FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Accelerating Nearest-Neighbours via KD-Trees on GPUs

Author:

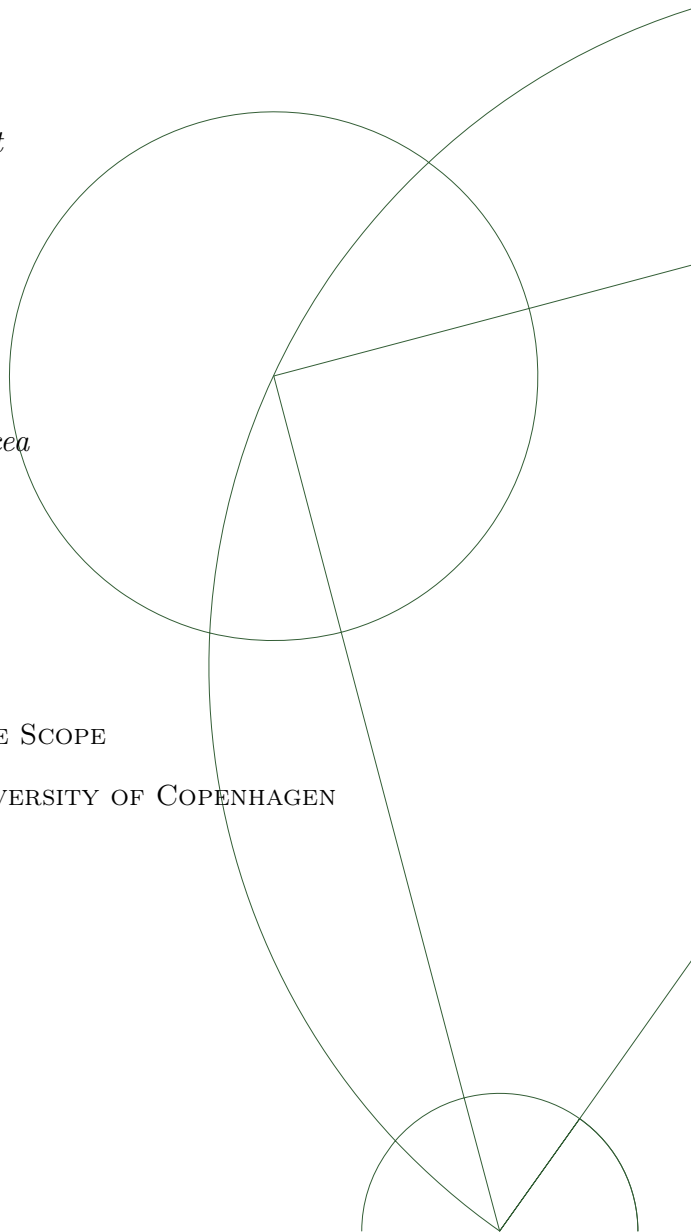
Sarah Maria Hyatt

zfj900@alumni.ku.dk

Supervisor:

Cosmin Eugen Oancea

cosmin.oancea@di.ku.dk



Project Outside Course Scope

Department of Computer Science, University of Copenhagen

April 28, 2020

Contents

1 Introduction

The original direction of this project was inspired by (He & Sun, 2012) who propose a solution for efficiently computing approximate nearest-neighbours (ANN) computations using propagation-assisted k-d trees. The performance bottleneck is the search for K-nearest neighbours (KNN) to improve accuracy, and while various other methods such as the k-means tree and the bbd-tree improve accuracy over the k-d tree, they spend extra effort in building the trees, e.g. the k-means tree uses 10-20 more time (He & Sun, 2012). However, we found that

in order to implement such algorithms we would still need all the components of the exact K-Nearest Neighbour (KNN) computations by k-d trees as we would for the approximate solution, thus, for this project, it was commonly agreed to concentrate on the exact KNN computation solution.

KNN is the core of many applications in various fields such as Computer Vision and Machine Learning, for instance, finding similarities between two images as the original direction proposed.

The naive approach of computing KNN is using brute force, and while brute force has been empirically observed to be efficient than the k-d tree traversal when dealing with a small number of reference points, such as in the hundreds, it is infeasible time-wise for larger datasets due to its complexity $O(n \times m)$.

The idea of KNN by k-d trees is to construct a spatial data-structure, which is a binary tree where each internal node semantically corresponds to a subset of the reference points that can be reached through them, such that each query does not have to be compared with each reference point, as it would in the naive approach, keeping the number of comparisons asymptotically cheaper than the O(n*m). In our approach, the leaves contain a set of points and we apply brute force at the leaf level due to because it has been empirically observed that brute force is more efficient than k-d tree traversal when the number of reference points is small such as in the hundred for GPU computation.

In order to compute the KNN for a query, the tree is traversed in which each visit to each node determines whether it is necessary to go into a recursion to visit the next node or to skip it. Typically we reason about whether to visit the next node by computing the distance to the worst nearest-neighbour with the distance from the query to the box that bounds that region represented by that internal node. The approach typically compares the distance between the query to the current median against the distance to the worst nearest-neighbour. However, we improve this test to be more accurate by considering the upper and lower bounds of all dimensions corresponding to that.

In principal this possibly reduces the asymptotic work to a factor of $O(n\ lg\ n)$, but in practice, the efficiency of the solution depends fundamentally on the dimensionality and the size of K. While this offers a cheaper traversal it still suffers from the curse of high dimensionality. Thus, the trade-off is accuracy for lower dimensionality and reduced algorithmic complexity. In practice there are two main problems that may affect the performance of the

solution in asymptotic terms (1) dimensionality, if it is too high, then you might have to compare against most of the reference points (2) and similarly K if K is too high and the distance to the worst neighbour becomes big then similarly you are going to visit more leaves.

