Nearest Neighbor Search using Kd-Tree

Doe-Wan Kim
Language and Media Processing Lab
Center for Automation Research
University of Maryland

March 28, 2000

Kd-Tree

- Multidimensional binary search tree
- Takes $O(N \log N)$ time to build
- Definitions

  For a node P,

  - $K_0(P), \cdots, K_{k-1}(P))$: k keys of P (COORD(P))
  - DISC(P): discriminator of P
  - HISON(P), LOSON(P)

$$\forall Q \in LOSON(P), K_j(Q) < K_J(P)$$

$$\forall R \in HISON(P), K_j(R) \geq K_J(P)$$

  - NEXTDISC(i) = (I+1) mod k

    NEXTDISC(DISC(P)) = DISC(LOSON(P))

Insertion Algorithm

```
KD-COMPARE(P,Q)
/* Return the son of tree rooted at Q in which P belongs */
begin
   if DISC(Q)='X' then
      if XCOORD(P) < XCOORD(Q) then return 'LEFT'
      else return 'RIGHT'
   else if YCOORD(P) < YCOORD(Q) then return 'LEFT'
         else return 'RIGHT'
end
```

Insertion Algorithm (cont'd)

```
KD-INSERT(P,R)
/* Insert P in tree rooted at R */
begin
  if R=NULL then R ← P
  else
      while (R!=NULL) and (P and R do not have equal coordinate)
            F ←R
            d ←KD-COMPARE(P,R)
            R ←SON(R,d)
      if R=NULL then
        SON(F,d)←P
        DISC(P) ←NEXT-DISC(F)
end
```

Deletion in KD-Tree

- Not every subtree of kd-tree is kd-tree
- Need to consider the case of deleting root node
- Procedure for deleting root node R
  - If HISON and LOSON of R is empty, replace R with empty node
  - From the condition, replacement node should be from HISON
  - Choose a node which has minimum value in HISON
  - If HISON(R) is empty, find the smallest node in LOSON, and attach LOSON(R) to right son of this node. Do recursion

Deletion in KD-Tree

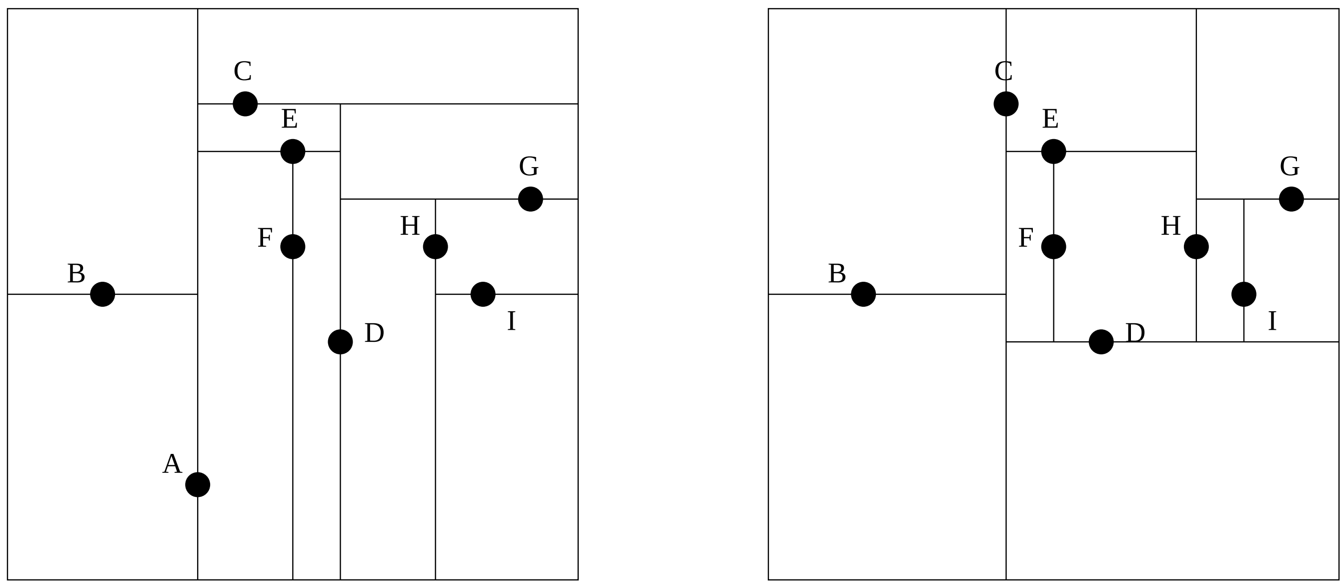

Figure 1: Deleting a root node in kd-tree

Deletion in KD-Tree (cont'd)

