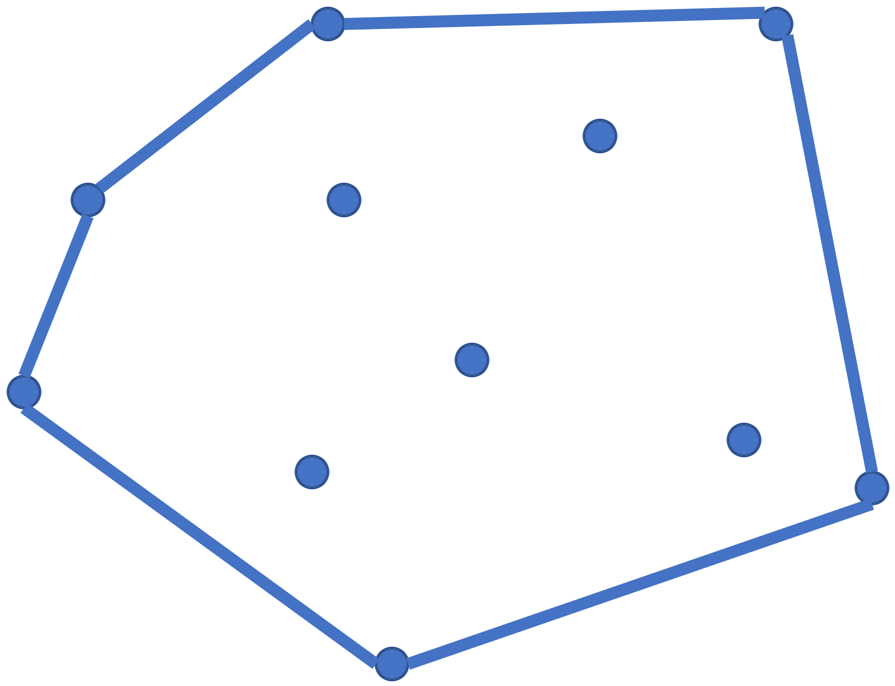
Randomized Incremental Convex Hull is Highly Parallel

Guy Blelloch, Yan Gu, Julian Shun and **Yihan Sun**

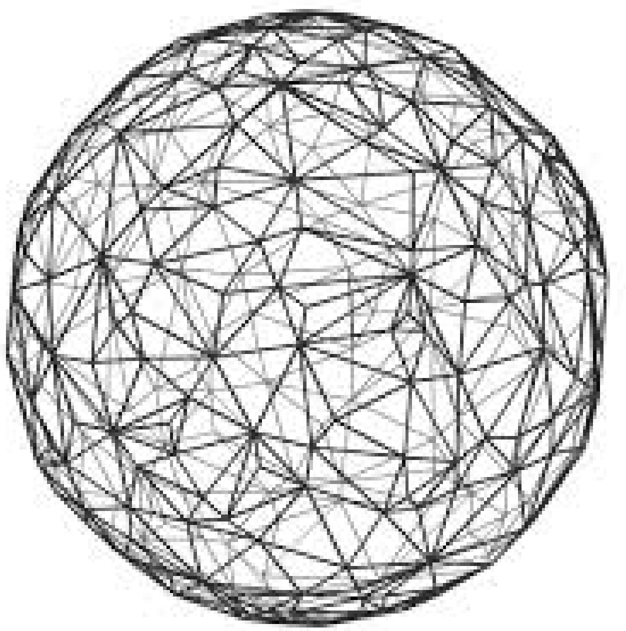
CMU	UC Riverside	MIT	UC Riverside

Convex Hull

- For a set of points X in d dimensions, find the smallest convex set that contains all points



A convex polygon in 2D



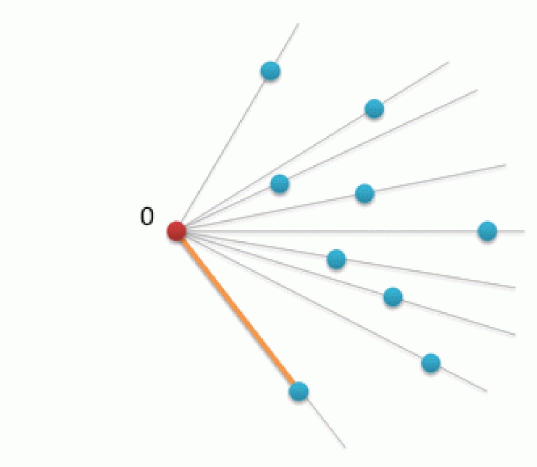
A convex polyhedron in 3D

Our contribution

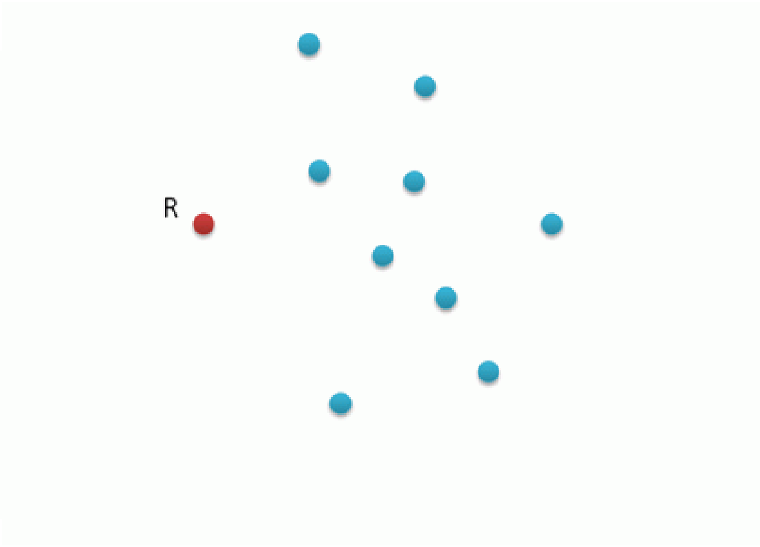
- Our contribution:
  - Proved the shallow depth of dependence in incremental convex hull construction
  - Proposed a new parallel incremental convex hull algorithm that is simple and efficient

Convex Hull Algorithms

- Scan-based algorithms
  - E.g., Jarvis March, Graham scan
  - Inherently sequential
- Divide-and-conquer
  - Parallelizable, but complicated
  - E.g., Amoto et al. achieved optimal work and $O(\log n)$ span
- Randomized incremental
  - Widely-used in practice

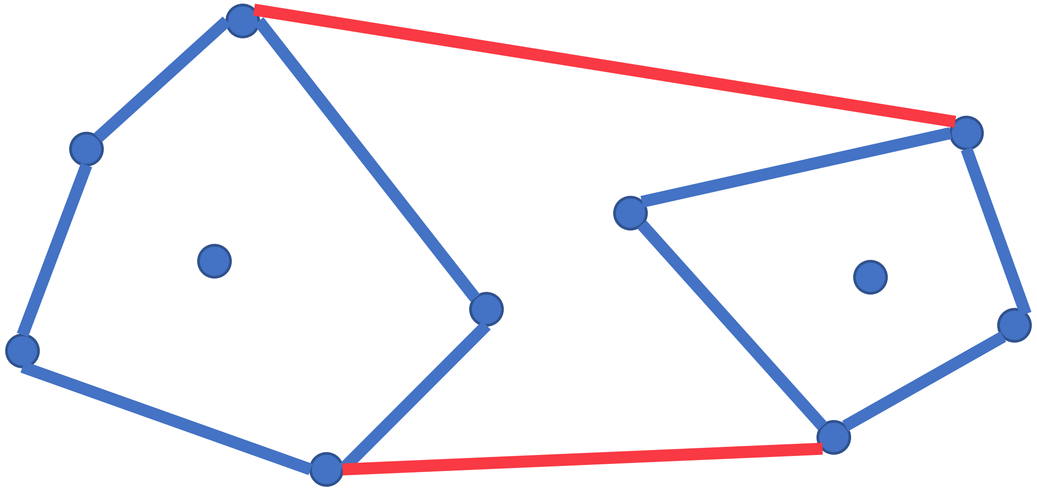


Graham scan



Jarvis March

Source: https://kukuruku.co/post/building-a-minimal-convex-hull/



Divide-and-conquer

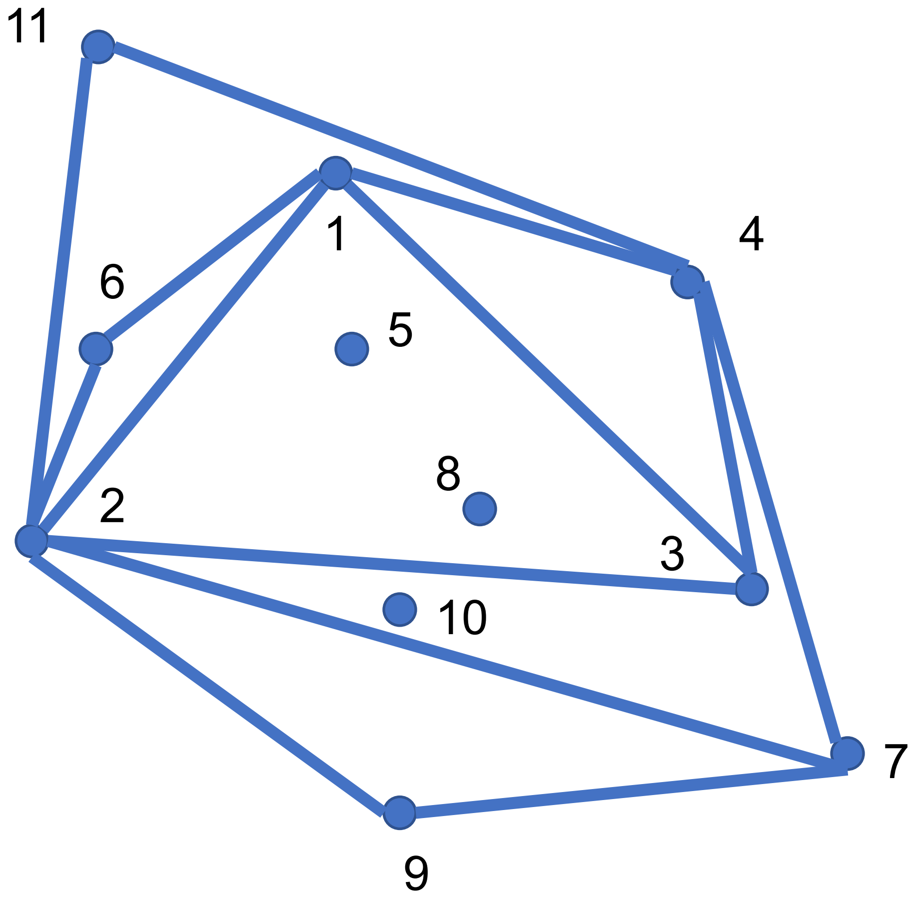
Convex Hull Algorithms

- Randomized incremental
  - Widely-used in practice
  - Easy to implement, easy extend to high dimensions
  - Sequentially, optimal work for d-dimensions in expectation [Clarkson and Shor]

