

maxflow1-handout (1)	2
maxflow2-handout (1)	18
RA-handout	36
hashing-handout-2024	70
NPC1_fast	101
NPC2_fast	140
exactexp-handout-2024	180
veb-handout-dec24	206
TriangulationMA (1)	228
approx1-jan2025	361
approx2-jan2025 (1)	439

Good Afternoon.

# Advanced algorithms and data structures

## Lecture 1: Max Flow 1

Jacob Holm ([jaho@di.ku.dk](mailto:jaho@di.ku.dk))

November 18th 2024

# Today's Lecture

## Introduction

## Max flow

- Definitions

- Ford-Fulkerson Method

- Max flow/Min cut Theorem

## Summary

# Introduction to AADS

This course is mostly about algorithms and how to analyse them.

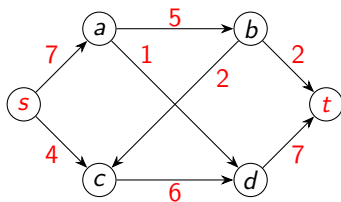
We want *efficient* solutions when possible, where the meaning of “efficient” may depend on the problem.

In particular, we will focus on Polynomial time algorithms (fast) versus Exponential time algorithms (slow).

This weeks topic has a polynomial time algorithm. Later we will touch on some problems where we don't know or expect polynomial time algorithms to exist, and some ways to deal with that.



## Flow network

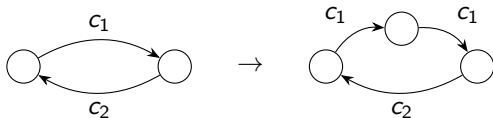
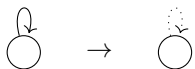


### Definition

A *flow network* consists of a directed graph  $G = (V, E)$ , a source  $s \in V$ , a sink  $t \in V \setminus \{s\}$ , and a capacity function  $c : V \times V \rightarrow \mathbb{R}$  such that

- ▶  $c(u, v) \geq 0$  for all  $u, v \in V$ , and
- ▶ if  $(u, v) \notin E$  then  $c(u, v) = 0$

We will assume that  $G$  has **no self-loops** and **no antiparallel edges**.



Example of a flow network.

Graph with nodes  $s$  and  $t$ . Send goods/data/water from  $s$  to  $t$ . Can not accumulate in intermediate nodes, so what goes in must come out (flow conservation).

Capacities, capacity constraint.

Max flow can be used by itself, or as black box for solving other problems.

No self-loops or antiparallel edges allowed by our algorithms/theorems. This is without loss of generality as we can always get rid of them.

# Flow and max-Flow

## Definition

A **flow** in flow network  $(G, s, t, c)$  is a function  $f : V \times V \rightarrow \mathbb{R}$  such that:

1.  $\forall u, v \in V: 0 \leq f(u, v) \leq c(u, v)$  (capacity constraints)

2.  $\forall v \in V \setminus \{s, t\}: \sum_{u \in V} f(u, v) = \sum_{w \in V} f(v, w)$  (flow conservation)

Equivalently:  $\sum_{(u,v) \in E} f(u, v) = \sum_{(v,w) \in E} f(v, w)$ . **Why?**

## Definition

The **value**  $|f|$  of a flow  $f$  is defined as:

$$|f| := \sum_{v \in V} f(s, v) - \sum_{u \in V} f(u, s) = \sum_{v \in V} (f(s, v) - f(v, s))$$

## Definition

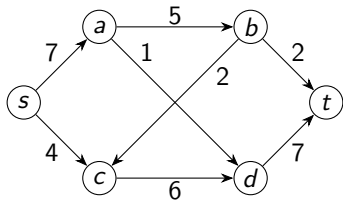
A **max-flow** is a flow of maximum value.

Value could also have been defined as net flow entering  $t$ .

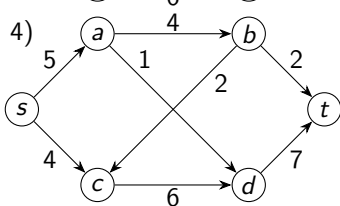
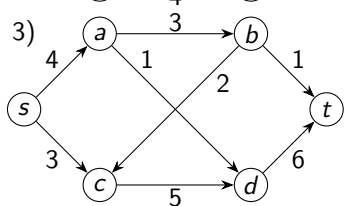
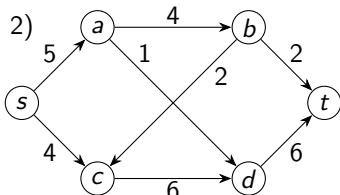
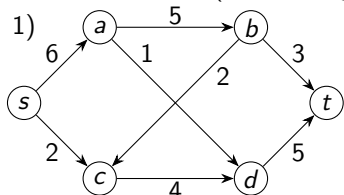
Our goal in the max-flow problem is to compute such a max-flow.

## Examples: Which of these are flows? What are the values?

Flow network  
(capacities on edges)



Candidate flows (flow on edges)



Ex 1: No. Capacity violation at  $(b, t)$ .

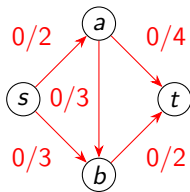
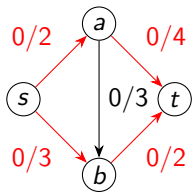
Ex 2: No. Flow conservation violation at  $d$ .

Ex 3: Yes. Value 7

Ex 4: Yes. Value 9. Actually a max flow.

## Ford-Fulkerson Method (informal)

```
function FORD-FULKERSON( $G = (V, E)$ ,  $s$ ,  $t$ ,  $c$ )  
   $f \leftarrow 0$   
  while  $\exists$  “augmenting path”  $p$  from  $s$  to  $t$  do  
    Send as much flow as possible along  $p$  and “add” this to  $f$ .  
  return  $f$ 
```



Example 1:

Example 2: cancelling flow

Break

## Residual network

Recall that  $G$  has no self-loops or anti-parallel edges.

### Definition

Given a flow  $f$  in  $(G, s, t, c)$ , the **residual capacity** is the function

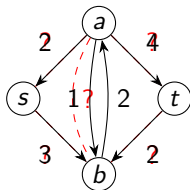
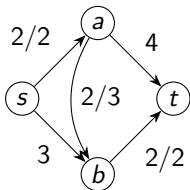
$c_f : V \times V \rightarrow \mathbb{R}$  defined by

$$c_f(u, v) := \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \text{ (i.e. how much more could be sent)} \\ f(v, u) & \text{if } (v, u) \in E \text{ (i.e. how much can be cancelled)} \\ 0 & \text{otherwise} \end{cases}$$

### Definition

The **residual network** consist of the graph  $G_f := (V, E_f)$  where  $E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$ , together with  $s, t$ , and the capacity function  $c_f$ .

Note that  $(G_f, s, t, c_f)$  is a flow network (but may have antiparallel edges).

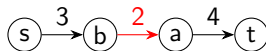


Example: What are the edges and residual capacities.

## Ford-Fulkerson Method

```
function FORD-FULKERSON( $G = (V, E), s, t, c$ )  
   $f \leftarrow 0$   
  while  $\exists$  (augmenting) path  $p$  from  $s$  to  $t$  in  $G_f$  do  
    Find a max flow  $f_p$  along  $p$  in  $G_f$ .  
     $f \leftarrow f \uparrow f_p$   
  return  $f$ 
```

What is the max flow along a path?



Illustrate “max flow along  $p$ ”.

$f \uparrow f'$  defined later.

Not an algorithm because we don't specify how to pick the path  $p$ .

## Augmented flow, Lemma 1

### Definition

Given a flow  $f$  in  $G$  and a flow  $f'$  in  $G_f$ , the **augmented flow**

$f \uparrow f'$ :  $V \times V \rightarrow \mathbb{R}$  is

$$(f \uparrow f')(u, v) := \begin{cases} f(u, v) + f'(u, v) - f'(v, u) & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

### Lemma

$f \uparrow f'$  is a flow in  $G$  of value  $|f \uparrow f'| = |f| + |f'|$ .

### Proof of capacity constraint.

Let  $(u, v) \in E$  (otherwise it is trivial), then

$$\begin{aligned} (f \uparrow f')(u, v) &= f(u, v) + f'(u, v) - f'(v, u) \\ &\leq f(u, v) + c_f(u, v) - 0 \\ &= f(u, v) + c(u, v) - f(u, v) = c(u, v) \end{aligned}$$

$$\begin{aligned} (f \uparrow f')(u, v) &= f(u, v) + f'(u, v) - f'(v, u) \\ &\geq f(u, v) + 0 - c_f(v, u) \\ &= f(u, v) + 0 - f(u, v) = 0 \end{aligned}$$

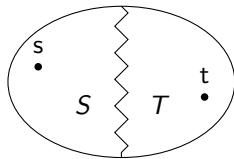
□



# Cut, flow across and capacity of

## Definition

A **cut** is a partition of  $V$  into subsets  $S \ni s$  and  $T \ni t$ .



## Definition

Given a flow  $f$  and a cut  $(S, T)$  we define the **net flow across  $(S, T)$**  as

$$f(S, T) := \sum_{u \in S} \sum_{v \in T} (f(u, v) - f(v, u))$$

## Definition

Given a cut  $(S, T)$  we define the **capacity of  $(S, T)$**  as

$$c(S, T) := \sum_{u \in S} \sum_{v \in T} c(u, v)$$

## Lemma 2 (net flow value)

### Lemma

Given a flow  $f$  in  $G$ , for all cuts  $(S, T)$  we have  $f(S, T) = |f|$ .

