МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КУБГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**Отчет**

**по индивидуальному заданию №2**

**«Максимальная клика. Задача оптимизации»**

**по курсу**

**«ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМОВ»**

Работу выполнила

Студентка 46 группы

Баева Д.Н.

Преподаватель:

Нигодин Е.А.

**Доказательство принадлежности классу NP**

Клика — это подграф графа, все вершины которого связаны друг с другом, то есть подграф является полным графом. Задача о максимальной клике состоит в том, чтобы найти клику максимального размера данного графа G, то есть полного графа, который является подграфом G и содержит максимальное количество вершин. Это проблема оптимизации. Соответственно, проблема решения клики состоит в том, чтобы определить, существует ли клика размера k в данном графе или нет.

Пусть известен граф . Пусть сертификат представляет собой набор S, состоящий из узлов клики, а S — подграф G.

Нам нужно проверить, существует ли в графе клика размера k. Следовательно, проверка того, равно ли количество узлов в S k, занимает время O (1). Проверка того, имеет ли каждая вершина исходящую степень (k-1), занимает время O(). (Поскольку в полном графе каждая вершина соединена с каждой другой вершиной через ребро. Следовательно, общее количество ребер в полном графе ). Следовательно, чтобы проверить, является ли граф, образованный k узлами в S, полным или нет, требуется время O() = O() (поскольку k<=V, где V — количество вершин в G).

Следовательно, проблема принятия решения кликой имеет проверяемость за полиномиальное время и, следовательно, принадлежит классу NP.

**Доказательство принадлежности к классу NP – полных задач**

Известно, что множество графа является кликой тогда и только тогда, когда каждая вершина из этого множества соединена ребром с каждой вершиной того же множества. Пусть – следующая задача: даны графы ,. Является ли граф изоморфным подграфу (т. е. ∃ ли подмножество вершин , и подмножестве ребер , такое что и ∃ ли взаимно однозначное соответствие вершин в подмножестве вершин V из такое, что ). Эта задача является NP-полной. Задача сводится к задаче о клике за полиномиальное время, так как для решения задачи о клике можно сначала использовать алгоритм решения задачи , а затем вычислить, если в графе ∃ изоморфизм подграфа к графу , то в графе ∃ клика размера ( и наоборот, если в графе ∃ клика, то в графе ∃ изоморфизм подграфа к графу ) (это выполняется за полиномиальное время).

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

Исходная задача является NP-трудной, так как соответствующая ей задача принятия решения является NP-полной. Для полных графов исходная задача имеет полиномиальную сложность если k фиксирована, так как проверка того, является ли граф полным, выполняется за полиномиальное (квадратичное) время относительно числа вершин в графе. Если клика фиксированная, то проблема клики разрешима за полиномиальное время.

Псевдополиномиальный алгоритм означает, что время выполнения алгоритма ограничивается полиномом от размера входных данных и значения параметров задачи. Если необходимо найти клику больше размера чем k , то задача сводится к NP–полной и время работы становится экспоненциальной, поэтому не существует псевдополиномиального алгоритма.

**Приближённые методы решения задачи и алгоритмы полного перебора.**

В ходе работы было реализовано 3 приближенных и 2 точных метод (полный перебор и алгоритм Брона-Кербоша) решения задачи.

**1 точный метод решения – полный перебор.**

Сперва функция is\_clique(graph, vertices) проверяет, является ли данное множество вершин кликой в графе. Для каждой пары вершин u и v из vertices, проверяется, что между ними есть ребро (значение в матрице смежности равно 1). Если хотя бы одна пара вершин не соединена ребром, функция возвращает False. В противном случае возвращает True.

Метод brute\_force\_max\_clique(graph) ищет максимальную клику в графе методом полного перебора. Для каждого подмножества вершин (представленного битовой маской) проверяется, является ли оно кликой и имеет ли больше вершин, чем текущая максимальная клика. Возвращается максимальную клику в виде списка вершин.

**Сложность алгоритма полного перебора (brute-force):**

Предположим, что граф имеет n вершин. Алгоритм перебирает все подмножества вершин. Для каждого подмножества выполняется проверка на клику, что занимает O() времени. Таким образом, общая сложность алгоритма составляет **O( \* )**.

Этот метод неэффективен для больших графов из-за экспоненциальной сложности.

**2 точный метод решения – алгоритм Брона-Кербоша.**

Функция N(vertex, graph) возвращает список соседей данной вершины. Она принимает два параметра: vertex - номер вершины, для которой нужно найти соседей и graph - матрица смежности графа. Возвращается список вершин, с которыми vertex соединена ребром (значение в матрице смежности равно 1).

