Лекция 3

Алгоритмы, работающие в реальном времени — Online algorithms

(Конспект: Н. Петрова)

3.1 Задача кэширования (paging problem)

Рассмотрим следующую задачу. Есть два вида памяти - дисковая, большая по объему, но медленная, и кэш, быстрая, но очень небольшая. Вся память разбита на блоки. Пусть в кэш-памяти k блоков, а на диске, соответственно, намного больше. К нам поступают запросы на те или иные блоки, и мы должны сразу же их обработать. Есть два варианта: либо этот блок уже есть в кэше, либо его там нет. Если он в кэше есть, то в этом случае мы почти не тратим время. И тогда время работы алгоритма можно измерять количеством обращений к винчестеру.

Алгоритмы отличаются друг от друга реакцией на запросы отсутствующих в кэше блоков. Понятно, что мы должны запрашиваемый блок подгрузить в кэш, непонятно только, какой из уже присутствующих там блоков мы должны для этого выгрузить. Конечно, хотелось бы выгрузить “ненужный” блок, но поскольку мы обрабатываем запросы в реальном времени, мы не знаем, какой блок нам понадобится в дальнейшем.

Встает и другой вопрос: как сравнить, какой алгоритм лучше.

Путь A — алгоритм, а r — последовательность запросов. Стоимость обработки последовательности запросов алгоритмом — $cost_A(r)$ — это количество обращений к диску. Пусть opt — некий оптимальный алгоритм. Тогда алгоритм A называется c-оптимальным (c-competitive), если для любого r, $cost_A(r) \le c \cdot cost_{opt}(r) + c_1$. Подразумевается, что c — константа относительно r, но она может зависеть от k или других параметров. Но это — для детерминированных алгоритмов, для вероятностных же рассматривается на стоимость, а ее математическое ожидание.

Мы будем рассматривать случай, когда A — вероятностный online алгоритм, а *opt* — детерминированный *offline* алгоритм (“oblivious adversary”), т.е. он *заранее* всю знает последовательность запросов r.

Алгоритм 3.1 (MARKER). Элементы кэша помечаются 0 или 1. Все время работы разделяется на периоды. В начале каждого периода все элементы кэша помечены 0.

Приходит запрос. Если это запрос на блок, который уже есть в кэше, то он помечается 1. Если запрашивают элемент, которого нет в кэше, то мы случайным образом выбираем из непомеченных (то есть помеченных 0) блок, куда мы будем подгружать требуемый. Подгрузив, мы помечаем его 1.

Рано или поздно у нас не останется блоков, помеченных 0. В этом состоянии мы можем работать, пока у нас просят блоки, находящиеся в кэше. Как только поступит запрос на элемент, не содержащийся в кэше, мы обнулим все пометки и начнем новый период. □

Теперь попробуем оценить, насколько же хорош этот алгоритм. Разделим все поступающие запросы на три вида: помеченные, устаревшие и чистые. Помеченный запрос — это запрос на блок, образ которого находится в кэше и помечен 1. Устаревший — это запрос на блок, образ которого находится в кэше и помечен 0, или на блок, которого в кэше нет, но который там был в предыдущем периоде. И соответственно, чистый запрос — это запрос на блок, которого в кэше нет и не было в предыдущем периоде.

Обозначим l_i количество чистых запросов за i-тый период; $d_{I,i}$ — разница между кэшами A и *opt*, то есть можность симметрической разности этих двух множеств (количество элементов, входящих ровно в одно из них), в начале i-того периода, а $d_{F,i}$ — в конце.

Оценим снизу количество загрузок, которые сделает *opt*. С одной стороны, это не меньше чем $(l_i - d_{I,i})$, так как l_i блоков алгоритм A обязан подгрузить, и *opt* может выиграть у A только за счет того, что в начале периода у него в кэше уже будут блоки, на которые поступят запросы. А так как разница между кэшами в начале периода составляет $d_{I,i}$ элементов, то и выиграть *opt* может не более $d_{I,i}$ загрузок.

С другой стороны, число загрузок не менее $d_{F,i}$, так как все блоки, лежащие в кэше алгоритма A к концу периода, были запрошены. Следовательно, они должны были побывать и в кэше алгоритма *opt*, и если какого-то из них нет, значит, на его место был загружен другой блок, а нет там ровно $d_{F,i}$ элементов из кэша A.

Таким образом, число загрузок, которые должен сделать оптимальный алгоритм, составляет не менее

$$\max(l_i - d_{I,i}, d_{F,i}) \geq \frac{l_i - d_{I,i} + d_{F,i}}{2}.$$

Это — за один период. Просуммировав по все периодам, получим, что число загрузок, которые должен сделать *opt* за все периоды не менее

$$\sum_i \frac{l_i - d_{I,i} + d_{F,i}}{2} = \sum_i \frac{l_i}{2} + \frac{d_{F,n}}{2} \geq \frac{L}{2},$$

где $L = \sum_i l_i$.

