KD-Tree Algorithm for Propensity Score Matching
PhD Qualifying Exam Defense

John R Hott

University of Virginia

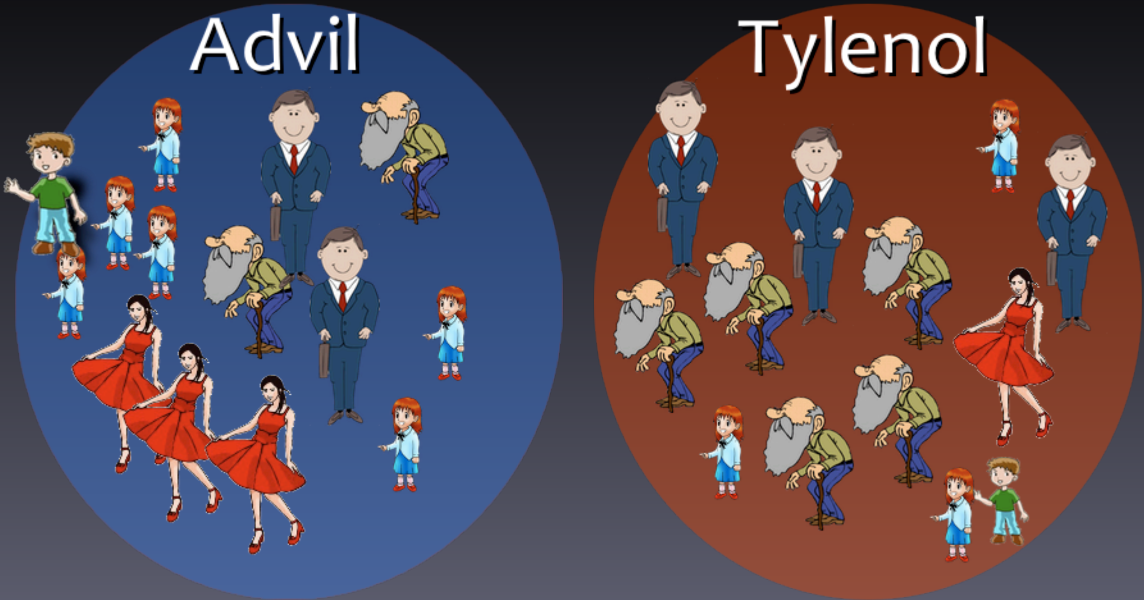
May 11, 2012

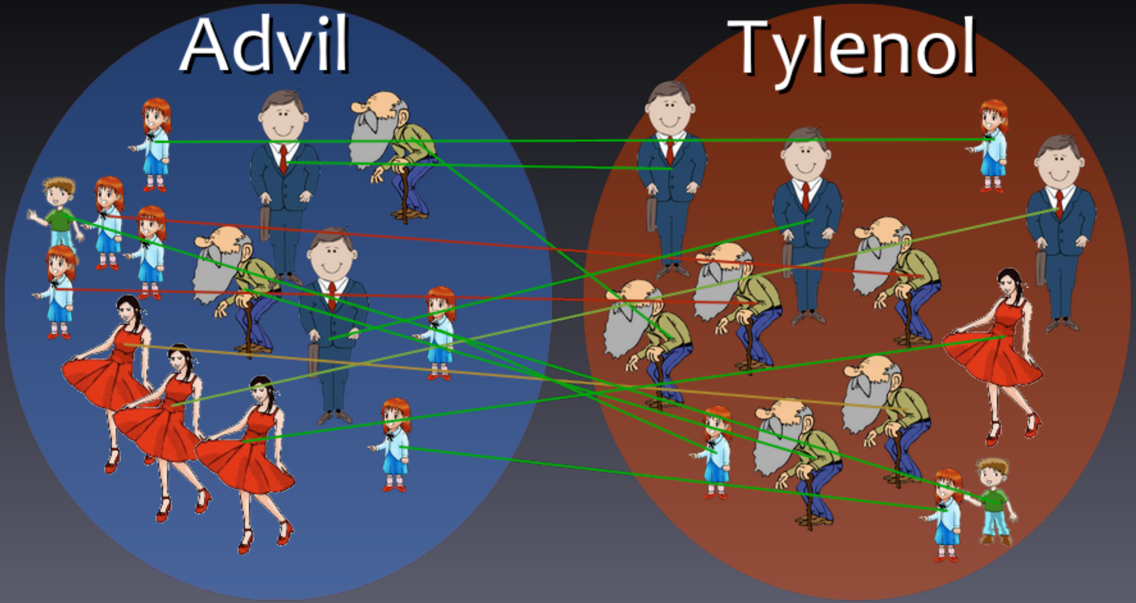
Motivation

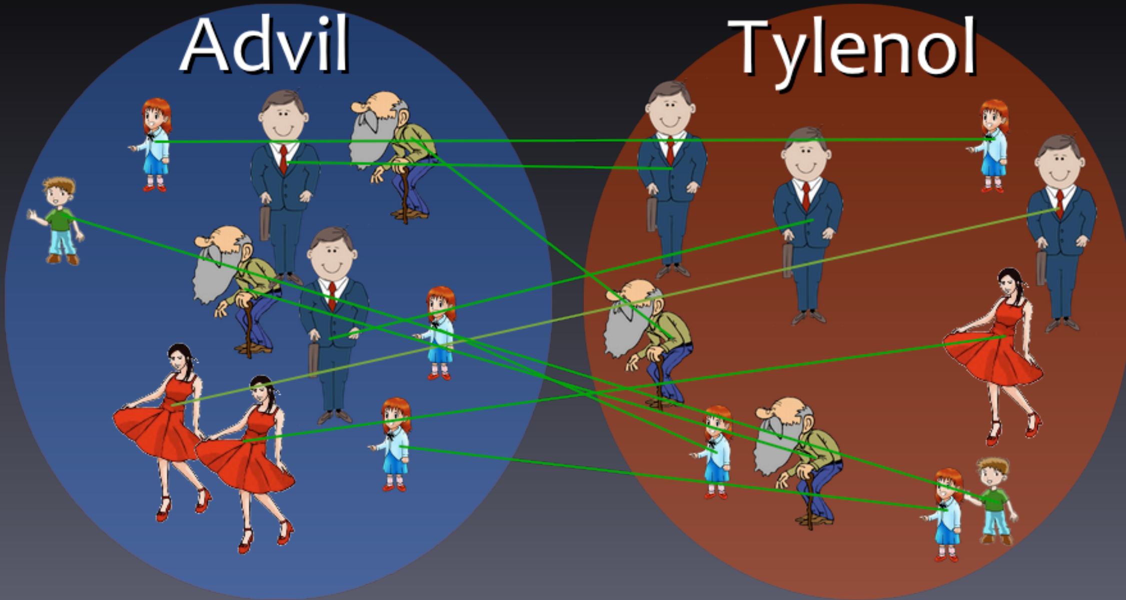
- Epidemiology: Clinical Trials
  - Phase II and III pre-market trials
  - After-market Phase IV trials against 3+ treatments
- Trials requiring similar groups to avoid confounding
  - Participants with similar comorbidities (diseases, ...)
  - Participants with similar traits (age, weight, height, ...)
- Propensity Scores
  - Probability, given certain factors, that a person will be given a certain treatment
  - Want to match participants with similar propensity scores