One of the advantages of the k-d tree, compared to various other similar structures, such as octal trees - is that is can in principal conduct NN computations over high dimensionality spaces. Octal trees are limited to three dimensions.

To address the curse of high dimensionality, we present a more accurate test that, in several cases, allows to efficiently process queries of higher dimensionality than the classical approach. While the typical solution computes the distance between the query to the current median against the distance to the worst nearest-neighbour, this solution considers the upper and lower bounds of all dimensions. Which, in return, improves temporal locality by sorting the leaves by accessing the same reference points in the same order. See more in section **ADD REFERENCE**.

Additional optimisation, such as improving temporal locality, have been done by sorting the queries such that they access the same reference points in the same order whenever we do brute force of the leaves.

Summarise the conclusions at a higher level.

for this dimensionality we obtain this amazing speedups and for small k's we increase the k's for a small dimensionality and this is roughly how the speed ups is xxx. (OBS the summary should back my previous defined claims)

The summarise the results : here we want to validate the points that we have just made : does sorting improve things or not? and more insight like - give people a magical pill so they can understand things - even if it is not that precise. for example, if i'm having a uniformly distributed dataset of 16 points then if i'm choosing k it's reasonable to compute this knn like by 5, over 5 I see the performance reduces drastically.

- what are the downsides of the solution - significant overhead and uses contains much more data - so that is why experiments are important to show which ds and ks run best. if you visit most of the leaves then you are going to be much much slower, due to the significant overhead. That's why experimental evaluation is important because it shows on what d's and k's it is beneficial - does it make sense to run the k-d tree this way.

1.1 Acknowledgement

2 Background

Computing KNN is widely applied due to its simplicity and excellent empirical performance, it has no training phase, it can handle binary as well as multiclass data, and while often applied for comparing two images, it faces the challenge of slow performance because both images are of image size.

When comparing two images; image A and image B, the goal of KNN is to find the patches of image B that are most similar to each query patch of image A, according to a measured distance. The naive approach is using brute force, in which each patch of image B is compared with each patch of image A, if each image has m patches the complexity of the search will be $O(m^2)$. See Brute Force. Although brute force is a simple solution, there exists several alternative techniques for computing KNN. One commonly applied method is using a generalisation of the binary tree; the k-d tree.

The k-d tree consists of a root, nodes and leaves; the root represents all patches, the nodes represent a partitioned segment of the patches, and the leaves collectively contain all patches spread out on 2^{h+1} leaves where h is the height of the tree excluding leaves.

Using a k-d tree has advances w.r.t search performance, thanks to the binary tree structure it yields a complexity of $O(m \ \lg \ m)$ (Friedman, Bentley, & Finkel, 1977).

Algorithms 1, 2 and 3 below represent a recursive implementation of the k-d tree construction and the tree traversal as high-level pseudo-code.

Cosmin: should I mention that the pseudo-code is based on the Python code and the authors?

Algorithm 1 is the `main` function in which dimensionality of images A and B are reduced using PCA[1], a k-d tree is created of image B, and all patches of image A are iterated through, of which a call to `TRAVERSE` yields the best neighbours for each patch in A.

[1]Given a collection of points in two or higher dimensional space a principal component analysis (PCA) can be created by first choosing the line that minimises the average squared distance from a point to the line, resulting in an Eigenvector, second choosing the line perpendicular to the first, resulting in a new Eigenvector, and repeating the process will result in orthogonal basis vectors. These vectors are called principal components, and several related procedures principal component analysis (PCA).

Algorithm 1 Main

```
1: procedure MAIN(k)
2:     imA ← PCA(imageA)
3:     imB ← PCA(imageB)
4:     tree ← BUILDTREE(imB, imB.1, 0, 0, NONE)
5:     neighbours ← NONE
6: foreach query in imA do:
7:     neighbours ← TRAVERSE(tree, query, 0, neighbours, k)
8: return neighbours
```

Algorithm 2 represents the construction of the k-d tree. The tree construction is divided into two statements; working with the nodes (lines 4-9) or working with the leaves (lines 20-21). This condition is measured by computing the height of the tree and checking the current level against the height.

The patches are partitioned by finding the dimension with the widest spread and choosing the median value of that dimension. The nodes, therefore, contain a dimension and a median value. Line 5 uses the function GetSplitDimension to attain the dimension, lines 6-7 sort the indices and patches w.r.t the chosen dimension and lines 9-10 pick out the median index and value. Once done, the median and the dimension are ready to be stored, which essentially creates that particular node of the tree. Lines 13-19 compute the left and right indices for the next iteration and recursively call the BuildTree function again, in order to complete the creation of nodes in the tree.

Last, line 21 sorts the leaves w.r.t the indices that were sorted and partitioned throughout the recursive node creation.

