МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

Федеральное государственное автономное образовательное

учреждение высшего образования

«Национальный исследовательский технологический
университет «МИСИС»

Комбинаторика и теория графов

Доклад по теме «Задача «объединить-найти». Алгоритмы решения.»

Выполнила:

Студентка БИВТ 23-7

Абрамова Дарья

https://github.com/DashaAbramova/combinatorics.git ,

Проверил:

Зайцев В. С.

Москва

2024

Оглавление

1. Формальная постановка задачи

Задача: Необходимо реализовать структуру данных для эффективного решения задачи динамического объединения и поиска, когда нужно быстро определять, принадлежат ли два элемента одной компоненте (множества) и объединять такие компоненты.

Это можно использовать, например, в задачах, связанных с представлением графов, для поиска компонент связности, или при решении задач по нахождению связных компонент в динамических графах.

Задача заключается в поддержке двух основных операций:

- Find: определить, к какому множеству принадлежит элемент.
- Union: объединить два множества в одно.

Алгоритм "Объединить-Найти" используется для эффективного решения этих задач.

2. Теоретическое описание алгоритма и его характеристики

Алгоритм Union-Find (или Disjoint Set Union, DSU) представляет собой структуру данных, которая поддерживает операции объединения и поиска.

Основные оптимизации, используемые в алгоритме:

- Путь сжатия (Path Compression): когда выполняется операция Find, пути всех предков текущего элемента сжимаются, то есть каждый узел в пути от элемента к корню указывает сразу на корень. Это ускоряет дальнейшие операции поиска.
- Объединение по рангу (Union by Rank): при объединении двух деревьев с различной высотой (рангом) присоединять дерево с меньшей высотой к дереву с большей высотой. Это помогает поддерживать сбалансированность деревьев и уменьшает глубину.

- Временная сложность: O(α(n)), где α(n) — очень медленно растущая функция, известная как обратная функция Аккермана. На практике она не превышает 5 для разумных значений n. Роберт Тарьян доказал в 1975 г. замечательный факт: время работы как Find, так и Union на лесе размера N есть O(α(N)). Под α(N) в математике обозначается обратная функция Аккермана, то есть, функция, обратная для f(N) = A(N, N). Функция Аккермана A(N, M) известна тем, что у нее колоссальная скорость роста. К примеру, A(4, 4) = 22265536-3, это число поистине огромно. Вообще, для всех мыслимых практических значений N обратная функция Аккермана от него не превысит 5. Поэтому её можно принять за константу и считать O(α(N)) ≅ O(1).

- Пространственная сложность: O(n), где n — количество элементов. Когда я искала материал, я нашла информацию, что можно присоединять ветки рандомно, не сравнивая их по рангу. Это еще больше ускоряет время работы программы. Но я не стала это реализовывать, так как нам необходим классический алгоритм Union – find.

Union-Find активно применяется в задачах на графах, связанных с динамическим разбиением элементов.

Структура данных

Массив parent: parent[i] хранит "родителя" элемента i. Если элемент i является представителем своего множества (корнем дерева), то $parent[i]=i$.

Массив rank: Оптимизация для балансировки деревьев. rank[i] хранит информацию о "высоте" дерева или количестве элементов в подмножестве с корнем i.

Операция Find

Операция Find(x) находит представителя множества, содержащего x. Если x не является корнем ($parent[x]\neq x$, выполняется рекурсивный вызов: $parent[x]=Find(parent[x])$). Это называется сжатием путей (path compression), которое упрощает структуру дерева, делая все элементы прямыми потомками корня.

Пример без сжатия путей: $parent$=[0,0,1,2]
Вызов $Find(3)$ даёт путь 3→2→1→0.
После сжатия путей все элементы будут указывать на корень сразу: $parent$[0,0,0,0].
Операция Union
Операция Union(x, y) объединяет два множества, содержащие элементы x и y.

- Сначала находим корни x и y с помощью Find(x) и Find(y).
- Если корни разные, выполняем слияние подмножеств.
    - Если rank[x] > rank[y], то y присоединяется к x.
    - Если rank[x] < rank[y], то x присоединяется к y.
    - Если ранги равны, выбираем произвольный корень (например, x) и увеличиваем его ранг.

Пример:

1. Изначально: parent = [0,1,2,3], rank=[0,0,0,0].
2. Выполняем Union(1,2):
    - Корни: Find(1) = 1,Find(2) = 2.
    - Объединяем множества: parent[2] = 1,rank[1] = 1.

Результат: parent=[0,1,1,3], rank=[0,1,0,0].
Эффективность

1. **Оптимизация с помощью "path compression":** При вызове Find(x) все элементы на пути от x до корня становятся прямыми потомками корня. Это существенно уменьшает глубину дерева.
2. **Оптимизация по рангу:** Гарантирует, что деревья остаются сбалансированными, минимизируя глубину.
3. **Амортизированное время:** С использованием оптимизаций, время на каждую операцию Find и Union становится O(α(n))O, где α(n) — обратная функция Аккермана, которая растёт крайне медленно. Для всех практических случаев α(n)≤4.

