Fast Hierarchical Clustering via Dynamic Closest Pairs

David Eppstein

Dept. Information and Computer Science
Univ. of California, Irvine

http://www.ics.uci.edu/∼eppstein/

My Interest In Clustering

What I do: find better algorithms for previously-solved problems

(rarely, find algorithms for new problems)

What is a better algorithm?

- Produces better answers than previous solutions (according to well-defined quality measure)
- Produces the same answers, in less time (theoretically or in practice)

My Interest In Clustering (continued)

My main interests:

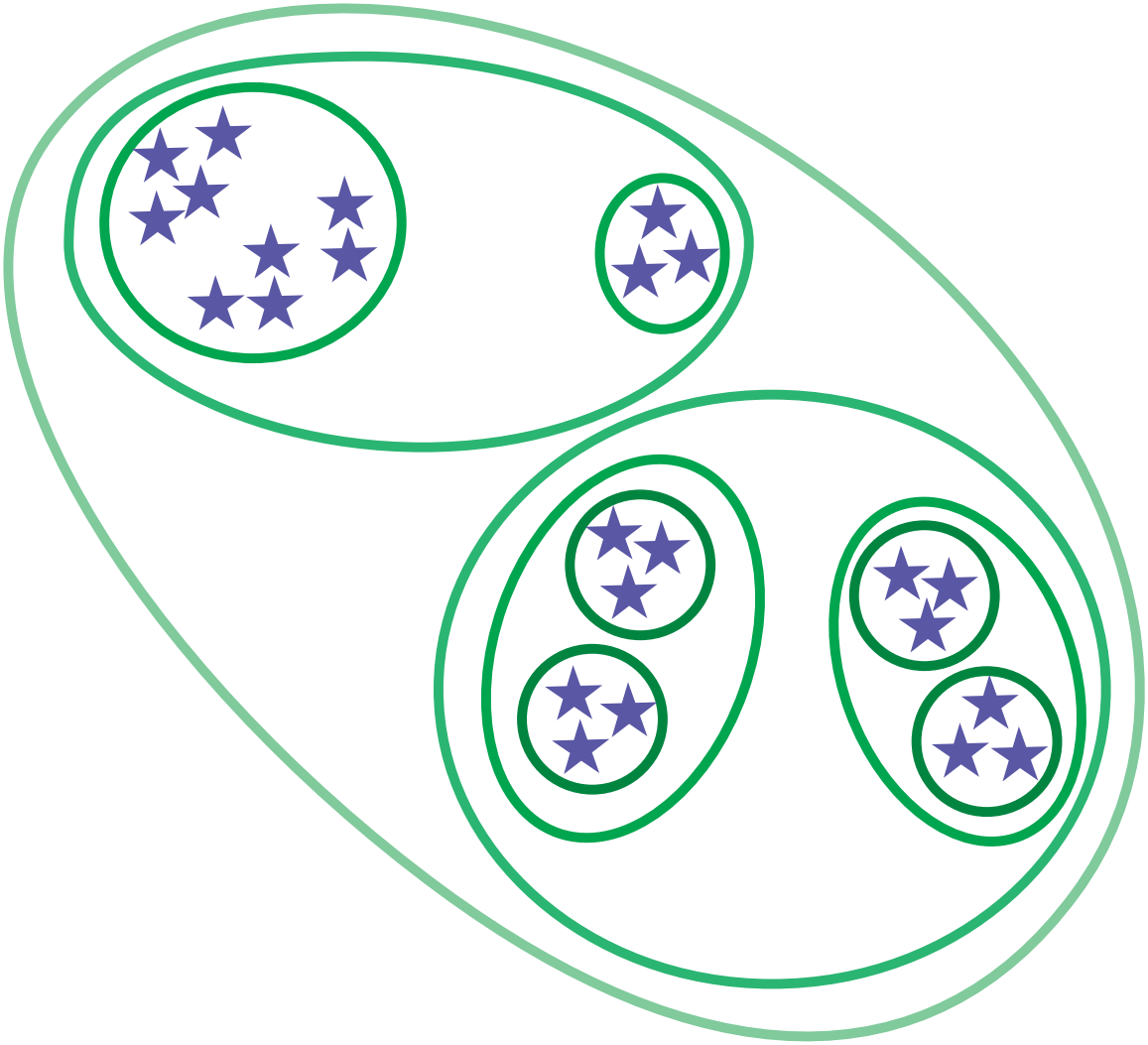
- Graph algorithms
- Computational geometry
- Computational molecular biology

Geometry and biology have both led to clustering

- Biology: motivation (evolutionary trees)
- Geometry: solution techniques

I. OVERVIEW OF THE PROBLEM

What is Hierarchical Clustering?



