Informatics 2 – Introduction to Algorithms and Data Structures

Coursework 1: A search engine for a large text corpus

October 22, 2019

The purpose of this coursework is to apply ideas from the course to the construction of a search engine for a potentially large but fixed set of text files, such as a corpus of classic literature. Part of the intention is to illustrate how algorithmic ideas may be usefully applied to large amounts of data (e.g. datasets too big to fit into main memory).

This is a **formative** coursework assignment: it is designed to help you develop and consolidate your understanding of the course material and give you further experience in Python programming, but it does not contribute to the final course mark. If you have not worked in Python before, the three Python lab sheets (available from the course Learn page) should give you enough to allow you to attempt the practical, although you will probably pick up more knowledge of the language as you proceed. The coursework consists of three Python programming tasks A, B, C, broadly of increasing difficulty, which may in principle be attempted and submitted independently (though in practice it may be harder to make sense of B and C without attempting A first.)

Submission instructions will be announced on Learn and by email. The deadline for submission is **4pm on Tuesday 5 November**; your marks and feedback will be returned to you by Tuesday 19 November.

In this practical, many of the more routine parts of the code have already been provided for you, leaving it to you to supply some of the more algorithmically interesting parts, Although the total volume of code you are expected to write is not too large (the sample solution takes around 125 lines including spaces and comments), it may also take you some time to **understand the set-up** and find your way around the existing body of code. You are therefore advised to make an early start on the practical to allow yourself time to assimilate all of this.

You are also reminded that it is your responsibility to **test** your programs before you submit them, and in particular to ensure that they work correctly in conjunction with the provided code. You may of course work mostly on your own machines, but before submitting you should do a final check that they run correctly on the `python3.6` installation on the DICE machines, since this is where they will be marked. (Marking will be done by a combination of automatic testing and human eye inspection of your code.)

Likewise, you should add enough **comments** to your code to make it readable. Not only is this a good general habit to get into, but it may actually help your mark by allowing the human marker to see readily that you understand what you are doing. Writing comments is something of an art form: they need not be numerous, they should be short, and they should not merely repeat what can be directly seen from the code. The provided code gives specimens of the level of commenting expected.

If you feel you need help on the Python side of the coursework, **lab sessions** will be running at the usual advertised times, and the demonstrators there may be able to help you. You are also welcome to post queries on the **Piazza forum**, as long as you do not post any snippets of code that might form part of a potential solution. One request, though: before posting, please check through previous postings to see whether your question has already been asked and answered! We have sometimes found that the volume of postings relating to a practical can grow very large and be hard to manage – we will try to answer all reasonable queries within 48 hours, but please bear in mind that this may not always be possible. You are also encouraged to help one another by replying to queries if you know the answer – again as long as you don't give away any part of the solution.

Overview

The three tasks of the practical are:

A: Compiling an index for a large collection of documents using a file-based version of MergeSort. If you have already worked through the Python lab sheets, this part should be relatively straightforward.

B: Using such an index to process search queries. For example, it should be possible to search for all places within the text corpus where *at least two* of the words *friends*, *romans*, *countrymen* appear within the space of 5 lines. (We work in lowercase.) For this part, you will not be following a given algorithm in detail, but will need to think for yourself about the approach you are going to adopt.

C: Reducing the size of the in-memory component of the index as far as possible, using the perfect hashing scheme of Belazzougui, Botelho and Dietzfelbinger (2008) as mentioned in the lectures. This will give you experience of a relatively recent development in the field which is nonetheless fairly simple to implement.

For each of the three parts, you will be inserting code into one of the *template files* provided. Please add your code at the point marked `# STUDENT CODE`, and **do not** make changes to any other part of the code: it is required that your code works correctly in conjunction with the code precisely as given. For each part, it is possible to give a natural solution within 45 lines of code. You need not worry if your solution is a little longer (or shorter) than this – but if you find yourself writing twice this much, you should stop to consider whether your approach is needlessly complicated.

Part of the idea of this practical is to illustrate how algorithms such as MergeSort can help us when the data involved becomes too large to fit in main memory, and has to be stored e.g. on disk. Nowadays, of course, even a modest laptop is likely to have 16 gigabytes or so of memory, so working with a dataset that *really* didn't fit into memory would require you to download some large and unwieldy files, and would also entail excessive amounts of computation time for sorting the index. For the purpose of illustrating the concepts, therefore, we shall be investigating how a relatively small amount of working memory (around 1 megabyte) can be used to process a dataset many times larger. You can imagine how this models on a small scale some issues that are of genuine practical concern for real-world search engines.

