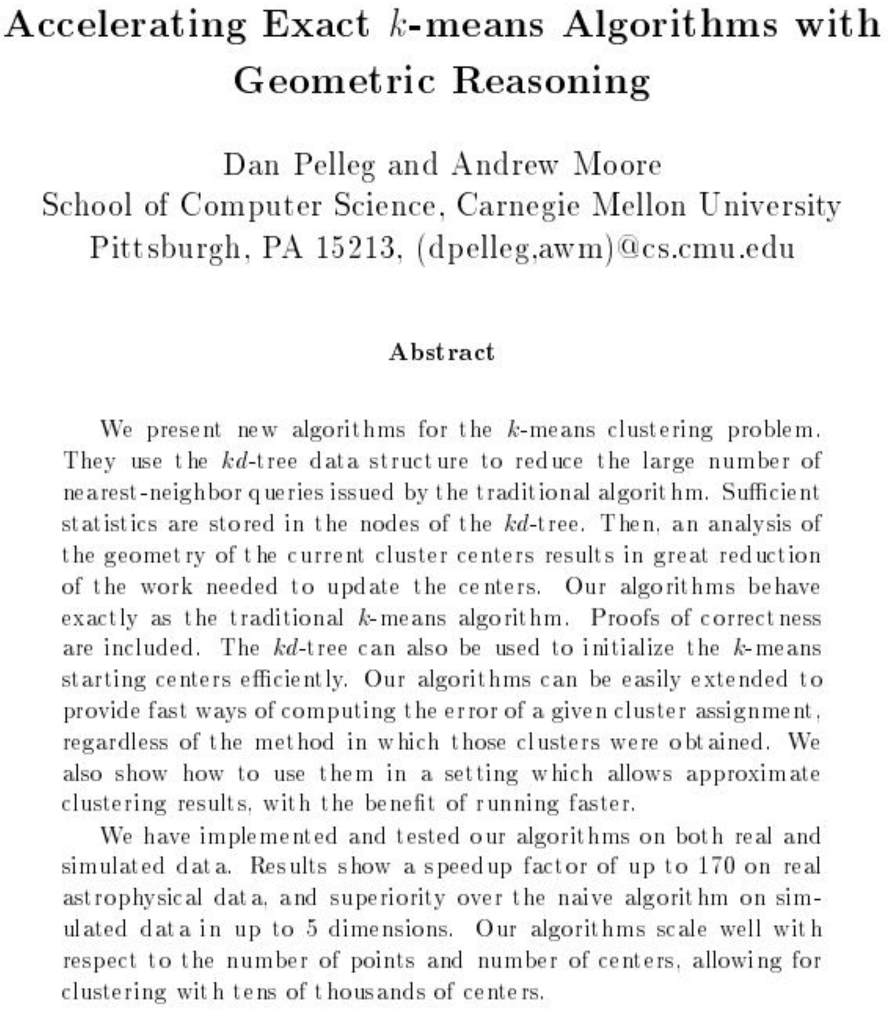
Accelerating Exact k-means Algorithms with Geometric Reasoning

Close reading of a paper

Stephen Huan

TJ Vision & Graphics Club, January 21, 2021

This paper:



Accelerating Exact k-means Algorithms with Geometric Reasoning

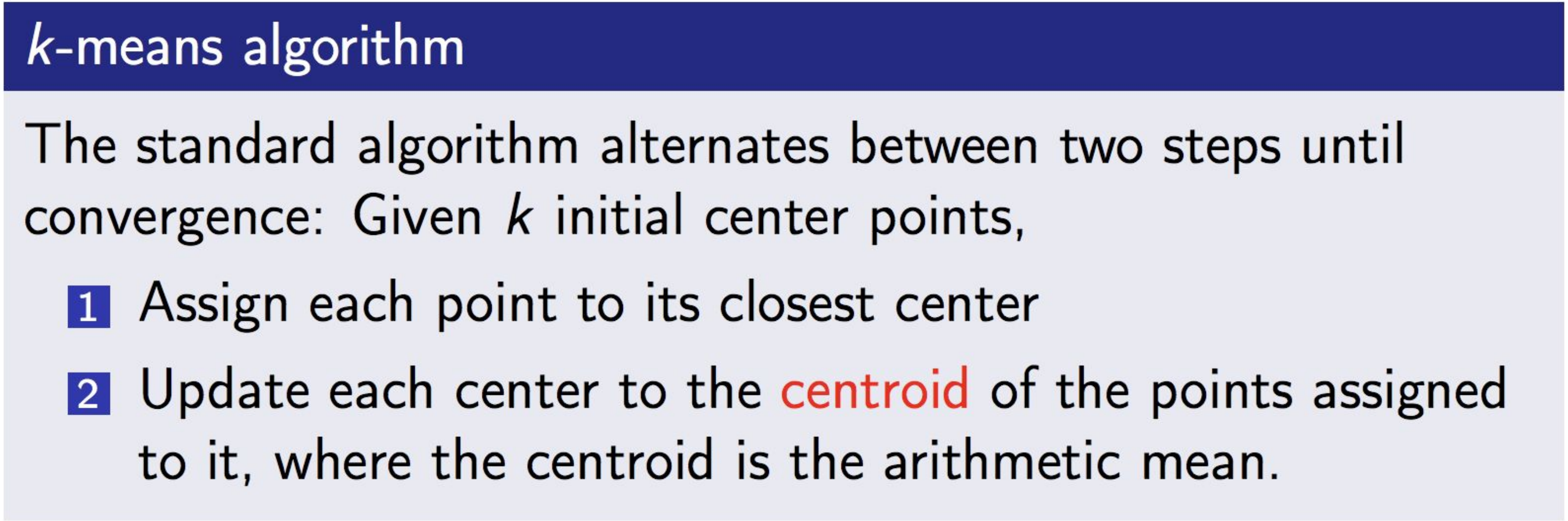
Dan Pelleg and Andrew Moore
School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213, (dpelleg,awm)@cs.cmu.edu

Abstract

We present new algorithms for the k-means clustering problem. They use the kd-tree data structure to reduce the large number of nearest-neighbor queries issued by the traditional algorithm. Sufficient statistics are stored in the nodes of the kd-tree. Then, an analysis of the geometry of the current cluster centers results in great reduction of the work needed to update the centers. Our algorithms behave exactly as the traditional k-means algorithm. Proofs of correctness are included. The kd-tree can also be used to initialize the k-means starting centers efficiently. Our algorithms can be easily extended to provide fast ways of computing the error of a given cluster assignment, regardless of the method in which those clusters were obtained. We also show how to use them in a setting which allows approximate clustering results, with the benefit of running faster.

We have implemented and tested our algorithms on both real and simulated data. Results show a speedup factor of up to 170 on real astrophysical data, and superiority over the naive algorithm on simulated data in up to 5 dimensions. Our algorithms scale well with respect to the number of points and number of centers, allowing for clustering with tens of thousands of centers.

Review: *k*-means



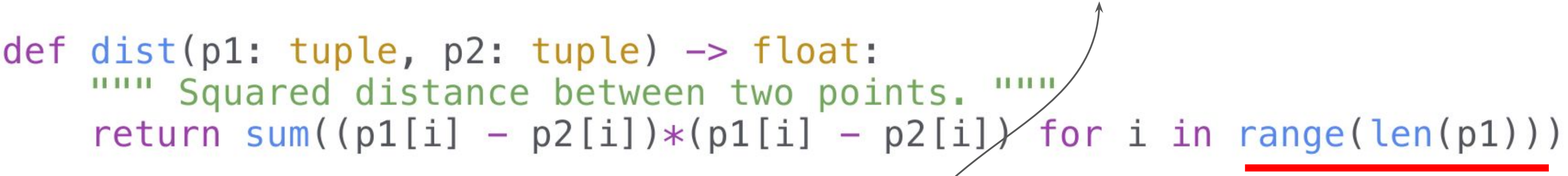
k-means algorithm

The standard algorithm alternates between two steps until convergence: Given k initial center points,

1. Assign each point to its closest center
2. Update each center to the centroid of the points assigned to it, where the centroid is the arithmetic mean.

k-means analysis

$$\mathtt{dist}(\vec{u},\vec{v}) = \|\vec{u}-\vec{v}\| = \sqrt{(u_1-v_1)^2+(u_2-v_2)^2+\dots}$$



```python
def dist(p1: tuple, p2: tuple) -> float:
    """ Squared distance between two points. """
    return sum((p1[i] - p2[i])*(p1[i] - p2[i]) for i in range(len(p1)))
```

I: Number of iterations (hard to know *a priori*)

```
For each iteration:
    For each point:
        Find its nearest center
```

```
For each center:
    Compute dist(point, center)
Pick minimum distance center
= O(KD)
```

= *O*(*INKD*), analyze per iteration -> *O*(*NKD*)

k-means analysis, cont.

