FANNG: Fast Approximate Nearest Neighbour Graphs

Ben Harwood and Tom Drummond
Department of Electrical and Computer Systems Engineering,
Monash University, Victoria, Australia
{ben.harwood,tom.drummond}@monash.edu

Abstract

We present a new method for approximate nearest neighbour search on large datasets of high dimensional feature vectors, such as SIFT or GIST descriptors. Our approach constructs a directed graph that can be efficiently explored for nearest neighbour queries. Each vertex in this graph represents a feature vector from the dataset being searched. The directed edges are computed by exploiting the fact that, for these datasets, the intrinsic dimensionality of the local manifold-like structure formed by the elements of the dataset is significantly lower than the embedding space. We also provide an efficient search algorithm that uses this graph to rapidly find the nearest neighbour to a query with high probability.

We show how the method can be adapted to give a strong guarantee of 100% recall where the query is within a threshold distance of its nearest neighbour. We demonstrate that our method is significantly more efficient than existing state of the art methods. In particular, our GPU implementation can deliver 90% recall for queries on a data set of 1 million SIFT descriptors at a rate of over 1.2 million queries per second on a Titan X. Finally we also demonstrate how our method scales to datasets of 5M and 20M entries.

1. Introduction

Large datasets of feature vectors are a common component of many computer vision tasks such as object and scene recognition[25], pose estimation[27, 21], relocalisation and loop-closing [20], 3D reconstruction[4] and machine learning[26]. Image features such as those described in SIFT[14], SURF[2] and GIST[22] compress local image regions to single points in a high dimensional space that use between 64 and 512 extrinsic dimensions. Calculating correspondences between feature vectors can be achieved by applying a distance function (usually Euclidean) with the assumption that the correct correspondences are more closely located in feature space than incorrect ones. The challenge of finding correspondences in large datasets is computationally demanding and becomes problematic when there is a requirement for real-time responses or when a large number of these queries are required. A naive solution to the feature correspondence problem is to linearly search all features in the dataset and evaluate each one as a possible correspondence for a query feature. Unfortunately this solution is only suitable for trivially small datasets and due to the high dimensionality of feature vectors there is no known algorithm for consistently returning correspondences in a sub-linear time. However for many applications an approximate search [16] can offer a less than perfect recall rate while having a considerably smaller query cost than that of an exhaustive linear search. It is also possible for the degree of this trade-off to be adjusted so it provides an acceptable recall rate for a particular application. For the applications being considered in this paper we will be using the Euclidean distance function to calculate the similarity between feature vectors.

Our approach for finding the approximate nearest neighbours of a dataset involves building a graph where each vertex represents a feature vector from the dataset being searched. In this paper we describe our new graph based method and demonstrate the following key contributions:

- We present algorithms for building graphs and for efficiently searching them during queries (Sections 3.1, 3.4).
- We exploit the limited dimensionality of the local manifold-like structure of a dataset without the need to directly compute that manifold (Section 3.2).
- We present a method that is guaranteed to find the absolute nearest neighbour for all queries within a tunable distance threshold of all values in the initial dataset (Section 3.3).
- We demonstrate that our method achieves much faster average query times for a given recall rate compared to current state-of-the-art algorithms (Section 4.2).

2. Related research

There is a significant literature on algorithms for approximate nearest neighbour search[19], which we divide into two main categories:

2.1. Hashing and quantisation techniques

Hashing techniques[1] and in particular locality-sensitive hashing algorithms[15, 30] are characterised by the construction of multiple hash tables that each map a query vector to a lower dimensional hash code that can then be efficiently compared against the hashes that were generated by the vectors in the dataset. The more hash tables used the more likely it is that one of the hashes of the query vector will end up close to its nearest neighbour in the hash code space. Ultimately memory constraints limit the number of tables that can be used. In general, hashing algorithms are most computationally efficient when there is a relatively small distance between a query and its absolute nearest neighbour. If this distance grows too large, as is common for real valued features, then the computational efficiency of the hash functions will rapidly decrease as the hash codes become separated. This is due to the need for a large linear search in the original vector space to enable matching between the discontinuous hash codes.

Quantisation techniques[9, 6, 10] seek to perform a similar dimension reduction to hashing algorithms, but in a way that better retains information about the relative distances between points in the original vector space. The major advantage of this approach is that both the access of the original dataset and any linear searching of candidate points can be performed in the reduced dimensional space. As such, quantisation techniques have been applied successfully to datasets of up to 1B image descriptors, the size of which would result in current hardware limitations reducing the efficiency of other techniques.

