Search Trees for Nearest Neighbor Problem

Ye Zhihao
09016319

Huang Ni
21016102

***Abstract*—This report will follow the footprint of how NNS problem is derived, defined and dealt with by some effective tree-search methods. First, we will clarify the concept of metric and the definition of NNS problem, and solve it using a min heap which is useful throughout. Then, dimension space and kd-tree search will be introduced. Next, cover tree will be brought up and discussed in detail. Finally, we will make a conclusion of the report and make some suggestions in choosing proper data structures for specific scenarios.**

***Keywords*— Metric; NNS; K-d Tree; Cover Tree**

I. Introduction

In this section, we will give an explanation of k-Nearest Neighbour Search, and then illustrate the weaknesses in simple linear search.

A. Metric

A metric is a binary function which takestwo points inS, and returns the similarity of two points.

Let V be a set. A function $d : V \times V \to \mathbb{R}$ is called a metric for V if

- $d(x, y) \geq 0$
- $d(x, y) = 0 \;\Leftrightarrow\; x = y$
- $d(x, y) = d(y, x)$
- $d(x, y) \leq d(x, y) + d(y, z)$

where $x, y, z \in V$. The pair (V, d) is then called the *metric space.*

A metric is not constraint to the common concept of distance, but anything that calculates similarity, such as the **Pearson Correlation Coefficient** in digital image processing.

In Haskell, the concept of metric can be easily introduced through typeclass:

```
type MetricFn p = p -> p -> Double

class (Eq p, Ord p) => Metric p where
   distance :: MetricFn p
```

B. Nearest Neighbour Problem

Consider a set of points $S \subset V$. Given another point $p \in V$, one of our interests is one distinct point $q \in S$ that is nearest to p:

$$\text{argmin}_{q \in S} d(p, q)$$

This problem is known as *nearest-neighbor problem.* The k-nearest neighbour search(k-NNS) is an natural extension of nearest neighbour search:

$$\text{min}_{q_i \in S} \sum_{i=1}^{k} d(p, q_i)$$

It is worth noting that, the definition of NN problem does not involve a so-called *dimension space* X, a concept often exploited to represent a point, whereas a *metric space* M is a must. As we will see later on, this feature leads to what is called the *metric tree*, enabling us to do NN search in everything that can calcluate similarity.

In Haskell, we denote any data class that is capable of doing NN searching the *Searcher*, and require it to implement NN-related interfaces:

```
class Searcher s where
type Pt s :: *
type Val s :: *
nearNeighbors :: (Metric p) => s p v ->
    Double -> p -> [(p, v)]
kNearest :: (Metric (Pt s)) => Int -> Pt s
    -> s -> [(Pt s, Val s)]
nearest :: (Metric (Pt s)) => Pt s -> s ->
    (Pt s, Val s)
nearest = (head .) . kNearestNeighbors 1
```

where Pt denotes the point type, Val denotes the label.

II. Some NNS Implementations

Before stepping into cover tree, this section will present two traditional ways to solve the NNS problem. Tools used and concepts introduced here will also be helpful for the understanding of next section [1].

A. Brute force and Max k heap

The first searcher to implement is the brute-force linear searcher. Obviously, computing the nearest neighbor using a sequential search requires linear time, and there is little we can do to improve in terms of distance calculation, as each point in the set S will be considered by the query.

However, there are something we can do in respect of nearest point preservation. To efficiently keep track of the k nearest points, we can preserve a max heap that holds only k elements.

It means that, we can use a function to quickly filter out all the points that is far enough, where k nearest points are remained:

```haskell
kMinsByHeap :: forall p v. (Ord p) => Int
    -> [(p, v)] -> [(p, v)]
```

Then, apply it to improve the brute-force search process:

```haskell
newtype BruteForce p v = BruteForce [(p,
    v)]
instance Searcher (BruteForce p v) where
   type Pt (BruteForce p v) = p
   type Val (BruteForce p v) = v
   kNearest k sample (BruteForce examples)=
      let distExamples =
          map (distance p sample,) examples
      in snd <$> kMinsByHeap k distExamples
```

As we can see, the *kMinsByHeap* function processes on a list of points whose distances are already calculated. It means that if we prune some of the points, just as what all the search tree does, we may still apply this function to pick out the k nearest points. Therefore, the heap method is a general component in KNN search implementation in practice.