Algorithm 2 Building the Tree

```
1:  procedure BuildTree(patches, indices, depth, index, tree)
2:      maxDepth ← ComputeMaxDepth(patches)
3:
4:      if depth < maxDepth-1 then
5:          dim ← GetSplitDimension(patches)
6:          indices ← SortByDim(indices, dim)
7:          points ← patches[indices]
8:
9:          medianIdx ← GetMedianIdx(indices)
10:         median ← GetMedian(points)
11:
12:         tree ← (median, dim)
13:         depth ← depth + 1
14:
15:         leftIdx ← AddToLeft(index)
16:         rightIdx ← AddToRight(index)
17:
18:         BuildTree(patches[: medianIdx], indices, depth, leftIdx, tree)
19:         BuildTree(patches[medianIdx :], indices, depth, rightIdx, tree)
20:     else
21:         leaves[index] ← SortLeavesByIndices(patches, indices)
```

The traversal is presented in Algorithm 3 below. It deals with three different stages; reaching a leaf, which is the first and second node to visit and a check to visit the second leaf. The first stage on lines 2-4, reaching a leaf, is straightforward. We reach a leaf; therefore, we perform brute force with the query patch and the patches of image B in that current leaf.

Algorithm 3 The Tree Traversal

```
1:  procedure TRAVERSE(tree, query, nodeIndex, neighbours, k)
2:      if IsLeaf(nodeIndex) then
3:          neighbours ← BRUTEFORCE(tree, query, nodeIndex, neighbours, k)
4:          return neighbours
5:
6:      dimens ← tree[nodeIndex].1
7:      median ← tree[nodeIndex].0
8:      queryV ← query[dimens]
9:
10:     if queryV <= median then
11:         first ← GOLEFT(nodeIndex)
12:         second ← GORIGHT(nodeIndex)
13:     else
14:         first ← GORIGHT(nodeIndex)
15:         second ← GOLEFT(nodeIndex)
16:
17:     neighbours ← TRAVERSE(tree, query, first, neighbours, k)
18:     worstNeighbour ← LAST OF neighbours
19:
20:     if (median - queryV) < worstNeighbour then
21:         neighbours ← TRAVERSE(tree, query, second, neighbours, k)
22: return neighbours
```

The third stage, lines 18-21, determine whether we should visit the second node by checking if the difference between the median and the query is less than the worst nearest neighbour. If this is the case; the second node will be visited in a recursive call of Traverse on line 21. The check is an estimate to assure that it makes sense to continue the traversal, because if the worst nearest neighbours is a better result in XXXX.

Argument the logic behind this.

The second stage on lines 6-17 determines which following nodes are the first and second,

respectively. The first is always the node that is on the same side of the current node's median as the query. We might think of this as going left first on lines 10-12 and right first on lines 13-15. The dimension and median of the current node is extracted from the tree, lines 6-7, and the query value based on the current dimension of that node, line 8. Line 17 executes a recursive call of Traverse in which the first node is the next node to be visited.

The goal is to implement the solution using Futhark since Futhark is a pure functional data-parallel array language, meaning it is syntactically and conceptually similar to other functional languages; however, Futhark can compile code to run on the CPU, or it can optimise and parallelise the code to run on the GPU. It is a great advantage to write fully parallel code in a high-level functional language. The downside is that Futhark has its limits; such as only supporting regular arrays and not supporting recursion. With this in mind, the pseudo-code, demonstrated above, will need to be rewritten without recursion. See more in the sections k-d Tree Construction and Tree Traversal.

2.1 Related Work

2.1.1 PatchMatch

PatchMatch uses a randomised algorithm for quickly finding approximate nearest neighbour fields (ANNF) between image patches.

While previous solutions use KD-Trees with dimensionality reduction, PatchMatch instead searches through a 2D space of possible patch offset. The initial step is choosing random patches followed by 4-5 iterations containing two processes: (1) propagation applying a statistic that can be used to examine the relation between two signals or data sets, known as coherence, (2) random search in the concentric neighbourhood to seek better matches by multiple scales of random offsets. The argument is that one random choice when assigning a patch is non-optimal, however applying a large field of random assignments is likely a good choice because it populates a larger domain.

The gains are particularly performance enabling interactive image editing and sparse memory usage, resulting in runtime O(mM log M) and memory usage O(M), where m is the number of pixels and M the number of patches. The downside is that PatchMatch depends on similar images because it searches in nearby regions in few iterations, which in turn also affects its

accuracy.

2.1.2 Coherency Sensitive Hashing

Coherency Sensitive Hashing (CSH) is inspired by the techniques of Locality Sensitive Hashing (LSH) and PatchMatch. CSH uses a likewise hashing scheme of LSH and is built on two overall stages, indexing and searching.

The indexing stage applies 2D Walsh-Hadamard kernels in which the projections of each patch in image A and B are initially computed onto the WH kernels. Subsequently the hash tables are created by the following. First, it takes a random line defined by a patch and divides it into bins of a constant width while shifting the division by a random offset. Second, the patch is projected onto the most significant 2D Walsh-Hadamard kernels. Last, a hash value is assigned, being the index of the bin it falls into.

Applying hash tables has the benefit that similar patches are likely to be hashed into the same entry.

The searching stage starts by initialising an arbitrary candidate map of ANNF. This is followed by iterations through each patch in image A where: (1) the nearest neighbour candidates of image B are found using the current ANNF and the current hash table, (2) the current ANNF mapping is updated with approximate distances.

Selecting candidates follows the appearance-based techniques of LSH and as well as the coherence-based techniques of PatchMatch. Choosing among the candidates is done in an approximate manner where the WH kernels rejection scheme for pattern matching is applied beneficially.

