Алгоритмы и структуры данных

MIPT DIHT

22 января 2015 г.

1 Сортировки

1.1 Quick Sort

1.1.1 Шаги

- Выбираем в массиве некоторый элемент, который будем называть опорным элементом. Известные стратегии: выбирать постоянно один и тот же элемент, например, средний или последний по положению; выбирать элемент со случайно выбранным индексом. Часто хороший результат даёт выбор в качестве опорного элемента среднего арифметического между минимальным и максимальным элементами массива.

- Операция разделения массива: реорганизуем массив таким образом, чтобы все элементы со значением меньшим или равным опорному элементу, оказались слева от него, а все элементы, превышающие по значению опорный — справа от него. Обычный алгоритм операции:

  1. Два индекса — l и r, приравниваются к минимальному и максимальному индексу разделяемого массива, соответственно.
  2. Вычисляется индекс опорного элемента m.
  3. Индекс l последовательно увеличивается до тех пор, пока l-й элемент не окажется больше либо равен опорному.
  4. Индекс r последовательно уменьшается до тех пор, пока r-й элемент не окажется меньше либо равен опорному.
  5. Если r = l — найдена середина массива — операция разделения закончена, оба индекса указывают на опорный элемент.
  6. Если l < r — найденную пару элементов нужно обменять местами и продолжить операцию разделения с тех значений l и r, которые были достигнуты. Следует учесть, что если какая-либо граница (l или r) дошла до опорного элемента, то при обмене значение m изменяется на r-й или l-й элемент соответственно, изменяется именно индекс опорного элемента и алгоритм продолжает свое выполнение.

- Рекурсивно упорядочиваем подмассивы, лежащие слева и справа от опорного элемента.

1.1.2 Оценка сложности

Разделение массива: O(n).
Лучший случай
Массив делится на две почти одинаковые части, максимальная глубина рекурсии $\log_2 n$. Количество сравнений: $C_n = 2 \cdot C_{n/2} + n$, что даёт общую сложность $O(n \cdot \log_2 n)$.
Среднее
Среднюю сложность при случайном распределении входных данных можно оценить лишь вероятностно. Прежде всего необходимо заметить, что в действительности необязательно, чтобы опорный элемент всякий раз делил массив на две одинаковых части. Например, если на каждом этапе будет происходить разделение на массивы длиной 75 процентов и 25 процентов от исходного, глубина рекурсии будет равна $\log_{4/3} n$, а это по-прежнему даёт сложность

$O(n \log n)$. Вообще, при любом фиксированном соотношении между левой и правой частями разделения сложность алгоритма будет той же, только с разными константами.
Будем считать «удачным» разделением такое, при котором опорный элемент окажется среди центральных 50 процентов элементов разделяемой части массива; ясно, вероятность удачи при случайном распределении элементов составляет 0,5. При удачном разделении размеры выделенных подмассивов составят не менее 25 процентов и не более 75 процентов от исходного. Поскольку каждый выделенный подмассив также будет иметь случайное распределение, все эти рассуждения применимы к любому этапу сортировки и любому исходному фрагменту массива.
Удачное разделение даёт глубину рекурсии не более $\log_{4/3} n$. Поскольку вероятность удачи равна 0,5, для получения k удачных разделений в среднем потребуется $2 \cdot k$ рекурсивных вызовов, чтобы опорный элемент k раз оказался среди центральных 50 процентов массива. Применяя эти соображения, можно заключить, что в среднем глубина рекурсии не превысит $2 \cdot \log_{4/3} n$, что равно $O(\log n)$ А поскольку на каждом уровне рекурсии по-прежнему выполняется не более O(n) операций, средняя сложность составит $O(n \log n)$.
Худший случай
В самом несбалансированном варианте каждое разделение даёт два подмассива размерами 1 и n-1, то есть при каждом рекурсивном вызове больший массив будет на 1 короче, чем в предыдущий раз. Такое может произойти, если в качестве опорного на каждом этапе будет выбран элемент либо наименьший, либо наибольший из всех обрабатываемых. При простейшем выборе опорного элемента — первого или последнего в массиве, — такой эффект даст уже отсортированный (в прямом или обратном порядке) массив, для среднего или любого другого фиксированного элемента «массив худшего случая» также может быть специально подобран. В этом случае потребуется n-1 операций разделения, а общее время работы составит $\sum_{i=0}^{n}(n-i) = O(n^2)$ операций, то есть сортировка будет выполняться за квадратичное время. Но количество обменов и, соответственно, время работы — это не самый большой его недостаток. Хуже то, что в таком случае глубина рекурсии при выполнении алгоритма достигнет n, что будет означать n-кратное сохранение адреса возврата и локальных переменных процедуры разделения массивов. Для больших значений n худший случай может привести к исчерпанию памяти (переполнению стека) во время работы программы.

