1. Роботу викнали Чупа Орест та Артем Крутько, команда №17.
2. Постановка задачі та експерименти. Задачею було реалізувати декілька алгоритмів, а саме – Прима, Крускала, Белмана-Форда та Флойда-Воршала.
Експериментів було декілька, більшість із них стосувалися ефективності алгоритму у порівнянні з іншими анлогічними алгоритмами.

Алгоритм Прима

```python
def has_cycle(graph):
    graph = rewrite_graph(graph)
    visited = set()

    def dfs(node, parent):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor, node):
                    return True
            elif neighbor != parent:
                return True
        return False

    for node in graph:
        if node not in visited:
            if dfs(node, None):
                return True
    return False
def rewrite_graph(graph):
    result = dict()
    edges = sorted(graph, key=lambda x: x[1])
    for edge in edges:
        edge = [str(edge[0]), str(edge[1])]
        if edge[0] not in result.keys():
            result[edge[0]] = []
            # print(result[edge[0]])
        if edge[1] not in result.keys():
            result[edge[1]] = []
            # print(result[edge[1]])
        result[edge[0]] = result[edge[0]]+[edge[1]]
        result[edge[1]] = result[edge[1]]+[edge[0]]
    return result
```

```python
def max_v(lst):
    result = 0
    for i in lst:
        num = max(i[0], i[1])
        if num > result:
            result = num
    return result

def prima(G, begin_point=0):
    edges = sorted(G.edges(data=True), key=lambda x: x[2]['weight'])
    result = []
    allowed_vertixes = {begin_point}
    max_vert = max_v(edges)
    for i in range(len(edges)):
        temp_edges = []
        if edges == []:
            break
        if len(allowed_vertixes) >= max_vert:
            break
        for edge in edges:
            if (edge[0] in allowed_vertixes or edge[1] in allowed_vertixes) and not has_cycle(result + [edge]):
                result.append((edge[0], edge[1]))
                allowed_vertixes.add(edge[1])
                allowed_vertixes.add(edge[0])
                edges.remove(edge)
                break
    return result
```

Прима був реалізований за допомогою трьох функцій, дві з яких (has_cycle та rewrite_graph) допоміжні.
has_cycle – перевіряє чи граф має в собі цикл. Це реалізовано за допомогою dfs алгоритму.
Суть цієї функції перевіряти чи при виборі та додаванні якогось ребра до фінального каркасу не з’явиться цикл.
max_v – вертає максимальну кількість вершин
rewrite_graph – переписує граф з [(0, 1),(0, 2), (1, 2)] у такий вигляд:
0: 1, 2
1: 0, 2
2: 0, 1
Тобто так, що одразу видно сусідів вершини.
prima – основний алгоритм. Якщо дуже коротко, то стартовою вершиною за замовчуванням є 0. Далі усі ребра графу сортуються за вагою, від меншого до більшого. Потім ми циклом перебираємо усі ребра, якщо знаходимо ребро яке виходить або входить у вершину яку ми маємо, то це ребро додається до результату, а нова вершина у яку ми прийшли до списку дозволених вершин. Також додане ребро видаляється з графу, щоб було менше ітерацій.
На кінець, ще є запобіжник який перевіряє чи не були знайдені шляхи до усіх вершин, якщо знайдено, то вертається результат. Деколи результат нашого алгоритму не співпадає з бібліотечним. Причина цьому різний підхід до вибору ребер з одинаковою вагою.

