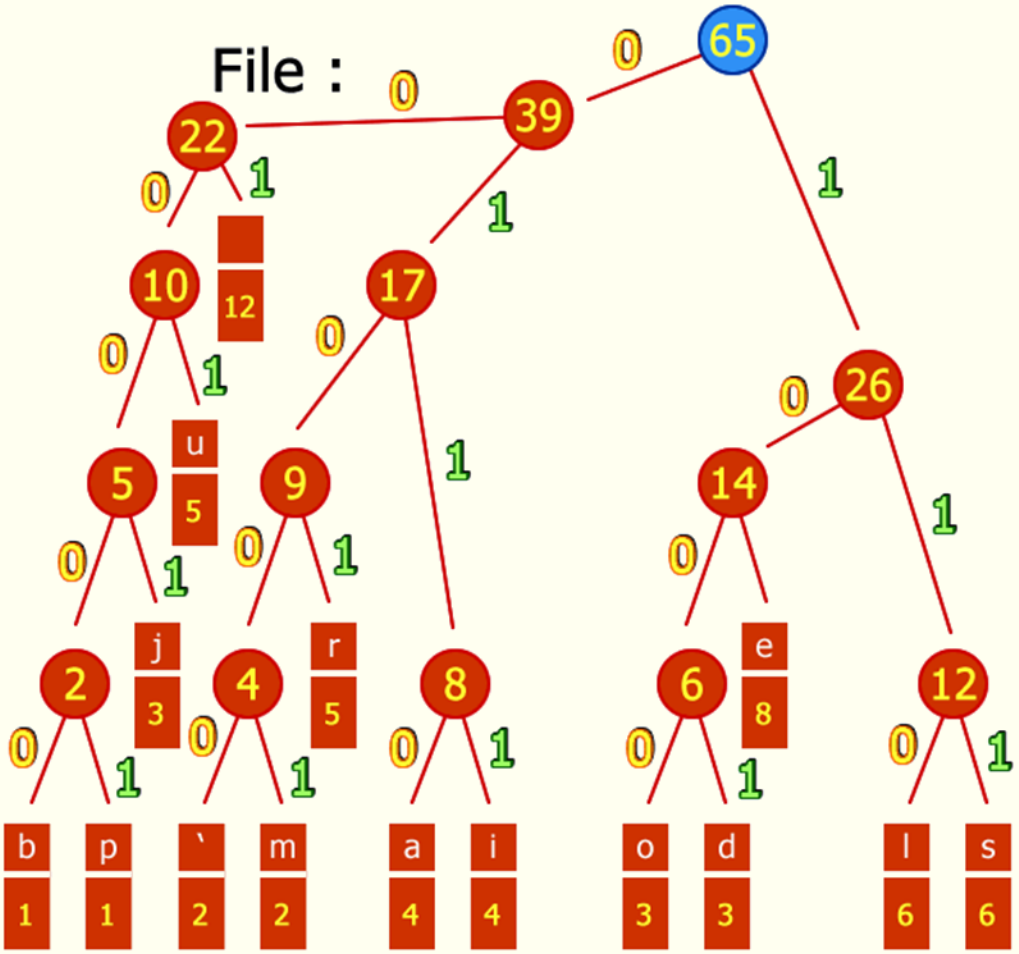
2013

СУ „Св. Климент Охридски“, ФМИ

Георги Кръстев
[ф.н. 61368]



[РАЗПРЕДЕЛЕН АЛГОРИТЪМ НА ХЪФМАН ЗА КОМПРЕСИЯ]

Курсов Проект по Разпределени Софтуерни Архитектури [РСА]

Условие на Задачата

Проблем

Алгоритъмът на Хъфман е сравнително прост универсален алгоритъм за компресия без загуба на данни. При него се предполага, че е даден краен поток от числа в някакъв предварително фиксиран интервал. Алгоритъмът се базира на простата идея, че най-често срещаните символи в поредицата трябва да се записват с най-малък брой битове. Така той построява нова азбука, която следва тази идея, и след това превежда информацията в новата азбука. Кодирането е обратимо, т.е. по кодираната последователност може да се декомпресира - да се намери първоначалната поредица.

