**Доклад: Представления графов в компьютере**

**по предмету Комбинаторика и теория графов**

**Выполнил обучающийся НИТУ МИСИС**

**Группа БИВТ-23-1**

**Нестеров Никита Михайлович**

**Ссылка на реализацию:**

**[GitHub](https://github.com/akayooo/KITG)**

[**1. Формальная постановка задачи**](#_sckw3y6i9gqv) **3**

[**2. Теоретическое описание алгоритма и его характеристики**](#_7oik8nxo2ats) **4**

[**3. Характеристики алгоритма**](#_95mv1k99mm7y) **5**

[**4. Сравнительный анализ представлений графов**](#_xzggb536itfm) **7**

[**5. Перечень инструментов для реализации**](#_83r79ml6n30d) **9**

[**6. Библиотеки и фреймворки**](#_hu9p72in1y2f) **10**

[**7. Инструменты для отладки и тестирования**](#_ndax9to7eifd) **10**

[**7. Инструменты для визуализации**](#_wl77u2egw5s1) **11**

[**8. Инструменты для анализа производительности**](#_nhholh2hdgjr) **11**

[**8. Описание реализации и процесса тестирования представлений графов на Python**](#_mjeejsmq4llu) **12**

[**9. Процесс тестирования**](#_cx6piumpe35f) **15**

[**10. Анализ временной сложности реализованных операций**](#_10m5lo521yum) **17**

[**11. Практические результаты**](#_mtt0ruk8zr9x) **18**

[**12. Заключение**](#_r2szbs2jij6b) **19**

# 1. Формальная постановка задачи

Задача:

Дан граф \( G = (V, E) \), где \( V \) — множество вершин, а \( E \) — множество ребер. Требуется выбрать и реализовать наиболее эффективное представление графа в компьютере, учитывая различные характеристики графа (плотность, количество вершин и ребер, тип графа — ориентированный или неориентированный).

Формально:

- Дано: Граф \( G = (V, E) \).

- Требуется:

- Выбрать и реализовать одно из представлений графа: список смежности, матрица смежности, список ребер или матрица инцидентности.

- Оценить эффективность выбранного представления с точки зрения временной и пространственной сложности.

Результат:

- Реализация выбранного представления графа на выбранном языке программирования.

- Анализ эффективности представления с точки зрения временной и пространственной сложности.

# 2. Теоретическое описание алгоритма и его характеристики

Теоретическое описание

Представления графов в компьютере:

1. Список смежности:

- Описание: Список смежности представляет собой массив списков, где каждый элемент массива соответствует вершине графа, а список, связанный с этой вершиной, содержит все вершины, смежные с ней.

- Пример: Для графа ( G = (V, E) ) с вершинами ( V = {A, B, C} ) и ребрами ( E = {(A, B), (B, C), (C, A)} ), список смежности будет выглядеть так:

```

A -> [B]

B -> [C]

C -> [A]

```

2. Матрица смежности:

- Описание: Матрица смежности представляет собой квадратную матрицу размером ( |V| times |V| ), где ( |V| ) — количество вершин. Элемент матрицы ( M[i][j] ) равен 1, если существует ребро между вершинами ( i ) и ( j ), и 0 в противном случае.

- Пример: Для того же графа ( G ), матрица смежности будет выглядеть так:

```

A B C

A 0 1 0

B 0 0 1

C 1 0 0

```

3. Список ребер:

- Описание: Список ребер представляет собой список пар вершин, где каждая пара соответствует ребру графа.

- Пример: Для графа ( G ), список ребер будет выглядеть так:

```

[(A, B), (B, C), (C, A)]

```

4. Матрица инцидентности:

- Описание: Матрица инцидентности представляет собой матрицу размером ( |V| times |E| ), где ( |V| ) — количество вершин, а ( |E| ) — количество ребер. Элемент матрицы ( M[i][j] ) равен 1, если вершина ( i ) инцидентна ребру ( j ), и 0 в противном случае.

- Пример: Для графа ( G ), матрица инцидентности будет выглядеть так:

```

e1 e2 e3

A 1 0 0

B 0 1 0

C 0 0 1

```

# 3. Характеристики алгоритма

Временная сложность:

- Список смежности:

- Добавление ребра: ( O(1) )

- Проверка наличия ребра: ( O(k) ), где ( k ) — количество соседей вершины.

- Обход всех соседей вершины: ( O(k) )

- Матрица смежности:

- Добавление ребра: ( O(1) )

- Проверка наличия ребра: ( O(1) )

- Обход всех соседей вершины: ( O(V) ), где ( V ) — количество вершин.