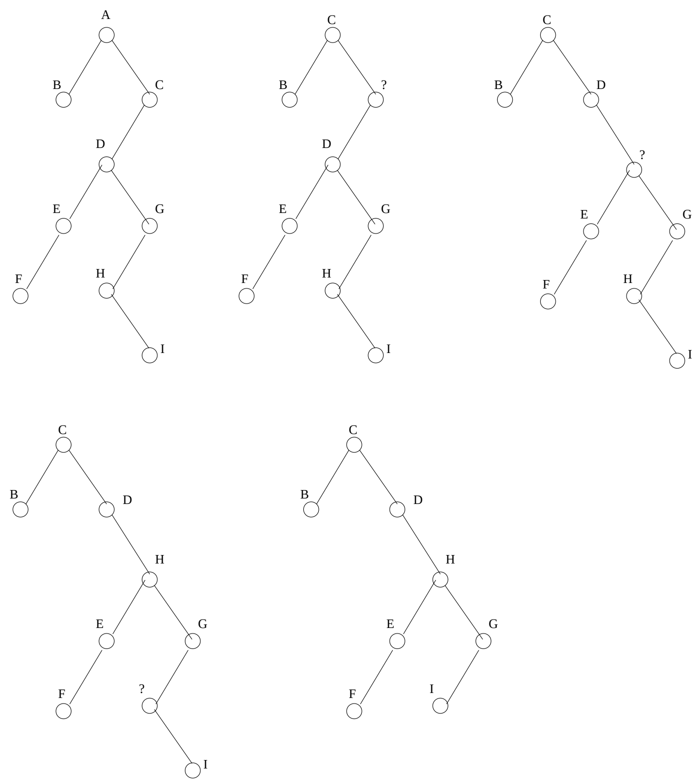

Figure 2: Deleting a root node in kd-tree

Deletion Algorithm

```
KD-DELETE(P,R)
/* Delete P from kd tree rooted at R */
begin
   N ←KD-DELETE1(P)
   F ←FIND-FATHER(P)
   if F=NULL then R ←N
   else SON(F,SONTYPE(P)) ←N
   return P
end
```

Deletion Algorithm (cont'd)

```
KD-DELETE1(P) /* Delete P and return pointer to root of resulting tree */
begin
  if LOSON(P)=HISON(P)=NULL then return NIL /* leaf*/
  else d ←DISC(P)
  if HISON(P)=NULL then /* When HISON is empty*/
    HISON(P) ←LOSON(P)
    LOSON(P) ←NIL
  R ←FIND-MIN(HISON(P),d)
  F ←FIND-FATHER(R)
  SON(F,SONTYPE(R)) ←KD-DELETE1(R)
  LOSON(R) ←HISON(P)
  HISON(R) ←LOSON(P)
  return R
end
```

Analysis of Deletion Algorithm

- Total path length of a tree built by inserting N nodes in random order is $O(N \log N)$
- Average cost of deleting random node has upper bound $O(\log N)$
- Deleting root node
  - Clearly bound by N
  - Time to find the minimum node in a subtree
  - Worst case: complete kd-tree at depth $k-1$
    Have to visit all nodes at depth 0 to $k-1$ ($2^k - 1$ nodes)
    Sums up to be $O(N^{1-1/k})$

Nearest Neighbor Search

- Query

  Given distance function D, set of points B, a point P.