### Proof.

$$\begin{aligned} f(S, T) &= \sum_{u \in S} \sum_{v \in T} (f(u, v) - f(v, u)) \\ &\stackrel{?}{=} \sum_{u \in S} \sum_{v \in S} (f(u, v) - f(v, u)) + \sum_{u \in S} \sum_{v \in T} (f(u, v) - f(v, u)) \\ &= \sum_{u \in S} \sum_{v \in V} (f(u, v) - f(v, u)) \\ &= \sum_{u \in \{s\}} \sum_{v \in V} (f(u, v) - f(v, u)) + \sum_{u \in S \setminus \{s\}} \sum_{v \in V} (f(u, v) - f(v, u)) \\ &\stackrel{?}{=} \sum_{v \in V} (f(s, v) - f(v, s)) + 0 \\ &=: |f| \end{aligned}$$

□

In other words, the net flow value across any cut  $(S, T)$  is equal to the value of the flow.

## Corollary (flow value upper bounded by cut capacity)

### Corollary

For any flow  $f$  and any cut  $(S, T)$ ,  $|f| \leq c(S, T)$ .

### Proof.

$$\begin{aligned} |f| &= f(S, T) && \text{(By Lemma 2)} \\ &= \sum_{u \in S} \sum_{v \in T} (f(u, v) - f(v, u)) && \text{(By definition of } f(S, T)) \\ &\leq \sum_{u \in S} \sum_{v \in T} (c(u, v) - 0) && \left( \begin{array}{l} \text{Since } f(u, v) \leq c(u, v) \\ \text{and } -f(v, u) \leq 0 \text{ by the} \\ \text{capacity constraints} \end{array} \right) \\ &= c(S, T) && \square \end{aligned}$$

Illustrate on number line, all possible flow values are left of all possible cut capacities.

Max flow/Min cut Theorem says they meet in the middle.

# Max flow/Min cut Theorem

## Theorem (Max flow/Min cut Theorem)

*Let  $f$  be a flow in  $(G, s, t, c)$ . Then the following 3 statements are equivalent:*

- 1.  $f$  is a max flow.*
- 2. There is no augmenting path (in  $G_f$ ).*
- 3. There exists a cut  $(S, T)$  such that  $|f| = c(S, T)$ .*

Q: What does this say about Ford-Fulkerson?

A: If it terminates, it returns the correct result.

# Summary

Today's topic was Max Flow. We have covered

- ▶ Definition of flow network, flow, etc
- ▶ The Ford-Fulkerson Method
- ▶ The Max flow/Min cut Theorem

Next time:

- ▶ Proof of Max flow/Min cut Theorem
- ▶ Worst case analysis of Ford-Fulkerson
- ▶ Edmonds-Karp Algorithm

Good Afternoon.

# Advanced algorithms and data structures

## Lecture 2: Max Flow 2

Jacob Holm ([jaho@di.ku.dk](mailto:jaho@di.ku.dk))

November 20th 2024

# Today's Lecture

## Max flow

- Recap

- Ford-Fulkerson analysis

- Edmonds-Karp

- Integrality Theorem

## Summary

## Recap 1

Flow network  $(G, s, t, c)$ , no self-loops or antiparallel edges.

Flow  $f : V \times V \rightarrow \mathbb{R}$  satisfies:

1.  $\forall u, v \in V : 0 \leq f(u, v) \leq c(u, v)$  (capacity constraints)
2.  $\forall u \in V \setminus \{s, t\} : \sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$  (flow conservation)

Value  $|f| = \sum_{v \in V} (f(s, v) - f(v, s))$ .

Residual capacity  $c_f : c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise} \end{cases}$

Residual network  $(G_f, s, t, c_f)$  where  $G_f = (V, E_f)$  and  $E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$ . This is a flow network.

Given flow  $f$  in  $G$  and  $f'$  in  $G_f$ ,

$(f \uparrow f')(u, v) = \begin{cases} f(u, v) + f'(u, v) - f'(v, u) & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$



## Recap 2

```
1: function FORD-FULKERSON( $G = (V, E), s, t, c$ )
2:    $f \leftarrow 0$ 
3:   while  $\exists$  (augmenting) path  $p$  from  $s$  to  $t$  in  $G_f$  do
4:     Find a max flow  $f_p$  along  $p$  in  $G_f$ .
5:      $f \leftarrow f \uparrow f_p$ 
6:   return  $f$ 
```

A cut is a partition of  $V$  into sets  $S \ni s$  and  $T \ni t$ .

Definition:  $f(S, T) := \sum_{u \in S} \sum_{v \in T} f(u, v) - f(v, u)$ .

Definition:  $c(S, T) := \sum_{u \in S} \sum_{v \in T} c(u, v)$ .

Lemma 1:  $f \uparrow f'$  is a flow in  $G$  of value  $|f \uparrow f'| = |f| + |f'|$ .

Lemma 2:  $\forall$  flows  $f$  and cuts  $(S, T)$ ,  $|f| = f(S, T)$ .

Corollary:  $\forall$  flows  $f$  and cuts  $(S, T)$ ,  $|f| \leq c(S, T)$ .

## Max flow/Min cut Theorem

Given a flow  $f$  in  $G$ , the following 3 statements are equivalent:

1.  $f$  is a max flow.
2. There is no augmenting path (a path  $s \rightsquigarrow t$  in  $G_f$ ).
3.  $\exists$  cut  $(S, T)$  such that  $|f| = c(S, T)$ .

(1)  $\implies$  (2).

Assume for contradiction  $f$  is a max flow and there exists an augmenting path  $p$ . Then by Lemma 1,  $f \uparrow f_p$  is a flow in  $G$  of value  $|f \uparrow f_p| = |f| + |f_p| > |f|$ , which is a contradiction.  $\square$

# Max flow/Min cut Theorem

Given a flow  $f$  in  $G$ , the following 3 statements are equivalent:

1.  $f$  is a max flow.
2. There is no augmenting path (a path  $s \rightsquigarrow t$  in  $G_f$ ).
3.  $\exists$  cut  $(S, T)$  such that  $|f| = c(S, T)$ .

(2)  $\implies$  (3).

Let  $S = \{v \in V \mid v \text{ is reachable from } s \text{ in } G_f\}$ ,  $T = V \setminus S$ . Then  $S, T$  partition  $V$ ,  $s \in S$  (why?) and  $t \in T$  (why? no augmenting path), so  $(S, T)$  is a cut.

Now let  $u \in S, v \in T$ . Then  $f(u, v) = c(u, v)$  (why?), otherwise  $c_f(u, v) > 0$  so  $(u, v) \in E_f$ . Since  $u$  is reachable from  $s$  in  $G_f$  that implies  $v$  is reachable from  $s$  in  $G_f$  thus  $v \in S$  contradicting  $v \in T = V \setminus S$ .

Similarly,  $f(v, u) = 0$  (why?), otherwise  $c_f(u, v) > 0$  and same problem.

Thus

$$\begin{aligned} |f| &= f(S, T) && \text{(By Lemma 2)} \\ &= \sum_{u \in S} \sum_{v \in T} (f(u, v) - f(v, u)) && \text{(Definition of } f(S, T)) \\ &= \sum_{u \in S} \sum_{v \in T} (c(u, v) - 0) && \text{(Above argument)} \\ &= c(S, T) && \text{(Definition of } c(S, T)) \quad \square \end{aligned}$$

$s \in S$  because obviously  $s$  is reachable from itself.

$t \in T$  because otherwise  $t$  would be reachable from  $s$  by some path  $p$ , but this would be a augmenting path contradicting 2.

## Max flow/Min cut Theorem

Given a flow  $f$  in  $G$ , the following 3 statements are equivalent:

1.  $f$  is a max flow.
2. There is no augmenting path (a path  $s \rightsquigarrow t$  in  $G_f$ ).
3.  $\exists$  cut  $(S, T)$  such that  $|f| = c(S, T)$ .

(3)  $\implies$  (1).

Let  $(S, T)$  be the cut from (3), and  $f'$  be any other flow in  $G$ . By the Corollary,  $|f'| \leq c(S, T) = |f|$ , so  $f$  is a max flow.  $\square$

## Ford-Fulkerson worst case analysis

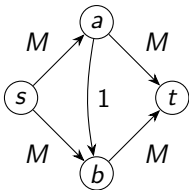
In general Ford-Fulkerson is not guaranteed to terminate,  
i.e. there exists flow networks and bad choices of augmenting paths such  
that it never terminates.

For this to happen, some capacities need to be irrational.

If all capacities are integers, F.F. does at most  $|f^*|$  iterations, where  $f^*$  is  
a max flow (why?).

Assuming each iteration can be done in  $\mathcal{O}(E)$  time, that gives a running  
time of  $\mathcal{O}(E \cdot |f^*|)$ .

Bad case example:



Edmonds-Karp algorithm avoids the bad case by always choosing the  
**shortest** augmenting path.

We are slightly abusing notation here and writing  $E$  instead of  $|E|$ .

