Advanced Data Structures

kd-trees

November 1, 2019

A *kd-tree* (short for "k-dimensional tree") is a special data structure that stores a collection S of points from k-dimensional space $S = \underbrace{A_0 \times \ldots \times A_{k-1}}$ where $A_0, \ldots, A_{k-1}$ are totally ordered sets[1] of values, which can be used to answer questions like

Given a point $X \in S$ and a finite collection of n points $M \subset S$

Find the point $P \in M$ which is *most similar* to X.

Remark: Most often, by *"most similar point to X"* we understand "*closest point to X.*" If M has n points in $\mathbb{R}^2$, then a naive algorithm can solve this problem in linear time $O(n)$:

```
FindNeaestPoint($S, X$)
bestdist := $\infty$
guess := Null
for each $P \in S$ do
    $d$ := $distance(P, X)$
    if $d$ < bestdist then
        bestdist := $d$
        guess := $P$
end for
return guess
```

In this lecture we show that, if we store the n points of M in a data structure called *kd-tree*, then, for a random distribution of X, we can find in $O(\log n)$ time the nearest neighbor of X in M. Even better, the kd-tree can be used to find the the K nearest neighbors of X in $O(\log n)$ time, where $K \geq 1$ is a constant.

A kd-tree with initial axis $i \in \{0, 1, \ldots, k-1\}$ for a set of points M is a binary tree whose nodes store the points in M, and satisfies the following conditions:

- For every node X at level n, if $(a_0, \ldots, a_{k-1})$ is the point stored in X, and $i = n \mod k$ then

[1] A set A is totally ordered if there is a binary relation $\leq$ on A which is **total** (for all $a, b \in A$, either $a \leq b$ or $b \leq a$), **reflexive** ($a \leq a$ for all $a \in A$), **antisymmetric** (for all $a, b \in A$, if $a \leq b$ and $b \leq a$ then $a = b$) and **transitive** (for all $a, b, c \in A$, if $a \leq b$ and $b \leq c$ then $a \leq c$).

– $a'_i \leq a_i$ for all points $(a_0, \ldots, a_{k-1})$ stored in the left subtree of X,

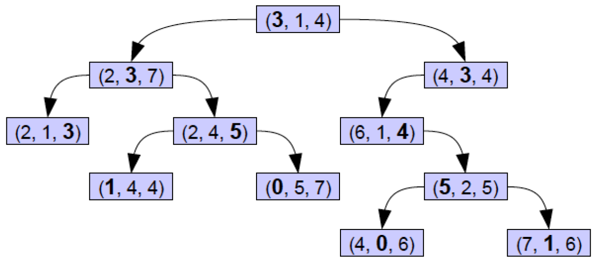
– $a_i < a'_i$ for all points $(a'_0, \ldots, a'_{k-1})$ stored in the right subtree of X.

Thus, a kd-tree is generalization of a binary search tree where the position of the search key in points depends on the level of the node: the search key at a point X stored in a node at level n is the $(n \mod k)$-th coordinate of X, where we assume that the point coordinates are indexed starting from 0.

For example, if $k = 3$, $A_0 = A_1 = A_2 = \mathbb{R}$, and

$$\begin{aligned} S = \{ \; & (0,5,7), (1,4,4), (2,1,3), (2,3,7), (2,4,5), \\ & (3,1,4), (4,0,6), (4,3,4), (5,2,5), (6,1,4), (7,1,6)\} \end{aligned}$$

then a kd-tree which stores the points of S is



1 Building a balanced kd-tree

Given L a list of k-dimensional points and an initial axis $i \in \{0, 1, \ldots, k-1\}$, we define recursively the kd-tree $T(L, i)$ with initial axis i for L as follows:

Base case: If L is empty, then $T(L, i)$ is the empty tree.

Recursive case: Let $L' = [P_1, \ldots, P_n]$ be the result of sorting L in increasing order of the i-th coordinate value of its points, $m_0 = \lfloor n/2 \rfloor$, and

$$m_1 := \max\{j \mid m_0 \leq j \leq n \text{ and } P_{m_0}, P_j \text{ have same } i\text{-th coordinate}\}.$$

Then $T(L, i)$ has

- point P_{m_1} stored in the root,
- left subtree $T([P_1, \ldots, P_{m_1 - 1}], (i+1) \mod k)$, and
- right subtree $T([P_{m_1+1}, \ldots, P_n], (i+1) \mod k)$.

