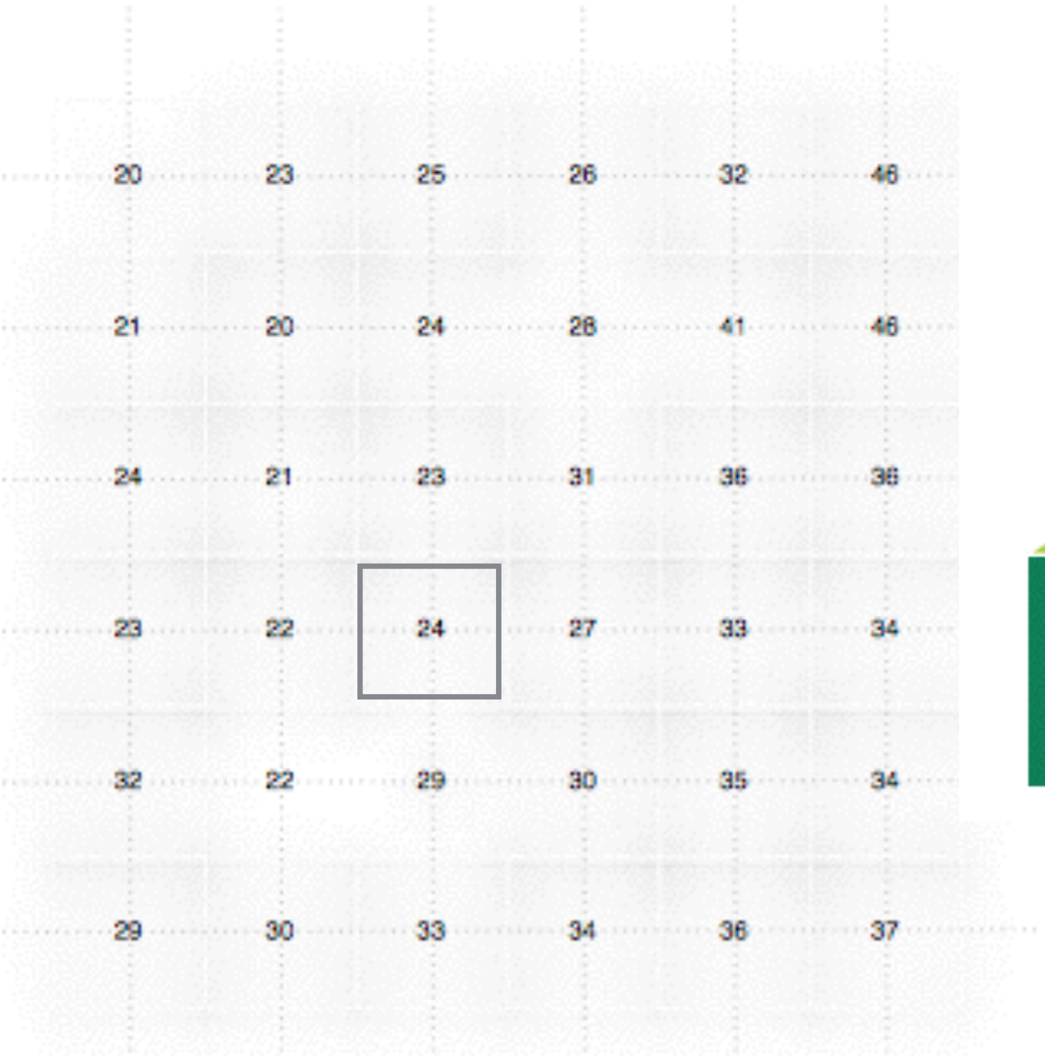
