CSC6210: Advanced multidimensional search — Fall 2009
CSE department, Chinese University of Hong Kong

Lecture 1: Introduction

By Yufei Tao (taoyf@cse.cuhk.edu.hk) Last revision: Sep 5, 2009

Forewords

In computer science, some problems are the building bricks of others. Sorting, for example, is one of them. Once you know this can be done in $O(n \log n)$ time, you plug it in anywhere you need it in solving your current problem. Binary search is another one. Whenever you have an ordered list of elements, you know any particular element can be found in $O(\log n)$ time. I personally hope that I know as many such building bricks as possible – the more I do, the shaper I will become in dealing with new problems, and the faster I will react to what is being said in a paper or in a talk.

In this course, we are going to discuss several such problems that are more difficult than sorting and binary search. All of them are geometry problems in multidimensional spaces. We are all familiar with one such space – the earth we live in, which has a dimensionality of 3. In research, however, often we have to deal with spaces of higher dimensionalities. They may look a bit scary at first, because a space with dimensionality of 4 or above is difficult to visualize. But as we will see, in many problems, once we have grasped some intrinsic properties, many high-dimensional spaces are not much more difficult to comprehend than the 2d one.

You may have heard of the term *computational geometry*. This is one of the most important areas in computer science, and covers the topics of this course. Its importance comes from the fact that, in many applications, there exist a natural way to model the underlying data as multidimensional points, rectangles, circles, and so on, and thereby, convert the original problem to a geometry one. In the sequel, we will illustrate this for some of the problems that we will study in detail later. At the end of the lecture, we will go a little technical by describing the memory model to be assumed.

1 Problems and applications

Range search. Imagine the territory of Hong Kong to be a 2d space (e.g., the two dimensions are the longitude and latitude, respectively). Assume that we have collected all the hotels in Hong Kong, and know exactly where they are. Compared to the size of HK, each hotel is so small that it can be represented as a point. Now, a tourist is coming to HK, and plans to stay in the Sha Tin district; so s/he would like to see where the hotels are in Sha Tin. Now that we are treating the hotels as points, this is equivalent to asking where the points are in the geometric region that corresponds to Sha Tin.

The above is an application of *range search*. Formally, let D be a set of points in a d-dimensional space. Given a query region q, the goal is to return all the points of D that lie in q. Figure 1 shows an example where q is a rectangle. Next, we will concentrate on rectangular query regions, for which the problem is often called *orthogonal range search*. We will omit the word "orthogonal" unless confusion may be caused.

Let us see an example of range search in a high-dimensional space. There are many apartments for

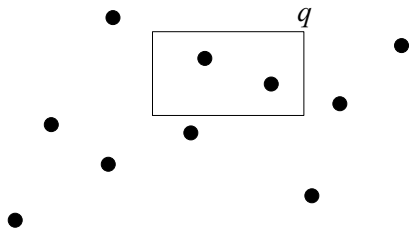

Figure 1: Range search

sale the real estate market. Each apartment has many attributes that a typical buyer is interested in. Here are some examples: (i) *size*, (ii) *price*, (iii) *pollution index of the neighborhood* (from 1 to 10), (iv) *security index of the neighborhood*, (v) *distance to the nearest subway station*, and so on. Each attribute is a value; hence, if we regard each attribute as a dimension, an apartment can be represented as a point in a 5d space.

Assume that I am looking for an apartment. The total number of apartments in the market is too large for me to consider, so I would have to eliminate a majority of them that do not satisfy my preferences. First of all, I cannot afford an apartment over 4 million HK dollars, but I do not want to consider anything below 3 million either. Furthermore, the apartment should have at least 800 square feet, and must be within 500 meters from a subway station. Moreover, I do not want to live any area where the pollution index is higher than 3, or the security index lower than 8. Putting all my preferences together, I get a 5d rectangle (whose extent along, for example, the *security-index* dimension, is $[0, 8]$), which an apartment must fall into in order for me to consider. This is another application of range search.

Now let us think a bit how to answer a range query. First, assume that no pre-processing is allowed; namely, we know nothing about the dataset D in advance. In this case, no trick can be done: we will have to scan the entire D to guarantee returning the correct result in any case. This means that the query will have to be $\Omega(n)$, where n is the cardinality of D (i.e., $n = |D|$). In practice, n can be fairly large such that spending $\Omega(n)$ time for every single query is just too expensive. To obtain better query time, we must be able to *pre-process* D, typically, into a certain data structure. Given a query, we can leverage the structure to focus our search only on a subset of D, which would be considerably faster than a bruteforce scan. In fact, the one-dimensional case is familiar to all of us – this is exactly the purpose of the *binary tree*.

