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**

**«Задача о самом длинном пути»**

**по курсу**

**«ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМОВ»**

Работу выполнил

Студент 36 группы

Воробьев А.Д.

Преподаватель:

Лапина О.Н.

Краснодар 2024

**Задача о самом длинном пути.** Дан граф *G*(*V*, *E*) и номера 2-х вершин А и В, а также число k.

- Задача оптимизации: Найти путь из А в В максимальной длины.

- Задача принятия решения: Существует ли путь из А в В длиной не менее k?

|  |  |
| --- | --- |
|  | Задача принятия решения |
|  | Задача оптимизации |

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

Определение класса NP задач:

Класс NP (Недетерминированная полиномиальная) 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 о принадлежности пути из А в В длиной не менее k, можно увидеть, что она относится к классу NP задач. Проверка правильности решения может быть выполнена следующим образом:

1. Проверка существования пути: Можно недетерминированно выбрать последовательность вершин и проверить, существует ли путь, соединяющий вершины А и В в графе G(V, E), используя данную последовательность вершин. Эта проверка может быть выполнена в полиномиальное время.
2. Проверка длины пути: Можно проверить, что длина пути, найденного в предыдущем шаге, не менее k. Это также может быть выполнено в полиномиальное время.

Таким образом, задача о принадлежности пути из А в В длиной не менее k относится к классу NP задач.

**Для задач оптимизации** описать двухэтапный недетерминированный процесс решения задач (см. Лекция NP-задачи). В ответ должен входить формат выхода недетерминированного шага, причем он должен включать в себя все элементы решения задачи. 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. Недетерминированный выбор пути:
2. Начинаем с вершины А.
3. Недетерминированно выбираем одну из соседних вершин и добавляем ее в текущий путь P.
4. Повторяем предыдущий шаг k-1 раз, каждый раз недетерминированно выбирая следующую соседнюю вершину и добавляя ее в путь P.
5. Завершаем процесс, когда достигаем вершины В или исчерпываем k шагов.

Формат выхода недетерминированного шага:

На каждом шаге выбора соседней вершины, записываем выбранную вершину в путь P.

1. Проверка оптимальности пути
   * 1. Проверяем, достигли ли мы вершины В. Если да, то P является потенциальным решением задачи.
     2. Если не достигли, возвращаемся к шагу 1 и повторяем процесс, выбирая другие недетерминированные варианты пути.

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

1. Проверяем, является ли P путем, то есть не содержит ли он повторяющихся вершин.
2. Проверяем, начинается ли P с вершины А и заканчивается ли на вершине В.
3. Проверяем, является ли длина P не меньше k.

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

**2. 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. Опишите соответствующее преобразование какой-либо NP-полной задачи к ЗАДАННОЙ. 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 доказательства принадлежности задачи к классу NP, необходимо показать, что для данной задачи существует верификатор, который может проверить правильность решения в полиномиальное время.

В данной задаче, верификатор может просто проверить, является ли данный путь из вершины А в вершину В длиной не менее k. Это можно сделать, пройдя по всем ребрам пути и проверив их длину, и затем сравнив длину пути с k. Верификатор может быть реализован за полиномиальное время от размера графа и k.

Таким образом, задача принадлежит классу NP.

Для доказательства NP-полноты или NP-трудности задачи, необходимо показать, что другая NP-полная задача может быть сведена к данной задаче. В данном случае, мы можем показать сводимость задачи о самом длинном пути к задаче о поиске гамильтонова пути.

Задача о поиске гамильтонова пути заключается в том, чтобы найти путь, проходящий через каждую вершину графа ровно один раз. Эта задача известна как NP-полная.

Для сводимости задачи о самом длинном пути к задаче о поиске гамильтонова пути, можно построить граф, в котором рёбра имеют вес 1, если они существуют в исходном графе, и вес 0, если их нет. Затем, найти гамильтонов путь в этом графе. Если длина найденного пути равна k, то существует путь из вершины А в вершину В длиной не менее k. Если же длина пути меньше k, то пути из А в В длиной не менее k не существует.