  - Also in many parallel implementations [CLP'04, DLP'04, GCNTH'13, GLOP'06, GS'03, LOP'05]
  - Never have cost analysis in the parallel setting

Randomized incremental convex hull algorithm

- Adding points in a random order (random permutation or random priority)



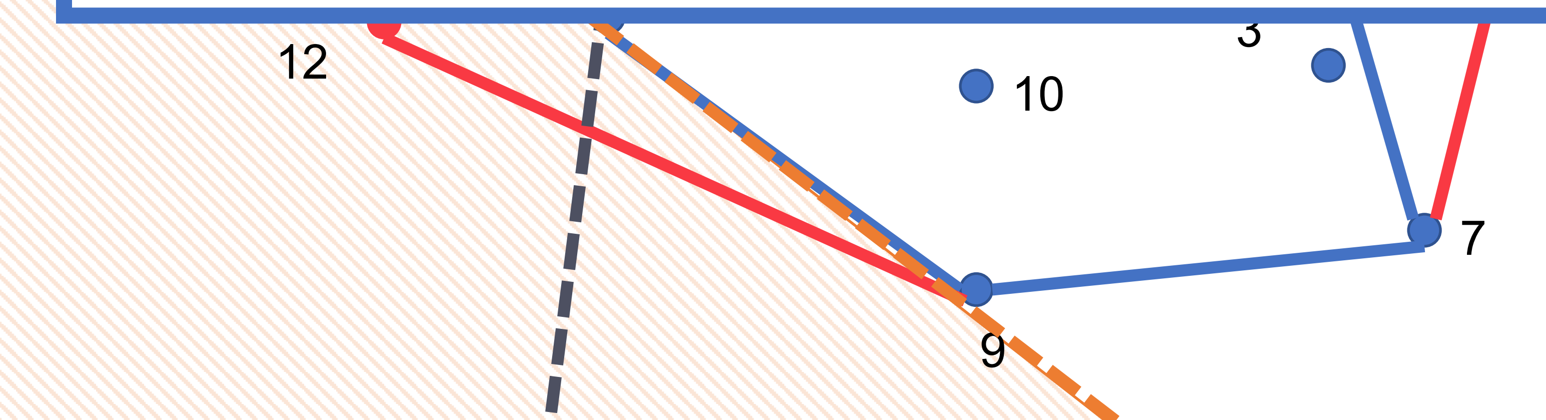

```
//P[i]: input points sorted by priority
//Maintains H: current convex hull

add_point(point p) {
  if (p in H) return;
  update H by adding p;
}

incremental_convex_hull(point* P) {
  for (i = 1 to n) {
     H = add_point(H, P[i]);
  }
}
```

Randomized incremental convex hull algorithm

- Parallelizable!
- Widely-used in parallel implementations

A parallel algorithm matching the sequential order

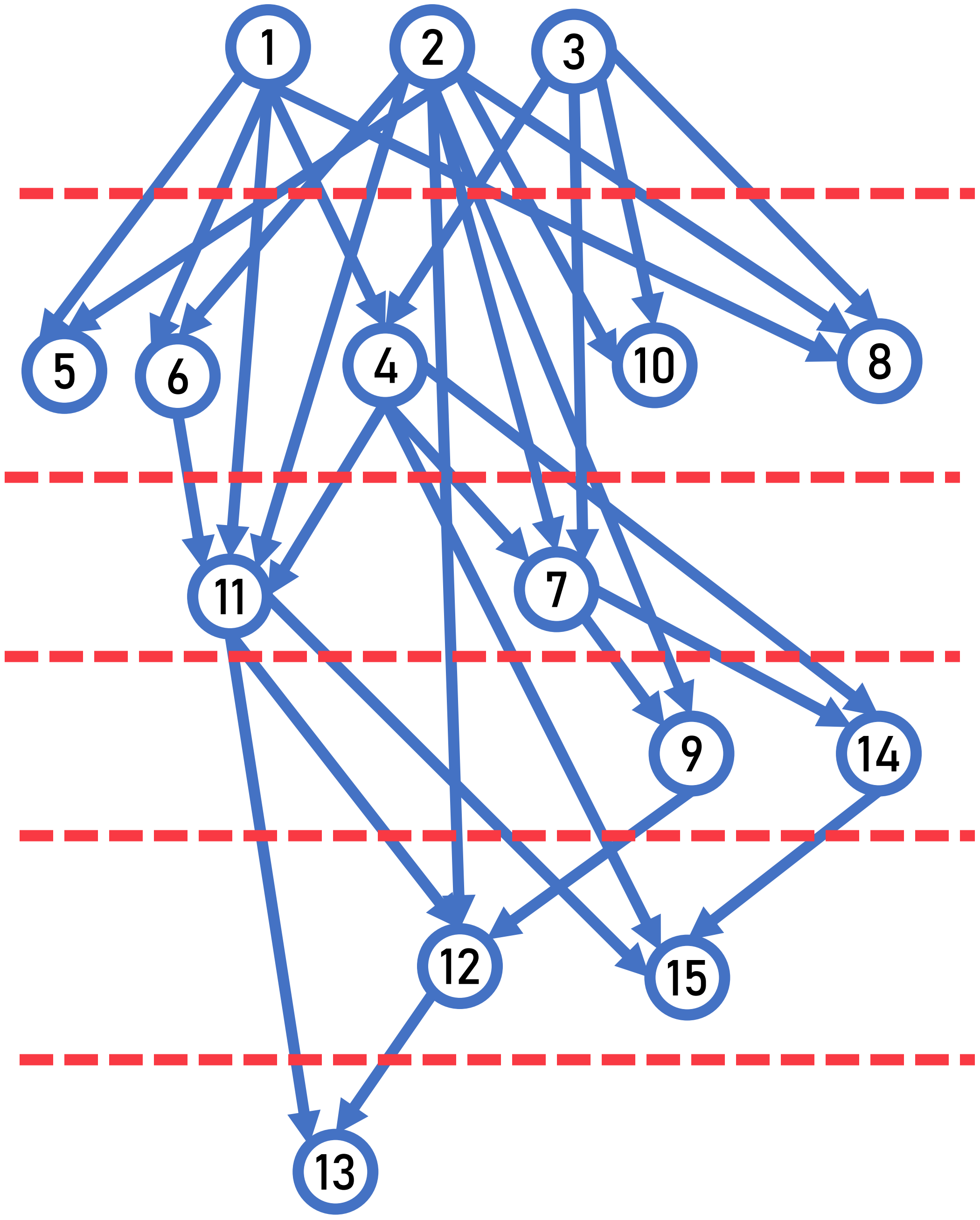
- **Determinism**: easy for programming
- **Easy reasoning/analysis**: directly use analysis in the sequential setting



- 12 comes first, so 15 has to wait

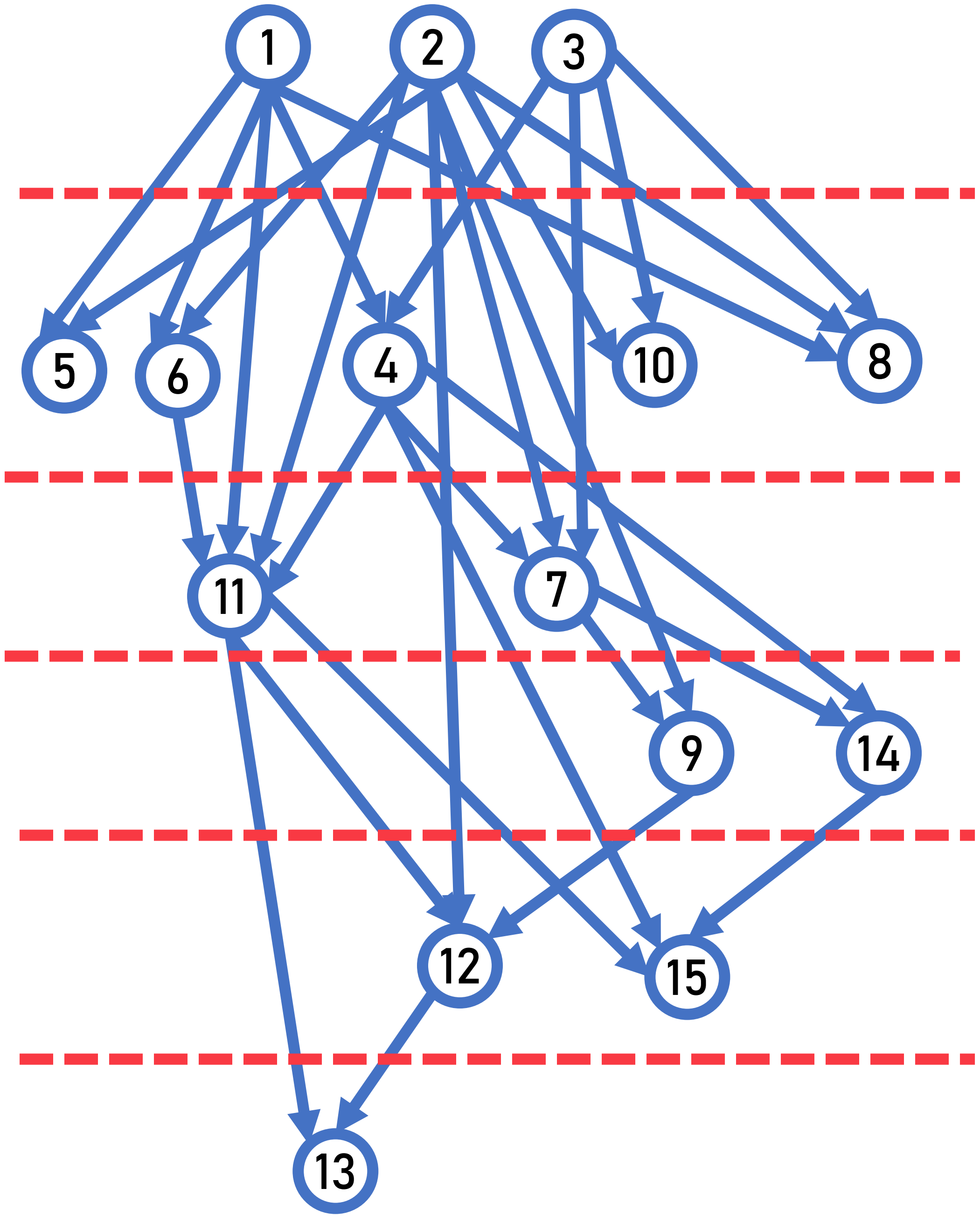
Dependence graph (DG)