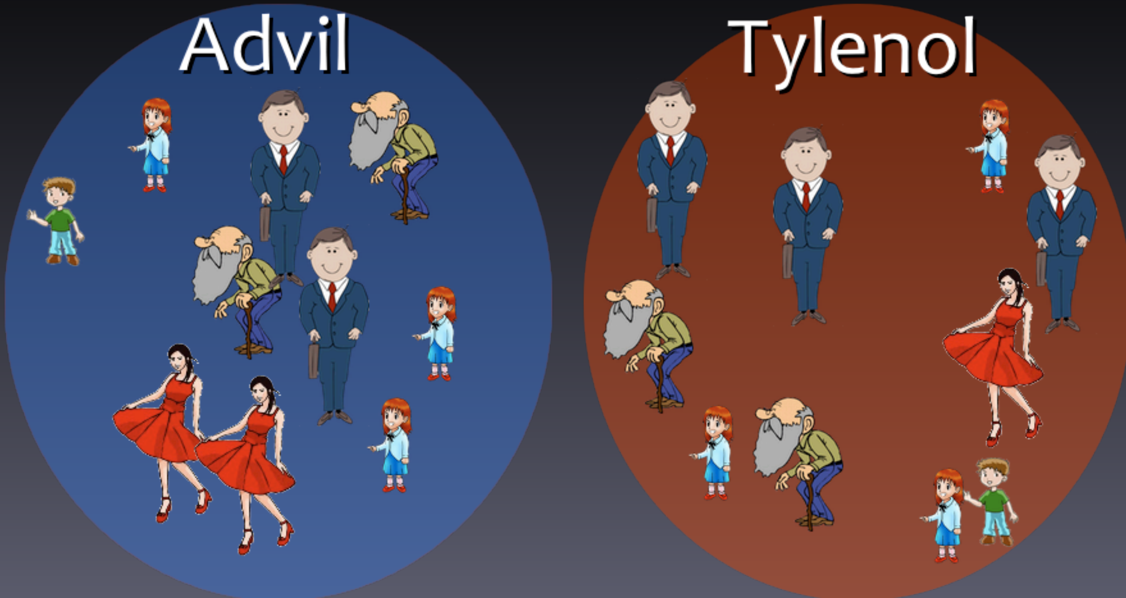

Advil

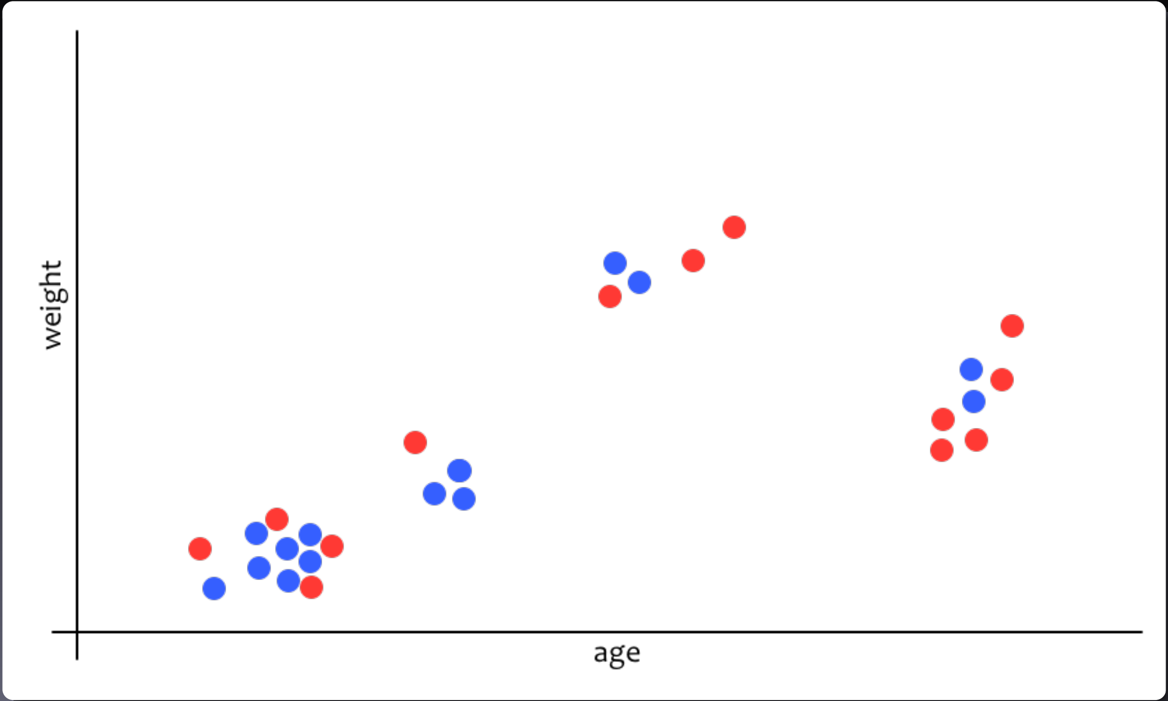

Tylenol

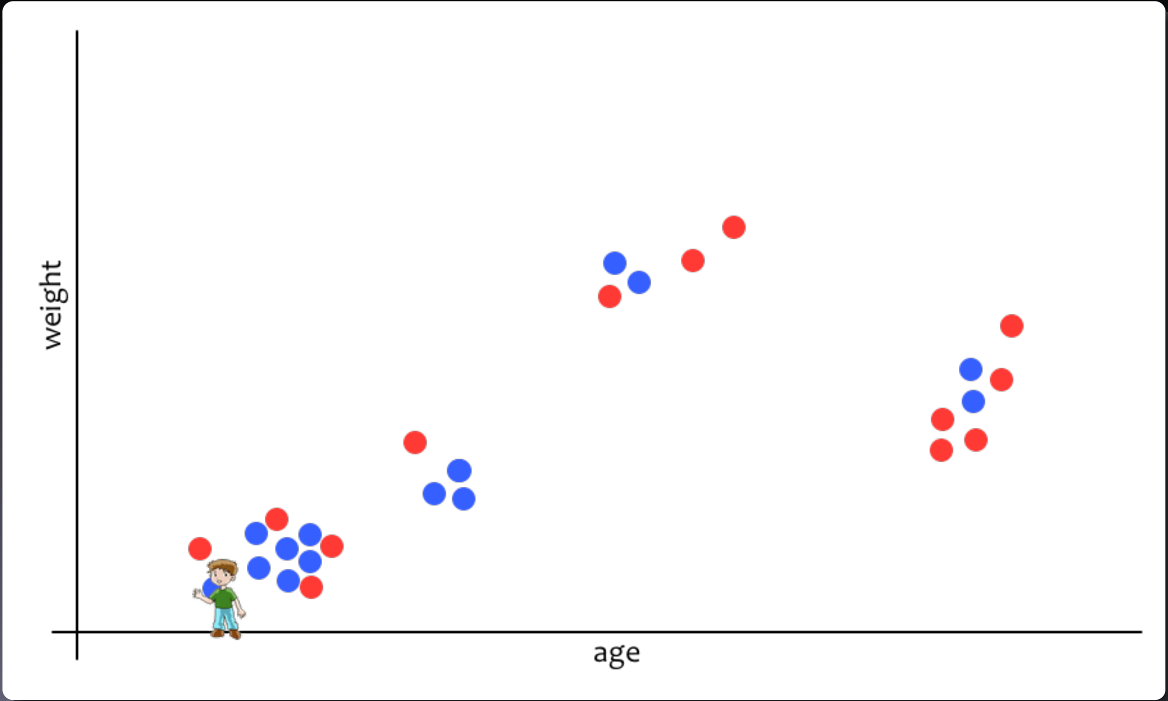

Advil
Tylenol

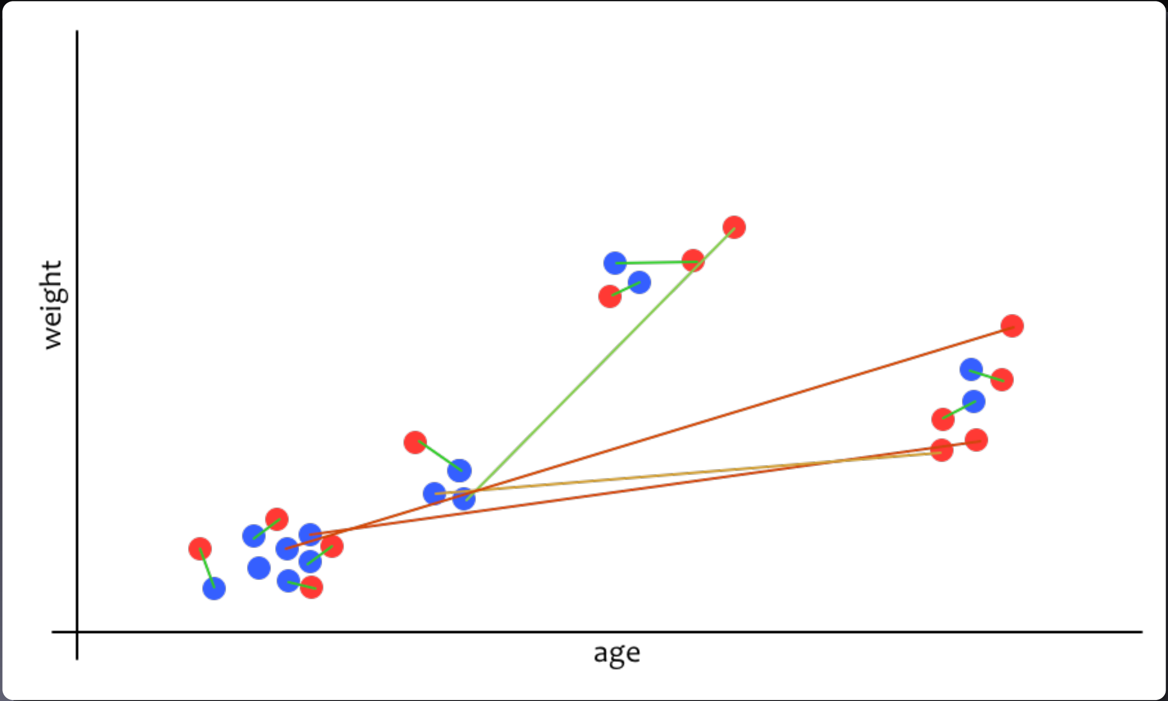

Advil
Tylenol

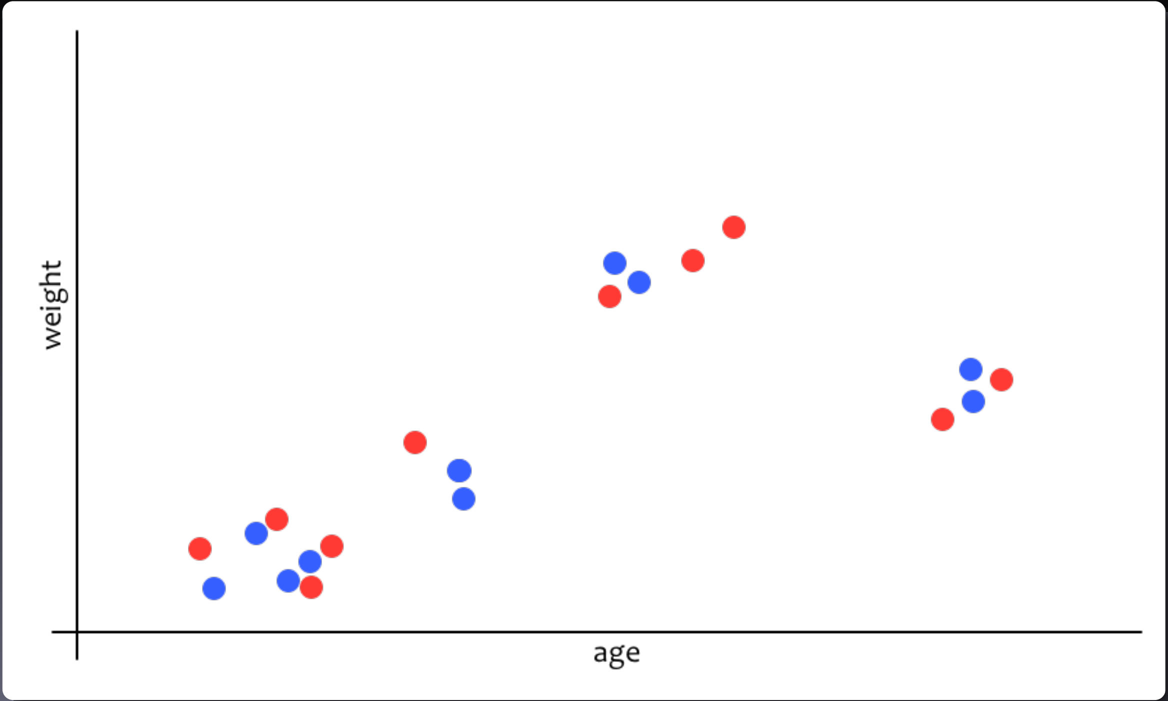

Advil
Tylenol


Advil
Tylenol

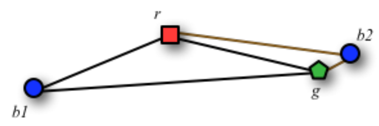

weight
age


weight
age


weight
age


weight
age

Why doesn't this nearest neighbor approach work for more groups?



- b_1 and g closest points to r
- Triangle rgb_2 has smaller perimeter than rgb_1

Current Approaches

For two treatment groups

- Propensity scores used to reduce dimensionality
- Brute force or nearest neighbor searches

For more than two groups

- Brute force
  - Requires $O(n^k \log n)$ time using $O(n^k)$ space
  - Less efficient brute force uses $O(n)$ space, but $O(n^{k+1})$ time
  - Problem: must consider all matches
  - Not feasible for large n

kd-tree algorithm, under a uniform distribution, will perform in $O(kdn^2)$ time and $O(n)$ space.

The End: Spoilers

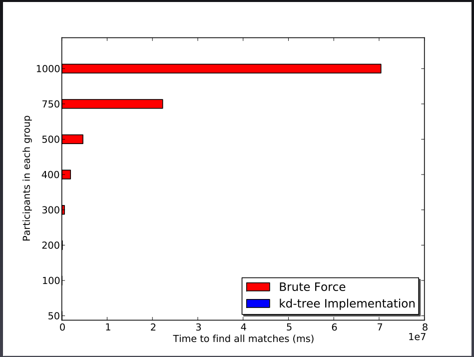


Figure: Results in 3-dimensions with 3 groups

For 1000 participants in 3 groups with 3 dimensions

- Brute Force: 19.6 hours
- kd-tree Algorithm: 3.6 seconds (19,427x speedup)

Problem Statement

Informally, we want to make the smallest n disjoint matches with one participant from each of k groups per match.

- Start with participants in one group,
- Find their closest matched participants of each other group (nearest neighbor),
- Search within a small neighborhood of these points for a smaller match, if one exists,
- Repeat, if necessary.

Outline

Participant Definitions

Definition (Participant)

Let the point $p \in \mathbb{R}^d$ be a **participant** *normalized over d defining characteristics*

Participant Definitions

Definition (Participant)

Let the point $p \in \mathbb{R}^d$ be a **participant** *normalized over d defining characteristics*

Definition (Set of all Participants)

The set $\mathcal{P} \subseteq \mathbb{R}^d$, is the **set of all participants**, *such that:*

- $\mathcal{P} = \cup_{i=1}^{k} G_i$, *where each G_i defines a treatment group*