Таким образом, задача о самом длинном пути является NP-полной, так как она принадлежит классу NP и может быть сведена к NP-полной задаче о поиске гамильтонова пути.

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

Задача о самом длинном пути является NP-полной, но не имеет псевдополиномиального алгоритма для ее решения.

Исходная задача является NP-трудной, так как соответствующая ей задача принятия решения является NP-полной.

NP-полная задача может иметь полиномиальное время выполнения от размера входных данных, но не от числовых значений входных данных. В случае задачи о самом длинном пути, размер входных данных определяется количеством вершин и ребер в графе, а не числовыми значениями, такими как k.

Для решения задачи о самом длинном пути требуется перебрать все возможные пути в графе, чтобы найти путь максимальной длины. Количество возможных путей в графе может быть экспоненциальным от размера графа, поэтому алгоритмы для решения этой задачи обычно имеют экспоненциальную сложность.

Таким образом, задача о самом длинном пути не имеет псевдополиномиального алгоритма, который бы решал ее в полиномиальное время от числовых значений входных данных. Следовательно, задача о поиске самого длинного пути в графе является NP-трудной в сильном смысле.

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

Задача о самом длинном пути в графе является NP-полной задачей комбинаторной оптимизации. Определение самого длинного пути может быть сформулировано как задача оптимизации или задача принятия решения в зависимости от поставленной задачи.

В задаче оптимизации необходимо найти путь из вершины А в вершину В, имеющий максимальную длину. Формально, целевая функция задачи может быть определена как максимизация длины пути. Целью алгоритма является поиск оптимального пути, который обеспечивает максимальную длину. В случае NP-полной задачи, в общем случае, невозможно найти полиномиальный алгоритм, который решает эту задачу точно для всех графов. Поэтому для этой задачи может быть использованы приближенные алгоритмы с гарантией приближения.

В задаче принятия решения необходимо определить, существует ли путь из вершины А в вершину В длиной не менее k. В этом случае, задача сводится к проверке существования пути заданной длины k. Для этой задачи можно использовать различные алгоритмы обхода графа, такие как поиск в глубину или поиск в ширину, для проверки наличия пути заданной длины. Если такой путь существует, ответом будет "Да", в противном случае - "Нет". Для больших графов задача о самом длинном пути может быть вычислительно сложной, и использование приближенных алгоритмов может быть полезным для получения результатов в разумное время.

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

*Оценка сложности данного алгоритма O(2^n)*

Точный метод решения задачи о самом длинном пути состоит в полном переборе всех возможных вариантов и состоит из следующих шагов:

1. Создается пустой список, который будет содержать самый длинный найденный путь.
2. Генерируются все простые пути (пути без повторяющихся вершин) между стартовой и конечной вершинами в графе.
3. Для каждого пути выполняется следующее:
   * Проверяется, если длина текущего пути больше длины самого длинного пути, сохраненного в списке.
   * Если условие выполнено, то текущий путь становится самым длинным путем.
4. После обработки всех путей, самый длинный путь возвращается в качестве результата.

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

*Оценка сложности данного алгоритма O(n\*m)*

Данный алгоритм является эвристикой.

1. Создается список, который будет содержать путь от начальной к конечной вершине.
2. Устанавливается текущая вершина равная начальной вершине.
3. Пока текущая вершина не равна конечной вершине, выполняются следующие шаги:
   * Получаются все соседние вершины текущей вершины .
   * Если список пуст (текущая вершина не имеет соседей), то возвращается значение None, так как путь до конечной вершины невозможен.
   * Иначе, выбирается следующая вершина из списка, которая имеет наибольшую степень (наибольшее количество связанных вершин).
   * Удаляется ребро между текущей вершиной и выбранной следующей вершиной в графе.
   * Выбранная вершина добавляется в список пути.
   * Текущая вершина обновляется и становится равной выбранной вершине.
4. По завершении цикла, возвращается список, который содержит самый длинный путь от начальной к конечной вершине.