- Randomized incremental algorithms can be parallelized
  - Existing algorithms: Knuth's shuffle [SGBFG'14], Delaunay triangulation [BGSS'16, BGSS'18], 2D LP [BGSS'16], …
- Iteration dependence graph (IDG) [SGBFG'14]
  - Nodes: iterations
  - Arcs: x to y means y can be processed only after x has finished
- For convex hull
  - Nodes: points
  - Arcs: $x \rightarrow y$ if y depend on x



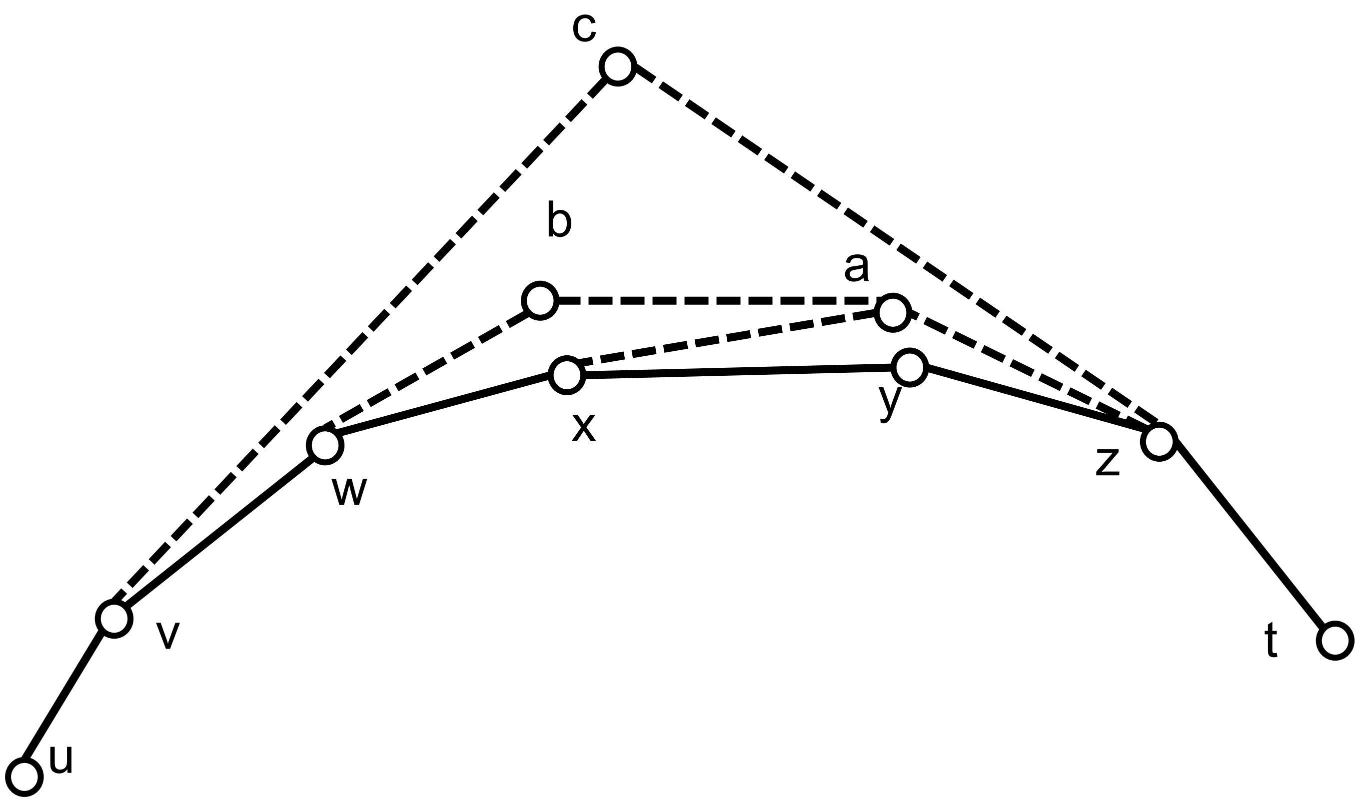
Dependence graph (DG)

- Randomized incremental algorithms can be parallelized
  - Existing algorithms: Knuth's shuffle [SGBFG'14], Delaunay triangulation [BGSS'16, BGSS'18], 2D LP [BGSS'16], …
  - Depth of DG is $O(\log n)$ w.h.p.
  - Parallel algorithm with polylog span (longest dependence chain in algorithm)
- Iteration dependence graph (IDG)
- For convex hull
  - No known result for the depth of DG



Dependence for Incremental Convex Hull

- Adding c depends on v, w, x, y, z, t
- A point can depend on an arbitrary number of other points
- Sequentially it's OK: work adds up:
  - Each point depends on a constant number of other points on average, the total work is bounded
- In parallel, if the fan-in of each node contribute to the possible #path in the DG multiplicatively
  - Cannot bound the span



Existing hull: u-v-w-x-y-z-t, adding a, b, c in order

Revisit - Dependence for Incremental Convex Hull

Adding c depends on ... ? u-v, v-w, and a-z, z-t

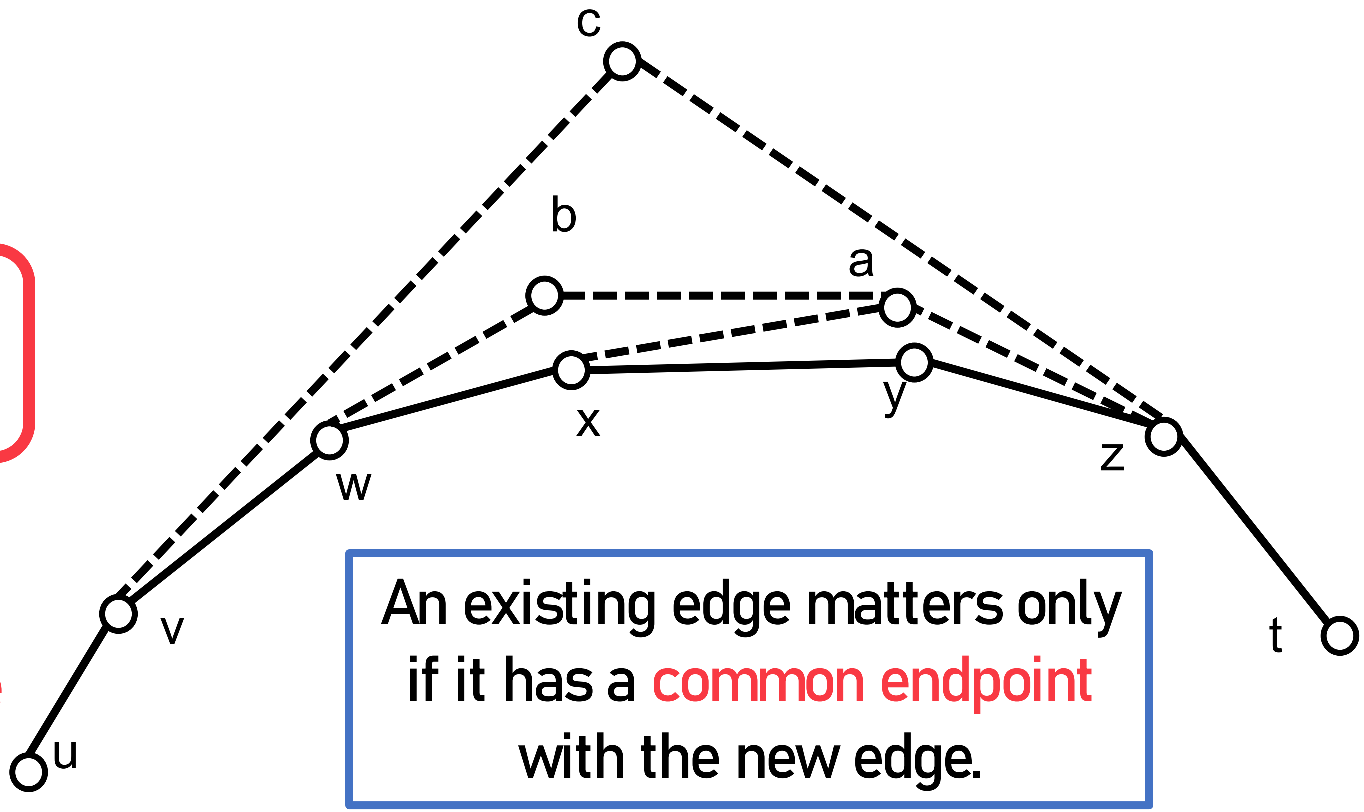
More precisely: c-v depends on u-v and v-w, and c-z depends on a-z and z-t

- c-v and c-z don't need to be added together: asynchrony
  - c-v can be added immediately, while c-z needs to wait (a-z isn't ready)

More relaxed/fine-grained order to add edges!

- Each edge depends on at most 2 other edges!
  - Identify true dependences from false dependences

Let's study edges instead of points!



An existing edge matters only if it has a common endpoint with the new edge.

Existing hull: u-v-w-x-y-z-t, adding a, b, c in order

Study the edges in convex hull –Configuration Space

Configuration space [CS'89, DK'89, Mulmuley'94]

- For incremental algorithms, study the configurations
  - E.g., triangles in Delaunay triangulations, edges in 2D convex hulls
- Objects: input elements (e.g., points for Delaunay triangulation or convex hull)
- Configurations
  - Defining set: objects that defines a configuration
  - Conflict set: objects that cannot co-exist with the configuration

Work-efficiency: total conflict set size can be bounded [CS'89]

Study the edges in convex hull –Configuration Space

Configuration space

- Objects: input elements
- Configurations
  - Defining set: objects that defines a configuration
  - Conflict set: objects that cannot co-exist with the configuration

2-D convex hulls

- Points
- Edges (Facets in high D)
  - 2 endpoints defining the edge

  (d-1 points defining the facet in high D)
  - Points visible from the edge

  (that will remove the edge/facet)

What about the analysis of span?

Work-efficiency: total conflict set size can be bounded [CS'89]

Study the edges in convex hull – Configuration DG

2-D convex hulls

Configuration space

- Prove work-efficiency

Dependence graph

- Analyze parallel depth

Configuration dependence graph [this paper]

Generally

- Build dependence for configurations, instead of iterations

For convex hull

- Build dependence for edges/facets, instead of points

Configuration DG

- A configuration x supports y if y depends on x

k-support

Configuration dependence graph

- Each configuration is supported by at most k other configurations

d-D convex hulls

- $k = 2$ is a constant
- $k = 2$ also for high dimensions

Theorem 1 (informally) (Shallow dependence for k support) A configuration dependence graph with k-support (k is a constant) has depth $O(\log n)$ w.h.p., where n is the number of objects.