B. Dimension space and K-d Tree

A point, in our common sense, is normally a vector of numbers, with each *axis* corresponding to one *coordinate*. To sum up, it is an element in the **Dimension Space**.

We model these concepts in Haskell:

```haskell
class (Enum a, Bounded a) =>
    DimensionSpace a where
   -- number of coordinates of a point.
   dimension :: a -> Int
   -- get all available axes.
   axes :: a -> [a]
   -- get the base axis.
   baseAxis :: a -> a
   baseAxis = head . axes
```

A dimension space should be bounded with fixed dimensions, and each dimension can be enumerated. We call the handle of each dimension *axis*, and there is a base axis for the simplicity of programming.

Then, we call the data residing in one axis **coordinate**:

```haskell
class (Ord c) => Coordinate c where
   average :: [c] -> c
   variance :: [c] -> c
   absDiff :: c -> c -> Double
```

The coordinate has some statistical features, such as means and variance, and we can represent each two coordinate's difference in $\mathbb{R}$.

Finally, we model *dimension space* and *coordinate* together to form a **point**:

```haskell
class (DimensionSpace a, Coordinate c) =>
    Point p a c where
   -- gets the point's k-th Coordinate.
   coord :: a -> p a c -> c
   toList :: p a c -> [(a, c)]
```

K-d Tree is such a data structure that exploits these concepts. It partitions the space by several hyperplane, each perpendicular to one specific axis. The well-known *binary search tree* is just the 1-d specialization of this tree.

We model the K-d tree node following the conventional practice:

```haskell
data KdNode a p v = Nil | KdNode {
   kdEntry :: (p, v),
   kdAxis :: a,
   kdLeft :: KdNode a p v,
   kdRight :: KdNode a p v
} deriving (Eq, Ord, Show, Read)
```

A KdNode holds a representative point, along with the label it shows, and an axis representing the current partitional hyperplane. The hyperplane splits the current space into two sub-space, each represented by a subtree.

Then, a Kd-Tree is a wrapper of its root node:

```haskell
data KdTree a p v = KdTree {
   kdSize :: Int,
   kdRoot :: KdNode a p v
} deriving (Show, Read)
```

Just consider the AVL tree or Red-Black tree in the domain of BST, it could be easily realized that maintaining the balance of K-d Tree is also an odyssey.

Therefore, this report only implements the construction of K-d Tree, which ensures the tree to be balanced, and the k-NN search process to serve as a *Searcher*.

*1) **Construction:***

Construct a BST, namely 1-d tree, from a point list is easy, as we only need to sort it once, and in every subtree construction the list is still sorted.

Things become different in K-d Tree ($k \geq 2$). No matter how we sort, we can only make sure coordinates from one axis to be ordered. It means that when we finish the construction of one node and step into the subtree, we cannot assert the points in current context is in order.

Given that, we turn to select median number of every axis instead of trying to sort the whole list. Searching for the medium number can be accomplished by the idea of *Quick Sort*, yet discarding the sub-interval which will never contain the median number:

```haskell
median :: (b -> b -> Ordering) -> [b] ->
    ([b], b, [b])
median cmp xs = kthBig (length xs/2) cmp xs
   where kthBig k cmp (x:xs)
       | k < n =
        let(l,m,r)=kthBig k cmp left
        in (l,m,r ++ x:right)
       | k > n =
        let(l,m,r)=kthBig (k-n-1) cmp right
        in (left ++ x:l, m, r)
       | otherwise = (left, x, right)
      where (left,right)=
                     partition ((==LT) .
                         (`cmp` x)) xs
               n = length left
```