Because these methods avoid distance calculations on the original data vectors, they typically return a set of R candidates that contains the nearest neighbour with some probability (the recall@R criterion). Because of this method of operation, it can be difficult to compare these methods for computational efficiency against tree and graph-based methods in ways other than measuring runtime (which won't account for specific implementations, optimisations and hardware). At least one quantisation method[8] shows gains of roughly a factor of two over FLANN[18] when they include the time cost for comparisons against the shortlist of candidates returned by their method. It can also be seen that the efficiency of these methods rapidly deteriorates when at higher recall. As recall approaches 1.0, the required length of the candidate list approaches the size of the dataset. It is in these areas of operation that methods capable of continuously partition the original data space can be found to be more efficient.

2.2. Tree and graph techniques

Tree structures[13, 12] offer a natural way to continuously partition a dataset into discrete regions at multiple scales. As such, many tree-based structures have been successfully applied to the nearest neighbour search problem. One commonly used method the kd-tree[3, 28] is known perform poorly on high dimensional data, however in the same way that building and applying multiple hash tables improves locality-sensitive hashing, building multiple kd-trees can greatly improve the recall rate of these methods for high dimensional data. Equally comparable to kd-trees in terms of recall rate and search efficiency is the k-means tree[21, 17]. Rather than clustering the data based on its extrinsic dimensions, as is done with kd-trees, the k-means algorithm attempts to group the data based on its intrinsic structure. The major drawback of using the intrinsic properties of the data comes as an additional cost when propagating queries through the k-means trees.

In general the propagation of a query from the top to the bottom of an approximate nearest neighbour tree is computationally efficient. However the average recall rates that are achieve with a single propagation are very low. For this reason, backtracking algorithms are used to increase the recall to a useful range. The need for large amounts of backtracking is an inherent property that is tied to the branching structure of the trees. During the propagation of a query vector, each time a branch is taken the decision is based on a threshold which represents only a small subset of the information needed to explore the search space. Whenever a query is close to a threshold value there is a significant probability that the desired nearest neighbour lies down a different branch than the one being taken. When low dimensional boundaries are used for high dimensional data, choosing an incorrect branch is almost guaranteed. Every time an erroneous path is taken there is no way to correct for the error in the lower layers of the tree, the only solution is to re-traverse the tree many times taking a slightly different route each time.

Nearest neighbour graphs are capable of partitioning the search space in a similar way to tree structures. Algorithms such as a Delaunay triangulation[11] form a graph with a vertex at each data point and edges that connect local neighbours. Delaunay graphs can be explored in a deterministic way that is usually very efficient and can guarantee that the absolute nearest neighbour to a query point will be found. Unfortunately, as the dimensionality of a dataset increases, Delaunay triangulated graphs rapidly reduce in computational efficiency as they very quickly become almost fully connected. K-nearest neighbour graphs[23, 5] provide an approximation of the local neighbourhoods formed in Delaunay graphs. These graphs are able to maintain efficient exploration costs by limiting the degree (number of outgoing edges) of each vertex in the graph. This restriction re-

moves any guarantee of finding the absolute nearest neighbour (returning to the idea of an approximate nearest neighbour search) as well as some of the efficiency of the graph exploration. By placing an artificial limit on the degree of each vertex some of the intrinsic structure, such as variation in density, is inevitably lost.

For large datasets the computational cost of computing the k-nearest neighbours for each vertex is large. Approximate k-nearest neighbour graphs[7, 24, 29] provide an alternative approach to approximating the edges in a Delaunay graph. These and other structures such as small world graphs[16] offer a significant speed-up for the offline building of a graph, but in addition to placing limits on the degree of each vertex, the decentralised construction of the graphs acts to further compromise the desirable structures of Delaunay graphs. This is demonstrated by the need to perform backtracking and multiple simultaneous graph explorations, as done with trees, in order to achieve higher average recall. But, since the graphs avoid using a global hierarchical structure, the costs of backtracking are uniform regardless of if an erroneous path is taken at the beginning or at the end of the exploration.

3. Fast approximate nearest neighbour graphs

The simplest way to use directed graphs for finding the nearest neighbour to an arbitrary query point is to start at some vertex in the graph, test each outgoing edge from that vertex and follow the first edge that gets closer to the query point. This is repeated until all outgoing edges point to vertices that are further away from the query. This is given more formally in Algorithm 1.

