

Оглавление

Введение	3
Постановка задачи	3
Описание алгоритмов	4
Доказательство корректности	5
Нежадные расписания имеют инверсии	6
Асимптотика решений	8
Результаты запуска программы	9
Заключение	11
Список литературы	12
Приложения	13

Введение

Существуют разные парадигмы проектирования алгоритмов, одна из них - жадная парадигма. Если говорить о том, что такое жадная парадигма, то это можно выразить так: это алгоритмическая парадигма, которая следует за локальным оптимальным выбором на каждом шаге, допуская, что конечное решение окажется правильным.

Существует множество видов жадных алгоритмов, к примеру, алгоритм Дейкстры ищет кратчайшее расстояние от стартовой вершины до любой другой. Количество жадных алгоритмов создает трудности при выборе конкретной реализации. Еще одной трудностью жадных алгоритмов является определение правильности выхода алгоритма для каждого возможного входа.

Из вышесказанного можно сделать выводы:

- 1) Легко придумать один или несколько жадных алгоритмов.
- 2) Легко проанализировать время выполнения алгоритма.
- 3) Трудно установить правильность алгоритма.

Одной из причин, по которой жадные алгоритмы труднодоказуемы заключается в том, что множество алгоритмов на самом деле не являются правильным на некоторых определенных входных данных, т.е. алгоритм не получит желаемый результат.

Постановка задачи

Реализовать алгоритм *GreedyDiff* и *GreedyRatio*. Вычислить взвешенную сумму времени выполнения расписания, выводимого алгоритмами на определенном массиве данных. Сравнить временные затраты для обоих задач и привести асимптотическую оценку временной сложности алгоритмов.

Описание алгоритмов

Рассмотрим базовый жадный алгоритм в задаче планирования работ. Предположим, что каждая работа R_i имеет длину L_i , которая является мерой времени, нужной на завершение работы R_i . Например, продолжительность пары, лекции, смены и т.д. Кроме времени затрачиваемой на работу также существует важность работы или вес, который будем обозначать w_j . Таким образом, чем больше веса у работы, тем больший у нее приоритет и ее срочность выполнения.

Расписание работ конкретизирует порядок выполнения работ, то есть говорит в каком порядке выполнять работы. К примеру, если существует задача с n работами, то существует $n!$ разных расписаний, что очень много. Если $n=10$, то $10! = 6$ миллионов, если $n=20$, то $20! = 2,4$ квинтиллиона. Видим, что считать факториал разных расписаний долго. Введем целевую функцию, которая будет представлять срок завершения работы j в расписании σ есть сумма длин работ, предшествующих j в σ , плюс длина самого j . Для чего его ввели целевую функцию будет объяснено чуть позже. Таким образом, срок завершения работы в расписании - это суммарное время, которое проходит до полной обработки работы.

В реальных задачах зачастую стремятся к минимальным срокам завершения работ, но компромиссы неизбежны - в любом расписании работы, запланированные ранее, убудут иметь и более ранние сроки завершения, а те, что запланированы ближе к концу, будут иметь более поздние сроки завершения.

Один из способов решения компромиссов состоит в минимизации суммы взвешенных сроков завершения, см. формулу 1.

$$\min_{\sigma} \sum_{j=1}^n w_j C_j(\sigma)$$

Формула 1. Минимизация сроков завершения.

Минимизация выполняется над всеми $n!$ возможными расписаниями σ , а $C_j(\sigma)$ обозначает срок завершения работы j в расписании σ . Это эквивалентно минимизации средневзвешенных сроков завершения работ, где усредняющие веса пропорциональны значениям w_j . К примеру, если взять работы с весами, равными $w_1 = 3$, $w_2 = 2$, $w_3 = 1$ и выполнять последовательно работы, то сроки завершения будут равны: $3 + 6 + 6 = 15$. Проверив все возможные

расписания можно убедиться, что это действительно является расписанием, которое минимизирует сумму взвешенных сроков завершения.

Чтобы решить задачу в общем случае, не перебирая всевозможные варианты, нам понадобится жадный алгоритм. Мы знаем, что более предпочтительной работой является та, что более короткая и с большим весом, но как поступить, если идет выбор между короткой с маленьким весом и длинной с большим весом? Для данного случая можно ввести формулу, которая описывала две величины одной. В качестве реализации можно представить два варианта: *GreedyDiff* и *GreedyRatio*. Первый представляет разность между весом и длиной j -ой работы - $j: w_j - l_j$. Тогда как *GreedyRatio*, судя по названию, является отношением величин: $j: \frac{w_j}{l_j}$.

