# kd-tree algorithm for k-point matching

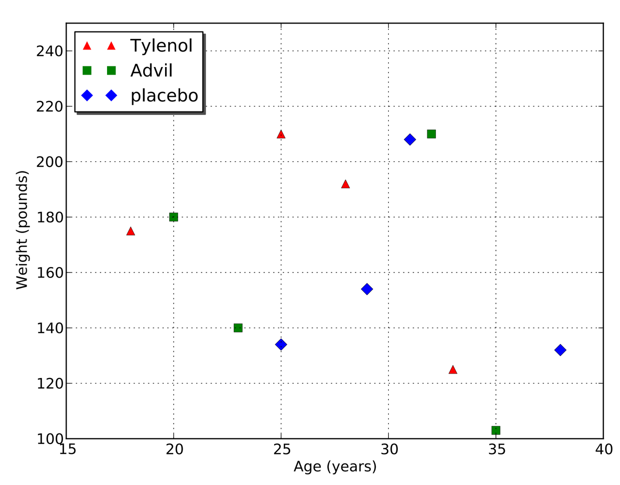

John R Hott, Nathan Brunelle, abhi shelat

University of Virginia
School of Engineering and Applied Science


## Motivation

### Drug Trials

- Every drug seeking FDA approval must go through Phase II and III clinical trial periods (on human participants) to determine safety and effectiveness [1].
- Drugs compared against other drugs and placebos to determine effectiveness
- Must compare participants with similar features to eliminate bias due to:
  - Age
  - Height
  - Gender
  - Weight
  - Ethnicity
- Consider the following example, with Tylenol and Advil compared with a placebo. Each participant is plotted in terms of weight and age:

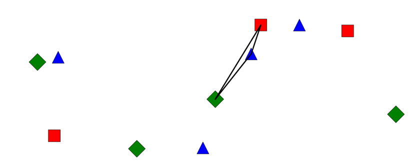



- Brute force solution:
  Try all possible combinations of people taking different drugs and pick the smallest ones
  Two possibilities:
  - Fast method: make all matches, sort, pick smallest
    $O(n^3 \log n)$ with $O(n^3)$ space complexity
    For 300 participants, there are 1,000,000 matches
  - Space efficient method: make smallest match, throw out those points, repeat
    $O(n^4)$ with $O(n)$ space complexity

- We want a faster solution that uses as little space as possible

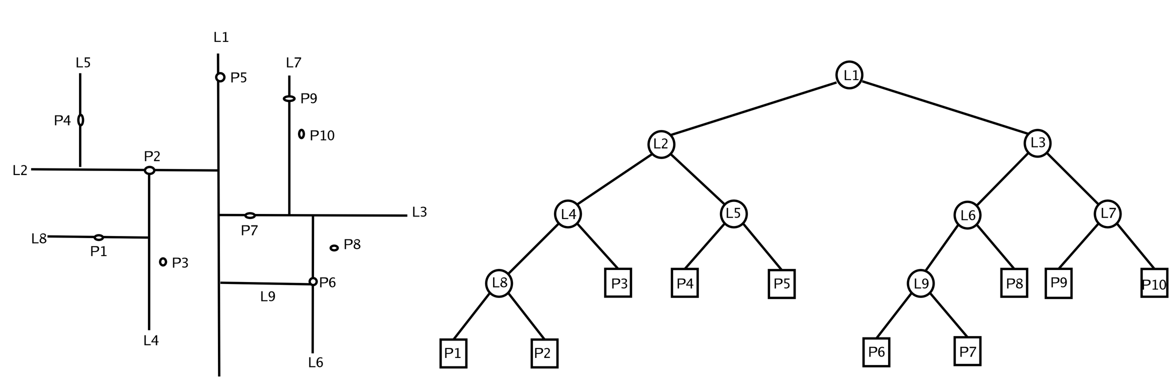
## Problem Statement

- Given a set of points, $K$, partitioned into $k$ sets of colors, $K_1, K_2, \ldots, K_k$, with $|K_1| = .. = |K_k| = n$
- Define a match $m = \{p_i \mid p_i \text{ in } K_i\}$ where $|m| = k$
  Each $m$ has one point from each color