3.1. Ideal graph structure

The key innovation of this paper is the design of a graph structure that gives rise to efficient searching. Here, the ideal graph is defined as a minimal graph for which Algorithm 1 always finds the correct solution when the query point matches a vertex of the graph. This guarantee still exists regardless of which vertex of the graph is given as the starting location. The insight that allows a simple graph to be constructed with these properties is that it's only necessary that at each vertex of the graph there is always an edge that leads to a vertex which is closer to the query. If this criteria holds then the graph can be traversed until the query vertex is reached. In other words, it is only necessary that Algorithm 1 be able to make progress and not 'get stuck' at any vertex other than the vertex that matches the query. Because the distance to query always decreases at each step, and there are a finite number of vertices, it *must* converge on the minimum distance of zero. This means that if the graph has an edge from p_1 to p_2, then it is not necessary for it to have an edge from p_1 to any vertex p_3 that is closer to p_2 than p_1. In this case, the edge from p_1 to p_2 *occludes*

Algorithm 1: Naive downhill search

Input: graph vertices P, directed graph edges E, query point Q, search start index v
Output: nearest neighbour index v

```
for each edge E_i with start vertex P_v do
    u ← index of end vertex of E_i
    if distance(Q, P_u) < distance(Q, P_v) then
        v ← u
return v
```

the edge from p_1 to p_3. To make building efficient, we only allow shorter edges to occlude longer ones. This process is illustrated in Figure 1. More formally, given data points $p_i \in \mathbb{R}^n$ and a distance function $d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, occlusion can be defined as:

$$\text{edge}(p_1, p_2) \text{ occludes } \text{edge}(p_1, p_3) \textbf{ if } \\ d(p_1, p_2) < d(p_1, p_3) \textbf{ and } d(p_2, p_3) < d(p_1, p_3) \quad (1)$$

When building these graphs it is sufficient to select edges for each vertex independently. All but the target vertex are then sorted by their distance from it and an edge list is built by considering all vertices in order from nearest to farthest and adding an edge to each vertex that is not occluded by any edges already added. This is given more formally in Algorithm 2.

3.2. Intrinsic dimensionality and vertex degree

Algorithm 2 has the favourable property of creating graphs of relatively low degree (number of outgoing edges per vertex). For SIFT data, the average vertex degree is typically around 25, despite the data living in a 128 dimensional space. By contrast [7] uses strict k-nearest neighbour graphs with degree up to 1000. This limited degree arises because the intrinsic dimensionality of SIFT data is much less than the space in which it is embedded and the occlusion rule in Equation 1 prunes the set of outgoing edges from a vertex so that they efficiently span the local neighbourhood of that vertex. One consequence of the occlusion rule is that the angle between edges must be at least 60^o, and thus the edges have to be well spread out.

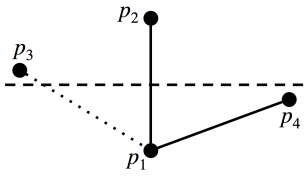

Figure 1. An edge from p_1 to p_2 occludes an edge from p_1 to p_3 because p_3 is closer to p_2 than p_1. The edge to p_4 is not occluded.

Algorithm 2: Naive ideal graph construction

Input: graph vertices P
Output: directed graph edges E

for *each vertex* P_i **do**
  $e \leftarrow$ empty sorted list of edges
  for *each vertex* $P_j \neq P_i$ **do**
    add edge $e(P_i, P_j)$ sorted by distance(P_i, P_j)
  for *each edge* e_j **do**
    $u \leftarrow$ index of end vertex of e_j
    $L \leftarrow$ length of e_j
    $occluded \leftarrow$ **false**
    for *each edge* E_k *with start vertex* P_i **do**
      $v \leftarrow$ index of end vertex of E_k
      if *distance*$(P_u, P_v) < L$ **then**
        $occluded \leftarrow$ **true**
    if not $occluded$ **then**
      add edge e_j to E
return E

The intrinsic dimensionality of a dataset can be estimated using a variant of Hausdorff dimension by computing all pairwise distances between points and counting the number of these that lie below a threshold radius r. If this threshold r is changed, then the number of distances that lie below it should vary as r^D where D is the intrinsic dimensionality of the data at the scale of r. By measuring at two different values of r, the dimensionality D can be estimated as:

$$D(r_1, r_2) = \frac{\log\left(\frac{n(r_1)}{n(r_2)}\right)}{\log\left(\frac{r_1}{r_2}\right)} \quad (2)$$

where $n(r)$ is the number of pairwise distances in the dataset that are less than r. Figure 2 shows this dimensionality calculation for a SIFT dataset containing 1M points.

In order to validate the impact of intrinsic dimensionality on the average degree of graphs constructed with Algorithm 2, several graphs were constructed on randomly sampled data that was selected uniformly from within an n-dimensional hypercube. The average degree of these graphs is plotted against measured intrinsic dimensionality of the hypercube data in Figure 3. As can be seen, the hypercube data suggests that average degree 25 is achieved with a dimensionality of around 11, confirming the observations of the SIFT data.

