Information Retrieval

WS 2016 / 2017

Lecture 3, Tuesday November 8th, 2016
(Efficient List Intersection)

Prof. Dr. Hannah Bast
Chair of Algorithms and Data Structures
Department of Computer Science
University of Freiburg

Overview of this lecture

- Organizational
  - Your experiences with ES 2 — Ranking and Evaluation
  - About the tutorials — Online vs. Personal
- Contents
  - List Intersection — Recap, Time Measurement
  - Non-algorithmic improvements — Arrays, Branching, Sentinels
  - Algorithmic improvements — Galloping Search, Skip Pointers

Exercise Sheet 3: implement list intersection and make it as fast as possible on a small benchmark we have prepared

Experiences with ES2 1/3

- Summary / excerpts
  - Time-extensive exercise sheet, but again mostly due to lack of programming experience or practice
  - Some mistakes in the TIP file, but quickly fixed when pointed out on the forum

    Please always watch the forum for updates !
  - Some problems with the floats (1.885) in the test cases
  - Some bugs not found by test cases
  - Lecture recording helped a lot
  - One does not program well with: a cold, lack of sleep, lack of concentration, starting late, …

Experiences with ES2 2/3

- Results
  - Small differences in the implementation can make a significant difference in the results
  - Variation of the BM25 parameters make some queries better, but don't affect others or make them worse
  - Boosting (popular documents, full match) helped a bit
  - Removing frequent words ("stop words") helped a bit
  - Best results: P@3 ≈ 60%, P@R ≈ 40%, MAP ≈ 40%
  - Bottom line: tuning a ranking algorithm is super important (for result quality) but also super hard

    In particular, it is very hard to understand / predict the effect of changes in the parameters / implementation

Experiences with ES2 3/3

- Test Cases
  - A working program written by yourself is a confirmation that you have really understood the stuff from the lecture
  - However: this only holds true if the program does what it should do and not just "something"

    Experience from (many) previous courses: a significant fraction of students submit code that technically "works" but is otherwise wrong or even completely wrong

    E.g. for ES2, a completely wrong ranking
  - This is why we make the effort to provide test cases for you and require from you that you implement them

    This has actually been a frequent request in previous years

Tutorials

- Online vs. personal
  - So far, the tutorials are completely online:

    You submit online, you get feedback online, and questions and answers are online
  - This is great because it gives you a lot of flexibility (you can work and ask any time you prefer)
  - However, some people prefer a personal meeting, at least from time to time … hence we offer:

    You can meet with your tutor anytime, just ask him/her

    There will be a personal tutorial every four weeks or so

    QUESTION: when would be a good time for you?

List Intersection 1/4

- Recap and motivation for today
  - In Lecture 1, we have intersected the inverted lists
  - In Lecture 2, we have merged the inverted lists
  - For efficiency reasons, many search engines only return results which contain all the query words

    Apache's Lucene, the most widely used open-source search engine, supports intersect (AND) and merge (OR)

    In most applications, intersect is used by default
  - Today we will focus on **efficiency** and therefore on list intersection

List Intersection 2/4

- Time measurement
  - Trickier than it may seem at first, because there can be significant variation between runs, for example due to:

    Other jobs running on your machine

    The Java garbage collector running unpredictably

    Data is partly in disk cache / L1-cache / TLB cache
  - Therefore, always repeat your time measurements, and take the average over all these

    For ES3, **repeat 5 times** for each measurement

    Note: repetition itself can also distort the truth because of caching effects ... but not an issue for us today

List Intersection 3/4

- Time measurement in **Java**
  - For **milli**second resolution

```
long time1 = System.currentTimeMillis();
// whatever code you want to time
long time2 = System.currentTimeMillis();
long millis = time2 – time1;
```

  - For **micro**second resolution

```
long time1 = System.nanoTime();
// whatever code you want to time
long time2 = System.nanoTime();
long micros = (time2 – time1) / 1000;
```

List Intersection 4/4

- Time measurement in **C++**
  - For **milli**second resolution (C-Style) #include <time.h>

```
clock_t time1 = clock();
// whatever code you want to time
clock_t time2 = clock();
size_t millis = 1000 * (time2 – time1) / CLOCKS_PER_SEC;
```

  - For **micro**second resolution (C++11) #include <chrono>

```
auto time1 = std::chrono::high_resolution_clock::now();
// whatever code you want to time
auto time2 = std::chrono::high_resolution_clock::now();
size_t micros = std::chrono::duration_cast
                             <microseconds>(…).count();
```