3. Сравнительный анализ с аналогичными алгоритмами

Для сравнения рассмотрим несколько аналогичных подходов для решения задачи поиска и объединения компонент:

1. Алгоритм Краскала – алгоритм, который начинается с пустого дерева, и строит остовное дерево, последовательно вставляя ребра из E в порядке возрастания стоимости. При перемещении по ребрам в этом порядке каждое ребро e вставляется в том случае, если при добавлении к ранее вставленным ребрам оно не создает цикл. Если же, напротив, вставка e порождает цикл, то ребро e просто игнорируется, а выполнение алгоритма продолжается.

Время работы O(ElogE)

Данный алгоритм называется жадным из-за того, что мы на каждом шаге пытаемся найти оптимальный вариант, который приведет к оптимальному решению в целом.

2. Поиск в глубину (DFS) — это алгоритм обхода или поиска в графах и деревьях. Он начинает с заданной вершины (обычно называемой *источником*) и проходит по графу, углубляясь в ветви до тех пор, пока возможно, затем возвращается назад для изучения других путей.

Подходит для:

- Проверки связности графа.
- Поиска пути между двумя вершинами.
- Нахождения компонент связности.
- Топологической сортировки.
- Обнаружения циклов.

Время работы O(V+E)

3. Алгоритм поиска в ширину (BFS, Breadth-First search) – простейший алгоритм, при котором просмотр ведется от s во всевозможных направлениях с добавлением одного “уровня” за раз.

Подходит для:

- Поиска кратчайшего пути в невзвешенных графах.
- Проверки связности графа.
- Поиска компонент связности.

Время работы O(V+E)

В отличие от них алгоритм Union-Find с оптимизациями работает значительно быстрее и является гораздо более эффективным как по времени, так и по памяти.

4. Перечень инструментов, используемых для реализации

- Язык программирования: C++
- Среда разработки: Visual Studio Code
- Библиотеки: стандартные библиотеки языка

5. Реализация и тестирование

Реализация алгоритма происходила в несколько этапов:

1. Создание классов для структуры данных, реализующих операции Find и Union.
2. Оптимизация при помощи сжатия путей и слияния по рангу.
3. Разработка тестов для проверки корректности работы алгоритма.

Тестирование осуществлялось на наборе случайных данных, а также на заранее подготовленных тестах для оценки производительности.

Код был написан в Visual Studio Code на C++. Полный код программы находится в приложении. Приложение 1.

В коде был использован поиск с сжатием пути.

```
    // Поиск с сжатием пути
        int find(int x) {
            if (parent[x] != x) {
                parent[x] = find(parent[x]); // Рекурсивное нахождение родителя и
сжатие пути
            }
            return parent[x];
        }
```

В коде был использован метод объединения с использованием рангов.

```
    // Метод объединения с использованием рангов
        void unionSets(int x, int y) {
            int rootX = find(x);
            int rootY = find(y);

            if (rootX != rootY) {
                // Присоединяем дерево с меньшим рангом к дереву с большим рангом
                if (rank[rootX] < rank[rootY]) {
                    parent[rootX] = rootY;
                } else if (rank[rootX] > rank[rootY]) {
                    parent[rootY] = rootX;
                } else {
                    parent[rootY] = rootX;
                    rank[rootX]++;
```

```
            }
        }
    }
```

И дополнительная функция для красивого вывода, чтобы посмотреть как работает программа.

```
    // Печать родительского массива (для отладки)
        void printParents() {
            for (int i = 0; i < parent.size(); i++) {
                std::cout << "Element: " << i << ", Parent: " << parent[i] <<
"\n";
            }
        }
```

Теперь перейдем к тестированию: я создала рандомный граф

```
    int main() {
        int n = 10; // Количество элементов
        UnionFind uf(n);

        // Пример объединений
        uf.unionSets(1, 2);
        uf.unionSets(2, 3);
        uf.unionSets(4, 5);
        uf.unionSets(6, 7);
        uf.unionSets(3, 7); // Объединение двух больших компонент
```

Затем сделала проверку принадлежности элементов к одному множеству и проверку того, принадлежат ли множество 1 и множество 7 одному множеству.

```
        std::cout << "Find(1): " << uf.find(1) << "\n";
        std::cout << "Find(7): " << uf.find(7) << "\n";

        if (uf.find(1) == uf.find(7)) {
            std::cout << "1 и 7 принадлежат одному множеству.\n";
        } else {
            std::cout << "1 и 7 принадлежат разным множествам.\n";
        }
```

Вывод программы, продублировала здесь для удобства, также вывод программы добавлен в приложение.

```
Find(1): 1
Find(7): 1
1 и 7 принадлежат одному множеству.
Родительский массив:
Element: 0, Parent: 0
Element: 1, Parent: 1
Element: 2, Parent: 1
Element: 3, Parent: 1
Element: 4, Parent: 4
Element: 5, Parent: 4
Element: 6, Parent: 1
Element: 7, Parent: 1
Element: 8, Parent: 8
Element: 9, Parent: 9
```

Описание операций

Объединение элементов:

Union(1,2)): теперь 1 и 2 принадлежат одному множеству.