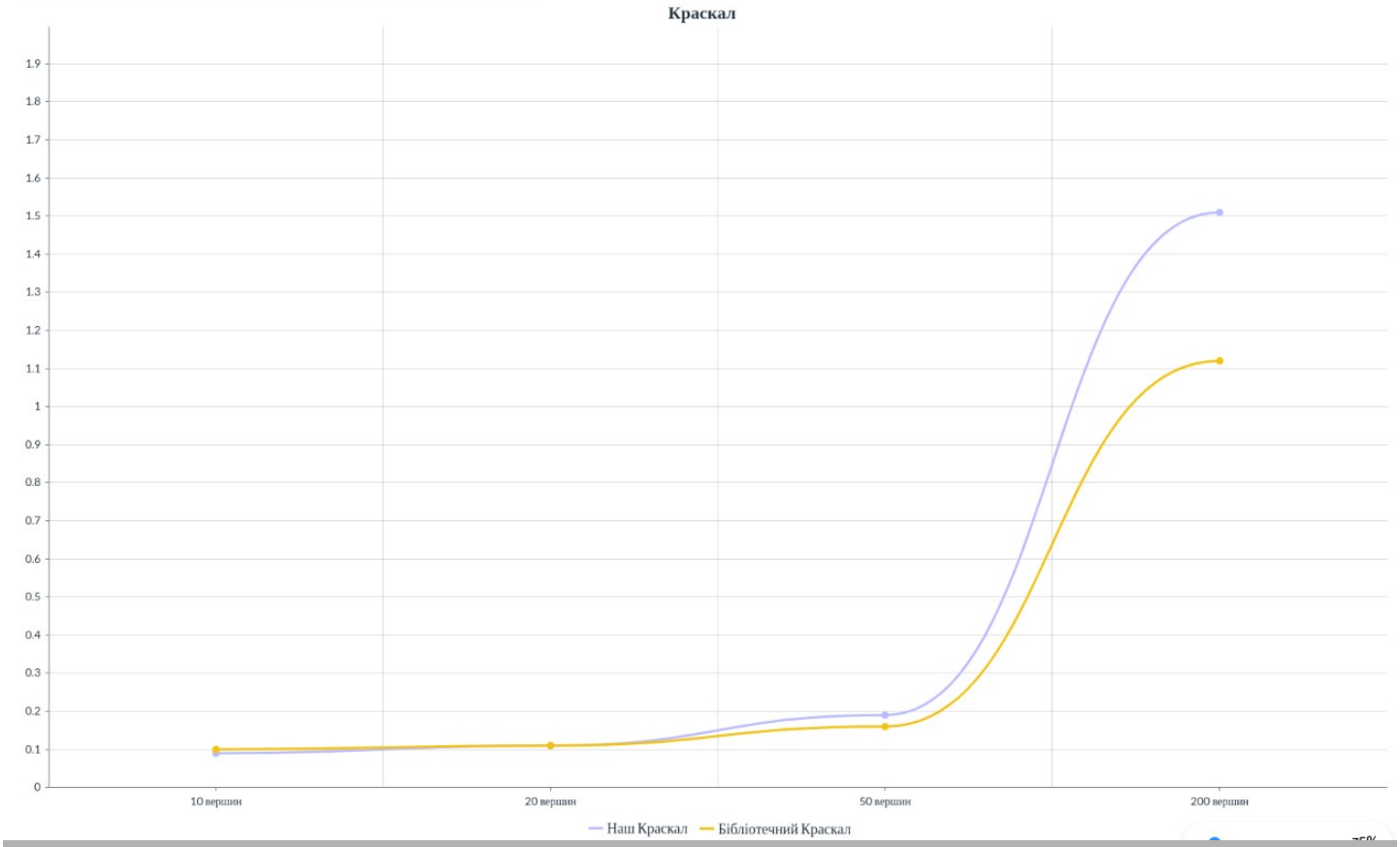
Якщо порівнювати швидкість цього алгоритму та вбудованого у бібліотеку networkx то результати будуть такими(з максимальною кількістю ребер):
Наш прима:
10 вершин – 0.12с
20 вершин – 0.13с
50 вершин – 0.18с
200 вершин – 9.2с
Бібліотечний пріма:
10 вершин – 0.105с
20 вершин – 0.108с
50 вершин – 0.16с
200 вершин – 1.13с
Чим більше вершин, тим більша різниця, якось так. Очевидно є ще якась деталь яка була пропущена під час розробки алгоритму.