Non-algorithmic improvements 1/4

- Motivation
  - Implementation details can have a great impact on performance (even with the same underlying algorithm)
  - Let us implement the basic "**zipper**" algorithm for list intersection from Lecture 1 and look at a few variations
  - We make a part of the code (reading from file and the basic algorithm) available to you in both **Java** and **C++**

    This should make ES3 easier / less work for you
  - During the lecture, I will implement in Java today

    Note that using **Python** makes little sense when studying efficiency issues: the overhead of its internal data types (i.p. Python's lists/arrays) weighs too heavy

Non-algorithmic improvements 2/4

- Native arrays
  - **Java:** ArrayList much worse than native [] array

    Elements of an ArrayList cannot be basic data types (e.g. int), but have to be objects (e.g. Integer)

    This causes inefficient byte code / machine code
  - **C++:** std::vector is as good as [] with option –O3

    Elements of an std::vector can be basic data types as well as objects

    Due to C++'s templating mechanism, machine code for std::vector<int> is almost the same as for int[]

Non-algorithmic improvements 3/4

- Predictable branches
  - Branches = all conditional parts in your code

    In particular, **if … then … else** parts
  - Modern processors do pipelining = speculative execution of future instructions before the current ones are done
  - For conditional parts they have to guess the outcome
  - So good to **minimize** amount of conditional parts and/or improve the predictability of conditionals

    A conditional has good predictability if it evaluates to the same Boolean value most of the time

Non-algorithmic improvements 4/4

- Sentinels
  - Special elements to avoid testing for index out of bound

    Less code + further reduction in number of branches
  - For list intersection: id ∞ at the end of both lists

    For Java, take: Integer.MAX_VALUE

    For C++, take: std::numeric_limits<int>::max()

Algorithmic improvements 1/8

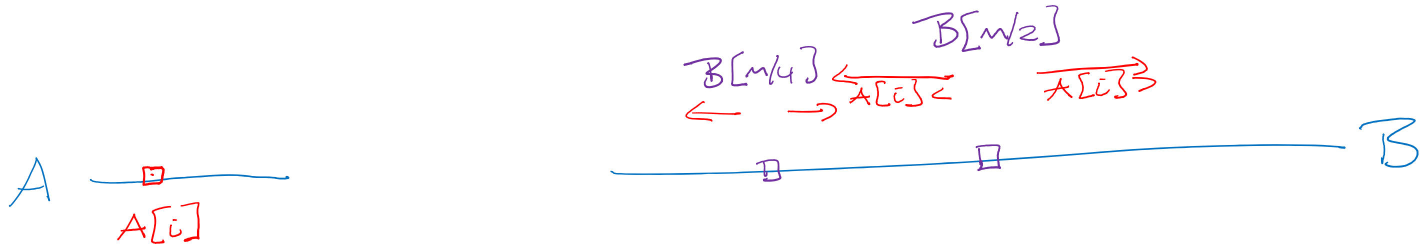
- Preliminaries
  - We have two lists, which we want to intersect
  - Let **A** be the smaller list, with **k** elements
  - Let **B** be the longer list, with **n** elements

    List intersection is commutative, so we can always assume that the first list is A, and the second is B
  - Recall that both lists are **sorted** … this is crucial for the basic algorithm and all the algorithms in the following

Algorithmic improvements 2/8

- Binary search in the longer list
  - Search each element from A in B, using binary search
  - This has time complexity $\Theta(k \cdot \log n)$

    Good for small k … but for $k = \Theta(n)$ this is $\Theta(n \cdot \log n)$, and hence slower than the "zipper"-style linear intersect



