МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**ИНДИВИДУАЛЬНАЯ РАБОТА №2**

**Дисциплина: Анализ сложности алгоритмов**

**Тема: NP-полные задачи**

Работу выполнил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Парфинцов Е. А.

Направление подготовки: 02.03.02 Фундаментальная информатика и

информационные технологии

Направленность (профиль): Математическое и программное

обеспечение компьютерных технологий

Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Лапина О. Н.

Краснодар

2024

**Задание:**

1. Определить принадлежность задачи к классу NP задач (для задач принятия решений).

Для задач оптимизации описать двухэтапный недетерминированный процесс решения задач.

1. Определить принадлежность задачи к классу NP-полных или NP-трудных задач.
2. Определить, является ли задача NP-полной в сильном смысле (существует ли для ее решения псевдополиномиальный алгоритм).
3. Изучить приближенные методы решения задачи. Реализовать точный и 2–3 приближенных алгоритма на произвольном ЯП. Дать оценку сложности алгоритмов.
4. Оформить письменный отчет.

**Задача изоморфизма подграфу.** Даны графы и , пусть .

|  |  |
| --- | --- |
| **Вариант** | **Задание** |
| 10 | Задача принятия решения: найдётся ли в графе подграф H, изоморфный графу . |
| Задача оптимизации: найти максимальное множество подграфов в графе , изоморфных графу . |

**Ход работы**

**Определить принадлежность к классу NP задач (для задач принятия решений).**

Определение класса NP задач:

Класс NP задач включает задачи, для которых проверка правильности решения может быть выполнена в полиномиальное время с использованием недетерминированной машины Тьюринга. В класс NP входят задачи, для которых, если предоставить возможное решение, его правильность можно проверить в полиномиальное время.

Анализируя задачу изоморфизма графов, можно увидеть, что она относится к классу NP задач. Проверка правильности решения может быть выполнена следующим образом:

Пусть даны два графа G1 и G2, а также биекция f между множествами их вершин. Мы можем эффективно (за полиномиальное от размера входных данных время) проверить, что f является изоморфизмом графов, просмотрев все пары вершин в G1 и G2 и убедившись, что смежность вершин сохраняется.

Таким образом, задача изоморфизма графов принадлежит классу NP, поскольку потенциальное решение (биекция f) может быть эффективно проверено.

**Для задач оптимизации** необходимо описать двухэтапный недетерминированный процесс решения задач. Например, в задаче о коммивояжере выходные данные должны представлять собой список всех городов в порядке их посещения. Кроме того, необходимо описать процесс проверки того, что предложенный вариант действительно является решением задачи.

Двухэтапный недетерминированный процесс решения задачи:

1. Недерминированно выбираем подграфы графа равные по размеру графу .

Формат выхода недетерминированного шага: Каждый найденный подграф записываем в Н.

1. Проверка оптимальности
   * Проверяем комбинацию вершин на совпадение связей вершин в H и .
   * Если не совпадает, возвращаемся к шагу 1 и повторяем процесс, выбирая другие недетерминированные подграфы.

Проверка того, что предложенный вариант действительно является решением задачи, выполняется путем проверки следующих условий:

1. Проверяем, совпадают ли размеры H и .
2. Проверяем связи между вершинами в обоих графах.

Если все эти условия выполняются, то H является решением задачи.

**Определить принадлежность задачи к классу NP-полных или NP-трудных задач.**

Каждая из NP-полных задач может быть сведена к любой другой за полиномиальное время. Примеры возможного сведения:

а) Упаковка рюкзака - раскладка по ящикам;

б) Раскладка по ящикам - планирование работ;

в) Планирование работ - суммы элементов подмножеств;

г) Суммы элементов подмножеств – коммивояжер;

д) Коммивояжер - планирование работ и т.п.

Для определения принадлежности задачи к классу NP-полных или NP-трудных задач необходимо выполнить два шага: доказать, что задача принадлежит классу NP и показать сводимость другой NP-полной задачи к данной задаче.