# Edmonds-Karp Algorithm

```
1: function EDMONDS-KARP( $G = (V, E), s, t, c$ )
2:    $f \leftarrow 0$ 
3:   while  $\exists$  (augmenting) path from  $s$  to  $t$  in  $G_f$  do
4:      $p \leftarrow$  shortest such path.
5:     Find a max flow  $f_p$  along  $p$  in  $G_f$ .
6:      $f \leftarrow f \uparrow f_p$ 
7:   return  $f$ 
```

## Theorem

*The number of iterations of Edmonds-Karp is  $\mathcal{O}(V \cdot E)$ .*

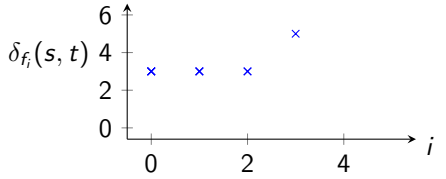
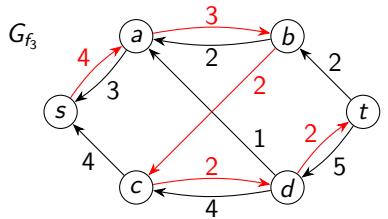
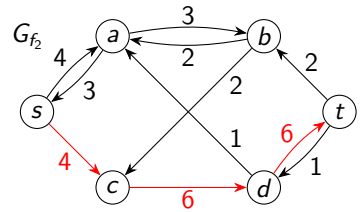
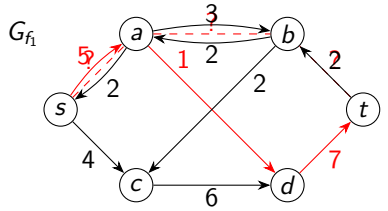
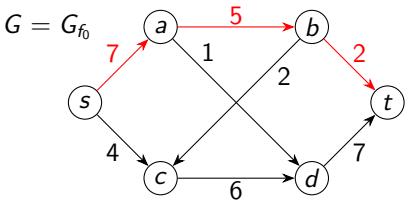
## Corollary

*Edmonds-Karp can be implemented to run in  $\mathcal{O}(V \cdot E^2)$  time.*

Thus, Edmonds-Karp is a polynomial-time algorithm for max flow.

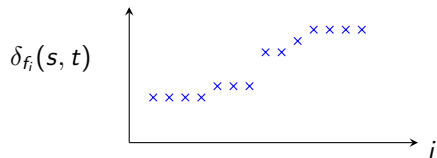
Break

# Edmonds-Karp Example





## Edmonds-Karp Proof sketch of #iterations



We will show that the distance  $\delta_{f_i}(s, t)$  is nondecreasing, and that it can only stay the same for at most  $E$  consecutive iterations.

This implies that #iterations in Edmonds-Karp is  $\mathcal{O}(V \cdot E)$ , **why?**

Each time the  $s \rightsquigarrow t$  distance changes, it is increased by at least 1, and if a path exists the distance is at most  $V - 1$ .  $\implies$  distance increases  $\mathcal{O}(V)$  times.

So we have  $\mathcal{O}(V)$  “runs”, each of at most  $E$  consecutive iterations where the  $s \rightsquigarrow t$  distance is unchanged.  $\implies \mathcal{O}(V \cdot E)$  iterations in total.

## Edmonds-Karp Level sets and forward/backward edges

Consider consecutive flows  $f_0, \dots, f_k$  found by Edmonds-Karp, where  $\delta_{f_0}(s, t) = \delta_{f_1}(s, t) = \dots = \delta_{f_k}(s, t)$ .

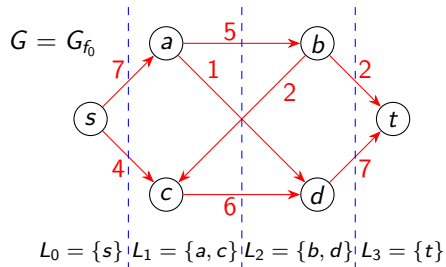
For  $d = 0, 1, \dots$  let

$L_d = \{v \in V \mid \delta_{f_0}(s, v) = d\}$ , i.e.

$L_d$  is the “BFS layer” consisting of vertices at distance  $d$  from  $s$  in  $G_{f_0}$ .

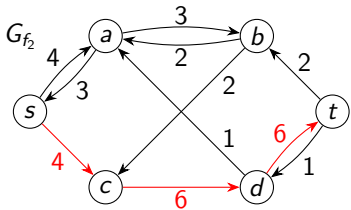
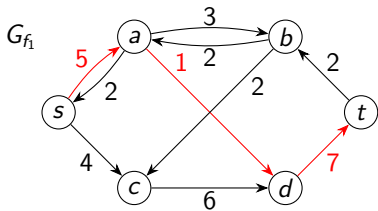
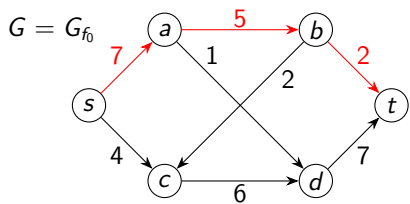
A **forward edge** (of  $G_{f_i}$ ) is an edge  $(u, v) \in G_{f_i}$  such that for some  $d$ ,  $u \in L_d$  and  $v \in L_{d+1}$ .

A **backward edge** (of  $G_{f_i}$ ) is an edge  $(u, v) \in G_{f_i}$  such that for some  $d$ ,  $u \in L_{d+1}$  and  $v \in L_d$ .



(It is possible for an edge to be neither forward nor backward)

## Edmonds-Karp Claim 1



Claim 1: For  $i = 0, \dots, k$ , Edmonds-Karp finds an augmenting path in  $G_{f_i}$  consisting only of edges that are forward edges in  $G_{f_0}$ .

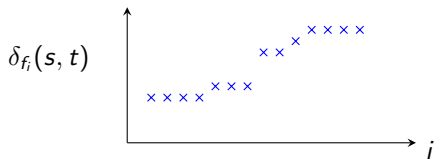
Note that Claim 1 implies  $k = \mathcal{O}(E)$ , **why?**

At least one forward edge gets saturated in iteration  $i$ , which removes it from  $G_{f_{i+1}}$ .

# Edmonds-Karp Claim 2

Claim 2: If there is an augmenting path in  $G_{f_{k+1}}$ , then  $\delta_{f_{k+1}}(s, t) \geq \delta_{f_k}(s, t)$ .

Note that Claim 1 and Claim 2 together gives the claimed #iterations for Edmonds-Karp.



## Edmonds-Karp Proof sketch for Claim 1 & 2

Claim 1: For  $i = 0, \dots, k$ , Edmonds-Karp finds an augmenting path in  $G_{f_i}$  consisting only of edges that are forward edges in  $G_{f_0}$ .

### Proof sketch.

This is clear for  $i = 0$ , because a shortest path can only use forward edges.

So suppose  $1 \leq i \leq k$ .  $G_{f_i}$  is obtained from  $G_{f_{i-1}}$  by removing forward edges (at least one!) and adding backward edges.

Since  $\delta_{f_i}(s, t) = \delta_{f_0}(s, t)$  any shortest  $s \rightsquigarrow t$  path can only use forward edges, which must have stayed forward edges since  $G_{f_0}$ .

The claim follows by induction.  $\square$

Claim 2: If there is an augmenting path in  $G_{f_{k+1}}$ , then  $\delta_{f_{k+1}}(s, t) \geq \delta_{f_k}(s, t)$ .

### Proof sketch.

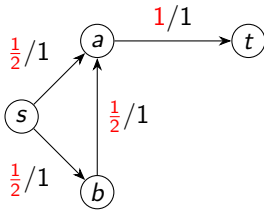
$G_{f_{k+1}}$  is obtained from  $G_{f_0}$  by removing forward edges and adding backward edges. This can never reduce the distance.  $\square$

## Integrality Theorem

Integrality Theorem: Given integer capacities, Ford-Fulkerson (and therefore Edmonds-Karp) will find an integer-valued flow

$f : V \times V \rightarrow \mathbb{Z}_{\geq 0}$  with  $|f| \in \mathbb{Z}_{\geq 0}$  an integer.

Note that not all max flows in a network with integer capacities have to be integer-valued. Q: Can you find a max flow in this example that is not integer-valued?



# Summary

This finished the topic on Max Flow. We have covered

- ▶ Proof of Max flow/Min cut Theorem
- ▶ Worst case analysis of Ford-Fulkerson
- ▶ Edmonds-Karp Algorithm
- ▶ Integrality Theorem

Next time:

- ▶ Linear Programming

# AADS, Lecture 4

## Randomized Algorithms

Jacob Holm (jaho@di.ku.dk)

November 27th 2024

Good afternoon. My name is Jacob Holm.

You can help by asking questions during class if there is anything that is not clear.

Remember, if it is not clear to you, then it is probably also unclear to at least one other person in the room.

You can help more than just yourself by asking for clarification.

I am also teaching the Randomized Algorithms course, and the next two lectures are a tiny taste of that.



# Why Randomized Algorithms?

- ▶ Faster, **but weaker guarantees**.
- ▶ Simpler code, **but harder to analyze**.
- ▶ Sometimes only option, e.g. Big Data, Streaming, Machine Learning, Security, (Differential) Privacy, etc.

**Therefore this course!**

# Today's Lecture

## Quicksort

- Linearity of expectation
- Expectation of indicator variable

## Min-Cut

- Conditional probabilities
- Time/error probability tradeoff

## Las Vegas vs Monte Carlo

## Summary

## AADS Lecture 4 (RA), Part 1

### Quicksort

## Basic Quicksort [Hoare]

```
1: function QS( $S = \{s_1, \dots, s_n\}$ )  
   ▷ Assumes all elements in  $S$  are distinct.  
2:   if  $|S| \leq 1$  then  
3:     return list( $S$ )  
4:   else  
5:     Pick pivot  $x \in S$ , (How?)  
6:      $L \leftarrow \{y \in S \mid y < x\}$   
7:      $R \leftarrow \{y \in S \mid y > x\}$   
8:     return QS( $L$ ) +  $[x]$  + QS( $R$ )
```

For each $y \in S \setminus \{x\}$ , compare to $y$ to $x$ once
--

Q: Does anyone see what essential part is missing from this description?

