КУРСОВ ПРОЕКТ

по дисциплината "ДИСКРЕТНИ СТРУКТУРИ"

за студентите от:

факултет: ФКСТ

специалност: КСИ

Име	**Фак. номер**	**Група**	**Семестър**	**Курс**
Боян Зарев	123222004	42	VI	III
Светлин Иванов	121222088	42	VI	III
Ивайло Николов	121222009	42	VI	III

Тема: Изоморфизъм в графи

Съдържание на обяснителната записка:

Дата на задаване на проекта: 24.03.2025 г.

Срок за предаване на проекта: 01.05.2025 г.

Проекта приел:
проф. дн инж. Валери Младенов

/……../

Проекта разработили:
Боян Зарев /……/
Светлин Иванов /……/
Ивайло Николов /……/

Оценка:

1. Задание

Изоморфизмът на графи е фундаментален проблем в теорията на графите и има широки приложения в компютърните науки, криптографията, химията (моделиране на молекули) и други области. Основната цел на курсовия проект е да се изследва проблемът и да се опишат някои методи за неговото решаване както и да се направи софтуерна реализация на тези алгоритми.

2. Формулиране на проблема

Графите са фундаментални обекти в дискретната математика и намират широко приложение в компютърните науки, инженерството, социалните мрежи, криптографията и др. Ключово свойство при работа с графи е тяхната изоморфност. Два графа се наричат изоморфни, ако съществува еднозначно съответствие между техните върхове, което запазва връзката между тях. Ако върховете на единия граф се преномерират, така че той да съвпадне с другия, то те са изоморфни. На пръв поглед задачата да се намери дали графите са изоморфни изглежда много лесна, но проверката може да се усложни и е много предизвикателна ако графът е с голям размер. В този случай решаването на проблема изисква добър подход за търсене и проверка, както и голяма изчислителна мощност.

Изоморфността на графи има съществено значение в различни области. Например в компютърните науки и теорията на базите данни намира приложение при оптимизацията на заявки и сравняването на различни структури от данни. В социалните мрежи и анализите на комуникационни системи се използва за идентифициране на сходни модели на взаимодействие.

Въпреки практическата важност на този проблем, не съществува още универсален алгоритъм, който да решава задачата за всички възможни графи.

Граф $G = (V, E)$ се нарича математическа структура, състояща се от две отделни множества: непразно множество от върхове V и множество E от ненаредени двойки от различни елементи, съдържащи се във V . Тези двойки при неориентиран граф се наричат ребра, а при ориентирани графи се наричат дъги. Всяко ребро (дъга) $e_{ij} \in E$ е множество $e_{ij} = \{V_i, V_j\}$, където $V_i, V_j \in V$. Два върха V_i, V_j в не ориентиран граф се наричат съседни, ако $e_{ij} = \{V_i, V_j\}$ е ребро от G. Ако в даден неориентиран граф се съдържа реброто $e_{ij} = \{V_i, V_j\}$, свързващо върховете V_i и V_j , то се казва, че e_{ij} е инцидентно с V_i и V_j . [1]

Графовият изоморфизъм е биекция между множества от върхове на двата графа, която запазва отношението на съседство. С други думи, два графа G_1 и G_2 са изоморфни, ако съществува еднозначно съответствие между техните върхове, такова че два върха са съседни в G_1 тогава и само тогава, когато съответните им върхове са съседни в G_2 .[2]

$G_1 =< V_1, E_1 >$
$G_2 =< V_2, E_2 >$

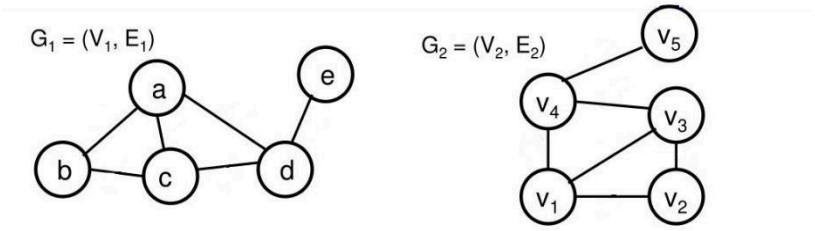
$G_1 \cong G_2 \leftrightarrow f: V_1 \rightarrow V_2$, където

(1) f е биекция

(2) $\forall a, b \in V_1: \{a, b\} \in E_1 \leftrightarrow \{f(a), f(b)\} \in E_2$

Теоретиците на графите се интересуват предимно от свойства на графите, които не се променят при преномериране на върховете. Отбелязваме, че ако $G_1 \cong G_2$, тогава много свойства на G_1 и G_2 трябва да бъдат еднакви. Например, броят на върховете и ребрата в двата графа трябва да е идентичен. G_1 е свързан тогава и само тогава, когато G_2 е свързан. Броят на върховете със степен 0, 1, 2 и т.н. трябва да е идентичен. Никое от изброените тук свойства не се променя при преномериране на върховете. Обратно, ако два графа G_1 и G_2 се различават по което и да е от тези свойства, тогава можем да бъдем сигурни, че G_1 и G_2 не са изоморфни.[3]

Пример 1. Нека G_1 и G_2 са графи, където $G_1 =< V_1, E_1 >$ и $G_2 =< V_2, E_2 >$.
$V_1 = \{a, b, c, d, e\}$, $E_1 = \{\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{c, d\}, \{d, e\}\}$,
$V_2 = \{v_1, v_2, v_3, v_4, v_5\}$, $E_2 = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_5\}\}$.



(фиг. 1) Графите G_1 и G_2 от пример 1.

Двата графа са изоморфни, защото може да се изгради еднозначна кореспонденция между върховете им, при която връзките между съответните върхове се запазват.

$f(a) = v_1$ $\quad\quad$ $\{a, b\} \rightarrow \{v_1, v_2\}$
$f(b) = v_2$ $\quad\quad$ $\{a, c\} \rightarrow \{v_1, v_3\}$
$f(c) = v_3$ $\quad\quad$ $\{a, d\} \rightarrow \{v_1, v_4\}$
$f(d) = v_4$ $\quad\quad$ $\{b, c\} \rightarrow \{v_2, v_3\}$
$f(e) = v_5$ $\quad\quad$ $\{c, d\} \rightarrow \{v_3, v_4\}$
$\quad\quad\quad\quad\quad\quad$ $\{d, e\} \rightarrow \{v_4, v_5\}$

3. Обзор на съществуващи до момента метода за намиране на изоморфизъм

Въпреки привидната си простота, проблемът с графовия изоморфизъм все още няма универсално решение, което да работи ефективно за всички типове графи. Съществуват обаче множество подходи, които показват добри резултати в различни конкретни случаи. Сред най-известните са: Naughty, Ullman, Brute Force, Color Refinement (известен още като тестът на Weisfeiler-Lehman или 1-WL), канонично етикетиране с помощта на BFS или DFS, VF2, както и техники, базирани на машинно обучение.

Алгоритъмът Naughty (No Automorphism, Yes?) е един от най-ефективните практически инструменти за определяне на изоморфизъм на графи. Създаден е от Brendan McKay и използва комбинация от пермутационни техники, разбиване на върхове и групиране, за да сведе проблема до сравнение на канонични форми. Този алгоритъм използва канонична маркировка на графите, която представлява подреждането на възлите по лексикографски редослед.

Алгоритъмът на Ullman използва семантично търсене с backtracking и филтриране на матрици на съседство.

Brute Force е наивен алгоритъм, който проверява всички възможни пермутации на върховете на един граф, за да установи дали съществува съпоставяне, запазващо структурата на ребрата. Той не дава добри резултати, когато графите имат голям брой върхове.

В този курсов проект ще бъдат детайлно разгледани и реализирани софтуерни имплементации на следните алгоритми за проверка на графов изоморфизъм: Color Refinement, Adjacency Matrix Comparison и VF2.

4. Color Refinement (Weisfeiler-Lehman Test, 1-WL) - Боян Зарев, ф.н. 123222004

Горе графът е дефиниран като математическа структура, състояща се от две отделни множества: непразно множество от върхове V и множество E от ненаредени двойки от различни елементи от V. За реализация на този алгоритъм, освен тези две множества графът ще се дефинира и с множество от характеристики на върховете. Това множество обикновено са цели числа или цветове.