Revisit - Dependence for Incremental Convex Hull

Adding c depends on … ?

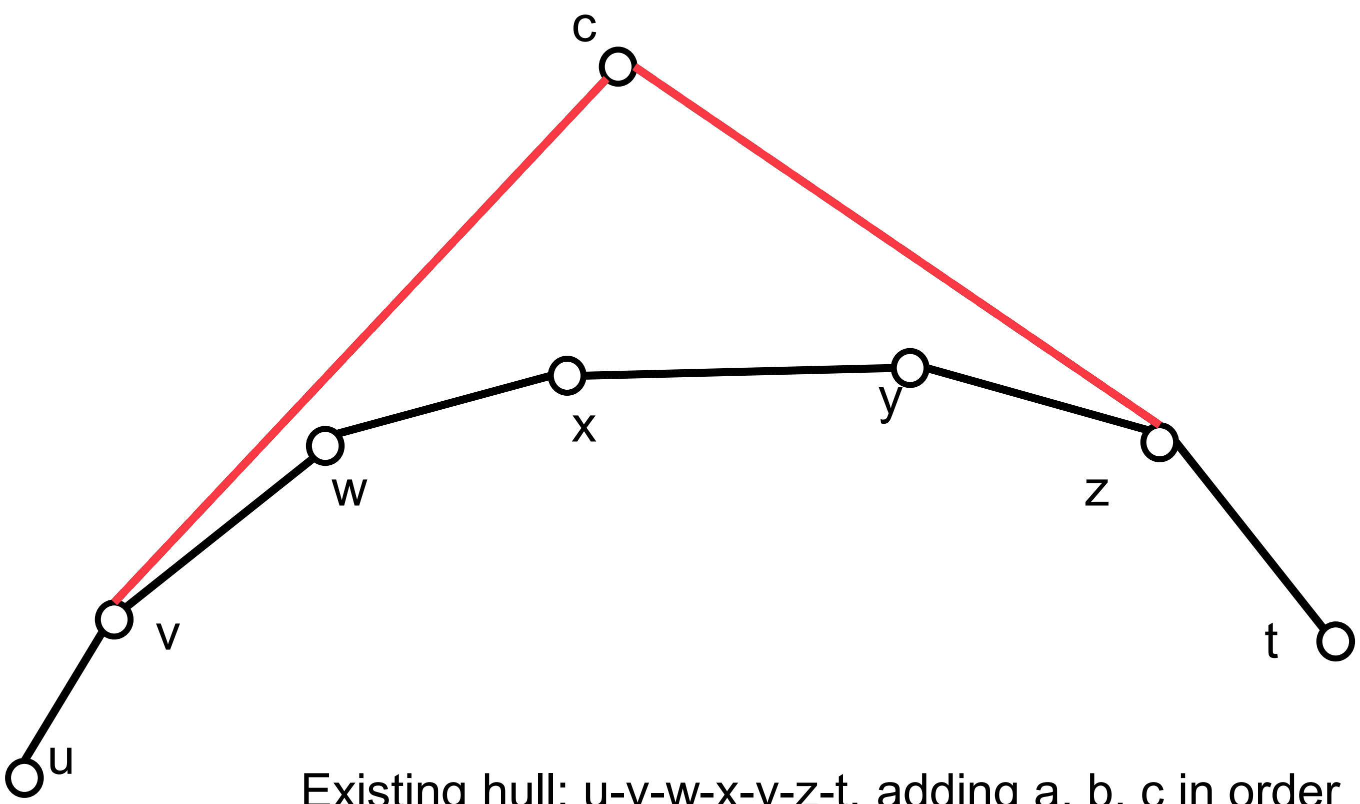
u-v-w, and y-z-t

More precisely:

u-v and v-w support c-v

y-z and z-t support c-z

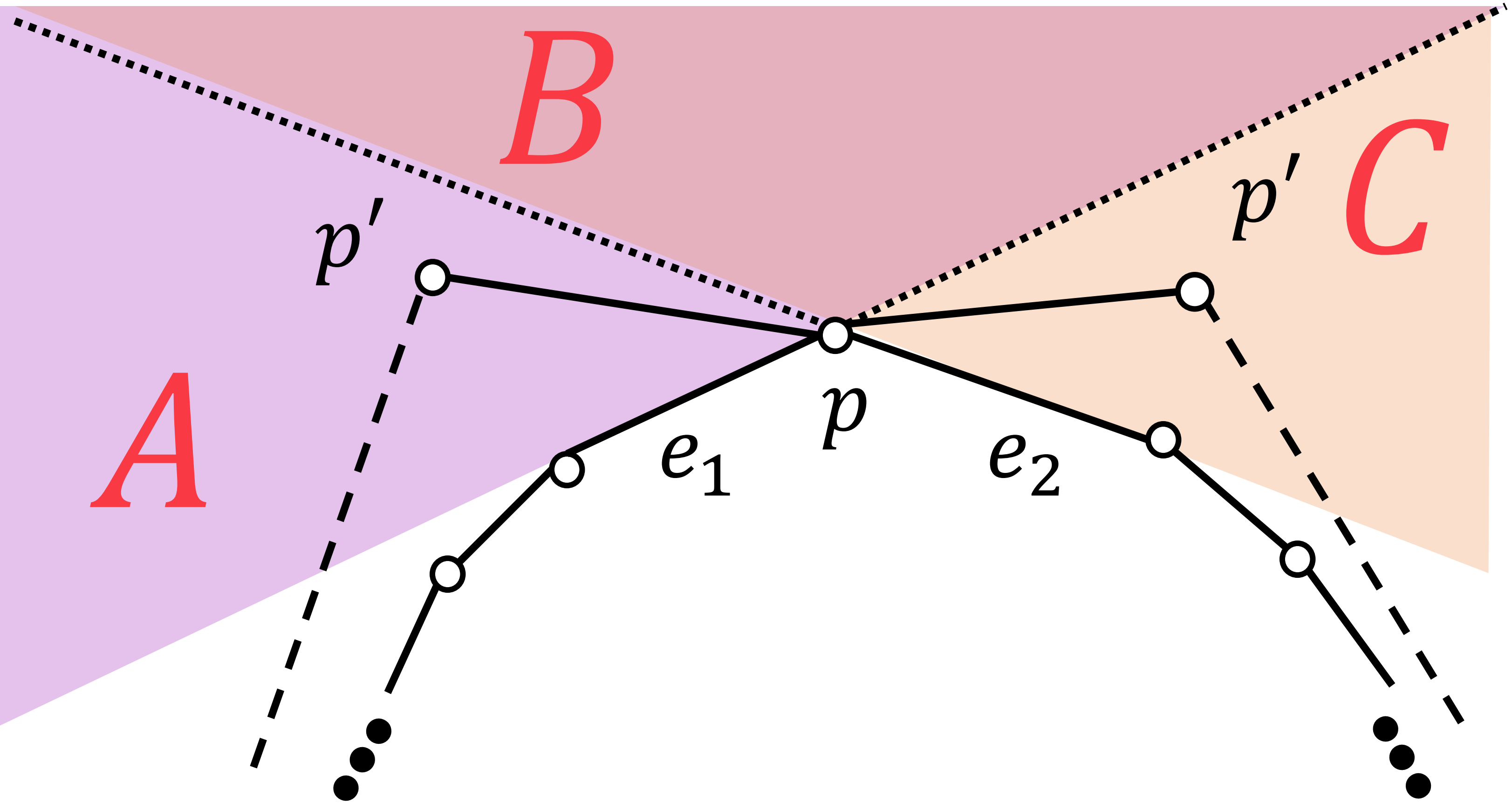
- c-v replaces v-w
- c-z replaces y-z
- c buries w-x and x-y (false dependence)



Existing hull: u-v-w-x-y-z-t, adding a, b, c in order

Replace or Bury?

- The point p' that will "change" the corner
  - Visible from e_1 or e_2
  - The earliest point in the random order

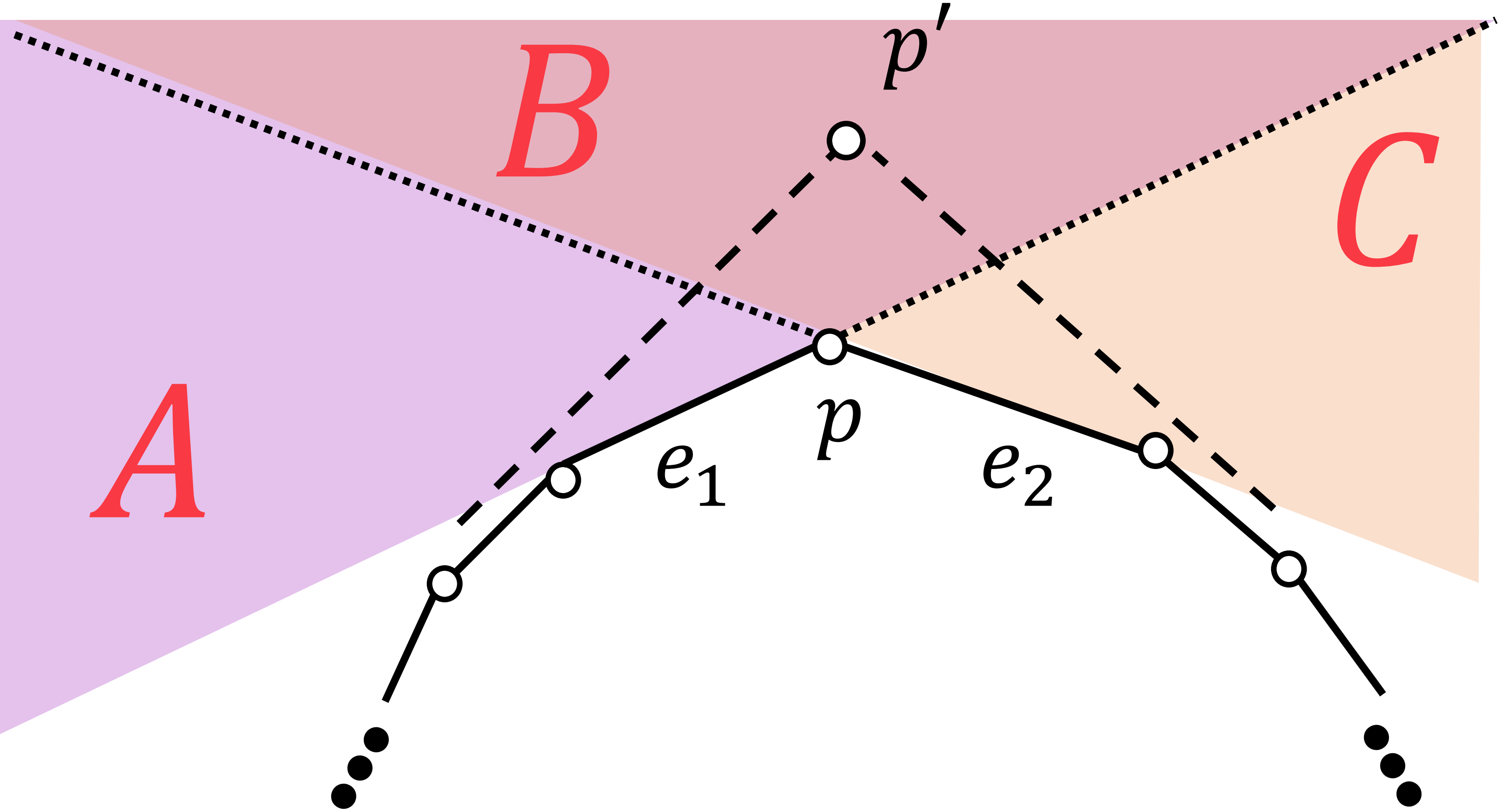


- $\boldsymbol{p'}$ in $\boldsymbol{A}$
  - visible from $\boldsymbol{e_1}$ but not $\boldsymbol{e_2}$
  - e_1-p-e_2 at the boundary of all edges p' will remove
  - p-p' *replace* e_1
  - The other edge incident on p' will be processed by another corner

- $\boldsymbol{p'}$ in $\boldsymbol{C}$ (symmetric)
  - visible from $\boldsymbol{e_2}$ but not $\boldsymbol{e_1}$
  - p-p' *replace* e_2

Replace or Bury?