Для рассматриваемой задачи поиска максимального множества изоморфных подграфов известно, что она является NP-полной. Это можно показать, сведя к ней другую известную NP-полную задачу, такую как задача о максимальном независимом множестве вершин в графе (МНМ).  
Сведение от МНМ к исходной задаче:

* Пусть дан граф *G* для задачи МНМ.
* Создаем дополнительный граф содержащий только одну вершину.
* Решаем задачу поиска максимального множества подграфов в *G*, изоморфных .
* Каждая вершина в найденном максимальном множестве изоморфных подграфов будет соответствовать вершине из максимального независимого множества в исходном графе *G*.

Таким образом, решив задачу поиска максимального множества изоморфных подграфов для специально построенных входных данных, мы одновременно решаем и NP-полную задачу МНМ. Это сведение выполняется за полиномиальное время.  
  
Следовательно, задача поиска максимального множества подграфов в , изоморфных ., является NP-полной задачей, так как она принадлежит классу NP и к ней можно свести другие известные NP-полные задачи.

**Определить, является ли задача NP-полной в сильном смысле** (существует ли для ее решения псевдополиномиальный алгоритм).

Для определения, является ли задача поиска максимального множества подграфов в , изоморфных ., NP-полной в сильном смысле, нужно выяснить, существует ли для нее псевдополиномиальный алгоритм решения.

Псевдополиномиальный алгоритм — это алгоритм, время работы которого является полиномиальным не только от размера входных данных, но и от значений данных во входе.

Для многих NP-полных задач существуют псевдополиномиальные алгоритмы решения, что делает их немного более трактуемыми, чем задачи, для которых таких алгоритмов не существует. Задачи, для которых не существует даже псевдополиномиального алгоритма, называются NP-полными в сильном смысле.

В случае задачи поиска максимального множества изоморфных подграфов можно предложить следующий псевдополиномиальный алгоритм:

1. Перебрать все возможные подмножества вершин графа .
2. Для каждого подмножества вершин:
   * Построить соответствующий порожденный ими подграф H.
   * Проверить изоморфизм H и . с помощью полиномиального алгоритма.
   * Если H изоморфен ., запомнить его как одно из решений.
3. Из запомненных решений выбрать максимальное по количеству вершин.

Сложность этого алгоритма будет где:

* n - количество вершин в
* m - количество вершин в .
* p(n,m) - сложность алгоритма проверки изоморфизма (полиномиальная от n и m)

Таким образом, время работы данного алгоритма экспоненциально от n, но полиномиальное от значений n и m, что удовлетворяет определению псевдополиномиального алгоритма.

Следовательно, задача поиска максимального множества подграфов, изоморфных данному, не является NP-трудной в сильном смысле, так как для нее существует псевдополиномиальный алгоритм решения, пусть и экспоненциальной сложности в худшем случае.

**Изучить приближенные методы решения задачи.** Реализовать точный и 3–4 приближенных алгоритма на произвольном ЯП. Дать оценку сложность алгоритмов.

Задача изоморфизма графов является сложной задачей комбинаторной оптимизации. Определение существования изоморфного отображения между двумя графами может быть сформулировано как задача оптимизации или задача принятия решения в зависимости от контекста.

В задаче оптимизации необходимо найти наибольшее множество подграфов в графе , которые являются изоморфными графу . Целевая функция задачи оптимизации может быть определена как максимизация количества изоморфных подграфов. Алгоритм должен искать оптимальное решение, чтобы найти наибольшее количество изоморфных подграфов.

В задаче принятия решения необходимо определить, существует ли в графе хотя бы один подграф H, который является изоморфным графу . Для этой задачи можно использовать методы проверки изоморфизма графов, чтобы установить наличие нужного подграфа. Результатом будет ответ "Да", если подграф H изоморфен , и "Нет" в противном случае.