Эти две реализации приводят к двум разным алгоритмам вычисления ответа. Встает вопрос, а какой алгоритм дает лучшие результаты? Способ это сделать - найти экземпляр, в котором два алгоритма выводят разные расписания с разными значениями целевой функции. И тот алгоритм, чьи значения будут в сумме хуже показывать - не оптимальный.

Например, возьмем 2 работы, см. таблицу 1.

	Работа 1	Работа 2
Длина	$l_1 = 7$	$l_2 = 3$
Вес	$w_1 = 4$	$w_2 = 1$

Таблица 1. Пример 2 работ.

Первая работа имеет более крупное отношение $\frac{4}{7}$ против $\frac{1}{3}$, но большую разность (-3 против -2). Таким образом, алгоритм *GreedyDiff* планирует вторую работу первой, тогда как алгоритм *GreedyRatio* планирует наоборот.

Доказательство корректности

Для доказательства корректности алгоритма *GreedyRatio* зададим множество работ с положительными весами w_1, w_2, \dots, w_n и длинами l_1, l_2, \dots, l_n . Докажем, что алгоритм *GreedyRatio* создает расписание, которое минимизирует сумму взвешенных сроков завершения. Примем два допущения: первое - работы проиндексированы в невозрастающем порядке отношения веса к длине: $\frac{w_1}{l_1} \geq \frac{w_2}{l_2} \geq \dots \geq \frac{w_n}{l_n}$. Второе - между отношениями нет совпадений значений: $\frac{w_i}{l_i} \neq \frac{w_j}{l_j}$ всякий раз, когда $i \neq j$. Первое допущение создано с целью минимизировать

издержки в формальных обозначениях. Переупорядочивание работ во входе не влияет на решаемую задачу. Следовательно, есть возможность переупорядочить и переиндексировать работы так, чтобы допущение соблюдалось. Второе допущение накладывает ограничение на вход. В сумме два допущения приводят к тому, что задания индексируются в строго убывающем порядке отношения веса к длине.

Будем исходить от противного. Для начала допустим, что *GreedyRatio* создает расписание заданных работ, которое не является оптимальным. Таким образом, существует оптимальное расписание σ^* этих работ со строго меньшей суммой взвешенных сроков завершения. Используем разницу между σ и σ^* для явного конструирования расписания, которое еще лучше, чем σ^* ; это будет противоречить допущению о том, что σ^* является оптимальным расписанием. Предположим, от противного, что алгоритм *GreedyRatio* производит расписание σ и что существует оптимальное расписание σ^* со строго меньшей суммой взвешенных сроков завершения. Согласно первому допущению, жадное расписание σ планирует работы в порядке индекса (сначала работа 1, затем работа 2, вплоть до работы n). Двигаясь снизу вверх в жадном расписании, индексы работ всегда поднимаются вверх. Это не относится ни к одному другому расписанию. Уточняя это логическое утверждение, определим последовательную инверсию в расписании как пару i, j работ, таких что $i > j$ и работа i обрабатывается непосредственно перед работой j . Для дальнейшего доказательства нам нужна лемма о том, что жадные расписания имеют инверсии.

Не жадные расписания имеют инверсии

Каждое расписание, отличное от жадного расписания σ , имеет, по крайней мере, одну последовательную инверсию.

Доказательство: мы доказываем противопоставление. В отсутствие последовательных инверсий индекс каждой работы по крайней мере на 1 больше, чем у работы, ей предшествующей. Существует n работ, и максимально возможный индекс равен n , поэтому между индексами работ, расположенными подряд, не может быть скачков, равных 2 или больше. Это означает, что $\hat{\sigma}$ совпадает с расписанием, вычисленным жадным алгоритмом. Ч. Т. Д.