1.2 Merge Sort

1.3 Heap Sort

2 Hash-table and hash-function

3 Динамическое программирование: общая идея. Линейная и матричная динамика. Динамика на отрезках

4 Амортизационный анализ

Амортизационный анализ — метод подсчета времени, требуемого для выполнения последовательности операций над структурой данных. При этом время усредняется по всем выполняемым операциям, и анализируется средняя производительность операций в худшем случае. Такой анализ чаще всего используется, чтобы показать, что даже если некоторые из операций последовательности являются дорогостоящими, то при усреднении по всем операциям средняя их стоимость будет небольшой за счѐт низкой частоты встречаемости. Подчеркнѐм, что оценка, даваемая амортизационным анализом, не является вероятностной: это оценка среднего времени выполнения операций для худшего случая.
Средняя амортизационная стоимость операций — величина a, находящаяся по формуле:

$$a = \frac{\sum_{i=1}^{n} t_i}{n}$$

где $t_1, \ldots, t_n$ - время выполнения операций $1, \ldots, n$ совершѐнных над структурой данных.
Амортизационный анализ использует следующие методы:

1. Метод усреднения (метод группового анализа).
2. Метод потенциалов.
3. Метод предоплаты (метод бухгалтерского учета).

Метод усреднения
В методе усреднения амортизационная стоимость операций определяется напрямую по формуле, указанной выше: суммарная стоимость всех операций алгоритма делится на их количество.
Пример
Рассмотрим стек с операцией $multipop(a)$— извлечение из стека элементов. В худшем случае она работает за $O(n)$ времени, если удаляются все элементы массива. Однако прежде чем удалить элемент, его нужно добавить в стек. Итак, если в стеке было не более элементов, то в худшем случае с каждым из них могли быть произведены 2 операции - добавление в стек и извлечение из него. Например, если было операций $push$- добавление в стек, стоимость каждой $O(1)$, и одна операция $multipop(n)$, то суммарное время всех операций — $O(2n)$, всего операций $n+1$, а значит, амортизационная стоимость операции — $O(1)$.
Математическое обоснование: Пусть n — количество операций, m — количество элементов, задействованных в этих операциях. Очевидно $m]leqn$. Тогда:

$$a = \frac{\sum_{i=1}^{n} t_i}{n} = a = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} t_{ij}}{n}$$

где t_{ij} — стоимость i-ой операции над j-ым элементом. Величина $\sum_{i=1}^{n} t_{ij}$ не превосходит 2, т. к. над элементом можно совершить только 2 операции, стоимость которых равна 1 — добавление и удаление. Тогда:

$$a \leq \frac{2m}{n} \leq 2$$

Таким образом, средняя амортизационная стоимость операций $a = O(1)$.