- Список ребер:

- Добавление ребра: ( O(1) )

- Проверка наличия ребра: ( O(E) ), где ( E ) — количество ребер.

- Обход всех соседей вершины: ( O(E) )

- Матрица инцидентности:

- Добавление ребра: ( O(1) )

- Проверка наличия ребра: ( O(E) )

- Обход всех соседей вершины: ( O(E) )

Пространственная сложность:

- Список смежности:\*\* \( O(V + E) \)

- Матрица смежности:\*\* \( O(V^2) \)

- Список ребер:\*\* \( O(E) \)

- Матрица инцидентности:\*\* \( O(V \times E) \)

Преимущества и недостатки:

- Список смежности:

- Преимущества: Эффективен для разреженных графов (где \( E \ll V^2 \)).

- Недостатки: Проверка наличия ребра может быть медленной.

- Матрица смежности:

- Преимущества: Быстрая проверка наличия ребра.

- Недостатки: Неэффективен для разреженных графов.

- Список ребер:

- Преимущества: Простота реализации.

- Недостатки: Медленная проверка наличия ребра.

- Матрица инцидентности:

- Преимущества: Явное представление инцидентности вершин и ребер.

- Недостатки: Неэффективен для разреженных графов.

# 4. Сравнительный анализ представлений графов

Список смежности vs. Матрица смежности

Список смежности:

- Временная сложность: ( O(1) ) для добавления ребра, ( O(k) ) для проверки наличия ребра, ( O(k) ) для обхода соседей.

- Пространственная сложность: ( O(V + E) ).

- Устойчивость: Эффективен для разреженных графов.

- Преимущества: Эффективен для разреженных графов.

- Недостатки: Проверка наличия ребра может быть медленной.

Матрица смежности:

- Временная сложность: ( O(1) ) для добавления ребра, ( O(1) ) для проверки наличия ребра, ( O(V) ) для обхода соседей.

- Пространственная сложность: ( O(V^2) ).

- Устойчивость: Эффективен для плотных графов.

- Преимущества: Быстрая проверка наличия ребра.

- Недостатки: Неэффективен для разреженных графов.

Список смежности vs. Список ребер

Список смежности:

- Временная сложность: ( O(1) ) для добавления ребра, ( O(k) ) для проверки наличия ребра, ( O(k) ) для обхода соседей.

- Пространственная сложность: ( O(V + E) ).

- Устойчивость: Эффективен для разреженных графов.

- Преимущества: Эффективен для разреженных графов.

- Недостатки: Проверка наличия ребра может быть медленной.

\*Список ребер:

- Временная сложность: ( O(1) ) для добавления ребра, ( O(E) ) для проверки наличия ребра, ( O(E) ) для обхода соседей.

- Пространственная сложность: ( O(E) ).

- Устойчивость: Простота реализации.

- Преимущества: Простота реализации.

- Недостатки: Медленная проверка наличия ребра.

Матрица смежности vs. Матрица инцидентности

Матрица смежности:

- Временная сложность: ( O(1) ) для добавления ребра, ( O(1) ) для проверки наличия ребра, ( O(V) ) для обхода соседей.

- Пространственная сложность: \( O(V^2) \).

- Устойчивость: Эффективен для плотных графов.

- Преимущества: Быстрая проверка наличия ребра.

- Недостатки: Неэффективен для разреженных графов.

Матрица инцидентности:

- Временная сложность: ( O(1) ) для добавления ребра, ( O(E) ) для проверки наличия ребра, ( O(E) ) для обхода соседей.

- Пространственная сложность: ( O(V times E) ).

- Устойчивость: Явное представление инцидентности вершин и ребер.

- Преимущества: Явное представление инцидентности вершин и ребер.

- Недостатки: Неэффективен для разреженных графов.

# 

# 5. Перечень инструментов для реализации

Языки программирования

- Python:

- Преимущества: Простой синтаксис, богатые библиотеки для работы с графами.

- Пример: Можно использовать встроенные структуры данных и функции для реализации графов.

- Java:

- Преимущества: Объектно-ориентированный подход, широко используется в промышленности.

- Пример: Можно использовать классы и методы для работы с графами.

- C/C++:

- Преимущества: Близость к аппаратной части, высокая производительность.

- Пример: Можно реализовать графы с использованием указателей и динамического выделения памяти.

- JavaScript:

- Преимущества: Подходит для веб-разработки, интерактивные визуализации.

- Пример: Можно использовать массивы и функции для реализации графов.