To get started, download the file `IADS_cwk1_materials.zip` from the course Learn page, and unpack it (on the DICE machines) using:

```
unzip IADS_cwk1_materials.zip
```

This will yield a directory containing four Python source files (three of which are incomplete templates), five plaintext files containing works of classic literature (three small, two large),[1] and one other sample file.

Part A: Building an index [30 marks]

Open the template file `index_build.py` in your favourite text editor. The following instructions will walk you through some parts of the existing code, giving you what you need to know in order to create the sorted index.

Start up the Python interpreter, and import the contents using

```
import index_build
from index_build import *
```

Look at the declaration of `CorpusFiles` near the start of the file. This creates a Python dictionary for the text files, with a three-letter code for each file serving as the key. The names of the two large files are commented out. We recommend that for the purpose of developing your code you use just the three smaller files to keep things light – once it is working, you are encouraged to add in the weightier offerings.[2]

Using a text editor, take a quick look at the contents of the 'Alice in Wonderland' text file to see what it is like.

Now move down the Python file to the declaration of `generateAllIndexEntries`. You can try this out using:

```
generateAllIndexEntries('raw_entries')
```

Looking in your directory, you'll see that this has created a text file `raw_entries`. Take a look at it. You will see that it has a one-line entry for each word occurrence in the files in question (including the boilerplate at the beginning of the file). The entry contains the three-letter code for the text file followed by a line number, padded so that all entries for a given file use the same number of digits (this facilitates sorting). Note that we convert everything to lowercase, and we also suppress words of length 3 or less.[3]

Now look again at the code for `generateAllIndexEntries` and the preceding functions, and check that you can see broadly how it is working.

To create an index for our search engine, we want to *sort* these raw entries by their words, which can obviously be achieved by sorting the entries themselves. Since we are pretending we only have 1MB of working memory available, we do this by breaking the list of raw entries into chunks which are individually small enough to be sorted in-memory, and then merging all of these chunks together.

The first part of this process is already done for you. Look at the definition of the file `splitIntoSortedChunks`. You can try out this function as follows:

```
splitIntoSortedChunks('raw_entries')
```

[1] The text files were obtained from the wonderful Project Gutenberg site (`www.gutenberg.org`), which makes a enormous number of literary works freely and legally available for purposes such as this. If you wish, you are welcome to add further literary works to your searchable corpus for your own interest: go to the Gutenberg site and look for the plaintext (`utf-8` encoding) versions of the books you want.

[2] OK: you *are* allowed to edit this bit of the provided code, just to vary the selection of files.

[3] This is a cheap measure to avoid cluttering our index with vast numbers of occurrences of words like *the*. Unfortunately, it also cuts out perfectly reasonable words like *sun*. A better solution (more typical of real-world search engines) would be to index all words not appearing in a specified set of *stopwords* (typically short, commonly occurring grammatical function-words).

This will create (in this instance) three new text files called `temp_0_1`, `temp_1_2` and `temp_2_3`. Take a look to see what they are like. Each contains roughly a third of the entries from `raw_entries`, and each has been sorted in-memory using the built-in Python `sort` method.[4] Note too that the call to `splitIntoSortedChunks` returns the number of chunks created (3 in this case).

Take some time to understand how the code for `splitIntoSortedChunks` is working. You will notice that this, like several of the preceding function, makes use of classes for buffered input and output which are defined in the source file `buffered_io.py`. (These are slightly adapted versions of classes that appeared in Lab Sheet 3, and the explanation given there may be helpful.) Study the code in both `buffered_io.py` and `index_build.py` until you understand how these classes are to be used in practice. Note that in this version, a real number ≤ 1.0 is used to specify the *proportion* of the available memory to be allocated to the buffer in question.

Now for the part you have to implement, at the point marked `# STUDENT CODE`. If you have worked through the MergeSort exercise on Lab Sheet 2, you will have a headstart here. There are two subtasks:

1. Write a function `mergeFiles(a,b,c)` which, given three integers a, b, c, merges the contents of two existing, already sorted input files called `temp_`a`_`b and `temp_`b`_`c to create a new sorted file in the same format called `temp_`a`_`c. (For instance, calling `mergeFiles(0,1,2)` with the files you already have should create `temp_0_2`.)