Метод потенциалов Теорема (О методе потенциалов): Введѐм для каждого состояния структуры данных величину Ф — потенциал. Изначально потенциал равен Φ_0, а после выполнения i-ой операции — Φ_i. Стоимость i-ой операции обозначим $a_i = t_i + \Phi_i - \Phi_{i-1}$. Пусть n — количество операций, m — размер структуры данных. Тогда средняя амортизационная стоимость операций $a = O(f(n,m))$ если выполнены два условия:

1. $\forall i \to a_i = O(f(n,m))$
2. $\forall i \to \Phi_i = O(nf(n,m))$

Доказательство:

$$a_i = \frac{\sum_{i=1}^{n} t_i}{n} = \frac{\sum_{i=1}^{n} a_i + \sum_{i=0}^{n-1} \Phi_i - \sum_{i=1}^{n} \Phi_i}{n} = \frac{nO(f(n,m)) + \Phi_0 - \Phi_n}{n} = O(nf(n,m))$$

Метод предоплаты
Представим, что использование определенного количества времени равносильно использованию определенного количества монет (плата за выполнение каждой операции). В методе предоплаты каждому типу операций присваивается своя учѐтная стоимость. Эта стоимость может быть больше фактической, в таком случае лишние монеты используются как резерв для выполнения других операций в будущем, а может быть меньше, тогда гарантируется, что текущего накопленного резерва достаточно для выполнения операции. Для доказательства оценки средней амортизационной стоимости $O(f(n,m))$ нужно построить учѐтные стоимости так, что для каждой операции она будет составлять $O(f(n,m))$. Тогда для последовательности из операций суммарно будет затрачено $O(f(n,m))$ монет, следовательно, средняя амортизационная стоимость операций будет $a_= \frac{\sum_{i=1}^{n} t_i}{n} = \frac{nO(f(n,m))}{n} = O(f(n,m))$.

Пример
Опять же рассмотрим стек с операцией $multipop(a)$. При выполнении операции $push$ будем использовать две монеты — одну для самой операции, а вторую — в качестве резерва. Тогда для операций pop и $multipop$ учѐтную стоимость можно принять равной нулю и использовать для удаления элемента монету, оставшуюся после операции $push$. Таким образом, для каждой операции требуется $O(1)$ монет, а значит, средняя амортизационная стоимость операций $a = O(1)$).

5 RMQ and LCA

5.1 RQM

Запрос минимума на отрезке
Вход: массив чисел, два числа (начало и конец отрезка в массиве)
Выход: минимальное значение на данном отрезке
Алгоритм:
Динамика (дерево отрезков)

1. Дополняем исходный массив до степени двойки бесконечно большими элементами.
2. Строим дерево отрезков (двоичное дерево: листья - значения элементов массива;вершины - минимум из дочерних листьев)
3. (Фундаментальный отрезок - такой отрезок, что существует вершина в дереве, которой он соответсвует)
   - Заведем два указателя l и r, и установим указывающими на концы исходного отрезка (если l (r) указывает на правый (левый) дочерний элемент, то эта вершина принадлежит разбиению на фундаментальные отрезки)
   - сдвигаем указатели на уровень вверх.
   - повторяем до тех пор пока указатели не совпадут.
   - Находя очередной фундоментальные отрезок, сравниваем его значение с имеющимся минимум, при необходимости обновляем минимум.

Оценка: препроцессинг O(n); запрос $O(\log_n)$.

5.2 LCA: сведение к RQM

Задача LCA (наименьший общий предок) для двух вершин u и v в корневом дереве T называется узел w, который среди всех узлов, являющихся предками как узла u, так и v, имеет наибольшую глубину. Пусть дано корневое дерево T. На вход подаются запросы вида (u,v), для каждого запроса требуется найти их наименьшего общего предка. Три основных алгоритма:

1. Алгоритм с препроцессингом
   Для каждой вершины определим глубину с помощью следующей рекурсивной формулы:

$$depth(u) = \begin{cases} 0, & u = root(T) \\ depth(v) + 1, & u = son(v) \end{cases}$$

   Ясно, что глубина вершины элементарным образом поддерживается во время обхода в глубину. Запустим обход в глубину из корня, который будет вычислять значения следующих величин:

   (a) Список глубин посещенных вершин d. Глубина текущей вершины добавляется в конец списка при входе в данную вершину, а также после каждого возвращения из еѐ сына.

