Constructing the delaunay skeleton in medium dimension

Rouge-Gazon
30 janvier, 2009

J.-D. Boissonnat, O. Devillers et S. Hornus

Definitions

P is a set of n points in $\mathbb{R}^d$.
A simplex σ is the convex hull of d+1 aff. ind. points.
q $\in \mathbb{R}^d$ and simplex σ conflict iff q is inside the circumsphere of σ.

A Delaunay triangulation of P, Del(P), is a maximal set of simplices with vertex-set in P, that conflict with no point of P.

Definition & motivation

The Delaunay graph, or Delaunay skeleton, or the graph, is the set of vertices and edges of the Delaunay triangulation.

In theory, this takes $O(n^2)$ worst case space, instead of the $O(n^{\lceil \frac{d}{2} \rceil})$ worst case space necessary for the full triangulation.

In practice, it does indeed shrink memory usage (and lengthen the construction time).

Main observation

Given the full Delaunay triangulation, a simplex, it takes constant time (in the d-cell/vertex representation) to access a neighbor.

Given the Delaunay graph and the vertex-set of a Delaunay simplex (Del-simplex) σ, it is possible to find the vertex-set of any neighbor of σ relatively quickly.

Applications

- We've computed a huge space-time (4D) Delaunay skeleton of a sample of some moving object [J-P Pons and E. Aganj]. We want to extract "slices" at time=constant. We extract only the relevant 4-simplices.
- Compute the graph in RAM while "streaming" the full triangulation to disk.

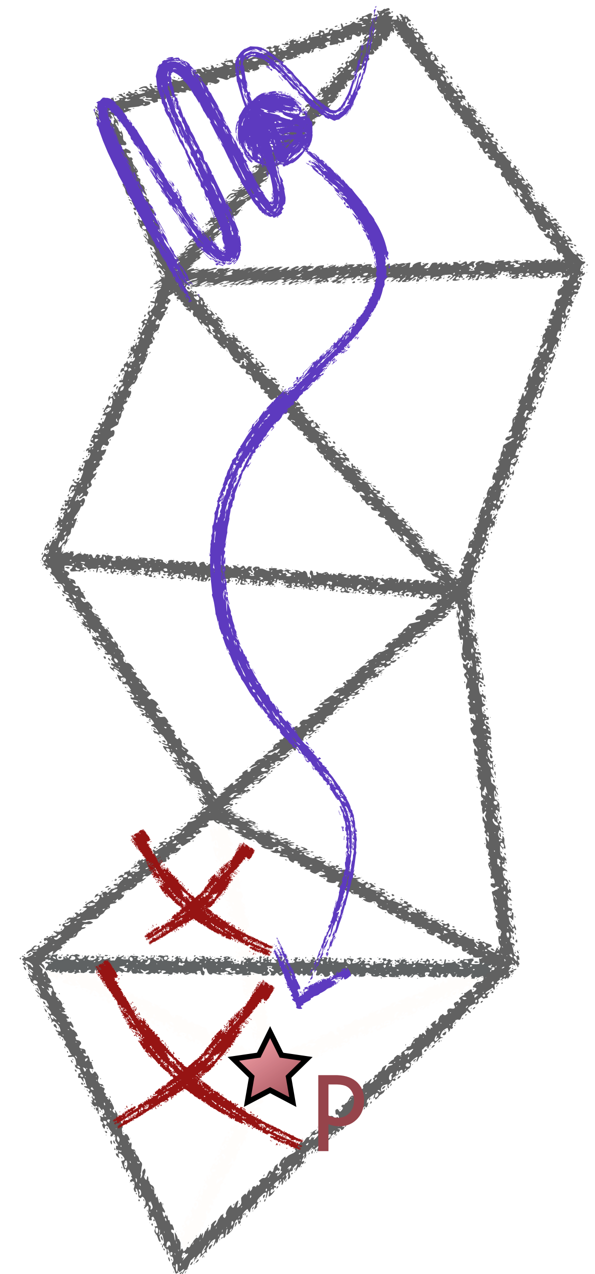
Sequel outline

1. Recap: incremental construction of Delaunay triangulation.
2. How to compute the Delaunay graph.
3. How to compute a neighbor of a Delaunay simplex.
4. Making it usable (optimizations).
5. Average case experiments (uniform random distribution)

Delaunay construction (recap)

Incremental algorithm: inserts points one after the other.

