FEATURE MATCHING IN GROWING DATABASES

Bernardo Rodrigues Pires[*]

School of Computer Science
Carnegie Mellon University
bpires@cmu.edu

José M. F. Moura

Department of ECE
Carnegie Mellon University
moura@ece.cmu.edu

ABSTRACT

As feature-based image matching is applied to increasing larger scale problems, it becomes necessary to match features across increasingly larger databases. Current approaches are able to conduct such feature matching, but are not flexible enough to be applied to databases that may grow at runtime. As a solution to this problem, we present the Iterative k-d tree that allows for the insertion of new features into the database at any time and stores information about previous queries so that previously searched features can updated without having to be re-run. This new data structure was successfully used in the `Spry` algorithm to achieve better and faster results in situations where there is large movement between images. Additionally, experimental results show that the proposed method is significantly faster than the current state of the art algorithms when the database of features grows at runtime.

Index Terms— Image Registration, Feature Matching, Nearest Neighbor Search, k-d Trees

1. INTRODUCTION

Feature-based image matching is a fundamental step in many computer vision and image processing applications, including scene recognition and detection, 3D modeling and reconstruction, image retrieval, motion tracking, robot localization, image registration, and photo management [1]. Most feature-based state of the art image matching algorithms (such as the SIFT, SURF, Harris-Affine, Hessian-Affine and `Spry` methods) can be divided into three stages, as argued in [1]: the *Detection* step finds features in the input images; the *Description* step computes local descriptors for each feature found; and the *Matching* step matches features between the query and reference images or between the query image and a database of previously extracted features. This paper is concerned with the third step, the *Feature Matching* problem.

Most of the approaches to the feature matching problem consider it as a nearest neighbor search. I.e., when one is trying to match a new, or query, image to a previously analyzed, or reference, image, each descriptor in the query image is matched to the descriptor in the reference image that is closest to it. Typical approaches, such as the original implementation of k-d Trees [2] and the Best Bin First algorithm [3], use balanced k-d trees to solve the nearest neighbor problem. The Fast Library for Approximate Nearest Neighbors (FLANN) [4], implements multiple methods of nearest neighbor search and achieves fast query times. However, all these approaches assume that all features in the reference image(s) have been detected and described before feature matching begins. To the best knowledge of the authors, there is no current approach that is specifically tailored to a situation where the database of reference descriptors may grow.

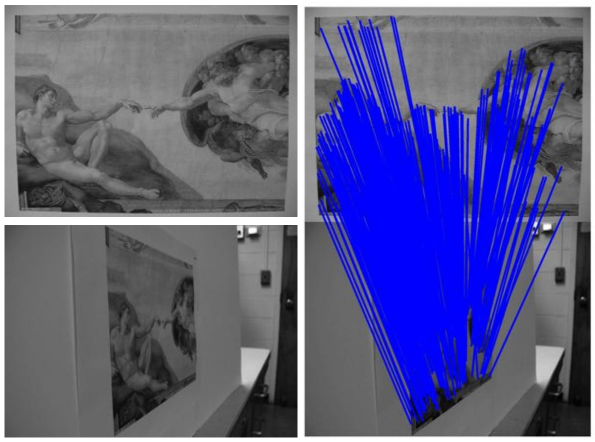

Fig. 1. Output of the `Spry` algorithm [5] on images from Morel and Yu's dataset [6]. The `Spry` algorithm uses feature matching on growing databases to refine the matches in situations where there is a large movement between images.

Nonetheless, there are multiple situations where the database of features grows at runtime, such as when a robot or unmanned aircraft is navigating in an unknown environment, when an algorithm is retrieving images from the web, or when new features are detected based on previous matches, as in the `Spry` algorithm [5] illustrated in Fig. 1. To deal with the problem of establishing matches in a growing database, we introduce a new feature matching algorithm based on the use of an *Iterative k-d Tree*. This new data structure is ideal for applications in which the number of features can increase because it can be easily expanded to incorporate more reference features and stores information about previous queries so that they do not need to be re-run when new reference descriptors are introduced in the tree. Experimental results demonstrate the superiority of the *Iterative k-d Trees* for the task of matching features in growing databases.