  P's nearest neighbor Q in B is

$$\forall R \in B, \{(R \neq Q) \Longrightarrow (D(R, P) \geq D(Q, P))\}$$

- bucket: terminal subsets of record
- Optimized kd-tree
  - Goal: minimize the expected number of records examined
  - Adjust discriminating key number and partition value at internal node
  - Discriminator: key with largest spread in value
  - Partition: median of the discriminator key values
  - Produce balanced binary tree
- Running time: $O(\log N)$ (time to descend from root to terminal)

Nearest Neighbor Search (cont'd)

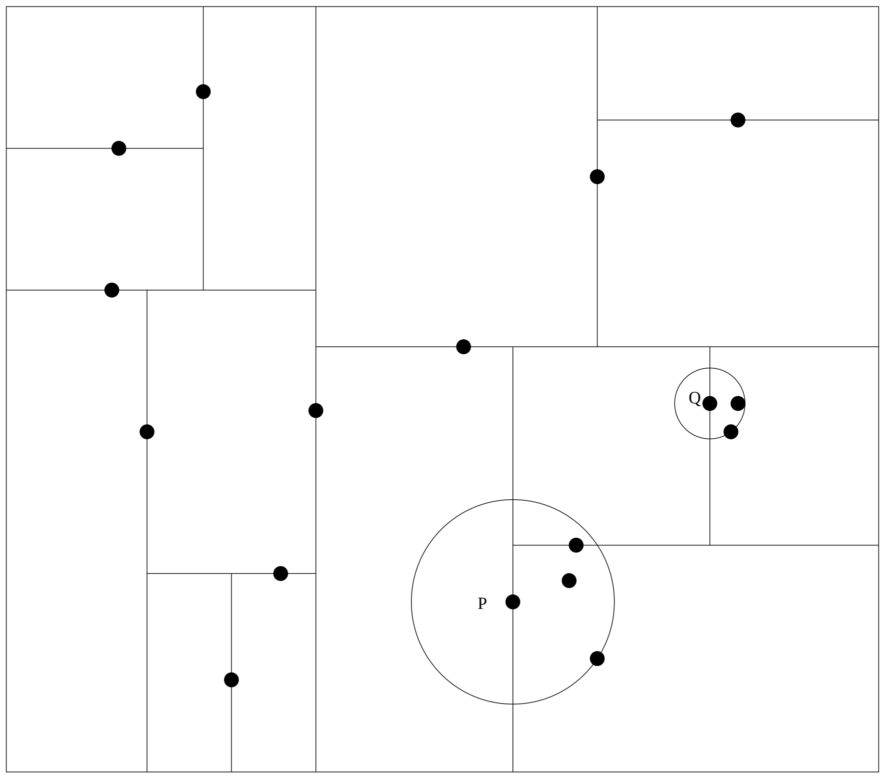

Figure 3: Searching for nearest neighbors of P and Q

Nearest Neighbor Search (cont'd)

- Definition
  - X[1:k] : key values of query node X
  - PQD[1:m] : priority queue of m closest distances. PQD[1] is the distance to the mth nearest neighbor so far encountered
  - PQR[1:m] : priority queue of record number
  - B+[1:k] : coordinate upper bounds
  - B-[1:k] : coordinate lower bounds
  - PARTITION: partition value of each node
- Initialize
  - PQD[1:m] $\leftarrow \infty$
  - B+[1:k] $\leftarrow \infty$
  - B-[1:k] $\leftarrow -\infty$

Nearest Neighbor Search Algorithm

```
SEARCH(P) /* Find m nearest neighbor of node X in tree rooted at P*/
begin
   if P is terminal then
      check records in P, update PQD,PQR
      if BALL-WITHIN-BOUNDS then done
      else return
   d ←DISC(P); p ←PARTITION(P)
   if X[d] < p then /* recursive call on closer son*/
      temp ← B+[d]; B+[d] ← p
      SEARCH(LOSON(P))
      B+[d] ← temp
   else
      temp ← B-[d]; B-[d] ← p
      SEARCH(HISON(P))
      B-[d] ← temp
```

Search Algorithm (cont'd)

```
/*continued from previous slide */
   if X[d] < p then /* recursive call on farther son, if necessary */
      temp ← B-[d]
      B-[d] ← p
      if BOUNDS-OVERLAP-BALL then SEARCH(HISON(P))
      B+[d] ← temp
   else
      temp ← B+[d]
      B+[d] ← p
      if BOUNDS-OVERLAP-BALL then SEARCH(HISON(P))
      B+[d] ← temp
   if BALL-WITHIN-BOUNDS then done else return
end
```