Th kd-tree built in this way is balanced if $m_0 = m_1$ in all recursive steps.

2 Finding a node in a kd-tree

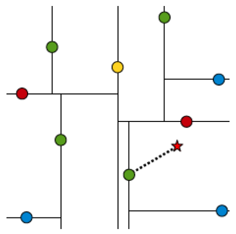
Suppose T is a kd-tree with initial axis i. The pseudocode to find the node of T for a point $A = (a_0, \ldots, a_{k-1})$ is:

```
FindPoint($T, i, A$)
if $T$ is empty then return 'node not found'
$(a'_0, \ldots, a'_{k-1})$ :=the point stored in the root of $T$
if $a'_j = a_j$ for $0 \leq j < k$ then return root of $T$
if $a_i \leq a'_i$ then
   return FindPoint($T$.left, $(i+1) \mod k, A$)
else /* $a_i > a'_i$ */
   return FindPoint($T$.right, $(i+1) \mod k, A$)
```

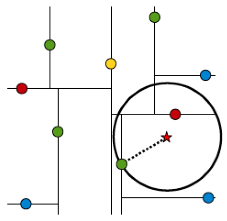
3 Finding the nearest neighbor of a point

Assume T is a kd-tree with initial axis i, and we want to find the point in T which is closest to a test point $A = (a_0, \ldots, a_{k-1})$.

The intuition behind the algorithm presented here is the following. Suppose that we have a guess of what we think the nearest neighbor to the test point is. E.g., suppose that the test point is indicated by the star and that we think the nearest neighbor is the point connected to the star by the dashed line:



Observation: If there is a point in this data set that is closer to the test point than our current guess, it must lie in the circle centered at the test point that passes through the current guess:



In a k-dimensional space, this circle is a hypersphere, called the **candidate hypersphere**.

- The previous observation lets us prune subtrees of the kd-tree which do not hold the nearest neighbor: If the hypersphere is entirely on one side of a splitting hyperplane, then the nearest neighbor can not be in the subtree for the points in the side opposite to the side of the hypersphere.

E.g., in the illustrated example, the circle is entirely to the right of the splitting hyperplane running vertically through the root of the tree. Therefore, any point to the left of the root of the tree cannot possibly be in the candidate hypersphere.

- The criterion to detect if a hypersphere with center $(a_0, \ldots, a_{k-1})$ and radius d is not completely on one side of a hyperplane defined by $x_i = \mathrm{curr}_i$ is straightforward: $|\mathrm{curr}_i - a_i| < d$.
- If the coordinates of points are real values, it is common to assume that the distance between points $X = (x_0, \ldots, x_{k-1})$ and $Y = (y_0, \ldots, y_{k-1})$ is the Euclidean distance, that is:
$$distance(X, Y) = \sqrt{(x_0 - y_0)^2 + \ldots + (x_{k-1} - y_{k-1})^2}.$$

Based on these remarks, we define the following algorithm (pseudocode) to find the point in T which is closest to the test point A:

```
1NN($T, i, A$)
Maintain a global best estimate of the nearest neighbor, called guess
Maintain a global value of the distance to that neighbor, called bestDist
guess := Null
bestDist := $\infty$
1NNAux($T$.root, $i, A$)
return guess
```

where

```
1NNAux(curr, $i, A$)
if curr is empty then return
if $distance(A,$ curr$)$ < bestDist then
  guess := curr
  bestDist := $distance(A,$ curr$)$
/* recursive search of $A$ in current kd-tree curr */
if $a_i \leq \mathrm{curr}_i$ then
  search :=' left'
  1NNAux(curr.left, $(i+1) \bmod k, A$)
else
  search :=' right'
  1NNAux(curr.right, $(i+1) \bmod k, A$)
/* Check if the candidate hypersphere crosses this splitting hyperplane */
if $|\mathrm{curr}_i - a_i|$ < bestDist then
  if search =' left' then
    1NNAux(curr.right, $(i+1) \bmod k, A$)
  else
    1NNAux(curr.left, $(i+1) \bmod k, A$)
```