Задача

Разглеждаме алгоритъма на Хъфман за компресия в частта му свързана с построяване на честотна таблица на входния поток от информация. Задачата е да се напише програма, която строи честотна таблица на даден двоичен или текстов (достатъчно голям) файл. Програмата да разпределя по подходящ начин работата за построяване на честотната таблица между две или повече нишки (задачи).

Не правим разлика между текстови и двоични файлове. Ще считаме, че става дума за символи, кодирани с ASCII код, т.е. ще разглеждаме информацията като поредица от байтове (числа в интервала 0..255).

Мотивация

В задачата разглеждаме само построяването на честотна таблица на файла, защото това се оказва и единствената част от алгоритъма, която може ефективно да бъде разпределена между повече нишки / процесори. Създаването на дърво на Хъфман и обхождането му за генериране кода на символите са по същество последователни операции, защото всяка следваща стъпка зависи от предходните по данни (data flow). Тази зависимост е непреодолима, но това не е голям проблем, тъй като тези изчислителни задачи не са твърде трудоемки за процесора. За съжаление работата по самата компресия на файла трудно може да бъде паралелизирана, защото е необходимо побитово писане, а няма как да се предвиди компресираната големина на даден сектор от файла.

Алгоритъм и Разпределение

Алгоритъм

Програмата реализира каноничен вариант на алгоритъма на Хъфман. Тук няма да опиша целия алгоритъм, защото е част от задачата, а ще посоча разликите между каноничната и класическата версия. Само ще приложа псевдокод на алгоритъма [Wikipedia]:

```
compute huffman code:
 input:  message ensemble (set of (message, probability)), base D
 output: code ensemble (set of (message, code))
 algorithm:
   1. sort the message ensemble by decreasing probability
   2. N is the cardinal of the message ensemble (number of different messages)
   3. compute the integer n0 such as 2 <= n0 <= D and (N-n0)/(D-1) is integer
   4. select the n0 least probable messages and assign them each a digit code
   5. substitute the selected messages by a composite message summing their
      probability and re-order it
   6. while there remains more than one message, do steps thru 8:
   7.   select D least probable messages and assign them each a digit code
   8.   substitute the selected messages by a composite message summing their
        probability and re-order it
   9. the code of each message is given by the concatenation of the code
      digits of the aggregate they've been put in
```

Псевдокод за канонично генериране кода на символите [Wikipedia]:

```
code = 0
while more symbols:
  print symbol, code
  code = (code + 1) << ((next bit length) - (current bit length))
```

При каноничната версия дървото на Хъфман не се използва директно за генериране кода на символите, а от него се изчислява само дължината на този код в битове. След това символите се сортират по дължина на кода, а при равенство по естествена подредба. Така генерираният списък се кодира лексикографски последователно, т.е. кодът започва от 0 и на всеки следващ символ се инкрементира с 1, а при увеличаване на броя битове, се извършва необходимия “shift left” (<<). По този начин получаваме еднозначно кодиране по алгоритъма на Хъфман. Това позволява да запишем в компресирания файл само таблица с дължината на кода за всеки символ, т.е. не е необходима сериализация на цялото дърво на Хъфман. Освен това този метод е най-лесен за програмна реализация.

Разпределение

Частта от алгоритъма, която трябва да бъде обработена паралелно, е генерирането на честотна таблица на дадения файл. По същество тази задача не е нищо повече от броене на байтове и актуализирането на бройката в подходяща структура от данни (СД). Съвсем естествено е да разпределим работата по данни, като за всяка нишка се определя сектор от файла за обхождане.