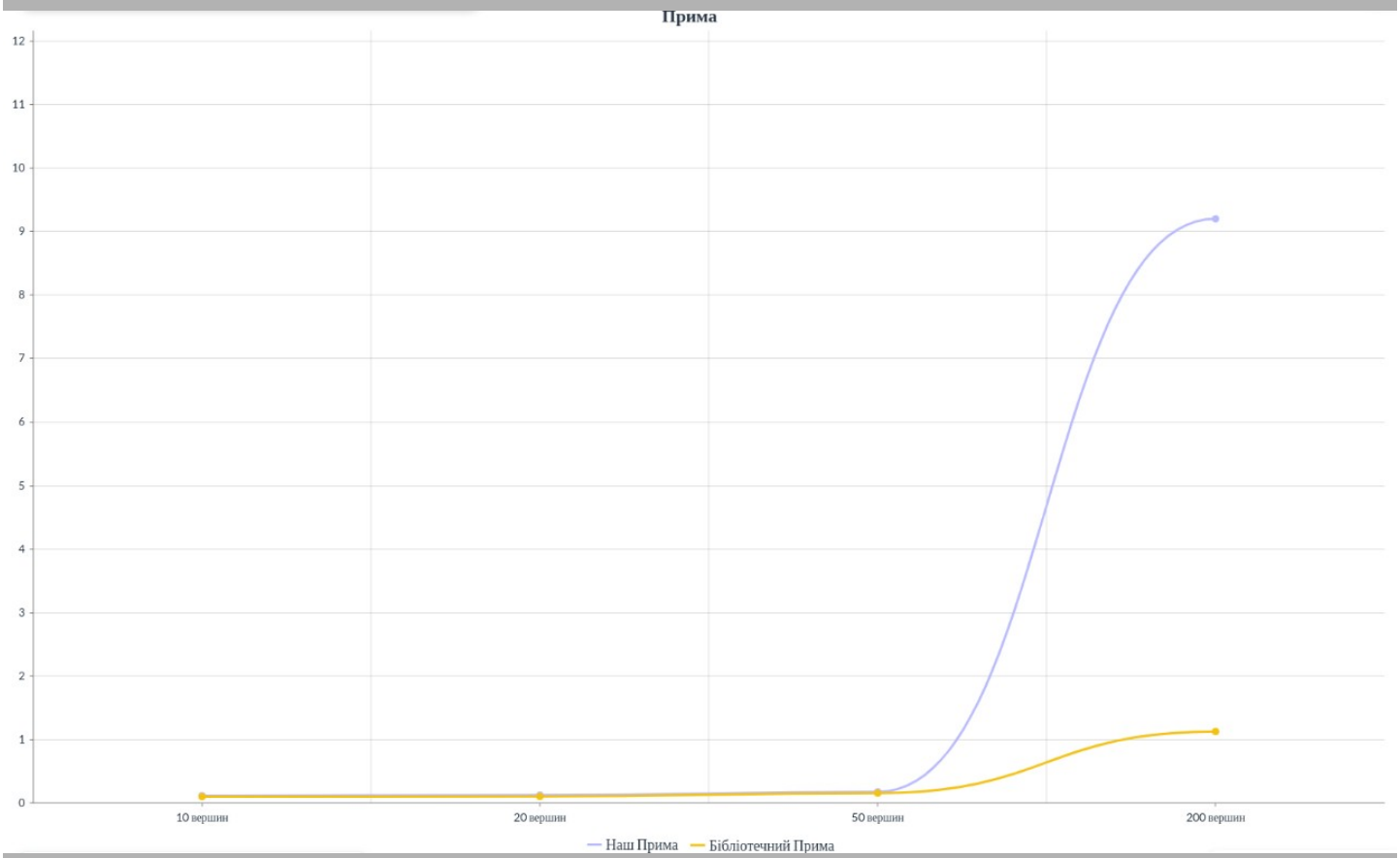
Алгоритм Краскала

```python
def kruskal(graph):
    edges = sorted(graph.edges(data=True), key=lambda x: x[2]['weight'])
    result = []
    vert = set()
    max_vert = max_v(edges)
    for edge in edges:
        if not has_cycle(result+[edge]):
            result.append((edge[0],edge[1]))
            vert.add(edge[0])
            vert.add(edge[1])
        if len(vert) == max_vert + 3:
            break
    return result
```