- The point $\boldsymbol{p'}$ that will "change" the corner
  - Visible from e_1 or e_2
  - The earliest point in the random order



- $\boldsymbol{p'}$ in $\boldsymbol{B}$
  - visible from both $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$
  - e_1-p-e_2 in the middle of all edges p' will remove
  - p' *buries* e_1 and e_2

More notes

- k-support with $k = 2$ still holds for higher dimensions.
- The parallel algorithm works similarly:
  - Edge => Facets, Corner => Ridge
- For d dimensions with constant d:
  - Work-efficient: $O(n^{\lfloor \frac{d}{2} \rfloor} + n \log n)$ expected work
  - Polylog(n) span w.h.p. (the concrete form of bound depends on computational model)

- Algorithm doesn't need to run in rounds
  - Can run in divide-and-conquer manner in nested parallelism

Summary

Proved the low depth of dependence graph for parallel incremental convex hull

- $O(\log n)$ for n objects w.h.p.
- Key technique: asynchrony and k-support
  - Asynchrony: Build dependence on facets instead of points
  - k-support: any facet depends on at most $k = 2$ other facets – fewer dependences

Designed a parallel incremental algorithm for convex hull

- Work-efficient with polylog span, for d dimension with constant d
- Key technique: asynchrony and k-support
  - Asynchrony: facets added by a point can be added in different rounds
  - k-support: Allowing burying ridges/edges: avoid these "false" dependence

Summary

Proved the low depth of dependence graph for parallel incremental convex hull

- $O(\log n)$ for n objects w.h.p.

Designed a parallel incremental algorithm for convex hull

- Work-efficient with polylog span, for d dimension with constant d

Key technique: asynchrony and k-support

- Technique applicable to any k-support configuration space
  - When k is a constant, we can prove shallow depth: Degeneracy in 3D convex hull, half-space intersection, unit circle intersection, …
  - Problems remains open (not constant-support): trapezoid decomposition, MIS, …

Parallel incremental convex hull algorithm

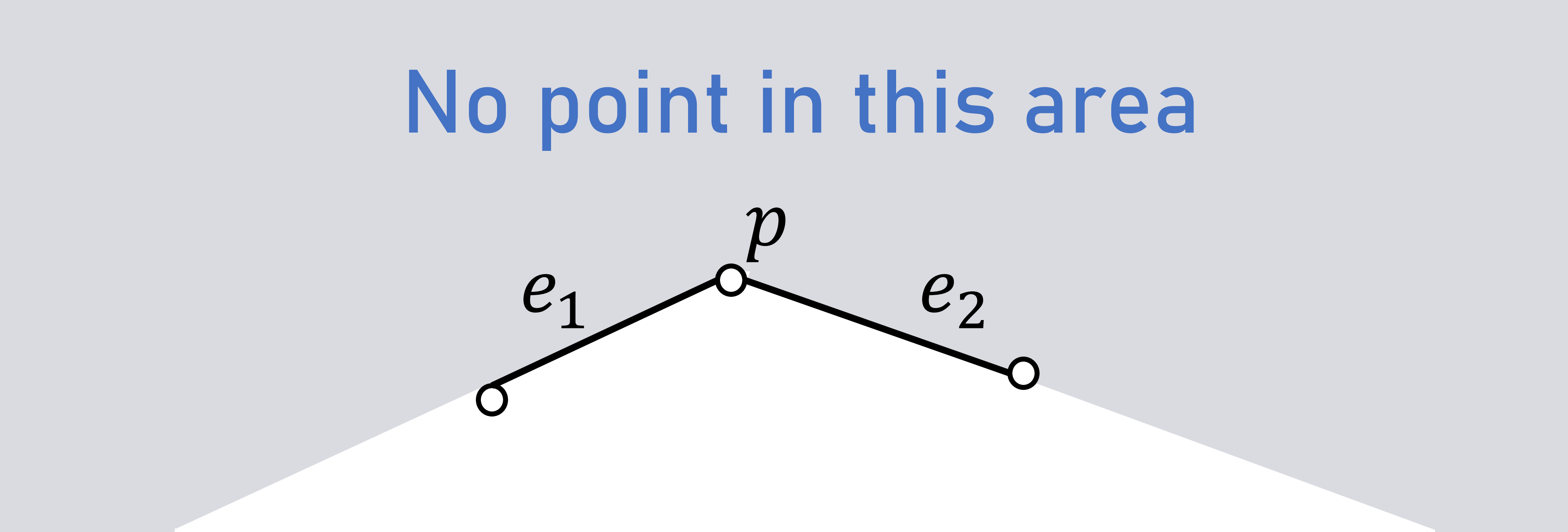
- In each round:
  - In parallel: look at each possible support set – a corner of two edges in 2D, or a ridge of adjacent facets for high dimensions
  - Process the corner/ridge to see if it will be removed (replaced/buried)
  - Newly formed corners/ridges will be for the next round

- For any edge e, the point that will remove it in this round is:
  - Among all points that are visible from e, the one with the highest priority (consistent with sequential order)
  - Call it the pivot point of e: pivot(e) – maintain the conflict set for each edge

Process corner (edge e_1, e_2 and point p)

Let p_1 = pivot(e_1), p_2 = pivot(e_2)

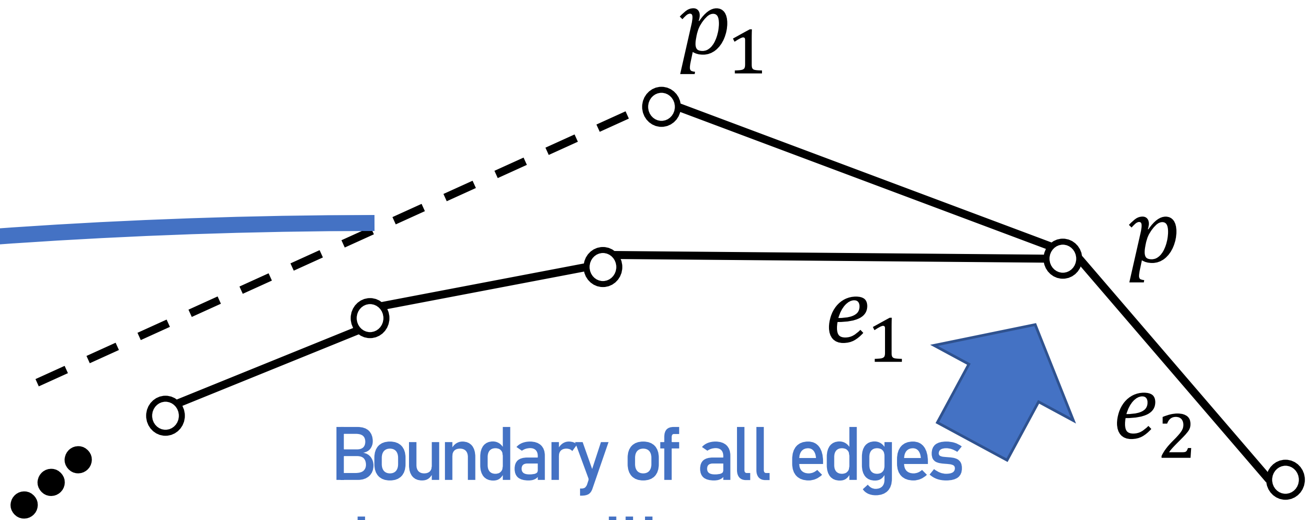
- Case 1: $p_1 = p_2 =$ null   no conflict points!
  - This corner is finalized and will never be removed
  - Return



Process corner (edge $\boldsymbol{e_1}$, $\boldsymbol{e_2}$ and point $\boldsymbol{p}$)

Let $\boldsymbol{p_1}$ = pivot($\boldsymbol{e_1}$), $\boldsymbol{p_2}$ = pivot($\boldsymbol{e_2}$)

- Case 2: $\boldsymbol{p_1}$ has higher priority than $\boldsymbol{p_2}$ (null has lowest priority)
  - p_1 is visible from e_1, but not e_2!
  - p_1-p depends on this corner e_1-p-e_2
  - Create edge p-p_1, update its conflict set
  - e_1 will be replaced by p-p_1



We don't know what happened (or will happen) to the other edge incident on p_1, but we don't care in this process

Boundary of all edges that p_1 will remove

Process corner (edge e_1, e_2 and point p)

Let p_1 = pivot(e_1), p_2 = pivot(e_2)