Then, we obtains all the coordinates from each axis:

```
-- arrange points in such a way that all
    coords in same axis in one list
coordsByAxis :: (Point p a c) => [p a c]
    -> [(a, [c])]
coordsByAxis = foldl zipPt initial . map
    toList
   where zipPt = zipWith (\(_, cs) (axis,
       c) -> (axis, c:cs))
        initial=zip (axes (undefined :: a))
            (repeat [])
```

We choose the ideal axis with greatest variance, since it will better divides all the points and increase the probablity of pruning in NN search:

```
-- median of the most spread dimension
    pivoting strategy
selectAxis :: (Point p a c, Metric (p a
    c)) => [p a c] -> a
selectAxis ps = fst $ maximumBy (comparing
    snd) vars
   where vars = map (\(axis, cs) -> (axis,
       variance cs)) $ coordsByAxis ps
```

Finally, we could comprise them together to construct a K-d Tree from a point list:

```
fromList :: (Point p a c, Metric (p a c))
    => [(p a c, v)] -> KdTree a (p a c) v
fromList pvs=Tree (length pvs) (build pvs)
   where build [] = Nil
        build pvs =
          let a = selectAxis (fst <$> pvs)
              (l, mid, r) = medianPartition
          in Node mid a (build l) (build r)
```

*2) **k-NN Search:***
The NN Search of K-d Tree proceeds like BST, except for that in each node it respects different axis of coordinates for ordering.

After reaching the leaf node, the search process sets the leaf node point as current nearest point and starts backtracking and go through all the subtrees that may contain node that is nearer.

The k-NN search process proceeds just the same as 1-NN search, except for that it keeps track of all the searched nodes in backtracking. After backtracking finished, it feeds the candidates into *kMinsByHeap* to get the k nearest neighbors.

In comparison with brute-force, K-d Tree just prunes all the points in the subtrees that will not contain nearer candidates, thus obtaining a better time complexity.

III. Cover Tree

A. Metric Tree

As has been stated in previous section, a point is a collection of features, which in most case can be modeled as a vector, with each dimension representing a unique feature.

However, a point is not constraint to such a structure. Instead, it can be any other well-structured model, such as image matrices, or even models of really irregular form like sentences in NLP. The inherent constraint of K-d Tree makes it hard to be applied to such non-vector objects.

So, what is truly needed is a data structure that solely depend on the feature of metric space. This is where the concept of metric tree derives.

A **metric tree** merely exploit properties of **Metric Space** to perform its function, immediately giving it the capability of working arbitrary objects like images and sentences.

One typical metric tree is what is called the *Ball Tree*. It was firstly brought up to solve K-d Tree's inharmony within the intersection of sphere and cube, which divides the points into two sub-ball-space. Every operation of ball does not involve the concept of *Dimension Space*, so a ball tree is exactly a *metric tree*.

Limited by insufficient vacant time, ball tree is not implemented in this report, which is in truth a regret.

Another outstanding instance of metric tree is what will be covered next, the cover tree.

B. Cover Tree: Definition

Cover tree requires only a metric for proper operation, and is oblivious to the representation of the points, which satisfies the requirement of metric tree. This allows the free mix of cartesian and polar coordinates, or just using implicitly defined points.

Cover tree is fast in practice, and have great theoretical significance because nearest-neighbor queries have guaranteed logarithmic complexity, and they allow the solution of the all-nearest-neighbors problem in linear time instead of $nlogn$.

A cover tree T is an infinitely leveled tree on dataset S with a metric d. Every node in the tree references a point and a label $(p, v) \in S$. All the nodes in each level i are grouped into an indexed set C_i.

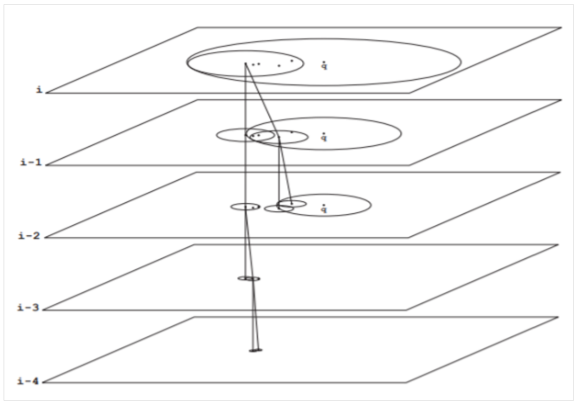

Fig. 1. Cover Tree

It can be viewed that a cover tree is composed of infinite levels of sets. But it is obvious that we cannot implement a tree with infinite node in a computer program. In fact, the author of the cover tree presents two form of representation: implicit and explicit, where the infinite one is the implicit form [2].

