k-NN and range search with kd-trees

Dominika Kubániová

November 11, 2019

kd-tree

- Binary tree
- Data points $\in \mathbb{R}^k$ (my case k is max 4)
- Root wraps data into hyperrectange (cell)
- Hyperrectangle iteratively splitted by orthogonal planes

Non-optimized structures

```
struct kd_node {
  kd_node * left, * right;
  byte split_dim;
  float split_value;
  bool is_leaf;
  float * p;
}
```

Note: Some implementations store points in nodes and leaves, some only in leaves

Search problems

- k-NN search
- Range search (rectangular and spherical queries)

Construction of a kd-tree

```
fun construct_kdtree(dataset of size N in dimension k):
    if dataset size < M:
        return nullptr
    kd_node * node
    [split_dim, split_value] = choose_split()
    // ...
    data_left = {p from dataset | p[dim] < value}
    data_left = {p from dataset | p[dim] > value}
    node->left = construct_kdtree(data_left )
    node->right = construct_kdtree(data_right)
    return node

```

Data should be sorted in every dimension

How to choose splitting dimension?

- Does not affect correctness of the algorithm
- Can increase complexity of the search
- Balanced tree
- We want ratio of the longest to shortest side to be bounded
- The number of visited leaves in search should be small
- Bad dataset distribution can ruin everything

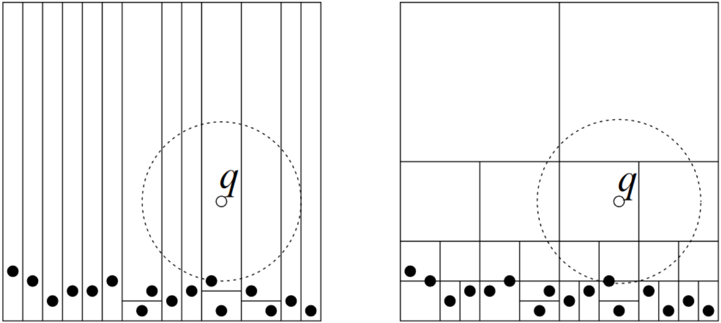
Dimension with maximal variance

- Splitting value = median
- Pros:
  - O(N) leaves and O(log N) depth
  - Probably every next chosen dimension will be different (not true when points have bad distribution)
- Cons:
  - Unbounded side length ratio (median does not need to be in the middle of the hyperrectangle)

Midpoint split

- Splitting value = half of the longest side of the hyperrectangle
- Balanced
- Pros:
  - Side length ratio bounded
- Cons:
  - Possibly a lot of empty nodes

Max variance and Midpoint

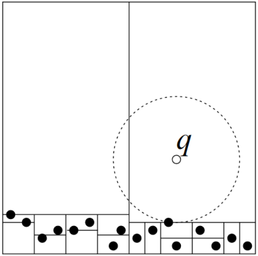


Source [2]

Large number of leaves is visited in k-NN search!

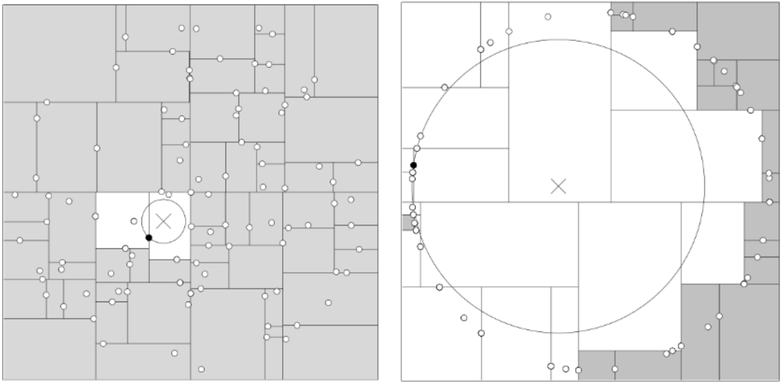
Sliding midpoint

- First step as midpoint
- If points lie on both sides of this split $\rightarrow$ continue
- Else $\rightarrow$ slide towards the closest point creating one NON-EMPTY leaf from it
- Pros:
  - O(N) leaves
- Cons:
  - What about its side length ratio? Is it balanced?