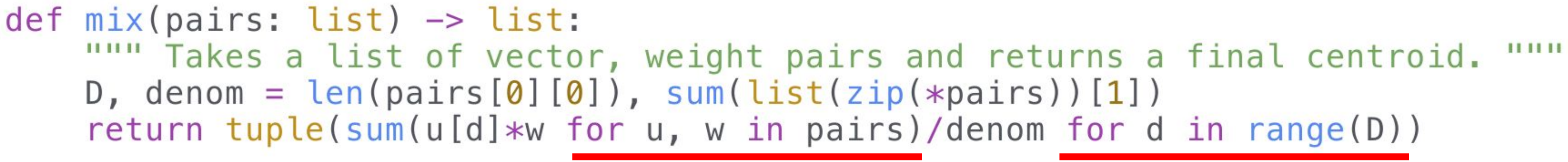
- Updating centers to centroid
- Assume we have point -> nearest center from last slide

For each center, go through its points and average.

$$E[S] = \frac{1}{\|S\|}\sum_{\vec{p}\in S}\vec{p}$$

Note: points may be weighted (duplicates)

[(0, 1), (5, 1), (2, 3), (0, 1), (2, 3)] -> [((0, 1), 2), ((2, 3), 2), ((5, 1), 1)]



```
def mix(pairs: list) -> list:
    """ Takes a list of vector, weight pairs and returns a final centroid. """
    D, denom = len(pairs[0][0]), sum(list(zip(*pairs))[1])
    return tuple(sum(u[d]*w for u, w in pairs)/denom for d in range(D))
```

= $D(S_1 + S_2 + \ldots + S_n) = O(DN)$ -> time dominated by assignment

Consider the following...

$N = 60{,}000$ pixels

$K = 256$ (8-bit image)

$I = 111$ iterations

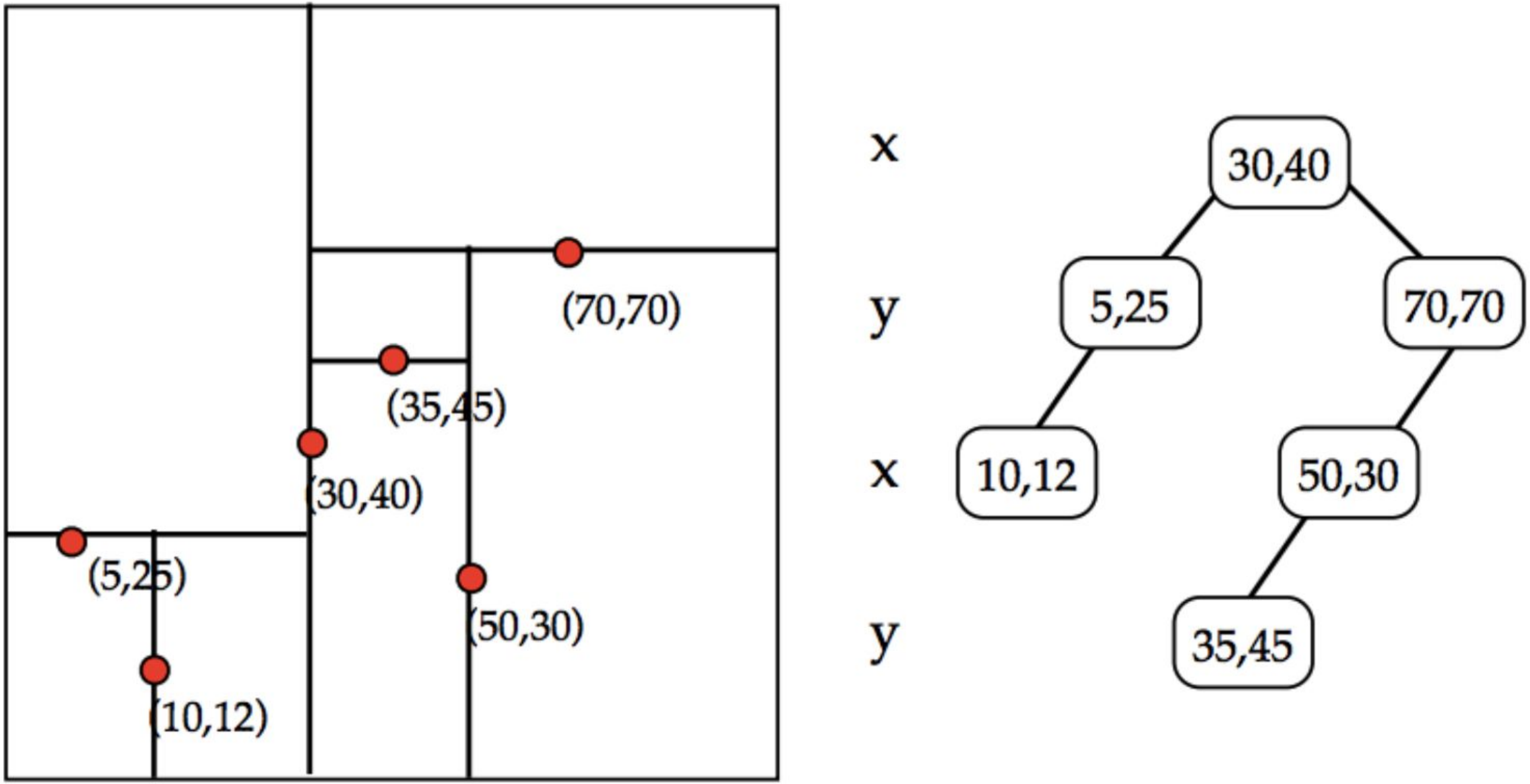
$NKI = 60{,}000*256*111 > 1.7$ billion

$1.7*10^9/1$ MHz = 1700 seconds/60 = $\boxed{\text{30 minutes!}}$

Can we do better?
Yes, kd-trees!

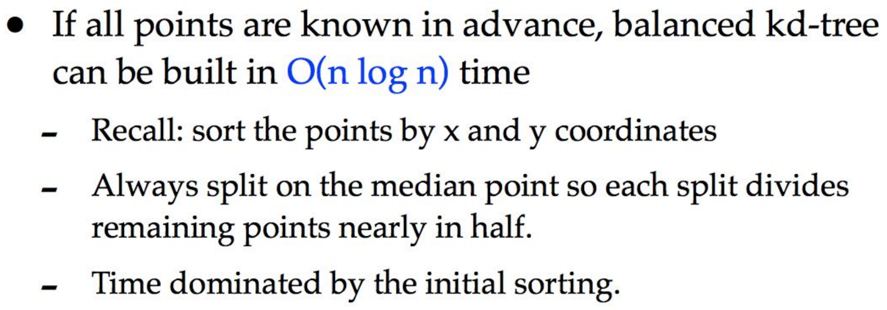
kd-tree, review

- BST insert: compare value against root’s value
  - If less, recur on the left subtree
  - If greater, on the right subtree.
- kd-tree: similar, but node holds *point* (vector), not scalar
  - Need some way of comparing
  - Pick an arbitrary *cutting dimension* to compare
  - Cycle through cutting dimensions
- Note: $x_{cd} = k$ defines a *hyperplane* (high dimensional plane)


(70,70)
(35,45)
(30,40)
(5,25)
(50,30)
(10,12)
x
y
x
y
30,40
5,25
70,70
10,12
50,30
35,45

Building a kd-tree

- See previous lecture
- $O(d^2 n \log n)$ “pre-sort”
  - $O(d\ n \log n)$ with careful representation
- $O(n \log n)$ “median” if you read too much *Introduction to Algorithms*
  - *Irrespective of dimensionality!*
  - High constant factor, however...
- tl;dr fast



- If all points are known in advance, balanced kd-tree can be built in O(n log n) time
  - Recall: sort the points by x and y coordinates
  - Always split on the median point so each split divides remaining points nearly in half.
  - Time dominated by the initial sorting.

Silly CMU, pre-sort is O(d n log n)!

mrkd-tree

- “Multi-resolution kd-tree”
- Additional data at each node
  - Bounding box
  - Number of points
  - Centroid
- Can we just annotate a regular kd-tree?

```python
def tighten(self, t: "KdNode"=None) -> None:
    """ Tighten bounding boxes in O(nd). """
    if t is None: t = self # called with None, set to the root
    l, r               = t.child[0], t.child[1]
    # recur on children
    if l is not None: self.tighten(l)
    if r is not None: self.tighten(r)


    if l is None and r is None:  # leaf node, box is just the singular point


    elif l is None or r is None: # one child, inherit box of child
        child = l if l is not None else r



    else:                        # two children, combine boxes



```

Using the kd-tree

- Can be used to speed up nearest neighbor queries
- ~$O(\log n + 2^d)$
  - log n from tree search
  - 2^d from nodes “near” the target point
- Build kd-tree on centers and use for queries
- Is this really the best approach?

Comparison

kd-tree on centers		kd-tree on points
● Few points, $K \leq N$ ○ In practice, $K < 10$ ○ Can maybe save a K factor vs log K	<	● Many points ○ In practice, $N > 100{,}000$ ○ Can save time as it’s based on N
● Dynamic, centers change every iteration ○ Need to build a new kd-tree	<	● Static, points are the same ○ Can build once and recycle
● Easily convert existing algorithm	>	● Requires a bit of adjustment

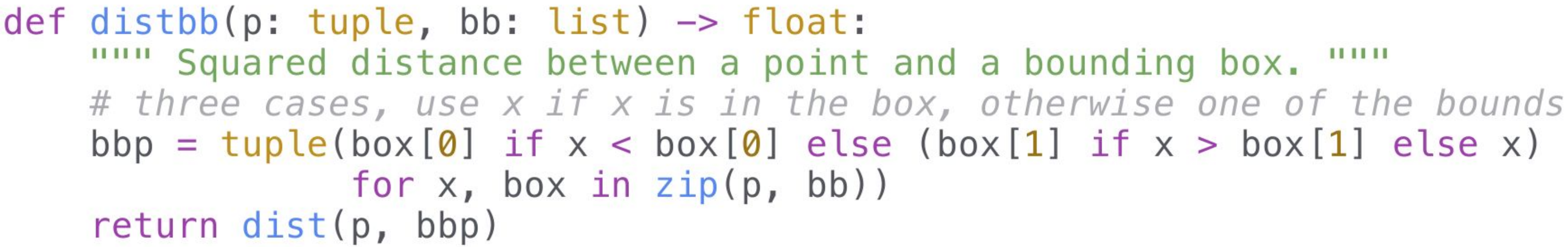
Clearly, we should build a kd-tree on the *points*, not the centers!
How?