If query time was the only concern, pre-processing would be fairly simple. For example, assume that the dimensionality is 2, and each axis has t possible coordinates. Hence, the whole data space consists of t^2 points. Every (rectangular) region is uniquely characterized by two corners. Hence, there are only $O(t^4)$ different queries. We can pre-compute the answer to each of them, and store all answers properly. At run time, given a query, we can directly return its (pre-computed) answer. The only issue left is to identify which query (of the $O(t^4)$) it is in order to obtain the memory address where its answer is placed. This can be easily done by hashing in $O(1)$ time. Hence, the overall query cost is $O(k)$, where k is the result size. Note that this is asymptotically the lowest cost because even enumerating k points itself takes $\Omega(k)$ time already.

The drawback of the above approach is that it requires prohibitively expensive space in practice. So we have to be smarter in order to achieve a more practical balance between the space and the

query cost. The ultimate question we would like to ask is: *if the space consumption must not exceed a certain bound (e.g., $O(n)$), what is the best query time possible?* A similar, but opposite, question is: *what would be the minimum space cost if we want to achieve a certain query time, e.g., $O(\log n + k)$?* We will answer these questions in this course.

Nearest neighbor search. Let D be a set of d-dimensional points. Given a query point q, a *nearest neighbor query* returns the point o^* in D that is closest to q, namely, there does not exist any other point $o \in D$ such that $\|o, q\| < \|o^*, q\|$, where $\|., .\|$ denotes the distance of two points, defined based on a certain metric (e.g., the "straightline distance", better known as the *Eudliean distance*, or the ℓ_2 *norm*). The object o^* is called the *nearest neighbor* (NN) of q. See Figure 2 for an example.

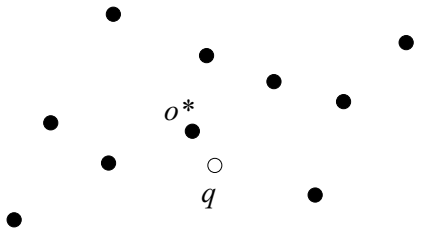

Figure 2: Nearest neighbor search

NN search has abundant applications in practice. The following are some representative ones.

- *Geographic information system.* For example, imagine that the black points in the above figure are the gas stations in HK. So an NN query can be used to find the station that is the nearest to my current location.
- *Profile-based marketing.* Recall the 5d dataset mentioned earlier about the apartments on sale. In that context, each customer may specify a point in the same 5d space to indicate a hypothetic apartment that s/he would be willing to buy. An agent may then suggest the apartment in the market that is the closest to that hypothetic point.
- *Content-based retrieval.* Assume that there is a collection of images. We may want to retrieve the image that is most similar to an image given by a user. For example, if the user gives a picture of sunset, it would be nice we could return another picture of sunset from our collection. An effective way to achieve the purpose is via *feature extraction*. Specifically, we may first identify certain features that can best distinguish the images in our database. These features can be, for example, the brightness, contrast, saturation, other color-oriented properties such as percentages of red, blue, and green, and so on. This way, each image has been converted into a point in a multidimensional space. After converting the user-specified picture into a point using the same approach, the original problem is essentially to find the NN of the query point.

  Similar ideas are useful in time series analysis, text retrieval, fingerprint recognition, duplicate detection, and many others.

A direct extension of NN queries is k *nearest neighbor search*, which finds the k points in D closest to the query point q. Here, k is by far smaller than the size n of D. kNN search has even more

extensive applicability than single NN search. For instance, a scenario which requires $k > 1$ is the *kNN classifier* widely adopted in pattern recognition. Here, we have a training set D, where each object has a class label. The objective of a *classifier* is to decide the class of a new, unknown, object q. A kNN classifier solves the problem by reporting the most frequent class label of the k NNs of q. A reasonable choice of k to achieve good accuracy ranges from 10 to 100.

In practice, a *near* neighbor (albeit, not the nearest) already fulfills the needs of many applications (e.g., the ones mentioned earlier). This motivates the problem of *approximate nearest neighbor search*. Given a point q, a *c-approximate NN query* returns a point $o \in D$ whose distance to q is at most c times the distance from q to its real NN o^*, or formally, $\|o, q\| \leq c\|o^*, q\|$. Note that the answer o may not necessarily be unique. Similarly, kNN search also has its approximate version. Here, let o_1^*, o_2^*, ..., o_k^* be the k exact NNs, sorted in ascending order of their distances to the query point q. The objective of *c-approximate NN query search* is to find a set $S \subseteq D$ of k objects o_1, o_2, ..., o_k (sorted in ascending order of their distances to q) such that o_i ($1 \leq i \leq k$) is a c-approximate version of o_i^*, namely, $\|o_i, q\| \leq c\|o_i^*, q\|$.