- Ruby:

- Преимущества: Чистый и выразительный синтаксис, удобный для быстрой разработки.

- Пример: Можно использовать массивы и методы для реализации графов.

# 6. Библиотеки и фреймворки

- Python:

- networkx: Библиотека для работы с графами, предоставляет множество функций для анализа и визуализации графов.

- numpy: Библиотека для работы с массивами, может быть полезна для быстрой обработки данных.

- Java:

- JGraphT: Библиотека для работы с графами, предоставляет множество алгоритмов и структур данных для графов.

- java.util.ArrayList: Класс для работы с динамическими массивами.

- C++:

- Boost.Graph: Библиотека для работы с графами, предоставляет множество алгоритмов и структур данных для графов.

- std::vector: Класс для работы с динамическими массивами.

- JavaScript:

- D3.js: Библиотека для визуализации данных, может быть полезна для визуализации графов.

- Graphology: Библиотека для работы с графами, предоставляет множество функций для анализа и визуализации графов.

# 7. Инструменты для отладки и тестирования

- Отладчики:

- GDB (GNU Debugger): Для C/C++.

- Visual Studio Debugger: Для C#, C++, Java и других языков.

- PyCharm Debugger: Для Python.

- Chrome DevTools: Для JavaScript.

- Тестовые фреймворки:\*\*

- JUnit: Для Java.

- pytest: Для Python.

- Google Test: Для C++.

- Mocha/Chai: Для JavaScript.

# 7. Инструменты для визуализации

- Python:

- Matplotlib: Библиотека для построения графиков и визуализации данных.

- Plotly: Интерактивная библиотека для визуализации данных.

- JavaScript:

- D3.js: Библиотека для визуализации данных, подходит для создания интерактивных визуализаций.

- Chart.js: Простая библиотека для создания графиков.

- Java:

- JavaFX: Фреймворк для создания графического интерфейса и визуализации данных.

- JFreeChart: Библиотека для создания графиков и диаграмм.

# 8. Инструменты для анализа производительности

- Профилировщики:

- cProfile: Для Python.

- VisualVM: Для Java.

- Valgrind: Для C/C++.

- Chrome DevTools Performance: Для JavaScript.

- Анализаторы кода:

- SonarQube: Инструмент для анализа качества кода.

- Pylint: Для Python.

- PMD: Для Java.

# 8. Описание реализации и процесса тестирования представлений графов на Python

Реализация представлений графов на Python

Шаг 1: Реализация списка смежности

```python

class Graph:

def \_\_init\_\_(self):

self.adj\_list = {}

def add\_edge(self, u, v):

if u not in self.adj\_list:

self.adj\_list[u] = []

if v not in self.adj\_list:

self.adj\_list[v] = []

self.adj\_list[u].append(v)

self.adj\_list[v].append(u)

def print\_graph(self):

for vertex in self.adj\_list:

print(vertex, "->", self.adj\_list[vertex])

# Пример использования

g = Graph()

g.add\_edge('A', 'B')

g.add\_edge('B', 'C')

g.add\_edge('C', 'A')

g.print\_graph()

```

Шаг 2: Реализация матрицы смежности

```python

class Graph:

def \_\_init\_\_(self, num\_vertices):

self.num\_vertices = num\_vertices

self.adj\_matrix = [[0] \* num\_vertices for \_ in range(num\_vertices)]

def add\_edge(self, u, v):

self.adj\_matrix[u][v] = 1

self.adj\_matrix[v][u] = 1

def print\_graph(self):

for row in self.adj\_matrix:

print(row)

# Пример использования

g = Graph(3)

g.add\_edge(0, 1)

g.add\_edge(1, 2)

g.add\_edge(2, 0)

g.print\_graph()

```

Шаг 3: Реализация списка ребер

```python

class Graph:

def \_\_init\_\_(self):

self.edges = []

def add\_edge(self, u, v):

self.edges.append((u, v))

def print\_graph(self):

for edge in self.edges:

print(edge)

# Пример использования

g = Graph()

g.add\_edge('A', 'B')

g.add\_edge('B', 'C')

g.add\_edge('C', 'A')

g.print\_graph()

```

Шаг 4: Реализация матрицы инцидентности

```python

class Graph:

def \_\_init\_\_(self, num\_vertices, num\_edges):

self.num\_vertices = num\_vertices

self.num\_edges = num\_edges

self.inc\_matrix = [[0] \* num\_edges for \_ in range(num\_vertices)]

def add\_edge(self, u, v, edge\_index):

self.inc\_matrix[u][edge\_index] = 1

self.inc\_matrix[v][edge\_index] = 1

def print\_graph(self):

for row in self.inc\_matrix:

print(row)

# Пример использования

g = Graph(3, 3)

g.add\_edge(0, 1, 0)

g.add\_edge(1, 2, 1)

g.add\_edge(2, 0, 2)

g.print\_graph()