### Lemma

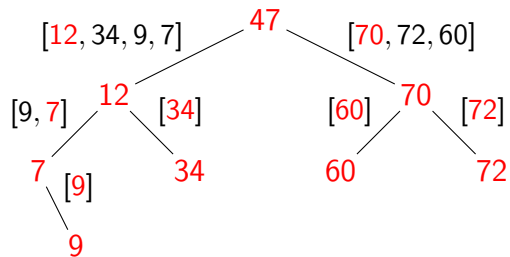
*For any pivoting strategy, QS correctly sorts the numbers.*

### Proof.

By induction on  $n$ .  $n = 0, 1$  is trivial, so assume it holds for up to  $n - 1$  numbers. Then by our induction hypothesis QS( $L$ ) and QS( $R$ ) are sorted, so QS( $L$ ) +  $[x]$  + QS( $R$ ) is sorted.  $\square$

## Quicksort Example 1

Sorting  $S = [70, 12, 34, 47, 9, 72, 60, 7]$ .



Total #comparisons:  $4 + 3 + 2 + 1 + 1 + 1 + 1 = 13$

Assuming we always pick the “middle” element we have

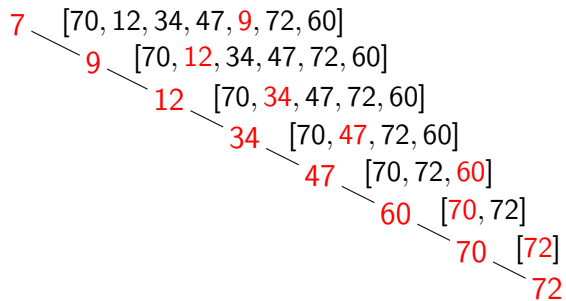
$T(n) \leq 2T(n/2) + \mathcal{O}(n)$ , so by the Master Theorem

$T(n) \in \mathcal{O}(n \log n)$ .

As you can see this looks rather like a balanced binary search tree, so you might expect the number of comparisons to be small. Something like  $n \log n$ .

## Quicksort Example 2

Sorting  $S = [70, 12, 34, 47, 9, 72, 60, 7]$ .



Total #comparisons:  $7 + 6 + 5 + 4 + 3 + 2 + 1 = 28$

Assuming we always pick an “extreme” element we have  
 $T(n) = T(0) + T(n-1) + \Theta(n)$ , and thus  $T(n) \in \Theta(n^2)$ .

As you can see this looks like an extremely unbalanced binary search tree, so you might expect the number of comparisons to be large. Something like  $n^2$ .

# Randomized Quicksort

```
1: function RANDQS( $S = \{s_1, \dots, s_n\}$ )  
   ▷ Assumes all elements in  $S$  are distinct.  
2:   if  $|S| \leq 1$  then  
3:     return  $S$   
4:   else  
5:     Pick pivot  $x \in S$ , uniformly at random  
6:      $L \leftarrow \{y \in S \mid y < x\}$    For each  $y \in S \setminus \{x\}$ ,  
7:      $R \leftarrow \{y \in S \mid y > x\}$    compare to  $y$  to  $x$  once  
8:     return  $\text{RANDQS}(L) + [x] + \text{RANDQS}(R)$ 
```

# Randomized Quicksort, Analysis

Q: What is the expected number of comparisons?

## Theorem

$$\mathbb{E}[\#comparisons] \in \mathcal{O}(n \log n)$$

Let  $[S_{(1)}, \dots, S_{(n)}] := \text{RANDQS}(S)$ .

For  $i < j$  let  $X_{ij}$  be the number of times that  $S_{(i)}$  and  $S_{(j)}$  are compared. We can then compute

$$\#comparisons = \sum_{i < j} X_{ij}$$

$$\mathbb{E}[\#comparisons] = \mathbb{E}\left[\sum_{i < j} X_{ij}\right] = \sum_{i < j} \mathbb{E}[X_{ij}]$$

Uses *linearity of expectation*:

$$\mathbb{E}[A + B] = \mathbb{E}[A] + \mathbb{E}[B]$$

Important:  $S_{(i)}$  is the  $i$ th element **in the final sorted order**.

Note that the  $\sum_{i < j}$  is really a shorthand for  $\sum_{1 \leq i < j \leq n}$ , or even more explicit  $\sum_{i=1}^{n-1} \sum_{j=i+1}^n$ .



## Randomized Quicksort, Analysis

Observe that  $X_{ij} \in \{0, 1\}$  (**why?**).

Since  $X_{ij} \in \{0, 1\}$ , it is an *indicator variable* for the event that  $S_{(i)}$  and  $S_{(j)}$  are compared. Let  $p_{ij}$  be the probability of this event. Then

$$\begin{aligned}\mathbb{E}[X_{ij}] &= \sum_{x \in \{0,1\}} \Pr[X_{ij} = x] \cdot x && \text{(Def. of } \mathbb{E} \text{)} \\ &= (1 - p_{ij}) \cdot 0 + p_{ij} \cdot 1 = p_{ij}\end{aligned}$$

Thus *the expectation of an indicator variable equals the probability of the indicated event.*

Therefore

$$\mathbb{E}[\text{\#comparisons}] = \sum_{i < j} \mathbb{E}[X_{ij}] = \sum_{i < j} p_{ij}$$

# Randomized Quicksort, Analysis

## Lemma

$S_{(i)}$  and  $S_{(j)}$  are compared iff  $S_{(i)}$  or  $S_{(j)}$  is first of  $S_{(i)}, \dots, S_{(j)}$  to be chosen as pivot.

## Proof.

Each recursive call returns some sublist  $[S_{(a)}, \dots, S_{(b)}]$ . Let  $x = S_{(c)}$  be the pivot.

Suppose  $a \leq i < j \leq b$ . 

$a$	$\dots$	$i$	$\dots$	$j$	$\dots$	$b$
-----	---------	-----	---------	-----	---------	-----

$c < i$  or  $c > j$ :  $S_{(i)}$  and  $S_{(j)}$  not compared now, but together in recursion. Recursion stops when  $i \leq c \leq j$ .

$i < c < j$ :  $S_{(i)}$  and  $S_{(j)}$  never compared.

$c \in \{i, j\}$ :  $S_{(i)}$  and  $S_{(j)}$  compared once.



## Randomized Quicksort, Analysis

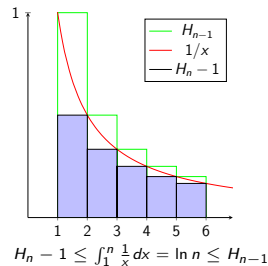
Thus,  $p_{ij}$  is the conditional probability of picking  $S_{(i)}$  or  $S_{(j)}$  given that the pivot is picked uniformly at random in  $\{S_{(i)}, S_{(i+1)}, \dots, S_{(j)}\}$ :

$$\begin{aligned} p_{ij} &= \Pr[c \in \{i, j\} \mid c \in \{i, i+1, \dots, j\} \text{ u.a.r.}] \\ &= \frac{2}{|\{i, i+1, \dots, j\}|} = \frac{2}{j+1-i} \end{aligned}$$

It follows that

$$\mathbb{E}[\text{\#comparisons}] = \sum_{i < j} p_{ij} = \sum_{i < j} \frac{2}{j+1-i}$$

# Randomized Quicksort, Analysis



$$\begin{aligned}
 \mathbb{E}[\text{\#comparisons}] &= \sum_{i < j} \frac{2}{j+1-i} \\
 &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j+1-i} \\
 &= \sum_{i=1}^{n-1} \sum_{k=2}^{n+1-i} \frac{2}{k} < \sum_{i=1}^n \sum_{k=2}^n \frac{2}{k} \\
 &= 2n \sum_{k=2}^n \frac{1}{k} = 2n \left( \left( \sum_{k=1}^n \frac{1}{k} \right) - 1 \right) = 2n(H_n - 1) \\
 &\leq 2n \int_1^n \frac{1}{x} dx = 2n \ln n \in \mathcal{O}(n \log n)
 \end{aligned}$$

- From before.
- Expanding the  $\sum_{i < j}$  notation.
- The denominator  $k = j + 1 - i$  takes each value from  $2, \dots, n + 1 - i$  once.
- Since all terms are positive, adding more terms can only increase the value
- Moving 2 outside the sums and noting that the inner sum does not depend on  $i$ .
- Adding and subtracting the term for  $k = 1$ .
- Using the definition of  $H_n$ .
- Observing that  $H_n - 1 \leq \int_1^n \frac{1}{x} dx = \ln n \leq H_{n-1} = H_n - \frac{1}{n}$ .

## Randomized Quicksort, Summary

When  $|S| = n$ , the expected number of comparisons done by  $\text{RANDQS}(S)$  is less than  $2nH_n \in \mathcal{O}(n \log n)$  for any input.

Even stronger (see Problem 4.14), we can show that the number of comparisons is  $\mathcal{O}(n \log n)$  with high probability.

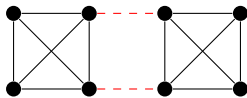
## AADS Lecture 4 (RA), Part 2

### Min-Cut

## Min-Cut

This is a related, but different notion of Min-Cut than in the “Max-Flow Min-Cut theorem” from lecture 1 and 2.

Problem: Given a connected graph  $G = (V, E)$



Find smallest  $C \subseteq E$  that splits  $G$ .

$C$  is called a *(global) min-cut*, and  $\lambda(G) := |C|$  is the *edge connectivity* of  $G$ .