Union(2,3) теперь 1, 2, и 3 принадлежат одному множеству.

Union(4,5): теперь 4 и 5 принадлежат одному множеству.

Union(6,7) теперь 6 и 7 принадлежат одному множеству.

Union(3,7): объединяются множества {1,2,3} и {6,7}.

Проверка принадлежности:

Результат Find(1) покажет корень множества, содержащего элемент 1.

Результат Find(7) покажет корень множества, содержащего элемент 7.

Печать родительского массива:

Будут выведены текущие родители для всех элементов после выполнения операций.

Объяснение вывода

Find(1): 1 и Find(7): 1:

После объединений 1 и 7 находятся в одном множестве, и их общий корень — 1.

"1 и 7 принадлежат одному множеству.":

У элементов 1 и 7 одинаковый корень, что подтверждает их принадлежность одному множеству.

Родительский массив:

Для каждого элемента отображается его родитель после выполнения всех операций:

Элементы {1,2,3,6,7} имеют общий корень 1.

Элементы {4,5} имеют общий корень 4.

Элементы {0,8,9} остались в своих отдельных множествах.

Алгоритм работает верно, у меня все получилось.

Применения:

Алгоритм Краскала: Для построения минимального остовного дерева объединяет рёбра графа, проверяя циклы.

Динамическое соединение компонент: Например, в сетевых задачах.

Задачи на эквивалентность: Проверка принадлежности элементов одному множеству.

Задачи на кластеризацию: Например, объединение данных по близости.

Алгоритм Union-Find благодаря своей эффективности стал важным инструментом во многих прикладных задачах.

Заключение

Алгоритм Union-Find с оптимизациями (путь сжатия и объединение по рангу) является эффективным решением для задач, связанных с динамическим объединением и поиском компонент. Реализованный алгоритм показывает отличные результаты по времени и памяти, что делает его пригодным для решения реальных задач с большими данными.

Приложение

```
#include <iostream>
#include <vector>

class UnionFind {
private:
    std::vector<int> parent; // Родитель каждого элемента
    std::vector<int> rank;   // Ранг дерева для каждого элемента

public:
    // Конструктор: инициализация n элементов
    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0); // Изначально все ранги равны 0
        for (int i = 0; i < n; i++) {
            parent[i] = i; // Каждый элемент является своим собственным
родителем
        }
    }

    // Поиск с сжатием пути
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // Рекурсивное нахождение родителя и
сжатие пути
        }
        return parent[x];
    }

    // Метод объединения с использованием рангов
    void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX != rootY) {
            // Присоединяем дерево с меньшим рангом к дереву с большим рангом
            if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }

    // Печать родительского массива (для отладки)
    void printParents() {
        for (int i = 0; i < parent.size(); i++) {
```

```
                std::cout << "Element: " << i << ", Parent: " << parent[i] << "\n";
            }
        }
    };
```

1-Код алгоритма

```
#include <iostream>
#include "Union-find.cpp"

int main() {
    int n = 10; // Количество элементов
    UnionFind uf(n);

    // Пример объединений
    uf.unionSets(1, 2);
    uf.unionSets(2, 3);
    uf.unionSets(4, 5);
    uf.unionSets(6, 7);
    uf.unionSets(3, 7); // Объединение двух больших компонент

    // Проверка принадлежности элементов к одному множеству
    std::cout << "Find(1): " << uf.find(1) << "\n";
    std::cout << "Find(7): " << uf.find(7) << "\n";

    // Проверка объединения 1 и 7
    if (uf.find(1) == uf.find(7)) {
        std::cout << "1 и 7 принадлежат одному множеству.\n";
    } else {
        std::cout << "1 и 7 принадлежат разным множествам.\n";
    }

    // Печать родительского массива
    std::cout << "Родительский массив:\n";
    uf.printParents();

    return 0;
}
```

2-Проверка работы алгоритма

```
Find(1): 1
Find(7): 1
1 и 7 принадлежат одному множеству.
Родительский массив:
Element: 0, Parent: 0
Element: 1, Parent: 1
Element: 2, Parent: 1
Element: 3, Parent: 1
Element: 4, Parent: 4
Element: 5, Parent: 4
Element: 6, Parent: 1
Element: 7, Parent: 1
Element: 8, Parent: 8
Element: 9, Parent: 9
```
3-Вывод