The Data Behind SodaVision

Tom Duerig (tduerig@gmail.com)
Omid Khalili (okhalili@cs.ucsd.edu)
CSE 262, Spring 2006

June 8, 2006

Abstract

SodaVision is a prototype system used in the CSE graduate student lounge soda machine that will use facial recognition to log in users in order to make their purchases and charge their account. Using a mounted camera, the system is trained on users faces, which are added to the repository. Recognition requires detecting a face, morphing the face, running pre-processing on the face, looking up the face in the repository, running an election over many frames, and finally logging in the user with the most votes in the election. The repository of this system is the class project for CSE 262.

We tested three types of lookup methods for our repository. The best repository uses the ANN kd-tree. Building a search tree only happens when learning a new face; therefore, we were concerned with speeding up the lookup time. Building the search tree for the full repository of 6,400 faces with ANN's kd-tree takes approximately 140ms. Doing a single best match lookup takes an average of 7ms , a speedup of 17 over a linear tree!

1 Background

The mere idea of a face recognizing soda-machine requires some explanation, as its purpose and goals are hardly commonplace. Chez bob, in the graduate student lounge, was started some 15 years ago to provide a communal acquisition mechanism for candy, coffee and soda. Money was placed in a jar, and once a week volunteers would go out to replenish the stocks of aforementioned programming fuels. A few years ago that system was upgraded to have a database that fully kept track of the deposits and purchases by every student. Last quarter we brought the system into the 21st century by integrating a thumb print reader for login to the soda-machine. Though the algorithms for thumbprint recognition were professional quality and licensed to us at a trivial discount we were dissatisfied with the system. This quarter the Chez bob development team undertook the latest and most aggressive project to date. Our goal is to have the soda-machine simply recognize who is standing in front of it when a soda is vended or money is inserted to charge or deposit into the appropriate account. And thus was born SodaVision.

With top face recognition researchers advising the initial design we were quickly assured that, even though the project is large in scope, it would be completely feasible. Volunteers were easy to find for the some aspects of the project, but were fairly lacking for what we refer to as the "gray matter" of the system. For this class we tackled the darkest of that gray matter, the actual data handling and repository; without the data handling and fast facial lookups, the system would be useless.

The next section describes our goal for the project. Section 3 describes the SodaVision system including details of the more important data structures. In section 4 we describe the three different methods we used for the repository and section 5 provides our

testing results comparing the different repositories.

2 Our Goal

The initial goal for the repository was to be able to store 128 pictures for at most 512 users and provide a lookup method that takes milliseconds to find the best matching face to a query face. Currently the repository has 128 pictures for 50 people and can do a best match lookup in 7ms, on average. So far we have met our goal; as the database of trained users grows we will see how well our lookup method works.

3 The SodaVision System

The SodaVision computer is an old car computer stored inside the soda machine. This computer has an Intel Celeron 2GHz processor with 512MB of RAM. The algorithms we used were in no way novel; the actual face comparison used simple Principle Component analysis [1]. The detection was done with a modern Viola Jones based face detection algorithm [2]. Both of those components were implemented for us by Intel's Open Computer Vision Library [3][1].

After capturing a frame from a low quality digital camera we first run the face detector over that image and get a single face from it (if there are multiple faces in the frame it selects the largest of those faces). Next, we morph the face in order to have regularized feature locations[2]. Next we do one of two things, depending on whether or not the system is in *learning mode*. If it's learning what a given person looks like, it determines if the current face is different enough, yet similar enough, to the images already stored for that person; if this is the case, then it gets added to the collection of images for that person. If the system is not learning, then it's in *recognition mode*. As we can't afford to store and compare actual images in real time, we store a decomposed list of Principle Component Values (PCVs) for the face. The face is taken, decomposed into its principle component values, resulting in the *eigenFace,* and finally the best match to the eigenFace is looked up.

From the literature we determined that a hybrid approach for determining a best match would be optimal. First we look for the closest face to the one we're trying to recognize. Next, we check to see if the distance of the best matched face and query face is less than a threshold. If it in fact is, we look for the k-nearest neighbors of the query face and make sure they all belong to the same person as the best match. If they do, the person is recognized.

