An Image-based Positioning System

Ankit Gupta, Rahul Garg, Ryan Kaminsky

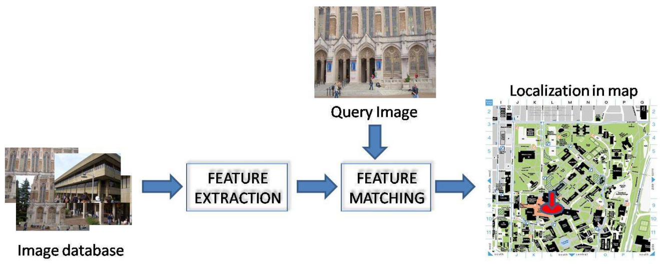

Figure 1: Flow Diagram of the Proposed Approach

Abstract

In this paper we address the problem of building an efficient database management system that allows querying for similar images. This system, which supports image-based queries on a database of photographs taken around the campus, helps the user to identify his/her location on a campus map. We have implemented and analyzed two classes of image retrieval techniques. In the first technique, the database is kept in memory and we study the performance of efficient data structures like kd-trees for approximate nearest neighbor search versus exhaustive linear search. In the second, the database is stored on the disk and the performance of exhaustive linear search is compared with the Local Polar Coordinate (LPC) based Indexed Nearest Neighbor (INN) search algorithm. We also propose novel ideas to further improve the performance of LPC-based INN search. We have built a demo system where a user can upload a University of Washington (UW) campus photo to a web application and receive a map of the campus with his/her position marked.

1 Introduction

Visitors to large areas such as a university campus or city downtown can quickly become disoriented by their surroundings and lose their positioning details. They are only able to rediscover their position from a map if it contains easily identifiable landmarks or position information. Image collections of campus and city locations are prevalent on the Web or easily obtained through manual means. Given such a database of images, we can make a machine learn the landmark descriptions so that a visitor could take an arbitrary picture of the surrounding buildings and use a software system to determine his/her position on a map (Figure 1).

This system has several advantages over a traditional GPS. First, it can be used on any mobile phone with Internet access and phones do not need to be GPS-enabled. Second, this system can work in environments where GPS signals are often blocked such as cities with densely-packed skyscrapers and even indoors. Moreover, our system can also provide location information inside a large indoor structure such as a museum or other tourism site. Finally, our system could be extended to provide more detailed information about a location, including information about buildings or nearby landmarks.

Our goal is to develop a system that will learn identifying features from a large database of images for the purpose of recognizing similar images. Research has shown that pixel colors alone are not sufficient to describe an image – we need to represent the image using higher level descriptors and use SIFT descriptors [11] for this purpose. We have created a database of features extracted from all input images that can be visualized as a set of points distributed in multidimensional space. We can choose to store the database either on the disk or in memory. Each feature point is associated with a location on the map based on the image from which it was extracted. Each unique location is referred to as a class. Thus, given a query image we need to extract its features, find the matching features from the database, and determine the class using a voting mechanism.

The matching problem is called the nearest neighbor (NN) search and is well studied. The issue is to find an efficient NN search algorithm that works well in high dimensional spaces. In addition to linear search (searching for a match by iteratively looking at all features), there exist other methods like approximate nearest neighbor (ANN) search [1] which use hierarchical space subdivisions for faster matching. Many recent papers have realized the curse of dimensionality in large databases and solutions have been proposed ([15],[17],[3],[4]). These methods seek to prune out candidates by operating in a reduced dimensional space and then perform complete matching on the pruned set.

In this paper, we have implemented and analyzed the performance of different approaches to this problem. We propose some novel ideas for the LPC-based INN scheme [4] and discuss their efficacy. We have also built an online system for image-based localization on the UW campus that currently works on a limited dataset.

The organization of the paper is as follows. Section 2 describes the concept of image representation using features and methods of choosing matches for an image feature. Section 3 explains our system architecture and other implementation details. In Section 4, we evaluate the search and storage strategies. In Section 5, we conclude the work with future research topics pertaining to the system.

2 Literature Review

There are two key aspects of the problem: Image representation using features and Nearest Neighbor matching.

2.1 Image representation using features

Feature points form a signature of an image. The detection process needs to be invariant to viewpoint and illumination changes. We use the interest point detector of [12]. Their method adapts the Harris interest point detector proposed in [8] to have scale and affine invariance. The Harris detector is known to detect corner-like and blob-like regions. To incorporate scale invariance into the detector, the notion of characteristic scale is defined, which is indicated by a local extremum over scale of normalized derivatives (the laplacian). The affine shape of a point neighborhood is estimated based on the covariance matrix. An image with detected key points and the affine invariant point neighborhoods is shown in Figure 2.