2. PROBLEM STATEMENT

In the descriptor matching step, feature matches are created based on the information of the k-dimensional feature descriptors $\boldsymbol{d} \in \mathbb{R}^k$ (with k typically greater than one hundred.) Since, in real applications, there is never an exact match between descriptors, the matching problem can be formulated as the search for the reference descriptor that is closest (is the nearest neighbor) to the query descriptor (where the Euclidean distance is used as a measure of "close-

[*]Bernardo Rodrigues Pires was supported by the Portuguese Fundação para a Ciência e Tecnologia under Fellowship SFRH/BD/37378/2007 and by the Office of Naval Research under Grant # MURI-N000140710747.

Reference	Query
$(0.6, 0.2)$	$(2.1, 1.2)$
$(1.2, 1.8)$	
$(0.4, 1.4)$	
$(1.8, 0.6)$	
$(2.4, 2.1)$	
$(2.2, 0.9)$	
$(1.9, 0.2)$	
$(0.2, 2.8)$	

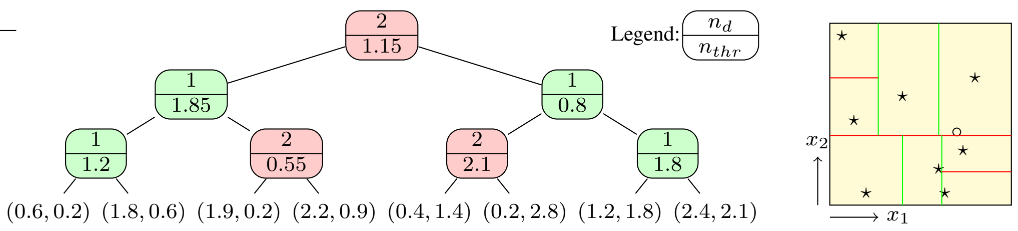

Fig. 2. Illustration of a k-d tree in two dimensional space. Left: Reference and Query descriptors. Middle: k-d tree created using the reference points. Each non-terminal node is denoted by the discriminator on top and the threshold on the bottom. Nodes in green correspond to discriminators in the first dimension of the descriptor x_1 and nodes in red correspond to discriminators in the second dimension of the descriptor x_2. Right: Illustration of the partition of the descriptor space induced by the tree.

ness", or similarity, between descriptors.)

Standard k-d Tree The k-d tree is used to generate a partition of the k-dimensional space. As described in [3], the k-d tree used for feature matching has the following properties:

– All nodes n in the tree correspond to regions in the k-dimensional feature space $\mathcal{R}_n \in \mathbb{R}^k$;
– Each non-terminal node n is associated with a *discriminator* $n_d \in \{1, \ldots, k\}$ and a *threshold* n_{thr};
– Each non-terminal node n partitions the descriptor space along dimension n_d so that:

- All nodes n' in the *left* sub-tree correspond to regions $\mathcal{R}_{n'}$ such that $\forall d \in \mathcal{R}_{n'} : d\,(n_d) \leq n_{thr}$;
- All nodes n' in the *right* sub-tree correspond to regions $\mathcal{R}_{n'}$ such that $\forall d \in \mathcal{R}_{n'} : d\,(n_d) > n_{thr}$;

– Each terminal node (leaf) in the tree corresponds to a region containing a single reference descriptor $\boldsymbol{d}$.

Fig. 2 shows an example of a k-d tree as well as the partition that it induces on the descriptor space. The reference descriptors are represented as stars $\star$, while the query descriptor is represented as a circle $\circ$. Note that the query descriptor is in a different bin from the closest reference descriptors. This problem, which is common to all approaches that partition the descriptor space, means that it is not sufficient to search the bin corresponding to the query image and it is necessary to also search its neighbor bins. As the dimension of the descriptor space grows, the number of neighboring bins that must be searched also grows (the "curse of dimensionality") and the problem quickly becomes very hard.