Nested family of sets of data points

Subset relation gives hierarchical structure

Overview of Clustering Techniques

- Top down:
  find binary partition of input
  recursively cluster each side

- Incremental:
  Add points one at a time
  Follow hierarchy to good branch point

- Bottom up:
  Find points which belong together
  Merge them into a cluster
  Continue merging clusters until one left

Top-down or incremental ok for search
(test if point exists; find nearby neighbor)

Bottom-up best for cluster analysis
but slow – can't run on large data sets

My goal: speed up bottom-up clustering

Bottom-up Clustering Algorithm

Given n objects (data points, DNA sequences, etc)

Form n single-object clusters

Repeat $n - 1$ times:

- Find two “nearest” clusters
- Merge them into one supercluster

Different clustering algorithms
(UPGMA, Ward’s, neighbor-joining etc)
based on different definitions of “nearest”.

Slow part: finding nearest clusters

Formalization

Given set S of objects (or clusters), undergoing insertions and deletions of objects, and given a distance function $d(x, y)$

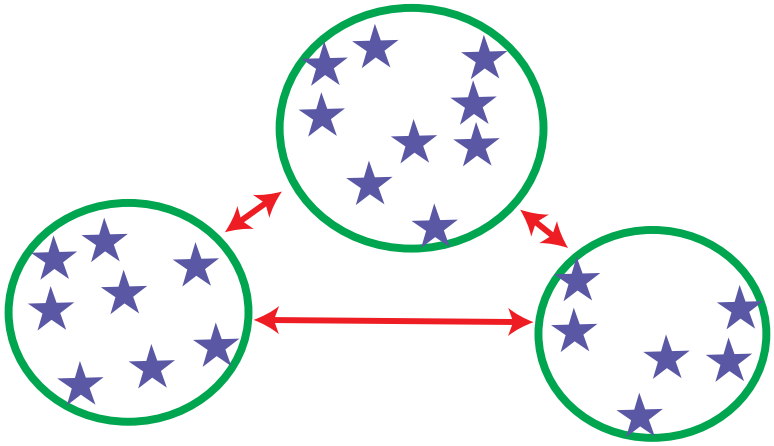
As set of objects changes, the pair (x, y) that minimizes the value $d(x, y)$ will also change

We want a data structure to quickly find this pair

(Then clustering can be performed by a sequence of $n - 1$ closest-pair queries, $2(n - 1)$ deletions of clusters, and $n - 1$ insertions of new superclusters)

What Can We Assume About Distances?

Need not satisfy triangle inequality:



Distance between two clusters may be much larger
than sum of distances to third cluster

Not usually monotonic

(closest distance may go up or down over course of algorithm)

But, safe to assume symmetry

(if $d(x, y) \neq d(y, x)$, redefine $d^*(x, y) = \min(d(x, y), d(y, x))$)

How Fast is the Distance Function?

We don't want our algorithms to make assumptions about distance function (to keep them as general as possible)

But, to analyze their running time, we need to know time per distance function evaluation.

Assumption: distance eval takes constant time

Not true in general!
(e.g. high-dimensional vectors, sequence alignment...)

What if it's not true?

- Interpret analysis as predicting number of distance evaluations rather than program runtime
- If enough extra memory available, compute and store distance matrix, then perform each distance eval by matrix lookup

II. PREVIOUSLY KNOWN SOLUTIONS

Brute Force

Just keep list of points in the set

To find closest pair, loop through all pairs

Time per update: $O(1)$
Time per query: $O(n^2)$

Easy to program but slow

Neighbor Heuristic

Each point stores its nearest neighbor

To insert: compute nearest neighbor of new point; for each old point, check if new point is nearer than old neighbor.