1. Walk in the triangulation towards the new point p. Stop upon finding a Del-simplex that conflicts with p.
2. Identify the set of Del-simplices that conflict with p (they form the conflict zone; a star-shaped geometric subset of $\mathbb{R}^d$).
3. Remove the conflicting Del-simplices



Delaunay graph construction

Incremental algorithm: inserts points one after the other.

- We basically use the same algorithm.
- To do so, we need a fast algorithm for the get_neighbor() function (see later slide).
- As we explore the conflicting Del-simplices, we do keep them in memory, in order to:
- Update the adjacencies of all the vertices on the conflict zone's boundary (see next slide).
- Throw away the now-outdated conflicting simplices.

Update of the adjacencies

- C is the conflict zone: the set of conflicting simplices.
- Let E be the set of Delaunay edges in C.
- Let B be the set of Delaunay edges in the boundary of C.
- For each edge *a*—*b* in E\B: disconnect vertices *a* and *b*.
- Let *v* be the newly created vertex. connect *v* to each vertex in B.

get_neighbor()

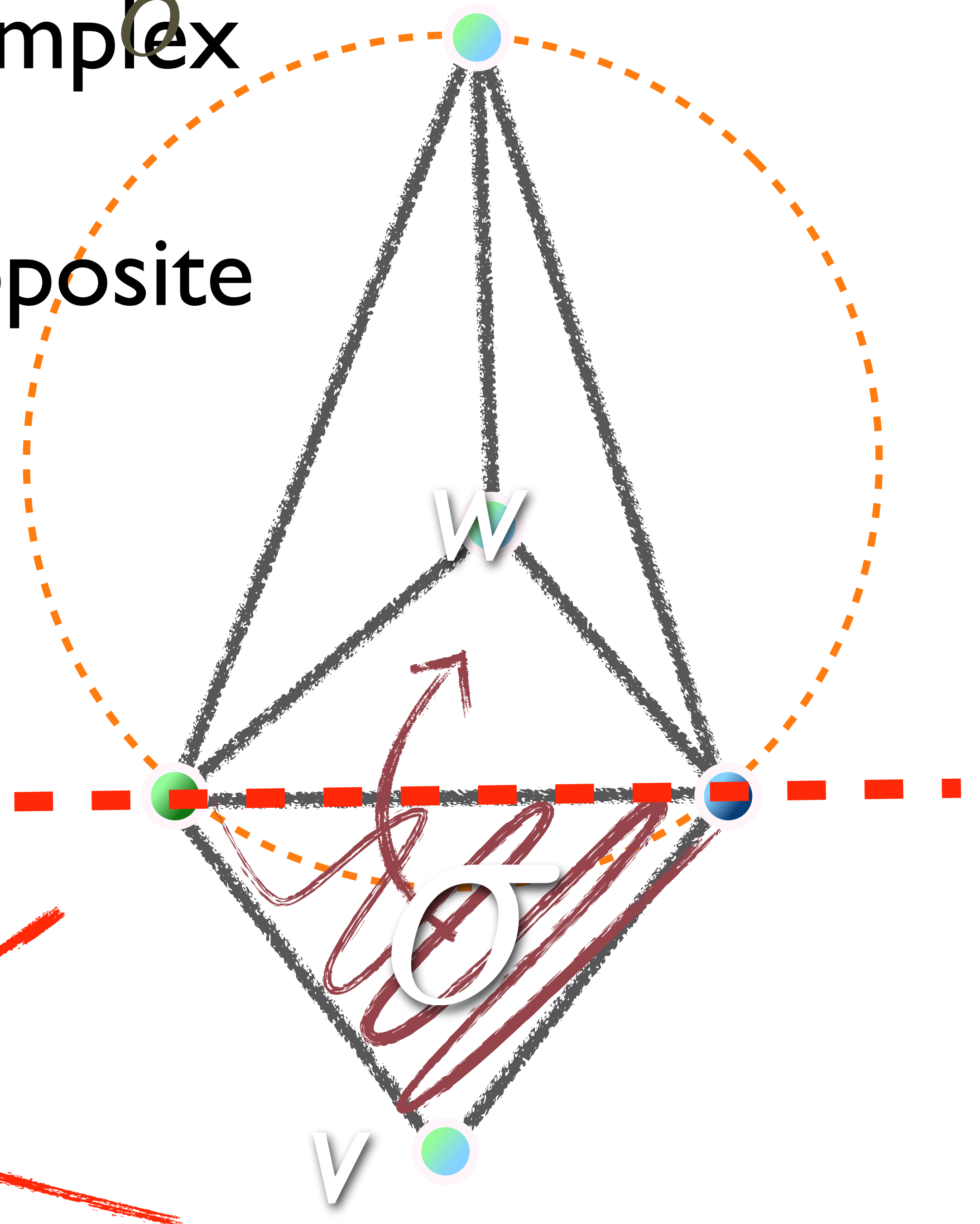
Input: the vertex-set of a Del-simplex σ and a vertex *v* of σ.