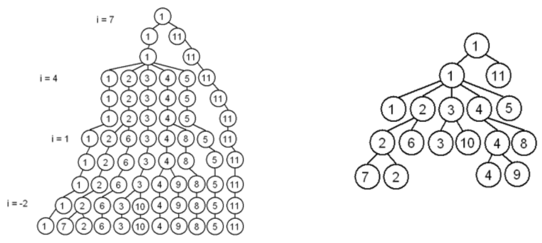

Fig. 2. Implicit v. Explicit

The explicit form coalesces all nodes in which the only child is a self-child.

Let's start modeling the cover tree in haskell:

```
type CoverNode p v = (p, v)
type CoverSet p v = [CoverNode p v]
```

We represent a node along with its children as a key-value map, where key is the parent, and value is the cover set of its children:

```
-- level -> parent -> children nodes
type CoverMap p v = M.Map p (CoverSet p v)
```

We from the data structure of cover tree as follows:

```
data CoverTree p v = CoverTree {
   levelRange :: (Int, Int),
   coverLevels :: IM.IntMap (CoverMap p v)
} deriving (Eq, Show, Read)
```

Here, *levelRange* denotes the effective range of explicit representation, and each level C_i is implemented as a *CoverMap*, making it easy to retrieve a node's children.

Then, let's define some basic functions:

```
-- Address level i's cover map
coverMap :: (Metric p) => Int -> CoverTree
    p v -> CoverMap p v

-- Address level i's node's cover set
coverSet :: (Metric p) => Int -> CoverNode
    p v -> CoverTree p v -> CoverSet p v

-- Address C_i
coverLevel :: (Metric p) => Int ->
    CoverTree p v -> CoverSet p v

-- Children acquisition
children :: (Metric p) => CoverTree p v ->
    (Int, CoverNode p v) -> CoverSet p v
children = flip $ uncurry (coverSet . pred)
```

Other data member related:

```
-- Level ranges
minLevel :: CoverTree p v -> Int
minLevel (CoverTree (l, _) _) = l

maxLevel :: CoverTree p v -> Int
maxLevel (CoverTree (_, r) _) = r

-- The whole dataset lies in the negative
    infinity level
dataSet :: (Metric p) => CoverTree p v ->
    CoverSet p v
dataSet = coverLevel (minBound :: Int)

-- Root set is the positive infinity level
    set with only one node
rootSet :: (Metric p) => CoverTree p v ->
    CoverSet p v
rootSet = coverLevel (maxBound :: Int)

-- Root node is the only node that lies in
    root set
root :: (Metric p) => CoverTree p v ->
    CoverNode p v
root = head . rootSet
```

As we can see, the cover tree is not implemented in the traditional tree structure. However, such explicit representation highly conforms to the mathematical concepts of origin paper, and brings a lot of benifits for implementing other concepts, especially the following invariants.

C. Cover Tree: Invariants

Cover tree holds three unique invariant properties [2]:

1) ***Nesting****:* $C_i \subseteq C_{i-1}$

Nesting indicates following features:

- Each level C_i can be viewed as upward filter layer, and there are infinite such layers.
- Each node in set C_i has a self-child. It means all nodes in set C_i are also nodes in sets $C_{i-k}, k > 0$.
- $C_{+\infty}$ contains the singleton root node, $C_{-\infty} = S$.

```
nesting t i = csI<=csI' where [csI,csI']=..
```

2) ***Covering Tree****:* For every point $p \in C_{i-1}$, there exists an only point $q \in C_i$ such that the $d(p, q) \leq 2^i$.