Construction of the Standard k-d Tree *If all reference descriptors are known before the algorithm starts*, then it is possible to construct a tree that minimizes search time for random queries [2]. To do so we begin with the set of all reference descriptors and iteratively divide the descriptor space along the dimension of greatest variance; i.e., we iteratively set the *discriminator* n_d to be the descriptor dimension that has maximum variance. The *threshold* for each cut n_{thr} is placed on the median of the data so as to ensure that half the descriptors will be divided equally into the left and right sub-trees and that the final tree is balanced. This was the procedure used to create the tree in Fig. 2.

Search in the Standard k-d Tree Before presenting the algorithm for searching on a k-d tree, it is useful to define the distance between the query descriptor $\boldsymbol{d}'$ and a node in the tree. Since each node n defines a region of the space $\mathcal{R}_n$, the distance between the query and the node can be defined as the distance to the point in the region $\mathcal{R}_n$ that is closest to $\boldsymbol{d}'$;

Using this notion of distance, the search algorithm can be informally described as follows [2]: Assuming the algorithm intends to find the m nearest neighbors, it starts with the bin that contains the query descriptor, and then visits the nearest regions to the query descriptor in order of their relative distance. As the search progresses, the algorithm maintains information about the regions it has not visited yet in a priority queue and about the nearest neighbors found so far in the nearest neighbor list. Any region that is further from the current m nearest neighbors can be discarded as it cannot contain a relevant descriptor. When the priority list is empty, it is guaranteed that the m nearest neighbors have been found.

Formally, we have the algorithm (assuming a non-empty tree):

```
1. (Initialize) Let $n_0$ be the current node to be inspected.
   Set $n_0$ to be the $root$;
2. (Descend) Repeat while $n_0$ is a non-terminal node:
   2.1. Let $n_l$ be the left son of $n_0$ and let $n_r$ be the right son of $n_0$;
   2.2. If $\boldsymbol{d}'(n_d) \leq n_{thr}$ set $n_0$ to be $n_l$ and push $(n_r, dist(\boldsymbol{d}', n_r))$
        onto the priority queue;
   2.3. otherwise set $n_0$ to be $n_r$ and push $(n_l, dist(\boldsymbol{d}', n_l))$ onto the
        priority queue;
3. (Store nearest neighbor) Let $\boldsymbol{d}$ be the descriptor inside the current
   node region (each leaf region contains a single descriptor.) Compute
   $dist(\boldsymbol{d}', \boldsymbol{d})$ and store it in nearest neighbor list;
4. (Update priority queue) Remove nodes at a distance greatest than
   the $m$-nearest neighbor from the priority queue;
5. (Backtrack) If the priority queue is not empty:
   5.1. Pop the top node in the priority queue; set $n_0$ to be that node;
   5.1. Return to 2.
```

Best Bin First (BBF) Algorithm As discussed in [3], because of the "curse of dimensionality", the complexity of the search algorithm is dominated by the number of times that it backtracks. For k larger than 10, the standard k-d tree algorithm is not significantly better than exhaustive search [7]. To deal with this limitation and to ensure that the running time is kept to a minimum, Beis et al. propose the Best Bin First (BBF) algorithm [3] that trades accuracy for speed by limiting the number of times that the algorithm backtracks to β Since the algorithm visits the nodes that are closest to the query first, it is reasonable to expect that, with high likelihood, the nearest neighbor will be found after the first few descents.

3. FIXED VERSUS GROWING DESCRIPTOR DATABASES

To model the growth of the database we consider a setup in which features are detected in batches. In a run of the algorithm, we ex-

