ЗАДАЧИ РАСПРЕДЕЛЕНИЯ В РЕАЛЬНОМ ВРЕМЕНИ ЛЕКЦИЯ 13 КУРСА «ТЕОРИЯ ЭКОНОМИЧЕСКИХ МЕХАНИЗМОВ»

СЕРГЕЙ НИКОЛЕНКО

Contents

1. Задача распределения

1.1. Постановка и верхняя оценка. Суть задачи

Рассмотрим проблему распределения задач в реальном времени для одного процессора. Вопрос заключается в том, чтобы последовательно выбирать, над какой из доступных в настоящий момент задач работать. Очевидно, что все задачи выполнить не получится; при поступлении задачи надо решать, что с ней делать. Задача — это пара (e, d): время исполнения и дедлайн. Если задача поступила в момент t_0, мы можем получить e единиц «дохода», выделив e слотов процессорного времени внутри интервала $t_0 \le t \le t_0 + d$. Таким образом, мы хотим максимизировать суммарное время полезной работы процессора. Основной трудностью является тот факт, что распределение задач должно строится в реальном времени, несмотря на то, что о каждой конкретной задаче ничего неизвестно до момента ее поступления.

Для решения проблемы необходим онлайн-алгоритм, который будет реагировать на поступающие задачи. Стандартным приемом для анализа онлайн-алгоритмов является сравнение с оптимальным ясновидящим алгоритомом — офлайн-алгоритмом, который обладает полной информацией о входных данных уже в момент своего запуска. Одной из интерпретаций такого подхода является игра между создателем онлайн-алгоритма и его противником. Вначале выбирается онлайн-алгоритм, затем противник подбирает такую последовательность задач (наихудшую для результата работы онлайн-алгоритма), чтобы максимизировать отношение оптимальности — отношение числа задач, выполненных офлайн-алгоритом, к числу задач, выполненных онлайн-алгоритмом. Онлайн-алгоритм r-оптимален, если он достигнет доли как минимум r от результата оптимального алгоритма.

Date: 25 апреля 2008 г.
Законспектировал Андрей Клебанов.

Пример и верхняя оценка

Будем считать, что временной промежуток между поступлением задачи и ее дедлайном равен времени исполнения задачи. Тогда для получения верхней оценки воспользуемся следующей идей. Для большой задачи S_i длиной L_i со временем прихода a_i, в тот же момент создадим последовательность маленьких задач, стоящих ϵ и начинающихся во время дедлайна предыдущей задачи в последовательности. Противник перестает поставлять маленькие задачи в тот момент, когда алгоритм решает прекратить выполнение большой задачи S_i и начать выполнение маленькой. Таким образом, выигрыш составит всего ϵ, а не L_i. В том случае, когда большая задача сопровождается последовательностью описанных выше маленьких задач, она называется приманкой.

Введем некоторые обозначения. Время делится на эпохи. В начале каждой эпохе противник создает большую задачу T_0 длины $t_0 = 1$. Потом, за ϵ до конца большой задачи T_i длиной t_i создает задачу T_{i+1} длиной t_{i+1}. Если алгоритм решает переключится на выполнение маленькой задачи, то эпоха заканчивается в момент поступления задачи T_{i+1}. Если же алгоритм решает переключится на выполнение задачи T_{i+1}, то процесс продолжается дальше, в противном случае эпоха заканчивается в момент поступления задачи T_{i+1}. Ни одна эпоха не продолжается дольше момента поступления задачи T_m для некоторого конечного числа m. Как следует из приведенного выше описания, все задачи, кроме T_{i+1}, приманки.

Заметим, что, во-первых, игрок ни разу не оставит приманку ради маленькой задачи, потому что она даст ему выигрыш за всю эпоху не превышающий ϵ. Во-вторых, можно сделать вывод о том, что в течение одной эпохи игрок получит либо стоимость одной задачи T_i, $i < m$, либо — задачи T_m.