CSH has ‘46%‘ better performance than PatchMatch and a higher level of accuracy with error rates that are ‘22%‘ lower in comparison

3 Implementation

3.1 The Brute Force Version

-
-

-
-
-
-
-
-

```
entry nnk [m] [n] (imA : [m][n]real)
                  (imB : [m][n]real) : [m][k](int,real) =
    map (\a_patch ->
        if a_patch[0] == real_inf
        then replicate k (-2i32, real_inf)
        else
        let  nn = replicate k (-1i32, real_inf)
        in loop nn for q < m do
            let b_patch = imB[q]
            let dist = euclidean a_patch b_patch
            let b_idx = q in
            let (_, _, nn') =
                loop (dist, b_idx, nn) for i < k do
                    let cur_nn = nn[i].1  in
                    if dist <= cur_nn then
                        let tmp_ind = nn[i].0
                        let nn[i] = (b_idx, dist)
                        let b_idx = tmp_ind
                        let dist  = cur_nn
                        in  (dist, b_idx, nn)
                    else    (dist, b_idx, nn)
            in  nn'
    ) imA
```

Listing 1: Futhark implementation of the Brute Force.

3.2 k-d Tree Construction

-

-
-
-
-
-
-
-

1

Listing 2: Futhark implementation of the tree creation.

3.3 Tree Traversal

- using an integer to represent the stack
- the first traversal logic (show figure)
- the continuous traversal logic (show figure)
- checking a median on one dimension
- checking all medians from all dimensions (show equation)
-
-
-

As mention in the Background, Futhark does not support recursion, which is why the pseudo-code from Algorithm 3 is rewritten into an imperative version. One such solution is using a stack to traverse the tree incrementally while deciding which nodes to visit next. The figure below demonstrates the traverse down to the first leaf; this example does not need a stack because, at this point, we do not know if we need to visit additional leaves. Figures X-X show the traversal continuing from the first leaf, in which a stack is necessary.

```
                    00000000000000000000100010000101
or                  00000000000000000000001000000000
                    --------------------------------
                    00000000000000000000101010000101
```

Fig. 1: Example of stack integer arithmetic.

3.3.1 Representing the Stack as an Integer

The naive solution of representing a stack is using a boolean list where the leaves and node are set to true once visited. Although this is a simple and correct solution, since the height of the tree never exceeds 32, it can be further optimised as an integer representation instead. The code in listing XX shows the bit arithmetic done to modify the right areas of the stack and the logic behind setVisited is demonstrated in XX.

```
1  let setVisited (stk: i32) (c: i32) : i32 =
2      stk | (1 << c)
3  let resetVisit (stk: i32) (c: i32) : i32 =
4      stk & !(1 << c)
5  let isVisited (stk: i32) (c: i32) : bool =
6      (stk & (1 << c)) > 0i32
```

Listing 3: Snippet of bit arithmetic for stack modifications.