Рекурсивный алгоритм Брона-Кербоша bronk(r, p, x, graph) используется для поиска максимальной клики. Параметры: r - текущая клика (список вершин), p - кандидаты на добавление в клику (список вершин), x - исключенные вершины (список вершин) и graph - матрица смежности графа.

Алгоритм работает следующим образом: если p (кандидаты) и x (исключенные вершины) пусты, то мы нашли максимальную клику и выводим ее.

Для каждой вершины vertex из p: создается новая клика r\_new, добавляя vertex. После обновляется p\_new и x\_new: p\_new содержит вершины из p, которые соединены ребром с vertex, а x\_new содержит вершины из x, которые соединены ребром с vertex.

Рекурсивно вызывается bronk с новыми значениями r\_new, p\_new и x\_new. Удаляется vertex из p и добавляется в x.

**Сложность алгоритма Брона-Кербоша:**

Предположим, что граф имеет n вершин. В худшем случае алгоритм перебирает все подмножества вершин. Каждое подмножество проверяется на клику, что занимает O() времени. Таким образом, общая сложность алгоритма составляет **O( \* ).**

Этот метод также неэффективен для больших графов из-за экспоненциальной сложности*.*

**1 приближенный метод решения – метод локального поиска.**

Функция local\_search\_max\_clique(graph, max\_iterations=1000) ищет приближенную максимальную клику в графе с помощью метода локального поиска.

Параметры: graph - матрица смежности графа, max\_iterations -максимальное количество итераций (по умолчанию 1000).

Алгоритм работает следующим образом: начальная случайная клика current\_clique выбирается из 10 случайных вершин (или всех вершин, если их меньше). Создается копия best\_clique, которая будет хранить лучшую найденную клику.

В цикле выполняется следующее: вычисляется множество соседей neighbors для текущей клики current\_clique. Из neighbors исключаются вершины, уже присутствующие в текущей клике. Если neighbors пусто (нет соседей для улучшения), алгоритм завершается.

Выбирается случайная вершина new\_vertex из neighbors. Создается новая клика new\_clique, добавляя new\_vertex к текущей клике. Если размер new\_clique больше размера текущей клики current\_clique, обновляется current\_clique. Если размер new\_clique больше размера лучшей клики best\_clique, обновляется best\_clique.

Возвращается максимальная клика в виде списка вершин.

**Сложность метода локального поиска:**

Предположим, что граф имеет n вершин.В худшем случае алгоритм выполняет max\_iterations итераций.На каждой итерации:вычисление соседей занимает O() времени (перебор всех вершин).Выбор случайной вершины занимает O(1) времени.Таким образом, общая сложность алгоритма составляет O(max\_iterations \* ).

Этот метод более эффективен, чем полный перебор, но все равно может быть неэффективным для больших графов.

**2 приближенный метод решения – жадный алгоритм.**

Функция greedy\_max\_clique(graph) ищет клику с помощью жадного алгоритма.

Алгоритм начинает с пустой клики max\_clique и множества всех вершин vertices.

На каждом шаге: выбирается вершина v, которая имеет наибольшее пересечение с текущим множеством вершин vertices. Вершина v добавляется в max\_clique. Обновляется vertices, исключая вершины, которые не соединены ребром с v. Процесс продолжается до тех пор, пока vertices не станет пустым.

Возвращается максимальная клика в виде списка вершин.

**Сложность жадного алгоритма:**

Предположим, что граф имеет n вершин. На каждом шаге выбора вершины v, вычисление пересечения с vertices занимает O() времени (перебор всех вершин). Всего выполняется n шагов. Таким образом, общая сложность алгоритма составляет O().

Этот метод более эффективен, чем полный перебор, но все равно может быть неэффективным для больших графов.

**3 приближенный метод решения – жадный алгоритм.**

Функция wilf\_approx\_max\_clique(graph) реализует приближенный алгоритм Уилфа, предназначенный для поиска максимальной клики в графе.

Алгоритм начинает с пустой клики max\_clique и множества всех вершин vertices.

На каждом шаге: выбирается вершина vertex, которая имеет наибольшее количество соседей в графе. Вершина vertex добавляется в max\_clique. Обновляется множество vertices, исключая вершины, которые уже присутствуют в текущей клике. Процесс продолжается до тех пор, пока vertices не станет пустым.

Возвращается максимальная клика в виде списка вершин.

**Сложность алгоритма Уилфа:**

Предположим, что граф имеет n вершин. На каждом шаге выбора вершины vertex, вычисление количества соседей занимает O() времени (перебор всех вершин). Всего выполняется n шагов. Таким образом, общая сложность алгоритма составляет O().

Этот метод является приближенным и может быть более эффективным для больших графов по сравнению с полным перебором.