**3 Приближенный метод решения – Случайный алгоритм**

*Оценка сложности данного алгоритма O(iterations \* (n + m))*

1. Выполняется цикл, в каждой итерации:
   * Устанавливается текущая вершина равной начальной вершине.
   * Создается список, который будет содержать путь от начальной к конечной вершине.
   * Устанавливается переменная равная 0, которая будет хранить длину текущего пути.
   * В другом цикле выполняются следующие шаги, пока текущая вершина не равна конечной вершине:
     + Получаются все соседние вершины текущей вершины.
     + Если список пуст (текущая вершина не имеет соседей), цикл прерывается.
     + Иначе, случайным образом выбирается следующая вершина.
     + Удаляется ребро между текущей вершиной и выбранной следующей вершиной в графе.
     + Выбранная вершина добавляется в список пути.
     + Текущая вершина обновляется и становится равной выбранной вершине.
     + Длина пути увеличивается на 1.
   * Если длина текущего пути больше значения самого длинного пути, то в переменную лучшего пути записывается значение длины текущего.
   * Восстанавливается исходный граф, возвращая удаленные ребра для текущего пути. Это делается путем добавления ребра между каждой вершиной пути (кроме последней) и последней вершиной пути.
2. По завершении всех итераций, возвращается лучший найденный путь.

**4 Приближенный метод решения** – **Алгоритм аппроксимации.**

*Оценка сложности данного алгоритма O(n \* m)*

1. Инициализируется список с начальной вершиной. Этот список будет содержать путь от начальной к конечной вершине.
2. Устанавливается текущая вершина равной начальной вершине.
3. В цикле выполняются следующие шаги, пока текущая вершина не равна конечной вершине:
   * Получаются все соседние вершины текущей вершины.
   * Если список пуст (текущая вершина не имеет соседей), алгоритм возвращает None, что означает, что путь до конечной вершины невозможен.
   * Иначе, случайным образом выбирается следующая вершина.
   * Удаляется ребро между текущей вершиной и выбранной следующей вершиной в графе.
   * Выбранная вершина добавляется в список пути.
   * Текущая вершина обновляется и становится равной выбранной вершине.
4. По завершении цикла, возвращается список, который содержит приближенный самый длинный путь от начальной к конечной вершине.

**Результаты работы программы:**

На рисунках 1, 2, 3, 4, 5 и 6 представлен результат работы программы -найденный самый длинный путь, отрисовка графа, матрица смежности графа.

Точное решение выделено зеленым цветом на графе.

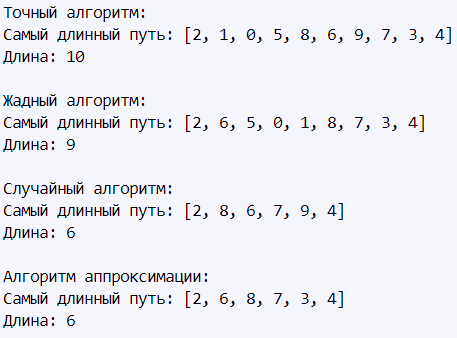


Рисунок 1 – Результат работы программы (Пример 1).