To delete: for each old point, if deleted point was neighbor, find new neighbor

To find closest pair: loop through all neighbors

Time per insert: $O(n)$
Time per deletion: $O(nk)$
Time per query: $O(n)$

> $k =$ points for which deleted point was neighbor;
> $k = O(1)$ expected case, $k = \min(3^d, n)$ worst case.
> Worst case: all points have same neighbor.

Not too complicated; ok in practice;
theoretically unreliable and unsatisfactory

Priority Queue

Maintain priority queue (e.g. binary heap)
of distance matrix values

Time per update: $O(n)$ PQ changes, $O(n \log n)$ total
Time per query: $O(\log n)$
Space: $O(n^2)$

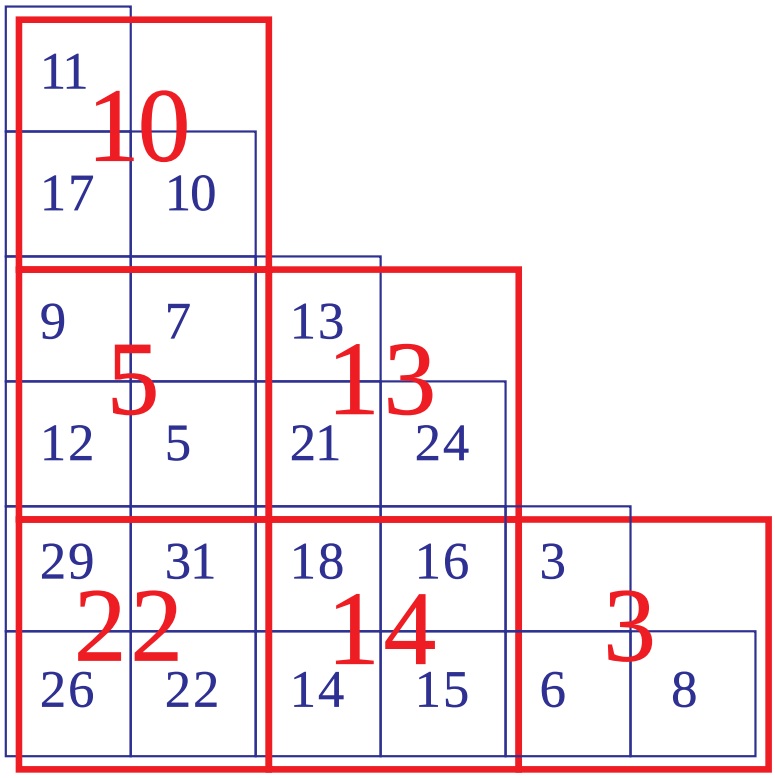
Complicated; ok in theory, but uses lots of space
Probably slower than neighbors in practice

III. NEW SOLUTIONS

Quadtree

Create lower triangular distance matrix

Overlay with coarser lower triangular matrix
value in coarse cell = min of four distances



Treate coarse matrix as set of distances on half as many points, maintain closest pair recursively

Quadtree Analysis

Insertion

Compute $n - 1$ new distances
Recompute distances in $n/2$ coarse matrix cells
then $n/4$ cells at next level, etc. Total: $2n - 1$ distance computations

Deletion

Recompute distances in $n/2$ coarse matrix cells
then $n/4$ cells at next level, etc. Total: $n - O(1)$ distance computations

Closest Pair Lookup

Find closest pair at base of recursion
At each level, find which value gives min
Total: $O(\log n)$

Conga Line Data Structure

Partition points into $\log n$ subsets

For each subset S_i, maintain digraph G_i
with edges connecting S_i and rest of points

(initially S_i is a path, becomes a set of paths as points are deleted and edges get removed)