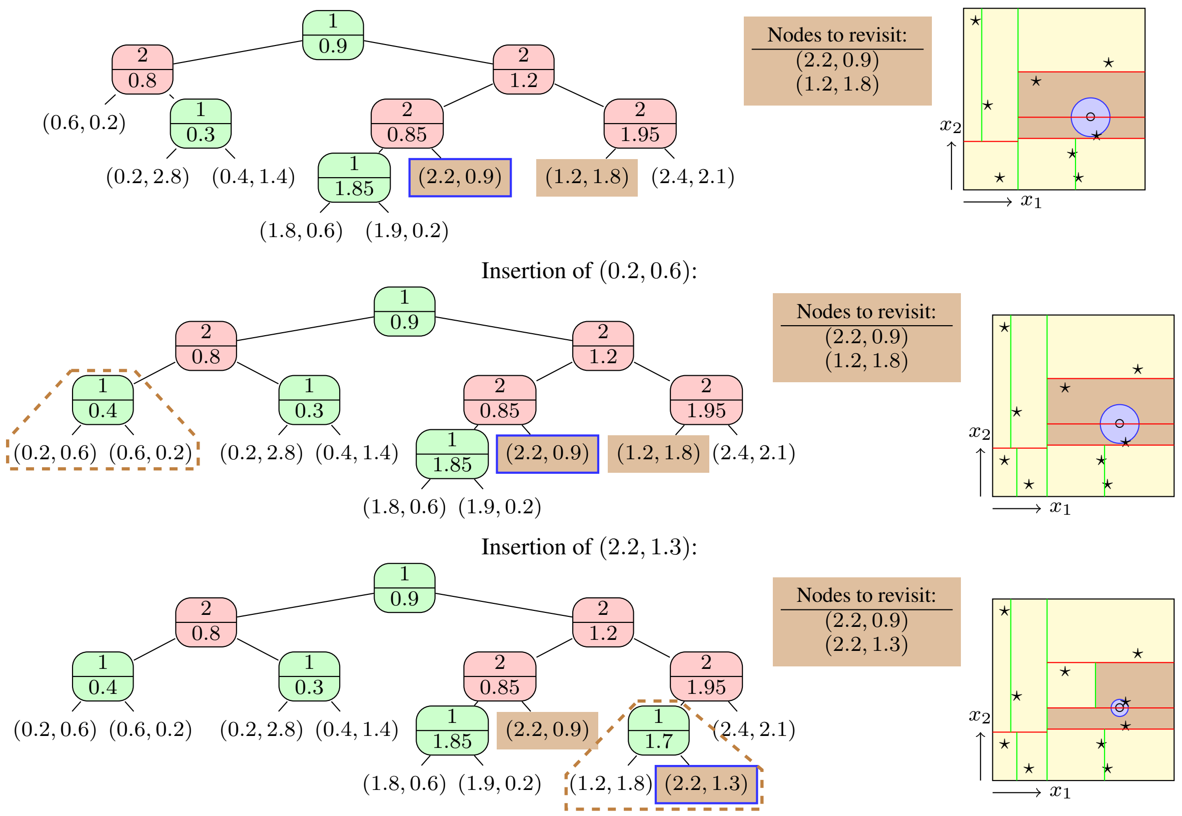

Fig. 3. Search and insertion of nodes on the Iterative k-d Tree. Top: Search algorithm is similar to standard k-d tree (the descriptor $(2.2, 0.9)$ is determined to be the nearest neighbor) with the critical difference that nodes which cross the nearest neighbor hyperball (in blue) are marked to be revisited. Middle: The insertion of the reference descriptor $(0.2, 0.6)$ does not alter the nodes in the list to revisit and therefore it is not necessary to repeat the previous query. Bottom: Insertion of the reference descriptor $(2.2, 1.3)$ alters the structure of one of the nodes on the list to revisit. The search for the nearest neighbor of the query node is resumed *only* on the new node. As a result of the search, the nearest neighbor is determined to be $(2.2, 1.3)$ and the list of nodes to revisit is updated.

tract R batches of N features from each image. So we start by extracting N reference features and N query features. After matching them, we may extract N more reference features and N mode query features for a total of $2N$ reference features and $2N$ query features. We now require an algorithm to compare all these features. It is not sufficient to just compare the newly extracted features since we might want to establish matches between the new and the old features.

When we apply the traditional algorithms (the standard k-d tree and its BBF variant) to this problem, we realize that they are inefficient for two main reasons. First, the construction of the k-d tree requires that all reference descriptors be known beforehand and therefore it is impossible to re-use a previously constructed tree. And second, all previous queries to the tree must be re-run since queries from previous searches may match with some of the new descriptors introduced in the tree.