Введём такие последовательности времён:

$$t_{i+1} = (ct_i - \sum_{j=0}^{i} t_j),\ c\text{— некая константа, которая будет определена в дальнейшем.}$$

Тогда, если игрок выполнил задачу T_i, то он получает выигрыш t_i, а противник — $\sum_{j=0}^{i+1} t_j$ (он выполнит все маленькие задачи до T_{i+1} и ее саму).

В этом случае отношение получается равным:

$$\frac{t_i}{\sum_{j=0}^{i+1} t_j} = \frac{t_i}{ct_i - \sum_{j=0}^{i} t_j + \sum_{j=0}^{i} t_j} = \frac{1}{c}.$$

А если игрок выполнил T_m, то он получает t_m, а противник — $\sum_{j=0}^{m} t_j$. Осталось выбрать c и m так, чтобы $\frac{t_m}{\sum_{j=0}^{m} t_j}$ тоже не превышало $\frac{1}{c}$. Т.е. надо найти максимальное c такое, что функция $t : \mathbb{N} \to \mathbb{N}$, заданная рекуррентным соотношением

$$t_0 = 1, \quad t_{i+1} = (ct_i - \sum_{j=0}^{i} t_j),$$

удовлетворяла бы условию

$$\exists m \geq 0 : \frac{t_m}{\sum_{j=0}^{m} t_j} \leq \frac{1}{c}.$$

Заметим, что $\exists m \geq 0 : \frac{t_m}{\sum_{j=0}^{m} t_j} \leq \frac{1}{c}$ эквивалентно $\exists l \geq 0 : t_{l+1} \leq t_l$.

Действительно, $\frac{t_m}{\sum_{j=0}^{m} t_j} = \frac{t_m}{t_m + \sum_{j=0}^{m-1} t_j} = \frac{t_m}{ct_{m-1}}$. Далее, $t_{i+2} = ct_{i+1} - \sum_{j=0}^{i+1} t_j = ct_{i+1} - t_{i+1} - \sum_{j=0}^{i} t_j = ct_{i+1} - t_{i+1} + t_{i+1} - ct_i = c(t_{i+1} - t_i)$. Таким образом, рекуррентное условие эквивалентно

$$t_0 = 1, \quad t_1 = c - 1, \quad t_{i+2} = c(t_{i+1} - t_i).$$

Его характеристическое уравнение $x^2 - cx + c = 0$ имеет корни $x_{1,2} = \frac{c \pm \sqrt{c^2 - 4c}}{2}$.

Если $c = 4$, то $t_i = i2^{i-1} + 2^i$. Очевидно, что в этом случае условие $t_{l+1} \leq t_l$ не выполняется. Аналогично можно проверить, что необходимое неравенство выполняется только при $c < 4$.

1.2. **Нижняя оценка, равная верхней.** 1/4-**оптимальный алгоритм** TD_1

Покажем, что верхняя оценка 1/4 достижима — существует онлайн-алгоритм TD_1, который выполнит как минимум четверть задач для любой входной последовательности.

Перед рассмотрением самого алгоритма введем необходимые обозначения. Будем считать, что все задачи имеют нулевую laxity — это означает, что исполнение задачи надо либо начинать сразу в момент ее поступления, либо отказываться от него. Подобное предположение позволит значительно упростить алгоритм.

Задача T_i описывается четверкой $T_i = (a_i, c_i, d_i, v_i)$:

- a_i — arrival time, время прихода задачи;
- c_i — computation time, необходимое время для ее исполнения;
- d_i — deadline, время, к которому задача должна быть решена;
- v_i — value, поощрение за исполнение задачи в срок.

Обозначим $l_i = d_i - c_i$ самое позднее время начала исполнения задачи i. Заметим, что в случае нулевой laxity, оно равно нулю.