Search Algorithm (cont'd)

```
BALL-WITHIN-BOUNDS
begin
  for d ← 1 to k
      if COOR-DIST(d,X[d],B-[d]) ≤ PQD[1]
      or COOR-DIST(d,X[d],B+[d]) ≤ PQD[1] then return FALSE
  return TRUE
end
```

Search Algorithm (cont'd)

```
BOUNDS-OVERLAP-BALL
begin
   sum ←0
   for d ← 1 to k
        if X[d] < B-[d] then
           sum ← sum + COORD-DIST(d,X[d],B-[d])
           if DISSIM(sum) > PQD[1] then return TRUE
        else if X[d] > B+[d] then
           sum ← sum + COORD-DIST(d,X[d],B+[d])
           if DISSIM(sum) > PQD[1] then return TRUE
   return FALSE
end
```

Range Search

- Modifications to kd-tree
  - All data points will be stored in terminal node
- Let $Q(N)$ be the number of intersected regions. Then it satisfies

$$Q(N) = 2(Q(n/4), \text{ if n>1}$$

which yields $Q(N) = O(\sqrt{N})$.

- When the number of reported points is s, the running time is $O(\sqrt{N} + s)$.

Range Search Algorithm

```
RANGE-SEARCH(P,R) /* Reports all points in range R */
begin
   if P is terminal then
      Report the points in P if it is in R
   else
     if region(LOSON(P)) is fully contained in R then
        reportsubtree(LOSON(P))
     else
       if regison(LOSON(P)) intersects R then
          RANGE-SEARCH(HISON(P))
     if region(HISON(P)) is fully contained in R then
        reportsubtree(HISON(P))
     else
       if regison(HISON(P)) intersects R then
          RANGE-SEARCH(HISON(P))
end
```

Variants of kd-tree

- Optimized kd-tree
  - Select split dimension
  - Choose dimension with greatest spread
- Dimensions of kd-tree variation
  - Split dimension
  - Split position
  - Distance representation

VAM kd-tree

- Indexing structure for disk-based implementation
- Split along dimension with largest variance
  - More robust to outliers

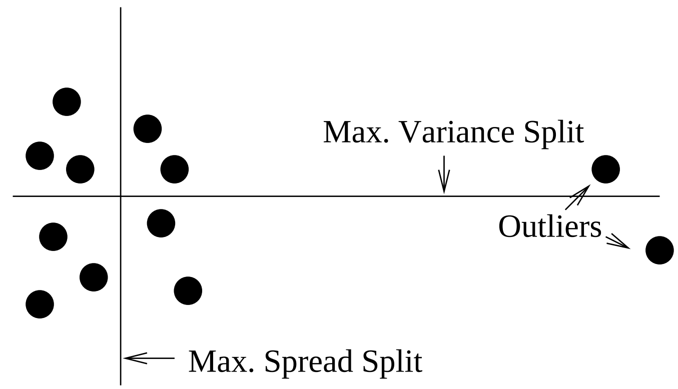

Figure 4: max. spread split vs. max. variance split

VAM kd-tree(cont'd)

- Split position

$$s_l = s_p - s_h = \begin{cases} \lfloor \frac{s_p}{2} \rfloor & \text{if } s_p \le 2b, \\ b\lfloor \frac{s_p}{2b} \rfloor & otherwise. \end{cases}$$

  with bucket size b.

- Guarantees minimum number of buckets

- Approximately median split

Application: Medical Image indexing

- Indexing MR images
- Shape based retrieval
- Similarity measure
  - For two curves C_1, C_2 and their length s_1, s_2, we have $s_2 = \phi(s_1)$.
  - Similarity measure $E(C_1, C_2)$ is

$$E(C_1, C_2) = \min_{\phi} E^*(C_1, C_2)$$

where

$$E^*(C_1, C_2) = \int_{C_1} \|T_1(s_1) - T_2(\phi(s_1))\|^2 ds_1$$