Возвращаясь к доказательству прошлой теоремы, мы допускаем, что существует оптимальное расписание σ^* заданных работ со строго меньшей суммой взвешенных сроков завершения, чем жадное расписание σ . Поскольку $\sigma \neq \sigma^*$, лемма выше применима к σ^* , и есть идущие подряд работы i, j в σ^* при $i > j$. Как использовать этот факт для выявления еще одного расписания σ' , которое еще лучше, чем σ^* , тем самым создавая противоречие? Идея заключается в выполнении обмена. Давайте определим новое расписание σ' , идентичное σ^* , за исключением того, что работы i и j обрабатываются в обратном порядке, где теперь j обрабатывается непосредственно перед i . Работы перед i и j совпадают с σ^* и σ' . То же касается работ, которые следуют и за i , и за j . В результате обмена работами i и j в последовательной инверсии

срок завершения C_j увеличивается на длину ℓ_j работы j , что увеличивает целевую функцию на $w_i \times \ell_j$. Преимущество в том, что срок завершения C_j уменьшается на длину ℓ_j , задания i , что уменьшает целевую функцию на $w_j \times \ell_i$. В итоге мы приходим к следующему -

$$\sum_{k=1}^n w_k C_k(\sigma') = \sum_{k=1}^n w_k C_k(\sigma^*) + w_i \ell_j - w_j \ell_i \quad (1)$$

, где $\sum_{k=1}^n w_k C_k(\sigma')$ - значение целевой функции σ' , $\sum_{k=1}^n w_k C_k(\sigma^*)$ - значение целевой функции σ^* , $w_i \ell_j - w_j \ell_i$ - эффект обмена.

Теперь используем тот факт, что σ^* запланировало i и j в неправильном порядке при $i > j$. Первое и второе допущение приводят к тому, что работы индексируются в строго убывающем порядке отношения веса к длине, поэтому $\frac{w_i}{\ell_i} < \frac{w_j}{\ell_j}$. После очищения знаменателя получаем $w_i \ell_j < w_j \ell_i$. Поскольку преимущество обмена превышает допустимую стоимость, уравнение (1) говорит о том, что значение целевой функции от σ' меньше значения целевой функции от σ^* .

Однако расписание σ^* должно быть оптимальным при наименьшей возможной сумме взвешенных сроков завершения.

В итоге мы получили противоречие, которое завершает доказательство формулы суммы взвешенных сроков завершения для случая, когда все работы имеют разные отношения веса к длине.

Теперь докажем правильность алгоритма *GreedyRatio* с учетом наличия совпадения значений (ничья) в коэффициентах веса к длине работы и исходя из того, что работы индексируются в неубывающем порядке отношения веса к длине, так как это не влечет потерю общности. Обозначим через $\sigma = 1, 2, 3, \dots, n$ расписание, вычисленное алгоритмом *GreedyRatio*. Рассмотрим произвольное расписание σ^* , показав непосредственно, последовательностью обменов работами, что сумма взвешенных сроков завершения расписания σ не больше, чем σ^* . Доказав это для каждого расписания σ^* , мы заключим, что σ , по сути дела, является оптимальным расписанием.

Теперь допустим, что $\sigma^* \neq \sigma$. По лемме выше σ^* имеет последовательную инверсию - две работы i и j , причем $i > j$, а j запланирована сразу после i . Возьмём σ' из σ^* , поменяв местами i и j в расписании. Как и для уравнения (1), стоимость и преимущества этого обмена составляют соответственно $w_i \ell_j$ и $w_j \ell_i$. Поскольку $i > j$ и работы индексируются в неубывающем порядке отношения веса к длине, $\frac{w_i}{\ell_i} \leq \frac{w_j}{\ell_j}$ и, следовательно, $w_i \ell_j \leq w_j \ell_i$. Другими словами, обмен не может увеличить сумму взвешенных сроков завершения - сумма может уменьшиться или остаться прежней.

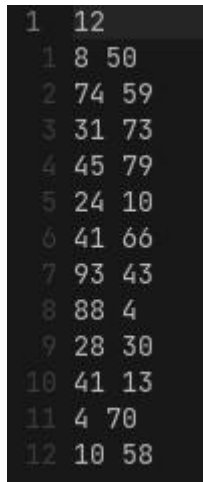
Асимптотика решений

Алгоритмы работают за $O(n \log(n))$. На сортировку элементов по возрастанию весов требуется $O(n \log(n))$. После сортировки происходит вычисление взвешенной суммы за $O(n)$ шагов. В итоге $O(n + n \log(n)) = O(n \log(n))$.

Результаты запуска программы

Для того, чтобы выяснить, какой алгоритм показывает лучшие результаты, *GreedyDiff* или *GreedyRatio*, нужно их сравнить.