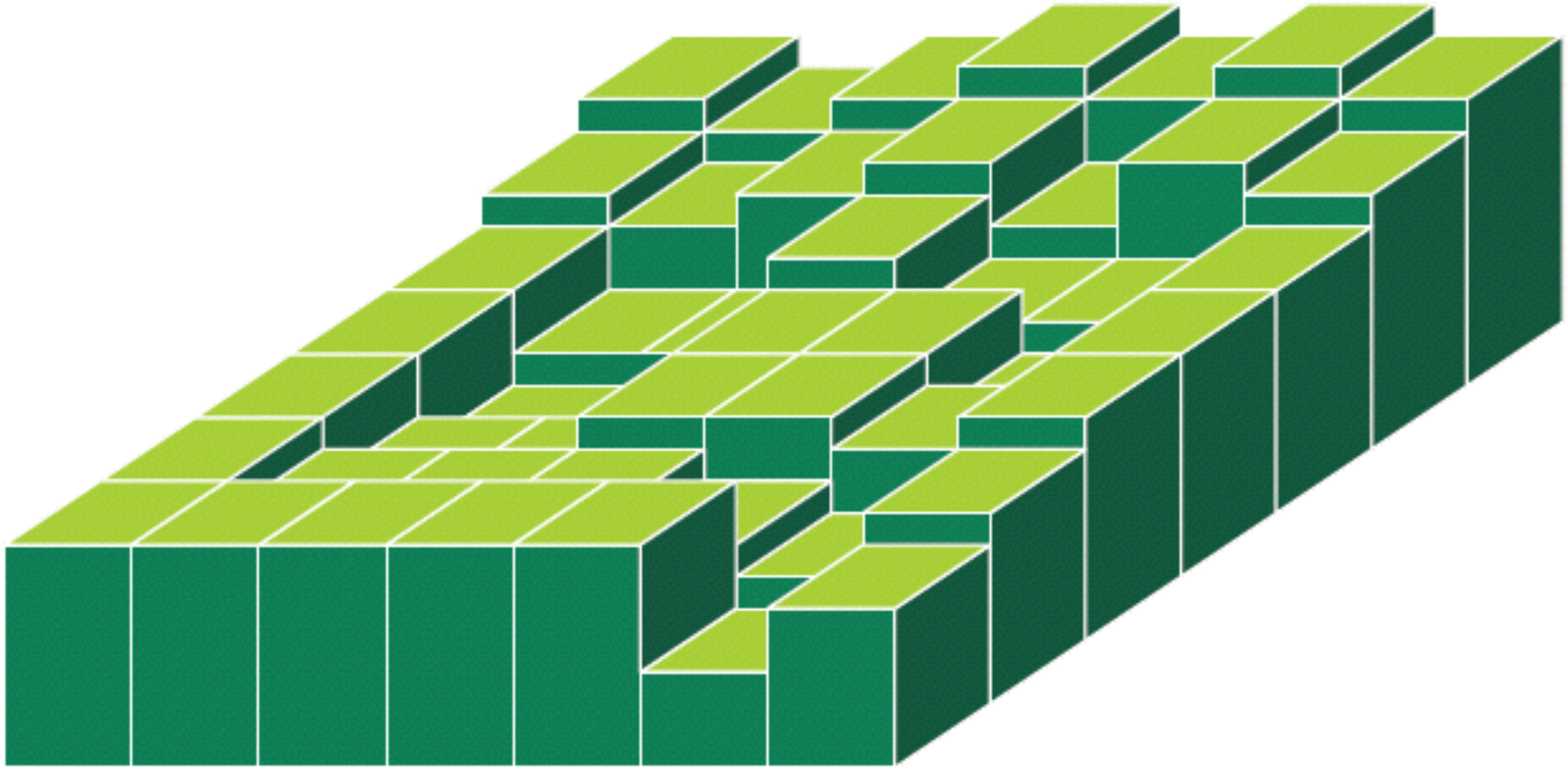
Algorithms for GIS:

Terrain simplification

Grid terrain models



20	23	25	26	32	46
21	20	24	28	41	46
24	21	23	31	36	36
23	22	24	27	33	34
32	22	29	30	35	34
29	30	33	34	36	37



thanks!!! to H. Haverkort

Motivation

- grid to TIN
  - uniform resolution means a lot of data redundancy
  - grids get very large very fast
- point cloud to grid
- point cloud to TIN

- Example:
  - Area if approx. 800 km x 800 km
  - Sampled at:
    - 100 resolution: 64 million points **(128MB)**
    - 30m resolution: 640 **(1.2GB)**
    - 10m resolution: 6400 = 6.4 billion **(12GB)**
    - 1m resolution: 600.4 billion **(1.2TB)**

Surface data: grid vs TIN

Raster

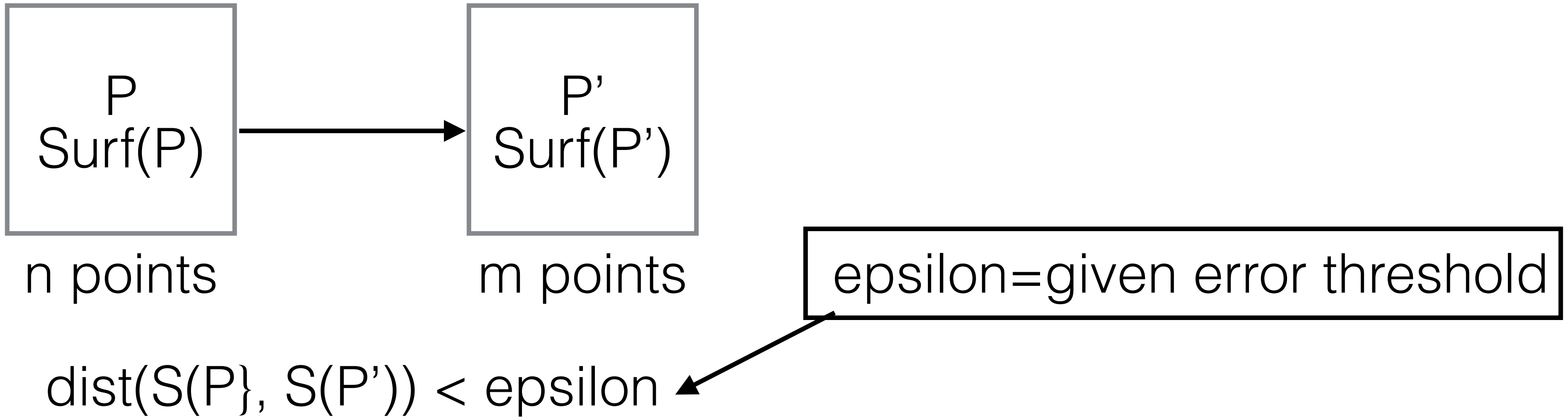
- Pros:
  - implicit topology
  - implicit geometry
  - simple algorithms
  - readily available in this form
- Cons:
  - uniform resolution ==> space waste

TIN

- Pros:
  - variable resolution
  - potentially space efficient
- Cons:
  - need to built and store topology
  - stored topology takes space
  - more complex programming (pointers..);

Terrain simplification

- P = { (x_1, y_1, z_1), (x_2, y_2, z_2), ……, (x_n, y_n, z_n) } a set of terrain elevation samples
  - For e.g. P could be a set of grid (aerial image) or in general a point cloud (from LIDAR)
  - sometimes called a “height field” (in graphics and vision)
- P + interpolation method ===> surface Surf(P) corresponding to P