3.3. Making nearest neighbour guarantees

While Algorithm 1 can be used on graphs built using Algorithm 2 for a arbitrary query point, it is only guaranteed to find the absolute nearest neighbour for queries that are

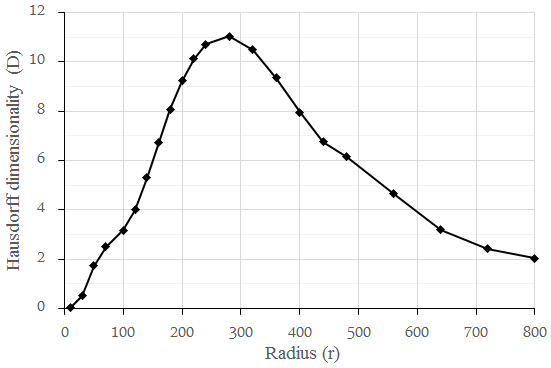

Figure 2. Hausdorff dimensionality of SIFT data measured at varying distance scales. The maximum dimensionality is shown to be approximately 11.

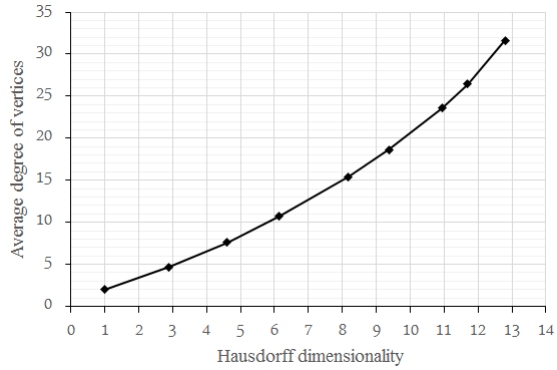

Figure 3. Average out-degree of graph vertices versus dimensionality of uniform hypercube data.

identical to a vertex of the graph. This is of limited use and in practice, it is desirable to have high recall for query points anywhere in the space. We present two methods for achieving this. The first, described in this section modifies the graph building method, while the second (found in Section 3.4) instead modifies the search algorithm.

In some situations, the user is only interested in the nearest neighbour to a query if it is within some distance τ of that query. This can arise when matching descriptors, where there is reason to believe that a true correspondence will have a distance less than some limit τ. In this case, the occlusion function used in graph building can be modified. Where Euclidean distance is used, the modified occlusion function is given by:

$$\begin{aligned} &\text{edge}(p_1, p_2) \text{ occludes edge}(p_1, p_3) \textbf{ if} \\ &\qquad d(p_1, p_2) < d(p_1, p_3) \textbf{ and} \\ &\qquad d(p_2, p_3)^2 < d(p_1, p_3)^2 - 2\tau\, d(p_1, p_2) \end{aligned} \quad (3)$$

This modified occlusion function moves the boundary

between p_1 and p_2 from the halfway point, a distance τ towards p_2. To demonstrate this, consider Figure 4 and the case of equality in the second condition for occlusion. The Pythagorean theorem gives

$$d(p_1,p_3)^2 = (\tfrac{1}{2}d(p_1,p_2)+\tau)^2 + l^2 \quad (4)$$

and

$$d(p_2,p_3)^2 = (\tfrac{1}{2}d(p_1,p_2)-\tau)^2 + l^2 \quad (5)$$

hence

$$d(p_1,p_3)^2 - d(p_2,p_3)^2 = 2\tau\, d(p_1,p_2) \quad (6)$$

This ensures that if an edge from p_1 to p_2 occludes an edge from p_1 to any vertex p_3 where $d(q,p_3) < \tau$ then since $d(q,p_2) < d(q,p_1)$, Algorithm 1 will keep moving until it finds p_3 (or an even nearer neighbour).

In situations where it is acceptable to achieve a guaranteed recall of less than 1.0 the graph can be built using a threshold less than the maximum tolerated distance to a query point τ_{max}. Table 1 shows results for graphs of 100K SIFT descriptors built according to various thresholds as a fraction of τ_{max}, which has been set as the largest distance between a query point and its true nearest neighbour. As can be seen in practice, it is possible to achieve perfect recall on the test set using a lower value for τ than is strictly necessary and as such this recall is obtained at a lower average cost.

It is important to note that building graphs according to this modified occlusion criterion significantly increases the average vertex degree. This has two negative consequences; it increases the space complexity of storing the graph and it increases the search time because there are more edges to be considered at each vertex.

3.4. Fast approximate search using backtracking

A (much) more efficient alternative to the guarantee offered by the build method above is to modify the downhill search algorithm. Rather than terminating when no progress can be made, the algorithm uses a version of depth-first-search to backtrack to the second closest vertex and considers any edges from that vertex that have not been explored yet. If that vertex's edges are exhausted, the third closest vertex is considered and so on.