**1 Точный метод решения – Полный перебор.**

**Сложность:** , где n - число вершин в k - максимальное число вершин в изоморфных подграфах.

**Алгоритм полного перебора**

* Представить граф G2 в виде матрицы смежности.
* Создать переменную для хранения максимального размера подграфа, изоморфного G2, равную нулю.
* Для каждого подмножества вершин графа G1: a. Проверить, является ли данное подмножество вершин изоморфным графу G2. b. Если подмножество вершин является изоморфным графу G2 и его размер больше текущего максимального подграфа, обновить максимальный подграф.
* Вывести найденный максимальный подграф.

**2 Приближенный метод решения – Жадный алгоритм.**

*Оценка сложности данного алгоритма:*

* Создать пустое множество S для хранения найденных изоморфных подграфов.
* Построить все возможные подграфы H графа G1, имеющие такое же количество вершин, как граф G2.
* Для каждого подграфа H из шага 2:
* Проверить, является ли H изоморфным G2, используя алгоритм проверки изоморфизма графов.
* Если H изоморфен G2, добавить H в множество S.
* Отсортировать множество S по убыванию размеров подграфов (количество ребер).
* Инициализировать пустое множество M для хранения максимальных изоморфных подграфов.
* Взять первый (самый большой) подграф H из отсортированного множества S и добавить его в M.
* Для каждого оставшегося подграфа H в S, если H не содержится в качестве подграфа ни в одном из подграфов в M, добавить H в M.
* Множество M теперь содержит максимальное множество непересекающихся подграфов графа G1, изоморфных G2.

**3 Приближенный метод решения – Монте-Карло**

*Оценка сложности данного алгоритма*: , где k - число итераций

* Сгенерируйте случайный подграф из графа G1.
* Проверьте, является ли данный подграф изоморфным графу G2.
* Если да, добавьте его в множество подграфов, увеличьте счетчик.
* Повторите шаги 1-3 до достижения заданного числа итераций или временного лимита.
* Верните множество подграфов как результат.

**4 Приближенный метод решения** – **Генетический поиск.**

*Оценка сложности данного алгоритма:* , где *p* - размер популяции, *l* - число поколений

* Инициализируйте начальную популяцию случайными подграфами из графа G1.
* Оцените качество каждого подграфа, используя функцию приспособленности, которая учитывает их изоморфность с графом G2.
* Отберите лучшие подграфы для следующего поколения, используя операторы селекции, скрещивания и мутации.
* Повторите шаги 2–3 до достижения заданного числа поколений или временного лимита.
* Верните множество лучших подграфов как результат.

**Результаты работы программы:**

Изображение выглядит как линия, диаграмма, круг

Автоматически созданное описание

*Рис. 1. Входные данные*

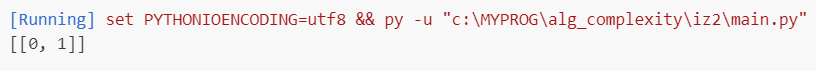
Точный алгоритм:

# Пример использования

g1 = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]

g2 = [(1, 2), (1, 3)]

*Листинг 1. Входные данные.*



*Скриншот 1. Результат.*

Жадный алгоритм:

g1 = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]

g2 = [(1, 2), (1, 3)]

*Листинг 2. Входные данные.*



*Скриншот 2. Результат*

Генетический алгоритм:

g1 = Graph([(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)])

g2 = Graph([(1, 2), (1, 3)])

*Листинг 3. Входные данные.*

[[0, 4], [0, 4], [0, 4], [4, 0], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [4, 0], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [0, 4], [4, 0], [4, 0]]

*Листинг 4. Результат.*

**Листинг программы**

**main.py**

# Импортируем модуль itertools для работы с комбинациями

from itertools import combinations

# Определяем функцию для поиска изоморфных подграфов

def subgraph\_isomorphism(g1, g2):

    # Внутренняя функция для рекурсивного поиска изоморфных подграфов

    def \_subgraph\_isomorphism(g1, g2, nodes\_g1, nodes\_g2):