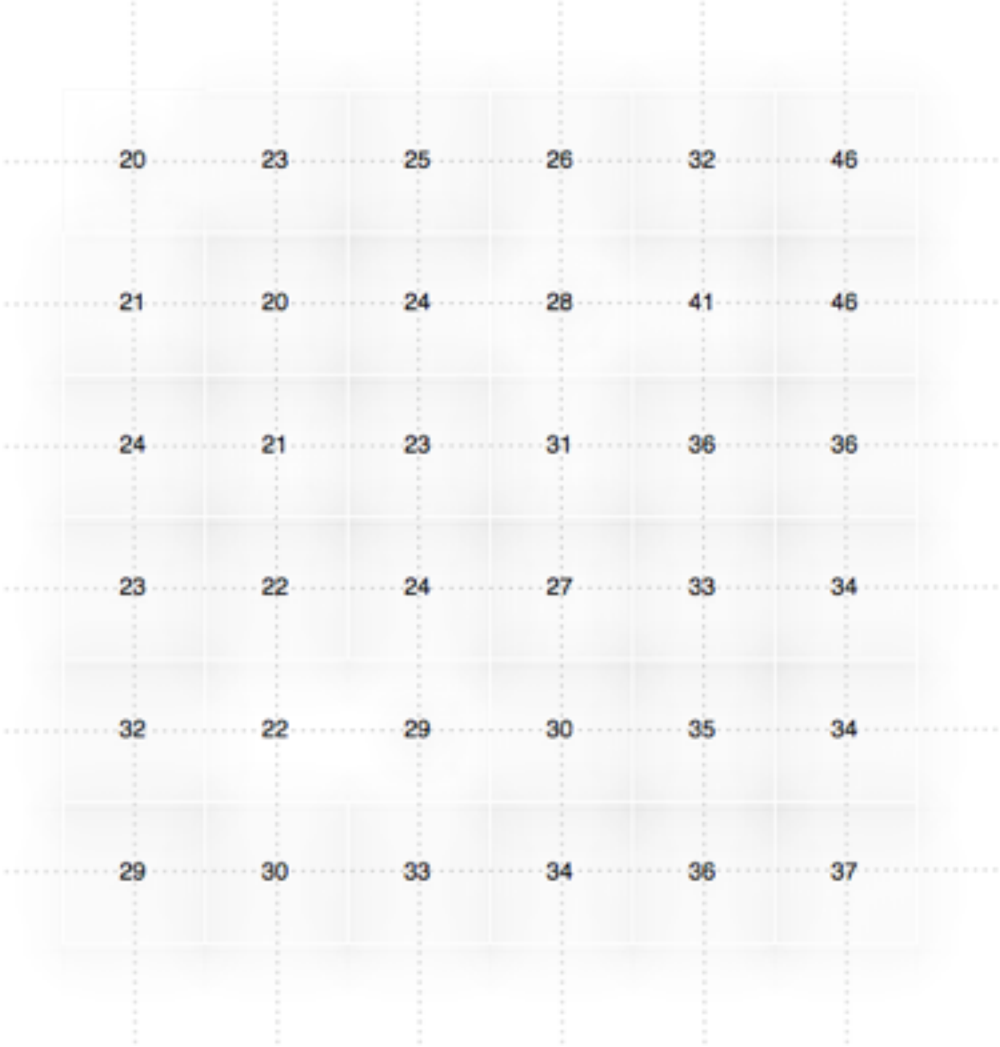


Simplification:

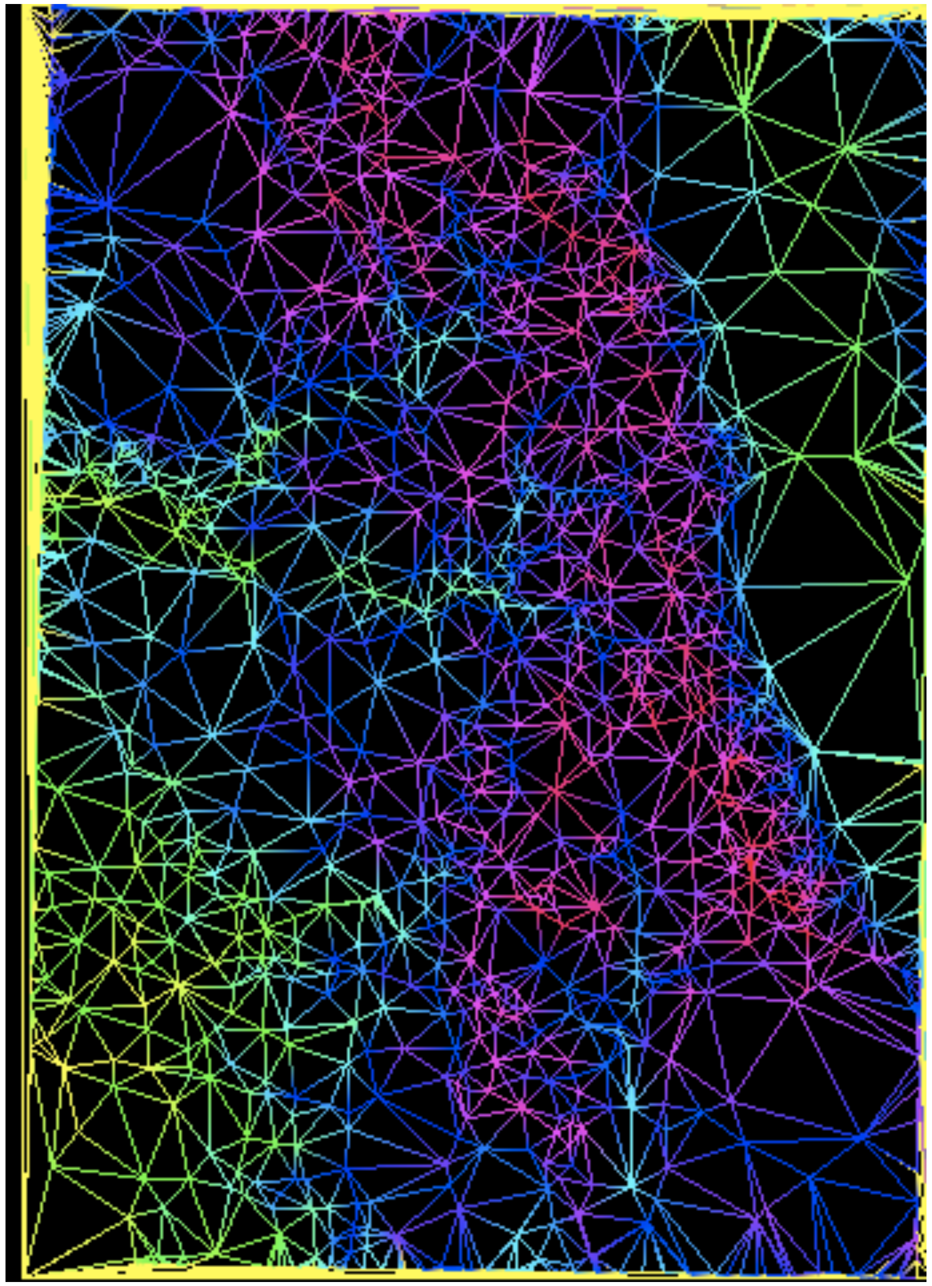
find an approximation S(P’) which approximates S(P) within the desired error threshold using as few points as possible