Теперь оценим сверху, сколько загрузок сделает алгоритм A. Ясно, что на чистые запросы ему придется подгружать блоки, на помеченные — нет. Сложнее дело обстоит с устаревшими запросами. Во-первых, сколько таких запросов может быть? Ответ: $k - l$. При устаревшем запросе мы обращаемся к блоку, который на предыдущем периоде был в кэше, но есть ли он там сейчас, мы сказать не можем. Он мог быть выгружен, когда пришел чистый запрос, либо устаревший запрос на блок, который до этого был уже выгружен.

Рассмотрим **первый** устаревший запрос. Поскольку нам надо оценить вероятность сверху, то мы можем рассмотреть наиболее неблагоприятный вариант, когда все чистые запросы произошли до данного. Тогда вероятнось выгрузить нужный нам блок будет наибольшей. Посчитаем, какова вероятность в этом случае оставить наш блок в кэше. При первом чистом запросе мы оставим наш блок с вероятностью $(k-1)/k$, при втором — $(k-2)/(k-1)$ и т.д. Вероятность того, что мы не выгрузим нужный блок при последнем, l-том чистом запросе, равна $(k-l)/(k-l+1)$. Перемножая эти вероятности, получим вероятность того, что мы оставим наш блок в кэше: $(k-l)/k$. Таким образом, вероятность того, что мы выгрузим его, не превосходит l/k.

Общий случай оставляется для самостоятельных раздумий. Именно, утверждается, что вероятность выгрузить нужный нам блок при **j-том** устаревшем запросе не превосходит $l/(k-j+1)$.

Таким образом, математическое ожидание количества загрузок устаревших блоков меньше или равно

$$l/k + \ldots + l/(k-(k-l)+1) = l \cdot (1/k + \ldots + 1/(l+1)) = l \cdot (H_k - H_l + 1),$$

где $H_n = 1 + 1/2 + \ldots + 1/n$. Вспомнив, что мы должны еще загрузить l блоков при чистых запросах, мы получим, что

$$\mathbf{E}(\text{число загрузок}) \leq l \cdot (1 + H_k - H_l + 1) \leq l \cdot H_k.$$

Это за один период. Просуммируем по всем периодам. Получим: $\mathbf{E}cost_A = H_k \cdot L$. Но $cost_{opt} = L/2$, и мы доказали следующую теорему.

Теорема 3.1. *Алгоритм MARKER является $2H_k$-оптимальным.*

Определение 3.1. Рассмотрим ситуацию, когда adversary не только знает все запросы наперед, но и может их придумывать в зависимости от наших действий.

Adaptive offline adversary так и поступает; придумав все запросы, он порождает свой вариант их обработки стоимостью $cost_{opt}$. Заметим, что хотя он порождает новые запросы, зная наши ответы на предыдущие, он не может предугадать наших *будущих* действий, ибо они зависят от выпавших нам случайных чисел. Естественно, в определении *c-оптимальности относительно adaptive offline adversary* фигурируют не все r, а лишь r, порождаемые adversary.

Adaptive online adversary отличается от adaptive offline adversary тем, что он обязан обрабатывать (“порождать свой вариант обработки”) придумываемые им запросы online.

Задача 3.1. Оценить силу алгоритма MARKER относительно adaptive online и adaptive offline или показать, что этого сделать нельзя.

Задача 3.2. Придумать алгоритм лучше.

3.2 k-server problem (задача о k официантах)

Обобщим задачу кэширования. Пусть у нас есть некоторое метрическое пространство с метрикой d, точки которого мы будем рассматривать как столики, и k официантов, то есть некоторое k-элементное подмножество точек этого пространства. Время от времени со столиков поступают заказы. Формально, заказ — это координаты столика. Заказы нужно обслуживать, то есть надо выбрать одного из официантов и переместить его в данную точку пространства. Стоимостью такого перемещения будет расстояние между исходной и конечной его точками. Каждый официант в каждый момент времени находится у какого-то столика, и состояние (конфигурация) системы описывается множеством координат столиков, у которых в данный момент находятся официанты. То есть состояние

X — это мультимножество (множество с кратными элементами), состоящее из k точек пространства. Раз и навсегда зафиксируем начальное состояние A_0 нашей системы.

Пусть $r = r_1 r_2 \dots r_n$ — последовательность заказов. Тогда *рабочей функцией* W_r называется функция, сопоставляющая состоянию X наименьшую стоимость обслужить r из начального состояния A_0, придя в состояние X. (Стоимость обслуживания — это сумма пройденного всеми официантами расстояния, причем наименьший путь определяется с помощью оптимального offline алгоритма, т.е. знающего r.)