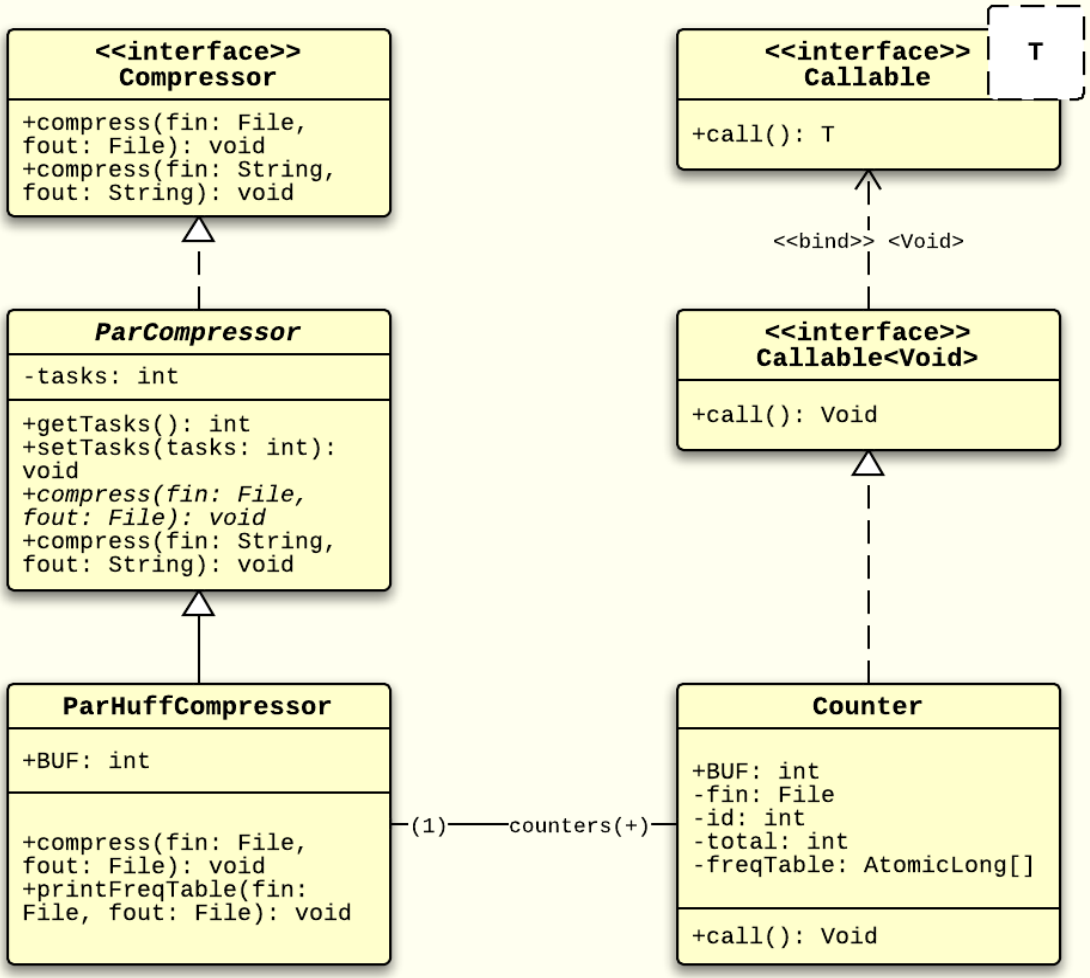
Програмата решава задачата многонишково на Java, следователно парадигмата за паралелна обработка е master – slave, която в случая е подходяща. Моделът на оперативната памет (ОП) в Java е споделена памет (shared memory – SM), което означава, че е необходима СД с безопасен многонишков достъп (thread safe) за съхранение на честотната таблица.

Наивното решение на проблема е СД със синхронизирани методи. Следните итерации при реализация на паралелния алгоритъм (ПА) водят от наивното решение до оптимизиран вариант чрез минимизиране на комуникацията и съответно синхронизацията между отделните нишки:

1. Наивно решение – най-подходяща структура от данни е `HashMap<Byte, Long>`, защото лесно може да бъде синхронизирана чрез метода `Collections::synchronizedMap`.
2. Понеже синхронизираните операции са много скъпи откъм процесорно време, по-ефективно е всяка нишка да работи върху локална СД и да актуализира глобалната честотна таблица, когато приключи работата си.
3. На този етап съобразяваме, че в задачата броят на нишките е експлицитно зададен – информация, която можем да използваме за ускорение на програмата, като заменим `HashMap` с `ConcurrentHashMap`. Тази структура от данни е до 2 пъти по-бърза [Java EE Support Patterns], защото е реализирана на базата на неблокиращи (non-blocking) алгоритми [Wikipedia], а при известен брой нишки се представя много добре.
4. За да оптимизираме допълнително програмата, е нужен по-фин контрол върху синхронизацията между нишките. Тук можем да използваме това, че боравим с фиксиран диапазон от символи, т.е. можем да фиксираме големината на честотната таблица. Това позволява да използваме масив от атомарни променливи (`AtomicLong[]`), което допълнително минимизира синхронизацията, защото получаваме независим достъп до различни индекси от таблицата, при това на цената на достъп до масив (без пресмятане на hash).
5. За да се възползваме максимално от този независим достъп, задаваме за всяка нишка различна стартова позиция при обхождане на таблицата за актуализация на резултатите. По този начин намаляваме вероятността за създаване на конкурентни условия (racing conditions).
6. Буферирано четене от файла, защото входно-изходните (I/O) операции са най-бавни – 64 KB буфер е оптимален за Java [Nadeau].