This is where the paper comes in...

Ownership

- Let C be the list of centers, len(C) = K
- Center c “owns” a hyper-rectangle h (kd-node and descendants)
    - Every point in h is closer to c than to any other center
    - “Every point” is really “every point”; not just the ones in the kd-tree
    - Obviously, all points assigned to c
- If we can show c owns h, no need to check points in h
- How to show ownership?

Point to Hyper-rectangle



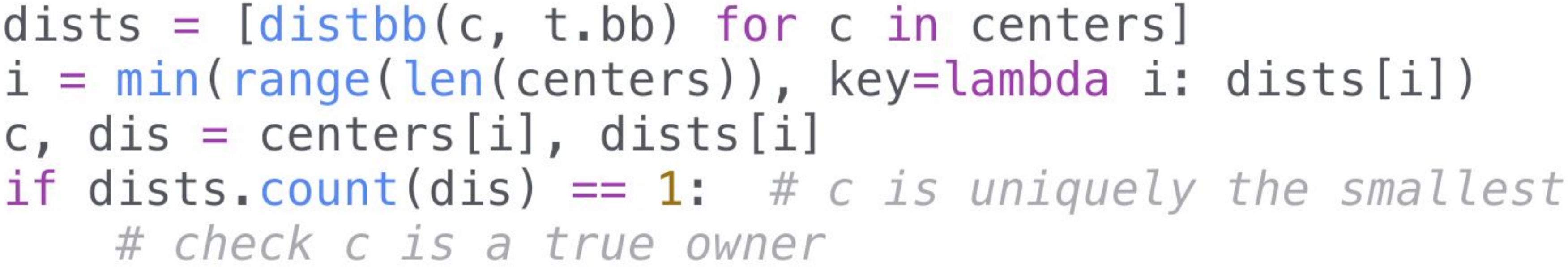
```python
def distbb(p: tuple, bb: list) -> float:
    """ Squared distance between a point and a bounding box. """
    # three cases, use x if x is in the box, otherwise one of the bounds
    bbp = tuple(box[0] if x < box[0] else (box[1] if x > box[1] else x)
                for x, box in zip(p, bb))
    return dist(p, bbp)
```

- Find the closest point to *p* in *bb, q*
- dist(*p, q*)

1. *p*[*d*] < *bb*[*d*][0] -> Take *bb*[*d*][0]
2. *p*[*d*] > *bb*[*d*][1] -> Take *bb*[*d*][1]
3. *bb*[*d*][0] ≤ *p*[*d*] ≤ *bb*[*d*][1] -> Take *p*[*d*]

Condition #1

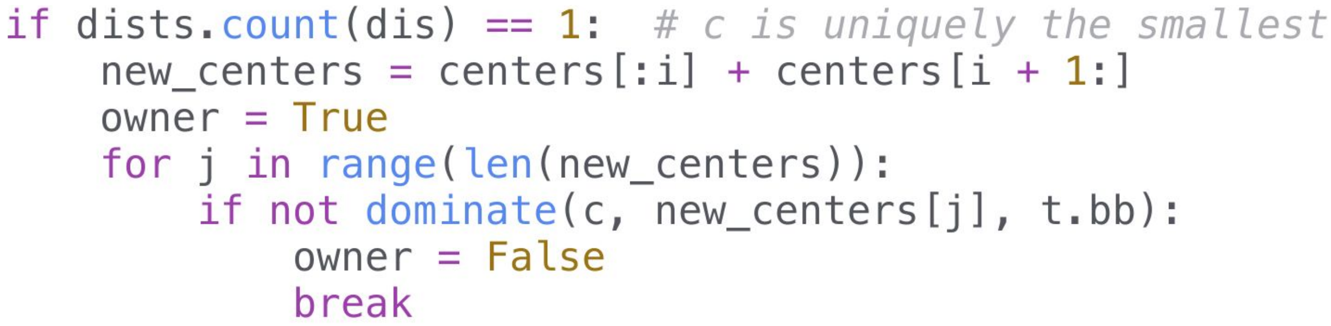
- *c* must have the minimum distance to *h*
  - If not true, contradiction!
  - Suppose other center *c′* with dist(*c′*, *h*) < dist(*c*, *h*)
  - Note that dist(*c*, *h*) *constructive*, finds closest point
  - Therefore there's a point closer to *c′* than *c*
  - Contradicts definition of owner
- Must also be unique
  - Can't have two points with minimum distance
  - Neither owner



```
dists = [distbb(c, t.bb) for c in centers]
i = min(range(len(centers)), key=lambda i: dists[i])
c, dis = centers[i], dists[i]
if dists.count(dis) == 1:  # c is uniquely the smallest
    # check c is a true owner
```

Condition #2

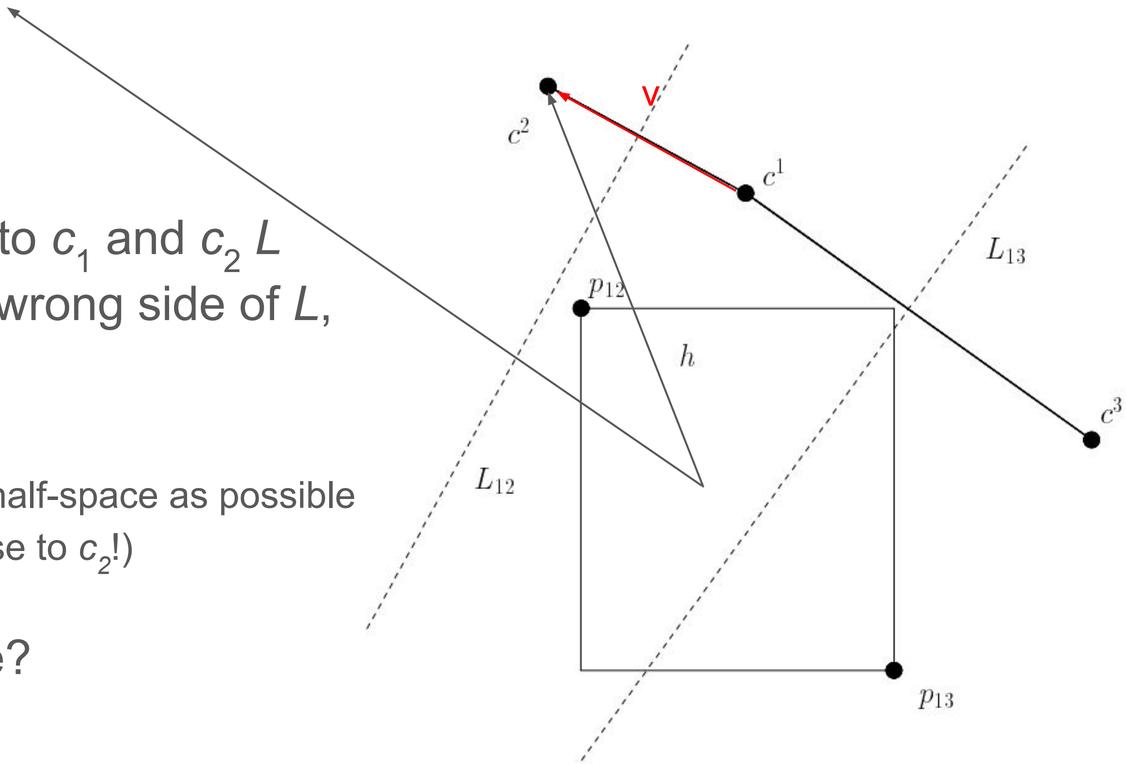
- Condition #1 narrows down the owner candidates to either 1 or 0
- If there isn’t an owner, we’re going to recur on the children
  - Left and right hyper-rectangles
- Let’s say we have a candidate c_1
- Is c_1 a true owner?
  - Check against all other centers, call one of them c_2
  - Is every point in *h* closer to c_1 than c_2? c_1 “dominates” c_2



```
if dists.count(dis) == 1:  # c is uniquely the smallest
    new_centers = centers[:i] + centers[i + 1:]
    owner = True
    for j in range(len(new_centers)):
        if not dominate(c, new_centers[j], t.bb):
            owner = False
            break
```

Domination

- Line of points equidistant to c_1 and c_2 L
- If there's a point p on the wrong side of L, c_1 doesn't dominate c_2
- Optimization problem:
  - Objective: Be as far in the half-space as possible (not the same as being close to c_2!)
  - Constraint: Contained in h
- How to measure objective?
- Look at $v = c_2 - c_1$
- v dot p
- Linear programming!