This is implemented by maintaining a priority queue of vertices whose edges have not yet been fully explored. Exploring an edge requires first computing the distance from the query point to vertex at the end of the directed edge and then placing the vertex in the priority queue according its distance (shortest first). The tradeoff between recall and computational cost is managed by placing a hard limit on the number of distances that will be computed. Once all of this computation is exhausted, the closest observed vertex is returned. This search strategy is detailed in Algorithm 3. Alternative strategies involving combinations of random restarts, edge weighting schemes and potential early termination conditions were explored and found to provide no significant gains to the computational efficiency.

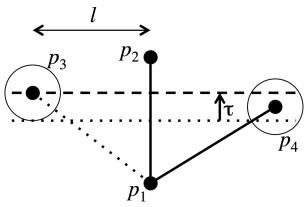

Figure 4. The occlusion boundary has been moved a distance τ towards p_2. p_3 lies on the new occlusion boundary and a circle of radius τ is shown around it. Any query point within this circle (within τ of p_3) must be nearer to p_2 than p_1 and so an edge from p_1 to p_3 is unnecessary. Although p_4 is nearer to p_2 than p_1, it is not occluded because it is possible for a query point within τ of p_4 to be nearer p_1 than p_2. Length l is the orthogonal distance to p_3 from the line joining p_1 and p_2.

τ/τ_{max}	Recall	Avg. cost per query	Avg. degree
0.00	0.7098	110.52	24.329
0.38	0.9984	360.05	127.51
0.50	0.9999	547.80	232.04
0.75	1.0000	1315.5	753.53
1.00	1.0000	3123.2	2182.9

Table 1. Results for graphs built with varying τ as a fraction of the worst case query distance τ_{max}. Cost per query is calculated as the number of distance calculations needed to search the graph on a given query.

3.5. Returning k nearest neighbours

It is often valuable for a nearest neighbour algorithm to return more than just the single nearest neighbour. Algorithm 3 can be easily modified to return an approximation of the k nearest neighbours to a query point by returning a list of the k vertices observed during graph exploration that where nearest to the query point. This can be implemented by maintaining a (possibly truncated) sorted list or heap of vertices visited instead of just the nearest neighbour seen. Efficient implementations (such as our GPU code) can merge this data structure with the priority queue and the *visited* test on line 8 of Algorithm 3.

3.6. Efficient graph construction

The ideal graph construction given in Algorithm 2 has complexity $O(n^2\log(n))$ because it requires sorting a list of n distances for each of n vertices. This complexity makes the construction method prohibitively expensive for building large graphs and so more efficient methods are needed. This section presents two algorithms for constructing approximations to this ideal with significantly lower cost.

Algorithm 3: Backtrack search

Input: graph vertices P, directed graph edges E, query point Q, search start index v, maximum distance calculations M
Output: nearest neighbour index n

1 $X \leftarrow$ empty priority queue `// closest to Q first`
2 add edge e_0 with start vertex P_v to X
3 $m \leftarrow 1$ `// count distance computed to Q`
4 $n \leftarrow v$
5 **while** $m < M$ **do**
6 $e_i \leftarrow$ remove top of X
7 $u \leftarrow$ = index of end vertex of e_i
8 **if** P_u *has not been visited yet* **then**
9 add edge e_0 with start vertex P_u to X
10 $m \leftarrow m + 1$ `// add 1 to compute count`
11 **if** *distance*$(Q, P_u) <$ *distance*(Q, P_n) **then**
12 $n \leftarrow u$
13 $v \leftarrow$ = index of start vertex of e_i
14 **if** $i <$ *number of edges with start vertex* P_v **then**
15 add edge e_{i+1} with start vertex P_v to X
16 **return** n

Algorithm 4: Traverse-add

Input: graph vertices P, directed graph edges E, search start index v_1, search end index v_2
Output: directed graph edges E

1 $u \leftarrow$ NaiveDownhillSearch(P, E, P_{v_2}, v_1)
2 **if** $u \neq v_2$ **then**
3 add edge $e(P_u, P_{v_2})$ to E `// keep E sorted`
4 **for** *each edge* E_i *with start vertex* P_u **do**
5 **if** $e(P_u, P_{v_2})$ *occludes* E_i **then**
6 remove edge E_i from E
7 $v \leftarrow$ index of end vertex of E_i
8 $E \leftarrow$ TraverseAdd(P, E, u, v)
9 $E \leftarrow$ TraverseAdd(P, E, v_2, u) `// test reverse`
10 **return** E

The first method takes in two randomly chosen vertex ids, v_1 and v_2. It then uses naive downhill search (Algorithm 1) to try to get from v_1 to v_2 and if it fails to arrive at v_2, an edge is added from the last vertex visited to v_2. If applied to an empty graph; this method will simply add an edge between v_1 and v_2. Otherwise, there may be some book-keeping where the newly inserted edge occludes a longer edge already in graph, in which case, the longer edge must be removed so that the occlusion rule is maintained. Some gains in efficiency can be achieved by testing that the destinations of the removed edges can still be reached as well as by checking the reverse direction for each pair of test vertices. This method is detailed in Algorithm 4. Our build phase calls this function repeatedly with randomly selected nodes. Typically we repeat until it is achieving a 90% success rate averaged over a sufficiently large number of calls to the naive downhill search function (we used $50N$ calls, where N is the size of the dataset).