Алгоритъмът разглежда неориентирани графи $G = (V, E, \mathrm{X_V})$, където $V = 1, \ldots, n := |n|$ са върховете; $E \subseteq V\ x\ V$ е множество от ребра, изпълняващи условието $(u, v) \in E$ тогава и само тогава, когато $(v, u) \in E$; и $\mathrm{X_V}$ е множество от характеристики на върховете: За всяко $v \in V\ \mathrm{X_V} \in \mathrm{R^d}$. Съседните върхове на върха v се дефинират с $N_G(v) = \{w :\ (v, w) \in E\}$.

Тестът на Weisfeiler-Lehman поддържа състояние (или цвят) за всеки връх. Той предефинира състоянията на възлите като събира и обобщава информацията на техните им съседи. За да изчисли и предефинира новото състояние на връх, WL използва инективна хеш функция дефинирана в различни обекти по модул на класовете на еквивалентност. Това означава, че обектите се групират в класове на еквивалентност и хеш функцията отчита тази

класификация. Ако два обекта принадлежат към един и същи клас на еквивалентност, те ще получат един и същи хеш. С други думи, $\forall v, w \in V \, hash(X_v) = hash(X_w) \leftrightarrow X_v = X_w$.[4]

Класическият 1-WL тест поддържа състояние на всеки възел, което се модифицира с помощта на състоянията на техните им съседи. Изходът е структура, която представлява състояние на графа кореспондираща на класификацията на всички върхове. Казваме че WL е успешно направил разлика на два неизоморфни графа G и G' ако $WL(G) \neq WL(G')$.[4]

За съжаление 1-WL не различава всички случаи на неизоморфни графи. Решението за това е дефиниране на множеството на характеристики X_v, така да то има повече дименсии, т.е. всеки връх ще има повече от една характеристика.

4.1. Софтуерна реализация на Color Refinement (Weisfeiler-Lehman Test, 1-WL)

В софтуерната реализация на алгоритъма Color Refinement използвани са модерни програмни техники и библиотеки за ефективна и ясна имплементация. Кодът е разработен на езика Python, тъй като той предоставя богат набор от инструменти за работа с графи. За представянето и обработката на графите използвана е библиотеката NetworkX. За графично представяне на графите използвана е библиотеката Matplotlib.

Характеристиката на върховете са представени като цели числа, получени от хеш функция. Хеш функцията в тази реализация е Hash Map (dictionary в Python), която при вече срещано състояние на връх дава кореспондиращата му стойност, докато при несрещано състояние добавя това състояние като нов елемент в тази структура свързано с ново цяло число. Под състояние на връх се подразбира неговата стойност на характеристиката от предишната итерация на алгоритъма и стойностите на характеристиката на техните му съседи. Първоначално, стойността на характеристиката на всеки връх е броят на неговите съседи. Във всяка итерация върховете получават нови стойности на характеристика. Състоянията са реализирани като tuple, чието съдържание е (current node label, neighbour nodes labels).

Алгоритъмът първо проверява дали броят на върховете и ребрата на двата графа си съответстват. Ако не, то тогава алгоритъмът веднага връща False. На края на всяка итерация се проверява означенията на върховете. Класификацията на върховете се сортира по нарастващ ред и на двата графа. Ако масивите са същи алгоритъмът връща True. В случай да не са, той продължава. Максималният брой на итерация в тази софтуерна реализация е 10.

За визуализация на стъпките, стойностите на характеристиката са свързани със цветовете от библиотеката Matplotlib. Характеристиката е визуализирана като стълбовидна диаграма от честотата на класове.

Функцията refine_labels като изход дава нови цветове на върховете (новите стойности на характеристиката), изпълнявайки хеш функцията. color е кортежа, който всъщност в себе има стойността на характеристиката на текущия връх, както и на техните му съседи по нарастващ редослед. В тялото на функцията се прави проверка дали този color съществува като ключ във речника color_labels. Ако не съществува, на съответното състояние се дава ново означение, като към брояча label_value се добавя 1.

```
def refine_labels(graph, node_labels):
    '''
        structure_labels is dict that maps node to its neighbourhood
structure.
     color_label is a dict that maps neighbourhood structure to its unique
hash value.

       if 2 nodes have the same previous label, and same neighbourhood
structure, they will have the same hash value.
    '''
    new_labels = {}

    color_labels = {}
    structure_labels = {}
    label_value = 0
    for node in graph.nodes:
     neighbors_coloring = sorted([node_labels[neighbor] for neighbor in
graph.neighbors(node)])
        color = (node_labels[node],) + tuple(neighbors_coloring) # adds
labels of neighbours to the current label of the given node and form it
like a tuple
          # the purpose of this is to create unique labels for the nodes,
having in consideration structure of neighbourhood
        structure_labels[node] = color

         if color not in color_labels:  # creating new "hash" value for new
label
            color_labels[color] = label_value
            label_value += 1

    for node in graph.nodes:
        new_labels[node] = color_labels[structure_labels[node]]

    return new_labels
```

Функцията wl_test като изход връща True или False в зависимост от това дали съществува изоморфизъм при подадените 2 графа като параметри. Тази функция освен тези два позиционни аргумента има и 3 keyword аргумента:

1. max_iteration - максимален брой на итерации, по подразбиране е 10;
2. visualise - може да има стойности True или False и той определя дали за всяка стъпка ще се прави визуализация на честотата на цветовете на двата графа. Когато е True, генерират се 2 графа, чиито върхове са оцветени със съответните им класове и две стълбовидни диаграми, на които се намират честотите на класовете. За да се продължи с изпълнението на алгоритъма, когато visualise е True, необходимо е да се затвори новосъздадения прозорец;
3. node_size - големина на върховете, по подразбиране е 300.

Функцията wl_test първоначално проверява дали броят на върховете и ребрата съответстват на двата графа. Ако не, то тогава означава че изоморфизмът не е наличен, и функцията връща False. Като първоначални стойности на характеристиката на върховете функцията задава броя на техните съседи. След това тези означения минават през функцията refine_labels. Получените резултати от функцията refine_labels на двата графа се проверяват дали са еднакви. Ако са еднакви, то тогава изоморфизмът е наличен и функцията wl_test връща True, а ако не алгоритъмът продължава със следващата итерация.

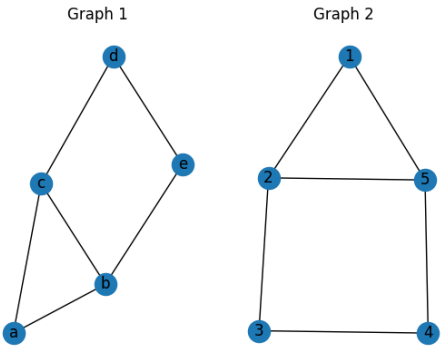
```
def wl_test(G1, G2, max_iterations=10, visualize=False,
node_size=default_node_size):
    '''
    Computes 1-WL test, return True if graphs are isomorphic, False when
they are not.
    G1, G2 are graph parameters.
    max_iterations is the default number of iterations that algorithm will
go through.

    visualize is a parameter that specifies if each iteration will be
visualized with matplotlib.
    When True it will visualize the given phase of the 1-WL test and
algorithm will NOT continue UNTIL user close the window
    in wich 2 graphs have been visualized.

    The initial labels of nodes are their degrees.
    '''
    if len(G1.nodes) != len(G2.nodes) or len(G1.edges) != len(G2.edges):
        return False

    g1_labels = {node : G1.degree(node) for node in G1.nodes}
    g2_labels = {node : G2.degree(node) for node in G2.nodes}

    if visualize:
        draw_color_2_graph(G1, g1_labels, G2, g2_labels,
node_size=node_size)

    for i in range(max_iterations):
        g1_labels = refine_labels(G1, g1_labels)
        g2_labels = refine_labels(G2, g2_labels)


        if visualize:
            draw_color_2_graph(G1, g1_labels, G2, g2_labels,
node_size=node_size)


        if sorted(g1_labels.values()) == sorted(g2_labels.values()):
            return True
       return False
```