Linear Programming

- Linear objective and linear constraints
- Yes! Objective *v* dot *p*, *v* is a constant
  - $= v_1 p_1 + v_2 p_2 + \dots + v_D p_D$, linear function of *p*
- Constraints of the form
  - $h[d][0] \le p[d] \le h[d][1]$ for *d* between 0 … *D* - 1
- Algorithms for linear programming irrelevant
  - Mathematical ideas relevant

Solving Linear Programs

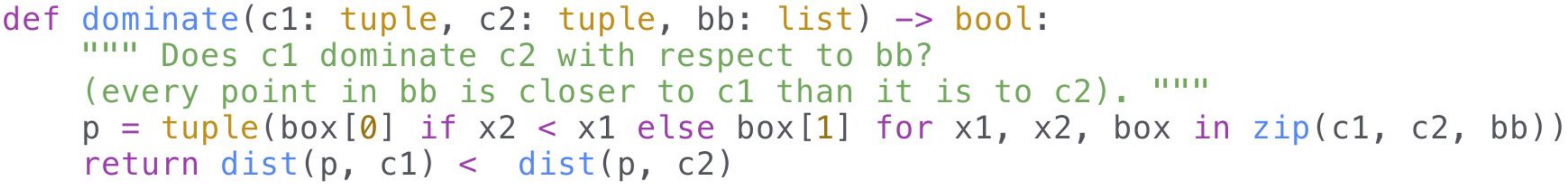
- Feasible value: satisfies constraints
- Region of feasible values: *simplex*
- Optimal value must be at a corner of the simplex
- *Proof.* Start at arbitrary point and follow the gradient (vector) until we reach the exterior. From here, move along edges. All edges are orthogonal to the gradient or not.
  (1) If all are, then our function value can't change, current value is optimal. Move along an edge to a corner.
  (2) If all aren't, move along an edge in the right direction. Must eventually run out of edges, at which point (1) applies.

(moving in direction of gradient increases function value)

(I don't know enough math to justify this)

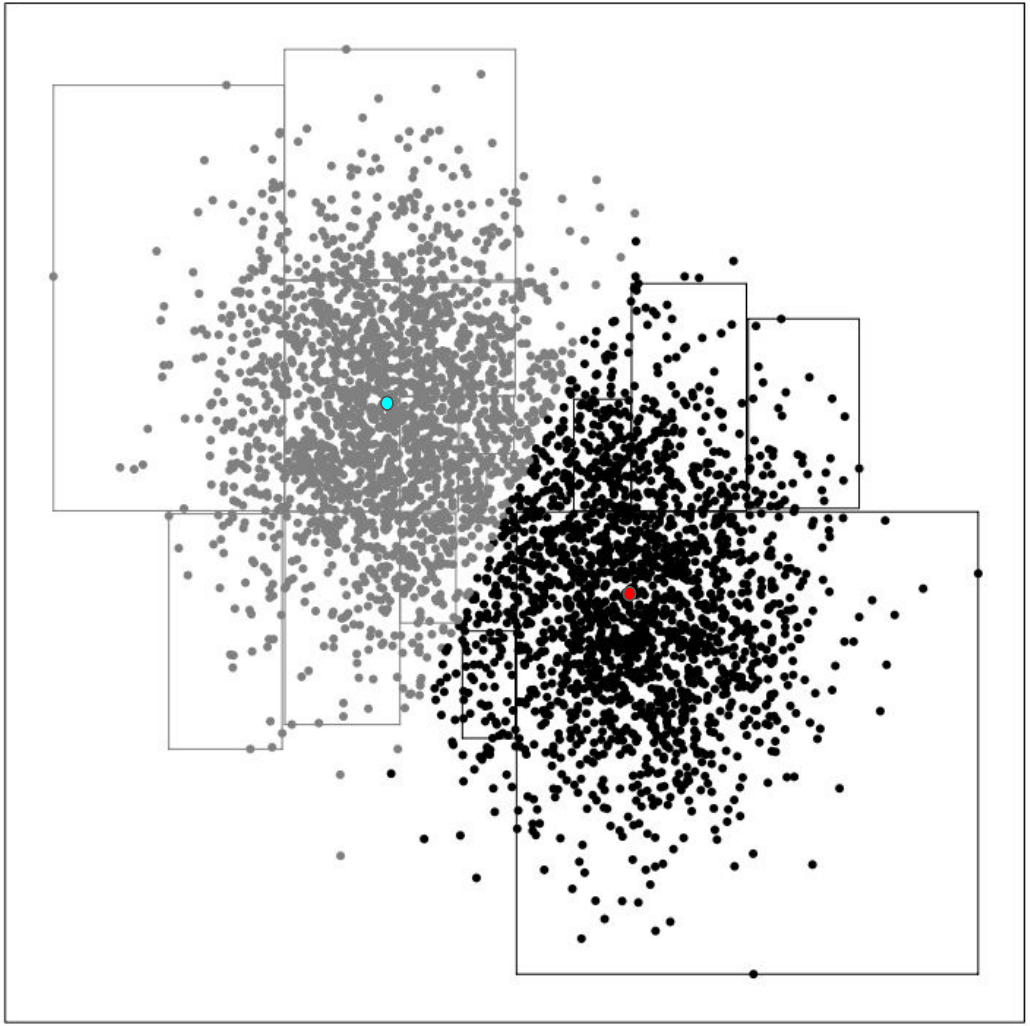
Back to Domination

- Gradient is simply $v = c_2 - c_1$, direction towards c_2's side of the line
- Simplex is the hyper-rectangle *h*
- Optimal value occurs at corners of simplex = corners of *h*
- Can efficiently compute best point *p*
    - Take max or min depending on which is in the direction of $c_2 - c_1$

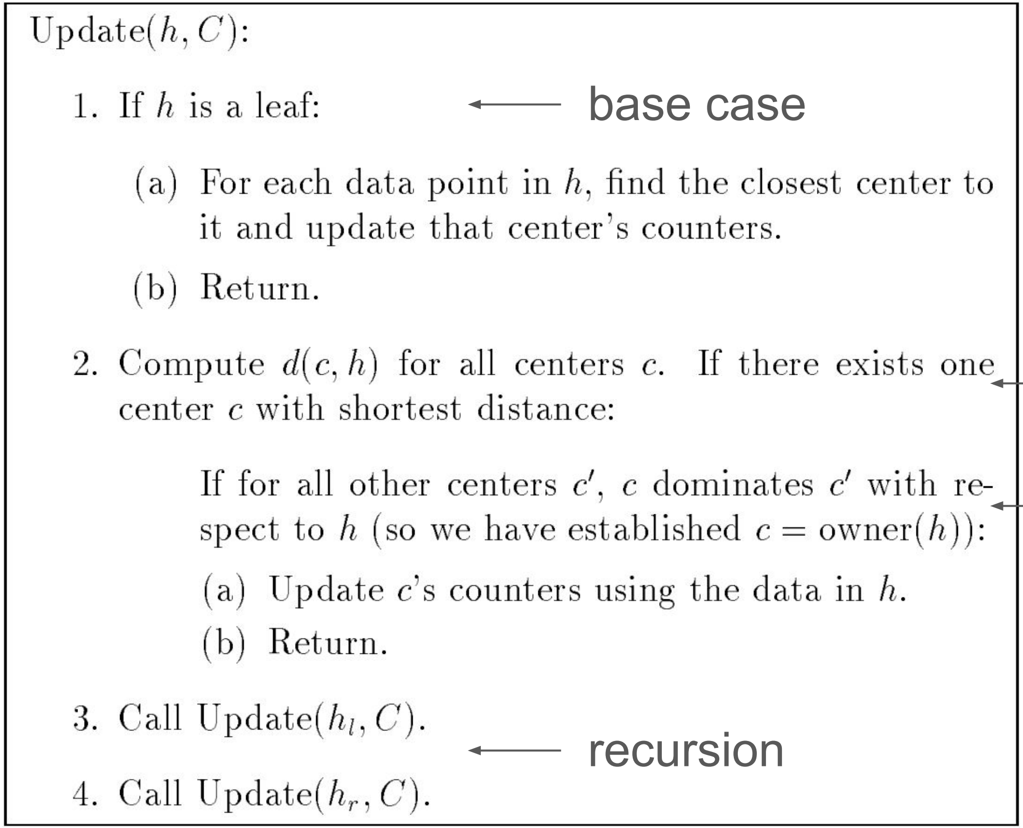


```python
def dominate(c1: tuple, c2: tuple, bb: list) -> bool:
    """ Does c1 dominate c2 with respect to bb?
    (every point in bb is closer to c1 than it is to c2). """
    p = tuple(box[0] if x2 < x1 else box[1] for x1, x2, box in zip(c1, c2, bb))
    return dist(p, c1) <  dist(p, c2)
```

Example



Putting it Together



```
Update($h, C$):

    1. If $h$ is a leaf:
        (a) For each data point in $h$, find the closest center to
            it and update that center's counters.
        (b) Return.

    2. Compute $d(c, h)$ for all centers $c$. If there exists one
       center $c$ with shortest distance:

            If for all other centers $c'$, $c$ dominates $c'$ with re-
            spect to $h$ (so we have established $c = \text{owner}(h)$):
            (a) Update $c$'s counters using the data in $h$.
            (b) Return.

    3. Call Update($h_l, C$).

    4. Call Update($h_r, C$).
```

base case

"For each data point in *h*" when *h* is a leaf and leaves of a kd-tree contain exactly one point? ¯_(ツ)_/¯