Алгоритм Краскала був пеалізований за допомогою тих самих допоміжних функцій що і Прима, тож я не буду дублювати код. Основна функція зображена вище. Коротко про логіку – усі ребра сортуються від найлегшого до найважчого. Після цього ми проходимося по них цмклом і додаємо кожне ребро яке не утворю цикл при врахуванні минулих ребр. Як тільки ми пройщли усі вершини цикл зупиняється.
Усі відповіді нашого алгоритму співпадають з бібліотечним.
Швидкість роботи функцій:
Наша функція:
10 вершин – 0.09с
20 вершин – 0.11с
50 вершин – 0.19
200 вершин – 1.51с
Вбудована функція:
10 вершин – 0.1с
20 вершин – 0.11с
50 вершин – 0.16с
200 вершин – 1.12с

Графіки часозатроності алгоритмів.





Якщо підводити підсумок, то обидва реалізовані нами алгоритми працюють гірше ніж бібліотечні. Якщо з Прима усе ясно – у нашому варіанті цикл в циклі, що і так тормозить алгоритм – то причини чуть-чуть повільнішою роботи Краскала мені уже не відомі. Що у нашої версії, що у бібліотечної перфоманс амйже одинаковий і середнє значеннння часу завжди коливається, тобто швидше за все вони одинаково майже однаково ефективні. Прима приблизно до 80-90 вершин працює одинаково. Якщо порівнювати між собою Прима та Краскала, то через кращу імплементації виграє Краскал, та і загалом сама логіка цього алгоритму простіша.

Алгоритм Флойда Уоршала

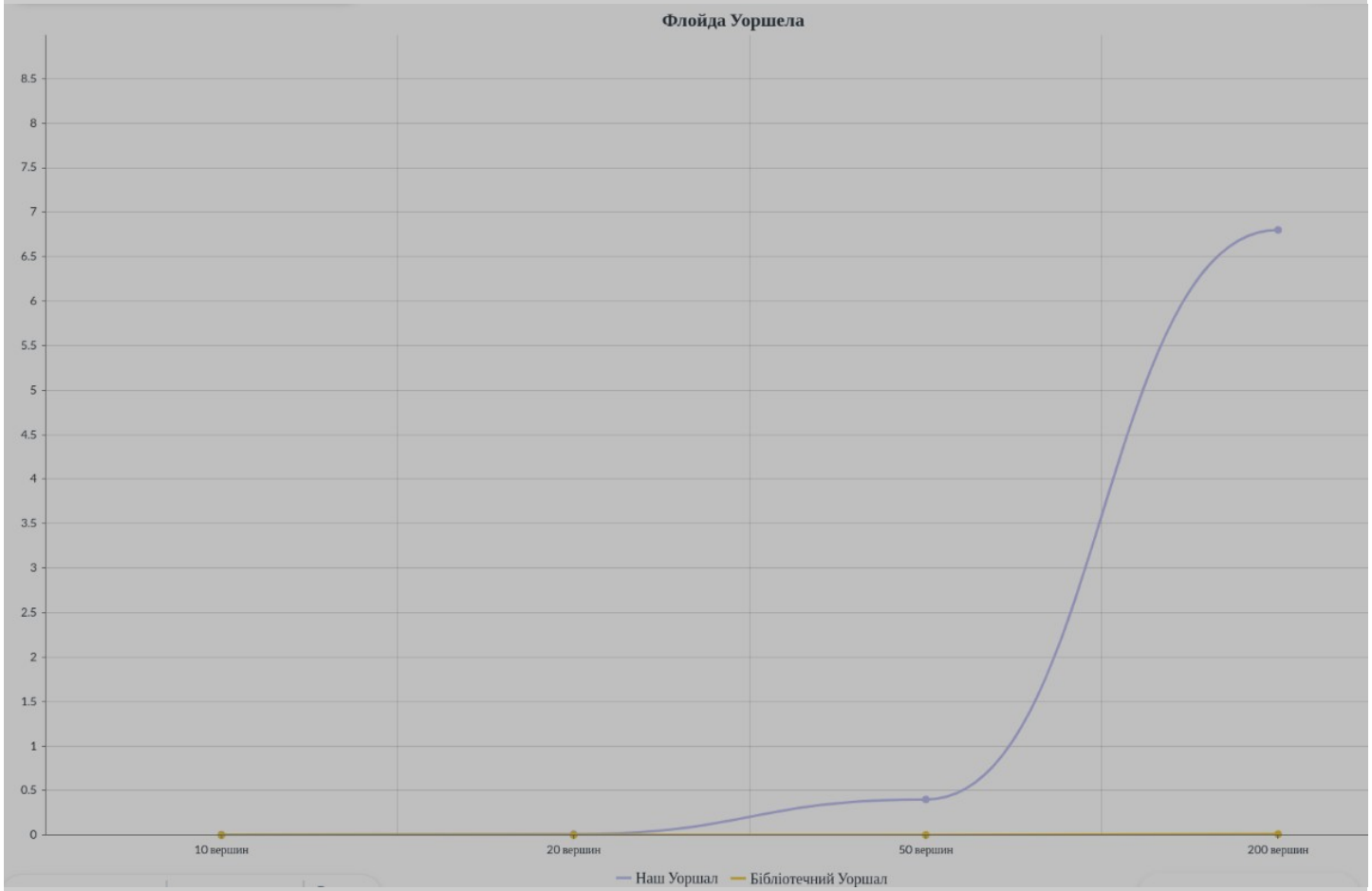
```python
def floyd_warshall(G):
    """
    Floyd-Warshall algorithm
    """
    dist = {node: {node: float('infinity') for node in G.nodes} for node in G.nodes}
    omegas = {node: {node: None for node in G.nodes} for node in G.nodes}

    for nd in G.nodes:
        dist[nd][nd] = 0
        omegas[nd][nd] = nd

    for u, v, w in G.edges(data=True):
        dist[u][v] = w['weight']
        omegas[u][v] = u

    for k in G.nodes:
        for i in G.nodes:
            for j in G.nodes:
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
                    omegas[i][j] = omegas[k][j]

    return dist, omegas
```