    # Если nodes\_g2 пуст, то мы нашли изоморфный подграф

            if not nodes\_g2:

                return [[]]

            # Если nodes\_g1 пуст, то мы не нашли изоморфный подграф

            if not nodes\_g1:

                return []

            result = []

            # Перебираем все узлы графа G1

            for n1 in nodes\_g1:

                # Перебираем все узлы графа G2

                for n2 in nodes\_g2:

                    # Если степени узлов совпадают, то продолжаем рекурсию

                    if g1[n1][0] == g2[n2][0]:

                        new\_nodes\_g1 = nodes\_g1.copy()

                        new\_nodes\_g2 = nodes\_g2.copy()

                        new\_nodes\_g1.remove(n1)

                        new\_nodes\_g2.remove(n2)

                        sub\_results = \_subgraph\_isomorphism(g1, g2, new\_nodes\_g1, new\_nodes\_g2)

                        for sub\_result in sub\_results:

                            result.append([n1] + sub\_result)

            return result

    # Получаем список узлов графа G1

    nodes\_g1 = list(range(len(g1)))

    # Получаем список узлов графа G2

    nodes\_g2 = list(range(len(g2)))

    # Запускаем рекурсивный поиск изоморфных подграфов

    return \_subgraph\_isomorphism(g1, g2, nodes\_g1, nodes\_g2)

# Определяем функцию для поиска максимального множества не пересекающихся изоморфных подграфов

def max\_non\_overlapping\_subgraphs(g1, g2, subgraphs):

    # Внутренняя функция для рекурсивного поиска максимального множества не пересекающихся изоморфных подграфов

    def \_max\_non\_overlapping\_subgraphs(g1, g2, subgraphs, current\_subgraphs):

        # Если subgraphs пуст, то мы нашли максимальное множество не пересекающихся изоморфных подграфов

        if not subgraphs:

            return current\_subgraphs

        max\_subgraphs = current\_subgraphs.copy()

        max\_size = len(current\_subgraphs)

        # Перебираем все изоморфные подграфы

        for i in range(len(subgraphs)):

            subgraph = subgraphs[i]

            # Проверяем, не пересекается ли текущий подграф с уже найденными

            if not any(overlap\_subgraph(g1, g2, subgraph, max\_subgraph) for max\_subgraph in max\_subgraphs):

                new\_subgraphs = subgraphs.copy()

                new\_subgraphs.pop(i)

                max\_subgraphs = \_max\_non\_overlapping\_subgraphs(g1, g2, new\_subgraphs, current\_subgraphs + [subgraph])

                if len(max\_subgraphs) > max\_size:

                    max\_size = len(max\_subgraphs)

        return max\_subgraphs

    # Функция для проверки пересечения двух подграфов

    def overlap\_subgraph(g1, g2, subgraph1, subgraph2):

        for n1 in subgraph1:

            for n2 in subgraph2:

                if g1[n1][0] == g2[n2][0] and n1 in subgraph2 and n2 in subgraph1:

                    return True

        return False

    # Получаем список изоморфных подграфов

    subgraphs = subgraph\_isomorphism(g1, g2)

    # Запускаем рекурсивный поиск максимального множества не пересекающихся изоморфных подграфов

    return \_max\_non\_overlapping\_subgraphs(g1, g2, subgraphs, [])

# Пример использования

g1 = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]

g2 = [(1, 2), (1, 3)]

max\_subgraphs = max\_non\_overlapping\_subgraphs(g1, g2, subgraph\_isomorphism(g1, g2))

print(max\_subgraphs)