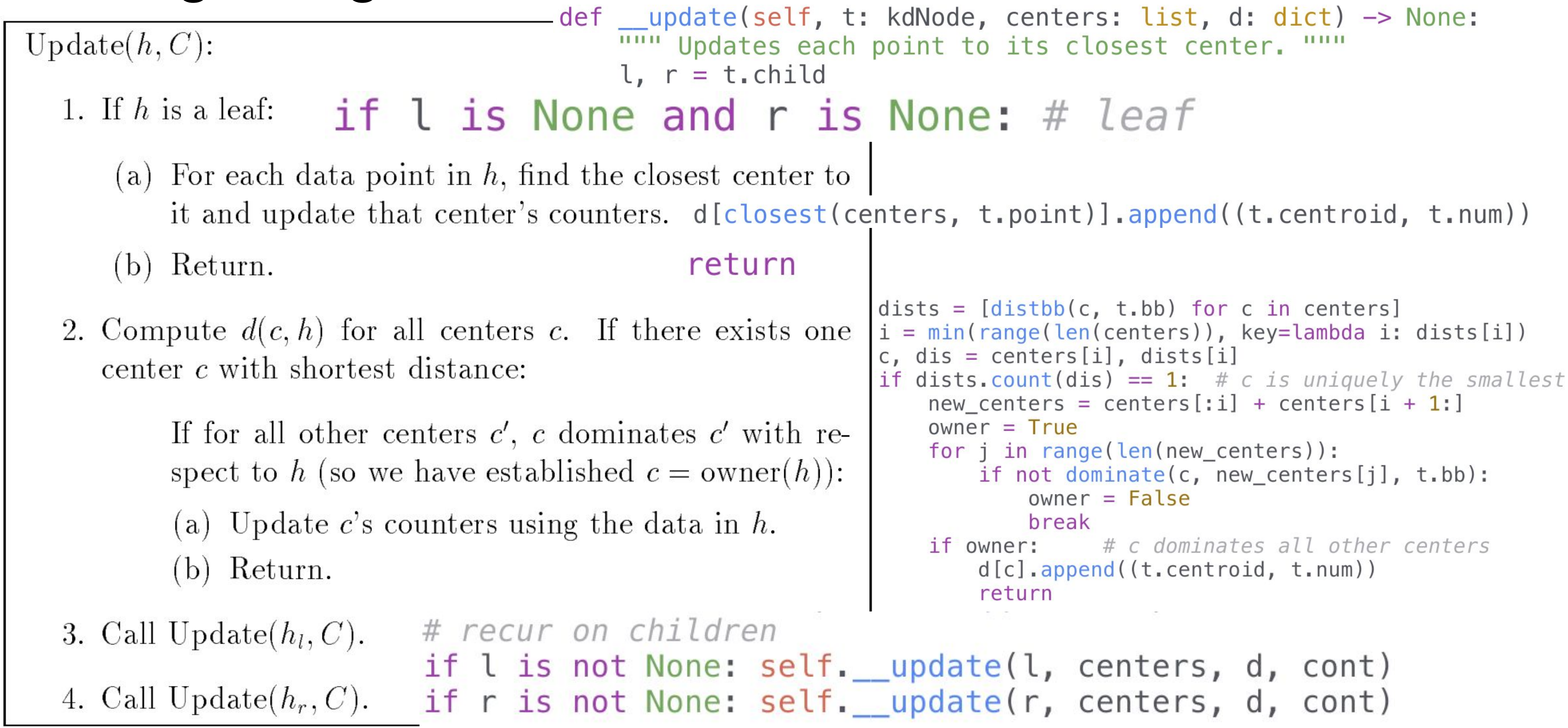
condition #1

condition #2

This is where the algorithm saves time!

recursion

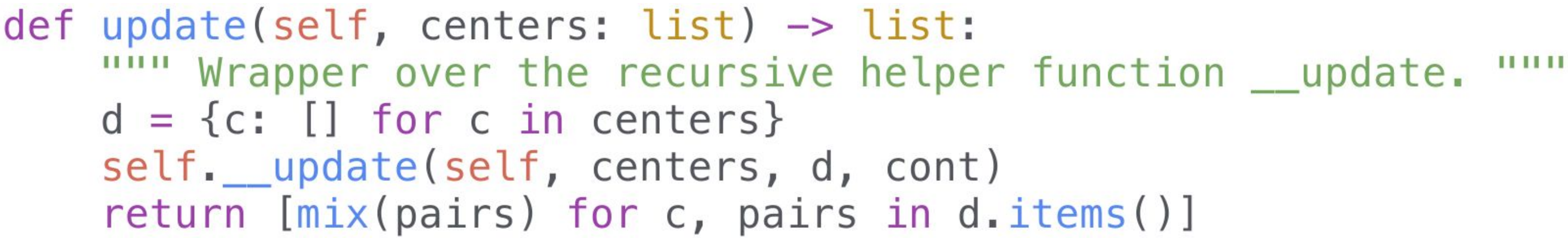
Putting it Together, Code



```
Update($h, C$):
    1. If $h$ is a leaf:
        (a) For each data point in $h$, find the closest center to it and update that center's counters.
        (b) Return.
    2. Compute $d(c, h)$ for all centers $c$. If there exists one center $c$ with shortest distance:
        If for all other centers $c'$, $c$ dominates $c'$ with respect to $h$ (so we have established $c = \text{owner}(h)$):
            (a) Update $c$'s counters using the data in $h$.
            (b) Return.
    3. Call Update($h_l, C$).
    4. Call Update($h_r, C$).
```

```python
def __update(self, t: kdNode, centers: list, d: dict) -> None:
    """ Updates each point to its closest center. """
    l, r = t.child
    if l is None and r is None: # leaf
        d[closest(centers, t.point)].append((t.centroid, t.num))
        return
    dists = [distbb(c, t.bb) for c in centers]
    i = min(range(len(centers)), key=lambda i: dists[i])
    c, dis = centers[i], dists[i]
    if dists.count(dis) == 1:  # c is uniquely the smallest
        new_centers = centers[:i] + centers[i + 1:]
        owner = True
        for j in range(len(new_centers)):
            if not dominate(c, new_centers[j], t.bb):
                owner = False
                break
        if owner:     # c dominates all other centers
            d[c].append((t.centroid, t.num))
            return
    # recur on children
    if l is not None: self.__update(l, centers, d, cont)
    if r is not None: self.__update(r, centers, d, cont)
```

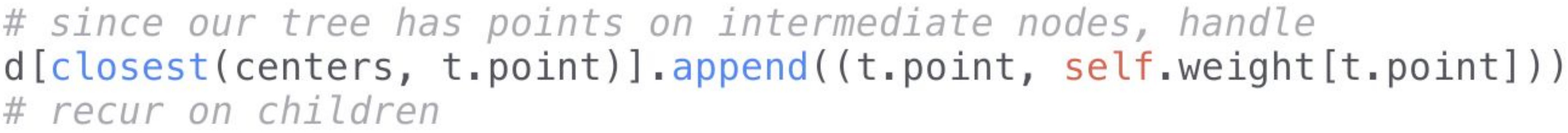
Finishing Touches



```python
def update(self, centers: list) -> list:
    """ Wrapper over the recursive helper function __update. """
    d = {c: [] for c in centers}
    self.__update(self, centers, d, cont)
    return [mix(pairs) for c, pairs in d.items()]
```

One Small Adjustment...

- Paper’s kd-tree stores points at leaves
- In our kd-tree, intermediate nodes also have points
- Simply add its point with brute force before recursion

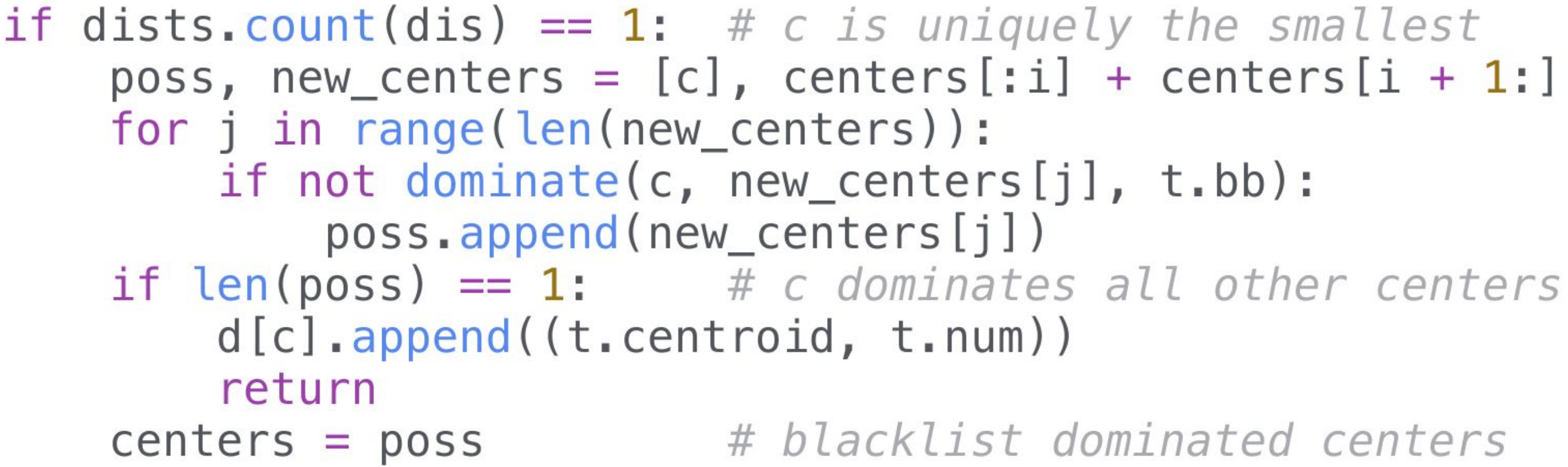


```
# since our tree has points on intermediate nodes, handle
d[closest(centers, t.point)].append((t.point, self.weight[t.point]))
# recur on children
```

Blacklisting

- Suppose c_1 is an owner candidate
- c_1 dominates c_2 but isn’t an owner overall
- Normally, recur and have to check again
- But c_2 can’t be an owner anymore!
  - If we recur on h', every point in h' is in h
  - Thus, c_1 still dominates c_2
  - c_2 can’t be an owner
- Saves time in multiple ways!
  - Checking ownership
  - If at a leaf, less centers to check