   You may follow the merge procedure described in lectures, but should pay particular attention to what happens when one of the input files is exhausted before the other. You may define other auxiliary functions if you wish.

   For input/output, you should use three buffers each of proportional size 0.3. All buffers should be closed/flushed after use, and your function should conclude by deleting the two input files (use `os.remove(`*filename*`)`).

2. Now use `mergeFiles` to define a recursive function `mergeFilesInRange(a,c)` which will merge all the chunk files `temp_`a`_`$(a+1)$ to `temp_`$(c-1)$`_`c inclusive (as generated by `splitIntoSortedChunks`) to produce a single output file `temp_`a`_`c. You should follow the pattern of the MergeSort algorithm described in lectures. In addition, your function `mergeFiles` should return the name of the output file it creates.

Your `mergeFiles` function will be called by `sortRawEntries`, which is defined immediately after your code. The bottom line is that executing

```
sortRawEntries('raw_entries')
```

should have the effect of sorting the whole file `raw_entries` into a single output file called (in this case) `temp_0_3`, and should return the string `'temp_0_3'`. You should also check that this still works correctly with Shakespeare and Tolstoy thrown in; in this case, of course, the number of chunks will be much larger.[5]

The next few provided functions perform some further stages of processing. First, we 'compress' our sorted list of entries into a master index file called `index.txt`, in which all the entries for a given word are combined in a single line. This will be the main index

[4] Looking at the source files for the implementation of Python itself, one discovers that this does a somewhat idiosyncratic variant of MergeSort with various bells and whistles, modestly dubbed by its developer Tim Peters as 'timsort'.

[5] If you open a window in your file manager to view the contents of your working directory, it's quite fun to watch the temporary files come and go as the computation proceeds.

used by our search engine. Next, we read through this file to create an in-memory Python dictionary, called the *meta-index*, which we can use to locate the entry for a given word. For instance, if `MetaIndex['above']` returns 12, this tells us that the index entry listing all occurrences of *house* may be found at line 12 of `index.txt`.

The function `buildIndex()` draws together all stages of the index creation process as described above. If your MergeSort implementation, then calling this function will create the whole index from scratch and report success. You should inspect the generated file `index.txt`, checking that it has the same form as the provided file `sample_index.txt`.

You should now be able to look up index entries for particular words like this:

```
indexEntryFor('above')
```

Part B: Processing search queries [35 marks]

Before proceeding further, use your Part A solution to generate an index file for just the three shorter text files. Or, if you wish to attempt Part B before completing Part A, execute the following:

```
index_build.IndexFile = 'sample_index.txt'
generateMetaIndex(index_build.IndexFile)
```

Let us now consider how we might locate, say, all the lines in our corpus containing both the words *table* and *chair*. We could use `indexEntryFor` to retrieve the entries for each of these words, which in effect list all their occurrences in the corpus. We could transform each entry into an actual Python list, and then effectively run the 'merge' procedure on these two lists, looking out for line references that appear in both of them.

This procedure would work fine if the number of occurrences of *table* and *chair* are fairly small. Suppose, however, that we had a very large corpus in which each of these words appeared several thousand times. Then the above method would require us to process the entire index entries before we could detect even a single hit for our search – even if such a hit were to occur very early in the corpus.

In this practical, we shall therefore explore a different approach: we process the index entries 'lazily' – that is, a little bit at a time, on demand – using them to generate *streams* of items which can be queried as required.

Open the template file `search_queries.py`, and look at the definition of the class `ItemStream`. This performs the transformation of index entries into streams. You will see what it does by importing the file, and trying the following:

```
indexEntryFor('above')
a = _
A = EntryStream(a)
A.next()
```

Now repeat `A.next()` lots of times, occasionally interspersed by `A.peek()`. You will see that the latter allows us to view the next item without advancing to the next position. Once there are no more items, calling `next` or `peek` will simply return `None` forever.

Your task is to write some code that finds all hits for a given query, by working with the item streams for the search terms involved. For our purposes, a *query* will be specified by the following information:

- A list `keys` of keys (search terms): e.g. `['friends', 'romans', 'countrymen']`.

- An integer `minRequired` ($\leq$ number of search terms) specifying the minimum number of these keys required for a hit.
- An integer `lineWindow` (≥ 1), specifying the size of the window of text within which the keys must occur.