S(P) has n points ==> S(P’) has m points (m << n)

Grid to TIN



20	23	25	26	32	46
21	20	24	28	41	46
24	21	23	31	36	36
23	22	24	27	33	34
32	22	29	30	35	34
29	30	33	34	36	37



Grid-to-TIN simplification

- We'll focus on grid-to-TIN simplification
    - The methods can be extended to deal with arbitrary (non-grid) data
- Methods
    - Multi-pass decimation methods
        - start with P and discard points (one by one)
        - E.g.: Lee's drop heuristic
    - Multi-pass refinement methods
        - start with an initial approximation and add points one by one
        - greedy insertion (e.g. Garland & Heckbert)
    - One-pass methods
        - pre-compute importance of points
        - select points that are considered important features and triangulate them
        - based on quad trees or kd-trees

Decimation: Lee’s drop heuristic

Refinement: Greedy insertion

- Notation:
  - P = set of grid points
  - P’ = set of points in the TIN
  - TIN(P’): the TIN on P’

Algorithm:

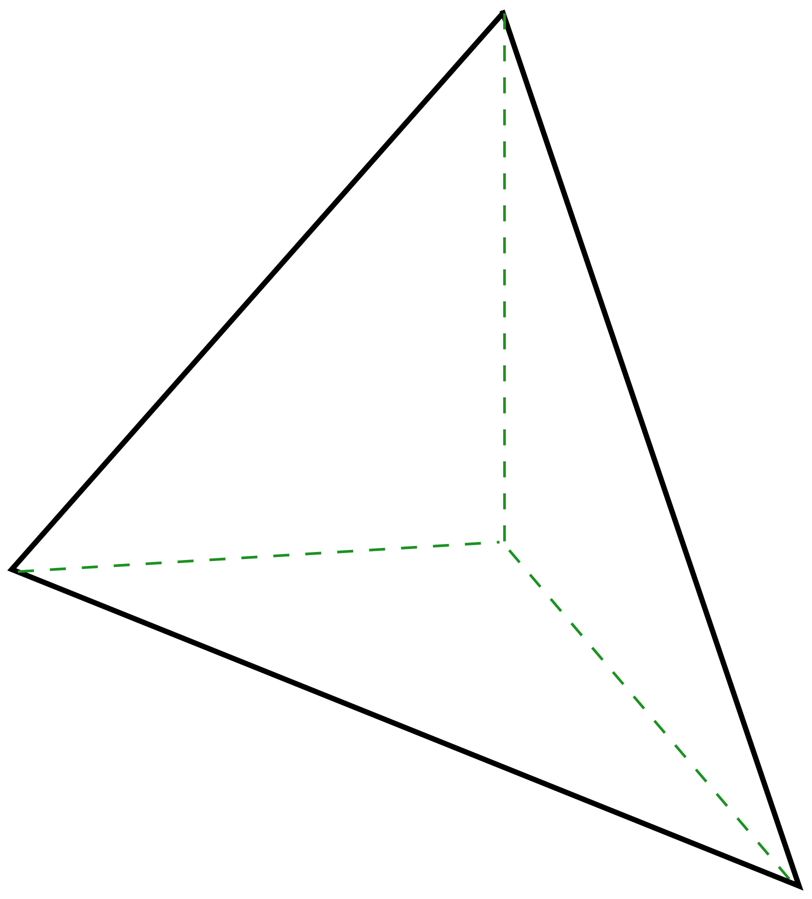
- P = {all grid points}, P’ = {4 corner points}
- Initialize TIN to two triangles with corners as vertices
- while not DONE() do
  - for each point p in P, compute error(p)
  - select point p with largest error(p)
  - insert p in P’ , delete p from P, and update TIN(P’)