## Randomized Min-Cut [Karger & Stein]

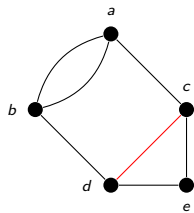
```
1: function RANDMINCUT( $V, E$ )  
2:   while  $|V| > 2$  and  $E \neq \emptyset$  do  
3:     Pick  $e \in E$  uniformly at random.  
4:     Contract  $e$  and remove self-loops.  
5:   return  $E$ 
```



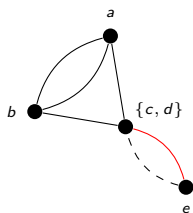
# Randomized Min-Cut, Example

- 1: **function** RANDMINCUT( $V, E$ )
- 2:     **while**  $|V| > 2$  and  $E \neq \emptyset$  **do**
- 3:         Pick  $e \in E$  uniformly at random.
- 4:         Contract  $e$  and remove self-loops.
- 5:     **return**  $E$

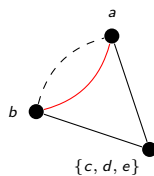
$G_0 = G$



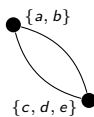
$G_1 = G_0 / e_1$



$G_2 = G_1 / e_2$



$G_3 = G_2 / e_3$



## Randomized Min-Cut, Analysis

```
1: function RANDMINCUT( $V, E$ )
2:   while  $|V| > 2$  and  $E \neq \emptyset$  do
3:     Pick  $e \in E$  uniformly at random.
4:     Contract  $e$  and remove self-loops.
5:   return  $E$ 
```

### Lemma

$\text{RANDMINCUT}(G)$  *always returns a cut.*

### Proof.

Proof by induction on the number  $k$  of iterations of the loop (note  $k \leq n - 2$ ). If  $k = 0$  it is trivial, so suppose that it is true for up to  $k - 1$  iterations. The first iteration constructs graph  $G'$  by contracting an edge from  $G$  and removing self-loops, and then do at most  $k - 1$  further iterations starting from  $G'$  so by the induction hypothesis we return a cut in  $G'$ . But every such cut is also a cut in  $G$ .  $\square$

## Randomized Min-Cut, Analysis

```
1: function RANDMINCUT( $V, E$ )  
2:   while  $|V| > 2$  and  $E \neq \emptyset$  do  
3:     Pick  $e \in E$  uniformly at random.  
4:     Contract  $e$  and remove self-loops.  
5:   return  $E$ 
```

### Observation

$\text{RANDMINCUT}(G)$  may return a cut of size  $> \lambda(G)$ .

### Lemma

A specific min-cut  $C$  is returned iff no edge from  $C$  was contracted.

# Randomized Min-Cut, Analysis

## Theorem

*For any min-cut  $C$ , the probability that  $\text{RANDOMCUT}(G)$  returns  $C$  is  $\geq \frac{2}{n(n-1)}$ .*

Let  $e_1, \dots, e_{n-2}$  be the contracted edges, let  $G_0 = G$  and  $G_i = G_{i-1}/e_i$ .

Let  $\mathcal{E}_i$  be the (good) event that  $e_i \notin C$ .

$C$  is returned iff  $\mathcal{E}_1 \cap \dots \cap \mathcal{E}_{n-2}$ .

Goal:  $\Pr[\mathcal{E}_1 \cap \dots \cap \mathcal{E}_{n-2}] \geq \frac{2}{n(n-1)}$

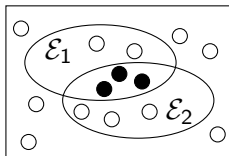
- In words,  $\mathcal{E}_i$  is the event that the  $i$ th edge contracted is not in  $C$ , i.e., the  $i$ th contraction does not destroy  $C$ .
- $\mathcal{E}_1 \cap \dots \cap \mathcal{E}_{n-2}$  is thus the event that  $C$  is not destroyed in any step of the algorithm.

## Conditional Probabilities

This is easy to prove by induction.

Given events  $\mathcal{E}_1, \mathcal{E}_2$  with  $\Pr[\mathcal{E}_1] > 0$ , the *conditional probability* of  $\mathcal{E}_2$  given  $\mathcal{E}_1$  is defined as

$$\Pr[\mathcal{E}_2|\mathcal{E}_1] = \frac{\Pr[\mathcal{E}_1 \cap \mathcal{E}_2]}{\Pr[\mathcal{E}_1]}$$



It follows that

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2] = \Pr[\mathcal{E}_1] \cdot \Pr[\mathcal{E}_2|\mathcal{E}_1]$$

And in general for events  $\mathcal{E}_1, \dots, \mathcal{E}_k$

$$\Pr[\cap_{i=1}^k \mathcal{E}_i] = \Pr[\mathcal{E}_1] \cdot \Pr[\mathcal{E}_2|\mathcal{E}_1] \cdots \Pr[\mathcal{E}_k | \cap_{i=1}^{k-1} \mathcal{E}_i]$$

## Randomized Min-Cut, Proof

$$\begin{aligned} & \Pr[\text{specific min-cut } C \text{ returned}] \\ &= \Pr[\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}] \\ &= \Pr[\mathcal{E}_1] \cdot \Pr[\mathcal{E}_2 | \mathcal{E}_1] \cdots \Pr[\mathcal{E}_{n-2} | \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-3}] \\ &= \prod_{i=1}^{n-2} p_i \quad \text{where } p_i = \Pr[\mathcal{E}_i | \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{i-1}] \end{aligned}$$

## Randomized Min-Cut, Proof

$G_i = (V_i, E_i)$  has  $n_i = n - i$  vertices. (why?)

Contractions can not decrease the min-cut size (why?)

so  $\lambda(G_i) \geq |C|$ .

It follows that each vertex  $v$  of  $G_i$  has degree  $d_i(v)$  at least  $|C|$ . (why?)

Summing up all degrees of  $G_i$ ,

$$|E_i| = \frac{1}{2} \sum_{v \in V_i} d_i(v) \geq \frac{1}{2} n_i |C|.$$

- Each contraction reduces the number of vertices by 1.
- Every cut in  $G_i$  is a cut in  $G_{i-1}$  and therefore in  $G$ .
- Note that the edges incident to a vertex  $v$  form a cut and so  $d_i(v) \geq \lambda(G_i) \geq |C|$ .
- We use that each edge is counted twice in the sum  $\sum_{v \in V_i} d_i(v)$ .

## Randomized Min-Cut, Proof

We have shown that  $G_i = (V_i, E_i)$  has  $n_i = n - i$  vertices and at least  $|E_i| \geq \frac{1}{2}n_i|C|$  edges. We want to lower bound

$$p_i = \Pr[\text{uniformly random } e \in E_{i-1} \text{ is not in } C \mid \cap_{j=1}^{i-1} \mathcal{E}_j]$$

The complementary probability, i.e. the probability of picking an edge of  $C$  in the  $i$ th iteration, given that no edge of  $C$  has been picked in a previous iteration, is

$$\begin{aligned} 1 - p_i &= \Pr[\text{uniformly random } e \in E_{i-1} \text{ is in } C \mid \cap_{j=1}^{i-1} \mathcal{E}_j] \\ &= \frac{|C|}{|E_{i-1}|} \leq \frac{|C|}{\frac{1}{2}n_{i-1}|C|} = \frac{2}{n_{i-1}} = \frac{2}{n - (i - 1)} \\ \Rightarrow p_i &\geq 1 - \frac{2}{n - i + 1} = \frac{n - i - 1}{n - i + 1} \end{aligned}$$

We have shown upper bound

$$1 - p_i \leq \frac{2}{n - i + 1}$$

so now we can lower bound  $p_i$ .



## Randomized Min-Cut, Proof

Telescoping product.

$\Pr[C \text{ returned}]$

$$\begin{aligned} &= \prod_{i=1}^{n-2} p_i \quad \text{where } p_i = \Pr[\mathcal{E}_i | \mathcal{E}_1 \cap \dots \cap \mathcal{E}_{i-1}] \\ &\geq \prod_{i=1}^{n-2} \frac{n-1-i}{n+1-i} \\ &= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} \\ &= \frac{2}{n(n-1)} \end{aligned}$$

## Randomized Min-Cut, Summary

So for given min-cut  $C$ ,  $\Pr[C \text{ is returned}] \geq \frac{2}{n(n-1)}$ .

Is this tight? I.e. do we have examples matching this bound?

Yes! Consider the cycle  $C_n$  on  $n$  vertices. Every one of the  $\binom{n}{2} = \frac{n(n-1)}{2}$  pairs of edges is a min-cut and all pairs are equally likely to be returned.

Is this probability good?

How can we improve it?

# Randomized Min-Cut, Tradeoff

Imagine calling  $\text{RANDMINCUT}(G)$   $t \frac{n(n-1)}{2}$  times and letting  $C^*$  be the smallest cut returned.

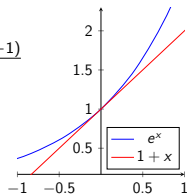
$$\begin{aligned}\Pr[C^* \text{ is not a min-cut}] &\leq \left(1 - \frac{2}{n(n-1)}\right)^{t \frac{n(n-1)}{2}} \\ &\leq \left(e^{-\frac{2}{n(n-1)}}\right)^{t \frac{n(n-1)}{2}}\end{aligned}$$

(This uses that  $1 + x \leq e^x$  for all  $x \in \mathbb{R}$ , see Proposition B.3.1)

$$= e^{-t}$$

$$\begin{aligned}\Pr[C^* \text{ is a min-cut}] &\geq 1 - e^{-t} \\ &= 1 - n^{-c} \quad (\text{If we set } t = c \ln n)\end{aligned}$$

Thus for any  $c > 0$  if we repeat  $c \cdot \frac{n(n-1)}{2} \cdot \ln n$  times, the probability of getting a min-cut is at least  $1 - n^{-c}$ . We call this *high probability of success*.