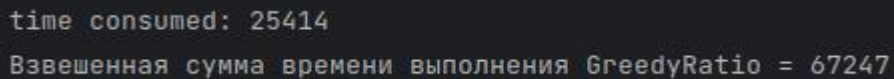
Проверим алгоритмы на коротком наборе данных, см. фото 1.



1	12
1	8 50
2	74 59
3	31 73
4	45 79
5	24 10
6	41 66
7	93 43
8	88 4
9	28 30
10	41 13
11	4 70
12	10 58

Фото 1. Малый набор данных для проверки работоспособности алгоритмов.

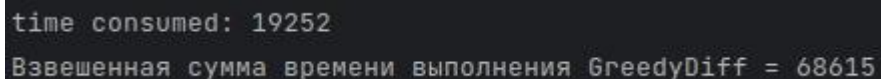
GreedyRatio выполняет свою работу за 25414 нс, взвешенная сумма сроков завершения работ равна 67247, см. фото 2.



```
time consumed: 25414
Взвешенная сумма времени выполнения GreedyRatio = 67247
```

Фото 2. Результат работы *GreedyRatio*.

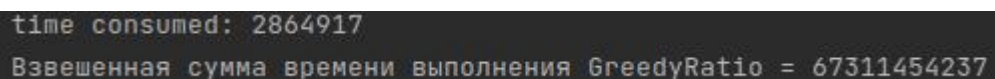
Проверим *GreedyDiff*. Время работы - 19252 нс, взвешенная сумма - 68615, см. фото 3.



```
time consumed: 19252
Взвешенная сумма времени выполнения GreedyDiff = 68615
```

Фото 3. Результат работы *GreedyDiff*.

Теперь проверим данные из задачи на 10 000 строк. *GreedyRatio* справился за 2864917 нс с ответом равным 67311454237, см. фото 4. *GreedyDiff*, в свою очередь, с временем в 2642450 нс и взвешенной суммой равной 69118616542, см. фото 5.



```
time consumed: 2864917
Взвешенная сумма времени выполнения GreedyRatio = 67311454237
```

Фото 4. *GreedyRatio*.


```
time consumed: 2642450  
Взвешенная сумма времени выполнения GreedyDiff = 69118616542
```

Фото 5. *GreedyDiff*.

По результатам видим, что алгоритм *GreedyRatio* показывает лучшие результаты по взвешенной сумме, чем *GreedyDiff*, но уступает по скорости. Весь исходный код лежит в **Приложении**.

Заключение

Оба жадных алгоритма решают поставленную задачу. Алгоритм GreedyRatio показывает лучшие выходные данные из-за выбранной им метрики $j: \frac{w_j}{l_j}$, но не превосходит алгоритм GreedyDiff по скорости вычисления, медленнее на 8% из-за деления, которое лежит в основе GreedyRatio, оно медленнее, чем сложение (от 1 до 7 тактов процессора на деление, против 1 у сложения).

Список литературы

1. Рафгарден Тим Совершенный алгоритм: Жадные алгоритмы и динамическое программирование / Рафгарден Тим — . — СПб: Питер, 2020 — 256 с.
2. Кнут Д.Э. Искусство программирования. Том 1. Основные алгоритмы. / Д.Э. Кнут.— Москва: Диалектика-Вильямс, 2019. – 720 с.
3. JDK 19 Documentation [Электронный ресурс]. URL: <https://docs.oracle.com/en/java/javase/19>.
4. И.Г. Семакин, А.П. Шестаков. Основы алгоритмизации и программирования. – М.: Академия, 2008. – 400 с.
5. Роберт Лафоре Структуры данных и алгоритмы в Java. – СПб.: Питер, 2016. – 704 с.