Введем определение (временного) интервала. Пусть t обозначает текущее время. Тогда интервал в t — это промежуток времени $[t_b, t_e)$, в котором есть занятый подпромежуток $[t_b, t_f]$, а за ним (возможно) подпромежуток простоя $[t_f, t_e)$, где $t_b \leq t$ — время, когда система начала работать, t_f $(t \leq t_f)$ — время, когда система снова перейдёт (ожидается, что перейдёт) в нерабочее состояние, потому что исполнение задачи завершится, и

$$t_e = \max(t_f, \max(d_{disc})),$$

где d_{disc} — дедлайны задач, от которых придётся отказаться на протяжении $[t_b, t_f]$.

Интервал *замкнутый*, если на нём завершилось выполнение задачи, *открытый* — в противном случае.

Рассмотрим интервал Δ. Его размер зависит от задач. Он растёт, поглощая новые открытые интервалы, пока не станет замкнутым.

Рассмотрим последовательность поступающих задач

$$(T_{a_1}, T_{a_2}, \ldots, T_{a_n}) \text{ и соответствующие интервалы } (\Delta_{a_1}, \Delta_{a_2}, \ldots, \Delta_{a_n}),$$

где Δ_{a_i} — интервал, в котором задача T_{a_i} поступила на вход алгоритму, а $\Delta_n = \Delta$.

Обозначим

$$(T_1, T_2, \ldots, T_k)$$

задачи, которые реально выполняются системой, причём T_i прерывает предыдущую задачу, и

$$(\Delta_1, \Delta_2, \ldots, \Delta_k), \ —$$

соответствующие интервалы. Последовательности интервалов монотонно возрастают в каждом случае.

Опишем алгоритм TD_1 с учетом предположения о нулевой laxity. Алгоритм работает следующим образом: когда приходит новая задача T_{next},

- обновить Δ_{run} (текущий интервал);
- если $v_{run} < \Delta_{run}/4$, то $T_{run} = T_{next}$.

Т.е. если новая стоимость вчетверо превышает текущий размер интервала, необходимо взять новую задачу.

Докажем, что этот алгоритм получает, по крайней мере, четверть от выигрыша оптимального алгоритма. Для этого докажем по индукции вспомогательное утверждение: $v_k > \frac{\Delta_k}{2}$.

База: $k = 1$ $v_1 = \Delta_1 > \Delta_1/2$.

Переход: пусть утверждение $v_i > \frac{\Delta_i}{2}$ верно для $k = i$, докажем для $i + 1$. Поскольку T_i прерывается T_{i+1}, то $v_i < \Delta_{i+1}/4$ и $\Delta_{i+1} < \Delta_i + v_{i+1}$. Наконец, $2v_{i+1} = v_{i+1} + v_{i+1} > v_{i+1} + \Delta_{i+1} - \Delta_i > v_{i+1} + 4v_i - \Delta_i > v_{i+1} + 2\Delta_i - \Delta_i = v_{i+1} + \Delta_i > \Delta_{i+1}$.

Для завершения доказательства рассмотрим два случая. Первый случай: $T_{a_n} = T_k$, т.е. после T_k никаких задач не приходило. Тогда просто $v_k > \frac{\Delta_k}{2} = \frac{\Delta}{2} > \frac{\Delta}{4}$. Второй случай: после T_k ещё приходили задачи, но они были отброшены, следовательно, $v_k \geq \frac{\Delta_{a_n}}{4} = \frac{\Delta}{4}$. $\square$

Теперь рассмотрим ситуацию, когда у задач есть laxities. Очередь Q используется для хранения ожидающих выполнения задач, отсортированных по времени l_i (самого позднего времени старта). Как только приходит новая задача, она помещается в очередь. Если система свободна, то она начинает выполнять первую задачу из очереди. В противном случае решение о выполнении задачи принимается в момент самого позднего старта первой задачи из очереди. Таким образом, первая задача в интервале имеет laxity, а остальные уже нет. Если алгоритм выполнил задачу с ненулевой laxity, а какие-то другие срочные задачи отклонил, то ясновидящий алгоритм мог приостановить выполнение задачи (наличие laxity позволяет это сделать), выполнить срочные, а потом вернуться к выполнению первой задачи, и тем самым обогнать онлайн-алгоритм. Для решения этой проблемы введём переменную pl (potential loss) — стоимость первой задачи в каждом интервале.

