Московский физико-технический институт
(Национальный исследовательский университет)

Проект по курсу
«Сложности вычислений»

Задача об оптимальном расписании

Студент:
Хорунжий И. Б.

Преподаватель:
Букреев Ф. С.

20 декабря 2022 г.

Содержание

1 Аннотация

В данной работе исследована задача об оптимальном расписании. Суть ее заключается в построении распределения работ по машинам так, чтобы все работы были выполнены и чтобы конечное время выполнения всех работ было минимально с учетом того, что дано множество работ, машин, и производительность всех машин одинакова. В теоретической части доказана **NP**-полнота языка $\{ (J, m, k) \mid$ на m машинах можно выполнить все работы из J за время не более $k \}$ при помощи сводимости к **NP**-полному языку SUBSETSUM. Был написан алгоритм, который дает $\frac{4}{3}$ - приближение для случая машин одинаковой производительности, проверен результат работы на различных тестах, сделаны выводы по полученным данным.

2 Введение

2.1 Историческая справка

Теория расписаний — раздел дискретной математики, занимающийся проблемами упорядочения. В общем случае проблемы формулируются так: Задано некоторое множество работ (требований) $J = \{J_1, J_2, \ldots, J_n\}$, с определённым набором характеристик: длительность обработки требования (простейший случай), стоимость обработки требования, момент поступления требования, директивный срок окончания обслуживания требования. Задано некоторое множество машин (приборов) $M = \{M_1, M_2, \ldots, M_m\}$, на которых требования должны обслуживаться в соответствии с некоторым порядком.

Ставится задача дискретной оптимизации: построить расписание, минимизирующее время выполнения работ, стоимость работ и т. п. Расписание — указание, на каких машинах и в какое время должны обслуживаться требования (выполняться работы).

Задачи теории расписаний можно разделить на две группы:

1. Задачи с прерываниями. В любой момент обслуживание требования на машине может быть прервано (с возможностью завершения позже на той же или другой машине) ради обслуживания другого требования.

2. Задачи без прерываний — каждое требование на машине обслуживается от начала до конца без прерываний.

Существуют различные варианты задач теории расписаний, часть из них является **NP**-полными, часть принадлежит к классу полиномиальных задач, для части задач так и не удалось доказать принадлежности к какому-либо классу сложности. Существует гипотеза, что задача, допускающая прерывания, не сложнее задачи без прерываний. Для большинства задач она соблюдается, кроме одной, где для варианта без прерывания доказана его принадлежность к классу полиномиальных задач, в то время как для аналогичной задачи с прерываниями не существует доказательств принадлежности к какому-либо классу сложности.

2.2 Постановка задачи

В нашем случае будет рассматривать простейший случай задачи построения оптимального расписания, когда у нас дано только время исполнения каждой работы на определенной машине, и каждая задача на машине работает без прерываний. Имеется множество работ J и множество машин M. Также задана функция p: $J \times M \to \mathbb{R}_+$. Значение p_{ij} означает время выполнение i-ой работы на j-ой машине. Требуется построить распределение работ

по машинам так, чтобы все работы были выполнены и чтобы конечное время выполнения всех работ было минимально. То есть требуется найти функцию $x : J \times M \to \{0, 1\}$ такую, что:

$$\max_{j \in M} \sum_{i} x_{ij} p_{ij} \to min$$

$$\forall i \sum_{j \in M} x_{ij} = 1,$$

В дальнейшем будем рассматривать частный случай задачи, когда производительности всех машин одинаковы, то есть $p_{ij} = p_i$.

Далее работа состоит из двух частей:

1. Доказательство, что задача об оптимальном расписании является NP-полной (подразумевается задача поиска расписания длительности не более k).

2. Построение алгоритма, дающего $\frac{4}{3}$-приближение для случая машин одинаковой производительности, а также его имплементация.

   Обозначим язык $\{ (J, m, k) \mid$ на m машинах можно выполнить все работы из J за время не более $k \}$ как MULTIPROCESSOR-PLANNING

3 NP-полнота

3.1 Принадлежность к NP

Одним из определением класса **NP** является определение на языке сертификатов.

