ИКН НИТУ МИСИС

Комбинаторика и теория графов

Задача «объединить-найти». Система непересекающихся множеств. Алгоритм со сжатием путей сложности O(nG(n))

Исполнитель:

Миронов Е.А. БИВТ-23-8

(<https://github.com/Valet-V0ult-de-Fur1e/combinatorics_and_graphs_repo>)

Москва 2024 год

**Определение**

Система непересекающихся множеств (СНМ) — иерархическая структура данных, позволяющая эффективно работать с множествами. Структура хранит набор объектов в виде непересекающихся множеств, у каждого множества есть конкретный представитель.

**Примеры применения**

Поиск компонент связности. Например, если два города лежат в разных множествах, то физически не существует пути между ними.

Остов минимального веса (алгоритмы Краскала и Прима). Нужно оставить граф связным, но из всех рёбер взять такие, сумма которых минимальна.

Задачи, связанные с генерированием связанных пространств, например, генерирование лабиринта. Например, если есть поле 3х3 клетки, между каждой клеткой есть стена, то каждая клетка — это множество, и удаление стены между двух клеток можно заменить на объединение двух множеств.

**Теоретическое описание**

Изначально имеется n элементов, каждый из которых находится в отдельном (своём собственном) множестве.

Структура поддерживает две базовые операции:

1. union(x, y) - объединить два каких-либо множества. При этом все элементы обоих множеств становятся элементами результирующего множества.
2. find(x) - запросить, в каком множестве сейчас находится указанный элемент.

СНМ часто используется в графовых алгоритмах для хранения информации о связности компонент.

**Устройство структуры**

Множества элементов хранятся в виде деревьев: одно дерево соответствует одному множеству. Корень дерева — это представитель (лидер) множества. Для описания множества используется номер вершины, являющейся корнем соответствующего дерева.

Для определения, принадлежат ли два элемента к одному и тому же множеству, для каждого элемента нужно найти корень соответствующего дерева (поднимаясь вверх, пока это возможно) и сравнить эти корни.

Для объединения множеств нужно подвесить корень одного за корень другого.

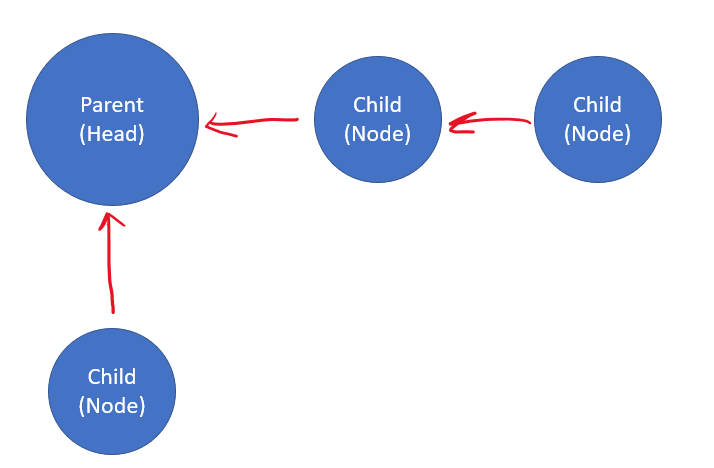


Рис.1 Схема базовой реализации

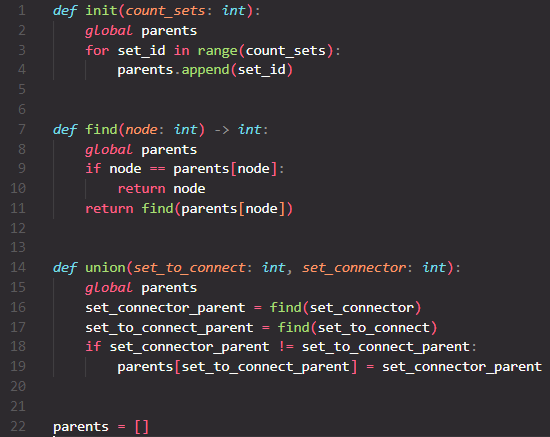


Рис.2 код базовой реализации

**Асимптотика базового поиска «главной» вершины**

В худшем случае такая реализация работает за 𝑂(𝑛)— можно построить «бамбук», подвешивая его n раз за новую вершину.