   (b) Список посещений узлов vtx, строящийся аналогично предыдущему, только добавляется не глубина а сама вершина.

(c) Значение функции $I[u]$, возвращающей любой индекс в списке глубин d, по которому была записана глубина вершины u (например на момент входа в вершину).

Будем считать, что $rmq(d, l, r)$ возвращает индекс минимального элемента в d на отрезке $[l..r]$. Тогда ответом на запрос $lca(u, v)$, где $I[u] \leq I[v]$, будет $vtx\big[rmq(d, I[u], I[v])\big]$.

2. Метод двоичного подъема
Данный алгоритм является on-line (то есть сначала делается препроцессинг, затем алгоритм работает в формате запрос-ответ).
Препроцессинг заключается в том, чтобы посчитать функцию $dp[v][i]$ — номер вершины, в которую мы придем если пройдем из вершины v вверх по подвешенному дереву 2^i шагов, причем если мы пришли в корень, то мы там и останемся. Для этого сначала обойдем дерево в глубину и для каждой вершины запишем номер ее родителя $p[v]$ и глубину вершины в подвешенном дереве $d[v]$. Если v — корень, то $p[v] = v$. Тогда для функции dp есть рекуррентная формула:
$$dp[v][i] = \begin{cases} p[v], & ß = 0 \\ dp\big[dp[v[i-1]]\big]\big[i-1\big] \end{cases}$$
Для того чтобы отвечать на запросы нам нужны будут только те значения $dp[v][i]$, где $i \leq \log_2 n$, ведь при больших i значение $dp[v][i]$ будет номером корня.
Всего состояний динамики $O(n \log n)$, где n - это количество вершин в дереве. Каждое состояние считается за $O(1)$. Поэтому суммарная сложность времени и памяти препроцессинга — $O(n \log n)$. Ответы на запросы будут происходить за время $O(\log n)$. Для ответа на запрос заметим сначала, что если $c = LCA(v, u)$, для некоторых v и u, то $d[c] \leq \min(d[v], d[u])$. Поэтому если $d[v] < d[u]$, то пройдем от вершины u на $d[u] - d[v]$ шагов вверх, это и будет новое значение u и это можно сделать за $O(\log n)$. Можно записать число в двоичной системе, это представление этого число в виде суммы степеней двоек$2^{i_1} + \ldots + 2^{i_t}$, и для всех j пройти вверх последовательно из вершины u в $d[u][i_j]$.
Дальше считаем, что $d[v] = d[u]$. Если $v = u$, то ответ на запрос v. Иначе найдем такие вершины x и y, такие что $x \neq y$, x — предок v, y — предок u и $p[x] = p[y]$. Тогда ответом на запрос будет $p[x]$.
Научимся находить эти вершины x и y. Для этого сначала инициализируем $x = v$ и $y = u$. Дальше на каждом шаге находим такое максимальное k, что $dp[x][k] \neq dp[y][k]$. И проходим из вершин x и y на 2^k шагов вверх. Если такого k найти нельзя, то значения x и y, это те самые вершины, которые нам требуется найти, ведь $p[x] = dp[x][0] = dp[y][0] = p[y]$.
Оценим время работы. Заметим, что найденные k строго убывают. Во-первых, потому что мы находим на каждом шаге максимальное значение k, а во-вторых, два раза подряд мы одно и то же получить не можем, так как тогда получилось бы, что можно пройти 2^{k+1} шагов, а значит вместо первого k, мы бы нашли $k + 1$. А значит всего значений k $O(\log n)$, их можно перебирать в порядке убывания. Сложность ответа на запрос $O(\log n)$.

