ДЗ-10 Перепелкин Богдан, М3135

1 10.1

Условие: Дан пустой граф на n вершин. Надо уметь за обратную функцию Аккермана

a) добавлять ребро в граф

b) находить число ребер в компоненте связности, где лежит x

Решение: будем использовать систему непересекающихся подмножеств (СМН), будем поддерживать 4 операции:

- `find` (н-ти представителя)
- `union` (объединить две вершины)
- `count_edges` (посчитать кол-во рёбер в множестве, где лежит текущая вершина)
- `add_edge` (создать ребро)

Описание структуры данных:

1. `массив parents` (для каждого элемента ссылка на его родителя) Изначально каждый элемент является своим родителем. Используем сжатие путей.
2. `массив ranks` (для каждого корня хранится глубина снм). Изначально все ранги $= 0$.
3. `массив edges` (для каждого корня хранится кол-во ребер в смн) Изначально у всех элементов по 0 ребер в их деревьях.

Функции:

1. find(v)

```
def find(v):
    if parents[v] == v:
        return v
    parents[v] = find(parents[v])
    return parents[v]
```

2. union(a,b)

```
def union(a,b):
    a = find(a)
    b = find(b)
    if ranks[a] > ranks[b]:
        swap(a,b)
    if ranks[a] == ranks[b]:
        ranks[b]++
    if a != b:
        parents[a] = b
        edges[b] += edges[a]
    edges[b] += 1
```

3. count_edges(v)

```
def count_edges(v):
    v = find(v)
    return edges[v]
```

4. add_edge

```
def add_edge(a,b):
    union(a,b)
```

Доказательство сложности обратной функции Аккермана: в функциях union и count_edges используется find и остальные операции за O(1). Тогда достаточно доказать, что find работает за Аккермана. Это верно, так как используется сжатие путей.

Это работает, так как мы храним количество ребер не зависимо от того, как связаны элементы со своим корнем. То есть достигается и сложность обратной функции Аккермана, и возможность валидно хранить количество ребер в каждой компоненте. Во всех следующих задачах доказательство сложности аналогичное.

2 10.2

Условие: необходимо поддерживать стркутуру с массивом из нулей и едениц, в которой все операции, описанные ниже будут работать за обратную от функции Аккермана.

- присвоить 1 i-му элементу
- найти ближайший ноль к i-му элементу массива

Структура данных:

1. массив a, хранит нули и единицы
2. массив rank - такой же как в номере 10.1
3. массив parents хранит подмассив для каждого элемента, где
   - родитель на 0 индексе
   - ближайший ноль слева на 1 индексе
   - ближайший ноль справа на 2 индексе

   Изначально хранит, что каждый элемент сам себе родитель и ближайший ноль слева, справа это сам элемент. Затем будем объединять в одну компоненту непрерывные последовательности из 1. Функция find такая же, как в номере 10.1 (она будет работать за обратную функцию Аккермана)

1. init_one(i) Данная функция будет присваивать единицу и при необходимости объединять эту единицу в соседние компоненты.

```
def init_one(i):
    a[i] = 1
    #union with left
    if a[i - 1] == 1:
        left = find(i - 1)
        parents[i][0] = left
        parents[left][2] = i
        if ranks[left] == ranks[i]: ranks[left]++ #two lonely "1" merged
    else:
        parents[i][1] = i - 1
    #union with right
    cur = find(i) #not just i, because we could merge with left on step
        before
    if a[i + 1] == 1:
        right = find(i + 1)
```

```
        if ranks[cur] > ranks[right]:
            swap(cur, right)
        parents[cur][0] = right
        parents[right][1] = parents[cur][1] #give cur's left zero to
            right's left zero
    else:
        parents[cur][2] = i + 1
```

Очевидно как сделать обработку граничных положений - сделать нули слева, справа от массива a и в конце union заифать на выход за пределы длины a. Не стал писать, ибо код и так получился довольно громоздким.

2. closest_zero(i)

```
def closest_zero(i):
    cur = find(i)
    if abs(parents[cur][1] - i) < abs(parents[cur][2] - i):
        return parents[cur][1]
    else:
        return parents[cur][2]
```

Это работает, потому что всегда сохраняеся инвариант - для каждой компоненты знаем ближайший ноль слева, справа

3 10.3

Условие: на пустом графе надо уметь добавлять ребра, находить число компонент связности, являющихся деревьями.

Реализация: проверка на то, дерево или нет компонента, происходит по следующей формуле. Дерево, если граф связен и $p = q + 1$, где p - количество вершин, q - количество ребер.

Почему это работает?

Потому что на wiki конспектах это условие, описанное выше эквивалентно тому, что рассматриваемая компонента - дерево. При этом проверка на $p = q + 1$ валидна благодаря поддерживаемым массивам size и edges, и она происходит только с элементами в одной компоненте, то есть между любой парой вершин есть хотя бы одна связь, следовательно компонента - связна.