When Algorithm 4 is called repeatedly, its progress in improving the graph will eventually slow down. It is at this stage that we switch to a second efficient graph construction method to further improve the graph. The second method uses the current graph to obtain a list of some thousands of approximate nearest neighbours to a vertex using the algorithm described in Section 3.5. Algorithm 3 is given the vertex in question as both the starting point for searching and the query point. This provides a list of neighbours that closely approximate a large set of nearest neighbours of the vertex. The set of outgoing edges for the vertex can then be rebuilt by running through this list applying the occlusion rule in the same manner as in the ideal construction Algorithm 2, lines 5-14. This method can be applied in parallel to each vertex in the graph to quickly build a graph of higher quality.

3.7. Truncating for increased efficiency

A final significant contribution to efficiency is obtained by truncating the edge list of each vertex in the graph to limit the degree to T edges. Although the mean degree for SIFT data is around 25, the maximum vertex degree is often around 300. By truncating to a maximum degree of between 25 and 32 a significant further speedup can be obtained. In practise, the optimal truncation depends on the recall rate demanded from nearest neighbour queries. High recall rates (e.g. 99% or 99.9%) are typically more efficient with slightly higher node degrees, while lower recall rates (e.g. 50% or 90%) are more efficient with lower degrees. Fortunately a graph of higher degree can be dynamically truncated at query time by passing T as an additional parameter to Algorithm 3 and adding a comparison $i < T$ to the test on line 14. This only has a small effect on efficiency (a few percent) and a single truncation value (e.g. 30 for SIFT) works well across a large range of recall values. Using dynamic truncation adds a second tuning parameter to the algorithm (the other being the maximum number of distance calculations M), while making the algorithm generic across all types of graphs.

The very last minor improvement in efficiency comes from a judicious choice of starting vertex. We select the vertex nearest to the mass centre of the data which typically gives a few percent speedup over making a random choice.

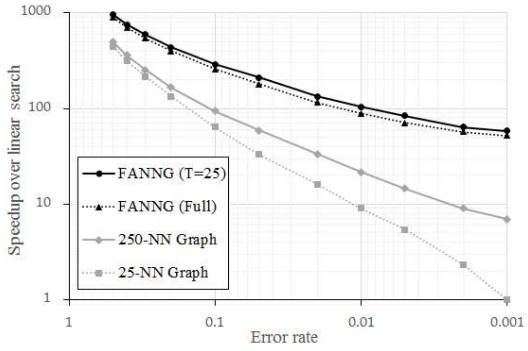

Figure 5. Comparison of k-nearest neighbour graphs with our ideal graph and a truncated ideal graph.

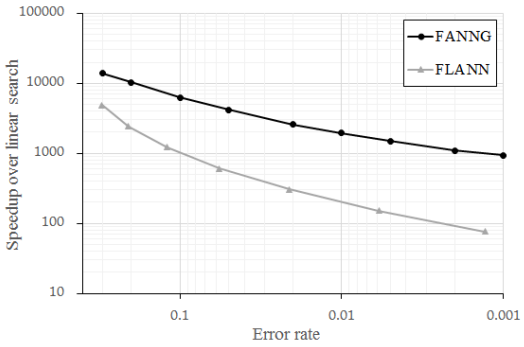

Figure 6. Comparison of FANNG to FLANN on a dataset of 5M SIFT descriptors.

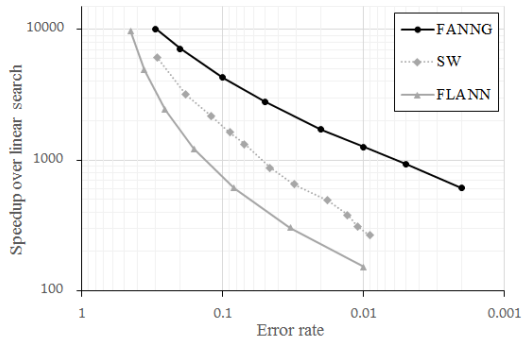

Figure 7. Comparison of FANNG to FLANN and Small World on a dataset of 5M SIFT descriptors using 10-NN overlap.