Функцията draw_color_2_graph прави визуализация на двата графа. Върховете оцветява с цвета от масива css_colors, кореспондиращ с техните им означения на класа, към който принадлежат. Под графите се намират две стълбовидни диаграми, показващи честотите на класовете във графите. Като аргументи взима двата графа и означенията на възлите им.

```
def draw_color_2_graph(graph1, graph1_labels, graph2, graph2_labels,
node_size=default_node_size):
    colors_graph1 = []
    colors_graph2 = []

    for node in graph1.nodes:
        color = css_colors[graph1_labels[node] % len(css_colors)]
        colors_graph1.append(color)

    for node in graph2.nodes:
```

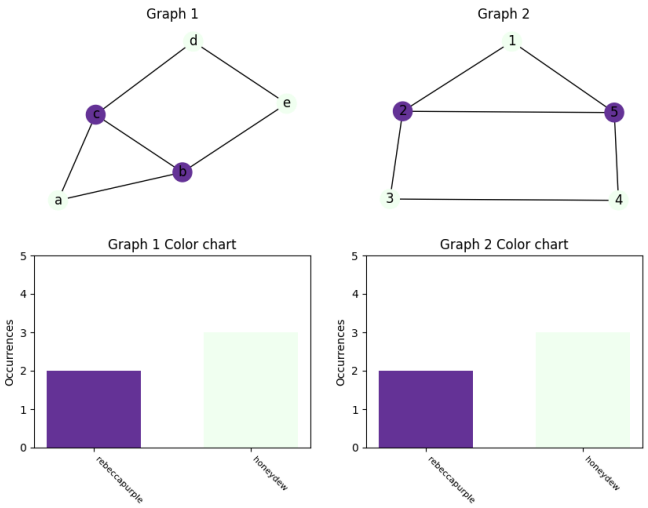
```
        color = css_colors[graph2_labels[node] % len(css_colors)]
        colors_graph2.append(color)

    pos_graph1 = nx.spring_layout(graph1, seed=layout_seed1)
    pos_graph2 = nx.spring_layout(graph2, seed=layout_seed2)

    plt.figure(figsize=(10, 7))
    subax1 = plt.subplot(221)
    nx.draw(graph1, pos=pos_graph1, with_labels=True,
node_color=colors_graph1, node_size=node_size)
    plt.title("Graph 1")

    subax2 = plt.subplot(222)
    nx.draw(graph2, pos=pos_graph2, with_labels=True,
node_color=colors_graph2, node_size=node_size)
    plt.title("Graph 2")


    color1_occurrences = {}
    for c in colors_graph1:
        if c not in color1_occurrences:
            color1_occurrences[c] = 1
        else:
            color1_occurrences[c] += 1
    color2_occurrences = {}
    for c in colors_graph2:
        if c not in color2_occurrences:
            color2_occurrences[c] = 1
        else:
            color2_occurrences[c] += 1

    colors1_bar = sorted(color1_occurrences.keys(), key=lambda x:
color1_occurrences[x])
    colors2_bar = sorted(color2_occurrences.keys(), key=lambda x:
color2_occurrences[x])

    subax3 = plt.subplot(223)
    plt.bar(colors1_bar, [color1_occurrences[c] for c in colors1_bar],
color=colors1_bar, width=0.6)
    plt.xticks(rotation=-45, fontsize=8, ha='left')
    plt.ylabel("Occurrences")
    plt.ylim(0, len(graph1))
    plt.title("Graph 1 Color chart")
    plt.margins(x=0.05)

    subax4 = plt.subplot(224)
    plt.bar(colors2_bar, [color2_occurrences[c] for c in colors2_bar],
color=colors2_bar, width=0.6)
    plt.xticks(rotation=-45, fontsize=8, ha='left')
    plt.ylabel("Occurrences")
    plt.ylim(0, len(graph2))
    plt.title("Graph 2 Color chart")
    plt.margins(x=0.05)

   plt.show()
```

Пример 2. Нека G_1 и G_2 са графи, където $G_1 =< V_1, E_1 >$ и $G_2 =< V_2, E_2 >$. $V_1 = \{a, b, c, d, e\}$, $E_1 = \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, e\}, \{c, d\}, \{d, e\}\}$, $V_2 = \{1,2,3,4,5\}$, $E_2 = \{\{1,2\}, \{1,5\}, \{2,3\}, \{2,5\}, \{3,4\}, \{4,5\}\}$.



(фиг. 2) Графи от пример 2.

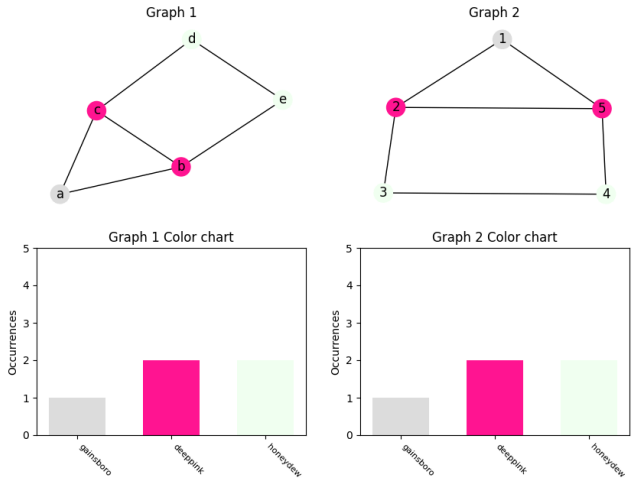
Преди алгоритъмът да почне да изчислява кой върх какво означение на класа ще получи, той задава стартови стойните на характеристиката. Тези стойности са степента на върха. На фигура 3. са илюстрирани върховете, които са оцветени съответно с цвета на техния начален клас. Класът с 2 съседа е оцветен в honeydew, докато класът с 3 съседа е оцветен с rebeccapurple.



(фиг. 3) Илюстрация на 1. стъпка в пример 2.

В следващата стъпка в граф 1 върховете c и d ще принадлежат в един и същи нов клас защото са с един и същи цвят и имат 2 съседа с honeydew цвят и 1 съсед с rebeccapurple. Съответно и върховете d и e ще принадлежат към един и същи клас защото имат по един съсед от двата предишни класа. Върха a няма да принадлежи в нито един от двата новосъздадени класа. Въпреки че броят на неговите съседи съответства на броя на съседите

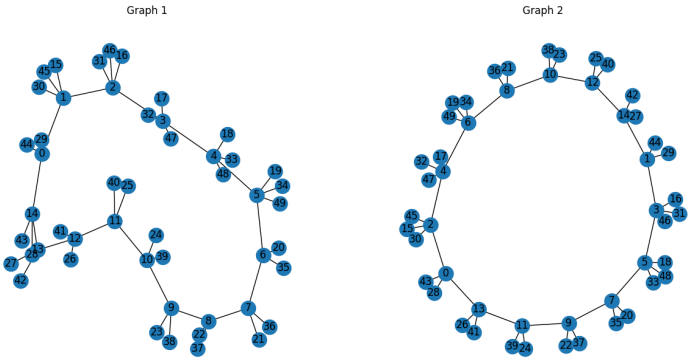
на d и e, върха a ще си създаде нов клас, защото има два съседа от rebeccapurple, което не е случай при върховете d и e. По същия принцип ще се класифицират и върховете в граф 2. Тази стъпка е илюстрирана на фиг. 4.



(фиг. 4) Илюстрация на 2. стъпка в пример 2.

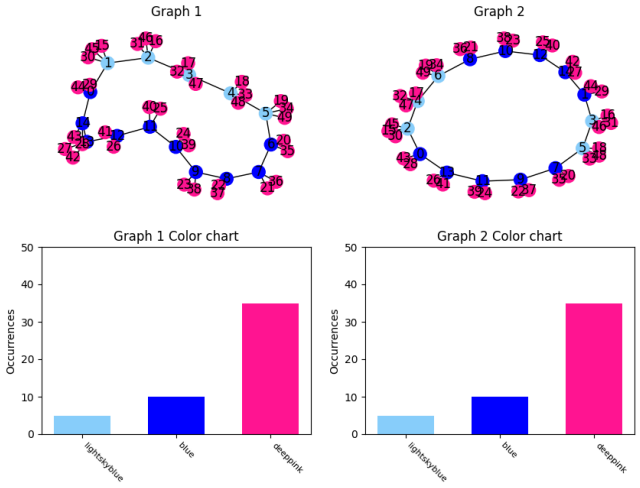
Върховете са получили нови означения на класовете в които са класифицирани. Функцията wl_test след всяка стъпка (освен инициализацията) проверява дали честотата на върховете в съответните класове на двата графа си съвпадат, т.е. проверява дали цветните бар-диаграми на двата графа си съвпадат. В примера се вижда, че те си съвпадат, което означава че съществува изоморфизъм. Функцията като изход връща True.