Unfortunately the accuracy for our approach is rather abysmal for single frames without massive pre and post processing [4], so we added *temporal boosting* to take advantage of the number of frames we get for each recognition task. Briefly, each frame in the last $N = 16$ frames votes for who it thinks should be recognized. If a person wins each the majority of votes, that person would be logged in. We represent that election with a nice bar graph that can be seen on the side of our screenshots, figure 1.

3.1 Data Structure and Algorithm Details

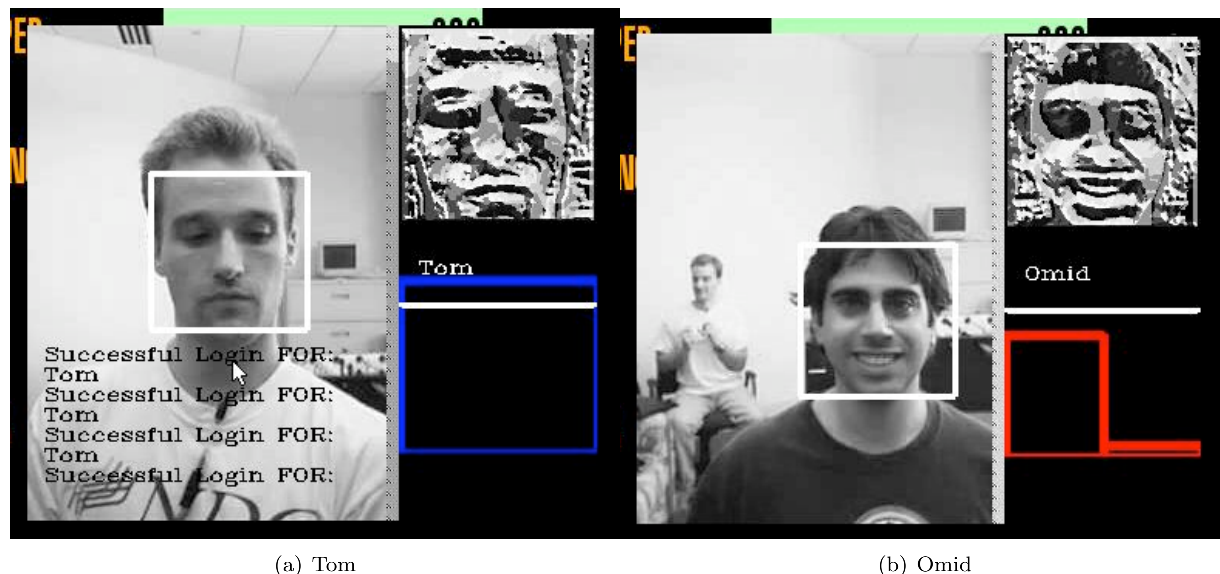
To keep track of the images and floating point arrays that represent the PCVs we created two sets of data-structures. One actually holds the records, loads them, stores them, associates them with a user, and keeps track of user statistics. The other, our part of the project, is used to point into the first in order to allow rapid nearest neighbor and k-nearest neighbor searching.

The first set of data-structures has a `Record` per image that contains an image or a UID depending on whether or not the image was resident, along with the PCV, the eigenFace, associated with the image. Each

[1] `http://www.intel.com/technology/computing/opencv/`

[2] This morphing was supposed to be completed sometime this quarter but was not. We expect it to be completed sometime over the summer. For now, we're just leaving the face non-pose-regulated. This means that all of our results in terms of actual face recognition are of approximately half the quality that they will be in the near future.

Figure 1: Screenshots of Tom and Omid getting recognized. The blue bar in Tom's screenshot shows that his election got the majority, but Omid only gets close.



(a) Tom (b) Omid

user has one `RecordCollection` that is filled with a given number of images (a constant we occasionally tweaked for more robust recognition, currently set to 128 images per person). `RecordCollections` are responsible for the management of a given user's data in its entirety and contain the actual data. The entire program keeps the `RecordCollections` in the `RCS` (a double name for either "Record CollectionS" or "Record Collection Set"). The `RCS` is in charge of responsibly loading images into the repository and rapidly getting a user's data.