Definition: Будем говорить, что y является сертификатом принадлежности x языку Q, если существует полиномиальное отношение (верификатор) R, такое что $R(x, y) = 1$ тогда и только тогда, когда x принадлежит Q.

Statement: MULTIPROCESSOR-PLANNING $\in$ **NP**

Proof: В качестве сертификата рассмотрим вектор размера n чисел от 1 до m, где n - количество работ, m - количество машин. Этот вектор обозначает соответсвие всех работ машинам. Верификатор же будет при данном сертификате считать время, затраченное каждой машиной. Если для каждой машины при заданном распределении работ затраченное время не более k, тогда верификатор возвращает 1, иначе - 0. Эта проверка выполняется не более, чем за полиномиальное время.

3.2 Язык SUBSETSUM

Для дальнейшего доказательства **NP** - полноты языка MULTIPROCESSOR-PLANNING нам потребуется язык SUBSETSUM , который определяется следующим образом: $\{(n_1, n_2, ..., n_k, N) \mid$ из набора чисел $n_1, ..., n_k$ можно выбрать набор n_i с суммой $N\}$

Definition: **NP** -полная задача - в теории алгоритмов задача с ответом «да» или «нет» из класса NP, к которой можно свести любую другую задачу из этого класса за полиномиальное время.

Statement: Задача SUBSETSUM является **NP** - полной задачей.

Proof: Ссылка

3.3 Сводимость к SUBSETSUM

Statement: SUBSETSUM $\leq_p$ MULTIPROCESSOR-PLANNING

Proof: Нам нужно построить функцию сводимости f, чтобы доказать **NP** - полноту MULTIPROCESSOR-PLANNING. Рассмотрим SUBSETSUM как частный случай MULTIPROCESSOR-PLANNING для двух машин. Имеется три варианта:

1. $\sum_{1 \leq i \leq k} n_i = 2N$

Тогда f задает соответствие из $(n_1, n_2, ..., n_k, N)$ в $((n_1, n_2, ..., n_k), 2, N)$. То есть если у нас найдется набор n_i с суммой N, отдадим этот набор задач первой машине, оставшиеся задачи - второй. Тогда каждая отработает за время N. Если же такой набор не нашелся, тогда найдется набор с суммой $\geq N+1$. Это означает, что время работы на какой-то машине будет $\geq N+1$, то есть такие задачи не принадлежат языку MULTIPROCESSOR-PLANNING. Значит, если есть решение в SUBSETSUM , то и есть решение в MULTIPROCESSOR-PLANNING, и наоборот.

2. $\sum_{1 \leq i \leq k} n_i < 2N$

   Тогда f действует из $(n_1, n_2, ..., n_k, N)$ в $\left((n_1, n_2, ..., n_k, 2N - \sum_i n_i), 2, N\right)$. То есть создаем еще одну задачу с оставшимся временем. Повторяем рассуждения из первого пункта: если есть набор n_i с суммой N, его отдаем первой машине. Второй машине приходит набор с оставшимися n_i вместе с дополнительной задачей $2N - \sum_i n_i$. Понятно, что набор таких задач принадлежит языку, так как обе машины отработают за время $\geq N$. Если не нашлось набора с суммой N, делаем выводы, как в пункте 1.

3. $\sum_{1 \leq i \leq k} n_i > 2N$

   Тогда f из $(n_1, n_2, ..., n_k, N)$ в $\left((n_1, n_2, ..., n_k, \sum_i n_i - 2N), \ 2, \ \sum_i n_i - N\right)$. Если есть решение в MULTIPROCESSOR-PLANNING, то у нас есть 2 набора в каждом из которых сумма - $\sum_i n_i - N$. В одном из наборов содержится n_{k+1}. Без него в наборе остаются элементы в сумме дающие $\sum_i n_i - N - (\sum_i n_i - 2N) = N$. Таким образом, мы нашли набор, который дает в сумме N. В случае отстутвия решения в MULTIPROCESSOR-PLANNING выводы аналогичные предыдущим.

NP - полнота MULTIPROCESSOR-PLANNING доказана.

