Programming Comps - January 18-24, 2011

Virtual Sticky Notes

Out: Tuesday, Jan 18th, 2011, 9AM
Due: Saturday, Jan 22nd, 2011, 8AM
Presentations: Monday, Jan 24th, TDB

1 Introduction

With the prevalence of portable devices that are aware of their geographical location, many new applications are possible. You just had the idea of a great use for this technology: you want to create a virtual sticky note service, through which visitors to a physical place, say, a monument or a museum, can leave virtual notes attached to a location that can be fetched by future fellow visitors.

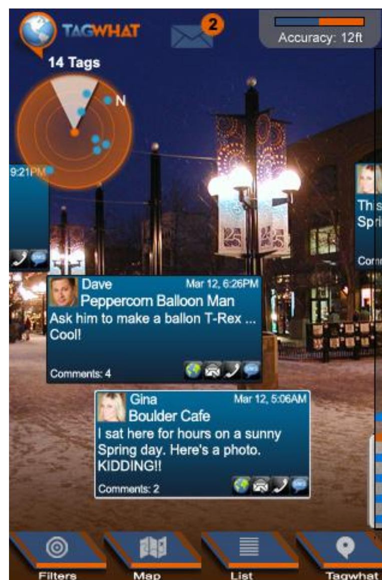


Figure 1: An example of what your interface might look like. (Pretend for a moment that this company, Tagwhat, is not already doing this...)

Since you don't yet have users, but have to demonstrate to your investors that your solution will work efficiently, you will build a prototype with a database of geo-tagged posts

that we gathered from Twitter for you. In this exam your task is to write two programs, `buildtree` and `querytree`. `buildtree` will read the input data and create a database on disk to organize the data. `querytree` will then efficiently answer a series of queries on the data, given in an input file. The queries are of the form: 'Give me all the posts that are within r Km of location Q'. When testing your program, all of the performance tests will have queries with a radius of at most 200Km, although your program must work correctly, even if not so efficiently, for larger query regions.

You will receive a sample file with geo-tagged Twitter posts. This is not the full set of posts we will evaluate your program with, but it is in the same format. We will evaluate the programs with up to 6,000,000 twitter posts; make sure you can handle this scale.

To answer the queries efficiently, you will index the posts using a spatial indexing data structure for point data, a kd-tree. A kd-tree is a type of binary tree that recursively partitions space into regions. Internally, you will convert the latitude and longitude coordinates into 3 dimensions in a cartesian space (detailed below), so your tree will be a 3d kd-tree.

The tree you will build will have two types of nodes, internal nodes and leaf nodes. The internal nodes are only index nodes, and record how you partitioned the space and how to find regions. The data itself will be stored in the leaves of the tree. Each leaf will represent a region of the space, with all the points contained in that region. (This is a slight departure from the original kd-tree, in which each leaf node represents one point). The reason you want to store several points in each leaf is that it makes it much more efficient to fetch records that are close in space from the disk.

You can assume that the entire dataset fits in memory for the purpose of building the tree. With this assumption, the tree you build should be a complete, balanced binary tree. You can choose the format of the disk representation, provided that `querytree` can understand it. For example, you can create a single file, or you can create one file for the index and another file for the data (the leaf nodes). Do not create one file per leaf node!

For answering queries, you will want to minimize the number of seeks to disk. One way to do that is to load the entire index into memory before answering any queries, and then, for each query, load the partition(s) that can contain points to answer the query. Since the query region may intersect more than one leaf node, you may need to load more than one node from disk. You can treat each query independently: you don't need to worry about caching leaf nodes, let the OS do that for you.

Although this is a programming exam, meaning that you have to deliver working programs according to these specs, you will also be judged on how you approach and solve the problem. The evaluation criteria are detailed in Section 4.4. In particular, beyond correctness, your program must be efficient in terms of time and space, legible and well-commented (as you would like it to be maintainable), and tested appropriately for correctness, performance, and scalability. After you hand in the required files, you will have a 10-minute presentation, where you will present and justify your choices, and have to convince the exam committee that your program works correctly and efficiently. The presentations will be on Monday, January 24th, and individual slots will be scheduled after you hand in the assignments.

2 kd-trees

kd-trees were introduced in the late 1970's to store multidimensional data in databases. A kd-tree is a generalization of binary trees to k dimensions.

The tree you will implement has two types of nodes, internal and leaf nodes. The internal nodes are index nodes that record successive partitions of subspaces, while the leaf nodes contain the data points.