DONE() :: return (max error below given epsilon) ? TRUE; FALSE;

Greedy insertion

- Come up with a straightforward implementation of the generic greedy insertion and analyze its running time.

- Assume straightforward triangulation (not Delaunay)
    - when inserting a point in a triangle, split the triangle in 3



Greedy insertion

	| P |	| P' |
	n	4 => O(1)
iteration 1	n-1	1 + O(1)
iteration 2	n-2	2 + O(1)
	.	.
	.	.
iteration k	n-k	k

at the end	n-m	m

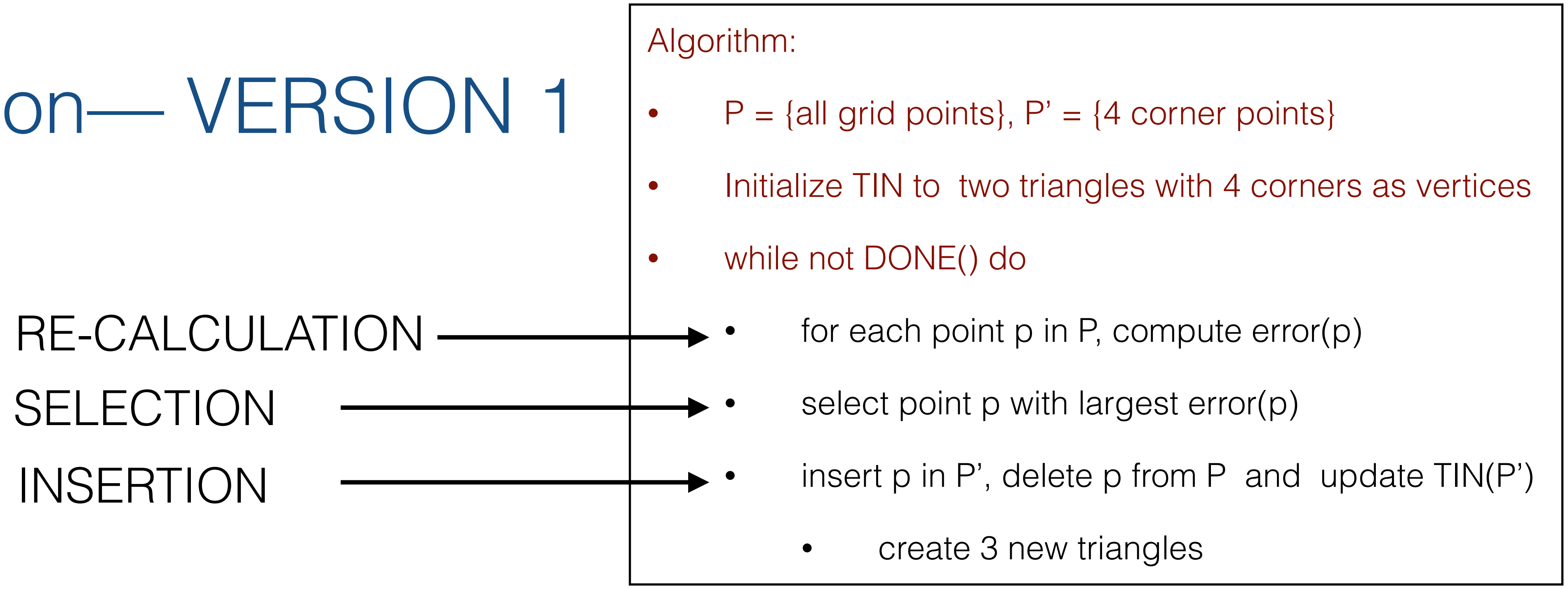
- Note:
  - m = nb of vertices in the simplified TIN at the end (when error of P' falls below epsilon)
  - usually m is a fraction of n (e.g. 5%)