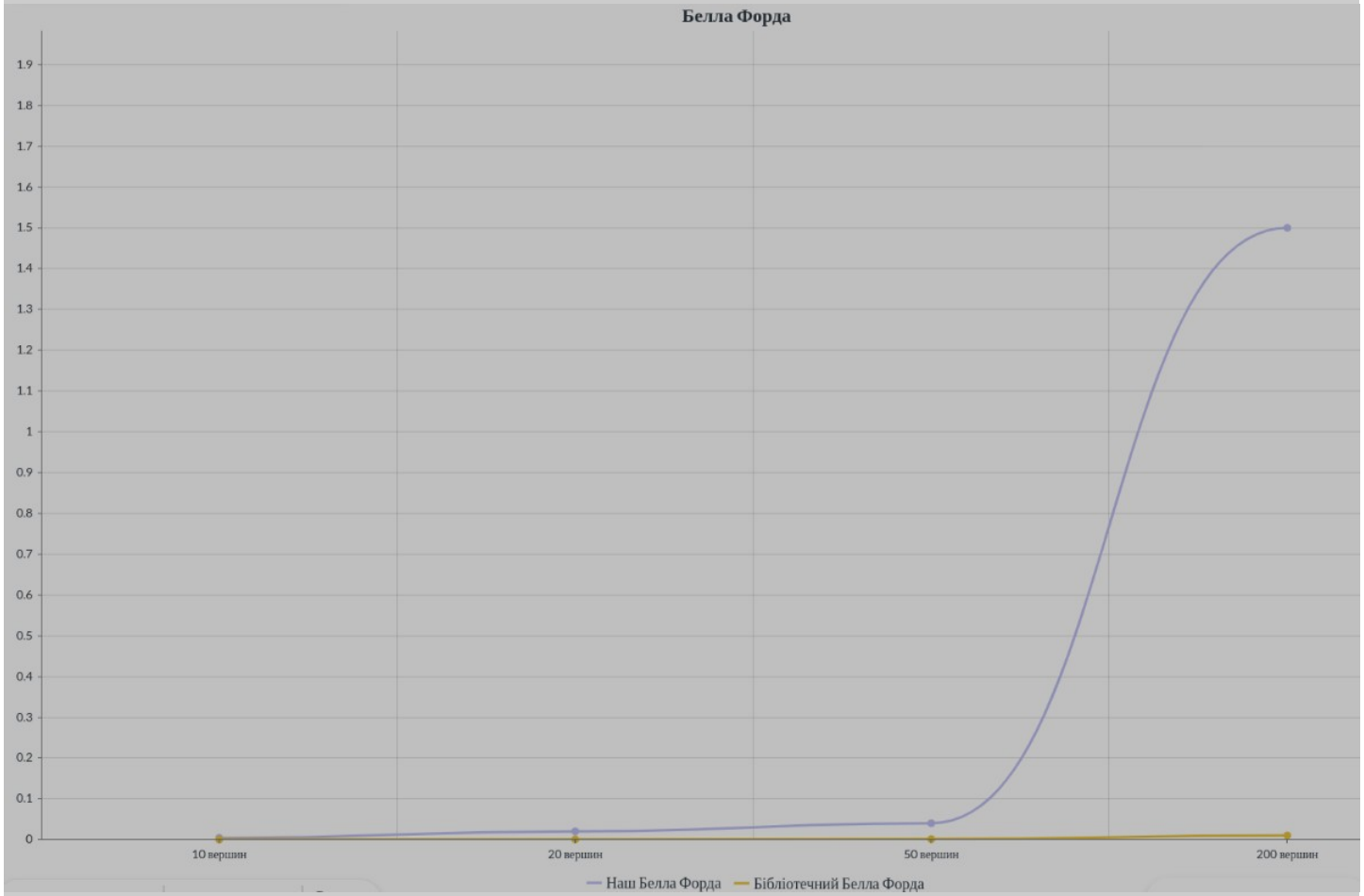
Логіка у алгоритму така сама як і у звичайного, по суті вони нічим не відрізняються.

```python
def bellman_ford(G, start):
    """
    Bellman-Ford algorithm
    """
    dist = {node: float('infinity') for node in G.nodes}
    dist[start] = 0

    for _ in range(len(G.nodes)-1):
        for u, v, w in G.edges(data=True):
            if dist[u] + w['weight'] < dist[v]:
                dist[v] = dist[u] + w['weight']

    for u, v, w in G.edges(data=True):
        if dist[u] + w['weight'] < dist[v]:
            raise ValueError("Graph contains a negative-weight cycle")

    return dist
```