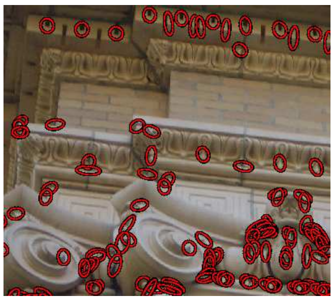

Figure 2: A subset of features detected in the image

Once a point is identified as a key point, we need to build a signature or a feature vector for it. Again, the representation should be invariant to illumination. We use the representation proposed in [11]. The construction of the descriptor is summarized in Figure 3.

2.2 Nearest Neighbor Matching

Given a set of features from a query image and a database of features extracted from the training images, we would like to find the nearest feature corresponding to each query image feature. This is the well studied problem of NN search and the popular approaches can be classified into two broad categories - (1) Exact Approaches and (2) Approximate Approaches. Exact approaches are guaranteed to yield the optimal solution while approximate approaches guarantee bounds on "optimality" of the returned solution. Even though approximate approaches are more suited to the problem we are pursuing, we also study and implement the exact approaches.

Exact Approaches

The naive method of finding the NN is to iterate over every data point, compute its distance from the query point, and return the point with the minimum distance. We refer to this naive solution as linear search. We study another method proposed in [4], the LPC-based

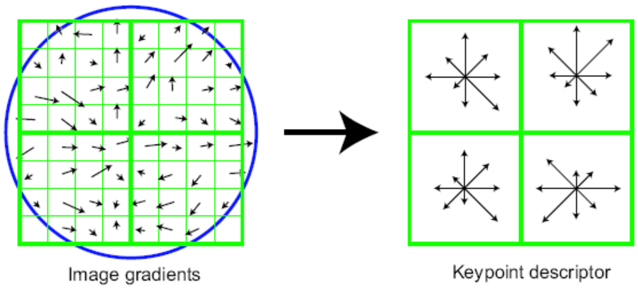

Figure 3: A keypoint descriptor is created by first computing the gradient magnitude and orientation at each image sample point in a region around the keypoint location, as shown on the left. These are weighted by a Gaussian window as shown by the overlaid circle. These samples are then accumulated into orientation histograms summarizing the contents over $4X4$ subregions, as shown on the right, with the length of each arrow corresponding to the sum of the gradient magnitudes near that direction within the region. This figure shows a $2X2$ descriptor array computed from an $8X8$ set of samples, whereas the actual descriptors use $4X4$ descriptors computed from a $16X16$ sample array. The final descriptor is a 128 dimensional vector generated by concatenation of the orientation histograms of each $4X4$ window.

INN search, which does not reduce the time complexity of the search algorithm but seeks to minimize the number of disk I/Os if the database is stored on the disk.

There is another popular class of multidimensional indexing methods (MIMs [6],[3],[7]) which partition the data space and prune the search space for queries. While this approach is quite successful for lower dimensional spaces, its performance degrades rapidly as the dimensionality of the space increases due to what is known as the *curse of dimensionality*. To understand this effect, let us consider the following scenario. Assume that a d-dimensional dataset lies inside a unit dimensional hypercube $[0,1]^d$ and the metric used is the L_2 norm. We also assume that data points and query points are uniformly distributed in the hypercube and the dimensions are independent. Under these assumptions, we can compute the expected nearest neighbor distance for a query. Given this distance, the NN query can be reformulated as a spherical range query and the average cost of the search can be measured by the number of disk blocks that intersect the range sphere. Consider a range sphere query with diameter s in each dimension. The probability that a point lies inside the sphere is given by s^d. It follows that even very large range queries are unlikely to contain a point and highlights the sparsity of the data in high dimensional spaces. For instance, at $d = 256$, a range query with length 0.95 in each dimension selects only 0.0002% of the data points. Hence the expected NN distance becomes much greater than the length of each dimension in higher dimensional spaces. Hence the pruning done in MIM methods in not efficient in high dimensional spaces.

The LPC-based INN strategy builds upon filtering strategies like those in [18]. The filtering approach uses a compact representation of vectors, and by first scanning these approximations, only a small fraction of features are identified as candidates for NN match. The vector approximation(VA)-file approach of [18] divides the whole space into 2^b hypercells where b denotes a user specified number of bits. The VA-file attaches a unique bit string to each cell and the string is used as a compressed representation for each of the data point lying inside the cell. In the filtering step, the bounds d_{min} and d_{max} are calculated for each cell and possible candidates are selected. In the second pass, the actual distance to each of the selected candidates is calculated by looking at their full representation and the nearest neighbor is returned. The selectivity during the filtration stage is controlled by b – the number of bits used in the cell division. Using a larger number of bits implies better filtering but also leads to a larger compact representation and a larger VA-file. Hence there is a trade off between the selectivity of the filter and the size of the VA-file.