Blacklisting, code



```python
if dists.count(dis) == 1:  # c is uniquely the smallest
    poss, new_centers = [c], centers[:i] + centers[i + 1:]
    for j in range(len(new_centers)):
        if not dominate(c, new_centers[j], t.bb):
            poss.append(new_centers[j])
    if len(poss) == 1:     # c dominates all other centers
        d[c].append((t.centroid, t.num))
        return
    centers = poss         # blacklist dominated centers
```

Approximate Pruning

- Speed up the algorithm by cutting off recursion
- Heuristic for when this is “safe”

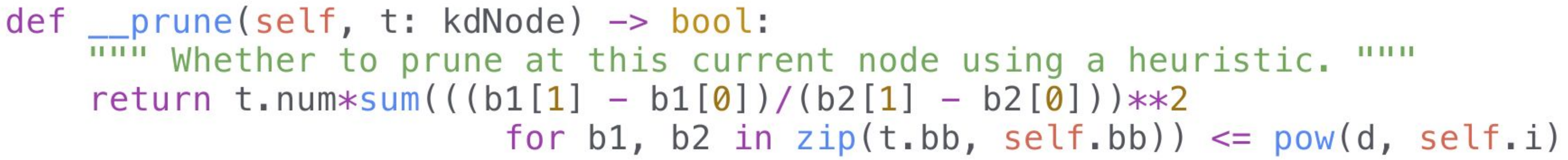
$$n \cdot \sum_{j=1}^{M} \left(\frac{\mathrm{width}(h)_j}{\mathrm{width}(U)_j} \right)^2 \leq d^i$$

- Few points, small rectangle, early on = less error
- Divide points between current centers

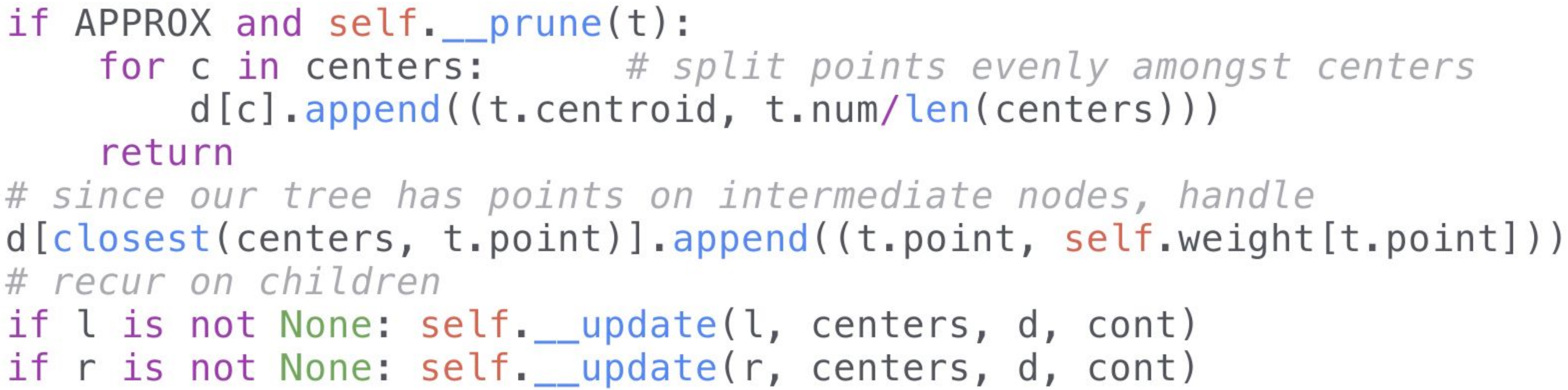
h: hyper-rectangle
n: number of points
U: hyper-rectangle
of all points
M: number of dimensions
i: iteration number
d: parameter,
recommended 0.8

Pruning, Code

$$n \cdot \sum_{j=1}^{M} \left(\frac{\mathrm{width}(h)_j}{\mathrm{width}(U)_j} \right)^2 \le d^i$$



```python
def __prune(self, t: kdNode) -> bool:
    """ Whether to prune at this current node using a heuristic. """
    return t.num*sum(((b1[1] - b1[0])/(b2[1] - b2[0]))**2
                        for b1, b2 in zip(t.bb, self.bb)) <= pow(d, self.i)
```



```python
if APPROX and self.__prune(t):
    for c in centers:      # split points evenly amongst centers
        d[c].append((t.centroid, t.num/len(centers)))
    return
# since our tree has points on intermediate nodes, handle
d[closest(centers, t.point)].append((t.point, self.weight[t.point]))
# recur on children
if l is not None: self.__update(l, centers, d, cont)
if r is not None: self.__update(r, centers, d, cont)
```

Paper is done!
How’s the results?

Benchmarks, Dataset

hanekawa.png: https://stephen-huan.github.io/assets/images/hanekawa.png

```
                Size: 570 x 517

              Pixels: 294690

Distinct pixel count: 59809

   Most common pixel: (250, 228, 213) => 19377
```

Benchmarks, Result

testcases/hanekawa.png, K = 8

k-means++ initialization scheme

	PyPy	Python
kd-tree	**5.220s**	**6.420s**
naive	8.379s	17.164s

Benchmarks, Results

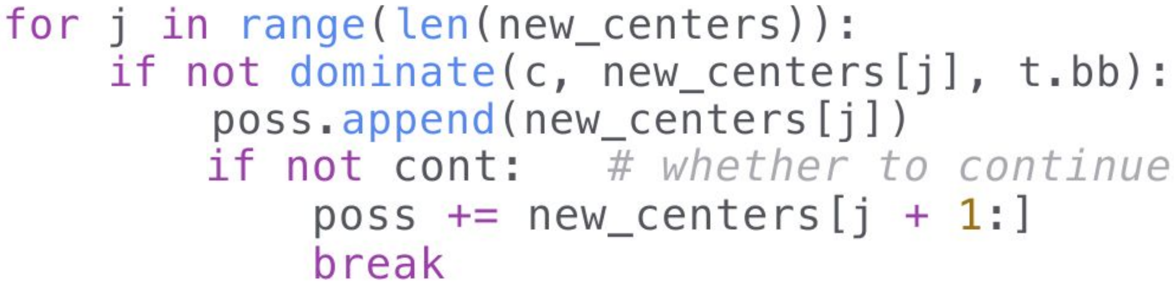
```
testcases/hanekawa.png, K = 256, PyPy
  naive (run on all pixels): 35m 22s073
  naive   (distinct pixels):  8m 38s343
kd-tree   median ( n log n):  3m 52s540
kd-tree pre-sort (dn log n):  3m 48s248
kd-tree geometric reasoning:  1m 47s550
kd-tree approximate (d=0.8):  2m 18s735 ←
```

Slower!
Implementation mistake? ¯_(ツ)_/¯
Worse centers -> more iterations
Overhead in calculating prune

Extensions

Continue Parameter

- Trying to show *c* is an owner
- If it isn't, should we break?
  - Don't waste time comparing against other centers
  - But continuing could save time if we removed them with blacklisting
- Usually faster to continue than not continue
  - 4m 0s716 without continue vs 26s802 with continue

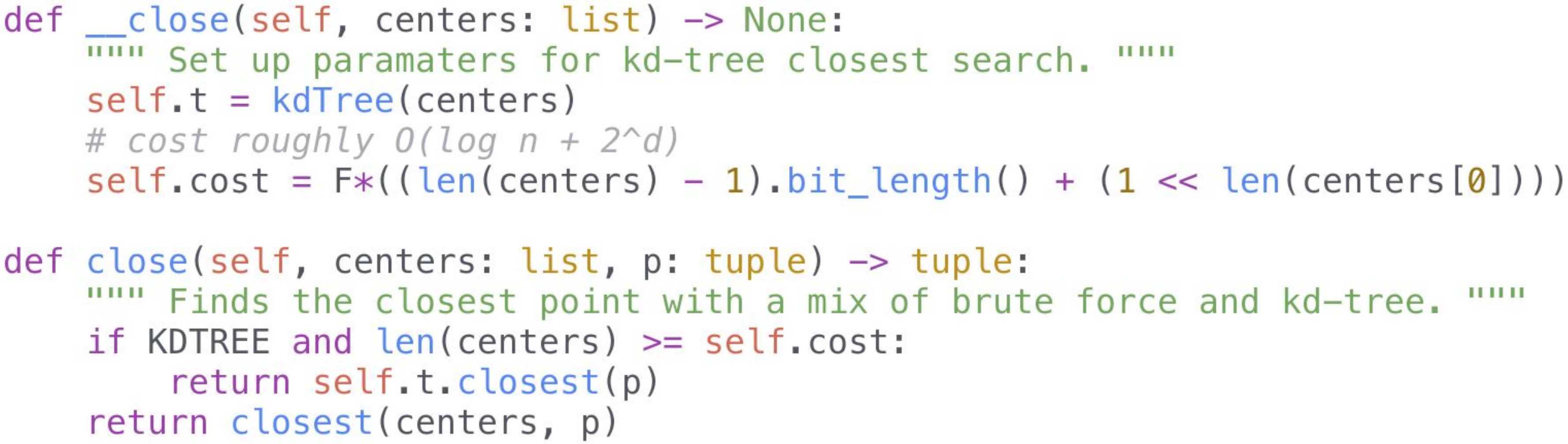


```
for j in range(len(new_centers)):
    if not dominate(c, new_centers[j], t.bb):
        poss.append(new_centers[j])
        if not cont:   # whether to continue
            poss += new_centers[j + 1:]
            break
```

kd-tree for Leaf Nodes

- $O(0)$ if point is pruned
- $O(K)$ if it isn't (leaf or intermediate)
- Can we speed the worst case up?
- Build a kd-tree on the centers again!
- ~ $O(\log K)$
- Doesn't work in practice, blacklisting removes too many centers
  - Not $O(K)$, actually O(average # of centers)