Структура данных:

1. `массив parents` вместе с функцией find такие же как и в 1 задании
2. `массив size` (для каждого корня хранится кол-во элементов в его компоненте)

   изначально все = 1 (т.к. один элемент есть)
3. `массив edges` (для каждого корня хранятся кол-во ребер в его компоненте)

   изначально все = 0 (т.к. пока что нет ребер)
4. `массив isTree` (для каждого корня хранятся является ли он деревом)

   изначально все = 1 (т.к. один элемент - это дерево)

Функции:

1. find(v) - аналогично номеру 10.1
2. union(a,b) - добавить ребро в граф

```
def union(a,b):
    a = find(a)
    b = find(b)
    if ranks[a] > ranks[b]:
        swap(a,b)
    if ranks[a] == ranks[b]:
        ranks[b]++
    if a != b:
        parents[a] = b
        edges[b] += edges[a]
        size[b] += size[a]
    edges[b] += 1
    if (isTree[a] and isTree[b] and size[b] - edges[b] == 1):
        isTree[b] = 1
    else:
        isTree[b] = 0
```

3. count_trees()

```
def count_trees():
    cnt = 0
    used = []
    for i in range(len(parents)):
        parent = find(i)
        if i == parent and parent not in used:
            if isTree[i] == 1:
                cnt++
        used.add(parent)
    return cnt
```

4. add_edge(v), find(v) (аналогично как в пункте 10.1)

Модифицировали union, чтобы обновлять информацию о том, является ли компонента деревом. Это позволило сделать функцию count_trees, подсчитывающую число компонент связности, являющихся деревьями

4 10.4

Условие: на пустом графе необходимо уметь выполнять запросы двух типов (вермя работы - обратная ф-я Аккермана)

- добавить ребро в граф
- найти кол-во компонент связности, являющихся циклами

Идея аналогичная той, что в номере 10.3, но немного с дополнениями. Сложность заключается в том, что формально компонента является циклом, если в ней количество элементов равно количеству ребер и каждая компонента связана с двумя другими. Будем это поддерживать, используя массив cheldren. Таким образом возможно корректно подсчитывать циклы, проверя на при каждом union, что у элементов, которые мы обновили есть ровно две связи.

Структура данных:

1. `массив parents`
2. `массив size` хранит размер каждой компоненты

3. `массив edge` хранит кол-во ребер в компонентах
4. `массив children` хранит кол-во связей у каждого элемента. Изначально у всех нули
5. `массив isCycle` хранит true/false
6. `переменная cnt` хранит ответ на вопрос о кол-ве циклов

Функции:

1. find(v) - со сжатием путей
2. union(a,b) - добавить ребро в граф

```
def union(a,b):
    root_a = find(a)
    root_b = find(b)
    if size[root_a] > size[root_b]:
        swap(a,b)
    children[a] +=1
    children[b] +=1
    if root_a != root_b:
        parents[root_a] = root_b
        size[root_b] += size[root_a]
        edge[root_b] += edge[root_a] + 1
    else:
        edge[root_b] += 1

    # checking for creating cycle
    if size[root_b] == edge[root_b] and children[a] == 2 and children[b]
       == 2:
        isCycle[root_b] = true
        cnt++
    else:
        isCycle[root_b] = false
        cnt--
```

3. count_cycles()

```
def count_cycles():
    return cnt
```

5 10.5

Условие: требуется добавить в СНМ 2 опреации:

- Увеличить веса всех элементов, находящихся в одном множестве с элементом x, на d.
- Найти вес элемента x.

Реализация: сложность заключается в том, что это нужно сделать ассимптотически за обратную функцию Аккермана. Для этого модифицируем алгоритм сжатия путей, чтобы прокидывать сдвиг по весам вместе с перекидыванием элементов к родителям повыше.

Структура данных:

1. Пусть w - массив с весами
2. shift - сдвиги по весам

3. rank - обычные ранги как до этого

4. массив parents - ссылки на родителей

Функции:

1. find(v)

```
def find(v):
    if parents[v] != v:
        root = find(parents[v])
        # transfer shift for weights at old parent to new parent
        w[v] += sift[parents[v]]
        shift[parents[v]] = 0
        parents[v] = root
    return parents[v]
```

2. union(a,b)

```
def union(a,b):
    root_a = find(a)
    root_b = find(b)
    if rank[root_a] > rank[root_b]:
        swap(root_a, root_b)
    if rank[root_a] == rank[root_b]:
        rank[root_b] ++
    if root_a != root_b:
        parents[root_a] = root_b
```

3. increase(v, d)

```
def increase(v, d):
    root = find(v)
    shift[root] += d
```

4. count_weight(v)

```
def count_weight(v):
    root = find(v)
    return w[v] + shift[root]
```