Алгоритм 3.2 (WORK FUNCTION ALGORITHM). Предположим, в данный момент наша система находится в состоянии X и к нам приходит запрос r. Тогда мы выбираем новое состояние X', которое содержит r и минимизирует следующую функцию от X': $W_{\rho r}(X') + D(X, X')$, где ρ — это последовательность запросов, которые мы уже обслужили, а $D(X, X')$ — это стоимость (оптимального) перехода из состояния X в X'. (Почему этот минимум достигается, будет ясно из леммы 3.0.) □

Лемма 3.0. *$\exists a \in X : X' = X - a + r$.*

($X - a$ обозначает здесь и в дальнейшем мультимножество $X \setminus \{a\}$, а $Y + b$ — мультимножество $Y \cup \{b\}$.)

Упражнение 3.1. Доказать лемму 3.0.

Введем обозначения: $W := W_\rho$ — "старая" рабочая функция и $W' := W_{\rho r}$ — "новая" рабочая функция. Заметим, что

$$W'(A) = \min_{a \in A}\{W'(A-a+r)+d(r,a)\} = \min_{a \in A}\{W(A-a+r)+d(r,a)\} \geq W(A)$$

для $\forall$ состояния A.

Докажем, что любая рабочая функция обладает свойством квазивыпуклости:

Определение 3.2. W назвается *квазивыпуклой*, если для $\forall$ состояний A и B существует биекция $h : A \to B$ такая, что для любого разбиения A на мультимножества X и Y, справедливо

$$W(A) + W(B) \geq W(X + h(Y)) + W(h(X) + Y). \tag{3.1}$$

Сначала докажем следующую лемму.

Лемма 3.1. *В определении 3.2 можно считать, что $\forall x \in A \cap B$ $h(x) = x$.*

Доказательство. Предположим, что h удовлетворяет условиям из определения квазивыпуклости, но для нее не выполняется утверждение леммы. Построим другую биекцию, которая будет удовлетворять обоим условиям. Построение будем проводить по индукции, постепенно удаляя из $A \cap B$ "плохие элементы".

Итак, пусть $\exists a \in A \cap B : h(a) \neq a$ ($\Rightarrow h^{-1}(a) \neq a$). Рассмотрим новую биекцию h': $h'(a) = a$, $h'(h^{-1}(a)) = h(a)$, и h' совпадает с h на всех остальных точках.

Возьмем произвольное разбиение множества A на X и Y и докажем, что для h' верно неравенство из определения квазивыпуклости. Пусть $h^{-1}(a) \in X$ (если $h^{-1}(a) \in Y$, то доказательство проводится аналогично).

Если $a \in X$, то неравенство верно и для h', поскольку оно верно для h (ведь в этом случае $h(X) = h'(X)$ и $h(Y) = h'(Y)$).

Пусть теперь $a \in Y$. Рассмотрим множества $X' = X + a$, $Y' = Y - a$. Тогда

$$
\begin{aligned}
&W(X + h'(Y)) + W(h'(X) + Y) = \\
&W(X' + h'(Y')) + W(h'(X') + Y') = \\
&W(X' + h(Y')) + W(h(X') + Y') \geq \\
&W(A) + W(B),
\end{aligned}
$$

так как h квазивыпукла, а в определении квазивыпуклости требуется, чтобы неравенство (3.1) выполнялось для любого разбиения, в частности, для X' и Y'. То есть неравенство (3.1) выполнено и для h', а количество элементов a таких, что $h'(a) \neq a$, по крайней мере на 1 меньше, чем для h.

Так как A и B конечны, то за конечное число таких шагов мы исправим h так, чтобы она удовлетворяла утверждению леммы. □

Лемма 3.2. *Все рабочие функции квазивыпуклы.*

Задача 3.3. Доказать лемму 3.2. Указание: доказывать по индукции и использовать лемму 3.1.

Определение 3.3. Состояние A назовем *минимизатором* точки a относительно W, если на нем достигается минимум $W(A) - \sum_{x \in A} d(x, a)$.

Лемма 3.3. *Если A — минимизатор для r относительно W_ρ, то A — минимизатор для r относительно $W_{\rho r}$.*

Задача 3.4. Доказать лемму 3.3.

Оценим, насколько хорош этот алгоритм. Вспомним, что

$$W'(X) = \min_{x \in X} \{W'(X - x + r) + d(x, r)\}.$$

Значит,

$$\exists x_0 \in X : W'(X) = W'(X - x_0 + r) + d(x_0, r) = W'(X') + d(x_0, r).$$

Стоимость одного шага нашего алгоритма при переходе от X к X' равна $d(x_0, r)$, а стоимость одного шага оптимального (offline) алгоритма — не менее $W'(X') - W(X)$. Рассмотрим сумму этих величин $W'(X') - W(X) + d(x_0, r) = W'(X) - W(X)$.