- $|G_i| = n$
- $|\mathcal{P}| = \sum_i |G_i| = kn.$

Match Definitions

Definition (Match)

A set $m \subseteq \mathcal{P}$ is a **match** *if it contains exactly one point from each G_i:*

- $|m| = k$,
- $|m \cap G_i| = 1, \forall i$.

Match Definitions

Definition (Match)

A set $m \subseteq \mathcal{P}$ *is a* **match** *if it contains exactly one point from each* G_i*:*

- $|m| = k$,
- $|m \cap G_i| = 1, \forall i$.

Definition (Set of all Matches)

Let $\mathcal{M} = \{m : m \text{ is a match}\}$ *be the* **set of all matches** *with*

$$|\mathcal{M}| = \prod_i |G_i| = n^k$$

Size of a Match

Definition

Match measure *function size*(m),

$$size : \mathcal{M} \to \mathbb{R},$$

independent of the order of the points in the match m, must give a consistent measurement of the match.

Ideal measure: minimize the sum of the distance between all points

- Fully connected graph
- Quadratic on k

Match Covering

Definition

M is a **match covering** *of $\mathcal{P}$ if M is a set of disjoint matches:*

- $M \subset \mathcal{M}$
- $|M| = n$
- $\forall m, l \in M$ *where* $m \neq l$ *then* $m \cap l = \emptyset$.

Match Covering

Definition

*M is a **match covering** of $\mathcal{P}$ if M is a set of disjoint matches:*

- $M \subset \mathcal{M}$
- $|M| = n$
- $\forall m, l \in M$ *where* $m \neq l$ *then* $m \cap l = \emptyset$.

Definition

WLOG, assume M is sorted on size(m): $\forall m_i, m_j \in M, i < j \implies size(m_i) < size(m_j)$.
Define ordering $<_M$ such that $M_0 <_M M_1$ if for some index i,

$$size(m_{0,i}) < size(m_{1,i}) \text{ and } \forall j < i, size(m_{0,j}) = size(m_{1,j})$$

- *Size of match in M_0 less than size of match in M_1 at the first place they differ*

Problem Statement

Find the minimal match covering, M_0, such that

$$\forall i, M_0 \leq_M M_i.$$

Match *size* function

What is the best method for measuring the size of a match?
Perimeter? Convex Hull? Something else?

Measuring Matches: by example

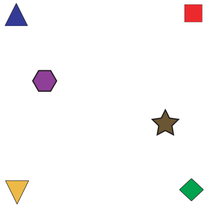
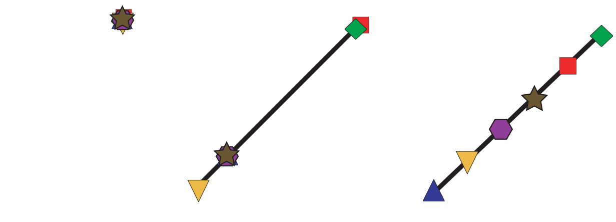


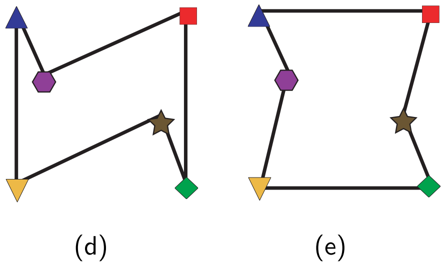
Figure: Sample points for one match.