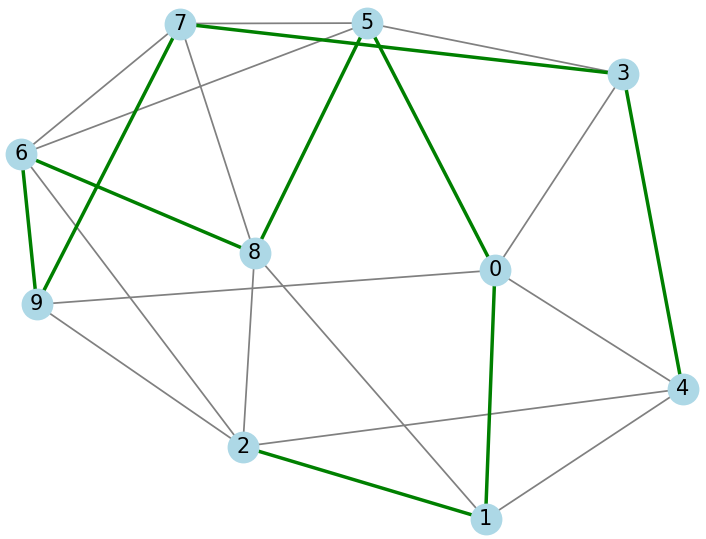


Рисунок 2 – сгенерированный граф с решением задачи (Пример 1).

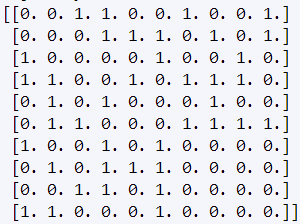


Рисунок 3 – матрица смежности сгенерированного графа (Пример 1).

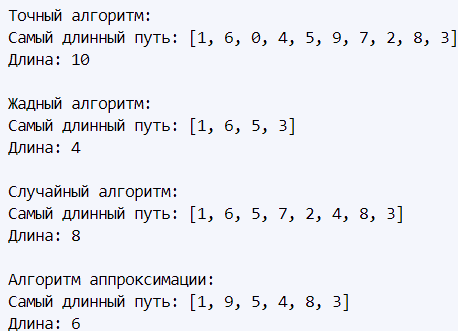


Рисунок 4 – Результат работы программы (Пример 2).

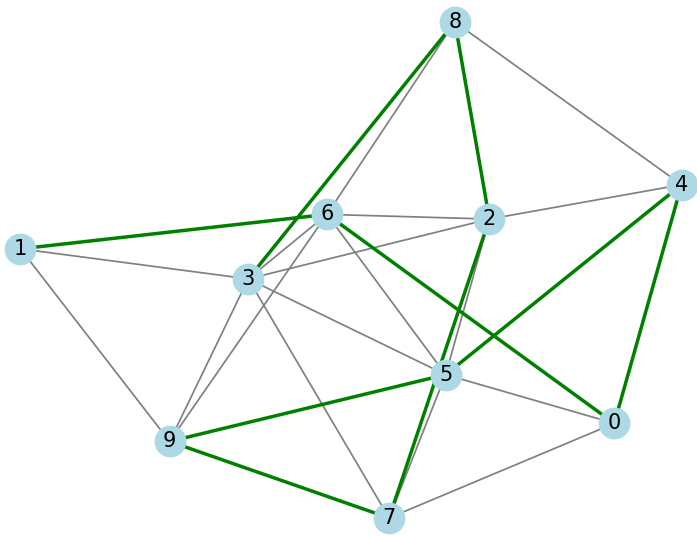


Рисунок 5 – сгенерированный граф с решением задачи (Пример 2).

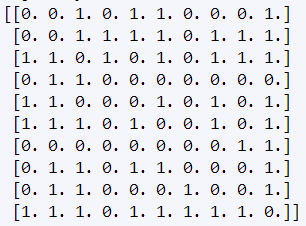


Рисунок 6 – матрица смежности сгенерированного графа (Пример 2).