Пример 3. Нека G_1 и G_2 са графи, където $G_1 =< V_1, E_1 >$ и $G_2 =< V_2, E_2 >$.
$V_1 = \{k \mid k \in N, k < 50\}$, $E_1 = \{(i, (i+1)\ mod\ 15) \mid i \in \{0,1,2,\dots,49\}\}$,
$V_2 = \{k \mid k \in N, k < 50\}$, $E_2 = \{(i, (i+2)\ mod\ 15) \mid i \in \{0,1,2,\dots,49\}\}$.



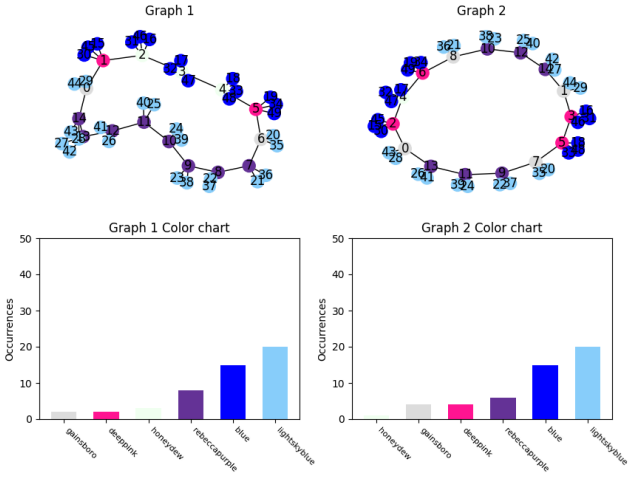
(фиг. 5) Графи от пример 3.

Първоначалната класификация е представена в фиг . 6. Върховете с по един съсед са оцветени с deeppink, с по два съседа са оцветени с blue и с по три съседа оцветени с lightskyblue.



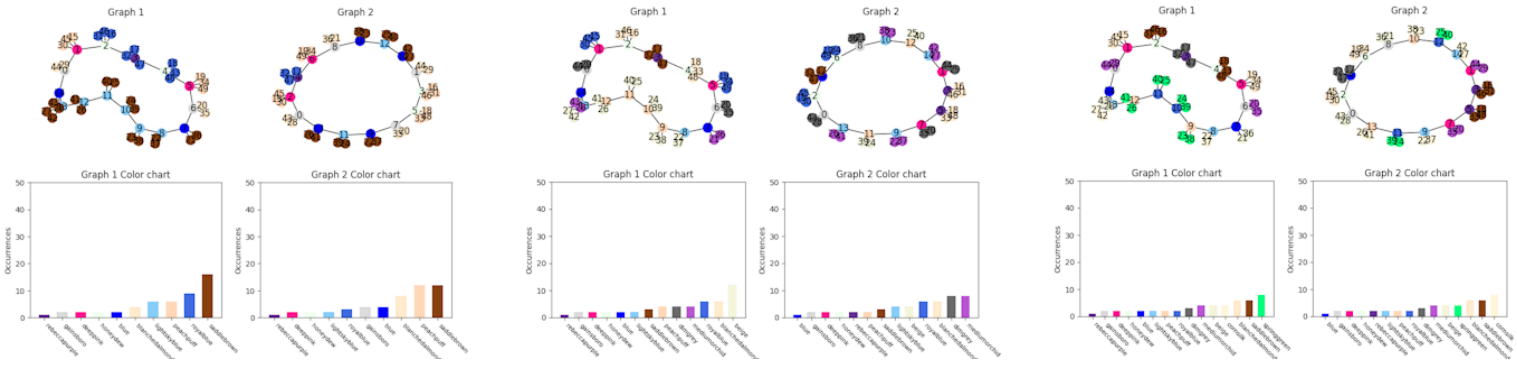
(фиг. 6) Илюстрация на 1. стъпка в пример 3.

В следващата стъпка честотите на класовете blue и lightskyblue са еднакви, но в останалите 4 класа има разместване. Тази стъпки е илюстрирана на фиг. 7.



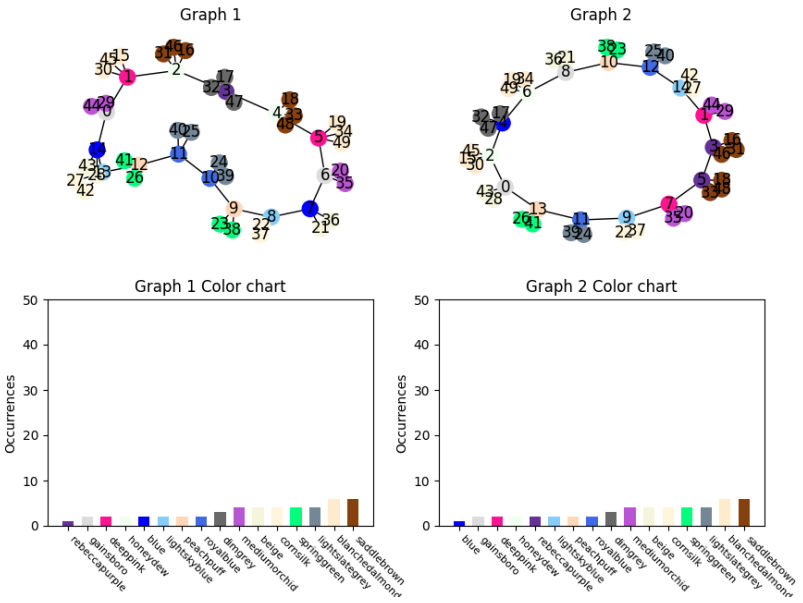
(фиг. 7) Илюстрация на 2. стъпка в пример 3.

Стъпките 3, 4, 5 са илюстрирани на фиг. 8.



(фиг. 8) Илюстрация на стъпките 3, 4, 5 в пример 3.

Стъпката 6 е последна стъпка в този тест, защото всички върхове от графовете са вече получили финалните си означения на класовете, т.е. колкото итерации да следват след тази, върховете ще бъдат класифицирани в същите класове, защото състоянията на върховете не могат повече да бъдат анализирани, така че да бъдат създадени нови класове.



(фиг. 9) Илюстрация на 6. стъпка в пример 3.

На пръв поглед, стълбовидните диаграми на двата графа изглеждат абсолютно също, но обаче съществува малка разлика: местата на класовете blue и rebeccapurple са разместени. В граф 1 в класа rebeccapurple е класифициран 1 връх, в класа blue са класифицирани 2, докато в граф 2 е обратното.

След изпълнение на броя на итерациите посочен в функцията wl_test като параметър, алгоритъмът преценява, че честотите на класовете не са същи при двата графа, и той като изход ще върне False.

5. Adjacency Matrix Comparison - Светлин Иванов, ф.н. 121222088

Adjacency Matrix Comparison е метод, който определя изоморфизма между два графа чрез директно сравнение на техните матрици на съседство. При този подход всеки граф се представя чрез квадратна матрица, в която редовете и колоните съответстват на върховете на графа, а стойностите в клетките са само 0 и 1, където 1 означава, че има ребро между два върха, а 0 че няма. Ако графът е неориентирана (т.е. всички нейни ръбове са двупосочни), матрицата на съседство е симетрична.

За прост граф с множество от върхове $U = \{u_1, ..., u_n\}$, матрицата на съседство представлява квадратна матрица с размер $n \times n$, в която елементът A_{ij} е равен на 1, ако съществува ребро от върха u_i към върха u_j, и 0, ако такова ребро няма. Понеже в простите графи не се допускат примки (ръбове, които започват и свършват в един и същи връх), всички диагонални елементи на матрицата са 0. [5]

Неориентиран граф обикновено се представя като двойка G = (V, E), където V е множеството от върхове, а E ⊂ V × V е множеството от ребра, удовлетворяващо условието $(u, v) \in E$ тогава и само тогава, когато $(v, u) \in E$. Съседите на връх v се дефинират като $N(v) = \{w:(v, w) \in E\}$. В теорията на графите казваме, че $G1 = (V1, E1)$ и $G2 = (V2, E2)$ са изоморфни, ако съществува биекция между множествата от върхове $V1$ и $V2$:

$f:V1 \to V2$

такава, че всеки два върха u и v от $G1$ са свързани в $G1$ тогава и само тогава, когато $f(u)$ и $f(v)$ са свързани в $G2$, т.е. $(u, v) \in E1$ тогава и само тогава, когато $(f(u), f(v)) \in E2$. Ако съществува изоморфизъм между два графа, те се наричат изоморфни и се означават като $G1 \simeq G2$, а f се нарича изоморфна функция между $G1$ и $G2$.