- Find the smallest $n$ matches such that each point is only used once

## KD-Trees

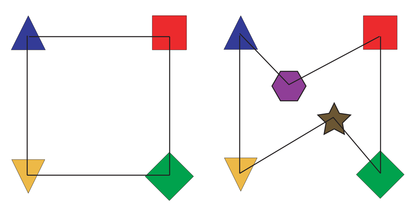



Division of points in the plane (left) and corresponding KD-tree (right) for a sample of points. [2]

- Partition $d$-dimensional space to create binary tree
  Level in the tree determines dimension partitioned
  Each non-leaf node in the tree defines a splitting hyperplane
- Designed for fast nearest-neighbor searching
  Worst case $O(n^{1-1/d})$ for $d$ dimensions
  Average case $O(\log n)$ for 2 dimensions
- Require one-time build cost of $O(n \log n)$

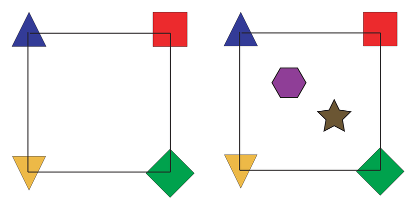
## Smallest Match Definition

### Perimeter



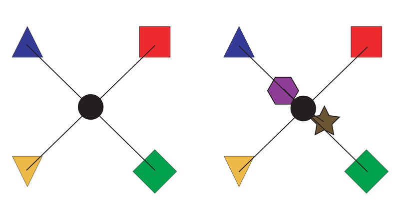
- Order-independent for up to 3 colors in $d$ dimensions
  $O(1)$ to compute
- Equivalent to Traveling Salesman in 2D as number of colors increases
  $(k-1)!/2$ $k$-gons formed for each match
- Not well defined in higher dimensions

### Convex Hull



- Order-independent for any number of colors
- Solves perimeter's TSP equivalence problem in 2D
- Still not well definied in higher dimensions

### Centroid



- Scales well to any dimensionality and number of colors
- Defined by the average of all points in the match
- Matches are intuitive
- We consider sum of squared distances to the centroid, as defined below:

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

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

## Algorithm

### Create Matches

- Creates the kd-trees
- Calls the addPutativeMatches subroutine for each first color point
- Possible matches are added to sorted PriorityQueue
- Matches are pulled in order
- If a match is invalid, and the first-color point no longer has matches in the queue, re-call addPutativeMatches for it

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

color_1 = read input from file
for i ← 2 to k do
    color_i = read input from file
    kd_i = makeKDTree(color_i)

pq = new PriorityQueue
matches = new ArrayList
foreach color_1 do
    addPutativeMatches(pq)

while pq not empty do
    x = pq.poll()
    if all points are clean then
        foreach color i do
            kd_i.remove(x.i)
        matches.add(x)
    else
        if color_1 is clean then
            if no more matches available then
                addPutativeMatches(pq)
```

### addPutativeMatches Subroutine

- Finds the closest point of each color to a given point with kd-tree lookups
- Creates a match of these points
- Searches a radius based on the size of this initial match for all possible points
- Makes matches with points found
- Returns 10 smallest matches

```
Input: PriorityQueue pq, current point pt_1 from color 1, kd-trees for each colors 2 to k
Output: list of 10 smallest matches for point pt_1

for i ← 2 to k do
    pt_i = kd_i.getnearest(pt_{i-1})

small = size(pt_1, pt_2, ... , pt_k)
search = get search distance from small
tq = new PriorityQueue
tq.add(match(pt_1,pt_2, ... ,pt_k))
for i ← 2 to k do
    list_i = kd_i.getnearest(pt_{i-1}, search)

foreach list_2 as pt_2 do
    foreach list_3 as pt_3 do
        ... foreach list_k as pt_k do
                dist = size(pt_1,pt_2, ... ,pt_k)
                if dist < small then
                    tq.add(match(pt_1,pt_2, ... ,pt_k))