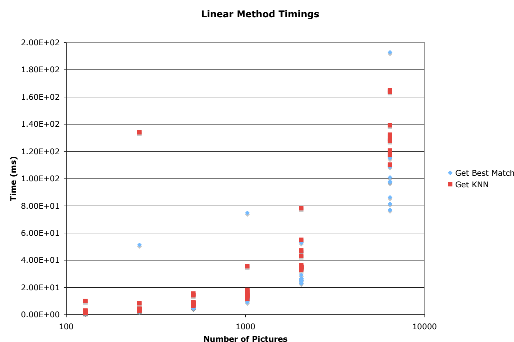
The second set of data-structures are the ones we focused on for this assignment. At the heart of the lookup is the `Repository`, which keeps pointers to all of the learned images and is arranged for rapid lookup. The repository is wrapped in a datastructure called `Space` to insulate changes we made in the repository from the rest of the program, thus making testing different repositories easier. In addition, `Space` does the PCA calculations and calls the post-processing functions. We tried three distinct implementations for the repository which will be described in detail in the next section.

4 Repository Lookup Methods

4.1 Linear

This brute force method was the easiest to implement. We used the STL vector class to store all `Records` in the repository. For a nearest neighbor lookup, we look through all elements in the vector, calculating the sums-squared distance between the query face to each eigenFace in the repository. The record with the minimum distance gets returned. For the k-nearest neighbors, the distance for each eigenFace to the query face is calculated in one pass, then the vector is sorted in order to return the k shortest distances.

Figure 2: Time to get the nearest neighbor (best match) and k-nearest neighbors using the Linear lookup.



4.2 ANN Trees

The other two repositories use one of two search trees provided in ANN [5][3]. In each of the repositories we build a search tree consisting of all the records in the repository. The time to build the search tree depends on the particular tree we use. Next, we can use a search function to search the tree for the best match and k-nearest neighbors.

The first tree structure, the kd-tree, recursively subdivides the total space into disjoint regions called *cells*. Each node in the tree is associated with a *bounding box* and the points within that box. The root node contains a bounding box consisting of the full space. When the number of nodes within a particular box reaches a predefined maximum, the *bucket size*, the box is split into two child nodes. With such a tree structure, when it comes time to search, it becomes unnecessary to look at every point in the space; the bounding boxes give a good estimate of which regions are closer to the query point, so no other regions should be checked.

One drawback of the kd-tree has to do with splitting a region. If points are highly clustered, then when the bucket size is reached, several splits may be required to partition the region. This typically leads to elongated bounding boxes that complicate searching. The box-decomposition tree, or bd-tree (our third repository structure), can be used to fix this problem using a *shrinking* operation. Let a region, B, reach the bucket size and require splitting. The shrinking operation then picks a box, B', axes aligned with B, making sure that B' is not an elongated rectangle. All points in B' are associated with the new box, and all others with the previous one. ANN provides different rules for splitting and shrinking and we used the defaults. Unfortunately, the bd-tree segfaults when the number of records in the repository reaches about 1,500; we are still looking into how to fix this issue.

[3] `http://www.cs.umd.edu/~mount/ANN/`

4.3 X-Tree

The x-tree was advertised as being well suited for high dimension datasets; unfortunately we were unable to implement it. Initially we had to hack their code and `makefile` to get it compile. As the number of records in the repository grew, segfaults due to an integer overflow started showing up. We tried to debug the x-tree library, however German comments and variable names made it futile to continue.

5 Results

5.1 Testing Methodology

We created a 10 second video of Tom and used that to test each repository. This way we could ensure that all our tests used the same scenario. There were approximately 100 timings for getting the best match and 30 timings for getting the k-nearest neighbors. There are less timings for the k-nearest neighbors because some of the best matches don't meet our threshold. We then used a script to collect runtimes for varied sizes of the repository from 128 to 2,048 records, in powers of two, and the full repository of 6,400 records. Because the bd-tree has segfaults when

the repository size grows over 1,500 records, we only present bd-tree timings for up to 1,024 records. The time to build the search tree was also calculated for both the kd and bd-trees for all the repository sizes. For the k-nearest neighbor searches, $k = 3$. All scatter plots show the middle ten measurements for these results. The data in the scatter plots are provided, in tabular form, on page 8.