**Оптимизация**

Для оптимизации можно при поиске «главной вершины» для конкретной подвешивать попутно встречающиеся вершины к «главной», что ускорит поиск

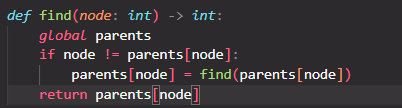


Рис.5 реализация оптимизации сокращения пути

**Асимптотика оптимизированного поиска «главной» вершины**

Применение оптимизации путей позволяет достичь логарифмической асимптотики:

O(log n) в среднем на один запрос. То есть при n запросах асимптотика будет равна O(nlogn)

**Доказательство**

Покажем, что применение одной эвристики сжатия пути **позволяет достичь логарифмическую асимптотику**:   на один запрос в среднем.

Заметим, что, поскольку операция представляет из себя два вызова операции и ещё  операций, то мы можем сосредоточиться в доказательстве только на оценку времени работы  операций .

Назовём **весом**  вершины u число потомков этой вершины (включая её саму). Веса вершин, очевидно, могут только увеличиваться в процессе работы алгоритма.

Назовём **размахом ребра**  разность весов концов этого ребра:   (очевидно, у вершины-предка вес всегда больше, чем у вершины-потомка). Можно заметить, что размах какого-либо фиксированного ребра  может только увеличиваться в процессе работы алгоритма.

Кроме того, разобьём рёбра на **классы**: будем говорить, что ребро имеет класс k, если его размах принадлежит отрезку . Таким образом, класс ребра — это число от 0 до .

Зафиксируем теперь произвольную вершину x и будем следить, как меняется ребро в её предка: сначала оно отсутствует (пока вершина x является лидером), затем проводится ребро из x в какую-то вершину (когда множество с вершиной x присоединяется к другому множеству), и затем может меняться при сжатии путей в процессе вызовов {\rm find\_path}. Понятно, что нас интересует асимптотика только последнего случая (при сжатии путей): все остальные случаи требуют  времени на один запрос.

Рассмотрим работу некоторого вызова операции : он проходит в дереве вдоль некоторого **пути**, стирая все рёбра этого пути и перенаправляя их в лидера.

Рассмотрим этот путь и **исключим** из рассмотрения последнее ребро каждого класса (т.е. не более чем по одному ребру из класса ). Тем самым мы исключили  рёбер из каждого запроса.

Рассмотрим теперь все **остальные** рёбра этого пути. Для каждого такого ребра, если оно имеет класс k, получается, что в этом пути есть ещё одно ребро класса k (иначе мы были бы обязаны исключить текущее ребро, как единственного представителя класса k). Таким образом, после сжатия пути это ребро заменится на ребро класса как минимум . Учитывая, что уменьшаться вес ребра не может, мы получаем, что для каждой вершины, затронутой запросом , ребро в её предка либо было исключено, либо строго увеличило свой класс.

Отсюда мы окончательно получаем асимптотику работы m запросов: , что (при ) означает логарифмическое время работы на один запрос в среднем.

