МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО АВТОНОМНОГО ОБРАЗОВАТЕЛЬНОГО УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ

**«Национальный исследовательский технологический университет «МИСИС»**

**Институт Компьютерных Наук**

**Отчет**

**Задача "объединить-найти". Алгоритмы решения.**

**По курсу:** Комбинаторика и теория графов

**Ссылка на репозиторий:**

<https://github.com/aegon-7n/combinatorics-and-graph.git>

Трушков Глеб Викторович

Группа БИВТ-23-6

**Содержание:**

1. Формальная постановка задачи
2. Теоретическое описание алгоритма и его характеристики
3. Сравнительный анализ с аналогичными алгоритмами
4. Перечень инструментов, используемых для реализации
5. Описание реализации и процесса тестирования

**1. Формальная постановка задачи**

**Задача:** Реализация структуры данных **Union-Find** (с поддержкой объединения и нахождения) для эффективного решения задач, где необходимо поддерживать и манипулировать группами элементов, например, для поиска компонент связности в графе.

**Условия задачи:**

* Необходимо поддерживать два основных типа операций:
  1. **Find(x)** — нахождение представителя элемента x.
  2. **Union(x, y)** — объединение двух компонентов, содержащих элементы x и y.
* Операции должны выполняться эффективно, с использованием оптимизаций:
  1. **Сжатие пути** при выполнении операции **Find**.
  2. **Объединение по рангу** при выполнении операции **Union**.

**Входные данные:**

* Множество элементов, каждый из которых является своим собственным представителем.
* Последовательность операций объединения и нахождения.

**Выходные данные:**

* Результат каждой операции **Find** — представитель (корень) множества, к которому принадлежит элемент.
* После операции **Union** — объединение двух множеств.

**2. Теоретическое описание алгоритма и его характеристики**

**Описание алгоритма:**

Алгоритм Union-Find использует два ключевых подхода:

1. **Сжатие пути (Path Compression)**: при нахождении представителя компоненты для каждого элемента в пути мы прямо указываем на корень, что ускоряет последующие операции.
2. **Объединение по рангу (Union by Rank)**: при объединении двух компонентов мы прикрепляем дерево меньшей глубины к дереву большей глубины, что предотвращает рост дерева.

Алгоритм состоит из двух основных операций:

* **Find(x)**: Находит корень компоненты, к которой принадлежит элемент x, и сжимает путь к корню.
* **Union(x, y)**: Объединяет две компоненты, содержащие элементы x и y, по принципу объединения деревьев по глубине.

**Характеристики алгоритма:**

* **Временная сложность**:
  + Амортизированная сложность операций **Find** и **Union** — **O(α(n))**, где α — это обратная функция от функции Аккермана, которая растет очень медленно и на практике почти константна.
* **Пространственная сложность**: **O(n)** для хранения массива родителей и рангов.

**3. Сравнительный анализ с аналогичными алгоритмами**

| **Критерий** | **Union-Find (с оптимизациями)** | **Обычные методы (например, поиск в глубину)** |
| --- | --- | --- |
| **Временная сложность** | O(α(n)) для **Find** и **Union** | O(n) для поиска в глубину для каждого запроса |
| **Подход** | Сжатие пути и объединение по рангу | Поиск всех путей с возможностью повторного поиска |
| **Сложность реализации** | Простая, но требует внимательности к оптимизациям | Простая, но менее эффективная для больших наборов данных |
| **Применимость** | Эффективен для динамических задач с множеством объединений и запросов | Удобен для задач с малыми наборами данных |

**Вывод:** Алгоритм Union-Find с оптимизациями значительно превосходит другие методы для задач с множеством объединений и запросов на нахождение представителя.

**4. Перечень инструментов, используемых для реализации**

**Для реализации использовались следующие инструменты:**

* **Языки программирования:**
  + **Python 3.9+** — для быстрой разработки и тестирования.
* **Среда разработки:**
  + **Visual Studio Code** — удобный редактор для написания кода.
  + **PyCharm** — альтернатива для работы с Python.
* **Библиотеки:**
  + **pytest** — для тестирования алгоритма.
* **Система контроля версий:**
  + **Git** — для управления версиями.
* **Текстовый редактор:**
  + **Notepad++** — для подготовки и редактирования входных данных.

**5. Описание реализации и процесса тестирования**

**Реализация алгоритма:**

Код алгоритма реализован в файле union\_find.py и включает в себя класс UnionFind, который реализует операции **Find** и **Union**, а также оптимизации с использованием сжатия пути и объединения по рангу.

Основные компоненты:

1. **Класс UnionFind:**
   * Метод find(x): выполняет поиск представителя компоненты с сжатием пути.
   * Метод union(x, y): выполняет объединение двух компонент.
   * Метод connected(x, y): проверяет, принадлежат ли два элемента одной компоненте.

**Пример кода:**

class UnionFind:

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

self.parent = list(range(n))

self.rank = [0] \* n

def find(self, x):

if self.parent[x] != x:

self.parent[x] = self.find(self.parent[x])

return self.parent[x]

def union(self, x, y):

rootX = self.find(x)

rootY = self.find(y)

if rootX != rootY:

if self.rank[rootX] > self.rank[rootY]:

self.parent[rootY] = rootX

elif self.rank[rootX] < self.rank[rootY]:

self.parent[rootX] = rootY

else:

self.parent[rootY] = rootX

self.rank[rootX] += 1

def connected(self, x, y):

return self.find(x) == self.find(y)

**Процесс тестирования:**

Тестирование проводилось с использованием библиотеки pytest. Для проверки корректности работы алгоритма были подготовлены следующие тесты:

1. **Пустой граф**: Проверяется отсутствие связей между элементами.
2. **Граф с одним ребром**: Проверяется корректность объединения двух элементов.
3. **Сложные графы**: Тестируются графы с несколькими объединениями и запросами на нахождение.
4. **Большие графы**: Проверяется производительность на графах с большим количеством элементов.

**Пример теста с использованием pytest:**

import pytest

from union\_find import UnionFind

def test\_initialization():

uf = UnionFind(5)

for i in range(5):

assert uf.find(i) == i

def test\_union\_and\_find():

uf = UnionFind(5)

uf.union(1, 2)

assert uf.find(1) == uf.find(2)

uf.union(3, 4)

assert uf.find(3) == uf.find(4)

assert uf.find(1) != uf.find(3)

def test\_multiple\_unions():

uf = UnionFind(5)

uf.union(1, 2)

uf.union(2, 3)

uf.union(3, 4)

assert uf.find(1) == uf.find(4)

assert uf.find(2) == uf.find(4)

assert uf.find(3) == uf.find(4)

if \_\_name\_\_ == '\_\_main\_\_':

pytest.main()

**Запуск тестов**:  
Для запуска тестов необходимо выполнить команду:

pytest test\_union\_find.py

**6. Преимущества реализации на Python**

**Python**:

* Быстрая разработка и тестирование.
* Легкость в понимании и модификации к

ода.

* Подходит для небольших и средних графов.
* Подходит для работы с большими графами.

**7. Заключение**

Реализация алгоритма **Union-Find** с оптимизациями на языке Python показала свою эффективность для решения задач, связанных с динамическими компонентами связности. Алгоритм с применением сжатия пути и объединения по рангу значительно ускоряет выполнение операций, особенно при большом числе запросов.

Реализация на Python позволяет быстро разрабатывать и тестировать алгоритм, а для более масштабных задач можно использовать C++ для улучшения производительности.