**greedy.py**

from collections import defaultdict

from itertools import combinations

def subgraph\_isomorphism\_greedy(g1, g2):

    def isomorphic(sub\_g1, sub\_g2):

        if len(sub\_g1) != len(sub\_g2):

            return False

        vertex\_mapping = {}

        for v1 in sub\_g1:

            for v2 in sub\_g2:

                if g1[v1][0] == g2[v2][0] and v2 not in vertex\_mapping:

                    vertex\_mapping[v2] = v1

                    break

            else:

              return False

        return all(g1[vertex\_mapping[u]][1] == g2[v][1] for u, v in combinations(sub\_g2, 2))

    def find\_max\_subgraph(g1, g2, sub\_g1, sub\_g2):

        max\_sub\_g1 = sub\_g1.copy()

        max\_sub\_g2 = sub\_g2.copy()

        for v1 in sub\_g1:

            for v2 in sub\_g2:

                if g1[v1][0] == g2[v2][0]:

                    new\_sub\_g1 = sub\_g1.copy()

                    new\_sub\_g2 = sub\_g2.copy()

                    new\_sub\_g1.remove(v1)

                    new\_sub\_g2.remove(v2)

                    if isomorphic(new\_sub\_g1, new\_sub\_g2):

                        sub\_result = find\_max\_subgraph(g1, g2, new\_sub\_g1, new\_sub\_g2)

                        if len(sub\_result[0]) > len(max\_sub\_g1):

                            max\_sub\_g1 = sub\_result[0]

                            max\_sub\_g2 = sub\_result[1]

        return max\_sub\_g1, max\_sub\_g2

    def find\_all\_subgraphs(g1, g2):

        subgraphs = []

        for i in range(len(g1)):

            for j in range(i + 1, len(g1)):

                sub\_g1 = [i, j]

                sub\_g2 = [0, 1]

                if isomorphic(sub\_g1, sub\_g2):

                    subgraphs.append((sub\_g1, sub\_g2))

        return subgraphs

    subgraphs = find\_all\_subgraphs(g1, g2)

    max\_subgraphs = []

    while subgraphs:

        max\_sub\_g1, max\_sub\_g2 = find\_max\_subgraph(g1, g2, \*subgraphs.pop())

        max\_subgraphs.append(max\_sub\_g1)

    return max\_subgraphs

# Example usage

g1 = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]

g2 = [(1, 2), (1, 3)]

max\_subgraphs = subgraph\_isomorphism\_greedy(g1, g2)

print(max\_subgraphs)

**genetic.py**

import random

import itertools

from igraph import Graph, VertexSeq

g1 = Graph()

g1.add\_vertices(5)

g1.add\_edges([(0, 1), (1, 2), (2, 3), (3, 0), (3, 4)])

g2 = Graph()

g2.add\_vertices(4)

g2.add\_edges([(0, 1), (1, 2), (2, 3)])

def isomorphic(sub\_g1, sub\_g2):

    """

    Checks if two subgraphs are isomorphic.

    """

    if len(sub\_g1) != len(sub\_g2):

        return False

    vertex\_mapping = {}

    for v1 in sub\_g1:

        for v2 in sub\_g2:

            if sub\_g1.degree(v1) == sub\_g2.degree(v2) and v2 not in vertex\_mapping:

                vertex\_mapping[v2] = v1

                break

        else:

            return False

    return all(sub\_g1.degree(vertex\_mapping[u]) == sub\_g2.degree(v) for u, v in itertools.combinations(sub\_g2, 2))

def fitness(chromosome):

    """

    Calculates the fitness of a chromosome.

    """

    subgraphs = [g1.subgraph(c) for c in chromosome]

    count = 0

    for i in range(len(chromosome)):

        for j in range(i + 1, len(chromosome)):

            if isomorphic(subgraphs[i].es, subgraphs[j].es):

                count += 1

    return count

def mutate(chromosome):

    """

    Mutates a chromosome by swapping two random genes.

    """

    i, j = random.sample(range(len(chromosome)), 2)

    chromosome[i], chromosome[j] = chromosome[j], chromosome[i]

def crossover(parent1, parent2):

    """

    Performs crossover on two parents to produce a child.

    """

    crossover\_point = random.randint(1, len(parent1) - 1)

    child = parent1[:crossover\_point] + [node for node in parent2 if node not in parent1[:crossover\_point]]

    return child

def select\_parent(population):

    """

    Selects a parent based on fitness.