Greedy insertion— VERSION 1

```
Algorithm:
•   P = {all grid points}, P' = {4 corner points}
•   Initialize TIN to  two triangles with 4 corners as vertices
•   while not DONE() do
        •   for each point p in P, compute error(p)
        •   select point p with largest error(p)
        •   insert p in P', delete p from P  and  update TIN(P')
                •   create 3 new triangles
```

find triangle that contains p and compute the vertical difference in height between p and its interpolation on the triangle

Greedy insertion— VERSION 1



RE-CALCULATION

SELECTION

INSERTION

Algorithm:

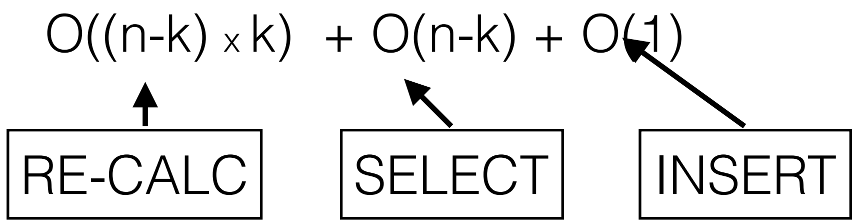
- P = {all grid points}, P’ = {4 corner points}
- Initialize TIN to two triangles with 4 corners as vertices
- while not DONE() do
    - for each point p in P, compute error(p)
    - select point p with largest error(p)
    - insert p in P’, delete p from P and update TIN(P’)
        - create 3 new triangles

ANALYSIS: At iteration k: we have O(n-k) points in P, O(k) points in P’

- RE-CALCULATION
    - compute the error of a point: must search through all triangles to see which one contains it ==> worst case O(k)
    - compute errors of all points ==> O(n-k) x O(k)
- SELECTION: select point with largest error: O(n-k)
- INSERTION: insert p in P’, update TIN ==> O(1)
    - unless each point stores the triangle that contains it, need to find the triangle that contains p
    - for a straightforward triangulation: split the triangle that contains p into 3 triangles ==> O(1) time

Greedy insertion— VERSION 1

Analysis worst case:



- iteration k: $O((n-k) \times k) + O(n-k) + O(1)$

RE-CALC

SELECT

INSERT

- overall: SUM $\{ (n-k) \times k \} = \ldots = O(m^2 n)$

- Note: dominant cost is re-calculation of errors (which includes point location)

- More on point location:
  - to locate the triangle that contains a given point, we “walk” (traverse) the TIN from triangle to triangle, starting from a triangle on the boundary (aka DFS on the triangle graph).
  - we must be very unlucky to always take O(k)
  - simple trick: start walking the TIN **from the triangle that contained the previous point.**
    - because points in the grid are spatially adjacent, most of the time a point will fall in the same triangle as the previous point or in one adjacent to it
  - average time for point location will be O(1)

Greedy insertion— VERSION 1

Worst-case: $O(m^2n)$

- iteration k: $O(n-k) \times O(k) + O(n-k) + O(1)$

RE-CALC SELECT INSERT

- overall: SUM $\{O(n-k) \times k\} = O(m^2n)$

Average case: $O(mn)$

- trick to seed up point location ==> average time for point location will be O(1)
- iteration k: $O(n-k) \times O(1) + O(n-k) + O(1)$

RE-CALC SELECT INSERT

- SUM $\{O(n-k)\} = O(mn)$

Greedy insertion— VERSION 2

Observation: Only the points that fall inside triangles that have changed need to re-compute their error.

