Система неперетинних множин

Підготував
Грибок Андрій

- Система неперетинних множин (DSU) — структура даних, яка дозволяє відстежувати множину елементів, розбиту на неперетинні підмножини. При цьому кожній підмножині призначається її представник — елемент цієї підмножини.

- Структура даних надає такі можливості. Спочатку є кілька елементів, кожен з яких знаходиться в окремій (своїй власній) множині. За одну операцію можна **об'єднати дві будь-які множини**, а також **можна запитати**, в якій множині зараз знаходиться зазначений елемент. У класичному варіанті вводиться ще одна операція — **створення нового елемента,** який поміщається в окрему множину — синґлетон.

- MakeSet — додання нового елемента; розміщення його в нову множину, що складається з одного нього;
- Union — об'єднання двох зазначених множин;
- Find — повернення значення, в якій множині знаходиться зазначений елемент. Насправді при цьому повертається один з елементів множини (званий представником). Цей представник вибирається в кожній множині самою структурою даних (і може змінюватися з плином часу). Наприклад, якщо виклик для якихось двох елементів повернув одне і те ж значення, то це означає, що ці елементи знаходяться в одній і тій же множині, а в іншому випадку — в різних множинах.

Наївна реалізація

Вся інформація про множини елементів зберігається у нас за допомогою масиву parrent.

- Щоб створити новий елемент (операція make_set(v)), ми просто створюємо дерево з коренем у вершині, зазначаючи, що її предок — це вона сама.
- Щоб об'єднати дві множини (операція union_set(a, b)), ми спочатку знайдемо лідерів першої і другої множини. Якщо лідери збіглися, то нічого не робимо — це означає, що множини і так вже були об'єднані. В іншому випадку можна просто вказати, що предок першої вершини дорівнює другій (або навпаки) — тим самим приєднавши одне дерево до іншого.
- Реалізація операції пошуку лідера (find_set(v)) проста: ми піднімаємося по предкам від вершини, поки не дійдемо до кореня. Цю операцію зручніше реалізувати рекурсивно (особливо це буде зручно пізніше, у зв'язку з доданими оптимізаціями).

Наївна реалізація

```
void make_set (int v) {
    parent[v] = v;
}

int find_set (int v) {
    if (v == parent[v])
        return v;
    return find_set (parent[v]);
}

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set (b);
    if (a != b)
        parent[b] = a;
}
```

Удосконалення

прискорення роботи find_set().

- Вона полягає в тому, що коли після виклику ми **знайдемо шуканого лідера множини**, то запам'ятаємо, **що у вершині v і всіх пройдених** по шляху вершин — саме цей лідер. Найпростіше це зробити, перенаправивши їх parrent[] на цю вершину.
- Таким чином, у масиву предків parrent сенс дещо змінюється: тепер це стислий масив предків, тобто для кожної вершини там може зберігатися не безпосередній предок, а предок предка, предок предка предка, і т. д.

- Така проста реалізація робить все, що задумувалося: спочатку шляхом рекурсивних викликів знаходиться лідера множини, а потім, цей лідер присвоюється parrent посиланнями для всіх пройдених елементів.

```
int find_set (int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set (parent[v]);
```

Удосконалення

Об’єднання за рангом

```
void make_set (int v) {
    parent[v] = v;
    size[v] = 1;
}

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set (b);
    if (a != b) {
        if (size[a] < size[b])
            swap (a, b);
        parent[b] = a;
        size[a] += size[b];
    }
```

Фінальна реалізація через глибину

```
void make_set (int v) {
    parent[v] = v;
    rank[v] = 0;
}

int find_set (int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set (parent[v]);
}

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set (b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap (a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            ++rank[a];
    }
}
```

Для чого це все?

Перевірка зв’язності графа

- Це одна з очевидних програм структури даних «система неперетинних множин», яка, вочевидь, і стимулювала вивчення цієї структури.
- Формально задачу можна сформулювати таким чином: спочатку заданий порожній граф, поступово в цей граф можуть додаватися вершини і неорієнтовані ребра, а також надходять запити — «чи в однакових компонентах зв'язності лежать вершини і?».
- Безпосередньо застосовуючи тут описану вище структуру даних, ми отримуємо рішення, яке обробляє один запит на додавання вершини / ребра або запит на перевірку двох вершин — за майже константний час у середньому.

Збір додаткової інформації

- "Система непересічних множин" дозволяє легко зберігати будь-яку додаткову інформацію, що стосується до множин.
- Простий приклад - це розміри множин: як їх зберігати, було описано вище.
- Таким чином, разом з лідером кожної множини можна зберігати будь-яку додаткову необхідну в конкретному завданні інформацію

Пошук компонент зв'язності на зображенні

- Одне з лежачих на поверхні застосувань DSU полягає у вирішенні наступного завдання: є зображення пікселів; спочатку все зображення біле, але потім на ньому малюється декілька чорних крапок. Потрібно визначити розмір кожної «білої» компоненти зв'язності на підсумковому зображенні.
- Для рішення ми просто перебираємо всі білі клітини зображення, для кожної клітини перебираємо її чотирьох сусідів, і якщо сусід теж білий — то викликаємо від цих двох вершин. Таким чином, у нас буде DSU з вершинами, відповідними пікселям зображення. Отримані в результаті дерева DSU — і є шукані компоненти зв'язності.
- Дану задачу можна вирішити простіше з використанням обходу в глибину (або обходу в ширину), однак у описаного тут методу є певна перевага: воно може обробляти матрицю по рядках (оперуючи тільки з поточним рядком, попередньої рядком і системою непересічних множин, побудованої для елементів одного рядка), тобто використовуючи порядку O (\ min (n, m)) пам'яті.

Ще задачі або доповнення

- Застосування DSU для стиснення «стрибків» по відрізку. Завдання про зафарбування підвідрізків в офлайні
- **Підтримка відстаней до лідера**
- Підтримка парності довжини шляху і завдання про перевірку дводольного графа в онлайн
- Алгоритм знаходження RMQ (мінімуму на відрізку) в офлайні