Each internal node consists of two pointers, $left$ and $right$, and a discriminator. A discriminator is a tuple $\langle d, v \rangle$, where d is a dimension identifier (ranging from 0 to $k-1$), and v is a value along that dimension. Let P be a point in k-space, and $C_d(P)$ be the value of the d-th dimension of P's coordinates. Then, an internal node with discriminator $\langle d, v \rangle$ and responsible for region R divides R in two regions R_{left} and R_{right} as follows:

$$R_{left} = \{P \in R | C_d(P) < v\} \tag{1}$$
$$R_{right} = \{P \in R | C_d(P) \geq v\} \tag{2}$$

The two pointers may either point to another internal node, or to a leaf node, which is a set of data points. There is a special root pointer, which may be null, if the tree is empty, or point to either an internal node or to a single leaf node. The root always corresponds to the entire space.

Figure 2 shows an example kd-tree with two dimensions.

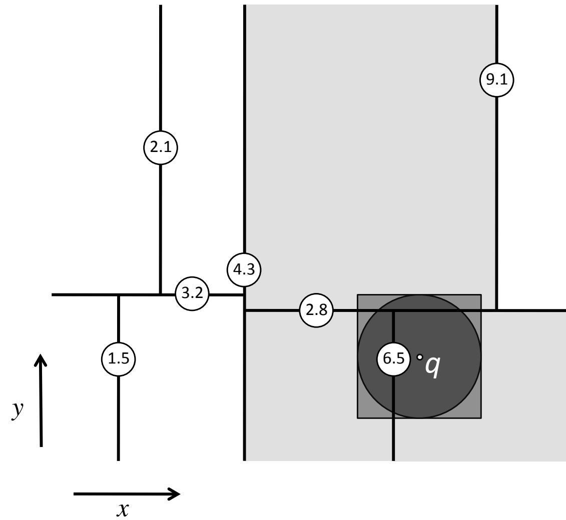
2.1 Building the Tree

You don't have to worry about dynamic insertions into the tree: assume you have all of the points before starting. (It was only in 2010 that Google started allowing changes to its index of the entire Web, before that it was periodically rebuilt!). You can assume all of the posts data fits into memory for the purposes of building the tree. Your first job is to build a *balanced* tree using these points: all leaf nodes must be at the same depth from the root. This may make storing the tree structure easier and more compact.

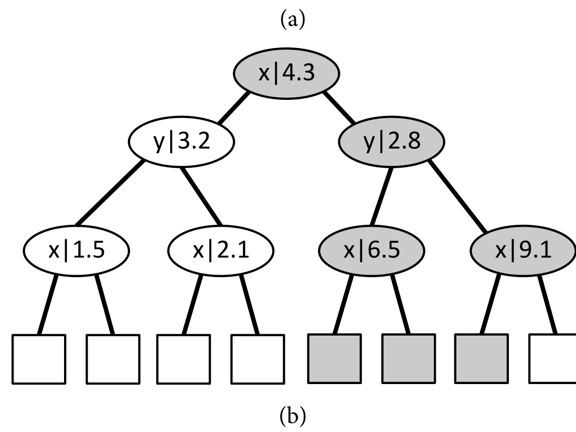
At each internal node you have to choose which dimension along which to split the space, and a value. One way to do this is cyclically: at level l of the tree you use dimension $d = l \bmod k$ to split the region. This seems to work well in practice. The other choice is where to split the region. You want the tree to be balanced, so you want the same number of points on each partition, ideally. Odd numbers of points and points with equal value may spoil your perfect division, but it should be good enough: use the median along dimension d as the splitting value according to equations 1 and 2.

To find the median at each level you must spend at most linear time. Note that you are allowed to sort the points once along each dimension beforehand.

Another decision you must make is how many points to store in each leaf node. You must justify your answer in the documentation with convincing arguments. You can store one datapoint per leaf, all the way to all data points (hint: both extremes are bad decisions!)



(a)



(b)

Figure 2: Spatial representation of a 2-d balanced kd-tree with 3 levels of division (a). q is a circular range query around a point, and is answered by walking the tree, pruning the subtrees that cannot intersect the query bounding box. (b) shows the corresponding binary tree. Each internal node in the tree shows the split dimension and value.The gray nodes are those visited to answer query q.