Output: the unique vertex *w* opposite

Compute: $\bigcap_{p \in \sigma \setminus \{v\}} \mathrm{Neighbors}(p)$

Remove: vertices on the wrong side.

Use in_circle() predicate: to select the correct opposite vertex.



Optimizing lists intersection

get_neighbor(): compute intersection of d lists.

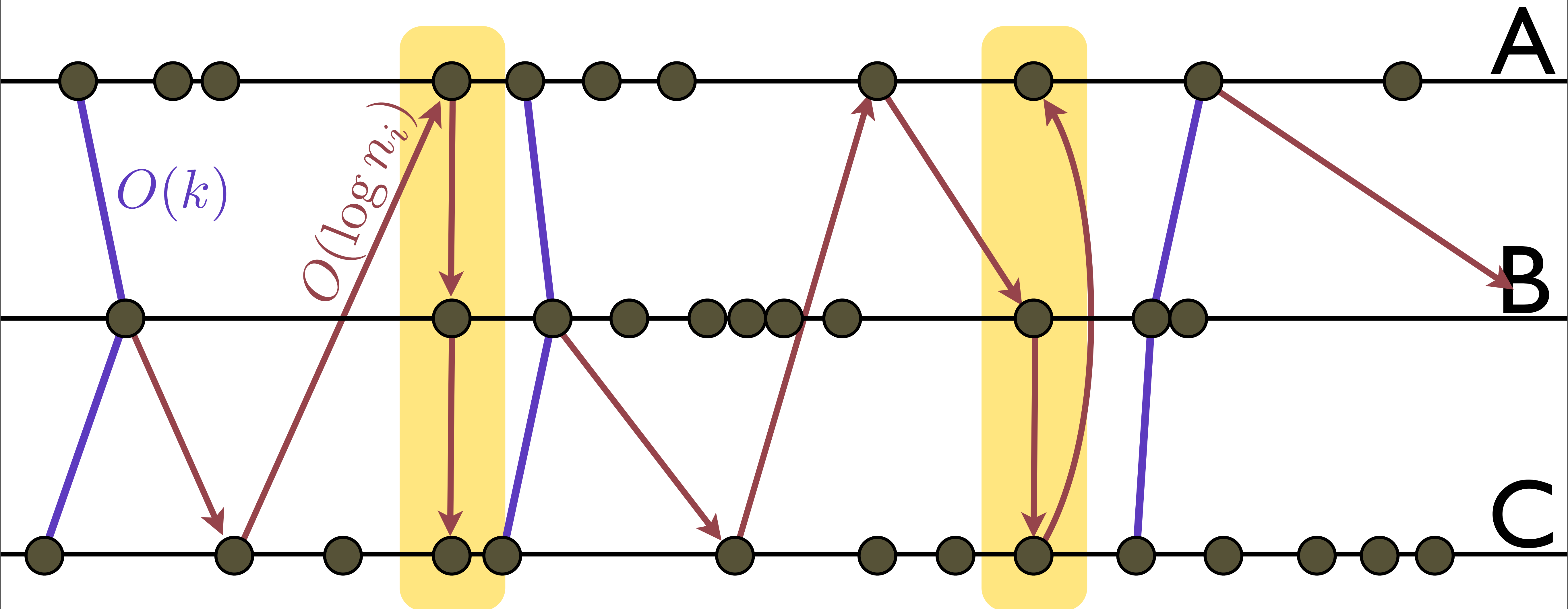
Lists are stored as std::set<...> (fast successor search).