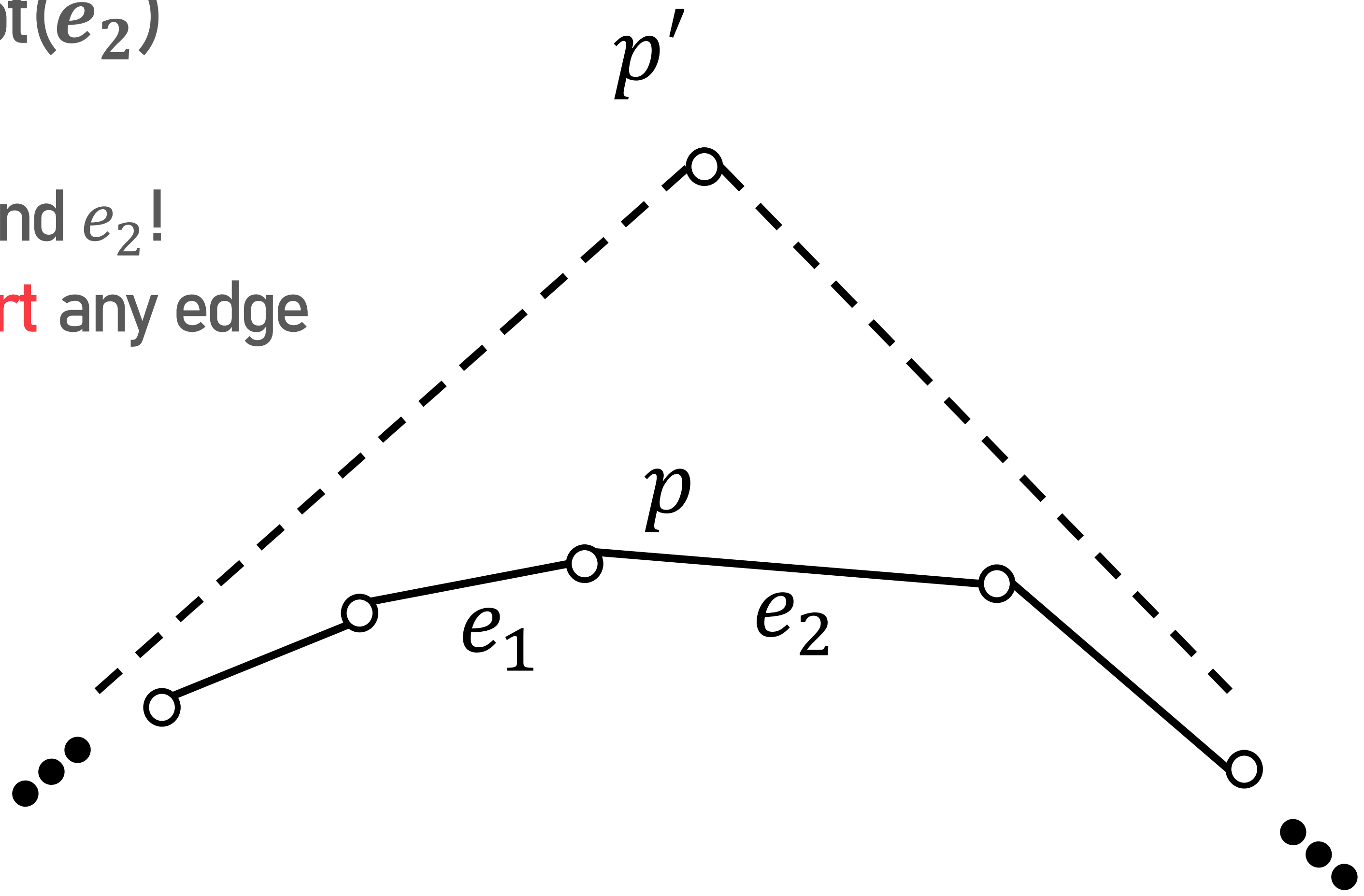
- Case 3: p_2 has higher priority than p_1 (null has lowest priority)

Symmetric

Process corner (edge e_1, e_2 and point p)

Let p_1 = pivot(e_1), p_2 = pivot(e_2)

- Case 4: $p_1 = p_2 = p'$
  - p' is visible from both e_1 and e_2!
  - The corner does not support any edge incident on p'
  - p' buries e_1-p-e_2
  - Delete e_1-p-e_2, return



p_1 will be processed by other corners

Parallel incremental convex hull algorithm

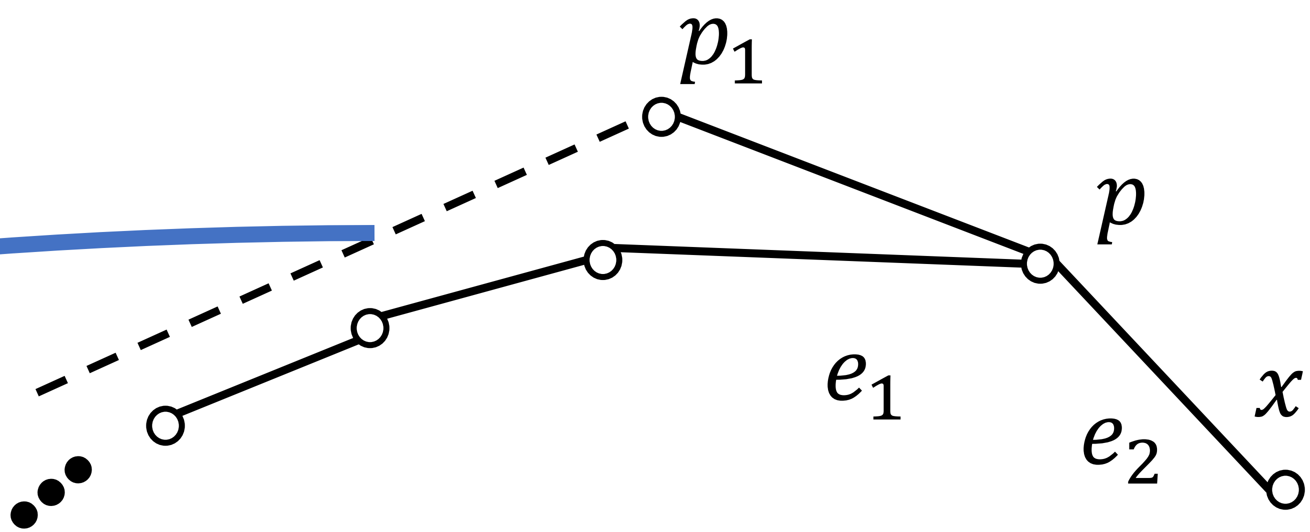
- In each round:
  - In parallel: look at each possible support set – a corner of two edges in 2D, or a ridge of adjacent facets for high dimensions
  - Process the corner/ridge to see if it will be removed (replaced/buried)
  - Newly formed corners/ridges will be for the next round

Process corner – form new corners

How is a new corner formed?

- The new corner $\boldsymbol{p_1}$-$\boldsymbol{p}$-$\boldsymbol{x}$ directly formed: process in the next round
- What about the corner at $\boldsymbol{p_1}$?

What happened (or will happen) to the other edge incident on p_1?

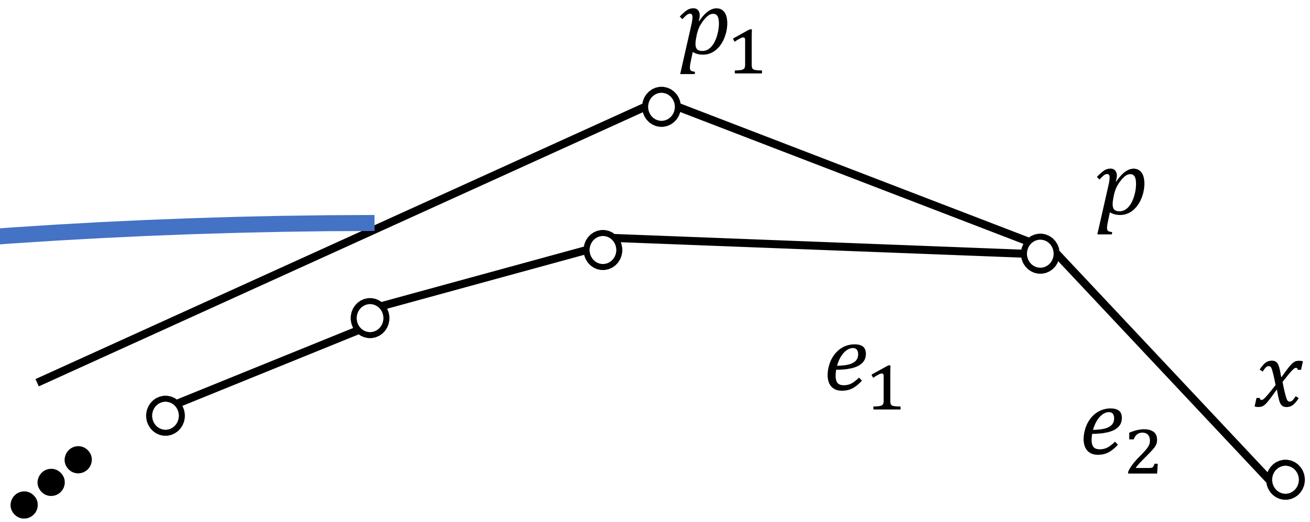


Process corner – form new corners

How is a new corner formed?

- The new corner $\boldsymbol{p_1}$-$\boldsymbol{p}$-$\boldsymbol{x}$ directly formed: process in the next round
- What about the corner at $\boldsymbol{p_1}$?
- Two edge form a corner: the later one comes makes the corner ready to be processed
- Use a parallel hash table
  - Let the two edges synchronize

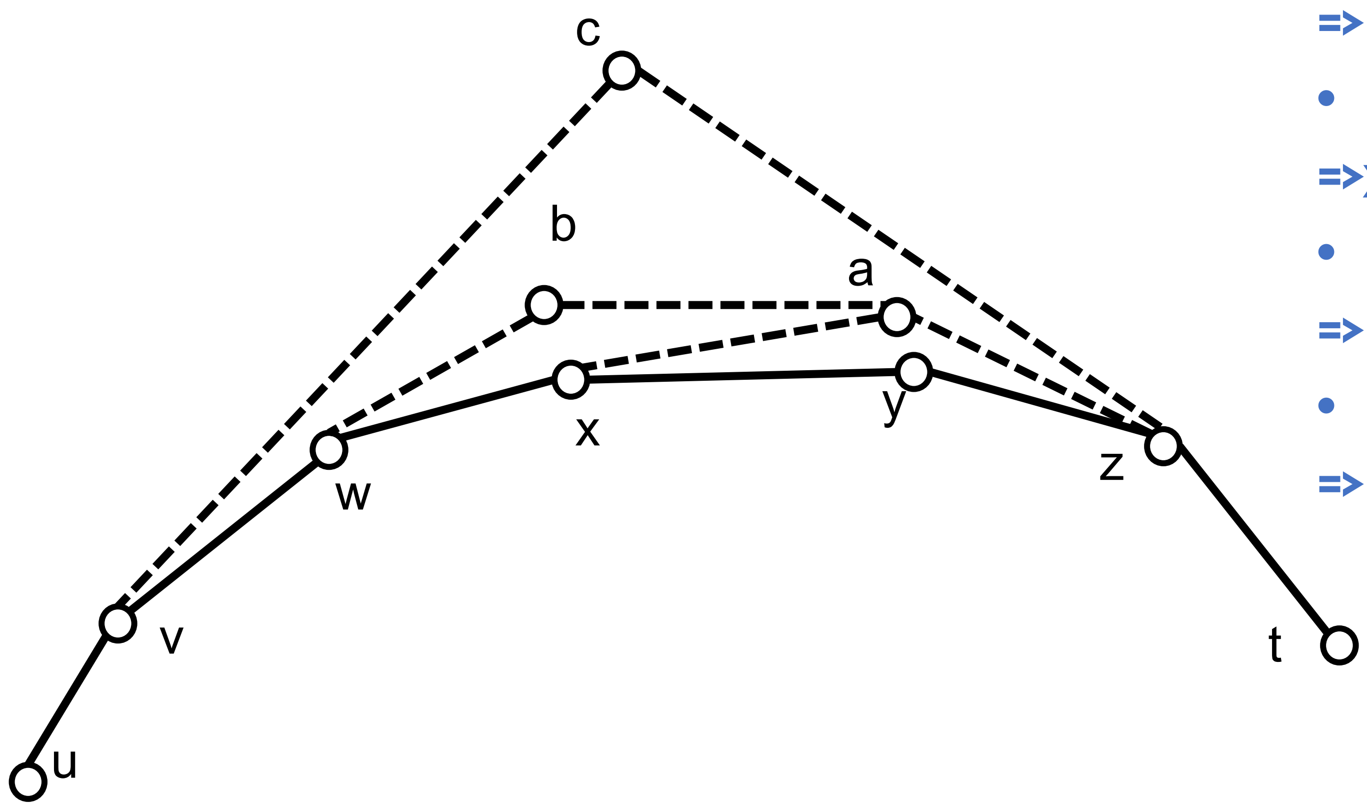
If the other edge p-p' already exists, add the corner p'-p_1-p to be processed in the next round