A bruteforce scan of D solves the NN problem in $O(dn)$ time. As in range search, to obtain better query time, we must pre-process D into a structure to be searched at run time. Finding a structure that allows sublinear query time for any dimensionality d turned out to be very difficult. Almost all the solutions that work well in low-dimensional spaces are inevitably outperformed by the bruteforce scan when d is large enough. After so many failed attempts, people out of frustration started to refer to the phenomenon as *the curse of dimensionality*. Fortunately, this curse was finally defeated by the discovery of an elegant technique called *locality sensitive hashing* (LSH). Since its debut in 1998, LSH has quickly become so famous that it has been widely applied in many disciplines. We will study LSH in detail in this course.

Other problems. Range and NN search are the main problems we will tackle in this course, but they are not the only ones, not even close. First, we are not ready to discuss how to solve the two problems right away. Some fundamental knowledge is still missing, so we will move forward step by step to pick it up along the way. In particular, we will do it in a mind-inspiring manner. That is, every time something is learned, we will put it to immediate use, by solving some relevant problems. All of these problems, such as *segment intersection*, *interval stabbing*, *3-sided range search*, and so on, are fundamental geometry topics that are frequently encountered in many research areas.

Furthermore, this will not be the only way new problems are introduced. The range-search and NN-search methods to be taught in this course are general approaches whose variations are good for many other problems as well. For example, the LSH technique mentioned earlier can be adapted to find the *closest pair* in a set of points. As another example, the R-tree, which is an all-around index for both range and NN queries, can also be used to perform *skyline search* and *top-k search* very efficiently. All of these problems will be discussed in detail.

2 Memory hierarchy

In complexity analysis, we will follow the standard convention that every (integer or real) value can be stored in $O(1)$ bytes. We consider that our datasets are too large to be kept in the main memory, and thus, need to be stored in the external memory (a.k.a. the disk). Indeed, in many applications, the amount of data is at the order of tera bytes (i.e., $\geq 10^9$ bytes), way larger than the capacity of

a standard computer's main memory.

The key difference between the main and external memories is in the *access unit*. In main memory, the access unit is a byte, but in external memory, it is a *page*, which contains a number B of bytes placed consecutively in the disk where B is the *page size*. More specifically, before a disk can be put into use, it must first be formatted into pages of size B (typical value in practice: 4096 bytes). Every time we read or write to the disk, we are doing so to B bytes at a time. These bytes must be kept in the main memory before the reading or after the writing; so the size of our main memory, denoted by M, must be at least $\Omega(B)$ bytes. Figure 3 illustrates the two-level memory hierarchy described earlier. In the sequel, we may refer to the main memory simply as *memory*; when the disk is referred to, we will always include the word "external".

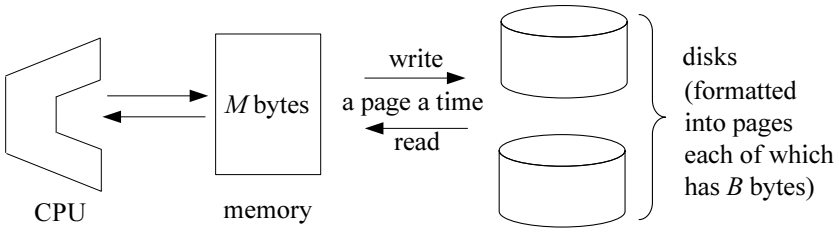

Figure 3: The two-level memory hierarchy assumed

Reading or writing to a page is both counted as an *I/O*. We will measure the cost of an algorithm in the number of *I/Os* performed. In other words, we will not consider the CPU time (spent on calculating, accessing its internal caches, accessing the memory, and so on). This makes sense because in practice the I/O cost significantly dominates the CPU time, at least for all the algorithms we will discuss. This phenomenon is becoming more obvious every day, as the gap between the CPU speed and the disk access time continues to increase.

Some of our existing notions about "what is a good complexity" also need to be adapted accordingly. For example, in memory, we need $\Omega(n)$ space to store n values. In the disk, however, $\Omega(n)$ pages is very expensive for the same purpose. Instead, n values can be accommodated in only $O(n/B)$ pages (which is also the least we need). Likewise, in memory, sorting n values in $O(n \log n)$ time is already the best we can do. In the disk, however, $O(n \log n)$ I/Os is intolerably long for sorting, which can be done in $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/Os (e.g., by using the external sort). By the way, this is also the lower bound for the time of sorting in the external memory.

Do not worry if you do not expect you will need to deal with massive datasets in your own research. External memory is typically *harder* to deal with than internal memory. The techniques to be taught in this course can be applied (in most cases, directly) to datasets that fit in memory as well, with their excellent performance guarantees retained.