4. ITERATIVE K-D TREE ALGORITHM

In contrast with the traditional k-d tree algorithm and its BBF variant, we propose the *Iterative k-d Tree*. Instead of attempting to create a balanced k-d Tree, this new algorithm creates the tree randomly so as to ensure fast insertion of new reference descriptors. When a new reference descriptor $\boldsymbol{d}^{new}$ is inserted, it first descends the tree turning left or right at each node n by evaluating whether $\boldsymbol{d}^{new}(n_d) \leq n_{thr}$ or $\boldsymbol{d}^{new}(n_d) > n_{thr}$. When the descriptor to be inserted reaches a leaf, a new node n' is created and a *discriminator* n'_d is assigned randomly. The *threshold* n'_{thr} is assigned as the mean of the values of the new descriptor and the descriptor inside the leaf $\boldsymbol{d}^{old}$ values along dimension n'_d. I.e., we make: $n'_{thr} = {}^1\!/_2 \left(\boldsymbol{d}^{new}(n'_d) + \boldsymbol{d}^{old}(n'_d)\right)$

The initial search on a *Iterative k-d Tree* is similar to the search on the normal k-d tree except that information about previous queries is stored. For each query, the list of nodes to revisit consists of the bins that intersect the nearest neighbor hyperball. Any node outside this list is irrelevant and does not need to be searched again, *even if it changes by the insertion of a new descriptor*. The top of Fig. 3 illustrates the search procedure and the list of nodes to revisit. As in the BBF algorithm, the number of times that the algorithm backtracks is limited to $\bar{\beta}$.

After the initial search, the insertion of more reference descriptors onto the tree proceeds as illustrated in the middle and bottom of Fig. 3. Unlike the traditional k-d trees, the insertion of additional nodes to the tree does not force the previous queries to be re-run. As shown in the middle of Fig. 3, if the inserted reference descriptor does not alter any node in the list to revisit, the previous query is not affected. Moreover, as shown in the bottom of Fig. 3, even if the inserted reference descriptor changes a node in the list to revisit, it is sufficient to consider only the new node created by the insertion, instead of restarting the search.

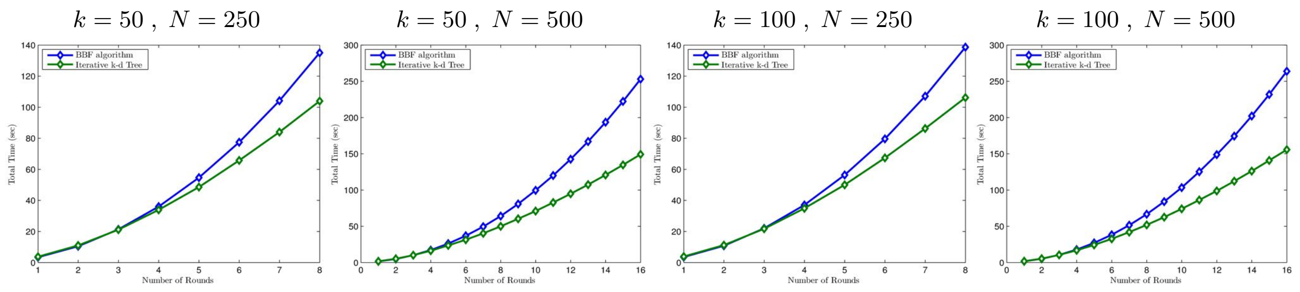

Fig. 4. Running time (in sec.) of the BBF method (blue) and the proposed algorithm (green) as a function of the number of batches R.

5. EXPERIMENTAL RESULTS

Simulated Databases We implemented the BBF algorithm and the proposed Iterative k-d Tree algorithm in Matlab. However, since most of the functions used by both algorithms are the same (including tree descent, backtracking and distance calculations), the same qualitative results would be found regardless of the language used to program the algorithms. In all experiments the reference features follow a uniform distribution $[0,\ 1]^k$ and the query features were simulated by randomly permuting the reference features and adding uniform noise in the range of $[0,\ 0.1]^k$. Additionally, 30% of the query descriptors were replaced by outliers. This setup was chosen to loosely simulate practical applications, but we found that, as long as the descriptors are chosen randomly, the distribution of the feature points has little impact in the relative performance of the nearest neighbor algorithms.