Алгоритм теперь такой: когда приходит новая задача, она помещается в очередь Q. Когда система освобождается и Q непуста, $T_{run} = \text{dequeue}(Q)$, $pl = v_{run}$. Когда система работает и звучит будильник (наступает l некоторой задачи из Q):

- $T_{next} = \text{dequeue}(Q)$; $\text{update}(\Delta_{run})$.
- Если $v_{run} < (\Delta_{run} + pl)/4$, то $T_{run} = T_{next}$.

Для этого алгоритма докажем, что он получает как минимум четверть от выигрыша оптимального алгоритма.

Рассмотрим такие же последовательности задач и интервалов T_i, T_{a_i}, Δ_i, Δ_{a_i}, что и в случае отсутствия laxity.

Сначала по индукции докажем, что $v_k \geq \frac{\Delta_k + pl}{2}$.

База: $k = 1$ $v_1 = \Delta_1 = (\Delta_1 + pl)/2$.

Переход: пусть утверждение $v_i > \frac{\Delta_i+pl}{2}$ верно для $k=i$, докажем для $i+1$. Поскольку T_i прерывается T_{i+1}, то $v_i < \Delta_{i+1}+pl/4$ и $\Delta_{i+1} < \Delta_i+v_{i+1}$. Наконец, $2v_{i+1} = v_{i+1}+v_{i+1} > v_{i+1}+\Delta_{i+1}-\Delta_i > v_{i+1}+4v_i-pl-\Delta_i > v_{i+1}+2(\Delta_i+pl)-pl-\Delta_i = v_{i+1}+\Delta_i+pl > \Delta_{i+1}+pl$.

Для завершения доказательства опять рассмотрим два случая. Первый случай: $T_{a_n} = T_k$, т.е. после T_k никаких задач не приходило. Тогда просто $v_k > \frac{\Delta_k+pl}{2} = \frac{\Delta+pl}{2} > \frac{\Delta+pl}{4}$. Второй случай: после T_k ещё приходили задачи, но они были отброшены, следовательно, $v_k \geq \frac{\Delta_{a_n}+pl}{4} = \frac{\Delta}{4}$. □

2. Дизайн механизмов для задачи распределения

2.1. **Постановка. Зачем применять дизайн механизмов**

Рассмотрим работу алгоритма TD_1 для трех задач:

- $a_1 = 0$, $d_1 = 0.9$, $c_1 = 0.9$, $v_1 = 0.9$;
- $a_2 = 0.5$, $d_2 = 5.5$, $c_2 = 4$, $v_2 = 4$;
- $a_3 = 4.8$, $d_3 = 17$, $c_3 = 12.2$, $v_3 = 12.2$.

Сначала алгоритм выполнит T_1, а потом начнёт выполнять T_2. В момент времени 4.8, когда надо будет принимать решение о выполнении T_3, он подсчитает, что

$$\frac{t_e - t_b + pl}{4} = \frac{17 - 0.9 + 4}{4} > 4 = v_2,$$

и начнёт выполнение третьей задачи. Дизайн механизмов можно применить следующим образом. Например, если агент, которому принадлежит вторая задача, соврет про ее дедлайн и объявит его равным $\hat{d}_2 = 4.7$, то в момент 0.7 эта задача будет иметь нулевую laxity. Алгоритм должен будет принимать решение о ее выполнении, он получит, что

$$\frac{t_e - t_b + pl}{4} = \frac{4.7 - 0 + 1}{4} > 0.9 = v_1,$$

и начнет выполнять вторую задачу, которая успеет завершиться до поступления третьей.