Running example

- The earliest visible point from an edge $\boldsymbol{e}$ or a corner $\boldsymbol{e_1}$-$\boldsymbol{p}$-$\boldsymbol{e_2}$: among all the points visible from the corresponding edges, the one that appears the earliest in the random order.
  - The first one that will be process in sequential order

Running example



- u-v-w: support c-v

=> c-v replaces v-w

- v-w-x: support w-b

=> w-b replaces wx

- w-x-y: support x-a

=>x-a replaces xy

- x-y-z: buried by a

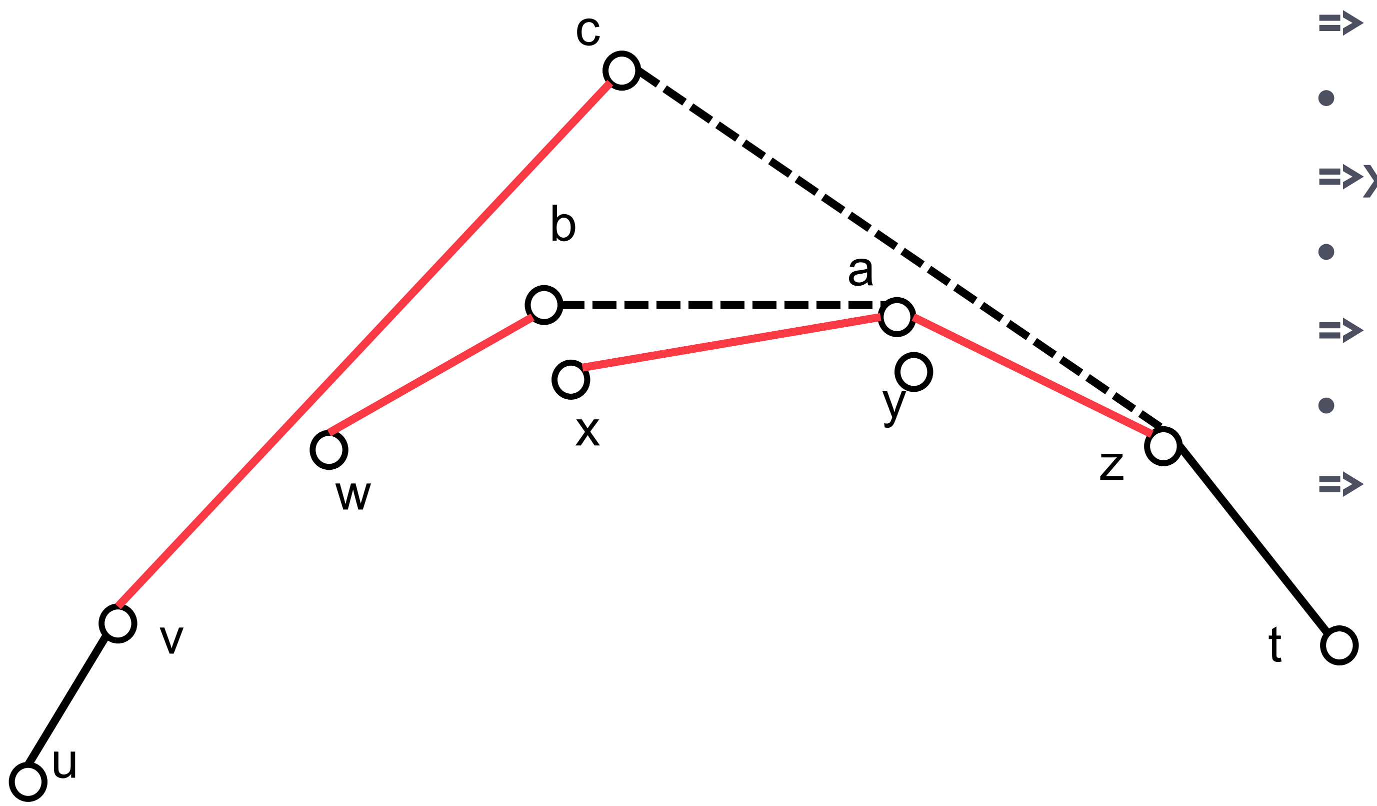
=> remove x-y and y-z

- y-z-t: support a-z

=> z-a replaces y-z

Existing hull: u-v-w-x-y-z-t, adding a, b, c in order

Running example



- u-v-w: support c-v
=> c-v replaces v-w
- v-w-x: support w-b
=> w-b replaces wx
- w-x-y: support x-a
=>x-a replaces xy
- x-y-z: buried by a
=> remove x-y and y-z
- y-z-t: support a-z
=> z-a replaces y-z