Fig. 4 shows the comparison of the running times of the BBF algorithm and the Iterative k-d Tree. Multiple combinations of parameters were tested with similar qualitative results, however, in the interest of brevity, we present results for $k = \{50, 100\}$ and $N = \{250, 500\}$. Fig. 4 clearly shows that, for a small number of rounds ($R \leq 2$,) both algorithms take a similar amount of time. However, as R grows, the proposed Iterative k-d Tree algorithm quickly becomes much faster.

***Spry* Algorithm** The proposed Iterative k-d Tree was used in the *Spry* algorithm [5] to improve feature matching by using the information in previous matches to generate additional features. Fig. 5 shows a small illustration of the output of SIFT, ASIFT [6], and *Spry* in situations where there is large movement between images. A detailed exposition of *Spry* and a comparison with other state of the art methods is presented in [5] where it is also shown that *Spry* is typically faster and produces more matches than other algorithms.

6. CONCLUSION

We tackled the problem of feature matching in a growing database. We showed that the traditional algorithms (standard k -d trees and its BBF variant) are unable to tackle this problem efficiently. As a solution to this limitation, we proposed the Iterative k -d Tree. This novel data structure allows the insertion of additional reference features at any time and stores information about previous queries so that the nearest neighbors of previously searched descriptors can be updated significantly faster. Our experimental results demonstrated the superiority of the Iterative k -d Trees versus the traditional trees for the problem of feature matching in growing databases.

7. REFERENCES

[1] Richard Szeliski, *Computer Vision: Algorithms and Applications*, Springer, First edition, 2010.

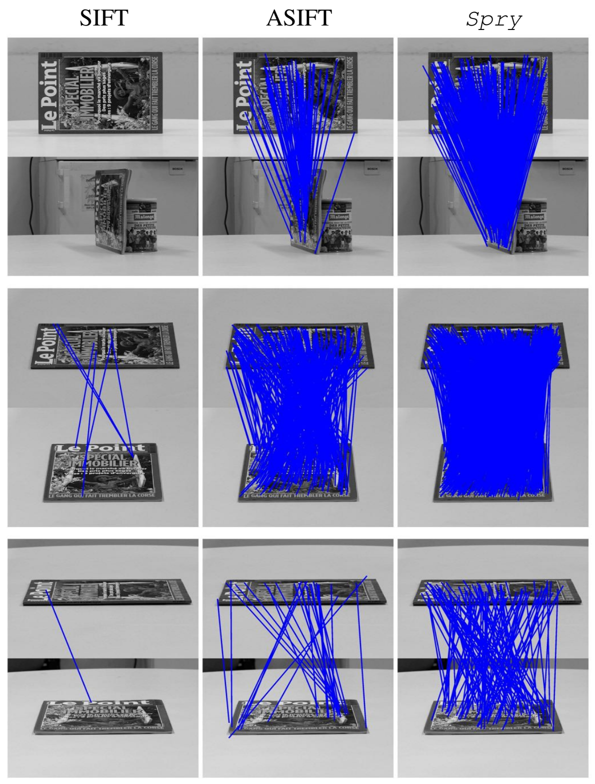

Fig. 5. Comparison of SIFT, ASIFT, and *Spry* [5] on a subset of the images in Morel and Yu's dataset [6].

[2] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM T. Math. Soft.*, vol. 3, pp. 209–226, 1977.

[3] Jeffrey S. Beis and David G. Lowe, "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces," in *Proc. of the CVPR*, 1997, pp. 1000–1006.

[4] Marius Muja and David G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *VISSAPP'09*, 2009, pp. 331–340.

[5] Bernardo Rodrigues Pires and José M. F. Moura, "Spry: Affine-invariant features without global motion constraints," in *Int. Journal of Computer Vision*, 2012, Submitted for publication.

[6] J.M. Morel and G. Yu, "ASIFT: A new framework for fully affine invariant image comparison," *SIAM Journal on Imaging Sciences*, vol. 2, no. 2, pp. 438–469, 2009.

[7] David G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comp. Vis.*, vol. 60, no. 2, pp. 91–110, 2004.