- Re-compute errors ONLY for points whose errors have changed
- Each point p in P stores its error, error(p)
- Each triangle stores a list of points inside it

Algorithm:

- P = {all grid points}, P' = {4 corner points}
- Initialize TIN to two triangles with 4 corners as vertices
- while not DONE() do
    - ~~for each point p in P, compute error(p)~~
    - select point p with largest error(p)
    - insert p in P', delete p from P and update TIN(P')
        - create 3 new triangles
        - for all points in triangle that contains p:
            - find the new triangles where they belong, re-compute their errors

Greedy insertion— VERSION 2

Worst-case: O(mn)

- iteration k: $\underbrace{-}_{\text{RE-CALC}} + \underbrace{O(n-k)}_{\text{SELECT}} + \underbrace{O(1) + O(n-k) \times O(1)}_{\text{INSERT + re-calc}}$
- overall: SUM {O(n-k) } = O(mn)

Average case: O(mn)

- if points are uniformly distributed in the triangles ==> O((n-k)/k) points per triangle
- iteration k: $\underbrace{-}_{\text{RE-CALC}} + \underbrace{O(n-k)}_{\text{SELECT}} + \underbrace{O(1) + O((n-k)/k) \times O(1)}_{\text{INSERT + re-calc}}$
- SUM {O(n-k) + O((n-k)/k} = O(mn)

SELECTION will be dominant!

Greedy insertion— VERSION3

- Version2, re-calculation goes down and selection becomes dominant

↓

- Version 3: improve selection
    - store a heap of errors of all points in P

Algorithm:

- P = {all grid points}, P' = {4 corner points}
- Initialize TIN to two triangles with 4 corners as vertices
- while not DONE() do
    - use heap to select point p with largest error(p)
    - insert p in P', delete p from P and update TIN(P')
        - for all points in the triangle that contains p:
            - find the new triangles where they belong, re-compute their errors
            - update new errors in heap

Greedy insertion— VERSION 3

Worst-case: O(mn lg n)

- iteration k: $\underset{\uparrow \atop \boxed{\text{RE-CALC}}}{-} \;+\; \underset{\uparrow \atop \boxed{\text{SELECT}}}{O(\lg (n-k))} \;+\; \underset{\uparrow \atop \boxed{\text{INSERT + re-calc}}}{O(1) \;+\; O(n-k) \times O(\lg (n-k))}$

- overall: SUM $\{(n-k) \lg (n-k)\} = O(mn \lg n)$

Average case: $O((m+n) \lg^2 n)$

- if points are uniformly distributed in the triangles ==> $O((n-k)/k)$ points per triangle

- iteration k: $\underset{\uparrow \atop \boxed{\text{RE-CALC}}}{-} \;+\; \underset{\uparrow \atop \boxed{\text{SELECT}}}{O(\lg (n-k))} \;+\; \underset{\uparrow \atop \boxed{\text{INSERT + re-calc}}}{O(1) + O((n-k)/k) \times O(\lg (n-k))}$

- SUM $\{\lg (n-k)\ + O((n-k)/k\} = O((m+n) \lg^2 n)$

heap updates will be dominant!

Greedy insertion— VERSION 4

- Version 3: selection is down, but updating the heap is now dominant

↓

- Version 4: store in heap only one point per triangle (point of largest error)

Algorithm:

- P = {all grid points}, P' = {4 corner points}
- Initialize TIN to two triangles with 4 corners as vertices
- while not DONE() do
    - use heap to select point p with largest error(p)
    - insert p in P', delete p from P and update TIN(P')
        - for all points in the triangle that contains p:
            - find the new triangles where they belong, re-compute their errors
            - find point with largest error per triangle
            - add these points (one per triangle) to the heap

Greedy insertion— VERSION 4

Worst-case: O(mn)

- iteration k: $\underbrace{-}_{\text{RE-CALC}} + \underbrace{O(\lg k)}_{\text{SELECT}} + \underbrace{O(1) + O(n-k) \text{ x } O(1) + O(1) \text{ x } O(\lg k)}_{\text{INSERT + re-calc}}$
- overall: SUM $\{\lg k + O(n-k)\} = O(mn)$

Average case: O((m+n) lg n)

- if points are uniformly distributed in the triangles ==> $O((n-k)/k)$ points per triangle
- iteration k: $\underbrace{-}_{\text{RE-CALC}} + \underbrace{O(\lg k)}_{\text{SELECT}} + \underbrace{O(1) + O((n-k)/k) \text{ x } O(1) + O(1) \text{ x } O(\lg k)}_{\text{INSERT + re-calc}}$
- SUM $\{\lg k + O((n-k)/k\} = O((m+n) \lg n)$

Triangulation vs Delaunay triangulation