Algorithmic improvements 3/8

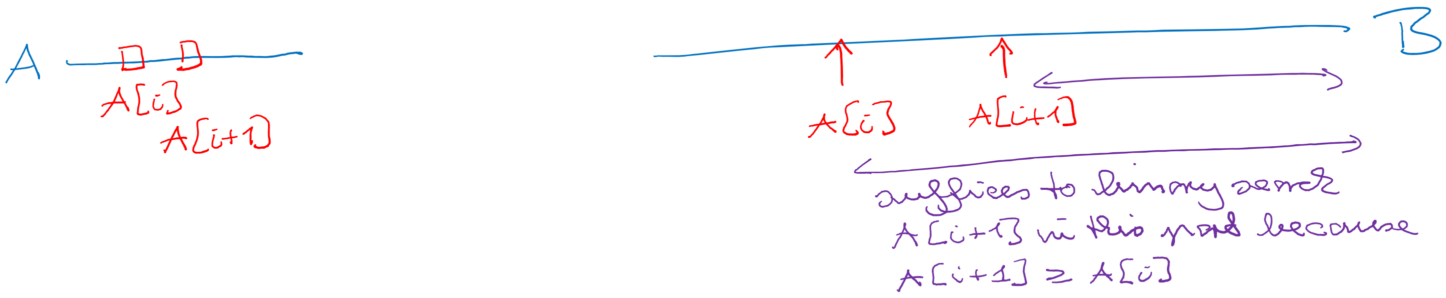
- Binary search in remainder of longer list
  - Time complexity in the best case $\Theta(k + \log n)$

    First element from A towards the end of list B
  - Time complexity in the worst case $\Theta(k \cdot \log n)$

    All elements of A at the beginning of list B
  - Time complexity in the "typical" case $\Theta(k \cdot \log n)$

    Elements of A "evenly distributed" over list B

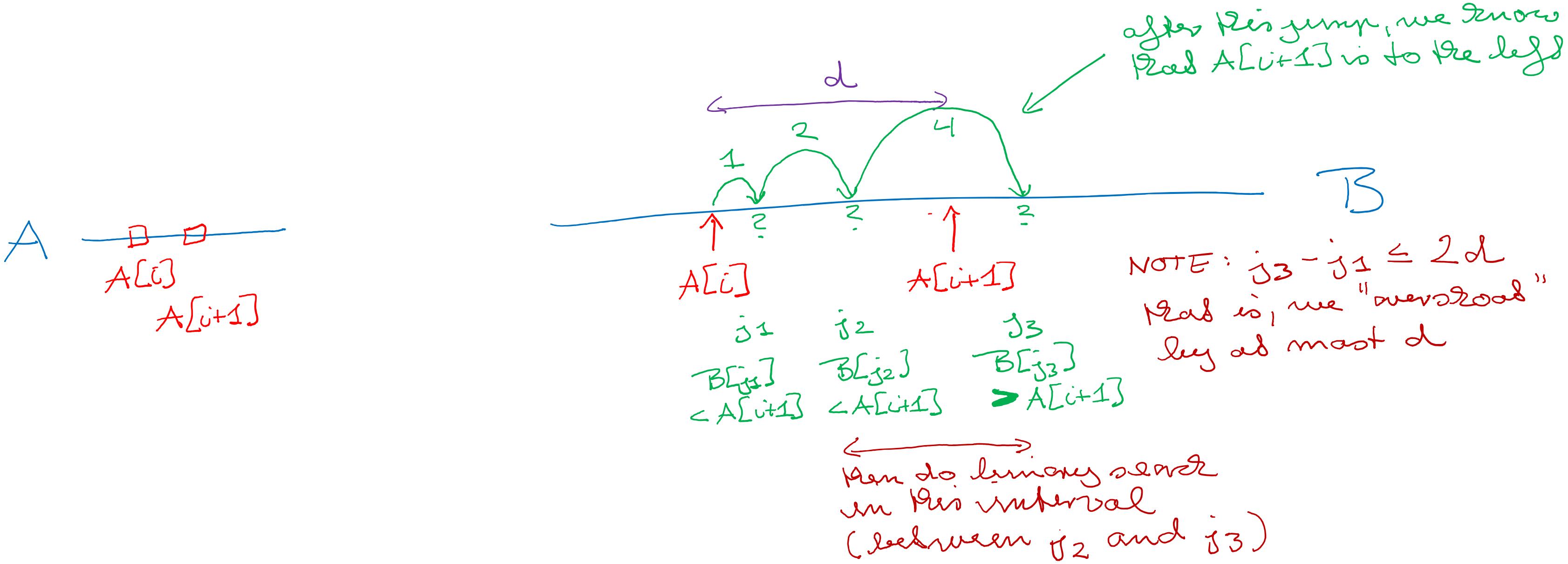
because $\log \frac{n}{k} \approx \log n$ if $k << n$