3. Сведение задачи RMQ к задаче LCA
Если у нас уже есть решение задачи задачи RMQ на отрезке, до построив декартово дерево по неявному ключу на массиве$A[1..N]$ по правилам:

- Корнем дерева является элемент массива, имеющий минимальное значение A, скажем $A[i]$. Если минимальных элементов несколько, можно взять любой
- Левым поддеревом является декартово дерево на массиве $A[1..i-1]$.
- Правым поддеревом является декартово дерево на массиве $A[i+1..N]$.

Тогда $RMQ(i, j) = LCA(A[i], A[j])$. Со сложностью $O(n)$. Доказательство:
Положим $w = LCA(A[i], A[j])$. Заметим, что $A[i]$ и $A[j]$ не принадлежат одновременно либо правому, либо левому поддереву w, потому как тогда бы соответствующий сын находился на большей глубине, чем w, и также являлся предком как $A[i]$ так и $A[j]$, что противоречит определению LCA. Из этого замечанию следует, что w лежит между $A[i]$ и $A[j]$ и, следовательно, принадлежит отрезку $A[i..j]$. По построению мы также знаем, что:

1. Любая вершина дерева имеет свое значение меньшим либо равным значению еѐ детей.

2. Поддерево с корнем в содержит в себе подмассив $A[i..j]$.

Суммируя, получаем, что w имеет минимальное значение на отрезке, покрывающем $A[i..j]$, и принадлежит отрезку $A[i..j]$, отсюда $RMQ(i,j) = w$.

5.3 Метод двоичного подъема

6 Алгоритмы на деревьях

6.1 Декартовы деревья

Декартово дерево — это двоичное дерево, в узлах которого хранятся:

- ссылки на правое и левое поддерево;
- ссылка на родительский узел (необязательно);
- ключи x и y, которые являются двоичным деревом поиска по ключу x и двоичной кучей по ключу y; а именно, для любого узла дерева n:
  - ключи x узлов правого (левого) поддерева больше (меньше либо равны) ключа x узла n;
  - ключи y узлов правого и левого детей больше либо равны ключу y узла n.

Ссылка на родительский узел не обязательна, она желательна только для линейного алгоритма построения дерева.По сути, декартово дерево - это структура данных, объединяющая в себе бинарное кучу и двоичное дерево поиска. Декартово дерево не является самобалансирующемся деревом в обычном смысле (в отличие от красно-черных деревьев).
Преимущества:

1. Легко и быстро реализуется, в отличие от красно-черных деревьев.
2. Для случайного набора ключей y (относительно кучи) хорошо строится.
3. Операция разделить по ключу x выполняется за линейное время.

Недостатки:

1. Большие расходы памяти: в каждой вершине нужно хранить два-три указателя и два ключа.
2. Скорость доступа в худшем случае - O(n), поэтому декартово дерево не применяется в ядрах ОС.

6.1.1 Операции на декартовых деревьях

1. Split Операция Split позволяет разрезать декартово дерево T по ключу k и получить два других декартовых дерева: T_1 и T_2, причем в T_1 находятся все ключи дерева T, не большие k, а в T_2 — большие k.
   Рассмотрим случай, в котором требуется разрезать дерево по ключу, большему ключа корня. Посмотрим, как будут устроены результирующие деревья T_1 и T_2:

   - T_1 : левое поддерево T_1 совпадѐт с левым поддеревом T. Для нахождения правого поддерева T_1, нужно разрезать правое поддерево T на T_1^R и T_2^R по ключу k и взять T_1^R
   - T_2 совпадѐт с T_2^R.