d-1 pairwise intersections is slow CPU-wise and RAM-wise

We use a simple and faster algorithm from SODA 2000 by Demaine, López-Ortiz and Munro, that cycles through the d lists at once.

Optimizing lists intersection

Demaine, López-Ortiz and Munro [SODA 2000]



Optimizing with a simplex cache

We store a Least Recently Used cache of Delaunay simplices.
Simplices in the cache are also accessible via a dictionary on their vertex-set (we use a std::unordered_map<...>).

Cached simplices store pointers to neighbors.

After each insertion, the cache is shrunk to its maximum allowed size by removing the oldest simplices. The user has explicit control on the size of the cache.

get_neighbors() using the cache

Now, get_neighbors() manipulates only in-cache simplices.

1. If the stored pointer is not NULL, return it.

2.1. Otherwise, get candidate opposite vertices by list intersection.
2.2. Member-query each candidate simplex using dictionary.
Return simplex if found.
3. Otherwise, use in_circle() predicate as before.
4. Update the cache and the neighbor pointers as needed.

Points common to `New_DT` and `Del_graph`

- Spatial sorting (Biased Randomized Insertion Order with “Hilbert” space-filling curve).
  - BRIO should be important for the cache efficiency.
- Ambient dimension d is a compile time parameter (modified CGAL kernel with dimension as C++ template-parameter)

On uniformly distributed points...

1	Dimension	2	3	4	5	6
2	Number of input points	1024K	1024K	1024K	256K	32K
3	Size of the simplex-cache	1K	1K	10K	300K	1000K
4	Size of the conflict zone	4.1	21	134	940	6145
5	Calls to `neighbor(,)`	12.2	84.6	671.2	5631	43021
6	Number of candidates	2	2.6	4	6.7	11.6
7	Fast cache hit (non-*null* pointer)	56.6 %	57.5 %	54.6 %	55.5 %	54.3 %
8	Cache hit	37 %	39.6 %	40.1 %	42.3 %	43.1 %
9	Cache miss	6.4 %	2.9 %	5.3 %	2.2 %	2.6 %
10	Time ratio (`Del_graph`/`New_DT`)	6.1	5.7	6.0	6.5	8.1
11	Space ratio (`Del_graph`/`New_DT`)	2.7	1.7	0.6	0.2	0.1
12	Number of simplices per vertex	6	27 $(\times 4.5)$	157 $(\times 5.8)$	1043 $(\times 6.7)$	7111 $(\times 6.8)$
13	Number of edges per vertex	6	15.5 $(\times 2.6)$	36.5 $(\times 2.4)$	73 $(\times 2)$	164.6 $(\times 2.25)$