The LPC approach proposes to increase the discriminatory power of the filter step discussed above by using the local polar coordinates of the data point within its cell. The information represented by the local polar coordinates is orthogonal to the information contained in the cell approximation and the two local polar coordinates – namely the distance from the corner of the cell and the angle with the cell diagonal are stored alongside the cell bit-string in a LPC file (Figure 4).

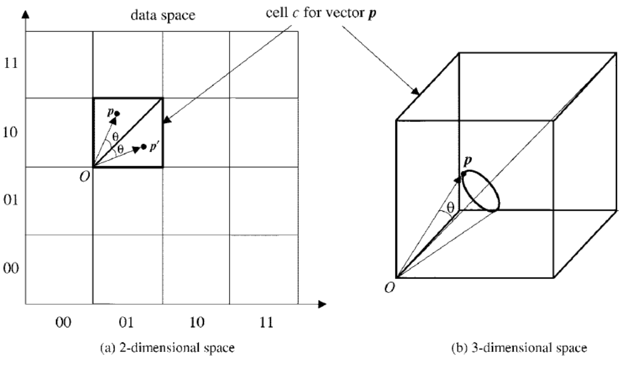

Figure 4: Computing the approximation of vector p in 2 and 3 dimensions

The detailed construction of the approximation a_i for each vector p_i is as follows. A user defined b number of bits are assigned to each dimension to divide the whole data space into 2^{bd} cells. Typically b is a small integer like 4 or 8. Each cell is represented by the concatenation of the binary bit patterns for each dimension in turn. In Figure 4, the cell c is represented by the bit string 01 10 where $d = b = 2$. The second step is to represent the vector p using the polar coordinates (r, θ) within the cell where p lies. Hence, the approximation of the vector p is represented by the triplet $a =< c, r, \theta >$ where c represents the approximation cell. Note that the approximation of the VA-file is only the cell c itself.

The approximation a represents the set of points which have local polar coordinates (r, θ) within the cell c. In 2 dimensions, there are two such points - points p and p' which have the same approximation a. In higher dimensions, this set of points is represented by a hypersphere with its center on the diagonal of the cell and radius $rsin(\theta)$.

Based on the approximation a, we now derive the lower and upper bounds, d_{min} and d_{max}, on the distance from a query point q (Figure 5). If $L_2(p, q)$ denotes the L_2 norm between the query point q and the data point p, we seek to find d_{min} and d_{max} such that $d_{min} <= L_2(p, q) <= d_{max}$. Applying cosine rule in triangle OAB, we obtain

$$AB^2 = OA^2 + OB^2 - 2 * OA * OB * cos(\phi)$$

$$|p - q|^2 = |p|^2 + |q|^2 - 2|p||q|cos(\phi)$$

The lower and upper bound are obtained when the value of $cos(\phi)$ is maximum and minimum respectively. The minimum angle and maximum angles can be seen to be $|\theta_1 - \theta_2|$ and $(\theta_1 + \theta_2)$. Hence d_{min} and d_{max} can be obtained using the equations below

$$d^2_{min} = |p|^2 + |q|^2 - 2|p||q|cos|\theta_1 - \theta_2|$$

$$d^2_{max} = |p|^2 + |q|^2 - 2|p||q|cos(\theta_1 + \theta_2)$$

The bounds obtained using the LPC approximation are much tighter than the bounds obtained using the VA-file. For more discussion see [4]. During the filter step, we maintain an upper bound on the distance of the nearest neighbor from the query point based on the data points processed so far. If the bound d_{min} of a point is greater than the current upper bound, the point is rejected, otherwise it is added to the set of candidates and the upper bound is updated with the value of its d_{max}.

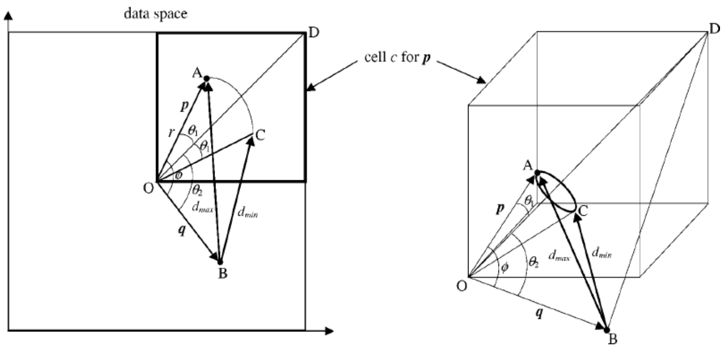

Figure 5: Calculation of d_{min} and d_{max} for $L_2(p, q)$ in 2 and 3 dimensions

Accelerating the filtering step in LPC based INN approach: As discussed above, the filtering is done by calculating the bounds d_{min} and d_{max} for each data point using the compressed LPC representation. The calculation uses the cosine rule and involves complex calculations which are inefficient. We propose a way to derive a coarser bound on d_{min}, which is efficient to calculate. The expensive computation of d_{min} and d_{max} can be avoided if the data point fails the test using the coarser estimate of d_{min}.