Remember that you should minimize the amount of data you read from the disk when answering queries, and you also want to minimize the number of random reads (seeks) from the disk. What are other implications of the leaf node size? What are the tradeoffs involved? Most queries will range from 1 to 200 Km in range. (Your program should still be correct for larger queries, even if less efficient).

Section 3.3 has very important information regarding the data you will store. In particular, the text of the posts is encoded in UTF-8, and you must answer the queries back with the same text that was read in.

2.2 Querying the Tree

In this assignment you will use the tree to answer queries of the following form, where Q is given in latitude and longitude coordinates:

List all posts within r Km of point Q.

Your program will have to list all of the points satisfying the query, including multiple posts with the same coordinates. Recall that you will use 3-dimensional cartesian coordinates for the points (see Section 3.4), derived from their latitude and longitude. Since the queries will have a relatively small range of up to 200 Km, instead of using the distance along the surface of the Earth, we will use the linear distance between the points to test if they satisfy the query. In other words, we are looking for the points that lie within a ball of radius r Km around point Q. Formally, the answer to the query is given by:

$$\text{All points } P \text{ s.t. } \|\overrightarrow{P} - \overrightarrow{Q}\|_2 \leq r \tag{3}$$

To answer this query, you will search the tree for all leaf nodes that intersect the cubic, axis-parallel bounding box of the query region, and then, for each of these leaf nodes, iterate over its points checking to see if they fall inside the ball. At each internal node you visit you only need to access the information in the node and in the query range to decide which of the two subtrees (or both) to visit. Figure 2(b) shows an example query in a 2-d kd-tree, with the nodes visited to answer the query shaded in gray.

Your `querytree` program should answer the queries using only the file(s) created by your `buildtree` program. You can load the entire set of internal nodes in memory, but should not load all of the data nodes, only those that intersect with the query bounding box. This means your index has to have the information on how to find the leaf nodes in the file on disk. Again, do not worry about caching these leaf nodes between queries.

3 Your Task

3.1 Programs

It is your task to *implement the algorithms outlined in this document*. Do not make up a different approach to tackle the problem. If you do, you will fail the exam. The choices of algorithm, data structure, and coordinate representation are fixed.

You will write two executable console programs: `buildtree` and `querytree`.

`buildtree` has one argument: the name of the raw input file, which will have a `.twitter` extension. The format of the input file is described in Section 3.3. `buildtree` will produce one or more files in the local directory. It will also report the running time, the number of entries in the database, the depth of the tree, and the minimum, average, and maximum number of entries per leaf node, according to the format below:

```
> buildtree geoposts.twitter
 built geoposts.kdb, geoposts.idx
 entries | total time (s) | depth | index nodes | leaf nodes | min | avg | max
 nnn | nnn | nnn | nnn | nnn | nnn | nnn | nnn
```

The second program, `querytree`, will load the database created by `buildtree` and answer queries from a query file. It will be invoked with the name of the data file that was used by `buildtree`, without the `.twitter` extension, and the name of a query file, and will produce an output file with the results of the queries. It must report the time it took to answer the queries in the file. Note that `querytree` will have to know how to find the database file(s) based on the name of the original datafile (in the local directory), so you have to create a naming convention for your database file(s) that allows this.

```
> querytree geoposts queries1
  loaded database with nnn entries
  processed queries in 3.12s
  results in geoposts.queries1.txt
```

If you use a language that does not produce executable files directly, such as Java, you must provide wrapper shell scripts that take the arguments as described above and call the programs with the right command line.

3.1.1 `querytree` input

The query files used as inputs to `querytree` have one query per line. Each query is simply a query id (an integer), the coordinates of a point, and radius of the query, given in an integer number of meters. The coordinates are given in the same format as in the twitter data: `[<latitude>,<longitude>]`, as floating point decimal degrees.

For example, the following is a valid query file:

```
1 [41.827218,-71.399654] 1500
2 [-19.957817,-43.914571] 100
3 [-29.2,80.43] 10000
```

You don't have to worry about malformed query files.