Да предположим, че графът $G1$ е изоморфен на графа $G2$. Нека означим техните върхове и матрици на съседство съответно като $V1$, $A1$ и $V2$, $A2$. Според дефиницията за графов изоморфизъм съществува биекция $f:V1 \to V2$, такава че $f(V1)=V2$. Нека върховете на $G1$ са $V1=\{1,\ldots,n\}$. Тогава върховете на $G2$ могат да бъдат изразени като $V2=\{f(1),\ldots,f(n)\}$. Дефинираме, $g(x) = f^t(i)$ където t = max $\{t \mid f^{t-1}(i) < i\}$. След това можем да преобразуваме матрицата на съседство $A1$ на $G1$ в матрицата на съседство $A2$ на $G2$ чрез следната последователност от разменни операции:

$A2 \leftarrow A2[i \leftrightarrow g(i)]$, $g(i)<i$, $i=1,\ldots,n$

От друга страна, да предположим, че съществува последователност от разменни операции $A[j,h(j)] \leftrightarrow j \le n$, която преобразува $A1$ в $A2$. Дефинираме $f:V1 \to V2$ по следния начин: $f(i) = h^t(i)$ където t = max $\{t \mid h^{t-1}(i) > i\}$. Тогава f е изоморфна функция между $G1$ и $G2$. Освен свързването на изоморфизма с разменните операции, теоремата също така предоставя ефективен алгоритъм за преобразуване на графа $V1$ в неговия изоморфен аналог $V2$, както е представено в фигурата. [6]

Algorithm 1 Transform graph G_1 to its isomorphic counterpart G_2 by transforming adjacency matrices A_1 to A_2

```
Let V1 = {1, 2, ··· , n}, V2 = {f(1), f(2), ··· , f(n)}
for i = 1 to n do
    t = f(i)
    for j = 1 to n do
        while t < i do
            t = f(t)
        end while
    end for
    g(i) = t
    A1 = A1[i, g(i)]
end for
```

(фиг. 10) Алгоритъм на трансформиране на граф в неговият изоморфен вид

5.1. Софтуерна реализация на Adjacency Matrix Comparison

Софтуерната реализация на алгоритъма за сравнение на матрици на съседство е постигната чрез компактни и ясно дефинирани функции, които изпълняват основните проверки за графов изоморфизъм. Кодът е написан на Python с използване на библиотеката NumPy за линейно-алгебрични операции и следва следната логика:

```
import numpy as np

def adjacency_matrix_comparison_test(g1, g2):
    """Check if two graphs are isomorphic by comparing adjacency
matrices"""

    adj1 = convert_graph_to_adj_matrix(g1)
    adj2 = convert_graph_to_adj_matrix(g2)

    if len(adj1) != len(adj2):
        return False

    if not basic_checks_pass(adj1, adj2):
        return False

    return True

def basic_checks_pass(adj1, adj2):
    n = len(adj1)

    deg1 = sorted(sum(row) for row in adj1)
    deg2 = sorted(sum(row) for row in adj2)
    if deg1 != deg2:
        return False

    if sum(sum(row) for row in adj1) != sum(sum(row) for row in
adj2):
        return False

    try:
        eig1 = sorted(np.linalg.eigvals(adj1))
        eig2 = sorted(np.linalg.eigvals(adj2))
        if not np.allclose(eig1, eig2, atol=1e-6):
            return False
    except:
        pass

    return True

def convert_graph_to_adj_matrix(graph):
    """Convert networkx graph to adjacency matrix"""
    nodes = sorted(graph.nodes())
    n = len(nodes)
    adj = [[0]*n for _ in range(n)]

    for i, u in enumerate(nodes):
        for j, v in enumerate(nodes):
            if graph.has_edge(u, v):
                adj[i][j] = 1
```

```
    return adj
```

Следната функция работи като проверява дали два графа са изоморфни чрез сравнение на техните матрици на съседство. Тя работи като: Преобразува графите в матриците на съседство чрез функцията convert_graph_to_adj_matrix; Проверява за броят върхове и ако те не съответстват един на друг връща False; Извършва основни проверки чрез функцията basic_check_pass (сравнява сортираните степени на върховете, проверява дали общият брой ребра съвпада и сравнява стойностите на матриците). При успешно преминаване на всички проверки връща True в противен случай False.

```
def adjacency_matrix_comparison_test(g1, g2):
    """Check if two graphs are isomorphic by comparing adjacency
matrices"""

    adj1 = convert_graph_to_adj_matrix(g1)
    adj2 = convert_graph_to_adj_matrix(g2)

    if len(adj1) != len(adj2):
        return False

    if not basic_checks_pass(adj1, adj2):
        return False

    return True
```

Функцията basic_checks_pass извършва три основни проверки за предварителна оценка на изоморфизъм между два графа, представени чрез техните матрици на съседство adj1 и adj2.

```
def basic_checks_pass(adj1, adj2):
    n = len(adj1)

    deg1 = sorted(sum(row) for row in adj1)
    deg2 = sorted(sum(row) for row in adj2)
    if deg1 != deg2:
        return False

    if sum(sum(row) for row in adj1) != sum(sum(row) for row in
adj2):
        return False

    try:
        eig1 = sorted(np.linalg.eigvals(adj1))
        eig2 = sorted(np.linalg.eigvals(adj2))
        if not np.allclose(eig1, eig2, atol=1e-6):
            return False
    except:
        pass

    return True
```

Изчислява степените на всички върхове за всеки граф, сортира степените и ги сравнява.

```
deg1 = sorted(sum(row) for row in adj1)
deg2 = sorted(sum(row) for row in adj2)
if deg1 != deg2:
    return False
```

Преброява всички ребра в двата графа.

```
if sum(sum(row) for row in adj1) != sum(sum(row) for row in adj2):
    return False
```

Изчислява собствените стойности (спектъра) на двете матрици и ги сравнява с точност 6 цифри след десетичната запетая.

```
try:
    eig1 = sorted(np.linalg.eigvals(adj1))
    eig2 = sorted(np.linalg.eigvals(adj2))
    if not np.allclose(eig1, eig2, atol=1e-6):
        return False
except:
    pass
```

Функцията convert_graph_to_adj_matrix преобразува граф от библиотеката NetworkX в матрица на съседство. Тя започва със сортиране на върховете на графа по възходящ ред, за да се гарантира консистентен ред при обработката. След това инициализира квадратна матрица с размери n x n, запълнена с нули, където n е броят на върховете. Чрез двойна итерация по всички двойки върхове, функцията проверява за наличие на ребро между тях. Ако ребро съществува, съответният елемент в матрицата се променя на 1, в противен случай остава 0. Резултатът е матрица на съседство, която визуализира връзките между върховете, като за неориентирани графи матрицата е симетрична спрямо главния диагонал. Тази матрица е полезна за бърз достъп до информация за връзките в графа, използва се в различни алгоритми за анализ на графи и проверка за изоморфизъм, но има ограничения при работа с много големи графи поради високата паметова сложност $O(n^2)$. Функцията не е предназначена за работа с ориентирани графи без допълнителни модификации.

```
def convert_graph_to_adj_matrix(graph):
    """Convert networkx graph to adjacency matrix"""
    nodes = sorted(graph.nodes())
    n = len(nodes)
    adj = [[0]*n for _ in range(n)]

    for i, u in enumerate(nodes):
        for j, v in enumerate(nodes):
            if graph.has_edge(u, v):
                adj[i][j] = 1
    return adj
```

6. VF2 - Ивайло Николов, ф.н. 121222009

VF2 е алгоритъм, който определя дали два графа са изоморфни като използва техника "разделяй и владей". При тази техника един проблем се разделя на по-малки такива, което позволява по-ефективното им решение. Например, разделяме проблема на десет малки части. Решаваме всяка една от тях и след това ги събираме по двойки като получаваме пет части. Решаваме всяка една от тях и отново ги събираме по двойки като получаваме три части. Чрез последователно прилагане на същия метод получаваме две части и накрая една. Решението на всяка една подчаст е по-лесно, понеже преди това е била решена още по-малка подчаст.

