1.268
```

Графік роботи вбудованого та нашого алгоритму:



Підсумок: Алгоритм працює гірше вбудованого. На малих вершинах алгоритм працює непогано. Це може бути пов’язано з способом перевірки на негативні цикли вбудованого алгоритму та нашого: можливо, вбудований алгоритм одразу перевіряє наявність негативних простих циклів та видає про це повідомлення, коли ж наш алгоритм робить це опісля всіх дій.

Загальний підсумок:

Алгоритми Краскла та Прима найкраще працюють на графах з малою кількістю вершин, очевидно, пов’язано з тим, що алгоритм має проходитись по значно більшій кількості ребер, що, практично, експоненційно збільшує його час.

Алгоритм Флойда- Ворашала, в порівнянні з вбудованим, працює просто прекрасно на будь- якій кількості вершин(як мінімум до 500). Якщо дивитись не в порівняні, то час так

само збільшується експоненційно, через більшу кількість операцій.

Наш алгоритм Белмана- Форда може зрівнятись з вбудованим до приблизно 50 вершин. Пізніше він видає гірші результати.

Нюанси: можлива похибка в графіках, яка пов’язана з особливістю роботи процесора, адже він постійно пробує ‘вгадати’ наступні операції, через це дані експерименту можуть різнитись на тисячні, а то й соті секунди, бо в одному випадку процесор ‘вгадав’ наступну операцію, а в іншому ні.

Невелике порівняння наших алгоритмів:

Краскалу на 250вершин вдалось знайти каркас за 0.542 секунди, коли ж Приму знадобилось 1.171 секунди.

Белману- Форду на подолання 250 вершин знадобилось 23 секунди, коли ж Флойду- Воршалу тільки 2.235 секунди.

Decision tree

1) Визначає gini, на осонові якого визначається наскільки вдалим є розприділення.

```python
@staticmethod
def gini(groups):
    '''
    A Gini score gives an idea of how good a split is by how mixed the
    classes are in the two groups created by the split.

    A perfect separation results in a Gini score of 0,
        whereas the worst case split that results in 50/50
        classes in each group result in a Gini score of 0.5
    (for a 2 class problem).
    '''

    total = sum(groups)

    return 1 - sum((p_k/total)**2 for p_k in groups)
```

2) Перевіряє всі можливі розподіли в межах O(N*F), N- кількість зразків, F- кількість ‘особливостей’.

```python
    def split_data(self, X, y) -> tuple[int, int]:

        # test all the possible splits in O(N*F) where N in number of samples
        # and F is number of features

        # return index and threshold value

        m = y.size
        if m <= 1:
            return None, None

        parent_num = [np.sum(y == c) for c in range(self.classes)]

        best_gini = self.gini(parent_num)

        best_ind, best_thr = None, None


        for ind in range(self.features):
            thresholds, classes = zip(*sorted(zip(X[:, ind], y)))
            left_num = [0] * self.classes
            right_num = parent_num[:]
            for i in range(1,m):
                cut = classes[i]

                left_num[cut] += 1
                right_num[cut] -= 1
```

```
            left_num[cut] += 1
            right_num[cut] -= 1

            if thresholds[i] == thresholds[i-1]:
                continue

            gini = (i * self.gini(left_num) + (m-i)*self.gini(right_num)) / m


            if gini < best_gini:
                best_gini = gini
                best_ind = ind
                best_thr = (thresholds[i] + thresholds[i-1]) / 2

    return best_ind, best_thr
```

3) Створює кореневе дерево, рекурсивно розділяючи по ‘gini’ до того моменту, поки не буде досягнуто максимальної ‘глибини’.

```
def build_tree(self, X, y, depth = 0):

    # create a root node

    # recursively split until max depth is not exeeced
    class_samples_num = [np.sum(y == i) for i in range(self.classes)]
    node = Node(
        X,
        np.argmax(class_samples_num),
        self.gini(class_samples_num),
    )

    if depth < self.max_depth:
        ind, thresh_hold = self.split_data(X, y)

        if ind is not None:
            X_left, y_left, X_right, y_right =[],[],[],[]
            for i, row in enumerate(X):
                if row[ind] < thresh_hold:
                    X_left.append(row)
                    y_left.append(y[i])
                else:
                    X_right.append(row)
                    y_right.append(y[i])
```

```
                else:
                    X_right.append(row)
                    y_right.append(y[i])


            node.feature_index = ind
            node.threshold = thresh_hold

            node.left = self.build_tree(np.array(X_left), np.array(y_left), depth + 1)
            node.right = self.build_tree(np.array(X_right), np.array(y_right), depth + 1)

        return node
```

4) Визначає кількість ‘зразків’, ‘особливостей’ та генерує дерево

```
def fit(self, X, y):

    # basically wrapper for build tree / train
    self.classes = len(set(y))
    self.features = X.shape[1]
    self.tree = self.build_tree(X, y)
```

5) Обходить все дерево, поки в нього все ще є ‘діти’, вертаючи при цьому передбачений клас

```
def predict(self, X_test):

    # traverse the tree while there is a child
    # and return the predicted class for it,
    # note that X_test can be a single sample or a batch

    classes = []

    for input in X_test:
        node = self.tree
        while node.left:
            if input[node.feature_index] < node.threshold:
                node = node.left
            else:
                node = node.right
        classes.append(node.y)

    return classes
```

6) Повертає точність наших передбачень

```
def evaluate(self, X_test, y_test):

    # return accuracy

    test = self.predict(X_test)

    return sum(test == y_test) / len(y_test)
```

В кінці ми просто викликаєм все:

```
my_clf = MyDecisionTreeClassifier(2)
my_clf.fit(X, y)
print(my_clf.evaluate(X_test, y_test))
```

Та отримуємо точність нашого передбачення щодо дерева:

```
0.9666666666666667
```