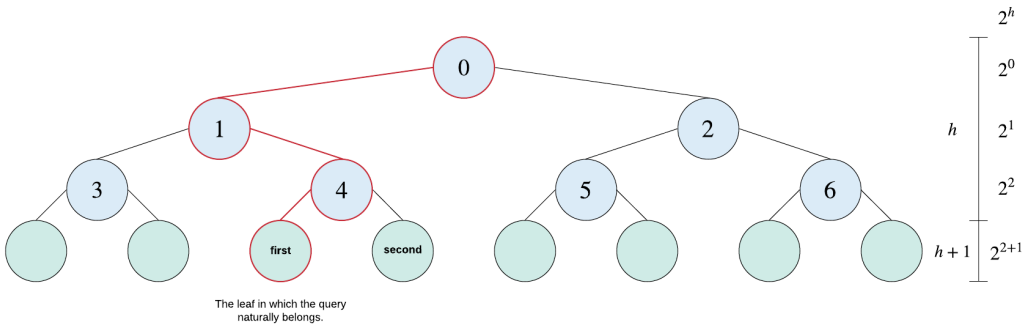


Fig. 2

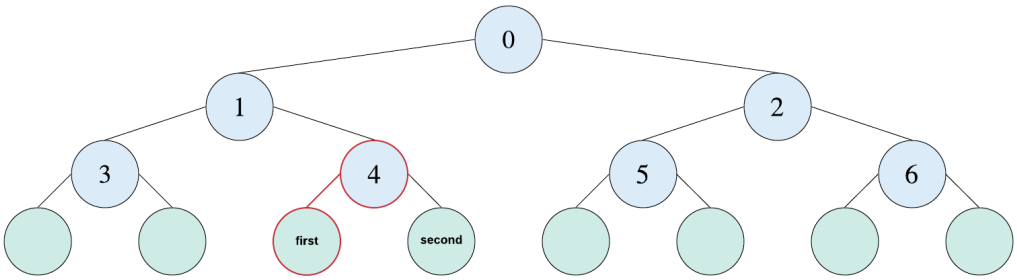


Fig. 3

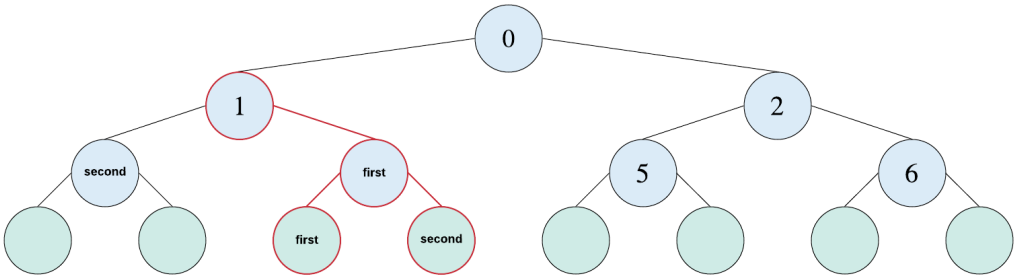


Fig. 4

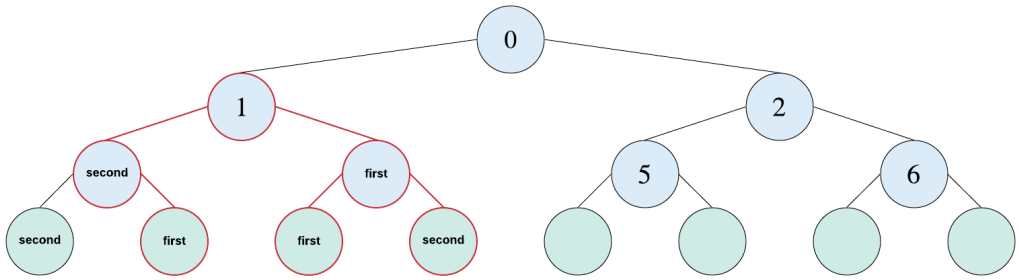

0
1
2
second
first
5
6
second
first
first
second

Fig. 5

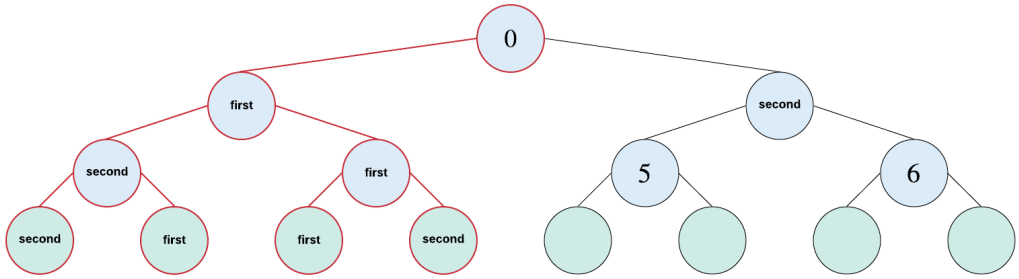

0
first
second
second
first
5
6
second
first
first
second

Fig. 6

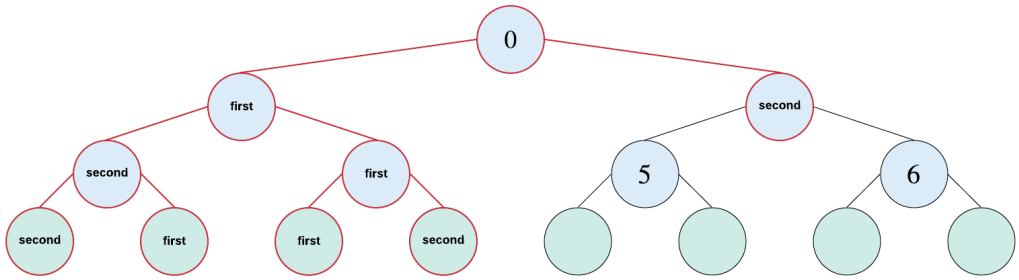


Fig. 7

3.3.2 Validating Whether to Look at the `second`

The original solution determines whether to visit the second node by XXX.

```
entry traverse [d][n][l] (height:             i32)  (median_dims:     [n]i32)
                         (median_vals:     [n]f32)  (wknn:               f32)
                         (query:           [d]f32)  (stack:              i32)
                         (last_leaf:          i32)  (lower_bounds: [l][d]f32)
                         (upper_bounds: [l][d]f32)  : (i32, i32) =

  let setVisited (stk: i32) (c: i32) : i32 =
      stk | (1 << c)
  let resetVisit (stk: i32) (c: i32) : i32 =
      stk & !(1 << c)
  let isVisited (stk: i32) (c: i32) : bool =
      (stk & (1 << c)) > 0i32

  let (parent_rec, stack, count, rec_node) =
      loop (node_index, stack, count, rec_node) =
           (last_leaf, stack, height, -1)
            while (node_index != 0) && (rec_node < 0) do
                let parent = getParent node_index
                let second = node_index + addToSecond node_index in

                if isVisited stack count
                then (parent, stack, count-1, -1)
                else
                  let ack =
                    loop ack = 0.0f32
                      for i < d do
                          let cur_q = query[i]
                          let lower = lower_bounds[second,i]
                          let upper = upper_bounds[second,i] in

                          if cur_q <= lower then
                              let res = (cur_q-lower)*(cur_q-lower)
                              in (ack + res)
                          else if cur_q >= upper then
                              let res = (cur_q-upper)*(cur_q-upper)
                              in (ack + res)
                          else (ack + 0.0)

                  let to_visit = (f32.sqrt ack) < wknn in
                  let to_visit = f32.abs(median_vals[parent] - query[median_dims[parent]]) < wknn in
                  if !to_visit
                  then (parent, stack, count-1, -1)
                  else
                    let second = node_index + addToSecond node_index
                    let stack  = setVisited stack count in
                    (parent, stack, count, second)

  let (new_leaf, stack, _) =
      if parent_rec == 0 && rec_node == -1
```

3.4 The Full Implementation

At each step you reason about tradeoffs/alternative design choices and justify why you did it in the way you did it (advantages/shortcommings, and why your approach is reasonable). Of course mostly related to performance.

Whenever possible, support your reasoning with (as in point to) experimental evaluation results (next).