```

# 9. Процесс тестирования

1. Ручное тестирование

Можно вручную проверить работу алгоритма на нескольких примерах:

```python

g1 = Graph()

g1.add\_edge('A', 'B')

g1.add\_edge('B', 'C')

g1.add\_edge('C', 'A')

g1.print\_graph()

g2 = Graph(3)

g2.add\_edge(0, 1)

g2.add\_edge(1, 2)

g2.add\_edge(2, 0)

g2.print\_graph()

g3 = Graph()

g3.add\_edge('A', 'B')

g3.add\_edge('B', 'C')

g3.add\_edge('C', 'A')

g3.print\_graph()

g4 = Graph(3, 3)

g4.add\_edge(0, 1, 0)

g4.add\_edge(1, 2, 1)

g4.add\_edge(2, 0, 2)

g4.print\_graph()

```

2. Автоматизированные тесты с использованием pytest

Для более систематического тестирования можно использовать фреймворк pytest:

```python

import pytest

def test\_adj\_list():

g = Graph()

g.add\_edge('A', 'B')

g.add\_edge('B', 'C')

g.add\_edge('C', 'A')

assert g.adj\_list == {'A': ['B', 'C'], 'B': ['A', 'C'], 'C': ['B', 'A']}

def test\_adj\_matrix():

g = Graph(3)

g.add\_edge(0, 1)

g.add\_edge(1, 2)

g.add\_edge(2, 0)

assert g.adj\_matrix == [[0, 1, 1], [1, 0, 1], [1, 1, 0]]

def test\_edge\_list():

g = Graph()

g.add\_edge('A', 'B')

g.add\_edge('B', 'C')

g.add\_edge('C', 'A')

assert g.edges == [('A', 'B'), ('B', 'C'), ('C', 'A')]

def test\_inc\_matrix():

g = Graph(3, 3)

g.add\_edge(0, 1, 0)

g.add\_edge(1, 2, 1)

g.add\_edge(2, 0, 2)

assert g.inc\_matrix == [[1, 0, 1], [1, 1, 0], [0, 1, 1]]

if \_\_name\_\_ == "\_\_main\_\_":

pytest.main()

```

3. Профилирование производительности

Для анализа производительности можно использовать модуль cProfile:

```python

import cProfile

g = Graph()

g.add\_edge('A', 'B')

g.add\_edge('B', 'C')

g.add\_edge('C', 'A')

cProfile.run('g.print\_graph()')

```

Это позволит увидеть, сколько времени затрачивается на каждую функцию и поможет выявить узкие места в реализации.

# 10. Анализ временной сложности реализованных операций

Теоретическая временная сложность

- \*\*Список смежности:\*\*

- Добавление ребра: \( O(1) \)

- Проверка наличия ребра: \( O(k) \)

- Обход всех соседей вершины: \( O(k) \)

- Матрица смежности:

- Добавление ребра: \( O(1) \)

- Проверка наличия ребра: \( O(1) \)

- Обход всех соседей вершины: \( O(V) \)

- Список ребер:

- Добавление ребра: \( O(1) \)

- Проверка наличия ребра: \( O(E) \)

- Обход всех соседей вершины: \( O(E) \)

- Матрица инцидентности:

- Добавление ребра: \( O(1) \)

- Проверка наличия ребра: \( O(E) \)

- Обход всех соседей вершины: \( O(E) \)

# 11. Практические результаты

Для анализа практической производительности можно использовать профилирование с помощью cProfile. Пример:

```python

import cProfile

g = Graph()

g.add\_edge('A', 'B')

g.add\_edge('B', 'C')

g.add\_edge('C', 'A')

cProfile.run('g.print\_graph()')

```

Результаты профилирования покажут, сколько времени затрачивается на каждую функцию, что позволит сравнить практические результаты с теоретическими оценками.

# 12. Заключение

Выбор представления графа в компьютере зависит от конкретных требований задачи, таких как плотность графа, количество вершин и ребер, тип графа (ориентированный или неориентированный) и другие факторы. Список смежности эффективен для разреженных графов, матрица смежности — для плотных графов, список ребер — для простых задач, а матрица инцидентности — для явной инцидентности вершин и ребер. Выбор оптимального представления позволяет повысить эффективность работы с графами в компьютере.