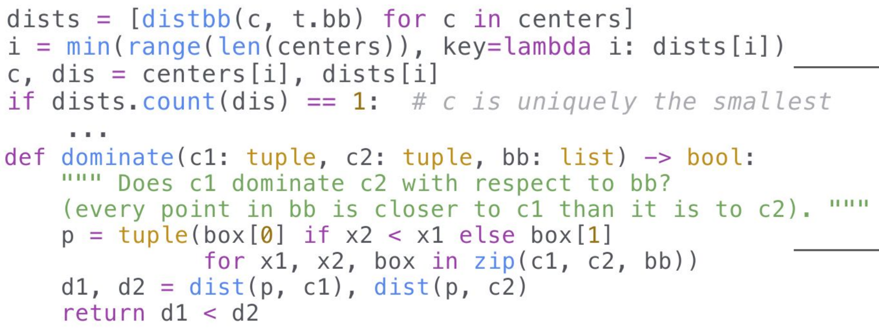
kd-tree for Leaf Nodes, Code



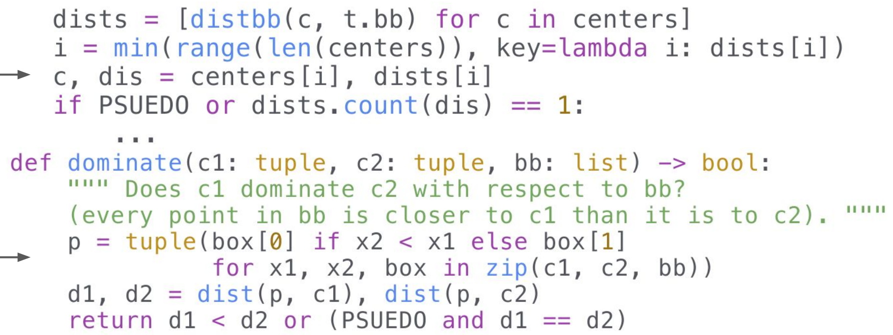
```python
def __close(self, centers: list) -> None:
    """ Set up paramaters for kd-tree closest search. """
    self.t = kdTree(centers)
    # cost roughly O(log n + 2^d)
    self.cost = F*((len(centers) - 1).bit_length() + (1 << len(centers[0])))

def close(self, centers: list, p: tuple) -> tuple:
    """ Finds the closest point with a mix of brute force and kd-tree. """
    if KDTREE and len(centers) >= self.cost:
        return self.t.closest(p)
    return closest(centers, p)
```

“Pseudo”-owner

- Original paper restrictions seem excessive
    - Does *c really* need to be the unique minimum?
    - Does *c really* need to completely dominate?
- No, can relax the conditions
    - Allowed to arbitrarily choose between equidistant centers
    - Maintains same quality and sometimes the same centers
- Reduces number of candidates and improves pruning



```
dists = [distbb(c, t.bb) for c in centers]
i = min(range(len(centers)), key=lambda i: dists[i])
c, dis = centers[i], dists[i]
if dists.count(dis) == 1:  # c is uniquely the smallest
    ...
def dominate(c1: tuple, c2: tuple, bb: list) -> bool:
    """ Does c1 dominate c2 with respect to bb?
    (every point in bb is closer to c1 than it is to c2). """
    p = tuple(box[0] if x2 < x1 else box[1]
              for x1, x2, box in zip(c1, c2, bb))
    d1, d2 = dist(p, c1), dist(p, c2)
    return d1 < d2
```



```
dists = [distbb(c, t.bb) for c in centers]
i = min(range(len(centers)), key=lambda i: dists[i])
c, dis = centers[i], dists[i]
if PSUEDO or dists.count(dis) == 1:
    ...
def dominate(c1: tuple, c2: tuple, bb: list) -> bool:
    """ Does c1 dominate c2 with respect to bb?
    (every point in bb is closer to c1 than it is to c2). """
    p = tuple(box[0] if x2 < x1 else box[1]
              for x1, x2, box in zip(c1, c2, bb))
    d1, d2 = dist(p, c1), dist(p, c2)
    return d1 < d2 or (PSUEDO and d1 == d2)
```

Benchmarks, Results

```
testcases/hanekawa.png, K = 256, PyPy
  naive (run on all pixels): 35m 22s073
  naive   (distinct pixels):  8m 38s343
kd-tree   median ( n log n):  3m 52s540
kd-tree pre-sort (dn log n):  3m 48s248
kd-tree approximate (d=0.8):  2m 18s735
kd-tree geometric reasoning:  1m 47s550
              w/out k-means++:     32s732
             w/ pseudo owners:     26s802
```

4x improvement (5x less pixels)

5x improvement

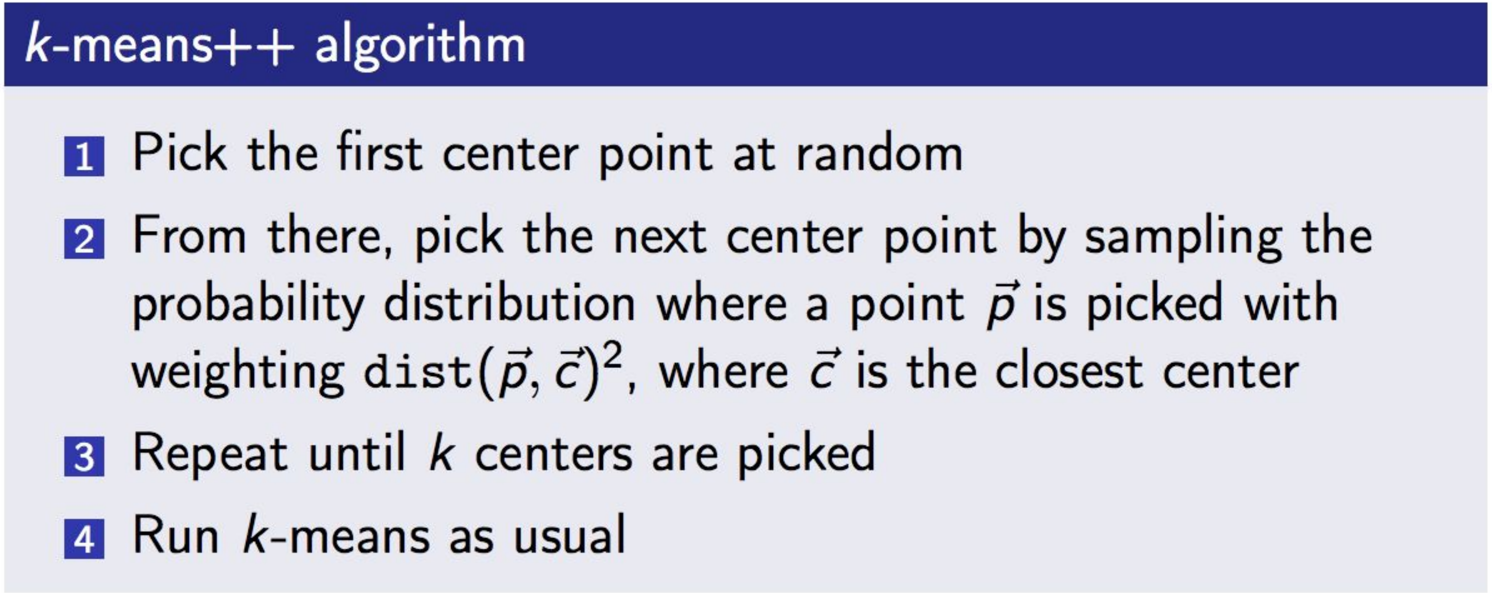
3x improvement (removing *k*-means++)

= 60x improvement!

k-means++

k-means++, Review

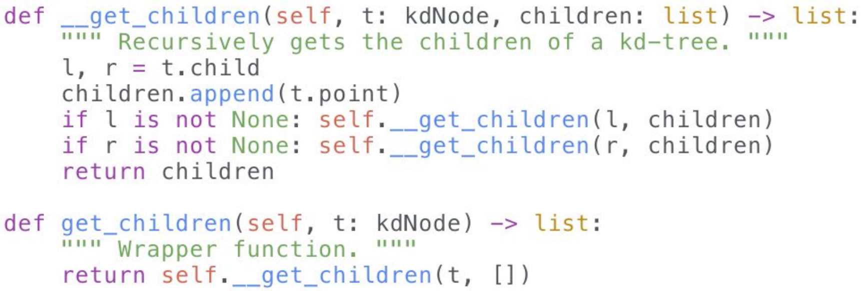
- Initialization strategy
- For more information, see the previous lecture



k-means++ algorithm

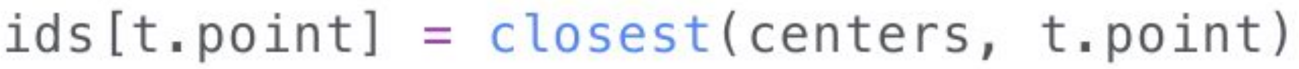
1. Pick the first center point at random
2. From there, pick the next center point by sampling the probability distribution where a point $\vec{p}$ is picked with weighting $\mathtt{dist}(\vec{p}, \vec{c})^2$, where $\vec{c}$ is the closest center
3. Repeat until k centers are picked
4. Run k-means as usual