4 Experimental Evaluation

- Plot with: partition vs. sorting
- Plot with: traverse vs. traverse all dimensions
- Plot with: sorting leaves vs. not sorting leaves
- Plot with: brute force with the best kd-tree
-
-
-
-

4.1 Optimisation using Sorting over Partition

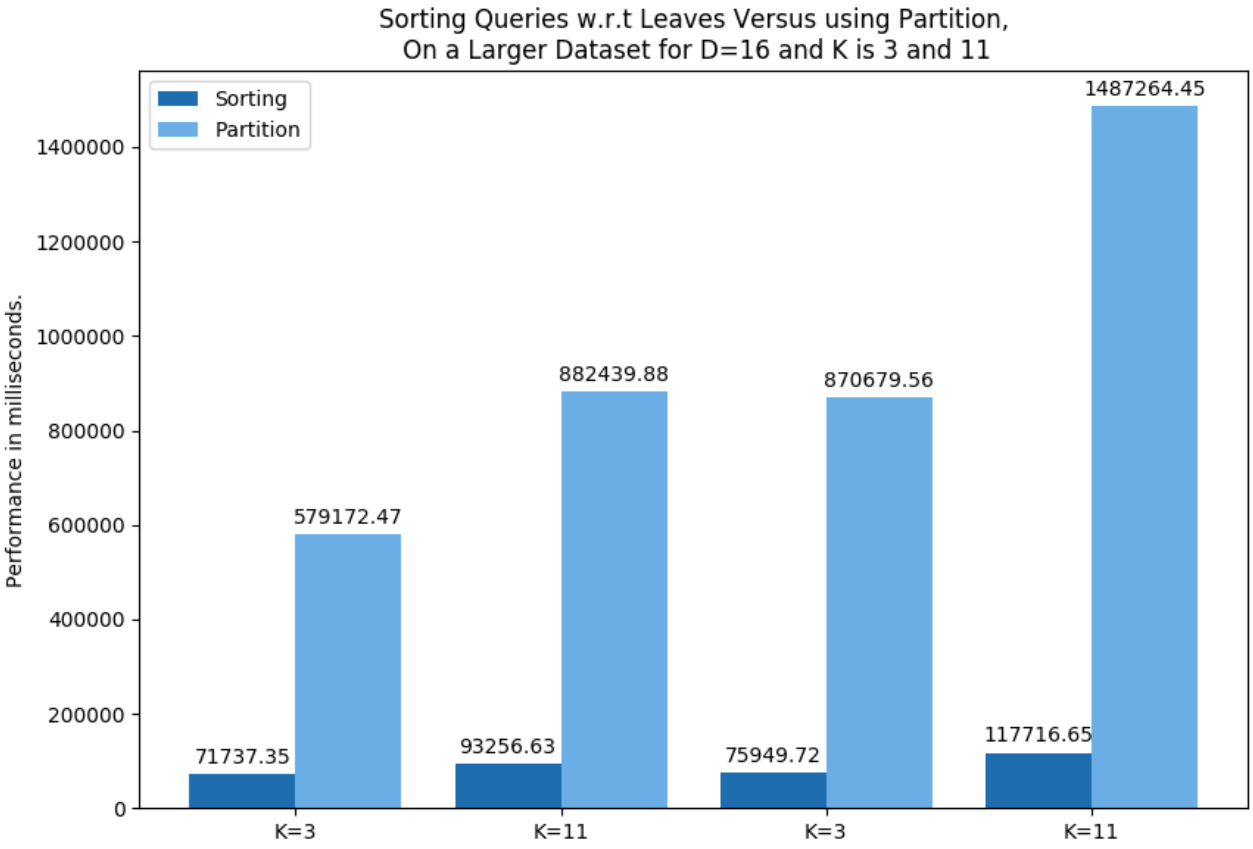
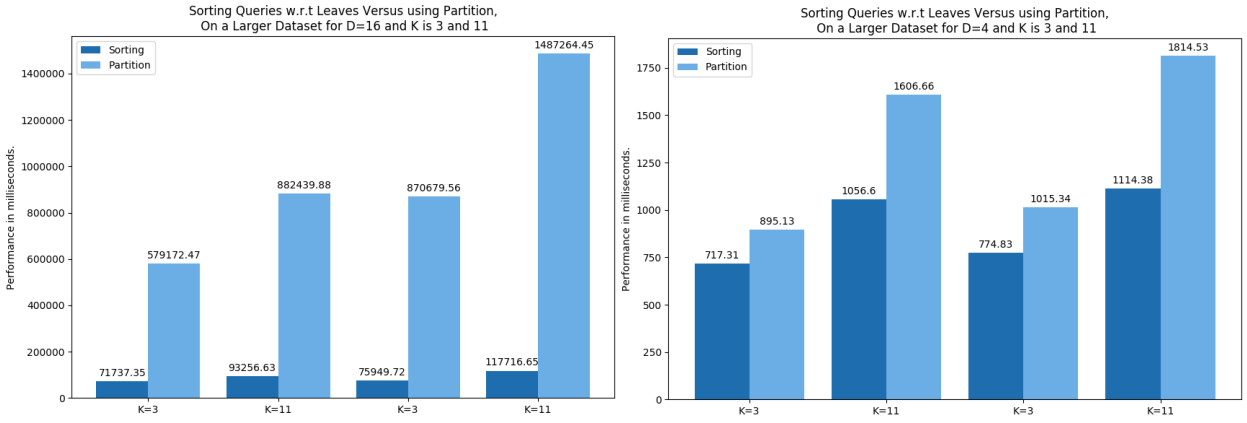


Fig. 8



(a) Flower one.

(b) Flower two.

Fig. 9: My flowers.