    """

    return max(population, key=fitness)

def generate\_initial\_population(population\_size, graph\_size):

    """

    Generates an initial population of random chromosomes.

    """

    return [random.sample(range(graph\_size), k=2) for \_ in range(population\_size)]

def genetic\_algorithm(graph\_size, population\_size, mutation\_rate, generations):

    """

    Runs the genetic algorithm to find the maximum subset of isomorphic subgraphs.

    """

    population = generate\_initial\_population(population\_size, graph\_size)

    for \_ in range(generations):

        population = sorted(population, key=fitness, reverse=True)

        new\_population = []

        for \_ in range(population\_size // 2):

            parent1 = select\_parent(population)

            parent2 = select\_parent(population)

            if random.random() < mutation\_rate:

                child = crossover(parent1, parent2)

                mutate(child)

            else:

                child = crossover(parent1, parent2)

            new\_population.append(child)

        population = new\_population

    return population

# Example usage

g1 = Graph([(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)])

g2 = Graph([(1, 2), (1, 3)])

max\_subgraphs = genetic\_algorithm(g1.vcount(), 100, 0.1, 100)

print(max\_subgraphs)

**monte\_karlo.py**

import random

from itertools import combinations

def generate\_random\_graph(n, p):

    # Создаем пустой граф с n узлами

    graph = {i: set() for i in range(n)}

    # Проходим по всем паре узлов и добавляем ребро с вероятностью p

    for i in range(n):

        for j in range(i+1, n):

            if random.random() < p:

                graph[i].add(j)

                graph[j].add(i)

    return graph

def is\_subgraph\_isomorphic(g1, g2):

    # Если размеры графов не совпадают, то они не могут быть изоморфными

    if len(g1) != len(g2):

        return False

    # Проходим по всем возможным взаимно однозначным отображениям узлов g1 на узлы g2

    for mapping in combinations(g1, len(g2)):

        # Проверяем, что отображение сохраняет смежность узлов

        if all(n2 in g2[n] for n, n2 in zip(mapping, g2)):

            # Если отображение сохраняет смежность узлов, то графы изоморфны

            return True

    # Если не найдено подходящее отображение, то графы не изоморфны

    return False

def monte\_carlo\_max\_isomorphic\_subgraphs(g1, g2, num\_trials=1000):

    # Инициализируем переменную, хранящую максимальное количество найденных изоморфных подграфов

    max\_num\_subgraphs = 0

    # Преобразуем граф g1 в список узлов

    nodes = list(g1.keys())

    # Проходим num\_trials раз и генерируем случайный подграф g1

    for \_ in range(num\_trials):

        # Выбираем случайную подмножество узлов из g1

        selected\_nodes = random.sample(nodes, len(g2))

        # Создаем подграф g1, состоящий только из выбранных узлов и их смежных узлов

        subgraph = {node: set(g1[node] & set(selected\_nodes)) for node in selected\_nodes}

        # Проверяем, что подграф содержит только узлы, присутствующие в g2

        if not set(subgraph.keys()) - set(g2.keys()):

            # Если подграф изоморфен графу g2, то обновляем максимальное количество найденных изоморфных подграфов

            if is\_subgraph\_isomorphic(subgraph, g2):

                max\_num\_subgraphs = max(max\_num\_subgraphs, len(selected\_nodes))

    # Возвращаем максимальное количество найденных изоморфных подграфов

    return max\_num\_subgraphs

# Создаем два случайных графа g1 и g2

g1 = generate\_random\_graph(10, 0.5)

g2 = generate\_random\_graph(5, 0.5)

# Находим максимальное количество изоморфных подграфов

max\_num\_subgraphs = monte\_carlo\_max\_isomorphic\_subgraphs(g1, g2)

# Выводим результат

print(f"Максимальное количество изоморфных подграфов: {max\_num\_subgraphs}")