   Случай, в котором требуется разрезать дерево по ключу, меньше либо равному ключа в корне, рассматривается симметрично.
   Псевдокод:
   Split(Treap t, int k, Treap &t1, Treap &t2)
   if t == NULL
   t1 = t2 = NULL;
   else

if k > t.x
Split(t.right, k, t.right, t2);
t1 = t;
else
Split(t.left, k, t1, t.left);
t2 = t;
Оценим время работы операции. Во время выполнения вызывается одна операция для дерева хотя бы на один меньшей высоты и делается ещѐ $O(1)$ операция. Тогда итоговая трудоѐмкость этой операции равна $O(h)$, где h — высота дерева

fix spacing

2. Merge

6.1.2 Декартово дерево по неявному ключу

6.2 Минимальное основное дерево

6.2.1 Алгоритм Прима

Алгоритм построения минимального остовного дерева (дерево, сумма весов ребер которого минимальна)
Вход: взвешенный граф (неориентированный)
Алгоритм:

1. Выбираем некоторую стартовую вершину
2. Из этой вершины строим самый дешевый путь
3. Берем связанные вершины и строим самые дешевые пути из них (не рассматриваем те, которые образуют цикл)
4. Продолжаем до тех пор, пока не свяжем все вершины

Примечания:
• Для быстрого нахождения минимальных путей рекомендуется использовать биномиальную кучу.
• Интерпретация задачи: есть некое множество городов (и расстояний между ними) - нужно построить самую дешевую сеть дорого (самую короткую)
Сложность зависит от способа представления графа и приоритетной очереди:

- Массив d, списки смежности (матрица смежности): $O(|V|^2)$
- Бинарная пирамида, списки смежности: $O(E \log V)$
- Фибоначчиева пирамида, списки смежности: $O(E + V \log V)$

6.2.2 Алгоритм Крускала

Качественный алгоритм для топологической сортировки
Условия: имеем ацикличный ориентированный граф. Упорядочиваем согласно частичному порядку.
Алгоритм:

1. Берем случайную вершину
2. Выполняем из нее DFS со следующей особенностью: черные вершины автоматически заносятся в стек, и помечаются использованными
3. Повторяем п.2 для неиспользованных вершин
4. Извлекаем ответ из стека

Сложность: $O(|E| + |V|)$

7 Минимальные потоки в сети

7.1 Метод Форда-Фалкерсона

Решает задачу нахождения максимального потока транспортной сети.
Алгоритм:

1. Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.
2. В остаточной сети находим любой путь из источника в сток. Если такого пути нет, останавливаемся.
3. 3 Пускаем через найденный путь (он называется увеличивающим путѐм или увеличивающей цепью) максимально возможный поток:
   - На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью c_{min}.
   - Для каждого ребра на найденном пути увеличиваем поток на c_{min}, а в противоположном ему — уменьшаем на c_{min}.
   - Модифицируем остаточную сеть. Для всех рѐбер на найденном пути, а также для противоположных им рѐбер, вычисляем новую пропускную способность. Если она стала ненулевой, добавляем ребро к остаточной сети, а если обнулилась, стираем его.
4. Возвращаемся на шаг 2.

Алгоритм (формально):
Вход: Граф с пропускной способностью, источник и сток
Выход: Максимальный поток f из s в t

1. $f(u,v) \leftarrow 0 \forall (u,v)$
2. Пока есть путь p из s в t в G_f, такой что $c_f(u,v) > 0 \forall (u,v) \in p$:
   (a) Найти $c_f(p) = \min\{c_f(u,v) | (u,v) \in p\}$
   (b) $\forall (u,v) \in p$:
   - $f(u,v) \leftarrow f(u,v) + c_f(p)$
   - $f(v,u) \leftarrow f(v,u) - c_f(p)$

Сложность: $O(E * f)$

7.2 Метод Эдмондса-Карпа (б/д)

8 Алгоритмы на графах

8.1 Обход в ширину и глубину

8.1.1 Обход в глубину

Сложность: $O(|E| + |V|)$

1. Присваиваем всем вершинам белый цвет
2. Берем произвольную белую вершину
3. Красим в серый
   - Если есть белый потомок, переходим в него и goto 3
   - Если нет, красим в черный, переходим к родителю и goto 3a
   - Если нет родителя и есть белые вершины goto 2
   - Не осталось белых – конец алгоритма