Определение 3.4. Введем характеристику алгоритма — *обобщенную стоимость*, равную $\max_X \{W'(X) - W(X)\}$.

Очевидно, обобщенная стоимость является оценкой сверху на сумму стоимостей нашего и оптимального алгоритмов.

Определение 3.5. *Полная обобщенная стоимость* — это сумма обобщенных стоимостей для всех шагов (запросов).

Лемма 3.4. *Пусть полная обобщенная стоимость* $\leq (c + 1) \cdot$ *полная стоимость оптимального алгоритма* $+ c'$. *Тогда WORK FUNCTION ALGORITHM является c-оптимальным.*

Доказательство. Очевидно. □

Лемма 3.5. *Пусть очередной запрос —* r. *Тогда обобщенная стоимость на этом шаге достигается на минимизаторе для* r.

Задача 3.5. Доказать лемму 3.5.

Теорема 3.2. *WORK FUNCTION ALGORITHM является* $(2k-1)$*-оптимальным.*

Доказательство. Введем

$$\Psi_W(U, B_1, \ldots, B_k) := k \cdot W(U) + \sum_{i=1}^{k} \left(W(B_i) - \sum_{b \in B_i} d(u_i, b) \right),$$

где $U = \{u_1, \ldots, u_k\}$ — состояние, $B_1, \ldots, B_k$ — тоже состояния. Введем также *потенциал*

$$\Phi(W) := \min_{U, B_1, \ldots, B_k} \Psi_W(U, B_1, \ldots, B_k).$$

Оценим $\Phi(W') - \Phi(W)$. Можно считать, что $\exists j$: $r = u_j$. Предположим, что это не так, но $\exists i$: $W'(U) = W'(U - u_i + r) + d(r, u_i)$. Заменим u_i на r. Тогда $\Psi_{W'}(U, \dots)$ уменьшится на

$$d(u_i, r) \cdot k + \sum_{b \in B_i} d(u_i, b) - \sum_{b \in B_i} d(r, b).$$

Это выражение неотрицательно по неравенству треугольника $\Rightarrow$ $\Psi_{W'}$ разве что уменьшится, т.е. можно считать, что минимум $\Psi_{W'}$ достигается именно на таком U (содержащем r). Также можно считать, что B_j — минимизатор для r. Остальные элементы, минимизирующие $\Psi_{W'}(\dots)$ зафиксируем, и именно их будем подразумевать, обозначая многоточием.

Тогда

$$\Phi(W') - \Phi(W) \geq \Psi_{W'}(\dots) - \Psi_W(\dots),$$

так как $\Phi(W') = \Psi_{W'}(\dots)$, а $\Phi(W) \leq \Psi_W(\dots)$: поскольку

$$\Phi(W) = \min_{V, C_1, \dots, C_k} \Psi_W(V, C_1, \dots, C_k) \leq \Psi_W(\dots).$$

Далее, вспомним, что

$$\begin{aligned}
\Psi_{W'} &= k \cdot W'(U) + \sum (W'(B_i) - \sum d(u_i, b)), \\
\Psi_W &= k \cdot W(U) + \sum (W(B_i) - \sum d(u_i, b))
\end{aligned}$$

и будем подставлять, что знаем, в их разность. Поскольку $r \in U$, имеем $W'(U) = W(U)$. Разность всех элементов внешних сумм кроме j-тых оцениваем нулем. Итого, $\Phi(W') - \Phi(W) \geq W'(B_j) - W(B_j)$, а это по лемме 3.5 — обобщенная стоимость, так как B_j — минимизатор. Таким образом, полная обобщенная стоимость не превосходит

$$\Phi(W_{r_1}) - \Phi(W_\varepsilon) + \Phi(W_{r_1 r_2}) - \Phi(W_{r_1}) + \dots = \Phi(W_\pi) - \Phi(W_\varepsilon),$$

где ε — пустая последовательность, а π — последовательность всех заказов. Нетрудно доказать, что $\Phi(W_\varepsilon)$ для фиксированного A_0 — константа.

Упражнение 3.2. Доказать это.

С другой стороны,

$$\begin{aligned}
&\Phi(W_\pi) = \min \Psi_{W_\pi} \leq \Psi_{W_\pi}(A_n, \dots, A_n) = \\
&k \cdot W_\pi(A_n) + \sum_{i=1}^{k} \left(W_\pi(A_n) - \sum_{a \in A_n} d(a_{ni}, a) \right) \leq \\
&2k \cdot W_\pi(A_n).
\end{aligned}$$

Вспомним, что $W_\pi(A_n)$ — это стоимость работы offline алгоритма на нашей последовательности запросов π. Применяя лемму 3.4, получаем требуемое. $\square$