Формулировка

Есть центр, контролирующий процессор, и N агентов, число агентов изначально неизвестно центру. Каждый агент владеет одной задачей i. Характеристики задачи определяют тип агента θ_i. В момент a_i агент i узнаёт свой тип θ_i и, начиная с этого времени, может предлагать задачу процессору. Он делает это, объявляя $\hat{\theta}_i = (\hat{a}_i, \hat{d}_i, \hat{c}_i, \hat{v}_i)$, а затем функция $g : \Theta \to O$ выбирает исход, т.е. расписание. Мы не будем отдавать задачу обратно агенту до его объявленного дедлайна $\hat{d}_i$, даже если выполнили ее раньше этого срока. Это решение является важным для описываемого механизма.

Полезность для каждого агента определяется следующим образом:

$$u_i(g(\hat{\theta}), \theta_i) = v_i \mu(e_i(\hat{\theta}, d_i) \geq c_i)\mu(\hat{d}_i \leq d_i) - p_i(\hat{\theta}),$$

где μ — индикаторная функция своего аргумента, p_i — выплата, которую должен сделать агент, e_i — количество времени на текущий момент, которое процессор проработал над задачей i. Полезность — квазилинейная функция от поощрения за выполнение задачи в срок и от выплаты агента центру. Мы предполагаем, что каждый агент стремится максимизировать свою полезность. Агент не может декларировать время исполнения задачи меньше настоящего, т.к. центр это заметит, и не может дать задачу

центру до наступления момента a_i потому, что сам еще ничего о ней не знает. Таким образом, агент может предложить центру $\hat{\theta}_i = (\hat{a}_i, \hat{d}_i, \hat{c}_i, \hat{v}_i)$, где $\hat{a}_i \geq a_i$, $\hat{c}_i \geq c_i$.

Заметим, что в этой постановке задачи v_i и c_i разные. Поэтому необходимо знать верхнюю оценку на отношение $\frac{v_i}{c_i} = \rho_i$, называемое плотностью. Пусть $\rho_{\min} = \min_i \rho_i = 1$, а $\rho_{\max} = \max_i \rho_i = k$. Наш механизм будет $((1+\sqrt{k})^2 + 1)$-оптимальным.

2.2. **Механизм и его анализ.** В отличие от алгоритма TD_1 механизм не дает предпочтений активной задаче (TD_1 требовал, чтобы новая задача была вчетверо выгоднее). Механизм просто исполняет работу с максимальным приоритетом

$$\hat{v}_i + \sqrt{k} e_i(\hat{\theta}, t)\rho_{\min}.$$

С агента, чью задача выполнилась, центр берёт сумму по правилу «второй цены»: минимальное v (при неизменных остальных параметрах), которое он мог бы заявить так, чтобы его работа всё же выполнилась. Тогда сразу, как в Викри-аукционах или VCG, автоматически получится рациональность и правдивость относительно стоимостей v_i. Осталось понять, почему механизм правдив относительно трёх других параметров: a_i, c_i и d_i. Во-первых, улучшать (уменьшать) a_i и c_i мы уже запретили. Во-вторых, улучшать (увеличивать) d_i тоже бессмысленно: агенту отдадут работу, когда будет уже поздно. Именно для этого нужно было возвращать задачу не сразу после выполнения, а в момент заявленного дедлайна. Остается неочевидным, почему агенту невыгодно ухудшать параметры работы, делать её строже. Опишем идею доказательства. Если увеличивать время вычисления задачи, то единственный эффект от этого — задержка выполнения или вообще отказ от выполнения (приоритет задачи ухудшается). Если приближать дедлайн, то можно добиться, что задачу выполнят раньше, но в нашей постановке полезность агента от этого не увеличится, наоборот увеличится вероятность того, что от задачи откажутся. Не так очевидно, почему нет смысла отодвигать $_i$ (время объявления работы центру), но это тоже можно строго обосновать.

Осталось доказать требуемою степень оптимальности механизма. Это делается примерно так же, как и в случае анализа алгоритма TD_1. Разобьём время на интервалы $(t_i^o, t_i^c]$, где в момент t_i^c завершается выполнение задачи i, а в момент t_i^o завершается задача $i-1$ для $i \geq 2$ и $t_1^o = 0$ для $i = 1$. Пусть время t_i^b — это первый момент, когда на интервале Δ_i процессор работает.