The most natural idea to compute a coarser estimate of d_{min} is to calculate the minimum distance of the query point from the cell containing the candidate data point. However, notice that the cell is a hypercube and computing the minimum distance is non-trivial. For instance, a cell in 128 dimensional space will have 2^{128} corners!. However, a coarser estimate can be obtained by considering the hypersphere centered at the center of the cell with radius equal to half the diagonal length – so that it passes through all the vertices of the cell. The distance of the query point from the surface of this hypersphere is a lower bound for the distance of the query point from the cell as shown in Figure 6. This can be calculated efficiently and the performance advantage obtained using this strategy is discussed in Section 4.

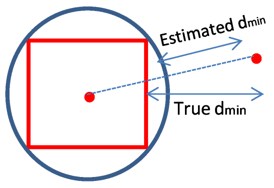

Figure 6: Coarse Estimation of d_{min}

Another interesting point is that the above calculation of d_{min} only depends on the cell in which the data point lies irrespective of the local polar coordinates. This allows us to process data points in batches. All points lying inside a particular cell can be filtered out when a single point fails the test using the coarser estimate of d_{min}. However, this does not lead to any performance gain. Potential causes are discussed in Section 4.

Approximate Approaches

It has been shown by Arya and Mount [1] that if the user is willing to tolerate a small amount of error in the search (returning a point that is not significantly further away from the query point than the true nearest neighbor), then it is possible to achieve significant improvements in running time.

The data in multi-dimensional space is first structured in the form of a kd-tree [5]. This data structure is based on a recursive subdivision of space into disjoint hyper rectangular regions called cells (Figure 7). Each node of the tree is associated with such region B, called a box, and is associated with a set of data points that lie within this box. The root node of the tree is associated with a bounding box that contains all the data points. As long as the number of data points associated with a node is greater than a small quantity, called the bucket size, the box is split into two boxes by an axis-orthogonal hyperplane that intersects this box.

The standard ANN search algorithm [5] proceeds recursively on this kd-tree. When first encountering a node of the kd-tree, the algorithm first visits the child that is closest to the query point. On return, if the box containing the other child lies within $\frac{1}{(1+\epsilon)}$ times the distance to the closest point seen so far, then the other child is visited recursively. The distance between a box and the query point is computed exactly (not approximated), using incremental distance updates, as described by Arya and Mount [2]. This procedure ends at a leaf of this tree. This strategy takes O($nlogn$) time for preprocessing and O($logn$) time for a query, where n is the number of records in database.

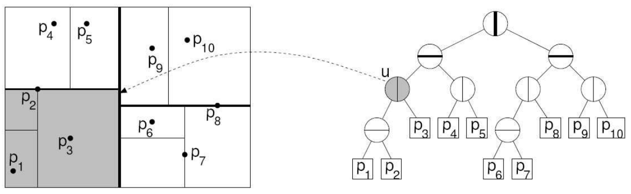

Figure 7: Example of a kd-Tree

Arya and Mount describe priority search, a modified version of ANN search, in [1]. Here, the cells are visited in increasing order of distance from the query point. This is done as follows. Whenever we arrive at a nonleaf node, we compute the distances from the query point to the cells of the two children. We enqueue the further child on a priority queue, sorted by distance, and then visit the closer child recursively. On arriving at a leaf node, we compute the distances to the points stored in this node, and continue by dequeing the next item from the priority queue. The search stops either when the priority queue is empty (meaning that the entire tree has been searched) or when the distance to the nearest cell on the priority queue exceeds the distance to the nearest point seen by a factor of more than $\frac{1}{(1+\epsilon)}$. Ideally this should converge faster than the standard ANN search.

3 System Implementation

3.1 Overview

The problem of creating an image-based positioning system has two separate stages: database creation and query processing. Figure 1 shows the high-level diagram of both stages of our system. The first stage requires the creation of a database of images of the desired locations. An efficient storage mechanism may be implemented at this stage to improve retrieval response.

In the query processing stage, given an input image taken at a location, key features need to be identified and matched with features in the existing database. Next, position information encoded along with the images in the database will then indicate where the input image was taken on a location map. The outline of our approach is as follows:

Stage 1: Database Creation Collect images of the UW campus and manually register them with the campus map. Next, extract feature points from the images and store them in a database. Each feature corresponds to a particular class where each class has a unique location on the campus map. The database may be stored either in memory or on the disk. The implementation must be efficient due to the high-dimensionality of the feature space and the number of features per image, which is likely to be in the thousands.