Measuring Matches: Area



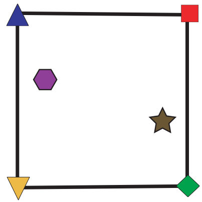
- All colinear points have 0 area, regardless of distance
- Favors colinear points

Measuring Matches: Perimeter



- Works for 2-dimensions, 3-colors
- Equivalent to Traveling Salesman as number of colors increases
- Not well defined in more dimensions

Measuring Matches: Convex Hull

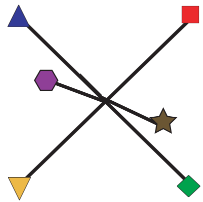


(f)

- Avoids TSP encountered with perimeter
- $\Omega(n^{\lfloor d/2 \rfloor})$ for $d > 3$

more

Measuring Matches: Centroid



- Linearly computable (on colors and dimensions)
- Statistical sense: distance to an average point
- Matches are intuitively small
- Possible Measurements
  - Max distance to centroid
  - Average distance to centroid (variance)
  - Sum of squared distances to centroid

more

Perimeter Measure

For 3 or fewer groups, perimeter matches our ideal measurement. In this case,

$$size(m) = perimeter(m).$$

This definition leads to a search radius of

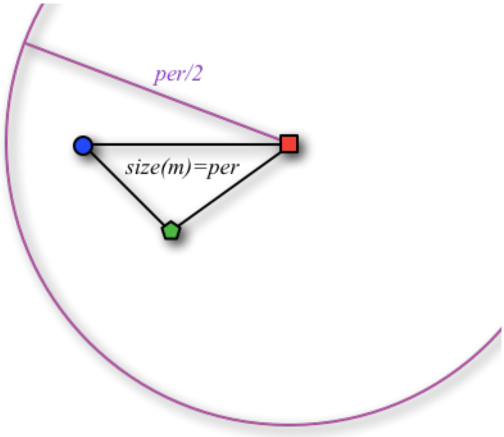
$$search(m) = \frac{1}{2} size(m).$$

Proof of Correctness: Perimeter

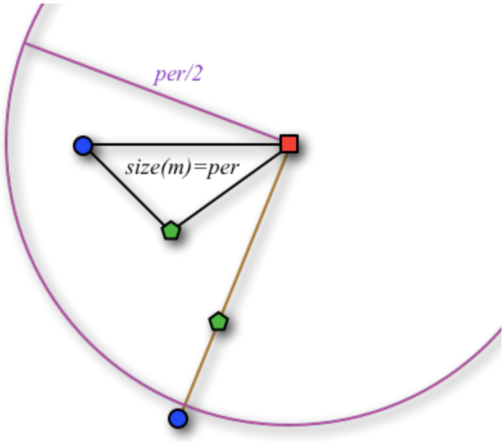
Theorem

Given an initial match m containing point $p_i \in G_1$, which contains random points p_{r_j}, one per G_j with $j \geq 2$, the perimeter will define the size of the match. Let us assume that $size(m) = q \in \mathbb{R}$. Our search radius will then be the disc centered at p_i with radius $\frac{q}{2}$. This area will contain the smallest match for p_i.

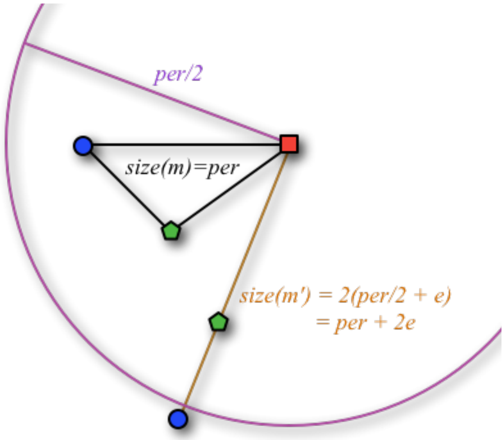
Proof of Correctness: Perimeter



Proof of Correctness: Perimeter



Proof of Correctness: Perimeter



Centroid Measure

For a *d*-dimensional space and *k* colors per match, we consider the sum of squared distances to the centroid.

The centroid of a match *m* is defined as

$$c(m) = \frac{1}{k} \sum_{i=1}^{k} p_i.$$

Our $size(m)$ function, using sum of squared distances to the centroid, is defined as

$$size(m) = \sum_{i=1}^{k} \left(\sum_{j=1}^{d} (p_{i,j} - c_j(m))^2 \right)^2.$$

This definition leads to a search radius of

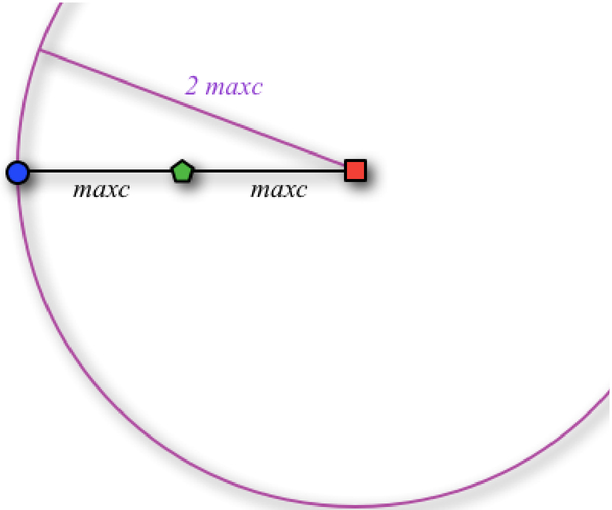
$$search(m) = k * \text{max distance to centroid}.$$

Proof of Correctness: Centroid

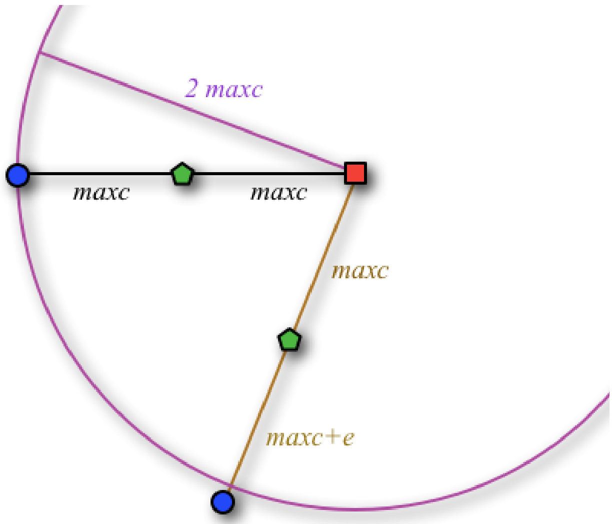
Theorem

Given an initial match m containing point $p_i \in G_1$, which contains random points $p_{r\,j}$, one per G_j with $j \geq 2$, the sum of squared distances to the centroid will define the size of the match. Our search radius will be the disc centered at p_i with radius $k * s$ where s is the max distance to centroid. This area will contain the smallest match for p_i.

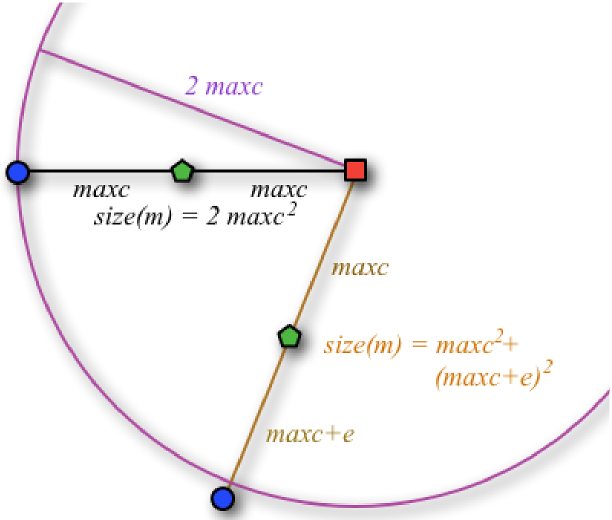
Proof of Correctness: Centroid



Proof of Correctness: Centroid



Proof of Correctness: Centroid



more

Outline

1. Motivation
2. Formal Definition
   - Definitions
   - Size Function
3. Current Approaches
4. kd-tree Algorithm
   - kd-trees
   - Algorithm Description
5. Algorithm Analysis
   - Worst Case
   - Expected Case
   - Empirical Study
6. Conclusions

Brute Force Approaches

Algorithm 2: $O(n^k \log n)$ time, but $O(n^k)$ space.

Input: k sets of n points
Output: set of n ordered smallest matches of k points each

read input
foreach $p_1 \in G_1$ **do**
  foreach $p_2 \in G_2$ **do**
    ... **foreach** $p_k \in G_k$ **do**
      $M \leftarrow m = \{p_1, p_2, ..., p_k\}$

sort(M)
foreach M **do**
  if $p_1, p_2, ..., p_k$ *clean* **then**
    $M_{ans} \leftarrow m$

return M_{ans}

Brute Force Approaches

Algorithm 1: $O(n)$ space, but $O(n^{k+1})$ time.

Input: k sets of n points
Output: set of n ordered smallest matches of k points each

```
read input
for i = 1 : n do
    smallest = MAX
    m_smallest = null
    foreach p_1 ∈ G_1 do
        foreach p_2 ∈ G_2 do
            ...foreach p_k ∈ G_k do
                if size(m = {p_1, p_2, ..., p_k}) < smallest then
                    m_smallest = m
                    smallest = size(m)
    M_ans ← m_smallest
    remove p_1, p_2, ..., p_k from G_1, G_2, ..., G_k
return M_ans
```

Voronoi Matching Algorithm

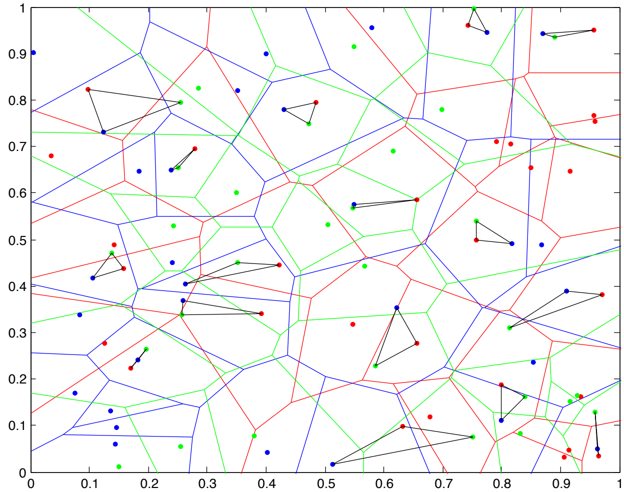


Figure: Voronoi Cells.