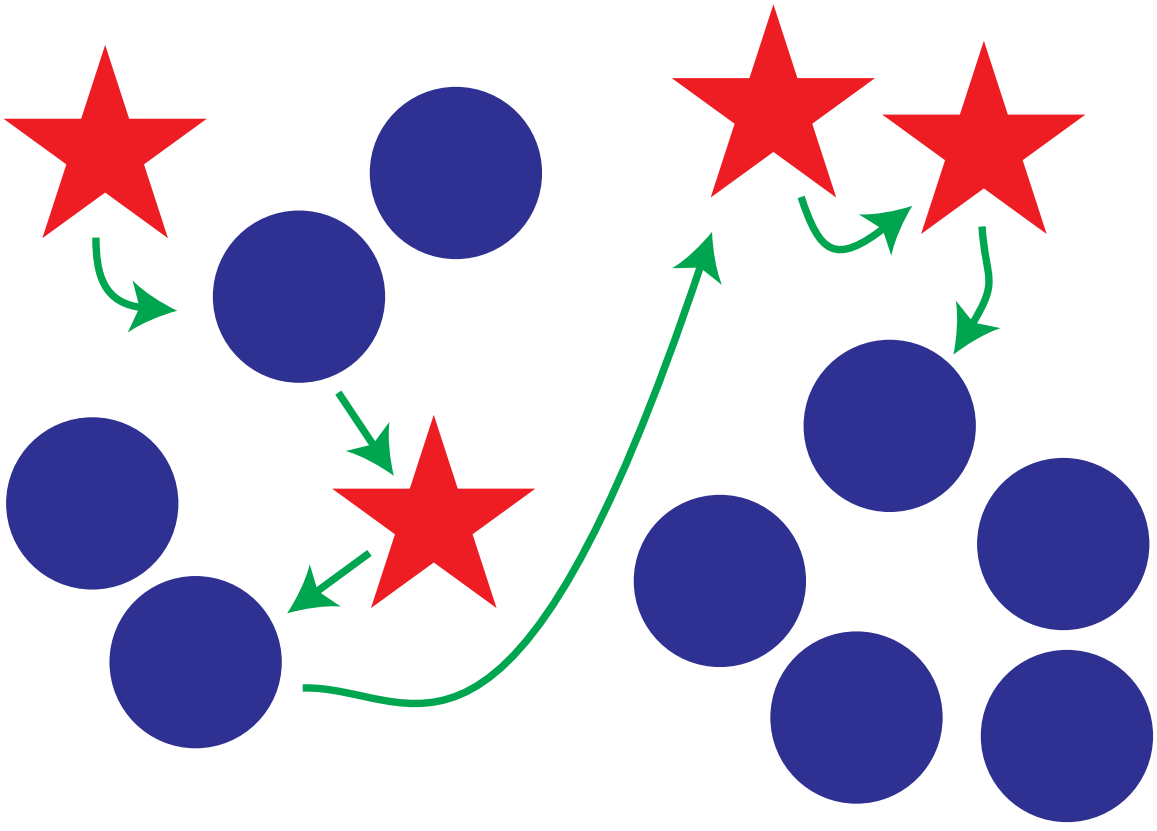
Closest pair will be guaranteed to form an edge in one of these graphs ($O(n)$ time per query)

[Simplified from geometric bichromatic closest pair data structure in Eppstein, *Disc. & Comp. Geom.* 1996, by removing the geometry and the colors and relaxing conditions on the sizes of the subsets.]

Conga line for a subset

Given subset of some of the objects,
choose any object to start the path

End of path chooses its favorite unchosen object
(if not in subset, must choose within subset)



Lemma: if $d(s,t)$ is minimized, and one of s or t is in the subset, then either s chooses t or t chooses s.