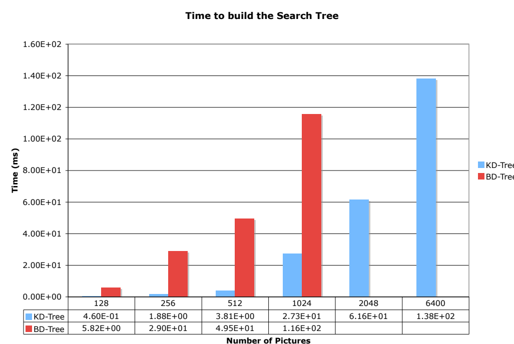
5.2 Linear Method

This approach has the advantage that inserts can be done incrementally and in constant time, however lookups grow linearly based on the size of the repository. Because inserts are trivial, we chose not to time them. Figure 2 shows the linear lookup times for the best match and k-nearest neighbors lookups illustrating how lookup times grow as the size of the repository grows. Another advantage of this approach is that it is easily pipelined in the processor and can easily be parallelized; however at least 10 processors would be needed to reduce the average best match time to something close the kd-tree lookup time.

5.3 ANN Search Trees

Both the kd and bd-trees provide a near constant search time. A side effect is that inserting records into the tree grows almost exponentially as the number of records grow. Furthermore, the bd-tree takes longer to build the search due to its more complicated splitting operation. The build time isn't too important, because we expect to be doing recognizing more the learned, once the system is set. Figures 3 and 4shows the lookup and build tree times for both trees. First, we see that the lookup times for both trees are pretty much constant, although growing a little as the size grows. The kd-tree lookup times average about 7ms for the full repository size. This will grow with a larger repository, but we expect the growth to be minimal, and thus acceptable. For building the search tree with a repository size of 1,024 records, the kd-tree takes 27ms whereas the bd-tree requires 116ms (about four times as long!). This ratio

Figure 4: Time for kd and bd-trees to build the search tree.

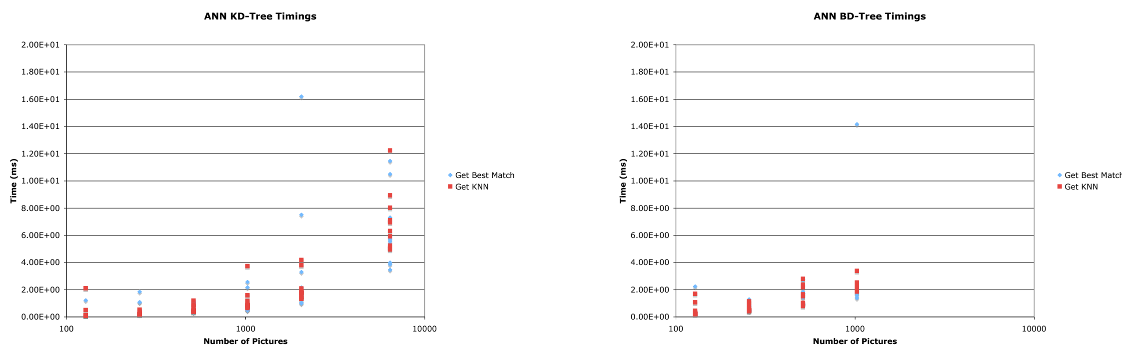


grows as the size of the repository grows, so we can assume that building the bd-tree for the full repository would not be feasible in real time. Therefore, the kd-tree is a good choice for our nearest neighbor search.

5.4 Get Best Match and k-Nearest Neighbors Comparison

To further illustrate the benefits of using a spatial search tree over a linear search, we present figure 5. The difference between the lookup times of the linear search and ANN tree searches is made much clearer here. It is interesting to see that getting the k-nearest neighbors doesn't take much longer than getting only the nearest neighbor for both the kd and bd-trees; however, this doesn't for the linear method. The kd-tree provides an average speedup of 17 and 22 for the best match and k-nearest neighbor lookups, respectively, compared to the linear method on the full sized repository!

Figure 3: Times for kd and bd-trees to get the nearest neighbor (best match), and k-nearest neighbors.