- In each call to  $\text{RANDMINCUT}(G)$ , the probability that a min-cut is not returned is at most  $1 - \frac{2}{n(n-1)}$ .
- Since the calls to  $\text{RANDMINCUT}(G)$  are independent, the probability that no min-cut is among the cuts returned is the product.
- $1 + x \leq e^x$
- Choosing e.g.  $t = 21$  we reduce the error probability to around one in a billion.
- Choosing  $t = c \ln n$  for constant  $c$ , we get a *high probability of success*, namely at least  $1 - e^{-c \ln n} = 1 - 1/n^c$ .
- We thus get a tradeoff between running time and probability of success.

## Randomized Min-Cut, Simple implementation

In practice, using a “Union-Find” data structure.

```
1: function RANDMINCUT( $V, E$ )
2:   for  $u \in V$  do
3:     MAKE-SET( $u$ )
4:    $C \leftarrow \emptyset$ ,  $\pi \leftarrow$  a random permutation of  $E$ ,  $r \leftarrow |V|$ 
5:   for  $uv \in E$  in the order  $\pi$  do
6:      $p_u \leftarrow \text{FIND}(u)$ ,  $p_v \leftarrow \text{FIND}(v)$ 
7:     if  $p_u \neq p_v$  then
8:       if  $r > 2$  then
9:          $r \leftarrow r - 1$ 
10:      UNION( $p_u, p_v$ )
11:     else
12:        $C \leftarrow C \cup \{uv\}$ 
13:   return  $C$ 
```

The running time for this is  $\mathcal{O}(m\alpha(n))$ . Running it  $\mathcal{O}(n^2 \log n)$  times to get high probability takes  $\mathcal{O}(n^2 m \alpha(n) \log n)$  time.

## Deterministic Min-Cut

Pick arbitrary  $s \in V$ . For each  $t \in V \setminus \{s\}$  compute max-flow from  $s$  to  $t$ . Return the minimum.

What is the running time? We run Ford-Fulkerson  $n - 1$  times. Each run takes  $\mathcal{O}(m|f^*|)$  time where  $|f^*| \leq m$ . In total  $\mathcal{O}(nm^2)$  time. For dense graphs this is much worse than  $\mathcal{O}(n^2 m \alpha(n) \log n)$ .

**Best algorithm known (2024):**  $\tilde{\mathcal{O}}(m)$ -time algorithm by Kawarabayashi and Thorup [JACM 2019].

Slightly improved by Henzinger, Rao and Wang [SICOMP20].

Extended to weighted graphs by Henzinger, Li, Rao and Wang [SODA 2024]

Edmonds-Karp does not help us here, at least not without a better analysis. For unit-capacity graphs like this one, the bound  $\mathcal{O}(m|f^*|) = \mathcal{O}(m^2)$  is better than the  $\mathcal{O}(nm^2)$  we get for Edmonds-Karp. On the other hand, since Edmonds-Karp *is* a version of Ford-Fulkerson, you can/should still use it, as the same improved analysis applies.

# Las Vegas vs Monte Carlo

What is the main difference between the guarantees of `RANDQS` and `RANDMINCUT`?

**Las Vegas:** Always returns correct answer. #steps used is a random variable.

**Monte Carlo:** Some probability of error. #steps used may be random or not.

## Converting L.V. $\leftrightarrow$ M.C.

As part of Assignment 2 you will prove that we can sometimes convert a Monte Carlo algorithm into a Las Vegas algorithm.

How about the other direction? Can we always take a Las Vegas algorithm running in expected  $\mathcal{O}(f(n))$  time and turn it into a Monte Carlo algorithm running in worst case  $\mathcal{O}(f(n))$  time?

# Summary

- ▶ We analyzed a Las Vegas randomized algorithm (RANDQS).
- ▶ We analyzed a Monte Carlo randomized algorithm (RANDOMCUT).
- ▶ We discussed how the two types of algorithms are related and may often be converted into each other.



Next time

Hashing.

Good Afternoon.

# Advanced algorithms and data structures

## Lecture 5: Hashing

Jacob Holm (`jaho@di.ku.dk`)

December 2nd 2024

# Today's Lecture

## Hashing

- Hashing fundamentals

- Application: Unordered sets/Hashing with chaining

- Application: Signatures

- Practical hash functions

- Application: Coordinated sampling

# Preliminaries

## Notation:

For  $n \in \mathbb{N}$ :

$$[n] = \{0, \dots, n-1\}$$

$$[n]_+ = \{1, \dots, n-1\}$$

Iverson bracket:

$$[\text{condition}] = \begin{cases} 1 & \text{if condition is true} \\ 0 & \text{if condition is false} \end{cases}$$

For a random variable  $X$ :

$$\mu_X = \mathbb{E}[X] \quad (\text{expectation})$$

$$\text{Var}[X] = \mathbb{E}[(X - \mu_X)^2] \quad (\text{variance})$$

$$\sigma_X = \sqrt{\text{Var}[X]} \quad (\text{std. deviation})$$

## Inequalities:

Expectation of indicator variable  $X$ :

$$\mathbb{E}[X] = \Pr[X = 1]$$

Linearity of expectation:

$$\mathbb{E}\left[\sum_i X_i\right] = \sum_i \mathbb{E}[X_i]$$

Sum of pairwise indep. variances:

$$\text{Var}\left[\sum_i X_i\right] = \sum_i \text{Var}[X_i]$$

Union bound:

$$\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$$

Markov's Inequality: For  $X \geq 0, t > 0$

$$\Pr[X \geq t] \leq \frac{\mathbb{E}[X]}{t} = \frac{\mu_X}{t}$$

Chebyshev's Inequality: For  $t > 0$

$$\Pr[|X - \mu_X| \geq t\sigma_X] \leq \frac{1}{t^2}$$

## AADS Lecture 5 (Hashing), Part 1

### Hashing fundamentals

# Hash function

Given a (typically large) universe  $U$  of keys, and a positive integer  $m$ .

## Definition

A *random hash function*  $h : U \rightarrow [m]$  is a randomly chosen function from some family of functions mapping  $U$  to  $[m]$ . Equivalently, it is a function  $h$  such that for each  $x \in U$ ,  $h(x) \in [m]$  is a random variable.

Cryptographic “hash functions” such as MD5, SHA-1, and SHA-256 are not *random* hash functions, and do not have most of the properties we want here. Do not confuse them!

# Hash function

When discussing random hash functions, we usually care about

1. Space (*seed size*) needed to represent  $h$ .
2. Time needed to calculate  $h(x)$  given  $x \in U$ .
3. Properties of the random variable.

# Hash function types

## Definition

A hash function  $h : U \rightarrow [m]$  is *truly random* if the variables  $h(x)$  for  $x \in U$  are independent and uniform in  $[m]$ .

Impractical, why? **Space! Require  $|U| \log_2 m$  bits to represent.**

## Definition

A random hash function  $h : U \rightarrow [m]$  is *universal* if, for all  $x \neq y \in U$ :  $\Pr_h[h(x) = h(y)] \leq \frac{1}{m}$ .

## Definition

A random hash function  $h : U \rightarrow [m]$  is *strongly universal* (a.k.a. 2-independent) if,

- ▶ Each key is hashed uniformly into  $[m]$ .  
(i.e.  $\forall x \in U, q \in [m] : \Pr_h[h(x) = q] = \frac{1}{m}$ )
- ▶ Any two distinct keys hash independently.

Or equivalently, if for all  $x \neq y \in U$ , and  $q, r \in [m]$ :  
 $\Pr_h[h(x) = q \wedge h(y) = r] = \frac{1}{m^2}$ .

There are  $m^{|U|}$  possible functions from  $U$  to  $[m]$ , so it takes at least  $\log_2(m^{|U|}) = |U| \log_2 m$  bits to store which one we picked.

We use  $\Pr_h[\dots]$  or  $\Pr_h[\dots]$  instead of just  $\Pr[\dots]$  to make it clear that the probability is based on the random choice of  $h$ , and *not* on e.g.  $x, y$  being chosen at random.

For many purposes  $c$ -approximately universal hash functions for some small constant  $c$  are enough. We will see examples of such functions a little later today.



# Hash function types

## Definition

A hash function  $h : U \rightarrow [m]$  is *truly random* if the variables  $h(x)$  for  $x \in U$  are independent and uniform in  $[m]$ .

Impractical, why? **Space! Require  $|U| \log_2 m$  bits to represent.**

## Definition

A random hash function  $h : U \rightarrow [m]$  is *c-approximately universal* if, for all  $x \neq y \in U$ :  $\Pr_h[h(x) = h(y)] \leq \frac{c}{m}$ .

## Definition

A random hash function  $h : U \rightarrow [m]$  is *c-approximately strongly universal* if,

- ▶ Each key is hashed *c-approximately* uniformly into  $[m]$ .  
(i.e.  $\forall x \in U, q \in [m] : \Pr_h[h(x) = q] \leq \frac{c}{m}$ )
- ▶ Any two distinct keys hash independently.

**Implying that** for all  $x \neq y \in U$ , and  $q, r \in [m]$ :  
 $\Pr_h[h(x) = q \wedge h(y) = r] \leq ?$  (See Assignment 3 exercise 3.2).

There are  $m^{|U|}$  possible functions from  $U$  to  $[m]$ , so it takes at least  $\log_2(m^{|U|}) = |U| \log_2 m$  bits to store which one we picked.