Аналогічно як і з Уоршалом, ніякої особливої різниці в порівнянні зі звичайним алгоритмом Уоршала.

Експерименти





Якщо підводити підсумки, то знову ж таки, обидва наші алгоритми відстають по швидкості від запропонованих у біліотеці. Якщо порівнювати між собою два наштх алгоритми, то перемагає Белла Форда, адже він значно краще поводить себе на великих графах.

Decision Tree Classifier

```python
class Node:

    def __init__(self, X, y, gini):
        self.X = X
        self.y = y
        self.gini = gini
        self.feature_index = 0
        self.threshold = 0
        self.left = None
        self.right = None



class MyDecisionTreeClassifier:
    """ A decision tree classifier. """

    def __init__(self: object, max_depth: int = 3):
        '''
        max_depth: maximum depth of the tree
        '''
        self.max_depth = max_depth

    def gini(self: object, classes: list[list]) -> list:
        '''
        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).
        '''

        gini = 1.0 - sum((group / sum(classes))**2 for group in classes)

        return gini
```

```python
def build_tree(self: object, X: list[list], y: list[int], depth: int = 0) -> Node:
    '''
    Build a decision tree.
    '''

    # create a root node

    # recursively split until max depth is not exceeded

    # if the node is pure, return it
    if len(set(y)) == 1:
        return Node(X, y, 0)

    # if max depth is reached, return the most common class
    if depth >= self.max_depth:
        return Node(X, y, self.gini([y.count(x) for x in set(y)]))

    # split the data
    self.split_data(X, y)
    print("Left: " + str(self.left) + ", Right: " + str(self.right))
    left_y = [self.left[key] for key in self.left]
    print("Left y: " + str(left_y))
    right_y = [self.right[key] for key in self.right]
    print("Right y: " + str(right_y))

    # create the node
    node = Node(X, y, self.gini([y.count(x) for x in set(y)]))

    for i in range(1, len(X)):
        node.left = self.build_tree(X[:i], y[:i], depth + 1)
        node.right = self.build_tree(X[i:], y[i:], depth + 1)

    # recursively build the tree
    # print("1")
    # node.left = self.build_tree(list(left_X.keys()), left_y, depth + 1)
    # print("2")
    # node.right = self.build_tree(list(right_X.keys()), right_y, depth + 1)
    # print("3")

    return node



def fit(self: object, X: list[list], y: list[int]) -> None:
    """
    Fit the model to the data passed as arguments.
    """

    # basically wrapper for build tree / train
    self.feature_count = len(X[0])
    self.root = self.build_tree(X, y)
```

```python
def split_data(self: object, X: list[list], y: list[int]) -> tuple:
    '''
    Split a dataset based on an attribute and an attribute value.
    '''

    # 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
    best_gini_left = 1
    best_gini_right = 1
    best_feature = 0
    best_threshold = 0
    best_left = {}
    best_right = {}

    dct = {tuple(el_x): el_y for el_x, el_y in zip(X, y)}


    feature = 0
    print("X: " + str(X))
    for threshold in X[0]:
        # feature = self.X_train[0].index(threshold)
        # print("Feature: " + str(feature))
        all_val_dct = {el_x[feature]: el_y for el_x, el_y in zip(dct.keys(), dct.values())}
        all_val_dct = dict(sorted(all_val_dct.items(), key=lambda item: item[0]))

        # print("Row by feature: " + str(all_val_dct))
        for i in range(1, len(all_val_dct)):
            # feature = i
            left = dict(list(all_val_dct.items())[:i])
            right = dict(list(all_val_dct.items())[i:])
            print("Left: " + str(left), ", Right: " + str(right))
            gini_left = self.gini([list(left.values()).count(x) for x in set(left.values())])
            gini_right = self.gini([list(right.values()).count(x) for x in set(right.values())])
            # print("Gini left: " + str(gini_left), ", Gini right: " + str(gini_right))

            if gini_left + gini_right <= best_gini_left + best_gini_right:
                best_gini_left = gini_left
                best_gini_right = gini_right
                best_feature = feature
                best_threshold = threshold
                best_left = left
                best_right = right

        feature += 1

    # print("Best feature: " + str(best_feature), ", Best threshold: " + str(best_threshold))

    self.gini = best_gini_left + best_gini_right
    self.feature_index = best_feature
    self.threshold = best_threshold
    self.left = best_left
    self.right = best_right
```

```
    def predict(self: object, X_test: list[list]) -> list[int]:
        """
        Predict the class for the data passed as arguments.
        """

        # 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

        predictions = []
        for sample in X_test:
            node = self.root
            while node.left:
                if sample[node.feature_index] < node.threshold:
                    node = node.left
                else:
                    node = node.right
            predictions.append(node.y)

        return predictions

    def evaluate(self: object, X_test: list[list], y_test: list[int]) -> float:
        """
        Evaluate the model on the test data.
        """

        # return accuracy

        predictions = self.predict(X_test)
        return sum(predictions == y_test) / len(y_test)


# test the model
X = [[1, 7], [2, 3], [3, 11], [4, 0.6], [5, 10]]
y = [0, 1, 1, 0, 1]

trial = MyDecisionTreeClassifier()
assert trial.gini([3, 8, 5, 7, 8, 2]) == 0.8025711662075298
# trial.fit(X, y)
# trial.evaluate(X_test, y_test)
# print(trial.split_data(X, y))

# trial.X_train = [[1, 7, 3], [2, 3, 5], [3, 11, 7], [4, 0.6, 9], [5, 10, 11]]
# trial.y_train = [2, 1, 2, 0, 2]
# trial.split_data()

trial.build_tree(X, y)

# trial.predict(X_test)
```

А ось і Tree Decision Clasifier. Мало що можна додати, усе розписано у коментарях та документації до функцій. Загалом цікава лабораторна була, нам сподобалося :)