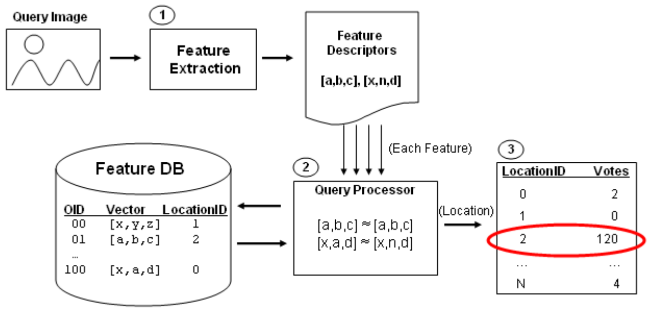

Figure 8: Block Diagram for the Query System

Stage 2: Query Processing (Figure 8)

1. Extract feature points from the given query image.

2. Match the extracted feature points to those stored in the database. Our database supports several different versions of NN search designed to improve efficiency while finding the maximum likelihood match - linear search, LPC-based INN and extensions, and ANN search using kd-tree [13]. Linear search is implemented with both in memory and on disk versions. LPC-based INN search works with the disk version of the database while the ANN strategies work with the memory version.

3. Once we have the nearest neighbors for every feature in the query image, the task is to assign an appropriate class. This is done through a voting mechanism. Each feature in the query image votes for the class in which its two nearest neighbors lie. If the two nearest neighbors are not in the same class, the feature does not vote. We follow a soft voting approach in which the votes are not discrete (1 or 0) but continuous in the interval [0,1]. This is ensured by evaluating a vote as e^{-d} where d is the distance from the nearest neighbor. The intuition behind this voting strategy is to give more weight to features that match well. This approach is found to work better than discrete voting. Based on the percentage of votes, we assign a confidence level to each class and the winning class is used to report the position on the map to the user.

3.2 Technical Details

In designing this system, our goal was to compare many different implementations of NN search, as well as extended versions of previous work. This led us to create a system with "pluggable" search strategies that can easily be interchanged (Figure 9). The same approach was taken to enable the use of an in-memory or an on-disk database. A search strategy has a database(FeatureCollection) that it operates on and the database has an access strategy which it uses to access its data (currently in memory or on disk). Through the use of interfaces, one can easily create a new search strategy or access strategy by implementing the methods of the appropriate interface. Using a new search strategy requires the implementation of the methods Init, which initializes the database and performs the required preprocessing, and MatchFeature, which returns the best match for a single query feature from the database. A new access strategy requires the implementation of GetNext, which acts as an iterator to return the next feature in the database, and GetByID, which returns a specific feature from the database by its id. The main program creates the database with the specified access strategy, the search type with the specified search strategy and runs queries.

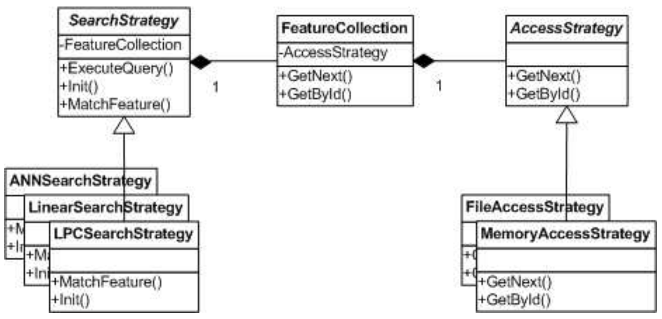

Figure 9: Class Diagram Demonstrating Pluggable Interfaces

3.3 Web Interface

To enable a GPS-like feel for the system, we created a web interface that will be usable by most mobile phones with internet access capabilities. The interface consists of a web page where users can upload an image of a building for the system to identify. The system resides on the server and processes the uploaded image as the query image as discussed in Section 3.1. It then returns the building matches in order of the confidence measure of the match. The web page processes the matching information and highlights the best match on a campus map. Known images of the matching building are also displayed. The user can view all of the buildings along with the confidence measures in this format by using the previous and next links on the web page. Figure 10 displays the system overview including the web interface. Figure 11 shows a screenshot of the web interface.