for i ← 1 to 10 do
    m = tq.poll()
    pq.add(m);
```

## Algorithm Analysis

### Worst Case



- Occurs when first *k-1* colors are coincident with each other and color $k$ points asymptotically converge to a point within the search area of any match

$$\begin{aligned}
T_{apm_{k,d}} &= O((k-1)(dn^{1-\frac{1}{d}}) + \log n + (k-1)(n - 1 + n\log n) \\
&\quad + n^{k-1}\log n + 10(2\log n)) \\
&= O(n^{k-1}\log n + kn\log n + kdn) \\
T_{part1_{k,d}} &= O\left(nT_{apm_{k,d}}\right) \\
&= O\left(n^k \log n + kn^2\log n + kdn^2\right) \\
T_{part2_{k,d}} &= O\left(n^k\log n + (k-1)n\log n + \frac{n^k - n}{10}T_{apm_{k,d}}\right) \\
&= O\left(n^{2k-1}\log n + kn^{k+1}\log n + kdn^{k+1}\right)
\end{aligned}$$

- Worst case complexity: $O(n^{2k-1} \log n)$

### Expected Case

- On average, we assume that:
  $\exists\delta$ such that $\forall\epsilon$-sized areas, there are $\delta\epsilon n$ point in that region

  In other words, the points are evenly distributed and the number of points in any region is proportional to the size of the region.
- Therefore, we consider the number of points in any small region to be constant

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

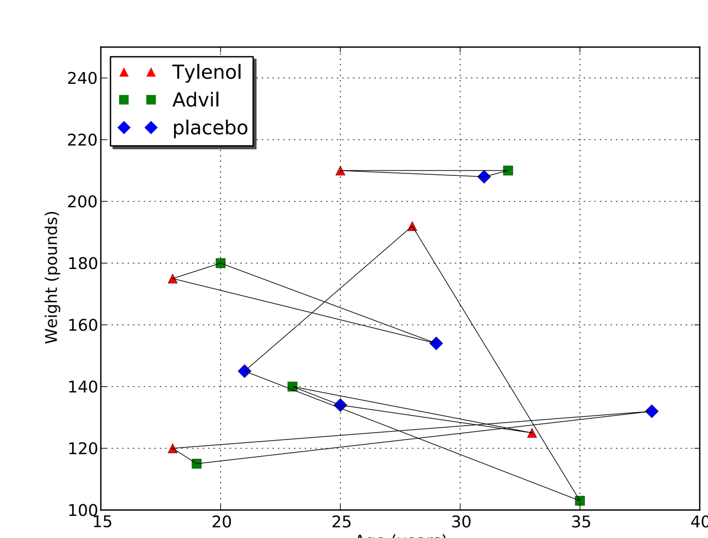
- Expected case complexity: $O(kdn^2)$

## Results

### 3 colors in 2 dimensions

- kd-tree algorithm outperforms brute force in expected case
  Brute Force: $O(n^3 \log n)$ with $O(n^3)$ space complexity
  $O(n^4)$ with $O(n)$ space complexity

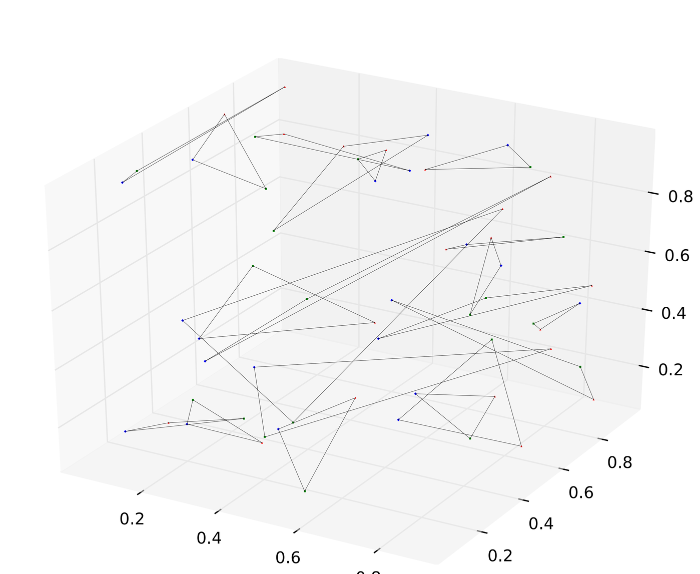
  Our Algorithm: $O(n^2)$ with $O(n)$ space complexity




### Arbitrary colors and dimensionality

- kd-tree algorithm outperforms brute force in expected case
  Brute Force: $O(dn^k \log n)$ with $O(n^k)$ space complexity
  $O(dn^{k+1})$ with $O(n)$ space complexity

  Our Algorithm: $O(kdn^2)$ with $O(n)$ space complexity




References
1. Blanchard Randall IV. *The u.s. drug approval process: A primer.* Congressional Research Service, June 1, 2001.
2. Kakde, Hemant. *Range Searching using Kd Tree.* 2008.