This property can be viewed from two perspectives:

- Covering. Each upper level node will exclusively cover a set of lower level points in a circle with radius 2^i.
- Tree. The 'exist and exactly one' constraint implies a one-to-many mapping of points, making it suitable for a tree structure.

```
coveringTree t i = let [csI, csI'] in
and [length [q | q <- csI, distance p q <=
    2^i] == 1 | p <- csI']
```

*3) **Separation**:* For all distinct points $p, q \in C_i$, the distance of p and q is greater than 2^i.

Since every node p has self-child in infinite lower levels, when a node q is proper to be the child of p, which level should it be inserted?

The seperation property ensures that each node will be inserted at lowest possible level.

```
  separation t i = let csI in
  and [p == q || distance p q > 2^i
        | p <- csI, q <- csI]
```

Finally, we could write a utility function to check whether a cover tree satisfies all the invariants:

```
  checkInvariant t = all (\i -> and (
    [n, ct, s] <*> [t] <*> [i]
  )) [minLevel t.. maxLevel t]
```

D. Cover Tree: Construction

Limited by insufficient time, the construction of cover tree is based on the single point insertion, rather than the batch construction [3], which is truly a regret.

The insertion of cover tree has two primary tasks: find appropriate parent, and insert it into appropriate level. With the explicit representation, there is some more subtle work to do when doing insertion.

First, we define the *uncoalesce* function to expand the self-child:

```
  uncoalesce :: (Metric p)
          => Int -- stop level
          -> CoverNode p v -- expand node
          -> CoverTree p v -- in
          -> CoverTree p v -- out
```

Then, we define a singleton construction to feed into first node:

```
  singleton :: (Metric p) => Int -> (p, v)
       -> CoverTree p v
```

Where the first parameter denotes the maxLevel. Normally it is set to the furtherest possible distance of the dataset. It works as an entry to the *fromList* method.

However, theoretically there is no need to specify the maxLevel, which is totally reasonable to starts from 0 and be adjusted dynamically.

Next, we define a child insertion, which directly put a node as a child of another node:

```
  insertChild :: (Metric p)
          => CoverTree p v -- in
          -> (Int, CoverNode p v) -- (lv, p)
          -> CoverNode p v -- child
          -> CoverTree p v -- out
```

Finally, we could implement the node insertion just as what papers reveals using Haskell's list comprehension:

```
insert :: (Metric p, Eq v) => CoverTree p v
    -> (p,v) -> CoverTree p v
insert t (p,v) = snd $ insert' p (rootSet t)
    (maxLevel t) where
insert' p qsI i
   | setDist p qs > 2^i = (False, t)
   | not parentFound && setDist p qsI <= 2^i =
      let q = head [q | q<-qsI, nodeDist p q
          <= 2^i]
      in (True, insertChild t (i, q) (p, v))
   | otherwise = (parentFound, newTree)
   where
      qs = cat [children t (i, q) | q<-qsI]
      qsI' = [q | q<-qs,nodeDist p q <= 2^i]
      (parentFound, newTree) = insert' p qsI'
          (i - 1)
```

It should be paid special attention that, the maxLevel used here does not correspond to the C_∞ used in origin paper. It is because that the actual appropriate level for new node insertion may be greater than the current maxLevel. So, unless specifying the maxLevel to be great enough in the initial construction, the insertion process should find the new appropriate maxLevel and uncalesce the root node reversely.

E. Cover Tree: k-NN Search

The fundamental idea of cover tree's NN searching is much the same as k-d tree's: pruning all the nodes that cannot lead to the nearest neighbor. The algorithm iteratively constructs sub-level nodes by expanding current level to its children, then throwing away hopeless ones.