8.1.2 Обход в ширину

Сложность: $O(|E| + |V|)$

1. Берем произвольную вергину
2. Добавляем в очередь (пометив как пройденную)
3. Пока очередь не пуста
   - Достаем первую вершин
   - Добавляем в очередь все непройденные потомки
   - goto 3

8.2 Поиск кратчайших путей в графе

8.2.1 Алгоритм Дейкстры

Вход: взвешенный граф (веса положительны), стартовая вершина
Выход: ветор расстояний до стартовой вершины (опционально - вектор предков, чтобы восстановить кратчайший путь)
Алгоритм:

1. Проставляем расстояния: для стартовой 0, для остальных inf
2. Пусть S - множество вершин, до которых ихзвестны кратчайшие пути
3. Добавляем в S вершину с минимальным расстоянием из вектора расстояний и еще не находящуюся в множестве. Помечаем ее, как добавленную.
4. Для всех соседей обновляем оценку:
   - Сосед ib
   - Вершина v
   - Новая оценка для i = min(старая, d(start, v)+d(v,i))
5. Если еще остались неиспользованные goto 2

Сложность: $O(|E| + |V|^2)$

8.2.2 Алгоритм Форда-Беллмана

Алгоритм поиска кратчайшего пути во взвешенном графе
Условие: взвешенный граф, можно с отрицательными весами и циклами (с добавлением дополнительной проверки), стартовая вершина.
Алгоритм (без отрицательных циклов):

1. Для всех вершин проставляем расстояние = inf
2. Для начальной вершины расстояние = 0
3. for i = 1 to (V - 1) $\forall(u, v)$ if $d[v] > d[u] + w(u, v)$ $d[v] = d[u] + w(u, v)$
4. Возвращаем вектор расстояний

Алгоритм (с отрицательными циклами):
Алгоритм Беллмана–Форда позволяет очень просто определить, существует ли в графе G отрицательный цикл, достижимый из вершины s. Достаточно произвести внешнюю итерацию цикла не $|V| - 1$, а ровно $|V|$ раз. Если при исполнении последней итерации длина кратчайшего пути до какой-либо вершины строго уменьшилась, то в графе есть отрицательный цикл, достижимый из s. На основе этого можно предложить следующую оптимизацию: отслеживать изменения в графе и, как только они закончатся, сделать выход из цикла (дальнейшие итерации будут бессмысленны).
Сложность: $O(|E|^*|V|)$

8.2.3 Алгоритм Флойда-Уоршелла

Вход: взвешенный орграф
Выход: матрица длин кратчайших путей.
Алгоритм:
for v = 1 to V
for i = 1 to V
for j = 1 to V
Table[i][j] = min (Table[i][j], Table[i][v] + Table[v][j]);
Сложность: $O(|V|^3)$

8.3 Поиск сильносвязных компонент в графе

ССК - максимальный по включению сильно связанный подграф. Сильно связный граф - орграф, т. ч. из любой вершины можно попасть в любую.
Алгоритм находит ССК по данному графу.
Условия: дается орграф.
Алгоритм (Косарайю):

1. Инвертируем граф (меняем направление всех ребер)
2. Запускаем DFS на транспонированном (инвертированном) графе DFS.
3. Для каждой вершины запоминаем время выхода (шаги), т. е. когда выходим из вершины окончательно (она становится черной)
4. Запускаем DFS на исходном графе, в очередной раз выбирая вершину с максимальным номером (временем выхода)
5. Полученные деревья в п.4 и есть искомые ССК (появляются, когда мы переходим к новой белой вершине в DFS)

Сложность: $O(|E| + |V|)$ для разреженного и $O(|V|^2)$ для плотного графа.

8.4 Мосты и точки сочленения в графе

9 STL и стандартные контейнеры

9.1 vector, deque, queue, priority_queue, set, map

9.2 Итераторы и компараторы