Conga line insertions

To insert an object:

Make new singleton subset

If too many subsets, merge two that are closest in size

Recompute conga lines

Analysis:

Each time object is involved in a recomputation, subset size increases by a constant factor, so $O(\log n)$ recomputations

Each time object is involved in a recomputation, takes $O(n)$ time to find its neighbor

Total per insertion: $O(n \log n)$

Conga line deletions

To remove an object:

Remove it from $O(\log n)$ conga lines (breaking each line in two)

Treat neighbors at broken ends of lines as if they were newly inserted objects

Analysis:

Each deletion causes $O(\log n)$ insertions

Total time per deletion: $O(n \log^2 n)$

Modified Conga Lines

Multi-Set Conga: never merge subsets

FastPair: when deletion would create a subset of k points, instead create k singleton subsets

(FastPair is very similar to neighbor heuristic, but creates initial neighbor values differently, and insertion never changes old neighbor values)

Insertion time: $O(n)$
Deletion time: $O(n)$ expected, $O(n^2)$ worst-case
Query time: $O(n)$

(Similar analysis to neighbor heuristic; which is best needs to be determined empirically.)

IV. EXPERIMENTAL RESULTS AND CONCLUSIONS

Hierarchical Clustering in $\mathbf{R}^{20}$

	BruteForce	Neighbors	Quadtree	CongaLine	Multiset	FastPair
$n = 250$	5.76s	0.60s	**0.36s**	1.09s	0.38s	**0.36s**
500	53.80s	2.48s	1.71s	5.98s	1.65s	**1.52s**
1000	456.98s	10.24s	7.94s	28.17s	7.10s	**6.75s**
2000	4145.91s	46.41s		154.25s	35.35s	**31.88s**
4000		204.14s		785.14s	165.58s	**148.76s**
8000		841.34s		3644.60s	747.80s	**659.85s**
16000		3337.03s			3051.22s	**2709.94s**

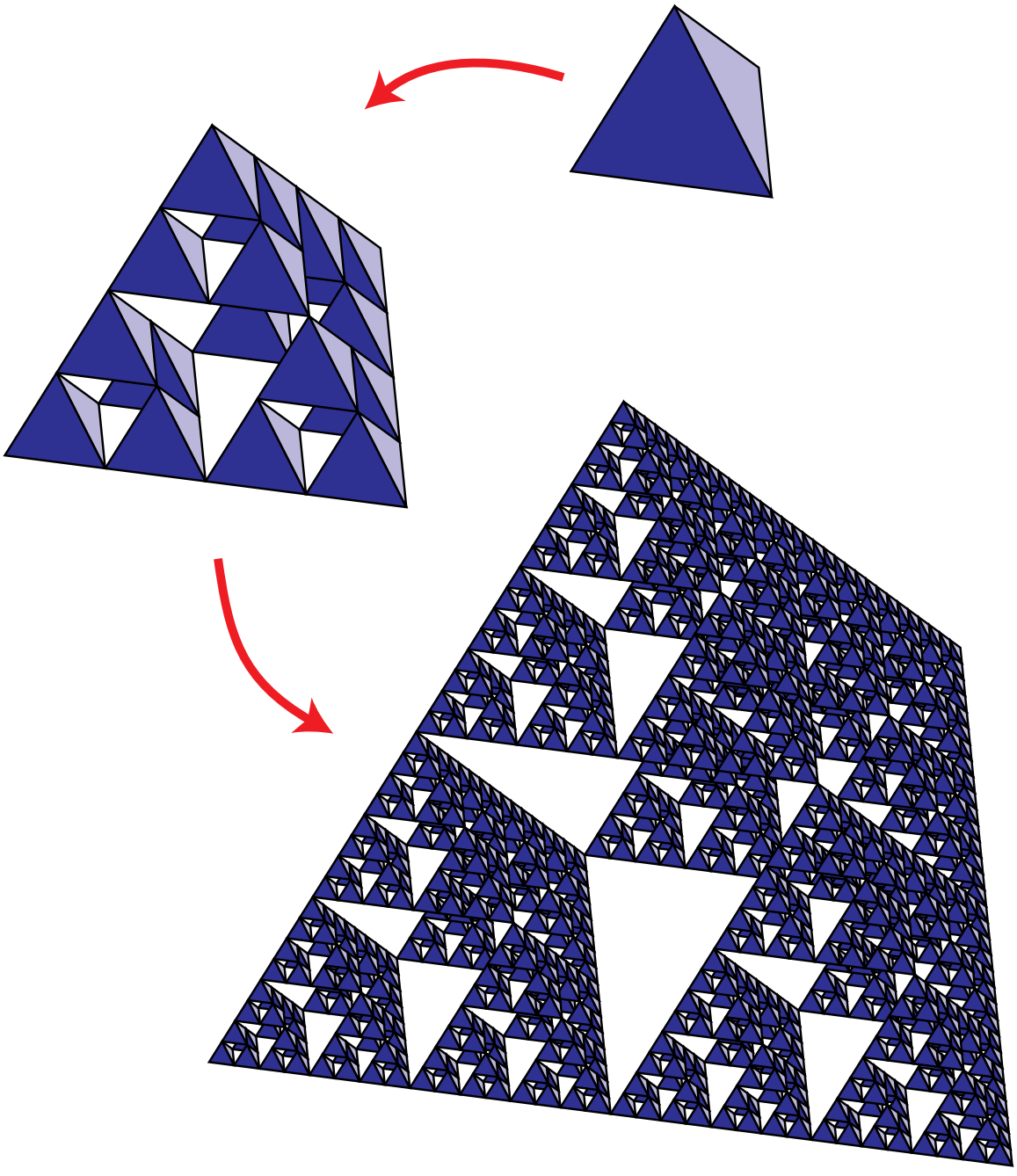
Clusters are combined by unweighted medians.
Points placed uniformly at random in the unit hypercube.