Архитектура на Решението

Архитектурата на решението е сравнително елементарна. Клас диаграма на частта от програмата, свързана с генериране на честотната таблица:



Класът `Counter` имплементира интерфейса `Callable<Void>`, за да може колекция от такива класове лесно да бъде изпълнена чрез метода `ExecutorService::invokeAll`. Класът `ParHuffCompressor` реализира разпределения алгоритъм на Хъфман за компресия, като включва и метод за тестване на честотната таблица – `printFreqTable`.

След прилагане на описаните оптимизации, кодът на метода `Counter::call` излгежда по следния начин:

[Counter.java]

```
…
public Void call() throws IOException {
    InputStream in = new FileInputStream(fin);
    try {
        long len = fin.length() / total, off = id * len;
        len += id < total - 1 ? 0 : fin.length() % total;
        long[] lcl = new long[ByteSym.RANGE];
        byte[] buf = new byte[BUF];
        in.skip(off);
        while (len > 0) {
            int l = in.read(buf, 0, (int) Math.min(len, BUF));
            len -= l;
            for (int i = 0; i < l; i++) {
                lcl[ByteSym.uByte(buf[i])]++;
            }
        }

        int sym = id * ByteSym.RANGE / total;
        for (int i = 0; i < ByteSym.RANGE; i++) {
            sym = ++sym % ByteSym.RANGE;
            if (lcl[sym] > 0) {
                freqTable[sym].addAndGet(lcl[sym]);
            }
        }

        return null;
    } finally {
        in.close();
    }
}
…
```

Тестови Резултати и Анализ

Програмата е тествана на 24-процесорна виртуална машина с 8 GB RAM, под Debian GNU Linux среда. Тестовият файл е видео (.mkv) с големина 3.5 GB, като преди първия тест, програмата се изпълнява 1 път, за да бъде кеширан файлът от операционната система (ОС) в ОП. Целта на това е да се избегнат грешки в замерванията поради неопределената скорост на входно-изходните операции, защото са най-скъпи откъм процесорно време и могат дори да надхвърлят ефекта от различен брой нишки.

Резултати

На следващата страница е поместена графика с осреднените резултати от 10 теста по отношение на времето за изпълнение (time), ускорението (speed up) и ефикасността (efficiency) на паралелната обработка при различен брой нишки в диапазона 1..24. Както се вижда на графиката, последователното изпълнение на алгоритъма отнема малко над 4 сек., а най-бързото време на програмата е малко над половин сек. Максималното достигнато ускорение е малко над 8 пъти.