Source: [2]

It is observed that the depth of the tree is not that dominant factor than the number of visited leaves. Example:



Source: [2]

k-NN search (with PQ)

```
Point query
float knn_dist = INF
PQ kNNq
PQ q

fun process_leaf(n):
  for p in n->points:
    if (kNNq.size) < k:
      kNNq.push(p)
    else if dist(p, query) < knn_dist:
      kNNq.push(p)
      if kNNq.size() > k:
        kNNq.pop_last()
        knn_dist = dist(kNNq.last, query)
```

k-NN search (with PQ) cont.

```
q.push([root, 0.0])
while !q.empty():
  [n, dist] = q.pop()
  if dis < knn_dist:
    while (!n->is_leaf):
      off_split_plane = query[n->split_dim] - n->split_value
      if off_split_plane < 0:
        off_box_boundary = query[n->split_dim] - n->bound
        if off_box_boundary > 0:
          off_box_boundary = 0
        dist_right = dist - off_box_boundary^2 + off_split_plane^2
        q.push([n->right, dist_right])
        n = n->left
      else:
        // analogously
    process_leaf(n)
```

Range search (spherical query)

```
Point query
float r_sq
float offs[k] // initialized 0
vector<Point> in_range
stack s

fun process_leaf(n):
  for p in n->points:
    if dist(p, query) < r_sq:
      in_range.push(p)

```

Range search (spherical query) cont.

```
s.push([root, dist, offs])
while !s.empty():
  [n, dist, off] = s.pop()
  if n->is_leaf:
    process_leaf(n)
  else:
    old_off_split_plane = off[n->dim]
    new_off_split_plane = query[n->dim] - n->split_value
    if new_off_split_plane < 0:
      new_dist = dist - old_off_split_plane^2 + new_off_split_plane^2
      if new_dist < r_sq:
        new_off = off
        new_off[n->dim] = new_off_split_plane
          s.push([n->right, dist, new_off])
      s.push([n->left, dist, off])
    else:
      // analogously

```

Range search (rectangular query)

```
Point query
float size[k] // only halves stored
float offs[k] // initialized 0
vector<Point> in_range
stack s

fun process_leaf(n):
  for p in n->points:
    dim = n->dim
    for i in range(k):
      if query[dim] - size[dim] < p[dim] < query[dim] + size[dim]
        in_range.push(p)
      else:
        break

```

Range search (rectangular query) cont.

```
s.push(root)
while !s.empty():
  n = s.pop()

  if n->is_leaf:
    process_leaf(n)
  else:
    if n->value < query[n->dim] - size[n->dim]:
      s.push(n->right)

    else if n->value > query[n->dim] + size[n->dim]:
      s.push(n->left)

    else:
      s.push(n->left)
      s.push(n->right)

```

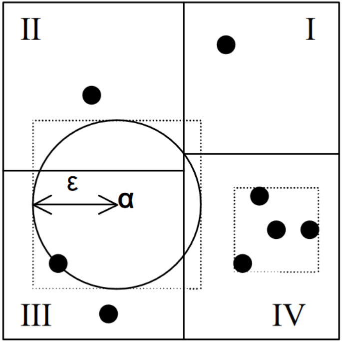
Optimized structures

```
struct kd_node {
    union {
        kd_node * left;
        float * p;
    }
    byte is_leaf_split_dim;
    float split_value;
}

```

Other simple heuristics

- Bounds-Overlap-Ball (BOB) test
  Effective when number of points stored in a leaf is high



Source: [2]

What next needs to be done

- Dataset generation (various distributions) in 2D-4D
- Dataset size: $10^3 - 10^8$
- Different implementation of priority queue
- Comparison (mainly with naïve algorithm)

References

- 1 Moore, Andrew. (2004). An Intoductory Tutorial on Kd-Trees.
- 2 Maneewongvatana, Songrit & Mount, David. (2000). It's Okay to Be Skinny, If Your Friends Are Fat.
- 3 Sample, Neal & Haines, Matthew & Arnold, Mark & Purcell, Timothy. (2001). Optimizing Search Strategies in k-d Trees.

Thank you for your attention