**Текст программы:**

Файл main.py:

import networkx as nx

import random

import matplotlib.pyplot as plt

import numpy as np

def create\_graph(num\_vertices, edge\_probability):

    graph = nx.Graph()

    for i in range(num\_vertices):

        for j in range(i + 1, num\_vertices):

            if random.random() <= edge\_probability:

                graph.add\_edge(i, j)

    return graph

num\_vertices = 10

edge\_probability = 0.5

graph = create\_graph(num\_vertices, edge\_probability)

def get\_adjacency\_matrix(graph):

    num\_vertices = len(graph.nodes)

    adjacency\_matrix = np.zeros((num\_vertices, num\_vertices))

    for edge in graph.edges:

        adjacency\_matrix[edge[0], edge[1]] = 1

        adjacency\_matrix[edge[1], edge[0]] = 1

    return adjacency\_matrix

# Вывод матрицы сложности графа

adjacency\_matrix = get\_adjacency\_matrix(graph)

print("Adjacency Matrix:")

print(adjacency\_matrix)

def longest\_path\_exact(graph, start, end):

    longest = []

    paths = nx.all\_simple\_paths(graph, start, end)

    for path in paths:

        if len(path) > len(longest):

            longest = path

    return longest

def longest\_path\_greedy(graph, start, end):

    path = [start]

    current = start

    while current != end:

        neighbors = list(graph[current])

        if not neighbors:

            return None

        next\_vertex = max(neighbors, key=lambda v: len(graph[v]))

        graph.remove\_edge(current, next\_vertex)

        path.append(next\_vertex)

        current = next\_vertex

    return path

def longest\_path\_random(graph, start, end, iterations=1000):

    best\_path = None

    best\_length = 0

    for \_ in range(iterations):

        current = start

        path = [current]

        length = 0

        while current != end:

            neighbors = list(graph[current])

            if not neighbors:

                break

            next\_vertex = random.choice(neighbors)

            graph.remove\_edge(current, next\_vertex)

            path.append(next\_vertex)

            current = next\_vertex

            length += 1

        if length > best\_length:

            best\_path = path

            best\_length = length

        # Восстановление исходного графа

        for vertex in path[:-1]:

            graph.add\_edge(vertex, path[-1])

    return best\_path

def longest\_path\_approximation(graph, start, end):

    path = [start]

    current = start

    while current != end:

        neighbors = list(graph[current])

        if not neighbors:

            return None

        next\_vertex = random.choice(neighbors)

        graph.remove\_edge(current, next\_vertex)

        path.append(next\_vertex)

        current = next\_vertex

    return path

num\_vertices = 10

edge\_probability = 0.5

graph = create\_graph(num\_vertices, edge\_probability)

start = random.randint(0, num\_vertices - 1)

end = random.randint(0, num\_vertices - 1)

while end == start:

    end = random.randint(0, num\_vertices - 1)

print("Граф:")

print(graph.edges())

print("\nТочный алгоритм:")

exact\_path = longest\_path\_exact(graph.copy(), start, end)

if exact\_path:

    print("Самый длинный путь:", exact\_path)

    print("Длина:", len(exact\_path))

else:

    print("Путь не найден.")

print("\nЖадный алгоритм:")

greedy\_path = longest\_path\_greedy(graph.copy(), start, end)

if greedy\_path:

    print("Самый длинный путь:", greedy\_path)

    print("Длина:", len(greedy\_path))

else:

    print("Путь не найден.")

print("\nСлучайный алгоритм:")

random\_path = longest\_path\_random(graph.copy(), start, end)

if random\_path:

    print("Самый длинный путь:", random\_path)

    print("Длина:", len(random\_path))

else:

    print("Путь не найден.")

print("\nАлгоритм аппроксимации:")

approx\_path = longest\_path\_approximation(graph.copy(), start, end)

if approx\_path:

    print("Самый длинный путь:", approx\_path)

    print("Длина:", len(approx\_path))

else:

    print("Путь не найден.")

# Визуализация графа

pos = nx.spring\_layout(graph)

nx.draw(graph, pos, with\_labels=True, node\_color='lightblue', edge\_color='gray')

# Выделение пути, полученного с помощью точного алгоритма

if exact\_path:

    for i in range(len(exact\_path) - 1):

        nx.draw\_networkx\_edges(graph, pos, edgelist=[(exact\_path[i], exact\_path[i + 1])], edge\_color='green', width=2)

plt.show()