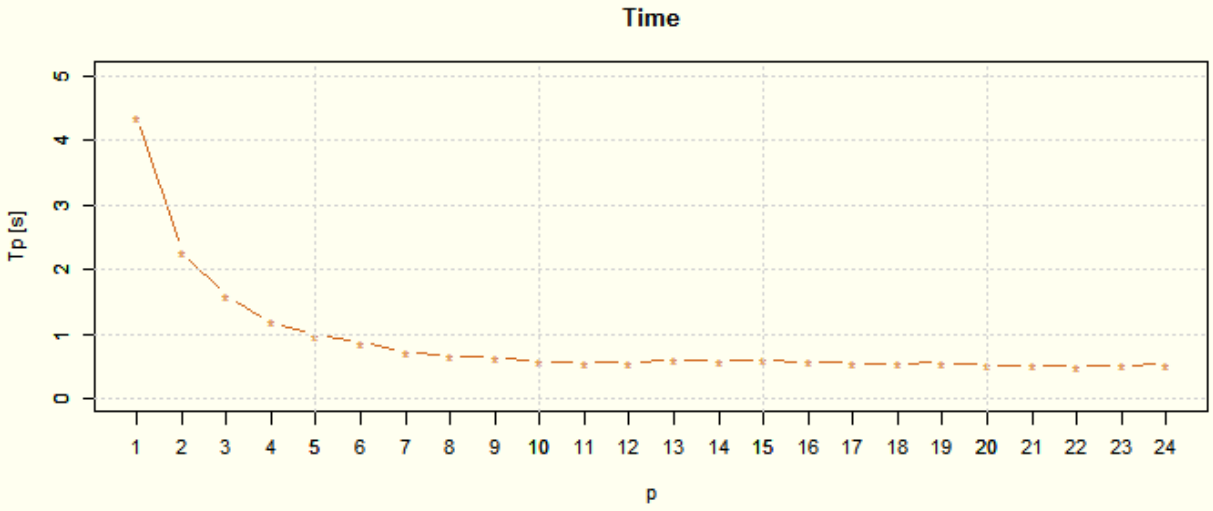
Анализ

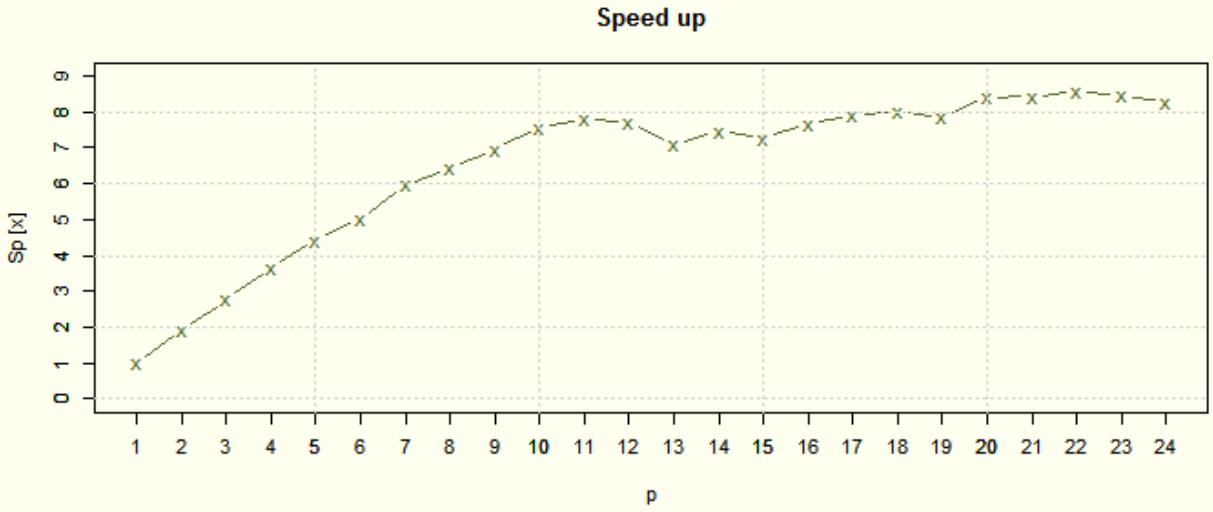
Резултати от тестовете, които виждаме на следващата страница, не са неочаквани. В началото на графиките (брой нишки 1..8) ускорението нараства приблизително линейно и съответно времето за изпълнение спада приблизително експоненциално. Сравнително малък брой нишки не създават твърде голям комуникационен (съотв. синхронизационен) и директивен товар (overhead). Също така наблюдаваме монотонен, но плавен спад в ефикасността на нишките до 80%.