Voronoi Matching Algorithm

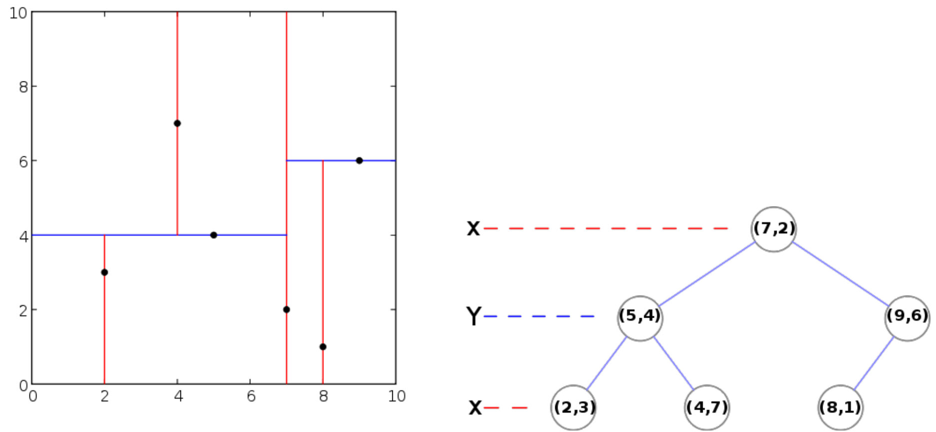
Problems with the Voronoi algorithm

- Provides only an approximation for M_0
- Worst and expected case complexity $O(n^4)$ for 3 groups
- Required searching for points in polygons

Outline

kd-trees

Binary tree data structure used to store points in d-dimensional space.



more

kd-tree Algorithm

Input: k sets of n points
Output: set of n ordered smallest matches of k points each

```
G_1 = read input
for i ← 2 to k do
    G_i = read input
    T_i = makeKDTree(G_i)

pq = new PriorityQueue
matches = new ArrayList
foreach p_i ∈ G_1 do
    addPutativeMatches(p_i, pq)

while pq not empty do
    m = pq.poll()
    if all points are unused then
        foreach i ≤ k do
            T_i.remove(m.i)

        matches.add(m)
    else
        if point m.1 ∈ G_1 is unused then
            if no more matches available then
                addPutativeMatches(m.1, pq)
```

addPutativeMatches Subroutine

Input: PriorityQueue pq, current point p_1 from G_1, kd-trees T_i for each G_2 to G_k
Output: list of 10 smallest matches for point p_1

```
for i ← 2 to k do
    p_i = T_i.getnearest(p_{i−1})
small = size(p_1, p_2, ... , p_k)
search = get search distance from small
tq = new PriorityQueue
tq.add(match(p_1,p_2, ... ,p_k))
for i ← 2 to k do
    list_i = T_i.getnearest(p_{i−1}, search)
foreach list_2 as p_2 do
    foreach list_3 as p_3 do
        ... foreach list_k as p_k do
            dist = size(p_1,p_2, ... ,p_k)
            if dist ≤ small then
                tq.add(match(p_1,p_2, ... ,p_k))
for i ← 1 to 10 do
    m = tq.poll()
    pq.add(m);
```

Empirical Study: addPutativeMatches returns

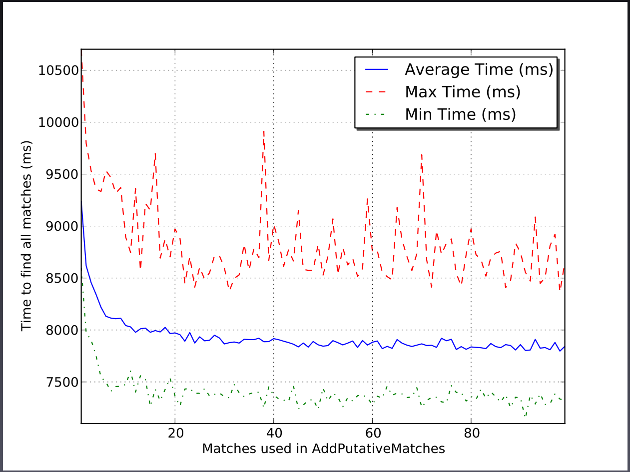


Figure: Empirical study varying number of matches returned by addPutativeMatches.

Outline

Worst Case

Figure: Worst case example (3 colors in 2 dimensions). Points in each G_i are coincident with each other ($\forall i : r_i = (-1, 0), g_i = (0, 0), b_i = (1, 0)$); all points are within the search area of any match (r_i, g_i, b_i).

addPutativeMatches Worst Case

Complexity	
$O(kdn^{1-1/d})$	**for** $i \leftarrow 2$ **to** k **do** $p_i = T_i$.getnearest(p_{i-1})
$O(1)$	$small = size(p_1, p_2, \ldots, p_k)$
$O(kd)$	$search$ = get search distance from small
$O(1)$	tq = new PriorityQueue
$O(\log n)$	tq.add$(match(p_1, p_2, \ldots, p_k))$
$O(kdn^{1-1/d})$	**for** $i \leftarrow 2$ **to** k **do** $list_i = T_i$.getnearest$(p_{i-1}, search)$

	foreach $list_2$ *as* p_2 **do**
	    foreach $list_3$ *as* p_3 **do**
	        … **foreach** $list_k$ *as* p_k **do**

$O(kdn^{k-1})$	$dist = size(p_1, p_2, \ldots, p_k)$
$O(n^{k-1} \log n^{k-1})$	**if** $dist \leq small$ **then** tq.add$(match(p_1, p_2, \ldots, p_k))$

$O(10 \log n^{k-1})$	**for** $i \leftarrow 1$ **to** 10 **do**
	    $m = tq$.poll()
	    pq.add(m);

$O(n^{k-1} \log n + kdn^{k-1} + 2kdn^{1-1/d})$

more

kd-tree Algorithm Worst Case

Complexity	
$O(n)$	G_1 = read input
	for $i \leftarrow 2$ **to** k **do**
	G_i = read input
$O(kn \log n)$	T_i = makeKDTree(G_i)
$O(1)$	pq = new PriorityQueue
$O(1)$	$matches$ = new ArrayList
	foreach $p_i \in G_1$ **do**
$O(n^k \log n)$	addPutativeMatches(p_i, pq)
	while *pq not empty* **do**
$O(n^k \log n)$	$m = pq$.poll()
	if *all points are unused* **then**
	foreach $i \leq k$ **do**
$O(nk \log n)$	T_i.remove($m.i$)
$O(n \log n)$	$matches$.add(m)
	else
	if *point* $m.1 \in G_1$ *is unused* **then**
	if *no more matches available* **then**
$O(n^{2k-1} \log n)$	addPutativeMatches($m.1, pq$)

$O\left(n^{2k-1} \log n + kn^{k+1} \log n + kdn^{k+1}\right)$

more

Expected Case Assumption

For n points, $\exists\delta$ such that $\forall\varepsilon$-sized areas, there are less than $\delta\varepsilon n$ points in that region.

- This assumes a uniform distribution, per study design
- Location of the ε-sized area is irrelevant
- As the density increases, the search radius becomes smaller
- For a small area $\varepsilon \approx 1/n$, number of points appears constant in that area (δ)
  - Since we look at the k closest neighbors to a point

Expected Case

$$
\begin{aligned}
T_{apm_{k,d}} &= O(2(k-1)dn^{1-\frac{1}{d}} + \log n) = O(kdn) \\
T_{part1_{k,d}} &= O\left(nT_{apm_{k,d}}\right) = O(kdn^2) \\
T_{part2_{k,d}} &= O\left(n \log n + nk \log n\right) = O((k+1)n \log n)
\end{aligned}
$$

with the total time complexity reducing to

$$
\begin{aligned}
T_{kdtree} &= T_{build_{kds}} + T_{part1_{k,d}} + T_{part2_{k,d}} \\
&= O\left((k-1)(n \log n) + kdn^2 + (k+1)n \log n\right) \\
&= O(kdn^2).
\end{aligned}
$$

Empirical Study: Brute Force vs KD-Tree

Parameter	Values
Number of Treatment Groups	3 - 4
Participants per Treatment Group	50, 100, 200, 300, 400, 500, 750, 1000
Confounding Factors per Participant	3

Table: Empirical test configurations.