Целта на VF2 е да състави изображение m: $V_1 \rightarrow V_2$, което е биекция. Най-напред се проверява дали броя върхове $|V_1|$ съвпада с броя върхове $|V_2|$. След това се започва с изображение m, което е празно множество. Следващата стъпка е да се генерира декартово произведение от съпоставки (a, b), $a \in V_1$, $b \in V_2$, които са "кандидати" за добавяне. Всеки един кандидат се проверява дали отговаря на дадени условия и ако отговаря се изпълнява рекурсивно целия алгоритъм като се започва с изображение, към което е добавен конкретния кандидат. Когато броя съпоставки в m стане равен на броя върхове във V_1, ($|V_1| = |V_2|$), тогава двете графи са изоморфни. Ако след всички обхождания и рекурсии броя съпоставки в m не достигне $|V_1|$, то графите не са изоморфни.

Примерна процедура на алгоритъма VF2 може да се разгледа на фиг. 3.1.

Algorithm 1 *A high level description of VF2*

```
1: procedure VF2(Mapping m, ProblemType PT)
2:     if m covers V1 then
3:         Output(m)
4:     else
5:         Compute the set Pm of the candidate pairs for extending m
6:         for all p ∈ Pm do
7:             if ConsPT(p, m) ∧ ¬CutPT(p, m) then
8:                 call VF2(extend(m, p), PT)
```

(фиг. 11) Процедура на VF2

Частите от тази процедура, които не са разгледани са: как точно се генерира множеството от кандидати P_m и какво представляват бинарните функции $Cons_{PT}(p, m)$, $Cut_{PT}(p, m)$.

Генерирането на P_m става посредством следните правила:

Нека:

$T_1(m) \coloneqq \{u \mid u \in V_1 \backslash D(m): \exists \tilde{u} \in D(m): (u, \tilde{u}) \in E_1\}$

$T_2(m) \coloneqq \{v \mid v \in V_1 \backslash R(m): \exists \tilde{v} \in R(m): (v, \tilde{v}) \in E_2\}$,

където $D(m)$ и $R(m)$ са съответно дефиниционното множество и определеното множество на m. Множеството на кандидатите е:

$$P_m = \begin{cases} T_1(m) \times T_2(m), & T_1(m) \neq \emptyset \wedge T_2(m) \neq \emptyset \\ (V_1 \backslash D(m)) \times (V_2 \backslash R(m)), & otherwise \end{cases}$$

$Const_{PT}(p, m)$ представлява бинарна функция, която връща "истина" или "лъжа", в зависимост от това дали кандидата p, чрез който ще се разшири изображението m, запазва

коректността на съпоставките в разширеното изображение. Самата функция представлява следното нещо:
Нека:

$\Gamma_1(u)=\{\mu \in V_1 : (u,\mu) \in E_1\}$

$\Gamma_2(v)=\{\gamma \in V_2 : (v,\gamma) \in E_2\}$

Тук $\Gamma_i(a)$ показва множеството от съседи на върха $a \in G_i$. Използвайки тези две множества извеждаме:

$$Const_{PT}((u,v),m)=(\forall \gamma \in \Gamma_2(v) \cap R(m) : (u, m^{-1}(\gamma)) \in E_1) \wedge (\forall \mu \in \Gamma_1(u) \cap D(m) : (v, m(\mu)) \in E_2)$$

Това показва, че върховете u и v, които ще добавяме към m трябва да са такива, че съседите на u трябва да са със същия брой и съотношение като съседите на v, за да се запази коректността на изобажението.

Функцията $Cut_{PT}(p,m)$ също е бинарна функция, която връща “истина” ако е невъзможно пълното разширение на вече разширено изображение m с p до размер равен на $|V_1|$. В противен случай връща “лъжа”. Това се постига по следния начин:
Нека:

$T_1'(m)=(V_1 \setminus D(m)) \setminus T_1(m)$

$T_2'(m)=(V_2 \setminus R(m)) \setminus T_2(m)$

Тогава имаме:

$$Cut_{PT}((u,v),m)=|\Gamma_2(v) \cap T_2(m)| < |\Gamma_1(u) \cap T_1(m)| \vee |\Gamma_2(v) \cap T_2'(m)| < |\Gamma_1(u) \cap T_1'(m)|$$

Когато за даден кандидат $Const_{PT}(p,m)$ е “истина”, а $Cut_{PT}(p, m)$ е “лъжа”, то кандидата отговаря на изискванията за разширяване на m и се извиква рекурсивно същата функция като и се подава вече разширеното изображение m. [7]

6.1. Софтуерна реализация на VF2

Софтуерната реализация се постига чрез функции, които генерират и изпълняват гореописаните правила, множества и изображения. Кода е сравнително кратък и прост, понеже на са съобразени оптимизации, а е реализиран чист VF2. Той представлява следното:

```
def vf2_isomorphism(graph1, graph2):
    """
    Check if two graphs are isomorphic using the VF2 algorithm.

    Parameters:
        graph1 (Graph): The first graph.
        graph2 (Graph): The second graph.

    Returns:
        bool: True if the graphs are isomorphic, False otherwise.
```

```
    """
    if len(graph1.nodes) != len(graph2.nodes):
        return False

    def is_feasible(mapping, u, v):
        # Check if the mapping is consistent with the graph structure
        for mapped_u, mapped_v in mapping.items():
            if v in mapping.values():
                return False
            if u in mapping.keys():
                return False
            if mapped_v in graph2.neighbors(v) and mapped_u not in
graph1.neighbors(u):
                return False
        return True

    def match(mapping):
        if len(mapping) == len(graph1.nodes):
            return True

        for u in graph1.nodes:
            if u not in mapping:
                for v in graph2.nodes:
                      if v not in mapping.values() and is_feasible(mapping,
u, v):
                        mapping[u] = v
                        if match(mapping):
                            return True
                        del mapping[u]
                return False
        return False

    return match({})
```

Подаваме му два графа graph1 и graph2. Те са изградени чрез пакета networkx за python, който се използва за работа с графи. Първо проверяваме дали двата графа са с еднакъв брой върхове и ако не са, то те не са изоморфни:

```
if len(graph1.nodes) != len(graph2.nodes):
    return False
```

След това създаваме функция, която ще проверява дали съпоставените върхове u и v отговарят на условията за добавянето им към m:

```
def is_feasible(mapping, u, v):
    # Check if the mapping is consistent with the graph structure
    for mapped_u, mapped_v in mapping.items():
        if v in mapping.values():
            return False
        if u in mapping.keys():
            return False
        if mapped_v in graph2.neighbors(v) and mapped_u not in
graph1.neighbors(u):
            return False
return True
```

Тук mapping представлява hashmap, който на всеки ключ mapped_u има стойност mapped_v. За всички съпоставки от mapping трябва u и v да не присъстват никъде в mapping. След това трябва u и v да са такива, че за всички съпоставки от mapping, никое mapped_v да не е съсед на v и всяко mapped_u да е съсед на u. Ако всички тези условия са изпълнени, is_feasable връща "истина".

Следващата функция е match, която взима кандидат (u, v) за добавяне към m, проверява дали е валиден като използва гореспоменатата функция is_feasable и ако е, го добавя към m и вика рекурсивно себе си с вече разширено m.

```
def match(mapping):
    if len(mapping) == len(graph1.nodes):
        return True

    for u in graph1.nodes:
        if u not in mapping:
            for v in graph2.nodes:
                  if v not in mapping.values() and is_feasible(mapping, u,
v):
                    mapping[u] = v
                    if match(mapping):
                        return True
                    del mapping[u]
            return False
    return False
```

Първо се проверява дали всички върхове от graph1 са покрити от mapping и ако са, директно се връща "истина". Ако това условие не е спазено, за всеки връх u от graph1, който не присъства в mapping и за всеки връх v, който не присъства в mapping и ако (u, v) отговаря на условията във функцията is_feasible, то към mapping се добавя (u, v), след което се извиква рекурсивно функцията match с вече разширения mapping и ако тя връща "истина", то и текущата функция match връща "истина". В противен случай се премахва добавеното към mapping (u, v). Ако нито един от върховете u или v не спазва условията, то се връща "лъжа".