Let's first implement the 1-NN search:

```
nearest p t =
   let qsInf = rootSet t
      qsInf' = foldl (\qsI i ->
         let qs = cat [children t (i, q) |
             q<-qsI]
         in [q | q<-qs,distance p (fst q) <=
             setDist p qs + 2^i]
      ) qsInf [maxLevel t..minLevel t]
   in argmin (distance p . fst) qsInf'
```

According to the original paper, the k-NN method is implemented with template the 1-NN version, except for that a new *Bound function* is delicately selected, in this case the distance to the kth closest point in candidates:

```
   kNearest k p t =
      let qsInf, qsInf', qsI, i = ...
         let ds = [dist p (fst q) | q <- qsI]
            dK = fst (last $ kMinsByHeap k ds)
            qs = ...
         ...
      in map snd $ kMinsByHeap k [(distance p
          (fst q), q) | q <- qsInf']
```

IV. Test

Due to the lack of benchmark relative knowledge in Haskell, we have encountered a handful of troubles when testing the three search methods, the most impactive of which is the laziness nature of Haskell.

Eventually, we use the ':set +s' switch in ghci to keep track of the execution time of a function, and set function return value to accuracy as benchmark. The benchmark contains construction time, execution time, as well as the dataset loading time in a single function.

```
testKdTree :: IO Double
testKdTree = do
   content <- readFile "test/test_set.txt"
   let rawData = reverse . splitOn "\t"
       <$> lines content
   let dataset = map (\(tag:point) ->
       (Vector $ map read point ::
       Vector3d, read tag :: Int)) rawData
   let (testSet, trainSet) = splitAt
       (length dataset `div` 3) dataset

   let kdTree = KdTree.fromList trainSet
   testOnDataset trainSet testSet kdTree
```

The dataset is a line-separated text file, with first three numbers in each line indicating three features, and last integer representing the label. Thus, it is a dataset composed of a Vector3d Point type and Int Value type. We use 3-cross validation to split the dataset.

```
15823 0.991725 0.730979 2
35432 7.398380 0.684218 3
53711 12.149747 1.389088 3
64371 9.149678 0.874905 1
9289 9.666576 1.370330 2
```

Test Result:

1) Brute Force:

- accuracy: 0.7657657657657657
- time, memory: (2.89 secs, 1,320,640,808 bytes)

2) K-d Tree:

- accuracy: 0.7687687687687688
- time, memory: (0.46 secs, 161,498,408 bytes)

3) Cover Tree:

- accuracy: 0.7666666666666667
- time, memory: (0.20 secs, 75,272,752 bytes)

It is manifest that cover tree is the most optimal both on time and space out of three methods.

V. Conclusion

In this report, we introduced and defined the nearest-neighbor problem step by step, and presented three different search method on solving this problem. We could see that none of these methods are isolated from others, but they all derive something inspiring from each other.

In conclusion, it is obvious that brute-force search is not very appropriate for k-NNS problem. But this is not definitely true. To be specific, the brute force method is an ideal solution for NNS problem when the number of dimension is not more than 2. In this case, the method can provide a $O(dlogn)$ query time, take $O(dn)$ space and $O(dnlogn)$ preprocessing time. For $d = 1$, it is in fact the binary search on a sorted array, whereas for $d = 2$ it is the use of Voronoi [4] diagrams and a fast planar point location algorithm. However, for $d > 2$, this method is less than ideal in that it works well only in the expected case and only for moderate d which is less than 10.

The two tree structure – K-d Tree and Cover Tree have their own cases for application, where k-d tree requires the point to conforms with a vector, whereas cover tree can be generally applied to anything in a metric space.

However, we could also see that cover tree has the best optimal space and time performance over all other methods, making it first recommended choice when solving NN problems.

References

[1] A. M. Kibriya and E. Frank, "An empirical comparison of exact nearest neighbour algorithms," in *European Conference on Principles & Practice of Knowledge Discovery in Databases*, 2007.

[2] A. Beygelzimer, S. M. Kakade, and J. Langford, "Cover trees for nearest neighbor," *ACM Conference*, 2006.

[3] C. R. S. Mike Izbicki, "Faster cover trees," *Proceedings of Machine Learning Research*, 2015.

[4] M. Kolahdouzan and C. Shahabi, "Voronoi-based k nearest neighbor search for spatial network databases," in *ACM Conference*, 2004.