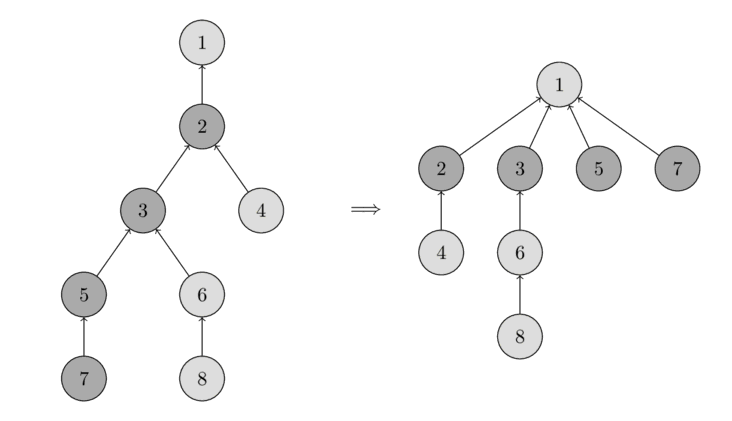


Рис.6 пример работы сокращения пути после поиска «главной» вершины для вершины 7

**Весовая эвристика**

Недостаток наивной реализации проявляется при слиянии относительно большого множества с множеством из одного элемента. В наивной реализации список указанный первым всегда подвешивается ко второму. Хотя в данном случае гораздо выгоднее подвесить меньший список к большему, обновив один указатель на представителя, вместо обновления большого числа указателей в первом списке. Отсюда следуют очевидная оптимизация — будем для каждого множества хранить его размер и изменять указатели на представителя всегда элементам из "меньшего" списка.

Для сравнения вершин при объединении множеств можно использовать количество вершин в каждом множестве.

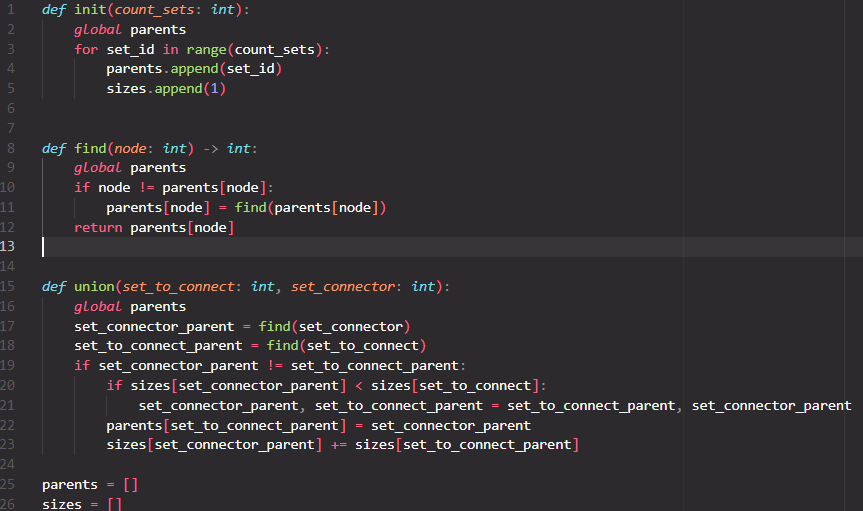


Рис.7 реализация через размерность множеств

Так же для сравнения вершин при объединении множеств можно использовать ранги, то есть количество уровней в дереве множества.



Рис.8 реализация через глубину множеств

**Анализ аналогов**

В качестве альтернативы для поиска компоненты связанности можно использовать dfs с асимптотикой О(n^2), n - количество вершин.

**Задачи в которых чаще всего используется**

* Задача о покраске подотрезков (Заливка).
* Алгоритм Краскала
* Алгоритм Прима
* Поддержка компонент связности графа
* Поиск компонент связанности на изображении
* Поддержка дополнительной информации для каждого множества
* Алгоритм нахождения минимума на отрезке
* Проверка чётности двудольности графа

**Ссылка на реализацию**

<https://github.com/Valet-V0ult-de-Fur1e/combinatorics_and_graphs_repo/tree/main/Задача%20объединить-найти%20Система%20не%20пересекающихся%20множеств.%20Алгоритм%20со%20сжатием%20путей%20сложности%20O(nG(n))>

**Список источников**

* <http://www.e-maxx-ru.1gb.ru/algo/dsu>
* <https://ru.algorithmica.org/cs/set-structures/dsu/>
* <https://neerc.ifmo.ru/wiki/index.php?title=СНМ_(наивные_реализации)>