Докажем вспомогательную лемму. Для любого интервала Δ_i верно следующее неравенство:

$$t_i^c - t_i^b \leq \left(1 + \frac{1}{\sqrt{k}}\right) v_i.$$

Доказательство. Обозначим t_j^s момент начала выполнения некоторой задачи j. На промежутке $[t_i^c; t_i^b]$ приоритет $v_i + \sqrt{k} e_i(\theta, t)$ активной задачи монотонно растет, потому что растет количество времени, которое процессор проработал над задачей (т.к. она активна). Пусть на интервале Δ_i процессор последовательно выполняет задачи $1, 2, \ldots, c$, тогда любая задача $j > 1$ из этой последовательности начинает выполняться в момент a_j, потому что ее приоритет не растет, пока она не активна. Покажем, что стоимость выполненной задачи c превосходит произведение $\sqrt{k}$ и времени, потраченного на задачи $1 \ldots c-1$. Формально: $v_c \geq \sqrt{k} \sum_{h=1}^{c-1} e_h(\theta, t_{h+1}^s) - e_h(\theta, t_h^s)$. Для доказательства этого неравенства мы докажем по индукции более сильное утверждение $v_i \geq \sqrt{k} \sum_{h=1}^{i-1} e_h(\theta, t_{h+1}^s)$ для любой задачи i из последовательности.

База: пусть $i = 1$. Тогда $v_1 \geq \sqrt{k} \sum_{h=1}^{0} e_h(\theta, t_{h+1}^s) = 0$ верно, потому что в сумме нет слагаемых.

Переход: пусть утверждение верно $v_i \geq \sqrt{k} \sum_{h=1}^{i-1} e_h(\theta, t_{h+1}^s)$ для i, докажем для $i+1$. В момент $t_{i+1}^s = a_{i+1}$ выполняется $v_{i+1} \geq v_i + \sqrt{k} e_i(\theta, t_{i+1}^s)$. Просуммировав эти два неравенства, получим, что $v_{i+1} \geq \sqrt{k} \sum_{h=1}^{i} e_h(\theta, t_{h+1}^s)$.

Из предположений о том, что минимальная плотность $\rho_{\min} = 1$ и о том, что время выполнения задачи процессором не превосходит ее длины, следует $t_i^c - t_c^s \leq c_c \geq v_c$. Замтим, что мы считаем, что на интервале Δ_i выполнилась задача c, т.е. задача c в описанной выше последовательности — это задача i в глобальной последовательности выполненных задач. Тогда $t_i^c - t_i^b = t_i^c - t_c^s + \sum_{h=1}^{c-1} e_h(\theta, t_{h+1}^s) - e_h(\theta, t_h^s) \geq (1 + \frac{1}{\sqrt{f}}) v_i$. $\square$

Теперь докажем, что в каждом интервале мы не отбрасываем слишком дорогие задачи. Формально: для каждого интервала Δ_i и задачи j, которая была в нем отброшена, верно неравенство:

$$v_j \leq (1 + \sqrt{k}) v_i.$$

Докажем от противного. Предположим, что существует такая задача, что

$$v_j > (1 + \sqrt{k}) v_i.$$

В момент t_i^c для приоритета задачи j верно $v_j + \sqrt{k} c_j < (1 + \sqrt{k}) v_j$. Приоритет активной задачи монотонно возрастает на промежутке $[t_i^o, t_i^c]$, следовательно, задача j будет иметь больший приоритет чем активная задача где-то в пределах этого промежутка. По монотонности получаем, что приоритет активной задачи в момент t_i^c будет превосходить $(1 + \sqrt{k}) v_i$, что противоречит тому факту, что он должен быть равен $(1 + \sqrt{k}) v_i$. $\square$

Дальнейший анализ опустим, но в результате получится требуемая оптимальность.

В общем, итог такой: дизайн механизмов помог разработать такую систему, в которой агенты заинтересованы в выполнении своих задач, но врать им при этом незачем.