4 Алгоритм

4.1 Описание алгоритма

Построим алгоритм следущим образом:

1. Получим массив задач, отсортированный по убыванию времени их выполнения.
2. Изначально текущее время для всех машин равно нулю.
3. При последовательной итерации по этому массиву задачу отдаем машине, имеющей наименьшую нагрузку, то есть текущее время для этой машины наименьшее.
4. Добавляем этой машине время задачи, которую мы ей отдали.

```
def find_approximate_schedule(number_of_machines: int,
                              times_of_tasks: list):
times_of_tasks.sort(reverse=True)
machines = [[0, machine_num] for machine_num
in range(number_of_machines)]
heapq.heapify(machines)
optimal_distribution = [[] * _ for _ in range(number_of_machines)]
for time in times_of_tasks:
    less_busy_machine = heapq.heappop(machines)
    less_busy_machine[0] += time
    optimal_distribution[less_busy_machine[1]].append('*' *
                                                      time + '|')
    heapq.heappush(machines, less_busy_machine)
optimal_scheduling_time = max(machines)[0]
return optimal_distribution, optimal_scheduling_time
```

4.2 Доказательство $\frac{4}{3}$ приближения алгоритма

Доказательство имеется здесь 421 - 426 стр.

Для убеждения в том, что реализованный алгоритм действительно имеет приближение $\frac{4}{3}$, построен алгоритм для поиска самого оптимального распределения задач по машинам. В нем происходит полный перебор распределения задач по машинам. Реализация следующая:

```
def find_optimal_schedule(number_of_machines: int,
                          times_of_tasks: list):
```

```
machines = [_ for _ in range(number_of_machines)]
machines_for_tasks = [machines] * len(times_of_tasks)
tasks_distribution_all_cases =
                      list(itertools.product(*machines_for_tasks))
optimal_scheduling_time = sys.maxsize
optimal_distribution = []
for case in tasks_distribution_all_cases:
    machines_time = [0] * number_of_machines
    distribution = [""] * number_of_machines
    for task, machine in enumerate(case):
        machines_time[machine] += times_of_tasks[task]
        distribution[machine] += '*' * times_of_tasks[task] + '|'
    if max(machines_time) < optimal_scheduling_time:
        optimal_scheduling_time = max(machines_time)
        optimal_distribution = distribution
return optimal_distribution, optimal_scheduling_time
```

4.3 Описание наборов тестов

Для проверки того, что алгоритмы работают, была написана функция, которая показывает распределение задач на машинах. Для набора задач с временами исполнения [7, 9, 4, 13, 10, 11, 15, 21, 3, 5, 6, 2, 5] на трех машинах выглядит это следующим образом:

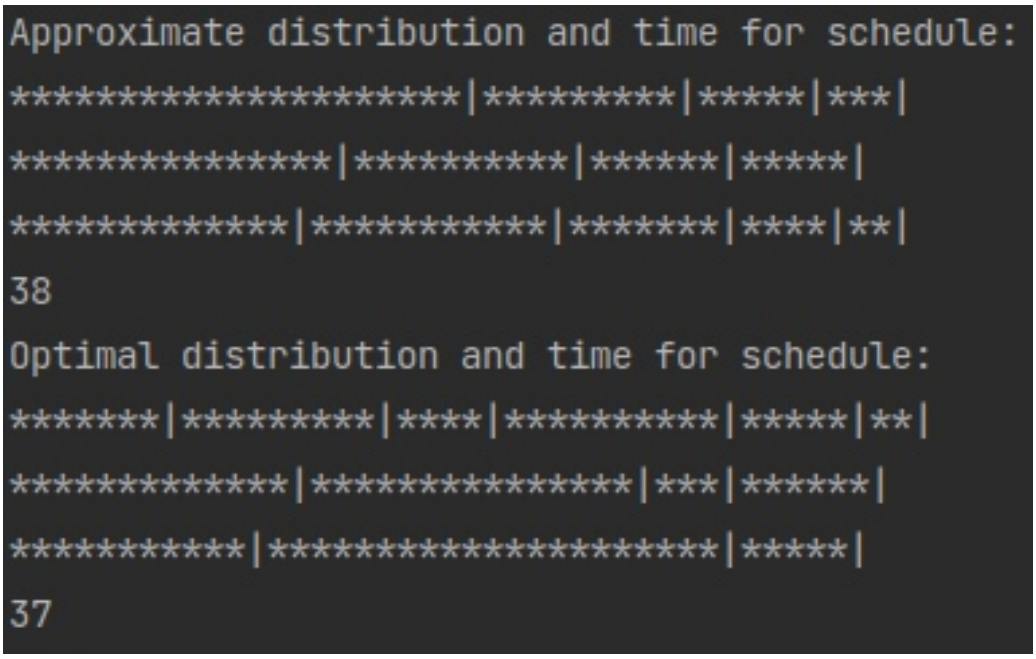

Рис. 1: Распределение задач по машинам

Далее были написаны следующие тесты:

1. Тесты для определенного количества задач. Выбирается это число N, затем генерируются времена задач от 1 до N для каждой из них. Получается N^N вариантов комбинаций времени задач. Затем выбирается число машин от 2 до $N-1$. После этого каждый набор задач дается приближающему алгоритму и алгоритму полного перебора. Если ответ на первом алгоритме - $\frac{4}{3}$ приближение второго - то это правильно.

   Количество задач выбиралось от 3 до 6.

```
def test_for_N_tasks(self):
    all_tasks_num = [3, 4, 5, 6]
    for tasks_num in all_tasks_num:
        tasks_time = [_ for _ in range(1, 1 + tasks_num)]
        tasks_combinations = [tasks_time] * tasks_num
        tasks_times_all_cases = list(itertools.product
                                     (*tasks_combinations))
        for task_set in tasks_times_all_cases:
            task_set = list(task_set)
            for machines_num in range(2, tasks_num):
                optimal_time = find_optimal_schedule(machines_num,
                                                     task_set)[1]
                approximate_time = find_approximate_schedule(
                                    machines_num, task_set.copy())[1]
                print("case: ", task_set, "machines: ", machines_num,
                      "optimal:", optimal_time, "approximate: ",
                      approximate_time)
                self.assertGreaterEqual(4 / 3 * optimal_time,
                                        approximate_time)
```

2. Выбрано 20 задач с большими временами работы, отдано 2 алгоритмам.

```
tasks_num = 20
task_set = [random.randrange(1, 10000) for _ in range(tasks_num)]
machines_num = 2
optimal_time = find_optimal_schedule(machines_num, task_set)[1]
approximate_time = find_approximate_schedule(machines_num,
                                             task_set.copy())[1]
print("case: ", task_set, "machines: ", machines_num,
      "optimal:", optimal_time, "approximate: ",
      approximate_time)
self.assertGreaterEqual(4 / 3 * optimal_time, approximate_time)
```

4.4 Анализ запусков

Для первого рода тестов для количества задач от 3 до 5 сравнение алгоритмов срабатывает довольно быстро, гораздо заметнее он замедляется при 6 задачах, так как нужно сгенерировать 6^6 задач, в алгоритме полного перебора сгенерировать m^6 вариантов разбиений по машинах, где m - количество машин, обработать каждый такой вариант. Несмотря на долгое время работы, все тесты прошли, что означает на данном классе задач построенный алгоритм работает с нужным приближением.

Во втором тесте были выбраны большие времена работы задач. Для выбранного числа задач алгоритм отработал за 28 секунд. Для задач с временами [5624, 8973, 7377, 1480, 425, 9389, 4628, 4486, 16, 1638, 3455, 1689, 3413, 4151, 9261, 1336, 5742, 1501, 922, 6464] приближенный алгоритм дал ответ 40989, а точный 40985, что является очень хорошим приближением.

4.5 Выводы по алгоритму

Приближенный алгоритм работает за $\mathcal{O}(N(logN + logM))$, где N - количество задач, а M - количество машин. Первое слагаемое из-за сортировки задач по времени, второе для выбора из кучи машины, обладающей наименьшей загруженностью. Данный алгоритм является оптимальным по времени в сравении с вариантом полного перебора, где асимптотика $\mathcal{O}(M^N)$. Во всех тестах ответ построенного алгоритма суммарного времени обработки всех задач отличается не более, чем на $\frac{4}{3}$ от ответа алгоритма полного перебора.

5 Дополнительный материал

1. Wikipedia - Job-shop scheduling
2. NP-полнота задачи о сумме подмножества
3. Graham - Bounds on multiprocessing timing anomalies