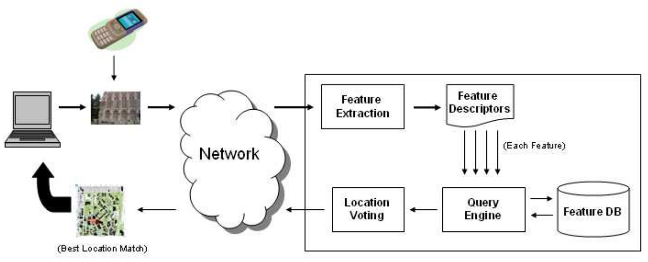

Figure 10: System Architecture including Web Interface

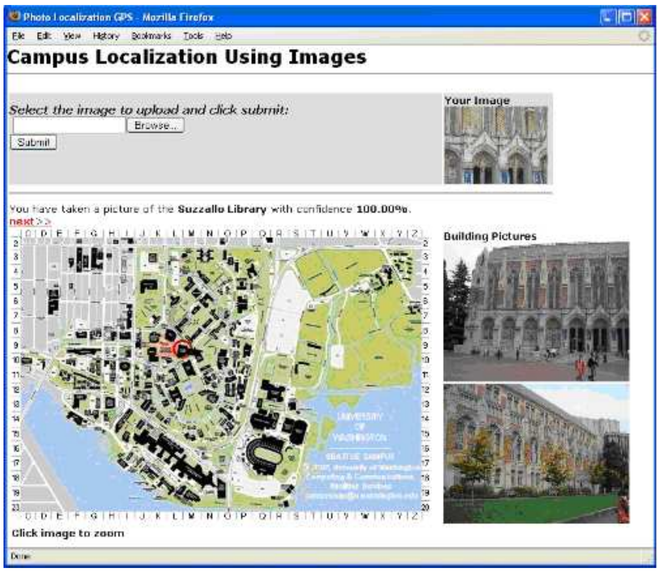
Photo Localization GPS - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Browse...
Submit
Your Image
next >>
Building Pictures
Click image to zoom
Done

Campus Localization Using Images

Select the image to upload and click submit:

You have taken a picture of the **Suzzallo Library** with confidence **100.00%**.

Figure 11: Web Interface for the System

4 Results

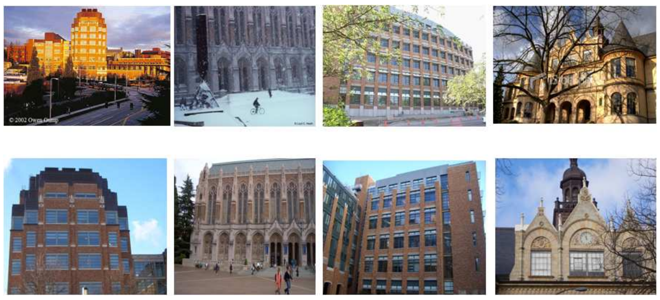

Figure 12: The top row shows query images collected from internet and the bottom row shows a training image for the corresponding class.

We conducted all tests using a database of 66 images (11 buildings, 6 images per building) shot around the University of Washington Campus (Figure 12). All tests were run on an Intel Core2 Duo 2.2 GHz machine with 2GBs of RAM.

On-Disk Storage-Based Strategies

We compare the LPC-based INN strategy with the linear search strategy when the data is stored on the disk. We also test two variants of the LPC strategy – LPC-S which is the LPC strategy with the sphere test included, and LPC-SJ, which uses the sphere test enhancement and skips all data points in a particular cell when a single data point inside that cell fails the sphere test. We give the average query time per feature and average I/O accesses per query feature in Table 1. The results are averaged over 200 queries with LPC strategy using 16 cells(b=4) per dimension.

As expected, LPC-S gives an improvement over the standard LPC. However, it is interesting to analyze why LPC-SJ fares worse. This would happen if the cells are very sparsely populated and the associated overhead of computing which data points to skip negates any advantage gained by skipping over those points. These observations are further corroborated by the statistics for the density of points in cells as shown in Figure 13, which again highlight the sparseness of the high dimensional feature space. The I/O accesses are nearly the same for LPC and LPC-S as expected since LPC-S optimizes over CPU operations. It is unclear why I/O accesses for LPC-SJ increase when they are expected to remain nearly the same.

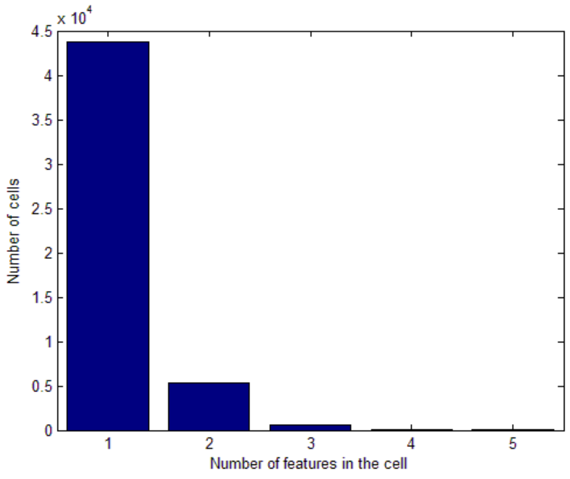

Figure 13: Frequency histogram for number of features per cell for 49787 features. There are no cells with more than 5 features per cell. The total number of cells in the space is 2^{4*128} and non-empty cells occupy only an insignificant proportion of this large set.

We can increase the density of points in a cell by using fewer cells per dimension, but this does not improve performance since it reduces the filtering efficiency of the LPC algorithm. The standard LPC algorithm filters out 97.23% of the features in the first pass on average. The LPC-S variant adds another level of filtering and filters out 50.30% of the candidates using the coarse estimates of d_{min} before moving on to the first pass.