We define a *hit* for such a query to be a pair `(doc,line)` such that:

- `doc` (a string) is a three-letter code for one of our text files T,
- `line` (an integer) is a line number within the file T,
- at least one member of `keys` appears within line `line` of T,
- at least `minRequired` of the keys appear in the segment of T from lines `line` to `line + lineWindow - 1` (inclusive). (For convenience, we adopt the convention that line n of T is the empty string if T has $< n$ lines.)

Notice that by this definition, a hit is actually a reference to a line rather than a set of actual occurrences of the search terms. This means that a given pair `(doc,line)` might be a hit for more than one reason, if there are multiple occurrences of search terms within the line window in question, but we still count these as just a single hit. Notice too that hits can be 'overlapping': e.g. if `chair` appears on line 18 and `table` on lines 17 and 19, then a search for both `table` and `chair` with line window 2 or more will yield both lines 17 and 18 as hits.

More specifically, your task is to define a class `HitStream` (at the point marked `# STUDENT CODE` which supports the following operations:

- Given a query as above, a hit stream `H` for the query may be constructed via:

  ```
  H = HitStream (itemStreams,lineWindow,minRequired)
  ```

  where `itemStreams` is a list of `ItemStream` objects, one for each of the keys.
- Calling `H.next()` repeatedly should yield the sequence of hits for the query, as `(doc,line)` pairs. Your implementation should return *all* hits, including overlapping ones. You should avoid returning duplicate hits (i.e. the same pair twice), although if you do sometimes return duplicates then only a small number of marks will be lost. Once the sequence of hits is exhausted, calling `H.next()` should yield `None` forever.

It is up to you to decide how your implementation will work. You may well find it helpful to include other methods in your class besides `next` and the `__init__` method. (See Lab Sheet 3 for an introduction to creating classes in Python.) Even a simple approach that works fine for small numbers of keys (e.g. ≤ 4) will earn 25 marks if it is well implemented.

For those last 10 marks, however, you should try to find an approach that will remain efficient even when the number of keys is large (say around 100).[6] This will involve devising some data structure that can be maintained internally by a `HitStream` object in order

[6] A semi-plausible use for this kind of search engine is as follows. Suppose we wish to search our corpus for passages relating generally to astronomy. (For instance, we might be wanting to add a touch of class to our paper on astronomy by means of a cute literary quotation.) We might do this by constructing a long list of likely key words (e.g. `star`, `moon`, `planet`, `telescope`), and then searching for all places where (say) at least 3 of these occur within the space of 5 lines.

to reduce repeated work. There is some scope for creativity here, but please keep it in proportion: e.g. don't attempt anything that would involve more than 80 lines of code.

The acid test is then whether your code works correctly in conjunction with the provided functions `easySearch` and `advancedSearch`. Just like a real search engine, these will return and display a small number of hits at a time (the default is 5); you can request more by calling the `more` function.[7]

The function `easySearch` will simply yield all lines that contain all of the keys. Widen your console window a bit for best results. A couple of nice ones to try (with Shakespeare loaded in) are:

```
easySearch(['exit','pursued'])
easySearch(['palpable','very'])
```

The function `advancedSearch` allows you to supply `lineWindow` and `minRequired` parameters, e.g.

```
advancedSearch(['friends','romans','countrymen'],5,2,20)
```

Part C: Meta-index compression via perfect hashing [35 marks]

The search engine we have constructed keeps most of its index information on disk (in `index.txt`), but also requires an in-memory dictionary `MetaIndex`. This itself still occupies significant space, since (for instance) it includes as keys all the words (of length ≥ 4) that appear in the corpus. The motivation for the last part of the assignment is to see whether we can get away with a much smaller 'meta-index': this would not only economize on space, load time etc., but might also allow the meta-index to live in 'fast memory' where it can be accessed very quickly.

The intention is that if we can construct a *perfect hash function* H mapping our keys to distinct integers, we may simply use this as our meta-index, provided we reshuffle our main index file so that the entry for each key k does indeed appear at the line number given by $H(k)$.[8] The algorithm we follow offers a good way to construct perfect hash functions that can be very compactly represented within memory.