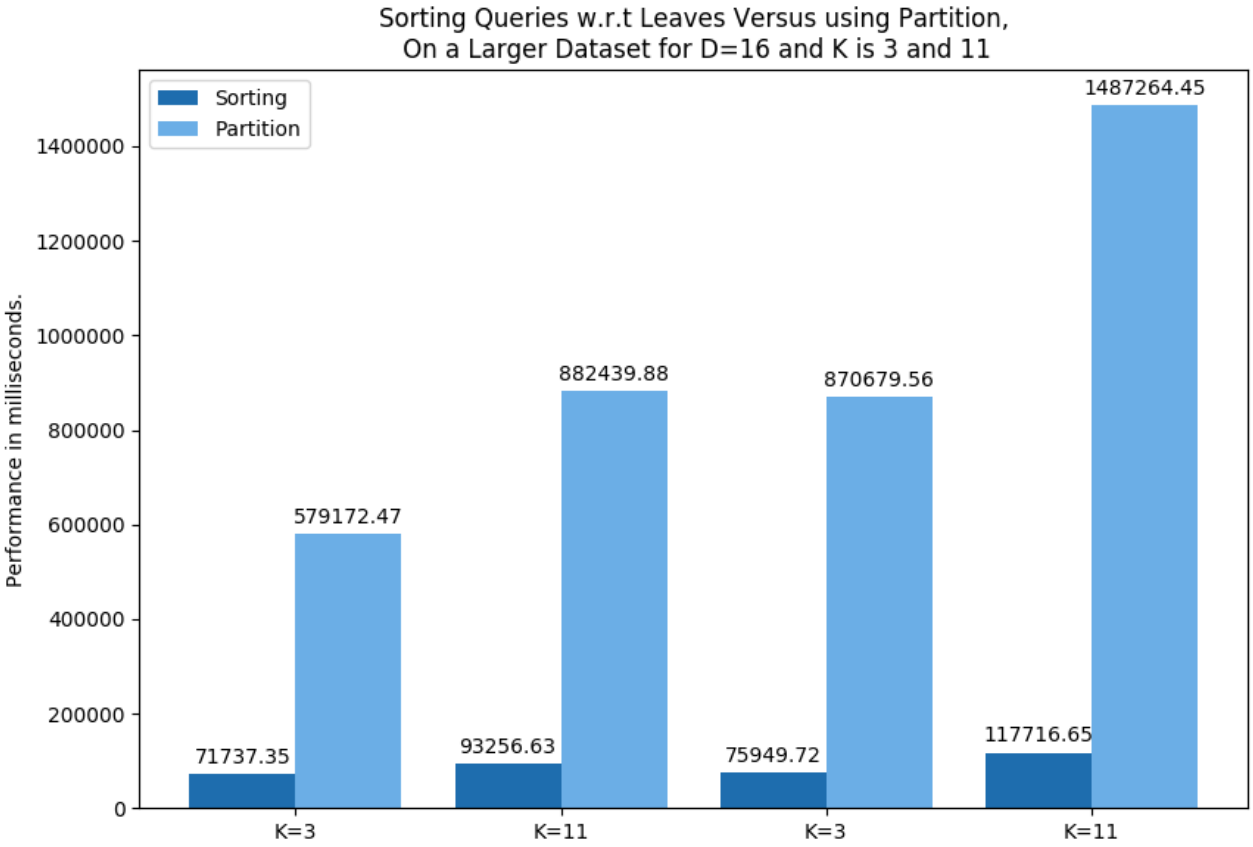
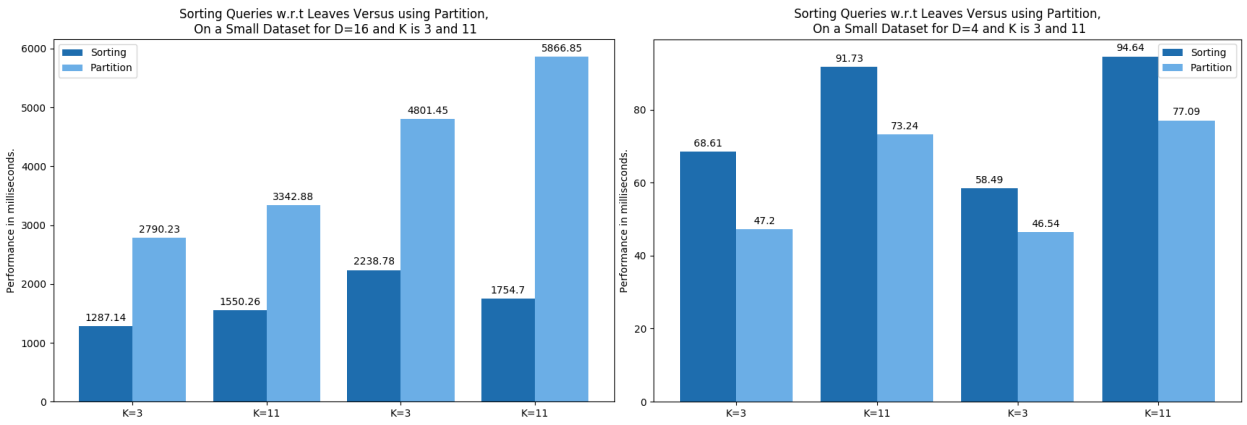


Fig. 10



(a) Flower one.

(b) Flower two.

Fig. 11: My flowers.