Each configuration repeated 50 times.
Centurion cluster nodes

- 1.6 GHz dual-core Opteron
- 3GB RAM

Empirical Study: Brute Force vs KD-Tree

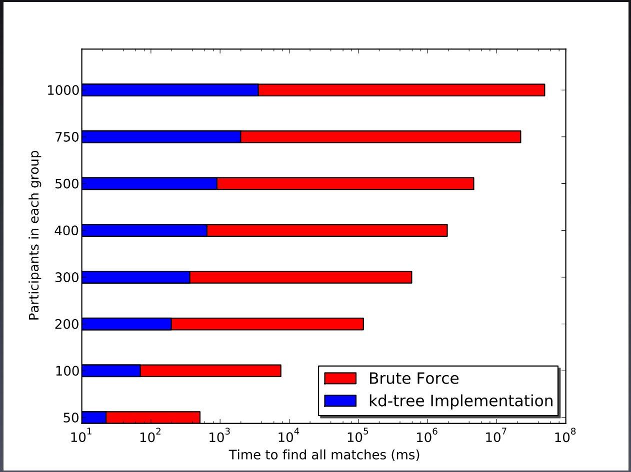


Figure: Results in 3-dimensions with 3 groups (log-scale x-axis)

Empirical Study: Brute Force vs KD-Tree

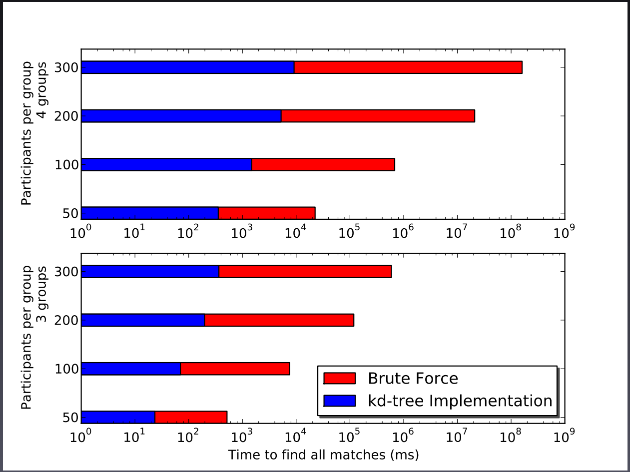


Figure: Empirical study comparing brute force and the kd-tree algorithm. (log-scale x-axis)

Outline

Future Directions

- Alternative applications for the algorithm
- Combining kd-trees and voronoi cells
  - Some research into using voronoi cells to speed kd-tree lookups
  - Utilize kd-trees to build effective voronoi diagrams (completed)
  - Extracting matches using the effective voronoi diagrams before completion using the kd-tree algorithm
- Other assumptions for expected cases
  - Alternate distributions
  - Slowly growing δ in our expected assumption
- Other match measure functions
- Reduce the search area once a smaller match is found

Research Plan

Proposed research directions:

- √ Generalize the kd-tree algorithm to an arbitrary k colors in d dimensions, as defined in the problem statement,
- √ Analyze the time complexity of the k-d tree algorithm for both worst-case and expected case running times,
- √ Examine other methods for defining the size of a match that are not dependent or limited by dimensionality, number of colors, or ordering of the points,
- √ Prove algorithm correctness.

Additional research directions:

- √ Perform an empirical study of addPutativeMatches return values,
- √ Perform an empirical study comparing brute force to the kd-tree algorithm for 3-5 groups in 3-4 dimensions.

Questions?

addPutativeMatches Analysis

Complexity

for $i \leftarrow 2$ **to** k **do**
    $p_i = T_i.\text{getnearest}(p_{i-1})$

$small = size(p_1, p_2, \ldots, p_k)$
$search =$ get search distance from small
$tq =$ new PriorityQueue
$tq.\text{add}(match(p_1, p_2, \ldots, p_k))$
for $i \leftarrow 2$ **to** k **do**
    $list_i = T_i.\text{getnearest}(p_{i-1}, search)$

foreach $list_2$ *as* p_2 **do**
    foreach $list_3$ *as* p_3 **do**
        ... **foreach** $list_k$ *as* p_k **do**
            $dist = size(p_1, p_2, \ldots, p_k)$
            if $dist \leq small$ **then**
                $tq.\text{add}(match(p_1, p_2, \ldots, p_k))$

for $i \leftarrow 1$ **to** 10 **do**
    $m = tq.\text{poll}()$
    $pq.\text{add}(m);$

back

addPutativeMatches Analysis

Complexity
$O(k)$

```
for i ← 2 to k do
    p_i = T_i.getnearest(p_{i−1})
small = size(p_1, p_2, ... , p_k)
search = get search distance from small
tq = new PriorityQueue
tq.add(match(p_1,p_2, ... ,p_k))
for i ← 2 to k do
    list_i = T_i.getnearest(p_{i−1}, search)
foreach list_2 as p_2 do
    foreach list_3 as p_3 do
        ... foreach list_k as p_k do
            dist = size(p_1,p_2, ... ,p_k)
            if dist ≤ small then
                tq.add(match(p_1,p_2, ... ,p_k))
for i ← 1 to 10 do
    m = tq.poll()
    pq.add(m);
```

back

addPutativeMatches Analysis

Complexity
$O(k)$
$O(dn^{1-1/d})$

```
for i ← 2 to k do
    p_i = T_i.getnearest(p_{i-1})
small = size(p_1, p_2, ... , p_k)
search = get search distance from small
tq = new PriorityQueue
tq.add(match(p_1,p_2, ... ,p_k))
for i ← 2 to k do
    list_i = T_i.getnearest(p_{i-1}, search)
foreach list_2 as p_2 do
    foreach list_3 as p_3 do
        ... foreach list_k as p_k do
            dist = size(p_1,p_2, ... ,p_k)
            if dist ≤ small then
                tq.add(match(p_1,p_2, ... ,p_k))
for i ← 1 to 10 do
    m = tq.poll()
    pq.add(m);
```

back

addPutativeMatches Analysis

Complexity

$O(kdn^{1-1/d})$
$O(1)$
$O(kd)$
$O(1)$
$O(\log n)$

```
for i ← 2 to k do
    p_i = T_i.getnearest(p_{i-1})
small = size(p_1, p_2, ... , p_k)
search = get search distance from small
tq = new PriorityQueue
tq.add(match(p_1,p_2, ... ,p_k))
for i ← 2 to k do
    list_i = T_i.getnearest(p_{i-1}, search)
foreach list_2 as p_2 do
    foreach list_3 as p_3 do
        ... foreach list_k as p_k do
            dist = size(p_1,p_2, ... ,p_k)
            if dist ≤ small then
                tq.add(match(p_1,p_2, ... ,p_k))
for i ← 1 to 10 do
    m = tq.poll()
    pq.add(m);
```

back

addPutativeMatches Analysis

Complexity

$O(kdn^{1-1/d})$
$O(1)$
$O(kd)$
$O(1)$
$O(\log n)$
$O(k)$

for $i \leftarrow 2$ **to** k **do**
  $p_i = T_i$.getnearest(p_{i-1})

$small = size(p_1, p_2, \ldots, p_k)$
$search$ = get search distance from small
tq = new PriorityQueue
tq.add($match(p_1, p_2, \ldots, p_k)$)
for $i \leftarrow 2$ **to** k **do**
  $list_i = T_i$.getnearest(p_{i-1}, $search$)

foreach $list_2$ *as* p_2 **do**
  foreach $list_3$ *as* p_3 **do**
    … **foreach** $list_k$ *as* p_k **do**
      $dist = size(p_1, p_2, \ldots, p_k)$
      if $dist \leq small$ **then**
        tq.add($match(p_1, p_2, \ldots, p_k)$)

for $i \leftarrow 1$ **to** 10 **do**
  $m = tq$.poll()
  pq.add(m);

back

addPutativeMatches Analysis

Complexity

$O(kdn^{1-1/d})$
$O(1)$
$O(kd)$
$O(1)$
$O(\log n)$
$O(k)$
$O(dn^{1-1/d})$

---

for $i \leftarrow 2$ **to** k **do**
  $p_i = T_i$.getnearest(p_{i-1})

$small = size(p_1, p_2, \ldots, p_k)$
$search$ = get search distance from small
tq = new PriorityQueue
tq.add($match(p_1, p_2, \ldots, p_k)$)
for $i \leftarrow 2$ **to** k **do**
  $list_i = T_i$.getnearest(p_{i-1}, $search$)

foreach $list_2$ *as* p_2 **do**
  foreach $list_3$ *as* p_3 **do**
    ... **foreach** $list_k$ *as* p_k **do**
      $dist = size(p_1, p_2, \ldots, p_k)$
      if $dist \leq small$ **then**
        tq.add($match(p_1, p_2, \ldots, p_k)$)

for $i \leftarrow 1$ **to** 10 **do**
  $m = tq$.poll()
  pq.add(m);

back

addPutativeMatches Analysis

Complexity	
$O(kdn^{1-1/d})$	**for** $i \leftarrow 2$ **to** k **do** $p_i = T_i$.getnearest(p_{i-1})
$O(1)$	$small = size(p_1, p_2, \ldots, p_k)$
$O(kd)$	$search =$ get search distance from small
$O(1)$	$tq =$ new PriorityQueue
$O(\log n)$	tq.add($match(p_1, p_2, \ldots, p_k)$)
$O(kdn^{1-1/d})$	**for** $i \leftarrow 2$ **to** k **do** $list_i = T_i$.getnearest(p_{i-1}, $search$)
$O(n)\times$	**foreach** $list_2$ *as* p_2 **do**
$O(n)\times$	**foreach** $list_3$ *as* p_3 **do**