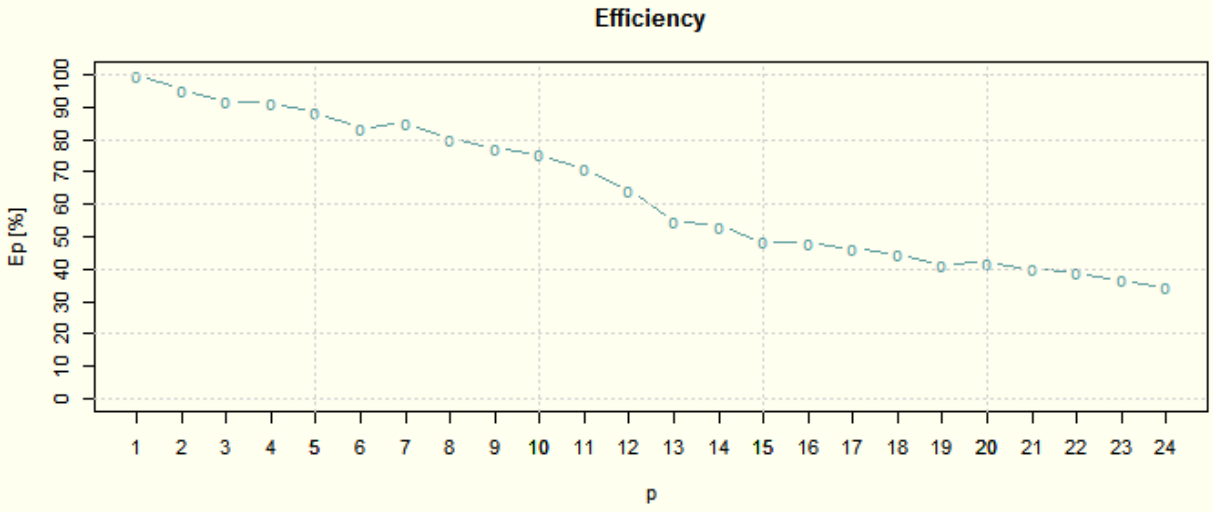
След брой нишки 10 кривата на ускорението достига точка на насищане и оттам нататък времето за изпълнение остава приблизително константно. Съответно в графиката на ефикасност наблюдаваме рязък спад, защото увеличаване на броя нишки не ускорява повече програмата.

Между брой нишки 12 и 20 се наблюдават леки аномалии (колебания) във времето за изпълнение и съответно в графиката на ускорение, но това е нещо нормално. В същото време ефикасността на нишките спада монотонно (с малко изключение при 7 нишки), както би се очаквало.

В крайна сметка при по-голям брой нишки (20..24) поради високия комуникационен (съотв. синхронизационен) и директивен товар, не наблюдаваме реално ускорение на програмата, но няма и забавяне. Ако продължим да увеличаваме броя на нишките над броя процесори (24), се очаква дори забавяне на изпълнението.


Time
Tp [s]
0 1 2 3 4 5
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
p


Speed up
Sp [x]
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
p


Efficiency
Ep [%]
0 10 20 30 40 50 60 70 80 90 100
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
p

Източници

- [Wikipedia]:
  - http://en.wikipedia.org/wiki/Huffman_coding – алгоритъм на Хъфман;
  - http://en.wikipedia.org/wiki/Canonical_Huffman_code – каноничен алгоритъм на Хъфман;
  - http://en.wikipedia.org/wiki/Non-blocking_algorithm – неблокиращи (non-blocking) алгоритми;
- [Java EE Support Patterns]:
  - http://javaeesupportpatterns.blogspot.de/2012/08/java-7-hashmap-vs-concurrenthashmap.html – анализ на производителност – `HashMap` vs. `ConcurrentHashMap`;
- [Nadeau]:
  - http://nadeausoftware.com/articles/2008/02/java_tip_how_read_files_quickly – анализ на производителност – различни методи за четене от файл в Java;
- http://www.cs.duke.edu/courses/fall09/cps100/assign/burrows/code/
  - [BitInputStream.java];
  - [BitOutputStream.java];

Source Code

- Изпълним файл (executable) – [dist/ParHuffCompr.jar];
- Сорс код на Java програмата (source code) – [ParHuffCompr/];
- Тестов скрипт (test script) – [test.py];
- Тестов файл (.mkv) – [/mnt/av/dvd/Epica-Retrospect_Live_2013.mkv];
- Честотна таблица на тестовия файл (frequency table) – [freqTable.txt];
- Тестова статистика – [stat.txt];
- Скрипт за конвертиране на тестовата статистика до CSV – [stat2csv.py];
- CSV със статистика за времената на изпълнение – [csv.txt];
- Скрипт на R за графично представяне на резултатите – [ParHuffCompr.R];
- Документация – [doc.pdf];