3.1.2 `querytree` output

The output of the `querytree` program must adhere to the following specification. Each response to a query will start with a query line containing:

```
q <query id> <latitude> <longitude> <range(m)> <number of points>\n
```

It will then be followed by `<number of points>` lines, one for each point. Each point line will have the format:

```
p <latitude> <longitude> <text>\n
```

The following is a valid query result file for the query file above:

```
q 1 41.827218 -71.399654 1500 2
p 41.837008 -71.408524 I don't know what to get #toohungry
p 41.827834 -71.386191 RT I don't know what to get #hungrytoo
q 2 -19.957817 -43.914571 100 1
p -19.956983 -43.914772 Que vista linda!
q 3 -29.2 80.43 10000 0
```

3.2 Implementation Language

You are free to choose the programming language to use, but should take care that the programs do not take an inordinate amount of time. Also, languages that deal well with UTF-8 text and allow for easy byte-based random access to files will make your life easier. *You should justify your choice of language in the documentation.*

3.3 Data Format

The dataset you will deal with consists of real Twitter posts from early December, 2010, only slightly cleaned. The text of the posts comes in UTF-8 text encoding. This means that *characters MAY NOT correspond to single bytes*. If the language you choose handles UTF-8 text, you should be fine. You don't have to process or filter the strings in any way, so you can choose to treat them as bytes if you wish. The only requirement is that the query results in the output file should be properly encoded UTF-8 strings, identical to the posts in the original data. Each post has at most 140 *characters*, and, because of the UTF-8 encoding, potentially more bytes than that! You don't have to worry about malformed input data, other than what is described here.

The posts are given in a file with one post per line. Each line is in Unix format, terminated by a single '0x0a' character. Each post has fields separated by three characters: ' | ' (0x20, 0x7c, 0x20). The posts have been cleaned so that there are no line feeds, carriage returns, or '|' characters (neither 0x0a, 0x0d, or 0x7c) in the text of a post.

Each post has the following fields:

- Coordinates: `[<latitude>,<longitude>]`
- Date (ignore)
- User id (ignore)
- GMT Offset (ignore)
- Text of post

You only need to care about the coordinates and the text, and may ignore the rest. Here is an excerpt of an input file:

```
[-22.996575,-43.346156] | Mon Dec 13 12:28:57 +0000 2010 | 47815782 | -10800 | I ju
st became the mayor of Asterisco: Capacitação e Treinamento on @foursquare! http://
4sq.com/btAkg2
[4.67769,-74.08343] | Mon Dec 13 12:28:57 +0000 2010 | 71845784 | -18000 | la mejor
 y más rápida opción en la mañana: Caminar
[-22.996575,-43.346156] | Mon Dec 13 12:28:57 +0000 2010 | 47815782 | -10800 | I'm
at Asterisco: Capacitação e Treinamento (Avenida das Américas, 3333 grupo 412., Rio
 de Janeiro). http://4sq.com/btAkg2
```

Note also that there may be multiple points with the same coordinates. When returning answers to a query, you must return all of them.

3.4 Spatial Representation and Distances

Instead of using latitude and longitude directly, and so you don't have to worry about spherical geometry, internally you will convert the latitude and longitude coordinates of the input data into 3-dimensional cartesian coordinates with origin at the center of the Earth: your kd-tree will be a 3d kd-tree.

Assume the Earth is a perfect sphere with a radius of $R = 6.3781x10^6m$, that the positive z-axis goes through the north pole, the positive x-axis goes through latitude, longitude (0,0), and that the positive y-axis goes through latitude, longigude (0,90), somewhere in the Indian Ocean. Thus, for a point i with latitude λ_i and longitude ϕ_i, you will have:

$$z_i = R \sin \lambda_i \quad (4)$$

$$x_i = R \cos \lambda_i \cos \phi_i \quad (5)$$

$$y_i = R \cos \lambda_i \sin \phi_i \quad (6)$$

Again, to evaluate the queries, you will approximate the distance from the query point with the linear euclidean distance between the points, which is fine for queries of up to a few hundred Kms.

The answers to queries must be given in latitude and longitude coordinates, so you may want to keep them around as part of the records instead of converting back.

4 Logistics

4.1 Material

This document, together with a dataset with 100,000 posts, and a small sample query file are located in `/home/rfonseca/comps`.

4.2 What to Hand In

You should hand in a single file named comps2011_<your_id>.tar.gz, containing:

- your source files
- a file called README with *detailed* instructions on how to compile both programs. You are responsible for testing that your instructions work on a standard departmental linux machine.
- a file called Documentation.pdf, documenting and evaluating your program. Please see section 4.3 for what to include in the evaluation. This file must be a typeset PDF document.

After compiling your program, there should be two executable files (which may be wrapper scripts) named `buildtree` and `querytree`, which we will invoke with different test datasets.

Do not hand in any sample input data; we will test the program with our own input files!