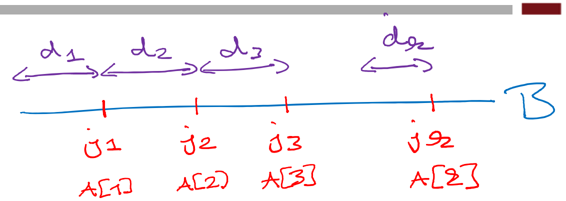
Algorithmic improvements 4/8

- Galloping search
  - **Goal:** when elements A[i] and A[i+1] are located at positions j_1 and j_2 in B, then, with $d := j_2 - j_1$ ("gap"):

    spend only time **Θ(log d)** to locate element A[i+1]
  - **Idea:** first do an exponential search, to get an upper bound on the range, then a binary search as before



Algorithmic improvements 5/8



- Galloping search, time complexity
  - Let $j_1, \ldots, j_k$ the positions of the elements of A in B
  - Let $d_i = j_i - j_{i-1}$ for $i > 1$ and $d_1 = 1$ (the "gaps")

    Note that $\Sigma_i\, d_i \leq n$ = the number of elements in B
  - Then the time complexity is $O(\Sigma_i \log d_i)$

    Not a nice formula, so let's find the maximum value, independent of the particular $d_1, \ldots, d_k$
  - Lemma: $\Sigma_i \log d_i$ is maximized when all $d_i = n / k$

    in case $k = n$: $\log \frac{n}{k} = 0$
  - Galloping search therefore takes time **$O(k \cdot \log(1 + n/k))$**

    This is always O(n) and hence never worse than "Zipper"

    VERIFY YOURSELVES!

Algorithmic improvements 6/8

NOTE: $\max \sum_{i=1}^{k} \log d_i \hat{=} \max \sum_{i=1}^{k} \ln d_i$

- Proof of Lemma … max $\Sigma_i \ln d_i$ under constraint $\Sigma_i d_i \leq n$
  - This is an instance of **Lagrangian optimization**:
    1. Write constraint as equation: $\boldsymbol{\Sigma_i d_i - n' = 0}$ … n' < n
    2. Define $\mathbf{L(d_1, …, d_k, \lambda)} = \Sigma_i \ln d_i - \lambda \cdot (\Sigma_i d_i - n')$
    3. Set partial derivatives = 0 to find all local optima and check the objective function at the borders

$$\frac{\partial L}{\partial d_i} = \frac{1}{d_i} - \lambda \overset{!}{=} 0 \Rightarrow d_i = \frac{1}{\lambda}$$

$\Rightarrow d_i$ are all equal (*)

$$\frac{\partial^2 L}{\partial d_i^2} = -\frac{1}{d_i^2} < 0 \Rightarrow \text{we have a MAX at } d_i = \frac{1}{\lambda}.$$

$$(*) \Rightarrow d_i = \frac{n'}{k} \Rightarrow \sum_{i=1}^{k} \log d_i \leq \sum_{i=1}^{k} \log \frac{n'}{k} \leq k \cdot \log \frac{n}{k} \quad \blacksquare$$

Algorithmic improvements 7/8

- Comparison-based lower bound
  - Recall the lower-bound for comparison-based sorting

    There are n! possible outputs, we have to differentiate between all of them, and only two choices per step

    Hence #steps required $\geq \log_2 (n!) = \mathbf{\Omega(n \cdot \log n)}$
  - We can use a similar argument for intersection / union:

    There are n+k over k ways how the k elements from A can be placed within the n elements from B, …

    Hence #steps required $\geq \log_2 (n/k)^k = \mathbf{k \cdot \log_2 (n/k)}$

    Galloping search is hence asymptotically optimal

Algorithmic improvements 8/8

- Skip Pointers
  - **Idea:** potentially skip large parts of longer list B
  - Skip pointer = special element in list B with a value x and the index j of the first element in B with B[j] ≥ x

    When intersecting, follow pointer if current A[i] ≥ x

    Placement of skip pointers is heuristic … for ES3 you can investigate good placements experimentally
  - Advantage: **very simple** to implement

    In particular, simpler than galloping search and thus often more effective in practice, even if not "optimal"

References

- Textbook

  Section 2.3: Faster intersection with skip pointers

- Literature

  A simple algorithm for merging two linearly ordered sets

  F.K. Hwang and S. Lin SICOMP 1(1):31–39, 1980

  A fast set intersection algorithm for sorted sequences

  R. Baeza-Yates CPM, LNCS 3109, 31–39, 2004

- Wikipedia

  http://en.wikipedia.org/wiki/Lagrange_multiplier