4. Results

We first present results to justify the design decisions presented in this paper and then compare FANNG to other methods. It is conventional to plot the performance of a method as speedup relative to brute force linear search (measured as the number of distance calculations made) against recall (the fraction of nearest neighbours returned that are correct) on a log-linear scale. These graphs can be hard to read at high recall because when for all methods considered, the speedup drops rapidly as recall approaches 1.0. Here we plot speedup against error rate (1-recall), using a log-log scale. This approach improves the clarity of the relative performance of different algorithms at high recall. We use the BIGANN dataset[8] of 1B SIFT descriptors at various levels of truncation and the set of 1M GIST descriptors for all the comparative results. We measure performance using the provided test set and ground truth files.

4.1. Occlusion pruning vs K-nearest neighbours

Figure 5 compares performance of our ideal graph and its truncation to 25 edges against two plain k-nearest neighbour graphs with all four of them using backtracking as the search algorithm. As the figure shows, k-nearest neighbour graphs are inefficient both in time and space complexity. Keeping the 25 nearest neighbours penalises computational efficiency substantially and even keeping the 250 nearest neighbours is computationally inefficient by comparison to our methods, while also being ten times less memory efficient. Truncating to a maximum of 25 edges per vertex gives us further small gains in time efficiency over the ideal graph as well as improving the memory efficiency.

4.2. Comparison to other methods

Here we compare to two state-of-the-art methods that perform well at high recall, Small World graphs [19] and FLANN [18]. As noted in section 2.1, it is much more difficult to compare to quantisation techniques. We note, however that [8] reports search times on the order of seconds when comparing to FLANN. As described below, our system can obtain query times of 812 ns per query at 90% recall (although they did not utilise a GPU). Because [18] and [19] use different performance measures we present results using both.

4.2.1 Comparison at 1-NN

[18] uses the recall (termed 'precision' in their paper) when a system is asked to return a single nearest neighbour. Figure 6 compares our work to FLANN on the 5M subset of BIGANN.

4.2.2 Comparison at 10-NN

[19] uses the mean overlap between the true 10 nearest neighbours and a system's estimate of the 10 neighbours. Figure 7 compares our work to both FLANN and Small World Graphs on the 5M subset of BIGANN.

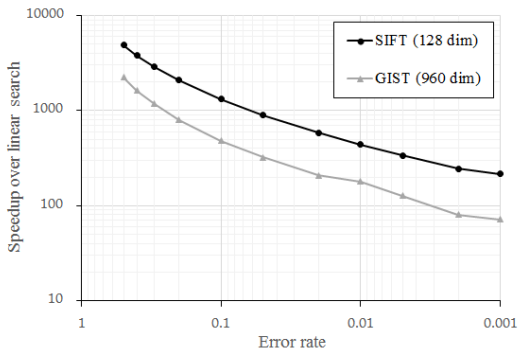

Figure 8. FANNG performance on 1M SIFT and 1M GIST. Searching the SIFT graph is consistently around 3x more efficient than searching the GIST graph.

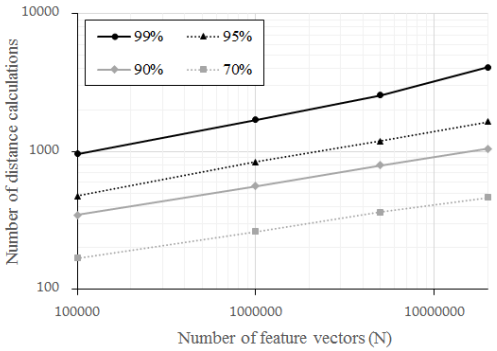

Figure 9. Growth in number of distance calculations against dataset size for fixed recall values.

4.3. Complexity

Here we show how the computational cost (average number of distance calculations per query) of our system changes with the dataset. Figure 8 shows performance on two different types of image descriptors. The higher dimensional data found in the GIST descriptors results in a higher average vertex degree than that of the SIFT graph. Despite the differing graph structures the computational cost of the search remains consistent across a wide range of recall. This suggests that our approach is free from any bias that would favour a particular type of data. Figure 9 shows how the number of calculations needed for various rates of recall changes as the size of the dataset increases. There are only four datapoints for each graph, however, the evidence here suggests that our method scales as roughly the fifth root of graph size, i.e. cost complexity and thus time complexity vary as $O(N^{0.2})$.

4.4. GPU Implementation

We have created specialised GPU implementations for SIFT data for both our method and for linear search to obtain real-world speedup timings that can take into account the book-keeping overhead of running our search algorithm. For both methods, we batch all 10K queries in the standard set into a single kernel call and then divide by 10K to obtain time per query.

Data points	Method	Recall	Time per query (μs)	GFLOPS
1M	linear search	1.0	473.7	861
	FANNG	0.95	1.264	273
	FANNG	0.9	0.812	265
5M	linear search	1.0	2433	789
	FANNG	0.9	1.743	187

Table 2. Timing results for our GPU implementations of linear search and backtracking on FANNG.