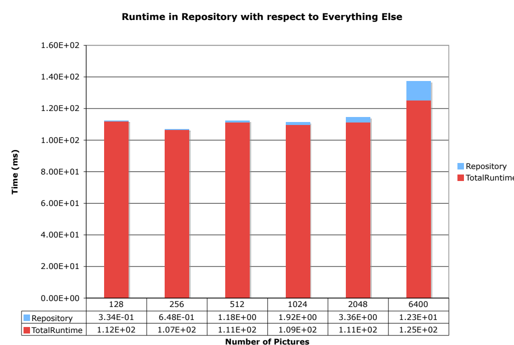


(a) kd-tree best match and k-nearest neighbor lookup times. (b) bd-tree best match and k-nearest neighbor lookup times.

5.5 Time in Repository

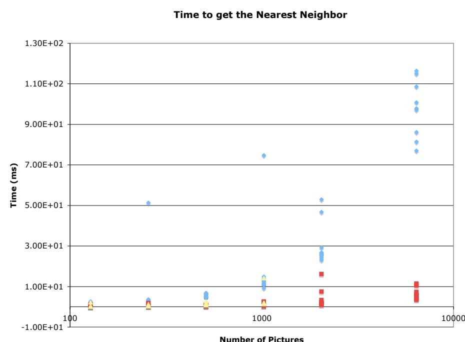
To see what fraction of the time is spent in the repository, we timed both the total time for processing a frame, and the time spent in the repository. This was done for the kd-tree, our preferred method, for varying sizes of the repository. Remember that processing a frame requires detecting a face, morphing the face (which isn't implemented yet), pre-processing the face, looking up the eigenFace in the repository, and voting in the election. Figure 6 illustrates how the average processing time per frame is much larger than the average repository lookup time. We see that for small repository sizes the time to lookup a frame is insignificant, compared to the total time to process the frame and look it up. For the full repository, the lookup is only 1/10-th of the total time! As Ahmdal's law says, we must now focus on speeding up the frame processing time before speeding up the repository times.

Figure 6: Runtime of the total time in the repository (doing a best match and k-nearest neighbors search) stacked on the total time for processing and looking up a frame.

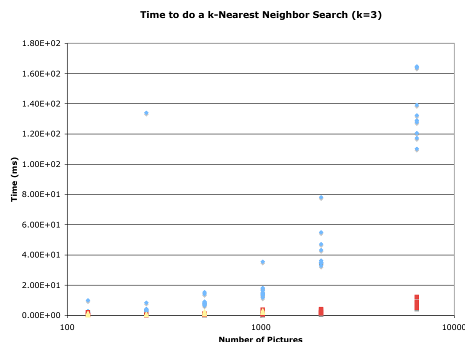


	128	256	512	1024	2048	6400
Repository	3.34E-01	6.48E-01	1.18E+00	1.92E+00	3.36E+00	1.23E+01
TotalRuntime	1.12E+02	1.07E+02	1.11E+02	1.09E+02	1.11E+02	1.25E+02

Figure 5: Time to get the best match and k-nearest neighbors for all three lookup methods. These graphs show the same data in figures 2 and 3 with all three method's best match and k-nearest neighbor runtimes to show how much better the search trees are.



(a) Time to get best match.



(b) Time to get k-nearest neighbors.

6 Conclusions

A SodaVision prototype is working. The kd-tree proved to be the best search tree for our purposes. Future work will include implementing face morphing, and better pre and post processing algorithms.

We worked together to decide on the framework for the repository. Omid wrote the linear lookup code and spent many hours trying to get x-tree to work while Tom implemented the ANN libraries using clever `#define`s to switch between the three repositories. Omid wrote the scripts to collect the performance data and we analyzed the data together. We also made our presentation together. Omid wrote the discussion of the performance data, while Tom wrote the background and description of the SodaVision system.

References

[1] M. A. Turk and A. P. Pentland, "Face recognition using eigenfaces," *IEEE*, 1991.

[2] P. Viola and M. J. Jones, *Robust Real-time Object Detection*. PhD thesis, Cambridge Research Laboratory, 2001.

[3] Intel, "Intel's open source computer vision library," 2006.

[4] W. Zhao, R. Chellappa, P. Phillips, and A. Rosenfield, "Face recognition: A literature survey," 2003.

[5] D. M. Mount and S. Arya, "Ann: A library for approximate nearest neighbor searching," 2005.

Tables of Data

Table 1: ANN bd-tree best match (NN) and k-nearest neighbor (KNN) timings. These are the middle ten of all the recorded timings and all measurements are ms.