Actually, the code for Part C will be quite independent of Parts A and B: we are simply showing how to construct a perfect hash table for a given list of strings, yielding a highly compressed meta-indexing scheme. You may, however, like to try out your solution to Part C on the list of keys for your search engine, given by `MetaIndex.keys()`. We also provide a way of testing whether the resulting hash function is a perfect one.

Take a look at the file `perfect_hashing.py`. There is some simple machinery for hashing of strings modulo a number p (maybe prime, maybe not). The precise details aren't important, but you should experiment a bit with the functions `modHash` and `buildModHashTable` to get the feel of what they do.

There is then some code for selecting a prime number close to a given n, followed by a function `stdHashEnum` which, for a given number `m`, yields a whole family of (reasonably 'independent') hash functions mapping strings to integers `0`,...,`m-1`. For example:

```
stdHashEnum(1000,0)('chair')
stdHashEnum(1000,1)('chair')
```

[7] Admittedly, this 'incremental' character is of rather dubious value in our setting, since our hits are not *ranked* in any way.

[8] We have not actually implemented this idea yet. Doing so is not necessary for the purpose of the coursework, since there are other ways to test your solution.

Again, the precise details aren't important, but you should experiment with this to get the idea.

Our goal is to construct a *perfect hash function* mapping a specified list of n words (given by `keys`) to integers $0, \ldots, m-1$ for some specified m. (The intention is that $m > n$, but not by too much.) The method proceeds as follows.

First, we 'mod hash' our set of keys down to numbers $0, \ldots, r-1$, where r is some suitably selected prime number. (This is done by the code for the `Hasher` class later in the file.) This results in a list `L` of r *buckets*, where each bucket is itself a (possibly empty) list of strings. Next, we try to assign to each of these buckets a number j such that `stdHashEnum(m,j)` is a suitable mini-hash function to apply to that bucket. 'Suitable' means that all these mini-hash functions *taken together* should map the contents of all buckets to numbers `0`,...,`m-1` with no clashes.

Your task is to write a function `hashCompress(L,m)` which takes a list `L` of buckets and a desired table size `m`, and suitably selects a number j for each bucket as above, storing the number j for the ith bucket at position i in a list `R`, whose length is r (the number of buckets). This array `R` is what your function should return; it will become the value of the field `hashChoices` in the `Hasher` class. The `hash` method for this class then gives the hash function we have constructed, which is intended to be a perfect hash function.

How does one go about choosing j-values to ensure this? We may adopt the following approach:

- First, sort the list of buckets `L` in order of decreasing size. In the sorted list, it will be useful to pair each bucket with a number i indicating the position in original list `L` where this bucket came from.
- Create a list `T` of m booleans, initially all `False`, intended to record which slots in the main table are currently 'taken'. Also create a list `R` of integers (initially zero), one for each bucket.
- Go through the buckets in order of decreasing size. For each bucket B, search for the smallest integer $j > 0$ such that the mini-hash function `stdHashEnum(m,j)` is *suitable* for B. This means that:
  - it maps all elements of B to currently free slots in the main table,
  - it doesn't map any two elements of B to the same slot.

  Once a suitable value of j is found,[9] record this value as `R[i]` (where i is the original position of the bucket B), and mark all the slots to which we are sending elements of B as 'taken'.

Although this may seem complicated, the list comprehension syntax of Python allows it to be coded quite economically. Your code for `hashCompress` should be added at the point marked `# STUDENT CODE`; you may also define any auxiliary functions you find helpful.

The provided class `Hasher` will then call your code to create a hash table. As you can see from the code, this class gives the user control over various parameters. A typical use would be:

```
H = Hasher (MetaIndex.keys(), 5.0, 0.8)
```

[9] Actually, there's no absolute guarantee that a suitable j will *ever* be found. But as long as our mini-hash functions are 'sufficiently random and independent', this will happen 'almost surely', and the approach works well in practice.

where the 0.8 is the desired load on the resulting table (i.e. n/m), and the 5.0 controls the value of r via $r = n/5.0$. Strings can then be hashed by calling e.g. `H.hash('chair')`. You may also use the function `checkPerfectHasher` to check whether a hasher is a perfect one, simply by tabulating its hash table for the given set of keys.

Finally (purely for interest), you may use `bestCompression` to glean some statistics on how much space would be required to store the hash function, given a simple compression scheme. A typical value for a hasher `H` constructed above would be 1.6 bits per key, or less – obviously tiny compared with the original `MetaIndex`!