Existing hull: u-v-w-x-y-z-t, adding a, b, c in order

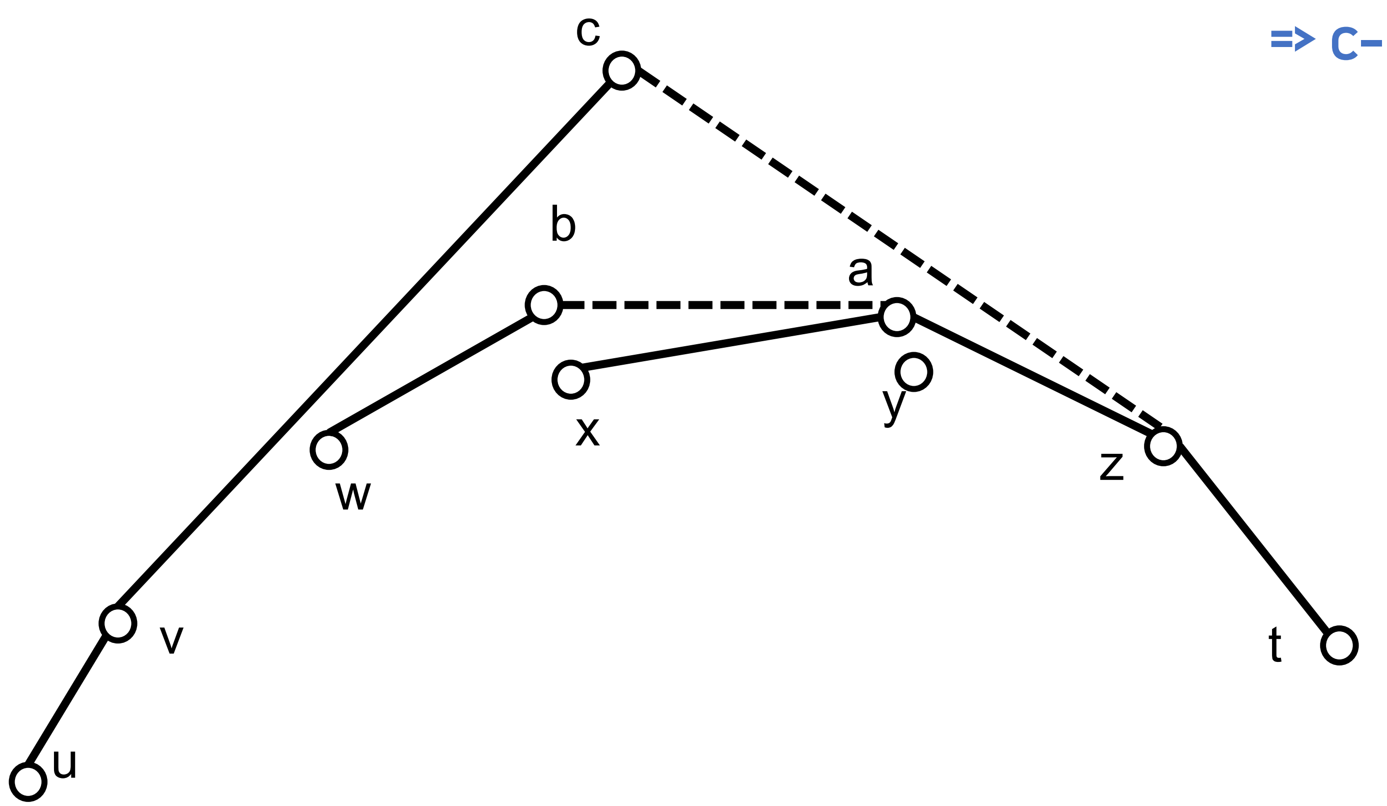
Running example

- x-a-z: support b-a

=> b-a replaces x-y

- a-z-t: support c-t

=> c-z replaces a-z



Existing hull: u-v-w-x-y-z-t, adding a, b, c in order

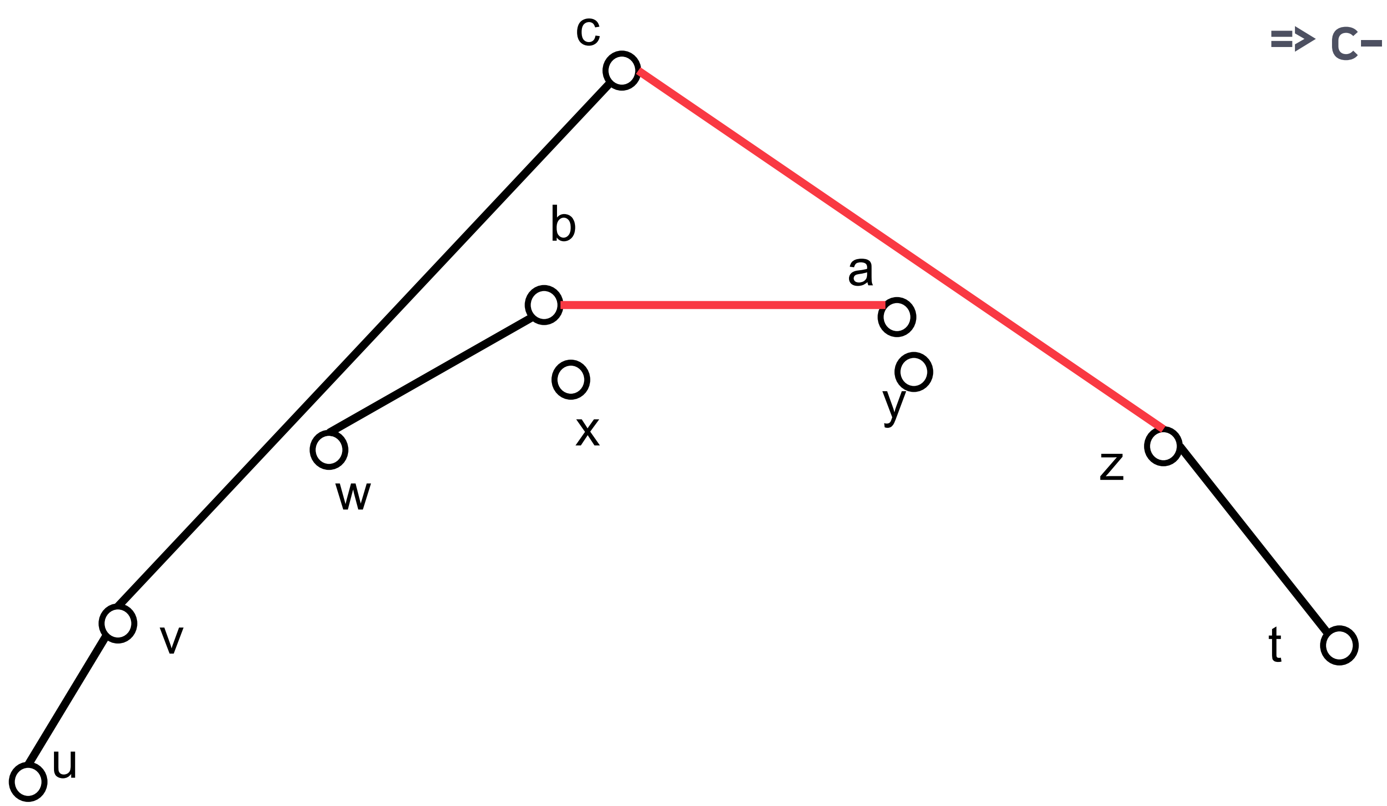
Running example

- x-a-z: support b-a

=> b-a replaces x-y

- a-z-t: support c-t

=> c-z replaces a-z



Existing hull: u-v-w-x-y-z-t, adding a, b, c in order

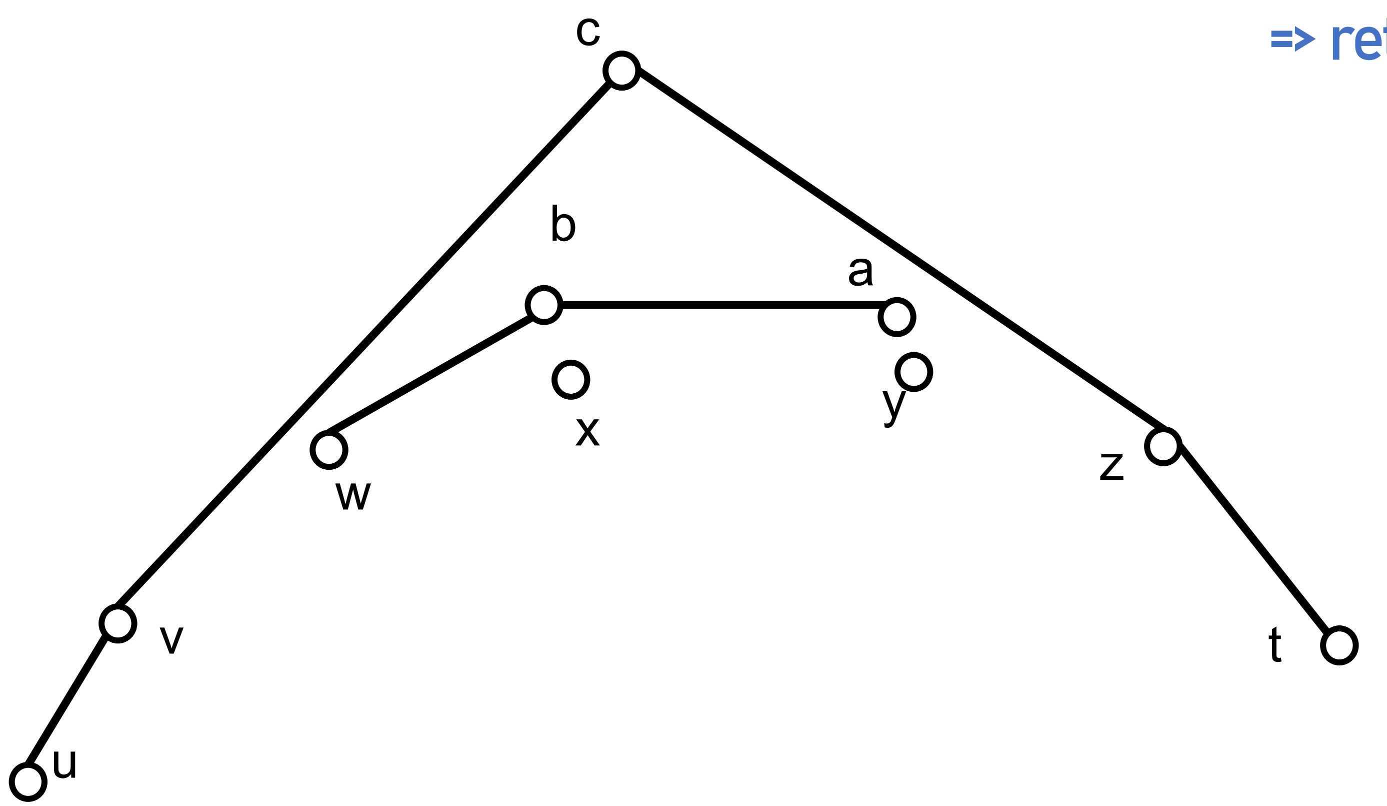
Running example

- w-b-a: buried by c

=> Remove w-b and b-a

- v-c-z: no point visible

=> return



Existing hull: u-v-w-x-y-z-t, adding a, b, c in order

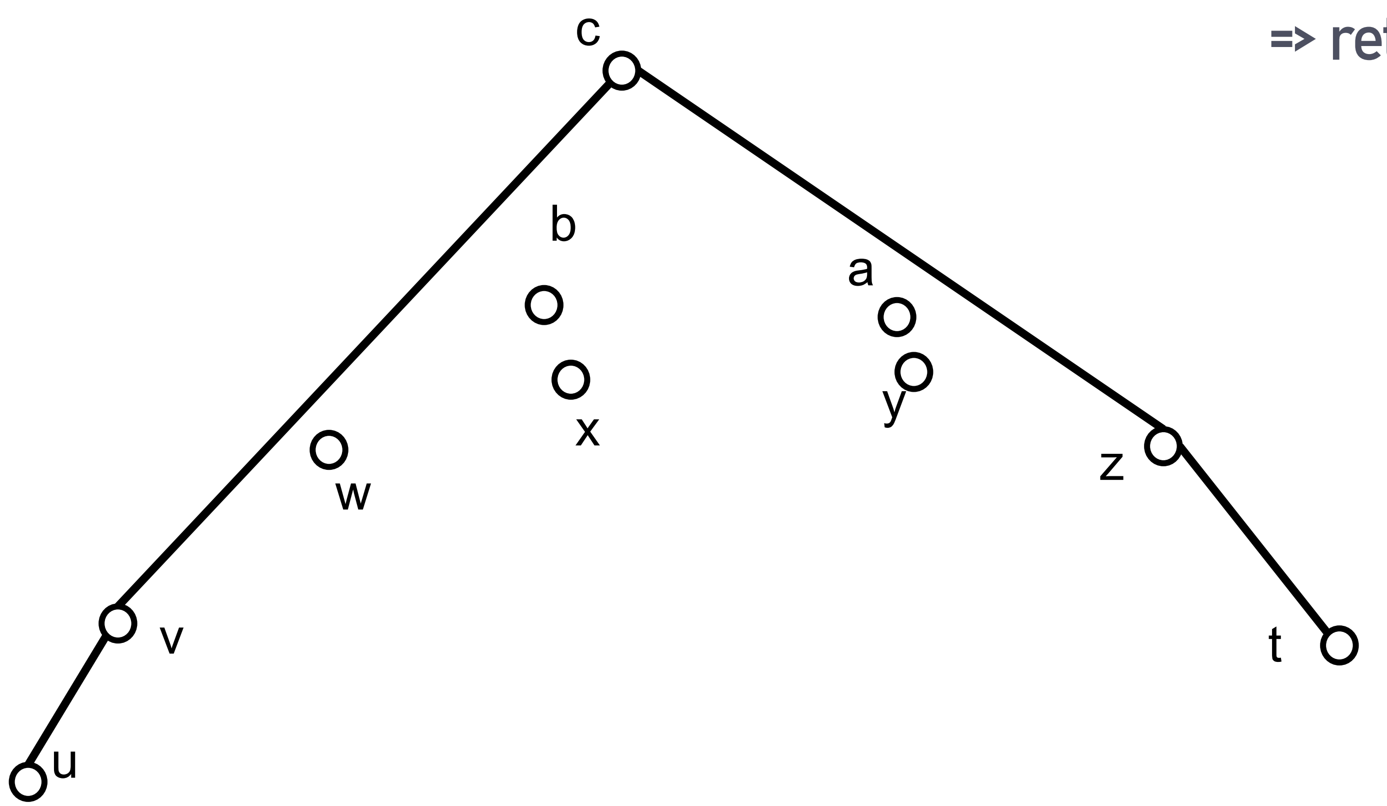
Running example

- w-b-a: buried by c

=> Remove w-b and b-a

- v-c-z: no point visible

=> return



Existing hull: u-v-w-x-y-z-t, adding a, b, c in order