Times include only the construction of the closest pair data structure and algorithm execution (not the initial point placement) and are averages over ten runs.

The quadtree data structure was only run on data sets of 1000 or fewer points due to its high storage requirements.

Code was written in C++, compiled and optimized by Metrowerks Codewarrior 10, and run on a 200MHz PowerPC 603e processor (Apple Powerbook 3400c).

Sierpinski Tetrahedron



Hierarchical Clustering in a 31-dimensional Fractal

	BruteForce	Neighbors	Quadtree	CongaLine	Multiset	FastPair
$n = 250$	12.71s	0.67s	**0.52s**	2.05s	0.68s	0.59s
500	107.90s	3.18s	**2.51s**	10.79s	3.03s	2.72s
1000	926.06s	14.38s	**11.18s**	55.67s	13.62s	12.41s
2000		61.26s		278.97s	64.07s	**56.79s**
4000		244.23s		1227.56s	269.56s	**233.05s**
8000		1014.02s		5354.00s	1128.76s	**972.92s**
16000		4492.64s			4624.10s	**4152.42s**

Clusters are combined by unweighted medians.
Points placed uniformly at random in the 31-dimensional generalized Sierpinski tetrahedron (formed by choosing 5 random binary values and taking bitwise exclusive ors of each nonempty subset)

Times include only the construction of the closest pair data structure and algorithm execution (not the initial point placement) and are averages over ten runs.

The quadtree data structure was only run on data sets of 1000 or fewer points due to its high storage requirements.

Code was written in C++, compiled and optimized by Metrowerks Codewarrior 10, and run on a 200MHz PowerPC 603e processor (Apple Powerbook 3400c).

Analysis of Experimental Data

Brute Force

Theoretically and in practice, takes time $O(n^3)$
Never the best choice

Quadtree

Theoretically and in practice, takes time $O(n^2)$
Computes few distances but high overhead
Good for small n, expensive distance computations

Conga Line

Theoretically takes time $O(n^2 \log^2 n)$
In practice, time seems to be $O(n^2 \log n)$
Good for wierd distances when other methods fail

Neighbors, Multiset, FastPair

Theoretically, worst case $O(n^3)$
In practice, time seems to be $O(n^2)$
FastPair is generally best of these three

Other Applications

Traveling Salesman Problem heuristics

Multi-Fragment: find shortest edge between endpoints of two different paths

Cheapest Insertion: find pair (edge xy in tour, vertex z not in tour) minimizing $xz + yz - xy$