Накрая външната функция vf2_isomorphism, на която се подават параметри graph1 и graph2 и която съдържа гореспоменатите функции is_feasible и match връща като резултат резултата от извикването на функцията match като за аргумент и се подава празен mapping:

```
return match({})
```

Това което може да се отбележи тук е, че сложността на този алгоритъм е много висока. Вижда се, че в match имаме два вложени цикъла като във втория се извиква функцията is_feasible, която също съдържа цикъл, т. е. стават три вложени цикъла. След това се прави рекурсия като всяка такава също има три вложени цикъла. Това означава, че сложността придобива вида $O(n^n)$. Понеже тук се получава претърсване в дълбочина на графите (depth first search), то тази сложност е напълно очаквана. Това, обаче, също означава, че той отнема много време за изчисление и при някои графи може да се изпълнява на практика "безкрайно дълго".

Пакета на networkx освен начини за представяне на графи се съдържат и методи за проверка на изоморфизъм. Има метод за намиране на изоморфизъм чрез VF2 като това става по следния начин:

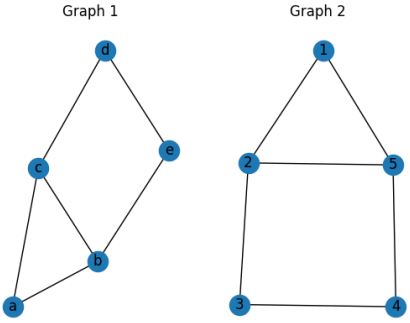
```
import networkx as nx
GM = nx.algorithms.isomorphism.GraphMatcher(graph1, graph2)
result = GM.is_isomorphic()
```

Реализацията на networkx на алгоритъма VF2 е по-оптимизирана и съответно - по-бърза от ръчно написаната по-горе. [8]

7. Получени резултати за решението на проблема

За да се демонстрира работата на описаните алгоритми за проверка на изоморфизъм, проведени са 9 теста, чиито графи и резултати са представени на фигурите долу.

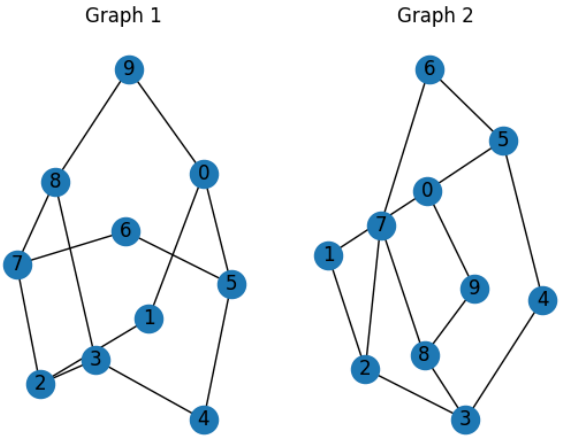
Тест 1. Изоморфни графи




```
Test 1:
VF2 : Graphs are isomorphic
Color Refinement : Graphs are isomorphic
Adjacency Matrix : Graphs are isomorphic
```

(фиг. 12) Резултати от тест 1.

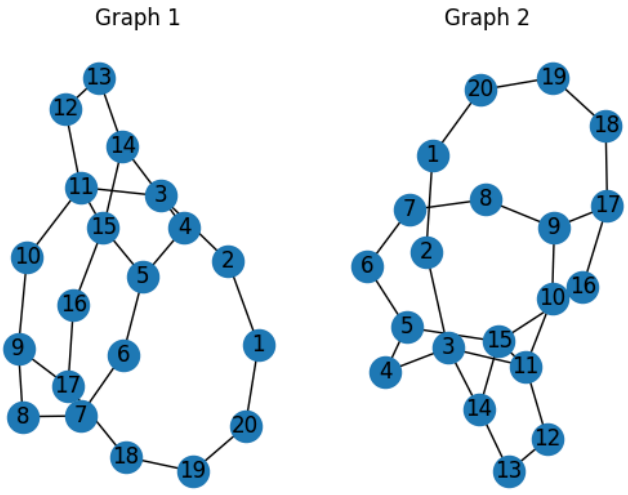
Тест 2.
Изоморфни графи




```
Test 2:
VF2 : Graphs are isomorphic
Color Refinement : Graphs are isomorphic
Adjacency Matrix : Graphs are isomorphic
```

(фиг. 13) Резултати от тест 2.

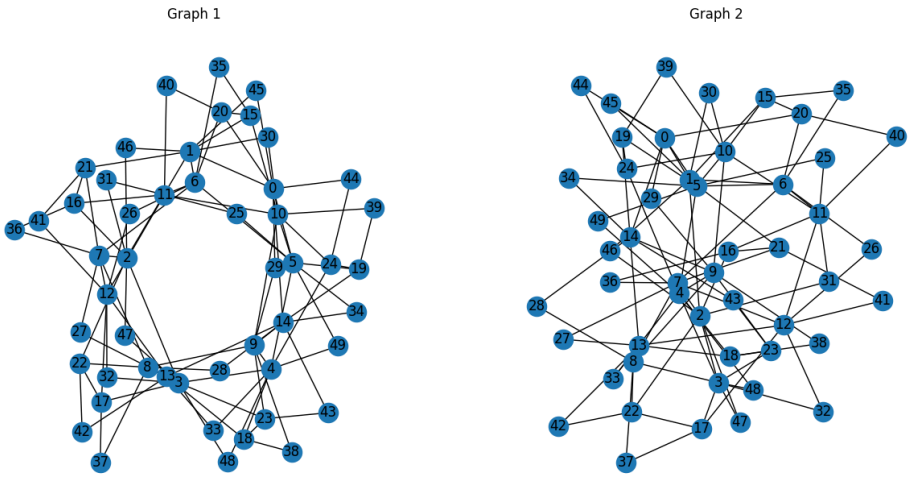
Тест 3. Изоморфни графи




```
Test 3:
VF2 : Graphs are isomorphic
Color Refinement : Graphs are isomorphic
Adjacency Matrix : Graphs are isomorphic
```

(фиг. 14) Резултати от тест 3.

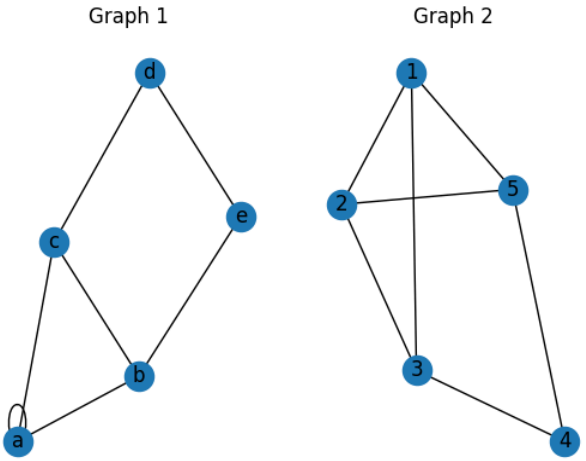
Тест 4. Изоморфни графи




```
Test 4:
VF2 : Graphs are isomorphic
Color Refinement : Graphs are isomorphic
Adjacency Matrix : Graphs are isomorphic
```

(фиг. 15) Резултати от тест 4.

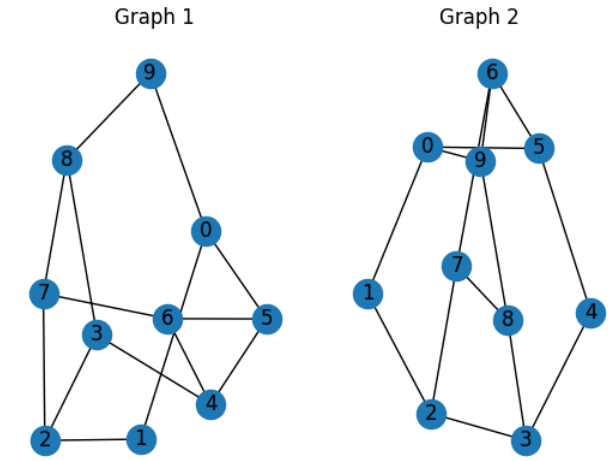
Тест 5. Неизоморфни графи




```
Test 5:
VF2 : Graphs are NOT isomorphic
Color Refinement : Graphs are NOT isomorphic
Adjacency Matrix : Graphs are NOT isomorphic
```

(фиг. 16) Резултати от тест 5.