$\times \ldots \times O(n)$	... **foreach** $list_k$ *as* p_k **do**
	            $dist = size(p_1, p_2, \ldots, p_k)$
	            if $dist \leq small$ **then**
	                tq.add($match(p_1, p_2, \ldots, p_k)$)

	for $i \leftarrow 1$ **to** 10 **do**
	    $m = tq$.poll()
	    pq.add(m);

back

addPutativeMatches Analysis

Complexity

$O(kdn^{1-1/d})$
$O(1)$
$O(kd)$
$O(1)$
$O(\log n)$

$O(kdn^{1-1/d})$

$O(n)\times$
$O(n)\times$
$\times \ldots \times O(n)$
$O(kd)$
$O(\log n^{k-1})$

---

for $i \leftarrow 2$ **to** k **do**
  $p_i = T_i$.getnearest(p_{i-1})

$small = size(p_1, p_2, \ldots, p_k)$
$search$ = get search distance from small
tq = new PriorityQueue
tq.add($match(p_1, p_2, \ldots, p_k)$)
for $i \leftarrow 2$ **to** k **do**
  $list_i = T_i$.getnearest(p_{i-1}, $search$)

foreach $list_2$ *as* p_2 **do**
  foreach $list_3$ *as* p_3 **do**
    … **foreach** $list_k$ *as* p_k **do**
      $dist = size(p_1, p_2, \ldots, p_k)$
      if $dist \leq small$ **then**
        tq.add($match(p_1, p_2, \ldots, p_k)$)

for $i \leftarrow 1$ **to** 10 **do**
  $m = tq$.poll()
  pq.add(m);

back

addPutativeMatches Analysis

Complexity	
	for $i \leftarrow 2$ **to** k **do**
$O(kdn^{1-1/d})$	$p_i = T_i$.getnearest(p_{i-1})
$O(1)$	$small = size(p_1, p_2, \ldots, p_k)$
$O(kd)$	*search* = get search distance from small
$O(1)$	*tq* = new PriorityQueue
$O(\log n)$	*tq*.add($match(p_1, p_2, \ldots, p_k)$)
	for $i \leftarrow 2$ **to** k **do**
$O(kdn^{1-1/d})$	$list_i = T_i$.getnearest(p_{i-1}, *search*)
	foreach $list_2$ *as* p_2 **do**
	foreach $list_3$ *as* p_3 **do**
	... **foreach** $list_k$ *as* p_k **do**
$O(kdn^{k-1})$	$dist = size(p_1, p_2, \ldots, p_k)$
$O(n^{k-1} \log n^{k-1})$	**if** $dist \leq small$ **then**
	tq.add($match(p_1, p_2, \ldots, p_k)$)
$O(10 \log n^{k-1})$	**for** $i \leftarrow$ *1* **to** *10* **do**
	$m = tq$.poll()
	pq.add(m);

addPutativeMatches Analysis

Complexity	
	for $i \leftarrow 2$ **to** k **do**
$O(kdn^{1-1/d})$	$p_i = T_i$.getnearest(p_{i-1})
$O(1)$	$small = size(p_1, p_2, \ldots, p_k)$
$O(kd)$	$search$ = get search distance from small
$O(1)$	tq = new PriorityQueue
$O(\log n)$	tq.add$(match(p_1, p_2, \ldots, p_k))$
	for $i \leftarrow 2$ **to** k **do**
$O(kdn^{1-1/d})$	$list_i = T_i$.getnearest$(p_{i-1}, search)$
	foreach $list_2$ *as* p_2 **do**
	foreach $list_3$ *as* p_3 **do**
	… **foreach** $list_k$ *as* p_k **do**
$O(kdn^{k-1})$	$dist = size(p_1, p_2, \ldots, p_k)$
$O(n^{k-1} \log n^{k-1})$	**if** $dist \leq small$ **then**
	tq.add$(match(p_1, p_2, \ldots, p_k))$
$O(10 \log n^{k-1})$	
	for $i \leftarrow 1$ **to** 10 **do**
	$m = tq$.poll()
	pq.add(m);

$O(n^{k-1} \log n + kdn^{k-1} + 2kdn^{1-1/d})$

back

kd-tree Algorithm Analysis

Complexity
$O(n)$

G_1 = read input
for $i \leftarrow 2$ **to** k **do**
    G_i = read input
    T_i = makeKDTree(G_i)
pq = new PriorityQueue
matches = new ArrayList
foreach $p_i \in G_1$ **do**
    addPutativeMatches(p_i, pq)
while *pq not empty* **do**
    $m = pq$.poll()
    if *all points are unused* **then**
        foreach $i \leq k$ **do**
            T_i.remove($m.i$)
        matches.add(m)
    else
        if *point* $m.1 \in G_1$ *is unused* **then**
            if *no more matches available* **then**
                addPutativeMatches($m.1$, pq)

back

kd-tree Algorithm Analysis

Complexity
$O(n)$
$O(k)$

```
G_1 = read input
for i ← 2 to k do
    G_i = read input
    T_i = makeKDTree(G_i)
pq = new PriorityQueue
matches = new ArrayList
foreach p_i ∈ G_1 do
    addPutativeMatches(p_i, pq)
while pq not empty do
    m = pq.poll()
    if all points are unused then
        foreach i ≤ k do
            T_i.remove(m.i)
        matches.add(m)
    else
        if point m.1 ∈ G_1 is unused then
            if no more matches available then
                addPutativeMatches(m.1, pq)
```

back

kd-tree Algorithm Analysis

Complexity
$O(n)$
$O(k)$
$O(n + n \log n)$

```
G_1 = read input
for i ← 2 to k do
    G_i = read input
    T_i = makeKDTree(G_i)
pq = new PriorityQueue
matches = new ArrayList
foreach p_i ∈ G_1 do
    addPutativeMatches(p_i, pq)
while pq not empty do
    m = pq.poll()
    if all points are unused then
        foreach i ≤ k do
            T_i.remove(m.i)
        matches.add(m)
    else
        if point m.1 ∈ G_1 is unused then
            if no more matches available then
                addPutativeMatches(m.1, pq)
```

back

kd-tree Algorithm Analysis

Complexity
$O(n)$

$O(kn \log n)$

$O(1)$
$O(1)$

```
G_1 = read input
for i ← 2 to k do
    G_i = read input
    T_i = makeKDTree(G_i)
pq = new PriorityQueue
matches = new ArrayList
foreach p_i ∈ G_1 do
    addPutativeMatches(p_i, pq)
while pq not empty do
    m = pq.poll()
    if all points are unused then
        foreach i ≤ k do
            T_i.remove(m.i)
        matches.add(m)
    else
        if point m.1 ∈ G_1 is unused then
            if no more matches available then
                addPutativeMatches(m.1, pq)
```

back

kd-tree Algorithm Analysis

Complexity

$O(n)$

$O(kn \log n)$

$O(1)$
$O(1)$
$O(n)$

G_1 = read input
for $i \leftarrow 2$ **to** k **do**
    G_i = read input
    T_i = makeKDTree(G_i)

pq = new PriorityQueue
$matches$ = new ArrayList
foreach $p_i \in G_1$ **do**
    addPutativeMatches(p_i, pq)

while *pq not empty* **do**
    $m = pq$.poll()
    if *all points are unused* **then**
        foreach $i \leq k$ **do**
            T_i.remove($m.i$)
        $matches$.add(m)
    else
        if *point* $m.1 \in G_1$ *is unused* **then**
            if *no more matches available* **then**
                addPutativeMatches($m.1$, pq)

back

kd-tree Algorithm Analysis

Complexity

Complexity	Algorithm
$O(n)$	G_1 = read input
	for $i \leftarrow 2$ **to** k **do**
	G_i = read input
$O(kn \log n)$	T_i = makeKDTree(G_i)
$O(1)$	pq = new PriorityQueue
$O(1)$	*matches* = new ArrayList
$O(n)$	**foreach** $p_i \in G_1$ **do**
$O(n^{k-1} \log n)$	addPutativeMatches(p_i, pq)

```
while pq not empty do
    m = pq.poll()
    if all points are unused then
        foreach i ≤ k do
            T_i.remove(m.i)
        matches.add(m)
    else
        if point m.1 ∈ G_1 is unused then
            if no more matches available then
                addPutativeMatches(m.1, pq)
```

back

kd-tree Algorithm Analysis

Complexity	
$O(n)$	G_1 = read input
	for $i \leftarrow 2$ **to** k **do**
	G_i = read input
$O(kn \log n)$	T_i = makeKDTree(G_i)
$O(1)$	pq = new PriorityQueue
$O(1)$	$matches$ = new ArrayList
	foreach $p_i \in G_1$ **do**
$O(n^k \log n)$	addPutativeMatches(p_i, pq)
$O(n^k)$	**while** pq *not empty* **do**

```
G1 = read input
for i ← 2 to k do
    Gi = read input
    Ti = makeKDTree(Gi)
pq = new PriorityQueue
matches = new ArrayList
foreach pi ∈ G1 do
    addPutativeMatches(pi, pq)
while pq not empty do
    m = pq.poll()
    if all points are unused then
        foreach i ≤ k do
            Ti.remove(m.i)
        matches.add(m)
    else
        if point m.1 ∈ G1 is unused then
            if no more matches available then
                addPutativeMatches(m.1, pq)
```