Greedy matching

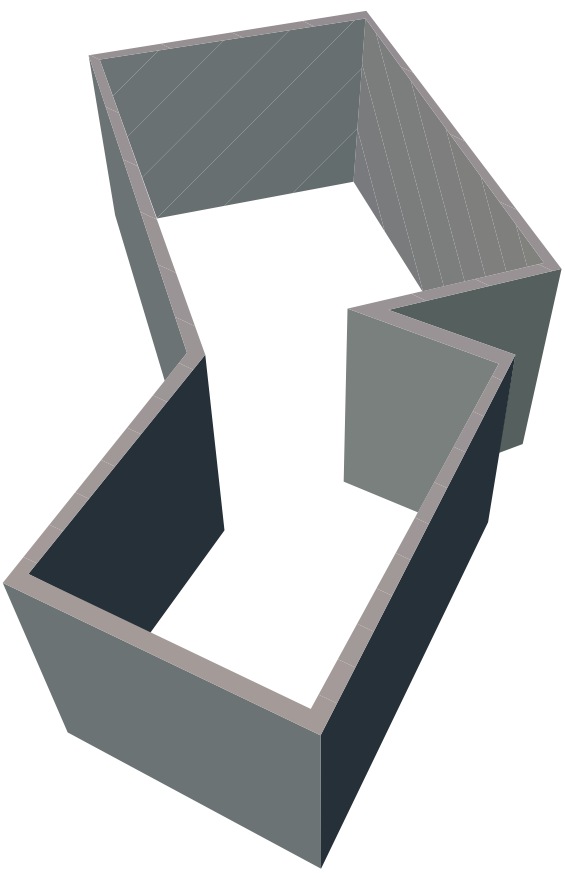
Computational symbolic algebra?

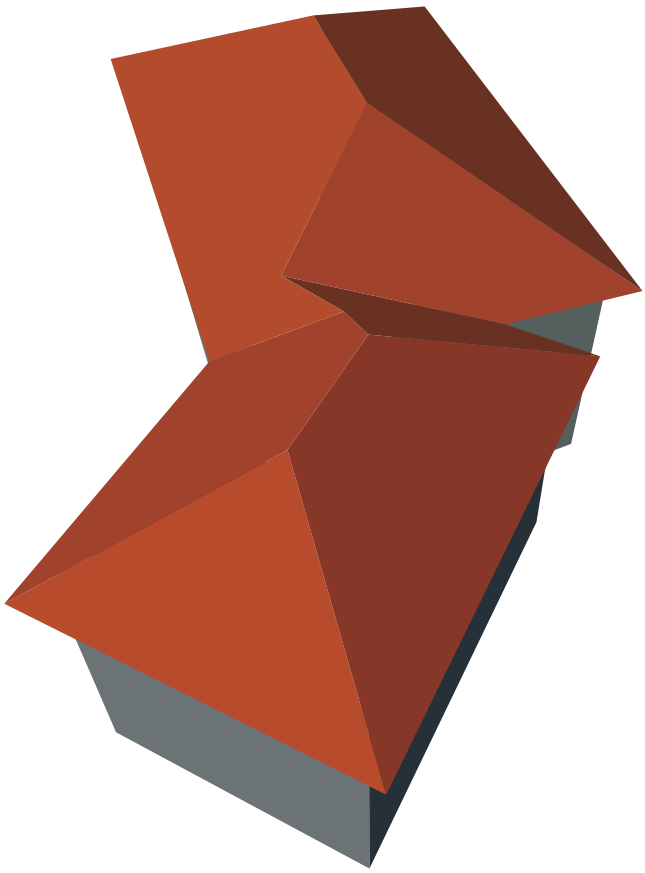
Gröbner Basis algorithm repeatedly interacts pairs of polynomials; use data structures to find best pair

Building roof design

(joint work with J. Erickson)

How to fit a roof to these walls?





Future Work

More experiments

Real data?

Account for cache size effects

At certain problem sizes, runtime jumps
probably due to data exceeding cache size

All methods repeatedly scan memory
Instead, process memory in cache-sized chunks

Neighbor-Joining

Clustering method used in computational biology
Distances are linear functions: $d(i,j) = a_{ij}n + b_{ij}$

Typical impl. $O(n^4)$ but easily improved to $O(n^3)$

Maintain convex hull of points (a_{ij}, b_{ij})
Minimum distance = binary search in hull
Total time: $O(n^2 \log n)$

Implementation and experimentation needed