In-Memory Storage-Based Strategies

We analyze the recognition accuracy and average run time for different memory-based storage strategies performing exact and ANN search in the 128 dimensional feature space. We run a set of 50 query images, separate from the training images, on the feature database and collect statistics. These images include photos, both shot around the campus and downloaded from the Internet. The program requires around 150 MB of main memory to run per query.

Table 2 shows the average recognition response times (in seconds) and percentage recognition accuracy. The importance of having a spatial hierarchy in n-dimensions becomes clearly visible when comparing the average response times of the exhaustive linear search with the exact NN search using a kd-tree. The efficacy of the approx-

Search Type	Avg Time Per Query Feature(ms)	Avg I/O Accesses per Query Feature
Linear Search	1703.36	101394207
LPC	265.94	3692842
LPC-S	247.88	3724397
LPC-SJ	320.62	9558080

Table 1: Average query time and I/O accesses of different strategies when the database is stored on the disk

imate matching is supported by a significant decrease in response time without any significant loss of accuracy.

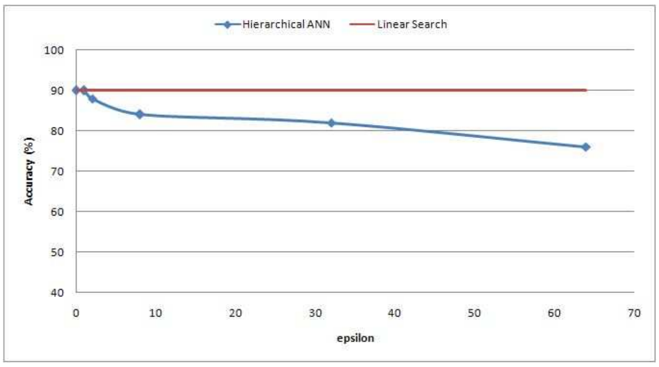

Figure 14: Variation of Recognition Rates of ANN Strategies with ϵ

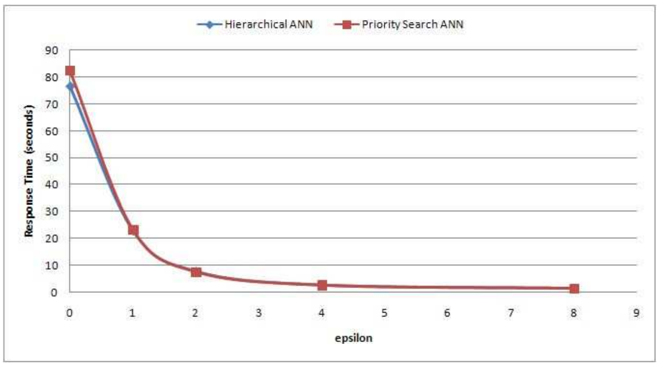

Figure 15: Variation of Response Time of ANN Strategies with ϵ

Figure 14 shows the recognition rates for the standard ANN search using kd-trees as we vary the distance optimality threshold (ϵ). As we allow for more approximation, the accuracy decreases. Figure 15 indicates that the response time for ANN search also decreases but saturates much earlier. Hence, the value of ϵ is a trade-off between the response time and accuracy of the system. The priority search statistics (Figure 15) provide an interesting insight. By definition, the priority search should converge much faster than standard kd-tree search because it looks at the spatial cells in increasing order of distance from the query. However, this is not observed in our experiments, perhaps because our data set in n-dimensions is sparse. This leads to creations of cells of non-uniform sizes containing very few points. Using priority search instead of standard hierarchical space traversal does not give a significant speedup.

It is interesting to note that even though we find the best possible (exact) feature match, we are not able to guarantee the correct answer for the query image. This is reflected in the sub 100% accuracy in exact NN search strategies. This can be explained by the occurrence of common features across different classes. Figure 16 shows some example images that were classified incorrectly. It is observed that in such cases of misclassification, the correct class does appear among the top classes when they are ranked by votes.

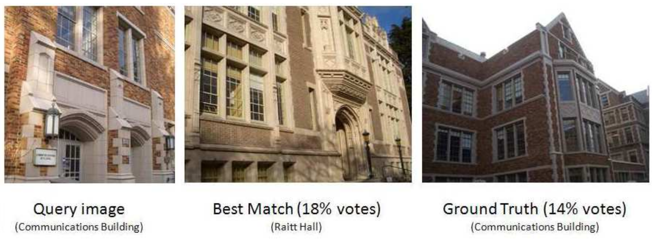

Figure 16: A sample misclassification with confidence values (Note the similarity in structures)

5 Conclusion and Future Work

Future work for our system would focus on two areas: Computer Vision enhancements and Database enhancements. Advancements can be made in both areas to improve both performance and accuracy.