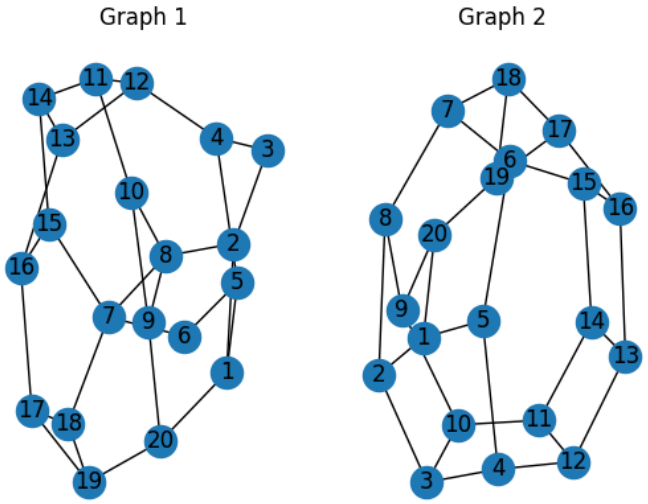
Тест 6. Неизоморфни графи




```
Test 6:
VF2 : Graphs are NOT isomorphic
Color Refinement : Graphs are NOT isomorphic
Adjacency Matrix : Graphs are NOT isomorphic
```

(фиг. 17) Резултати от тест 6.

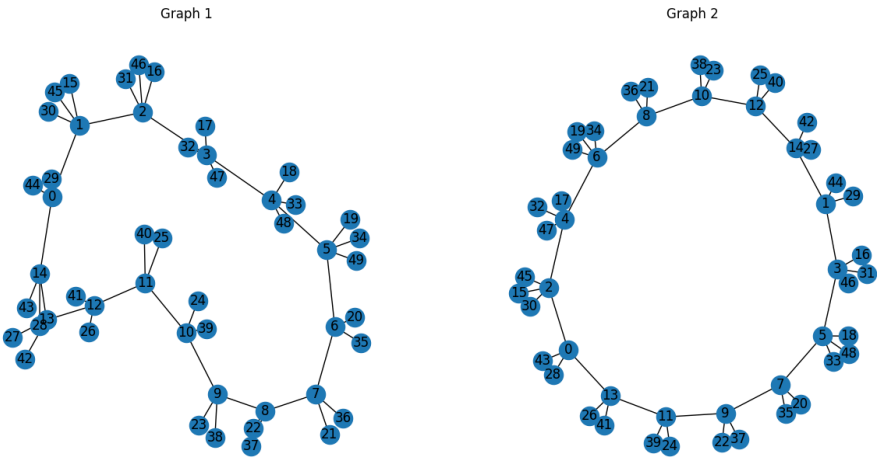
Тест 7. Неизоморфни графи




```
Test 7:
VF2 : Graphs are NOT isomorphic
Color Refinement : Graphs are NOT isomorphic
Adjacency Matrix : Graphs are NOT isomorphic
```

(фиг. 18) Резултати от тест 7.

Тест 8. Неизоморфни графи

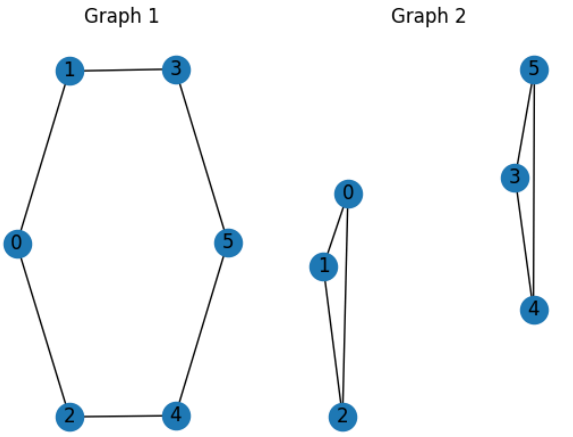



```
Test 8:
VF2 : Graphs are NOT isomorphic
Color Refinement : Graphs are NOT isomorphic
Adjacency Matrix : Graphs are NOT isomorphic
```

(фиг. 19) Резултати от тест 8.

В тестовете 7 и 8 за VF2 е използван алгоритъмът от библиотеката networkX, чиято имплементация включва допълнителни оптимизации, които ускоряват работата и подобряват точността. В имплементацията в подточка 6.1. ги няма оптимизациите, поради което програмата зацикля.

Тест 9. Неизоморфни графи




```
Test 9:
VF2 : Graphs are NOT isomorphic
Color Refinement : Graphs are isomorphic
Adjacency Matrix : Graphs are NOT isomorphic
```

(фиг. 20) Резултати от тест 9.

Причината защо 1-WL теста е дал резултат, че двата графа са изоморфни е съвпадението на множествата на съседите на върховете в двата графа. Резултатът е грешен защото графите са с различна топология.

8. Заключение

В настоящата работа бяха разгледани три алгоритъма за анализ на изоморфизъм на графи. Всеки от разгледаните алгоритми има своите предимства, ограничения и области на приложение, което ги прави подходящи за различни сценарии в теорията на графите и приложните ѝ области.

1-WL алгоритъмът се доказа като мощен инструмент за класификация на графи според тяхната структура чрез итеративно преоразмеряване на етикетите на върхове. Той е особено ефективен при работа със стандартни графи, но проявява ограничения при разпознаването на някои специфични структури, което го прави подходящ предимно за предварителна филтрация.

VF2 алгоритъмът предлага точен и надежден метод за проверка на изоморфизъм чрез систематично търсене на съответствия между върховете. Въпреки по-високата му изчислителна сложност при големи графи, той гарантира прецизност и е особено ценен в задачи, изискващи строга коректност.

Adjacency Matrix Comparison се оказва бърз и интуитивен подход, полезен за малки графи или ситуации, в които равенството на структурите може да бъде проверено директно. Въпреки това, методът е неефективен за мащабни графи поради високата си изчислителна сложност и липсата на способност за разпознаване на изоморфизъм при пренаредени върхове.

Изисквания като бързодействие, точност или мащабируемост оказват огромно влияние върху изборът на подходящ алгоритъм и той може значително да повлияе на ефективността на решението. Бъдещи изследвания биха могли да се фокусират върху хибридни подходи, комбиниращи силните страни на разгледаните методи, за да се постигне оптимален баланс между скорост и прецизност.

9. Използвана литература

[1] Симона Филипова-Петракиева, Валери Младенов. (2023). Решени примери по Дискретни структури. АВАНГАРД ПРИМА

[2] Geeks for Geeks (2024, September 27). Mathematics | Graph Isomorphisms and Connectivity, https://www.geeksforgeeks.org/graph-isomorphisms-connectivity/

[3] University of Pennsylvania, lecture notes. (2016, February 22). Graph isomorphism. Академична публикация. https://www2.math.upenn.edu/~mlazar/math170/notes05-2.pdf

[4] Ningyuan (Teresa) Huang, Soledad Villar. (2021). A SHORT TUTORIAL ON THE WEISFEILER-LEHMAN TEST AND ITS VARIANTS. Академична публикация. https://par.nsf.gov/servlets/purl/10299993

[5] Wikipedia - Adjacency matrix, https://en.wikipedia.org/wiki/Adjacency_matrix

[6] Graph isomorphism—*Characterization and efficient algorithms,* *https://www.sciencedirect.com/science/article/pii/S2667295224000278*

[7] Alpár Jüttner/Péter Madarasi (2018, March 29). *VF2++—An improved subgraph isomorphism algorithm*, https://www.sciencedirect.com/science/article/pii/S0166218X18300829

[8] NetworkX documentation. *VF2 Algorithm*, https://networkx.org/documentation/stable/reference/algorithms/isomorphism.vf2.html

10. Сорс-код на разгледаните алгоритми

Кодът е качен в гит репоситори на следния линк:
https://github.com/boce1/isomorphism_DS_project.git