Repository Size							
128		256		512		1024	
NN	KNN	NN	KNN	NN	KNN	NN	KNN
0.35	0.30	0.50	0.47	0.92	0.90	2.04	1.91
0.36	0.22	1.29	0.52	1.85	0.95	2.04	1.86
0.25	0.21	0.44	1.03	0.93	1.03	1.73	2.31
0.25	0.23	0.47	0.80	1.62	1.66	1.84	2.27
0.24	0.22	0.46	0.43	1.72	2.79	1.92	2.22
0.24	0.20	0.43	0.65	0.97	1.52	1.56	2.52
0.27	0.45	0.73	1.14	1.72	0.80	1.90	3.38
0.27	1.69	0.48	0.50	1.42	0.99	1.39	2.33
0.28	0.21	0.86	0.47	2.72	2.36	1.89	1.99
2.24	1.07	0.47	0.49	0.89	2.18	14.17	2.10

Table 2: Search tree build times for ANN kd and bd-trees. All times are in ms.

Repository Size	kd-tree	bd-tree
128	0.46	5.82
256	1.88	29.04
512	3.81	49.46
1024	27.27	115.81
2048	61.65	NA
6400	138.22	NA

Table 3: Linear method best match (NN) and k-nearest neighbor (KNN) timings. These are the middle ten of all the recorded timings and all measurements are ms.

Repository Size											
128		256		512		1024		2048		6400	
NN	KNN	NN	KNN	NN	KNN	NN	KNN	NN	KNN	NN	KNN
1.17	1.72	2.60	3.34	6.59	8.78	11.08	35.57	52.96	32.92	192.85	164.78
1.73	1.68	51.37	3.84	4.77	14.19	11.86	17.82	25.16	36.31	77.05	110.23
1.28	1.38	3.48	134.04	5.07	8.52	12.60	11.84	26.17	34.70	86.20	358.71
2.63	1.90	2.46	3.63	6.31	6.65	14.93	14.56	26.78	35.23	81.44	117.37
1.45	1.54	2.41	4.29	5.21	7.39	74.76	14.88	23.10	43.20	100.82	120.58
1.18	1.63	2.39	2.86	6.84	8.17	13.13	14.27	29.17	54.99	116.44	139.16
1.55	1.61	2.77	8.47	6.10	9.19	11.36	18.25	23.98	78.26	108.71	127.77
1.20	2.93	3.18	4.09	4.97	15.43	11.19	13.38	26.44	35.01	114.97	132.29
1.38	10.06	3.67	3.39	6.80	7.99	9.27	13.80	46.74	47.16	97.95	163.86
1.25	2.34	3.55	4.16	4.75	7.89	13.48	16.33	25.48	34.73	97.17	129.09

Table 4: ANN kd-tree best match (NN) and k-nearest neighbor (KNN) timings. These are the middle ten of all the recorded timings and all measurements are ms.

Repository Size											
128		256		512		1024		2048		6400	
NN	KNN	NN	KNN	NN	KNN	NN	KNN	NN	KNN	NN	KNN
0.14	0.12	0.51	0.20	1.10	1.19	0.97	1.59	1.85	1.46	7.30	4.96
0.12	0.10	0.21	0.20	1.06	0.80	0.70	0.87	2.16	1.76	10.50	6.31
0.11	0.09	0.19	0.20	0.36	0.44	0.48	0.77	16.20	3.81	3.47	6.96
0.13	0.10	0.23	0.54	0.39	0.40	1.00	0.87	1.05	4.18	5.39	7.10
0.14	0.09	1.07	0.18	0.37	0.38	0.52	0.83	1.88	1.42	11.46	8.93
0.13	0.09	0.23	0.21	0.40	0.49	0.52	3.73	7.51	1.33	3.85	8.03
0.15	0.10	1.84	0.19	0.40	0.39	2.18	0.75	1.48	2.10	5.71	5.25
1.22	0.11	1.09	0.32	0.75	0.89	0.56	0.65	0.99	1.75	5.56	12.23
0.14	0.50	0.31	0.20	0.97	0.46	0.63	0.84	3.31	1.88	5.91	5.04
0.14	2.11	0.23	0.20	0.37	0.46	2.57	1.17	1.83	1.76	3.99	5.92