We use  $\Pr_h[\dots]$  or  $\Pr_h[\dots]$  instead of just  $\Pr[\dots]$  to make it clear that the probability is based on the random choice of  $h$ , and *not* on e.g.  $x, y$  being chosen at random.

For many purposes *c-approximately universal* hash functions for some small constant  $c$  are enough. We will see examples of such functions a little later today.

## AADS Lecture 5 (Hashing), Part 2

Application:

Unordered sets/Hashing with chaining

# Unordered sets/Dictionarys

Maintain a set  $S$  of at most  $n$  keys from some unordered universe  $U$ , under

$\text{INSERT}(x, S)$  Insert key  $x$  into  $S$ .

$\text{DELETE}(x, S)$  Delete key  $x$  from  $S$ .

$\text{MEMBER}(x, S)$  Return  $x \in S$ .

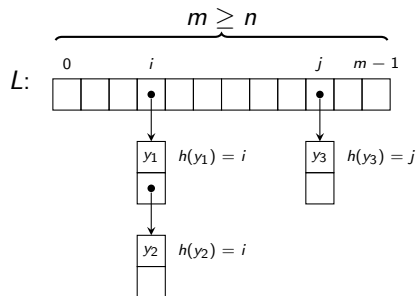
We could use some form of balanced tree to store  $S$ , but they usually take  $\mathcal{O}(\log n)$  or  $\mathcal{O}(\log \log U)$  time per operation, and we want each operation to run in expected constant time.

# Hashing with chaining

Idea: Pick  $m \geq n$  and a *universal*  $h : U \rightarrow [m]$ .

Store array  $L$ , where

$L[i] =$  linked list over  $\{y \in S \mid h(y) = i\}$ .



Then  $x \in S \iff x \in L[h(x)]$ .

Each operation takes  $\mathcal{O}(|L[h(x)]| + 1)$  time.

# Hashing with chaining

## Theorem

For  $x \notin S$ ,  $\mathbb{E}_h[|L[h(x)]|] \leq \frac{n}{m} \quad (\leq 1)$

## Proof.

$$\begin{aligned}\mathbb{E}_h[|L[h(x)]|] &= \mathbb{E}_h[|\{y \in S \mid h(y) = h(x)\}|] \\ &= \mathbb{E}_h\left[\sum_{y \in S} [h(y) = h(x)]\right] \\ &= \sum_{y \in S} \mathbb{E}_h[h(y) = h(x)] \\ &= \sum_{y \in S} \Pr_h[h(y) = h(x)] \\ &\leq |S| \frac{1}{m} \leq \frac{n}{m} \leq 1\end{aligned}$$

□

By definition of  $L[i] := \{y \in S \mid h(y) = i\}$ .

Here we use the *Iverson Bracket* notation

$$[P] := \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{if } P \text{ is false} \end{cases}$$

This can often be used as a shorthand for an indicator variable.

In this case  $[h(y) = h(x)]$  becomes an indicator variable for the event  $h(y) = h(x)$ .

Linearity of expectation.

Expectation of indicator variable

Since  $x \notin S$  and  $y \in S$ , we have  $x \neq y$ .

Then by definition of a universal hash function  $h : U \rightarrow [m]$ ,  $\Pr_h[h(y) = h(x)] \leq \frac{1}{m}$ .

## Hashing with chaining

Similarly, one can show that:

### Theorem (Exercise)

$$\text{For } x \in S, \mathbb{E}_h[|L[h(x)]|] \leq 1 + \frac{n-1}{m} \quad \left( \leq 2 \right)$$

So we can conclude that each of the operations  $\text{INSERT}(x, S)$ ,  $\text{DELETE}(x, S)$ , and  $\text{MEMBER}(x, S)$  takes expected constant time when using hashing with chaining with a universal hash function.

## AADS Lecture 5 (Hashing), Part 3

### Application: Signatures

## Application: Signatures

Problem: Assign a unique “signature” to each  $x \in S \subseteq U$ ,  
 $|S| = n$ .

Solution: Use universal hash function  $s : U \rightarrow [n^3]$ .

Then we fail (by having a collision) with probability

$$\begin{aligned} & \Pr_s[\exists \{x, y\} \subseteq S \text{ with } s(x) = s(y)] \\ &= \Pr_s\left[\bigcup_{\{x, y\} \subseteq S} \text{event } s(x) = s(y)\right] \\ &\leq \sum_{\{x, y\} \subseteq S} \Pr_s[s(x) = s(y)] && \text{(Union bound)} \\ &\leq \frac{\binom{n}{2}}{n^3} && (s \text{ universal}) \\ &< \frac{1}{2n} \end{aligned}$$

Thus with “high probability” we have no collisions.

It is tempting to try to calculate a probability of no collisions more directly as a product over all pairs of the probability of that pair not colliding, i.e. as:

$$\begin{aligned} \prod_{\{x, y\} \in S} \Pr_s[s(x) \neq s(y)] &\geq \left(1 - \frac{1}{n^3}\right)^{\binom{n}{2}} \\ &= \left(1 + \frac{1}{n^3 - 1}\right)^{-\binom{n}{2}} \\ &\geq e^{\left(\frac{-\binom{n}{2}}{n^3 - 1}\right)} \\ &\geq e^{-\frac{1}{2n}} \\ &> 1 - \frac{1}{2n} \end{aligned}$$

While this happens to give the right answer in this case, the calculation is invalid because the events are not independent.

We use the union bound instead because that does not require independence.



## AADS Lecture 5 (Hashing), Part 4

### Practical hash functions

## Multiply-mod-prime

Let  $U = [u]$  and pick prime  $p \geq u$ . For any  $a, b \in [p]$ , and  $m < u$ , let  $h_{a,b}^m : U \rightarrow [m]$  be

$$h_{a,b}^m(x) = ((ax + b) \bmod p) \bmod m$$

Is this a random hash function? **NO!**

## Multiply-mod-prime

Let  $U = [u]$  and pick prime  $p \geq u$ . For any  $a, b \in [p]$ , and  $m < u$ , let  $h_{a,b}^m : U \rightarrow [m]$  be

$$h_{a,b}^m(x) = ((ax + b) \bmod p) \bmod m$$

Choose  $a, b \in [p]$  independently and uniformly at random, and let  $h(x) := h_{a,b}^m(x)$ .

Then  $h : U \rightarrow [m]$  is a 2-approximately strongly universal hash function.

## Multiply-shift

Let  $U = [2^w]$  and  $m = 2^\ell$ . For any odd  $a \in [2^w]$  define

$$h_a(x) := \left\lfloor \frac{(ax) \bmod 2^w}{2^{w-\ell}} \right\rfloor$$

Choose odd  $a \in [2^w]$  uniformly at random, and let

$$h(x) := h_a(x).$$

Then  $h : U \rightarrow [m]$  is a 2-approximately universal hash function.

(Assignment 3 exercise 3.4 asks you to show that it is not  $c$ -approximately strongly universal for any constant  $c$ ).

## Multiply-shift, C

For  $U = [2^{64}]$  the C code looks like this:

```
#include<stdint.h>
uint64_t hash(uint64_t x, uint64_t l, uint64_t a)
{
    return (a*x) >> (64-l);
}
```

## Reminder: Strongly universal

Recall:

### Definition

A random hash function  $h : U \rightarrow [m]$  is *strongly universal* (a.k.a. 2-independent) if,

- ▶ Each key is hashed uniformly into  $[m]$ .  
(i.e.  $\forall x \in U, q \in [m] : \Pr_h[h(x) = q] = \frac{1}{m}$ )
- ▶ Any two distinct keys hash independently.

## Reminder: Strongly universal

Recall:

### Definition

A random hash function  $h : U \rightarrow [m]$  is *c-approximately strongly universal* if,

- ▶ Each key is hashed *c-approximately* uniformly into  $[m]$ .  
(i.e.  $\forall x \in U, q \in [m] : \Pr_h[h(x) = q] \leq \frac{c}{m}$ )
- ▶ Any two distinct keys hash independently.

## Strong Multiply-shift

Let  $U = [2^w]$  and  $m = 2^\ell$ , and pick  $\bar{w} \geq w + \ell - 1$ . For any pair  $(a, b) \in [2^{\bar{w}}]^2$  define

$$h_{a,b}(x) := \left\lfloor \frac{(ax + b) \bmod 2^{\bar{w}}}{2^{\bar{w}-\ell}} \right\rfloor$$

Choose  $a, b \in [2^{\bar{w}}]$  independently and uniformly at random, and let  $h(x) := h_{a,b}(x)$ .

Then  $h : U \rightarrow [m]$  is a strongly universal hash function.



## Strong Multiply-shift, C

For  $\ell \leq w = 32$  and  $\bar{w} = 64$  we have  $U = [2^{32}]$  and the  $C$  code looks like this:

```
#include<stdint.h>
uint32_t hash(uint32_t x, uint32_t l,
              uint64_t a, uint64_t b)
{
    return (a*x+b) >> (64-l);
}
```

## AADS Lecture 5 (Hashing), Part 5

Application: Coordinated sampling

## Application: Coordinated sampling

Suppose we have a bunch of *agents* that each observe some set of events from some universe  $U$ . Let  $A_i \subseteq U$  denote the set of events seen by agent  $i$ , and suppose  $|A_i|$  is large so only a small sample  $S_i \subseteq A_i$  is actually stored.

At some later point we are interested in a subset  $A^* \subseteq \bigcup_i A_i$  (not known in advance), and in particular we want to estimate the size  $|A^*|$ .

## Application: Coordinated sampling

If each agent independently just samples a random subset of the seen events, there is very little chance that two agents that see an event make the same decision.