4.2 Optimising Tree Traversal with Full Dimensionality Checking

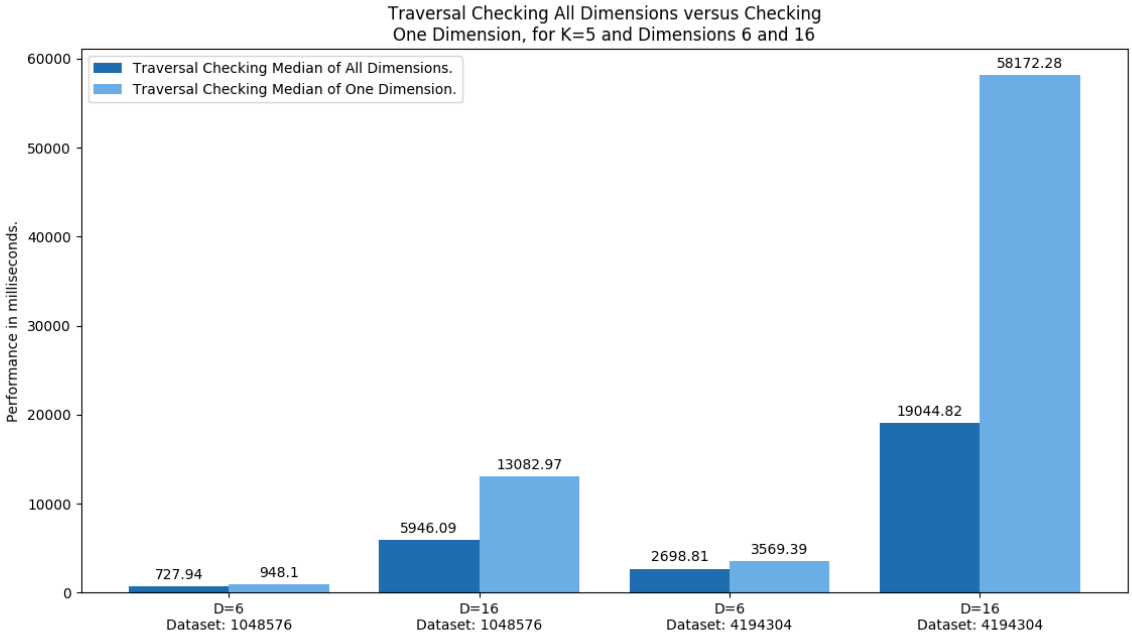


Fig. 12: Traversal comparing results w.r.t varying dimension sizes on a smaller and larger dataset.

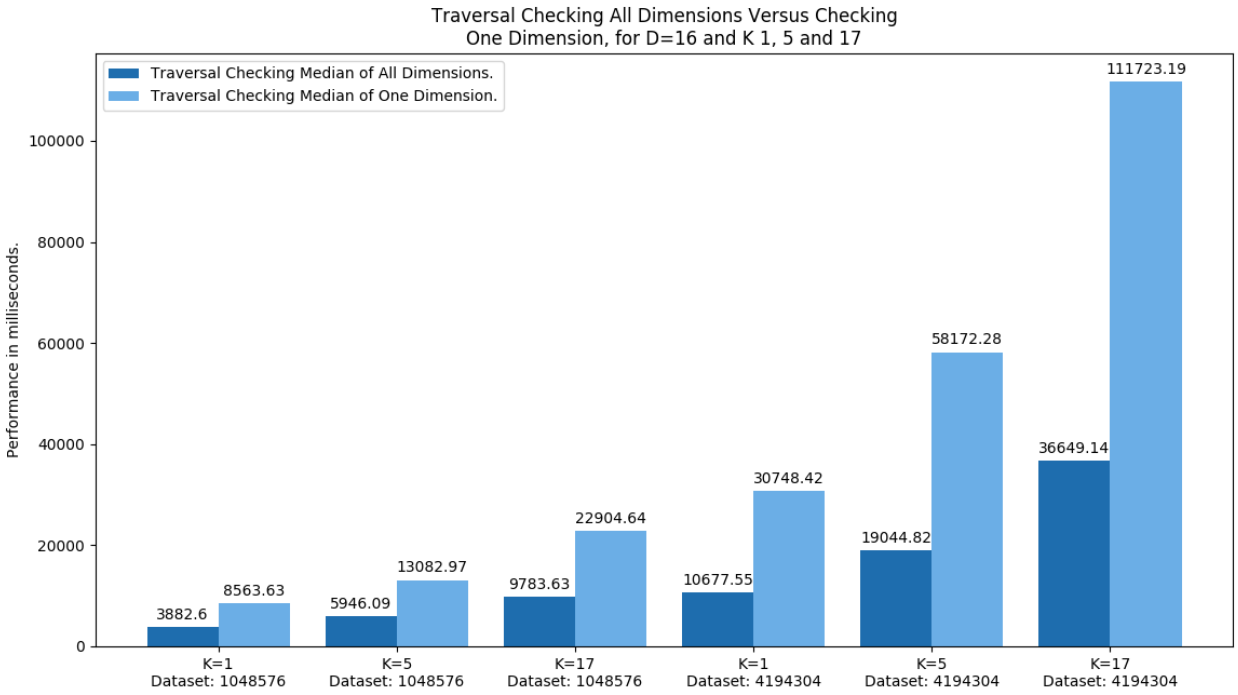


Fig. 13: Traversal comparing results w.r.t varying sizes for K on a smaller and larger dataset.

4.3 Brute Force versus k-d Trees for Computing KNN

5 Conclusion

References

Friedman, J. H., Bentley, J. L., & Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*.

He, K., & Sun, J. (2012). Computing nearest-neighbor fields via propagation-assisted kd-trees. *Microsoft Research Asia*.