**Результаты работы программы:**

На рисунке 1 представлен результат работы программы.

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

Рисунок 1 – Результат работы программы.

**Листинг программы:**

Файл Algorithms.py

import random

def generate\_adjacency\_matrix(num\_vertices):

"""

Генерирует случайную матрицу смежности для графа с num\_vertices вершинами.

"""

matrix = [[random.randint(0, 1) for \_ in range(num\_vertices)] for \_ in range(num\_vertices)]

for i in range(num\_vertices):

matrix[i][i] = 0 # Нет петель (ребер из вершины в саму себя)

return matrix

def is\_clique(graph, vertices):

"""

Проверяет, является ли данное множество вершин кликой в графе.

"""

for u in vertices:

for v in vertices:

if u != v and not graph[u][v]:

return False

return True

def brute\_force\_max\_clique(graph):

"""

Ищет максимальную клику в графе методом полного перебора.

"""

n = len(graph)

max\_clique = []

for i in range(1, 2\*\*n):

vertices = [v for v in range(n) if (i & (1 << v)) != 0]

if is\_clique(graph, vertices) and len(vertices) > len(max\_clique):

max\_clique = vertices

return max\_clique

def N(vertex, graph):

"""

Возвращает список соседей данной вершины.

"""

return [i for i, val in enumerate(graph[vertex]) if val == 1]

def bronk(r, p, x, graph):

"""

Рекурсивный алгоритм Брона-Кербоша.

r: текущая клика

p: кандидаты на добавление в клику

x: исключенные вершины

graph: матрица смежности графа

"""

if not p and not x:

print(r) # Выводим максимальную клику

return

for vertex in p[:]:

r\_new = r + [vertex]

p\_new = [val for val in p if val in N(vertex, graph)] # p пересекается с N(vertex)

x\_new = [val for val in x if val in N(vertex, graph)] # x пересекается с N(vertex)

bronk(r\_new, p\_new, x\_new, graph)

p.remove(vertex)

x.append(vertex)

def local\_search\_max\_clique(graph, max\_iterations=1000):

"""

Ищет приближенную максимальную клику с помощью метода локального поиска.

"""

n = len(graph)

current\_clique = set(random.sample(range(n), min(10, n))) # Начальная случайная клика

best\_clique = current\_clique.copy()

for \_ in range(max\_iterations):

neighbors = set()

for v in current\_clique:

neighbors |= set(graph[v])

neighbors -= current\_clique

if not neighbors:

break # Нет соседей для улучшения

# Попробуем добавить новую вершину

new\_vertex = random.choice(list(neighbors))

new\_clique = current\_clique | {new\_vertex}

if len(new\_clique) > len(current\_clique):

current\_clique = new\_clique

if len(new\_clique) > len(best\_clique):

best\_clique = new\_clique

return list(best\_clique)

def greedy\_max\_clique(graph):

"""

Ищет приближенную максимальную клику с помощью жадного алгоритма.

"""

n = len(graph)

vertices = set(range(n))

max\_clique = set()

while vertices:

v = max(vertices, key=lambda x: len(vertices & set(graph[x])))

max\_clique.add(v)

vertices &= set(graph[v])

return list(max\_clique)

def wilf\_approx\_max\_clique(graph):

"""

Приближенный алгоритм Уилфа для поиска максимальной клики в графе.

"""

max\_clique = set()

vertices = set(graph.keys())

while vertices:

# Выбираем вершину с наибольшим числом соседей

vertex = max(vertices, key=lambda v: len(graph[v]))

max\_clique.add(vertex)

# Обновляем множество соседей

vertices -= graph[vertex]

return max\_clique

# Получаем количество вершин от пользователя

num\_vertices = int(input("Введите количество вершин в графе: "))

adj\_matrix = generate\_adjacency\_matrix(num\_vertices)

print("Сгенерированная матрица смежности:")

for row in adj\_matrix:

print(row)

print("Полный перебор - максимальная клика:", brute\_force\_max\_clique(adj\_matrix))

print("Алгоритм Брона-Кербоша:")

bronk([], list(range(num\_vertices)), [],adj\_matrix)

print("Приближенная максимальная клика - метод локального поиска:", local\_search\_max\_clique(adj\_matrix))

graph = {}

for i in range(num\_vertices):

graph[str(i)] = set(str(j) for j in range(num\_vertices) if adj\_matrix[i][j])

approx\_max\_clique = wilf\_approx\_max\_clique(graph)

print("Приближенная максимальная клика - жадный алгоритм:", greedy\_max\_clique(adj\_matrix))

print("Приближенная максимальная клика- алгоритм Уилфа:", approx\_max\_clique)