4 Finding the K nearest neighbors of a point

The 1NN algorithm can be generalized to find the K nearest neighbors of a test point $A = (a_0, \ldots, a_{k-1})$ in a kd-tree T with initial axis i: instead of keeping track only of a global best guess `guess`, we keep track of global structure where we store the best K guesses found so far. A convenient auxiliary data structure for this purpose if a bounded priority queue (BPQ), whose behavior is described in the Appendix.

```
kNN(T, i, A)
Maintain a BPQ of the candidate nearest neighbors, called bpq
Set the maximum size of bpq to K
kNNAux(T.root, i, A)
return bpq
```

where

```
kNNAux(curr, i, A)
if curr is empty then return
/* add curr to bpq */
enqueue curr into bpq with priority distance(A, curr)
/* recursively search the half of the tree that contains the test point A */
if $a_i < curr_i$ then
  search :=' left'
  kNNAux(curr.left, $(i+1) \mod k$, A)
else
  search :=' right'
  kNNAux(curr.right, $(i+1) \mod k$, A)
p :=priority of max.-priority element of bpq
if bpq is not full OR $|curr_i - a_i| < p$ then
  if search =' left' then
    kNNAux(curr.right, $(i+1) \mod k$, A)
  else
    kNNAux(curr.left, $(i+1) \mod k$, A)
```

Figure 1: Algorithm kNN.

The generalized algorithm is called kNN (short for K nearest neighbors). Its pseudocode is shown in Figure 1.

Labworks

1. (A data structure for kd-trees) Use `C++` to implement a kd-tree data structure `kdTree` for sets of points represented as instances of the structure

   ```
   struct Point {
       float x[3]; // the coordinates of the point
   }
   ```

   and implement the methods

   - `makeKdTree(vector<Point> v)` which creates a balanced kd-tree for the nodes stored in vector `v`
   - `insert(T, P)` which inserts point `P` in the kd-tree `T`.

2. (The kNN problem) Write a program that does the following:

   (a) It creates a balanced kd-tree T for a set of n points, by reading their coordinates from a text file `points.txt` with the following structure:

   > It has n lines, and each line contains three floating-point numbers separated by spaces, representing the coordinates of a point.

   For example, `points.txt` could have the following content:

   ```
   1.0 4.0 5.14
   -17.3 25 6.42
   0 0 0.3
   3.0 4.0 5.0
   ```

   (b) It asks the user to type in

   - the value of K, and
   - the coordinates of a test point A

   and returns the K nearest neighbors of A from T.

   Use the structures `Point` and `BPQ` available from the website of this lecture, and the class `kdTree` from Labwork 1.

A Bounded priority queues

A bounded priority queue (BPQ for short) is a special data structure similar to a regular priority queue, except that there is a fixed upper bound on the number of elements that can be stored in the BPQ. Whenever a new element is added to the queue, if the queue is at capacity, the element with the highest priority value is ejected from the queue. For example, suppose that we have a BPQ with maximum size five that holds the following elements:

Value	A	B	C	D	E
Priority	0.1	0.25	1.33	3.2	4.6

Suppose that we want to insert the element F with priority 0.4 into this bounded priority queue. Because this BPQ has maximum size five, this will insert the element F, but then evict the highest-priority element (E), yielding the following BPQ:

Value	A	B	F	C	D
Priority	0.1	0.25	0.4	1.33	3.2

Now suppose that we wish to insert the element G with priority 4.0 into this BPQ. Because G's priority value is greater than the maximum-priority element in the BPQ, upon inserting G it will immediately be evicted. In other words, inserting an element into a BPQ with priority greater than the maximum-priority element of the BPQ has no effect.

Note: an implementation of BPQ, designed to work with elements which represent points in $\mathbb{R}^3$, is provided in `BPQ.cpp` and `BPQ.h`, which can be downloaded from the website of this lecture.