To create the hand-in archive, please follow these steps to maintain the anonymity of your submission:

1. Create a directory called `comps`.
2. Copy the files to be handed in into the `comps` directory.
3. From `comps`' parent directory, issue the following command:

```
tar -c -v -z --owner <id> --group <id> -f comps2011_<id>.tar.gz comps
```

If you don't use the owner and group options, the archive will contain your real user and group names, and won't be anonymous.

You will hand in the archive file by email to `marek@cs.brown.edu` by the deadline, Saturday, January 22nd, 2011, at 8AM. This is a HARD deadline, and won't be extended. If you don't hand in the exam by then, you will fail.

4.3 Documentation and Evaluation

As much as it is a programming exam, this is also a test of how you approach and solve problems. As in the course of the PhD program and after you graduate you will routinely face problems for which there is no prior solution; you will have to convince yourself that

your solution is correct and efficient. Whenever you publish your results, you will create a permanent record or your work, which will be open to the scrutiny of the entire research community, forever. Thus, it is of utmost importance that you are confident of the results you publish.

Accordingly, in this exam, you must use sufficient means to know that what you are delivering is correct, and works according to the specification. This involves, but may not be limited to, having enough test cases. We will not give out extra sample query files or answers, and the sample dataset that you are given has more than one order of magnitude less posts than our largest dataset.

A key requirement of this exam is that you describe, test and evaluate your programs. Testing asserts the correctness of the programs. Evaluation asserts that the programs perform and scale as they should. All this must be in the documentation.

The documentation should allow someone to understand the important algorithms in your code. You should also justify all of your choices not covered in the specification. For example, the size of the leaf nodes is an important aspect. What are the tradeoffs involved? How did you choose it?

Your documentation should, at least:

- Describe your implementation. What are the algorithms you used for key parts? For example, how do you find the median when building the tree?
- Describe your testing strategy. How do you know the answers you are returning are correct? Did you create test sets for which you knew the answer? Did you evaluate corner cases?
- Evaluate the scalability of your programs. Does it agree with what you would expect or theoretically predict? Is it linear, logarithmic, polynomial, etc? How does the performance change with varying:
  - Number of records in the database (linear here would be bad)
  - Size of the query region
  - Density of points in the query region
  - Size of the leaf nodes
- Justify your choices: implementation language, specific algorithms, parameters. If you can, show experimental or analytical evidence to support your choices.

Graphs and tables are your friends.

Lastly, part of the documentation is in your code. The code should be well organized, legible, and well commented. A person unfamiliar with the code should be able to read it and maintain it.

4.4 Evaluation Criteria

This exam is partly about programming, and partly about how you approach the problem, present your solution, and convince yourself and the committee that it works.

You will be evaluated based on *all* of the following criteria:

- Correctness, efficiency, scalability.
- Structure, readability, and maintainability of your code
- Quality of documentation, according to Section 4.3

It is important that your work passes a critical threshold with respect to all criteria. A super-efficient code that is poorly documented and hardly maintainable will be regarded as a failure. So will a software engineering masterpiece that does not work as specified or that does not perform well. Definitely do not code in assembler or shell scripts!

All work must be your own. This includes code and the thinking behind it. You may not discuss the problem in any way with anyone else. Your only conduit for clarifications is by email to `rfonseca`. Responses to such questions will be cc'd to the comps mailing list when appropriate.

Good luck and have fun!

Appendix: A Useful Tool

This is not mandatory, but may be helpful. You can use Google Maps to visualize a set of points given their latitude and longitude very easily. All you have to do is create a KML file with the points, and point Google Maps to it.

1. Create a KML file with your points, and place it into a web accessible location. Let's say that you chose http://www.cs.brown.edu/you/points.kml
2. Go to http://maps.google.com/?q=http://www.cs.brown.edu/you/points.kml
3. See your points on the map!

You can also zip-compress the file, and name it .kmz . It may not be a good idea though to have too many points. Lastly, you can also load the file in Google Earth.

Here's a simple KML file you can base yours on. Each document can have several folders, and each folder can have several placemarks.

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Document>
 <Folder>
```

```
  <name>Query 1</name>
  <Placemark>
    <description>
     I'm at Kementerian Perhubungan RI (Jl. Medan Merdeka Barat No. 8, Jakarta
       Capital Region) w/ 13 others. http://4sq.com/bJQ5FA
    </description>
    <Point>
      <coordinates>106.82234287,-6.17485965</coordinates>
    </Point>
  </Placemark>
 </Folder>
</Document>
</kml>
```