Instead of relying on querying individual images in a database, a collection of images of buildings around campus could be used to create a 3D reconstruction of the scene as is done in [16]. Then, the query image could be registered with each 3D building reconstruction and the most accurate registration would be considered the best match. Additionally, with 3D information geometric constraints are implicitly added to more accurately match the images. Camera parameters from the reconstruction can be used to more accurately determine the actual coordinates of the user. Besides this, we can explore various new image descriptors and ANN algorithms like Locality Sensitive Hashing [9] for enhanced performance.

The large size and dimensionality of the database present a problem for NN queries due to the number of features examined. ANN searches focus on reducing this number while giving an upper bound on the possible error. The size of the database can also be reduced by clustering feature points together and assigning a location class to the entire cluster. During a query, only the feature considered the "center" of the cluster will be

Search Type	Avg Response Time for an Image(seconds)	Accuracy (%)
Exhaustive Linear Search	86.12	90.0
kd-Tree Exact Search	76.76	90.0
kd-Tree Standard ANN Search ($\epsilon = 2$)	7.78	88.0
kd-Tree Priority ANN Search ($\epsilon = 2$)	7.61	88.0

Table 2: Average response time and percentage accuracy for memory-based strategies

examined by the system. This will lead to faster query time as fewer features need to be queried due to clusters. Randomly sampling features from the query image to match or from the database to match against could also be explored as a method to improve response time.

To reduce the dimensionality of the query space we can examine techniques presented in [17]. There are pros and cons of the hard disk-based and memory-based storage strategies. If the features are stored on the disk, an update can be as simple as appending new features to a file. Whereas in memory, it can be as complex as forcing us to rebuild the tree structure. Further, memory-based storage strategies have scalability issues. We would like to design a hybrid scheme which keeps a small part of the large disk database in memory in the form of a tree and will be updated on disk I/Os. This will require tree structures that allow for efficient updates, which will also help updating our database of training images on-the-fly. New data structures like md-trees [14]and G-trees [10] have been proposed which allow for efficient updates. Arya and Mount also propose a new data structure called bd-tree [15] which uses better splitting rules to avoid highly skewed divisions which can occur in kd-trees. Further, our problem involves repeated queries with each query involving a pass over the database. This deteriorates the performance considerably if the database is stored on a physical disk. This can be optimized by updating and evaluating all the queries simultaneously while doing a single pass over the database.

In this paper, we have presented an image-based positioning system using Computer Vision and database query techniques. We compared the accuracy and performance of several exact NN and ANN search algorithms using both in memory and on disk database implementations. We also enhanced the LPC-based INN algorithm resulting in improved query response time. Our system is engineered in a flexible manner allowing the algorithms and database type to easily be interchanged. We also provide a web interface that can be used "in the field" to determine location from a laptop or mobile phone with Internet access.

References

[1] Arya and Mount. Approximate nearest neighbor queries in fixed dimensions. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1993.

[2] Sunil Arya and David M. Mount. Algorithms for fast vector quantization. In J. A. Storer and M. Cohn, editors, *Proceedings DCC'93 (IEEE Data Compression Conference)*, pages 381–390, Snowbird, UT, USA, 1993.

[3] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An index structure for high-dimensional data. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A., 1996. Morgan Kaufmann Publishers.

[4] Guang-Ho Cha, Xiaoming Zhu, Dragutin Petkovic, and Chin-Wan Chung. An efficient indexing method for nearest neighbor searches in high-dimensional image databases. *IEEE Transactions on Multimedia*, 4(1), March 2002.

[5] Jerome H. Freidman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.

[6] C. GH and C. CW. A new indexing scheme for content-based image retrieval, 1998.

[7] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM.

[8] C. Harris and M.J. Stephens. A combined corner and edge detector. In *Alvey88*, pages 147–152, 1988.

[9] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. of 30th STOC*, pages 604–613, 1998.

[10] A. Kumar. G-tree: A new data structure for organizing multidimensional data, 1994.

[11] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[12] K. Mikolajczyk and C. Schmid. Scale and affine invariant interest point detectors. *International Journal of Computer Vision*, 60(1), 2004.

[13] David M. Mount. *ANN Programming Manual*, 2006.

[14] Y. Nakamura, S. Abe, Y. Ohsawa, and M. Sakauchi. Md-tree: A balanced hierarchical data structure for multi-dimensional data with highly efficient dynamic characteristics. In *ICPR88*, pages I: 375–378, 1988.

[15] Y. Ohsawa and M. Sakauchi. The bd-tree–a new n-dimensional data structure with highly efficient dynamic characteristics. In *IFIP Conference*, 1983.

[16] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3d. In *SIGGRAPH Conference Proceedings*, pages 835–846, New York, NY, USA, 2006. ACM Press.

[17] Khanh Vu, Kien A. Hua, Hao Cheng, and Sheau-Dong Lang. A non-linear dimensionality-reduction technique for fast similarity search in large databases. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 527–538, New York, NY, USA, 2006. ACM.

[18] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 194–205, 24–27 1998.