back

kd-tree Algorithm Analysis

Complexity

Complexity	Algorithm
$O(n)$	G_1 = read input
	for $i \leftarrow 2$ **to** k **do**
$O(kn \log n)$	G_i = read input
	T_i = makeKDTree(G_i)
$O(1)$	pq = new PriorityQueue
$O(1)$	*matches* = new ArrayList
	foreach $p_i \in G_1$ **do**
$O(n^k \log n)$	addPutativeMatches(p_i, pq)
$O(n^k)$	**while** *pq not empty* **do**
$O(\log n)$	$m = pq$.poll()
	if *all points are unused* **then**
	foreach $i \leq k$ **do**
	T_i.remove($m.i$)
	matches.add(m)
	else
	if *point* $m.1 \in G_1$ *is unused* **then**
	if *no more matches available* **then**
	addPutativeMatches($m.1$, pq)

back

kd-tree Algorithm Analysis

Complexity	
$O(n)$	G_1 = read input
	for $i \leftarrow 2$ **to** k **do**
$O(kn \log n)$	G_i = read input
	T_i = makeKDTree(G_i)
$O(1)$	pq = new PriorityQueue
$O(1)$	$matches$ = new ArrayList
	foreach $p_i \in G_1$ **do**
$O(n^k \log n)$	addPutativeMatches(p_i, pq)
	while *pq not empty* **do**
$O(n^k \log n)$	m = pq.poll()
	if *all points are unused* **then**
	foreach $i \leq k$ **do**
$O(n(k \log n + \log n))$	T_i.remove($m.i$)
	$matches$.add(m)
	else
	if *point* $m.1 \in G_1$ *is unused* **then**
	if *no more matches available* **then**
	addPutativeMatches($m.1$, pq)

kd-tree Algorithm Analysis

Complexity	
$O(n)$	G_1 = read input
	for $i \leftarrow 2$ **to** k **do**
$O(kn \log n)$	G_i = read input
	T_i = makeKDTree(G_i)
$O(1)$	pq = new PriorityQueue
$O(1)$	*matches* = new ArrayList
	foreach $p_i \in G_1$ **do**
$O(n^k \log n)$	addPutativeMatches(p_i, pq)
	while *pq not empty* **do**
$O(n^k \log n)$	m = pq.poll()
	if *all points are unused* **then**
	foreach $i \leq k$ **do**
$O(n(k \log n + \log n))$	T_i.remove($m.i$)
$O(n^k - n)$	*matches*.add(m)
	else
	if *point* $m.1 \in G_1$ *is unused* **then**
	if *no more matches available* **then**
	addPutativeMatches($m.1$, pq)

back

kd-tree Algorithm Analysis

Complexity	
$O(n)$	G_1 = read input
	for $i \leftarrow 2$ **to** k **do**
$O(kn \log n)$	G_i = read input
	T_i = makeKDTree(G_i)
$O(1)$	pq = new PriorityQueue
$O(1)$	$matches$ = new ArrayList
	foreach $p_i \in G_1$ **do**
$O(n^k \log n)$	addPutativeMatches(p_i, pq)
	while *pq not empty* **do**
$O(n^k \log n)$	m = pq.poll()
	if *all points are unused* **then**
	foreach $i \leq k$ **do**
$O(n(k \log n + \log n))$	T_i.remove($m.i$)
$O(n^k - n)$	$matches$.add(m)
	else
	if *point* $m.1 \in G_1$ *is unused* **then**
$O(n^{k-1} \log n)$	**if** *no more matches available* **then**
	addPutativeMatches($m.1$, pq)

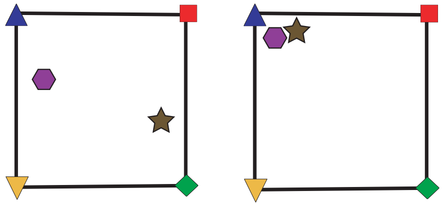
kd-tree Algorithm Analysis

Complexity	
$O(n)$	G_1 = read input
	for $i \leftarrow 2$ **to** k **do**
$O(kn \log n)$	G_i = read input
	T_i = makeKDTree(G_i)
$O(1)$	pq = new PriorityQueue
$O(1)$	*matches* = new ArrayList
	foreach $p_i \in G_1$ **do**
$O(n^k \log n)$	addPutativeMatches(p_i, pq)
	while *pq not empty* **do**
$O(n^k \log n)$	$m = pq$.poll()
	if *all points are unused* **then**
	foreach $i \leq k$ **do**
$O(n(k \log n + \log n))$	T_i.remove($m.i$)
	matches.add(m)
	else
	if *point* $m.1 \in G_1$ *is unused* **then**
	if *no more matches available* **then**
$O(n^{2k-1} \log n)$	addPutativeMatches($m.1, pq$)

$O\left(n^{2k-1} \log n + kn^{k+1} \log n + kdn^{k+1}\right)$

back

Measuring Matches: Convex Hull



(a) (b)

- Avoids TSP encountered with perimeter
- $\Omega(n^{\lfloor d/2 \rfloor})$ for $d > 3$

back

kd-tree Data Structure

kd-trees

- Multi-dimensional data structure introduced by Bentley (1975)
- Based on binary search trees
- Each level i divides the search space in dimension $i \mod d$

kd-tree Data Structure

Insert

- Search for node in the tree, if not found, add node
- Average cost: $O(\log n) \approx 1.386 \log_2 n$ (by Knuth)
- Can use Insert to build kd-tree
  - Inserting random nodes to build kd-tree is statistically similar to building bst
  - Build cost: $O(n \log n)$ for sufficiently random nodes

Optimize

- Given all n nodes, build an optimal kd-tree
- Uses the median for each dimension as discriminator for that level
- $O(n \log n)$ running time
- Maximum path length: $\lfloor \log_2 n \rfloor$

kd-tree Data Structure

Delete

- Must replace node with j-max element of left tree or j-min element of right tree
- Worst Case Cost: $O(n^{1-1/d})$, dominated by find min/max
- Average Delete Cost: $O(\log n)$

Nearest Neighbor Queries

- Bentley's Original algorithm: empirically $O(\log n)$ (redacted)
- Friedman and Bentley: empirically $O(\log_2 n)$
- Lee and Wong ('80): Worst case: $O(n^{1-1/k})$

back

Proof of Correctness: Centroid

We want to find r such that given m with k points,

$$s = max\left(\sum_{l=1}^{d}(p_l - c_l(m))^2\right), \forall p \in m,$$

where s is the maximum Euclidean distance to centroid.

- First, consider $size(m) \leq ks^2$. Remember,

$$size(m) = \sum_{i=1}^{k}\left(\sum_{j=1}^{d}\left(p_{i,j} - c_j(m)\right)^2\right)^2$$

Since $\sum_{j=1}^{d}\left(p_{i,j} - c_j(m)\right)^2 \leq s$ for all i, this is trivially true.

Proof of Correctness: Centroid

- Second, there exists p_j outside of r, with $p_j, p_i \in m'$. Let x and y be the distance from p_i and p_j to $c(m')$, respectively. By assumption, $x + y \geq r$. Since $size(m) \leq ks^2$, we show

$$size(m) \leq ks^2 \leq x^2 + y^2 \leq size(m').$$

With minimal $x + y$, $x + y = r$. Then we know

$$\frac{r^2}{2} \leq x^2 + y^2 \leq r^2.$$

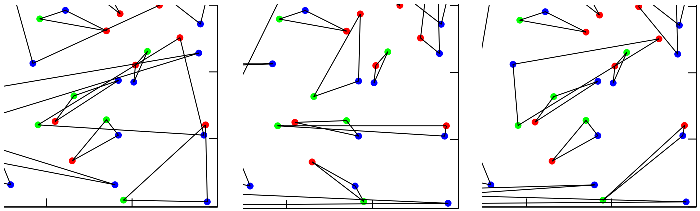
Therefore

$$\begin{aligned} ks^2 &\leq \frac{r^2}{2} \\ ks^2 &\leq \frac{k^2 s^2}{2} \\ 2 &\leq k. \end{aligned}$$

back

Centroid Measures

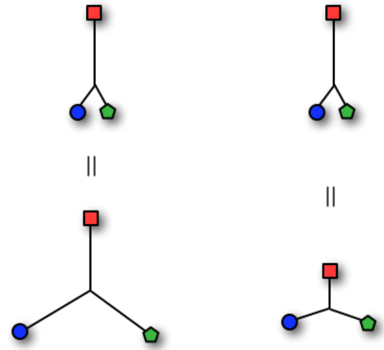
Visible differences between max, average (variance), and sum of squared distances to the centroid.



(c) Max distance (d) Average distance (e) Sum of squares

Centroid Measures

Equivalent matches under each measure to the centroid.



(f) Max distance (g) Average distance

back

kd-tree Empirical Performance