Speedup



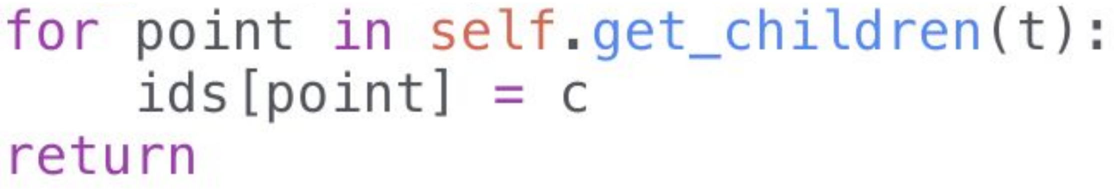
```
def __get_children(self, t: kdNode, children: list) -> list:
    """ Recursively gets the children of a kd-tree. """
    l, r = t.child
    children.append(t.point)
    if l is not None: self.__get_children(l, children)
    if r is not None: self.__get_children(r, children)
    return children

def get_children(self, t: kdNode) -> list:
    """ Wrapper function. """
    return self.__get_children(t, [])
```

- We can apply the kd-tree algorithm
- Simply change the counters
- Leaf/internal node:



```
ids[t.point] = closest(centers, t.point)
```

- Pruned point:



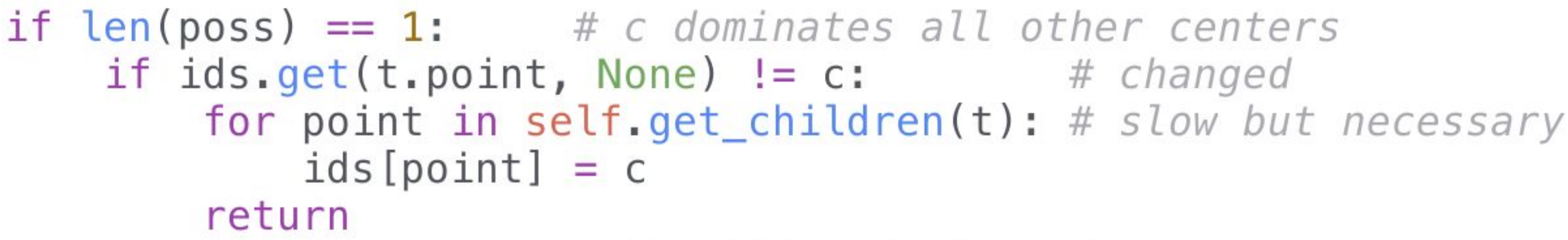
```
for point in self.get_children(t):
    ids[point] = c
return
```

Speedup, Analysis

- Might not seem like we're doing much
- Need to touch every point still
- However, save $O(K)$ distance checking if pruned
  - Save O(average # of centers) after blacklisting for leaves
- Could be up to K times faster
- Can we do better?

Caching IDs

- Need to iterate over each point when pruned
  - Original algorithm can simply skip over
- Centers aren’t changing, only added
- Cache IDs if the new center doesn’t change ownership
  - If kd-node points to *c* and *c* owns *h*, then *c* must have owned *h* in the past
  - If *c* owned *h* and *c* dominates the new center, all points already point to *c*
  - No need to do anything



```
if len(poss) == 1:     # c dominates all other centers
    if ids.get(t.point, None) != c:        # changed
        for point in self.get_children(t): # slow but necessary
            ids[point] = c
        return
```

Caching IDs, Analysis

- Expected time savings?
- Assume points are split evenly among centers
  - K = 1 -> need to update all points
  - K = 2 -> need to update half of the points
  - etc.
- $N + \frac{1}{2} N + \frac{1}{3} N + \ldots + 1/K\, N = N\,(1 + \frac{1}{2} + \frac{1}{3} + \ldots + 1/K)$
- $= N\, H_K = O(N \log K)$
- Very optimistic
- In practice:
  - 108,248 for K = 8 vs NK = 480,000
  - 4,794,349 for K = 256 vs NK = 15,360,000

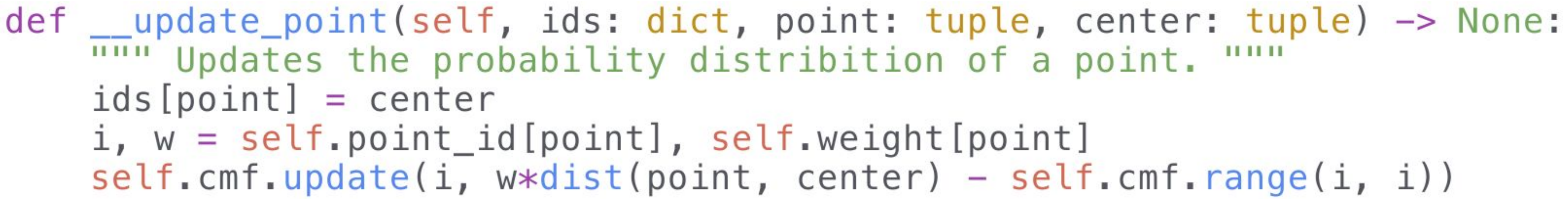
Probability Distribution

- Still maintaining a list of distances
- Need to sample from this PMF
- = $O(N)$, $O(NK)$ over K centers
- Can we speed this up?

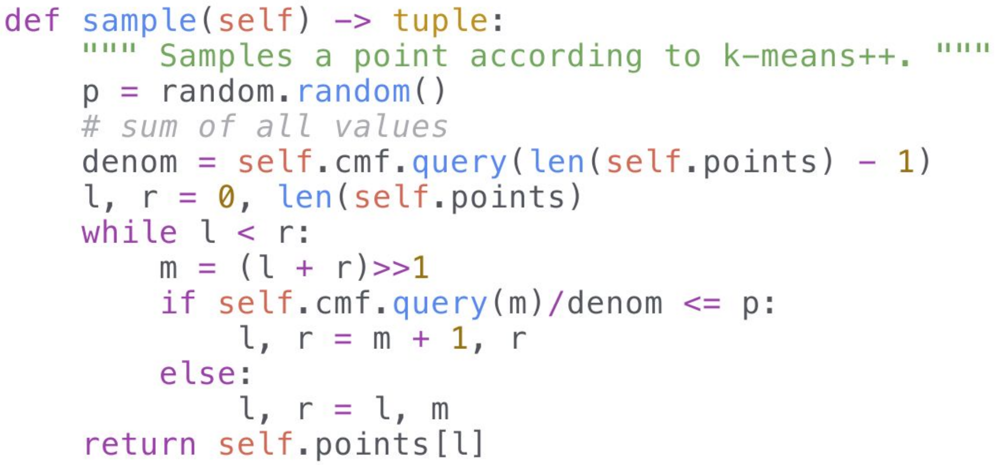
BIT CMF

- Maintain the *cumulative mass function* (CMF) instead
- CMF is just the prefix sum of the PMF
- Only update when a point changes
- Can efficiently update with segment trees or binary indexed trees (BITs)
    - My BIT and segtree implementation here
- Sample by binary searching on the CMF
    - Why does this work? See: my analysis of Mudae (page 9) and the previous lecture

BIT CMF, Code



```
def __update_point(self, ids: dict, point: tuple, center: tuple) -> None:
    """ Updates the probability distribition of a point. """
    ids[point] = center
    i, w = self.point_id[point], self.weight[point]
    self.cmf.update(i, w*dist(point, center) - self.cmf.range(i, i))
```



```
def sample(self) -> tuple:
    """ Samples a point according to k-means++. """
    p = random.random()
    # sum of all values
    denom = self.cmf.query(len(self.points) - 1)
    l, r = 0, len(self.points)
    while l < r:
        m = (l + r)>>1
        if self.cmf.query(m)/denom <= p:
            l, r = m + 1, r
        else:
            l, r = l, m
    return self.points[l]
```

BIT CMF, Analysis

- Sample runs in $O(\log^2 N)$
  - $\log N$ queries from the binary search, each query is $\log N$
  - = $O(\mathrm{K} \log^2 N)$ over K centers
- The total amount of work for updating IDs is harder
- Optimistically, $O(N \log K)$ total updates
  - = $O(\mathrm{N} \log N \log K)$ overall
- In practice, closer to $O(NK) = O(N \log N\ K)$
- Naive is $O(NK^2)$
  - K iterations of finding the closest center for N points

Benchmarks

```
testcases/hanekawa.png, K = 256, PyPy
k-means++               naive: 311.600s, 111 iterations
k-means++             kd-tree: 122.594s, ---
k-means++ geometric reasoning:  75.269s, ---
k-means++           cache ids:  53.181s, ---
k-means++             BIT CMF:  44.143s, ---
              random.sample():   0.001s, 108
```

k-means++, Judgement

- *k*-means++ has strong theoretical properties
- On “real-world” datasets, irrelevant
  - *K* iterations over *N* is just way too much!
  - Can’t parallelize, each iteration depends on previous
- Need some way of speeding up...

Conclusion

- Build a kd-tree on the points, not the centers
- Works well in practice and in theory
- Many avenues for improvement

256 color image generated in less than 30 seconds

Appendix

- How do we test if *k*-means has converged?
- Centers hasn’t changed
- Sort new center list, check if equal to previous
- Not necessarily *O*(D *K* log *K*) (although this is a fairly trivial cost)
- Some sorting algorithms proportional to displacement, e.g. bubble sort
- Can run in nearly linear time if the centers don’t change that much

References

- *Accelerating Exact k-means Algorithms with Geometric Reasoning*
- k-means, kd-Trees, and Median of Medians (CV club lecture)
- CMU kd-tree, kd-tree continued