As can be seen from the raw GFLOPS results in Table 2, the book-keeping overhead is roughly a factor of 3, i.e. our efficient search method accesses elements and compares distances at $\frac{1}{3}$ of the rate of linear search. Despite this, our method is approximately 500 times faster than linear search at 90% recall on a dataset of size 1M and approximately 1400 times faster on a dataset of size 5M.

5. Conclusions

This paper offers a new approach for finding the approximate nearest neighbours of high dimensional datasets. Our method focuses on building graphs as a structure for efficiently exploring the dataset during a nearest neighbour query. More specifically our method utilises a directed graph structure that is able to minimise the backtracking costs associated with tree structures. Additionally, much of the efficiency of our method comes from our ability to exploit the local intrinsic structures of a dataset without needing to directly compute a manifold that approximates the feature vectors being searched. Our method is able to directly trade-off computation time against recall by choosing a limit on the number of distance comparisons per query. A strengths of our approach to building explorable graphs is the guarantee it offers for finding the absolute nearest neighbour for all queries within a tunable distance threshold of all values in the initial dataset. Lastly when we compared our approach to a number of current state-of-the-art algorithms we found our method capable of achieving faster average query times than any of the other methods.

6. Acknowledgements

This work was supported by the Australian Research Council Centre of Excellence for Robotic Vision (project number CE1401000016).

References

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 459–468. IEEE, 2006. 2

[2] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Speeded-up robust features (surf). *Computer vision and image understanding*, 110(3):346–359, 2008. 1

[3] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 1000–1006. IEEE, 1997. 2

[4] J. Cheng, C. Leng, J. Wu, H. Cui, and H. Lu. Fast and accurate image matching with cascade hashing for 3d reconstruction. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE Computer Society, 2014. 1

[5] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pages 577–586. ACM, 2011. 2

[6] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2946–2953, 2013. 2

[7] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1312, 2011. 3

[8] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(1):117–128, 2011. 2, 7

[9] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: re-rank with source coding. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 861–864. IEEE, 2011. 2

[10] Y. Kalantidis and Y. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2321–2328, 2014. 2

[11] D.-T. Lee and B. J. Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980. 2

[12] H. Lejsek, B. Þ. Jónsson, and L. Amsaleg. Nv-tree: nearest neighbors at the billion scale. In *Proceedings of the 1st ACM International Conference on Multimedia Retrieval*, page 54. ACM, 2011. 2

[13] T. Liu, A. W. Moore, K. Yang, and A. G. Gray. An investigation of practical approximate nearest neighbor algorithms. In *Advances in neural information processing systems*, pages 825–832, 2004. 2

[14] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004. 1

[15] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, pages 950–961. VLDB Endowment, 2007. 2

[16] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014. 1, 3

[17] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP International Conference on Computer Vision Theory and Applications*, 2:331–340, 2009. 2

[18] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36(11):2227–2240, 2014. 2, 7

[19] B. Naidan, L. Boytsov, and E. Nyberg. Permutation search methods are efficient, yet faster search is possible. *Proceedings of the VLDB Endowment*, 8(12):1618–1629, 2015. 2, 7

[20] P. Newman and K. Ho. Slam-loop closing with visually salient features. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 635–642. IEEE, 2005. 1

[21] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. In *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, volume 2, pages 2161–2168. IEEE, 2006. 1, 2

[22] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision*, 42(3):145–175, 2001. 1

[23] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro. Practical construction of k-nearest neighbor graphs in metric spaces. In *Experimental Algorithms*, pages 85–97. Springer, 2006. 2

[24] A. Ponomarenko, N. Avrelin, B. Naidan, and L. Boytsov. Comparative analysis of data structures for approximate nearest neighbor search. *Journal of Mathematical Sciences*, 181(6):782–791, 2012. 3

[25] B. Russell, A. Torralba, C. Liu, R. Fergus, and W. T. Freeman. Object recognition by scene alignment. In *Advances in Neural Information Processing Systems*, pages 1241–1248, 2007. 1

[26] B. Russell, A. Torralba, C. Liu, R. Fergus, and W. T. Freeman. Object recognition by scene alignment. In *Advances in Neural Information Processing Systems*, pages 1241–1248, 2007. 1

[27] G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 750–757. IEEE, 2003. 1

[28] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *Computer Vision and Pattern*

Recognition, 2008. CVPR 2008. IEEE Conference on, pages 1–8. IEEE, 2008. 2

[29] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. Scalable k-nn graph construction for visual descriptors. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 1106–1113. IEEE, 2012. 3

[30] X. Zhang, Z. Li, L. Zhang, W.-Y. Ma, and H.-Y. Shum. Efficient indexing for large scale visual search. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 1103–1110. IEEE, 2009. 2