Приложения

Реализация алгоритма GreedyRatio.

```
public class GreedyRatio {

    protected ArrayList<RatioJob> jobs;

    public GreedyRatio(ArrayList<RatioJob> jobs) {
        this.jobs = jobs;
    }

    private void createSequence() {
        Collections.sort(jobs);
    }

    public long calculateEfficiency() {
        createSequence();
        long result = 0;
        long currentLength = 0;
        long time_start = System.nanoTime();

        for (RatioJob job: jobs) {
            currentLength += job.getLength();
            result += job.getWeight() * currentLength;
        }
        long time_end = System.nanoTime() - time_start;
        System.out.println("time consumed: " + time_end);
        return result;
    }

    public void FromFile(String path) throws IOException {
        InputStream inputStream = new FileInputStream(path);
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(inputStream));
        int a = Integer.parseInt(reader.readLine());
        while (a != 0) {
            a--;
            String[] splittedLine = reader.readLine().split("\\s+");
            this.addJob(new
GreedyRatio.RatioJob(Integer.parseInt(splittedLine[0]),Integer.parseInt(splittedLine[1])));
        }
        inputStream.close();
        reader.close();
    }

    public void addJob(RatioJob job) {
        jobs.add(job);
    }

    public static class RatioJob implements Comparable<RatioJob> {

        int length;
        int weight;

        public RatioJob(int length, int weight) {
            this.length = length;
        }
    }
}
```

```

        this.weight = weight;
    }

    public int getLength() {
        return length;
    }

    public void setLength(int length) {
        this.length = length;
    }

    public int getWeight() {
        return weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }

    @Override
    public int compareTo(RatioJob o) {
        if( (this.weight * 1.0 / this.length) - (o.weight * 1.0 / o.length) > 0) {
            return -1;
        } else if ((this.weight * 1.0 / this.length) - (o.weight * 1.0 / o.length) < 0) {
            return 1;
        } else {
            return 0;
        }
    }
}

```

Реализация алгоритма *GreedyDiff*

```

public class GreedyDiff {

    protected ArrayList<DiffJob> jobs;

    public GreedyDiff(ArrayList<DiffJob> jobs) {
        this.jobs = jobs;
    }

    private void createSequence() {
        Collections.sort(jobs);
    }

    public void FromFile(String path) throws IOException {
        InputStream inputStream = new FileInputStream(path);
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(inputStream));
        int a = Integer.parseInt(reader.readLine());
        while (a != 0) {
            a--;
            String[] splittedLine = reader.readLine().split("\\s+");
            this.addJob(new DiffJob(Integer.parseInt(splittedLine[0]), Integer.parseInt(splittedLine[1])));
        }
        inputStream.close();
        reader.close();
    }

    public long calculateEfficiency() {
        createSequence();
    }
}

```

```

    long result = 0;
    long currentLength = 0;
    long time_start = System.nanoTime();

    for (DiffJob job: jobs) {
        currentLength += job.getLength();
        result += job.getWeight() * currentLength;
    }
    long time_end = System.nanoTime() - time_start;
    System.out.println("time consumed: " + time_end);
    return result;
}

    long estimatedTime = System.nanoTime() - time_start;
    System.out.println("Время на подсчет для GreedyDiff:" + estimatedTime + "ns");
    return result;
}

public void addJob(DiffJob job) {
    jobs.add(job);
}

public static class DiffJob implements Comparable<DiffJob> {

    int length;
    int weight;

    public DiffJob(int length, int weight) {
        this.length = length;
        this.weight = weight;
    }

    public int getLength() {
        return length;
    }

    public void setLength(int length) {
        this.length = length;
    }

    public int getWeight() {
        return weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }

    @Override
    public int compareTo(DiffJob o) {
        return (o.weight - o.length) - (this.weight - this.length);
    }
}
}

```

Реализация вспомогательной структуры данных *Job*

```

public class Job implements Comparable{

    private int length;
    private int importance;
}

```

```

public Job(int length, int importance) {
    this.length = length;
    this.importance = importance;
}

public int getLength() {
    return length;
}

public void setLength(int length) {
    this.length = length;
}

public int getImportance() {
    return importance;
}

public void setImportance(int importance) {
    this.importance = importance;
}

public String showDifference() {
    return "Diff: " + (importance - length);
}

public String showRatio() {
    return "Ratio: " + 1.0 * importance/length;
}

@Override
public String toString() {
    return "Importance: " + importance + " Length: " + length;
}

@Override
public int compareTo(Object o) {
    return 0;
}
}

```

Реализация интерфейса запуска и тестирования алгоритмов

```

public static void main(String[] args) throws IOException {

    System.out.println();

    GreedyDiff Diff = new GreedyDiff(new ArrayList<>());
    GreedyRatio Ratio = new GreedyRatio(new ArrayList<>());

    Ratio.FromFile("../test_data_big.txt");
    System.out.println("Взвешенная сумма времени выполнения GreedyRatio = " +
    Ratio.calculateEfficiency());

    System.out.println();
    Diff.FromFile("../test_data_big.txt");
    System.out.println("Взвешенная сумма времени выполнения GreedyDiff = " +
    Diff.calculateEfficiency());
}

```

}