⇒ The samples are incomparable (almost useless).

Coordinated sampling means that all agents that see an event make the same decision about whether to store it.

⇒ Samples can be combined, i.e.

- ▶  $S_i \cup S_j$  is a sample of  $A_i \cup A_j$
- ▶  $S_i \cap S_j$  is a sample of  $A_i \cap A_j$

In particular,  $A^* \cap \bigcup_i S_i$  is a sample of  $A^*$  and can be computed from  $\bigcup_i S_i$  just by determining for each element whether it belongs to  $A^*$ .

## Application: Coordinated sampling

Let  $h : U \rightarrow [m]$  be a strongly universal hash function, and let  $t \in \{0, \dots, m\}$ . Send  $h$  and  $t$  to all the agents.

Each agent samples  $x \in U$  iff  $h(x) < t$ .

Thus if an agent sees the set  $A_i \subseteq U$ , the set

$S_{h,t}(A_i) := \{x \in A_i \mid h(x) < t\}$  is sampled. Note that

- ▶  $S_{h,t}(A_i) \cup S_{h,t}(A_j) = S_{h,t}(A_i \cup A_j)$
- ▶  $S_{h,t}(A_i) \cap S_{h,t}(A_j) = S_{h,t}(A_i \cap A_j)$

Each  $x \in U$  (if seen) is sampled with probability

$\Pr_h[h(x) < t] = \frac{t}{m}$ . Why? **Strong universality**  $\implies$   **$h(x)$  uniform in  $[m]$**

For any  $A \subseteq U$ ,  $\mathbb{E}_h[|S_{h,t}(A)|] = |A| \cdot \frac{t}{m}$ .

Thus we have an unbiased estimate

$$|A^*| \approx \frac{m}{t} \cdot |S_{h,t}(A^*)| = \frac{m}{t} \cdot |A^* \cap \bigcup_i S_i|.$$

How good is this estimate, i.e. what can we say about the

$$\text{relative error} := \left| \frac{\text{estimated value} - \text{actual value}}{\text{actual value}} \right| = \left| \frac{\text{estimated value}}{\text{actual value}} - 1 \right|?$$

$$\begin{aligned} \mathbb{E}_h[|S_{h,t}(A)|] &= \mathbb{E}_h \left[ \sum_{x \in A} [h(x) < t] \right] \\ &= \sum_{x \in A} \mathbb{E}_h[h(x) < t] \\ &= \sum_{x \in A} \Pr_h[h(x) < t] \\ &= \sum_{x \in A} \frac{t}{m} \\ &= |A| \cdot \frac{t}{m} \end{aligned}$$

# Concentration bound

## Lemma

Let  $X = \sum_{a \in A} X_a$  where the  $X_a$  are *pairwise independent* 0–1 variables.  
Let  $\mu = \mathbb{E}[X]$ . Then  $\text{Var}[X] \leq \mu$ , and for any  $q > 0$ ,

$$\Pr[|X - \mu| \geq q\sqrt{\mu}] \leq \frac{1}{q^2}$$

## Proof (not curriculum).

For  $a \in A$  let  $p_a = \Pr[X_a = 1]$ . Then  $p_a = \mathbb{E}[X_a]$  and

$$\begin{aligned}\text{Var}[X_a] &= \mathbb{E}[(X_a - p_a)^2] = (1 - p_a)(0 - p_a)^2 + p_a(1 - p_a)^2 \\ &= (p_a^2 + p_a(1 - p_a))(1 - p_a) = p_a(1 - p_a) \leq p_a\end{aligned}$$

$$\text{Var}[X] = \text{Var}\left[\sum_{a \in A} X_a\right] = \sum_{a \in A} \text{Var}[X_a] \leq \sum_{a \in A} p_a = \mu$$

Finally, since  $\sigma_X = \sqrt{\text{Var}[X]} \leq \sqrt{\mu}$  we get:

$$\begin{aligned}\Pr[|X - \mu| \geq q\sqrt{\mu}] &\leq \Pr[|X - \mu| \geq q\sigma_X] \\ &\leq \frac{1}{q^2} \quad (\text{Chebyshev's ineq.}) \quad \square\end{aligned}$$

## Application: Coordinated sampling

Let's apply this lemma to the estimate  $|A^*| \approx \frac{m}{t} |S_{h,t}(A^*)|$  from our coordinated sampling.

Let  $X = |S_{h,t}(A^*)|$  and for  $a \in A^*$  let  $X_a = [h(a) < t]$ . Then  $X = \sum_{a \in A^*} X_a$  and for any  $a, b \in A^*$ ,  $X_a$  and  $X_b$  are independent. Also, let  $\mu = \mathbb{E}_h[X] = \frac{t}{m} |A^*|$ .

Then for any  $q > 0$ ,

$$\begin{aligned} \Pr_h \left[ \left| \frac{\frac{m}{t} |S_{h,t}(A^*)|}{|A^*|} - 1 \right| \geq q \cdot \frac{1}{\sqrt{\frac{t}{m} |A^*|}} \right] \\ &= \Pr_h \left[ \left| |S_{h,t}(A^*)| - \frac{t}{m} |A^*| \right| \geq q \cdot \sqrt{\frac{t}{m} |A^*|} \right] \\ &= \Pr_h [ |X - \mu| \geq q \cdot \sqrt{\mu} ] \\ &\leq \frac{1}{q^2} \end{aligned}$$

We needed strong universality in two places for this to work.

Where?  **$h$  must be uniform to get unbiased estimate, and pairwise independent for the lemma.**

# Summary

Today's topic was hashing, and we have covered

- ▶ What is a random hash function, and what properties do we want.
- ▶ Two applications of universal hashing — unordered sets and signatures.
- ▶ Some concrete universal or strongly universal hash functions.
- ▶ An application of strongly universal hashing — coordinated sampling.
- ▶ Next time: Computational complexity, P, NP, and NP-completeness with Jakob Nordström.
- ▶ Later: An ordered set data structure that is not comparison based, and an application of hash tables.



# NP-Completeness, part I

Christian Wulff-Nilsen  
Advanced Algorithms and Data Structures  
DIKU

December 12, 2022

## Overview for today

- Problems and decision problems
- Polynomial-time solvable problems
- Definition of  $P$
- Polynomial-time verifiable problems
- Definition of  $NP$
- Reducibility
- NP-completeness
- The circuit-satisfiability problem

## Definition of a problem

- Consider a set  $I$  of *instances* and a set  $S$  of *solutions*.
- An abstract *problem* is a binary relation between  $I$  and  $S$ , i.e., a subset of  $I \times S$ .
- For SHORTEST-PATH, an instance is a triple  $\langle G, s, t \rangle$ .
- A solution is a sequence of vertices forming a shortest  $s$ -to- $t$  path.

## Decision problems

- Unless otherwise stated, we only consider decision problems in this lecture and the next, i.e., problems with 1/0 (yes/no) answers.
- Hence,  $S = \{0, 1\}$ .
- Example of a decision problem: PATH.
- $\text{PATH}(\langle G, u, v, k \rangle) = 1$  if there is a  $u$ -to- $v$  path in  $G$  with at most  $k$  edges.
- Otherwise,  $\text{PATH}(\langle G, u, v, k \rangle) = 0$ .
- We can regard a decision problem as a mapping from instances to  $S = \{0, 1\}$ .
- Instances with solution 1 are called *yes*-instances.
- Instances with solution 0 are called *no*-instances.
- Optimization problems (like SHORTEST-PATH) can usually be turned into decision problems (like PATH).

## Polynomial-time solvable problems

- We assume that instances of a problem are encoded as binary strings.
- An algorithm *solves* a problem in time  $O(T(n))$  if for any instance of length  $n$ , the algorithm returns a solution (0 or 1) in time  $O(T(n))$ .
- If  $T(n) = O(n^k)$  for some constant  $k$ , the problem is *polynomial-time solvable*.
- Suppose we define  $P$  as the class of polynomial-time solvable problems.
- What is missing in this definition? Which encoding of the input is assumed?

## Which encoding to pick?

- Suppose an instance of some problem is a single number  $k$ .
- Suppose there is a  $\Theta(k)$  time algorithm for the problem.
- We could choose an encoding of  $k$  in unary:

$$\overbrace{11 \dots 1}^k.$$

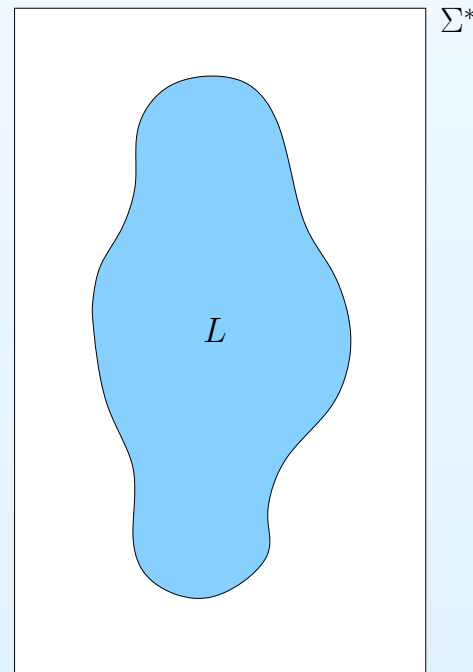
- In this case, the input size is  $n = k$  and the algorithm runs in  $\Theta(n)$  time which is polynomial in the input size.
- We could also choose a much more compact binary encoding, giving input size  $n = \lfloor \lg k \rfloor + 1$ .
- In this case, running time is  $\Theta(k) = \Theta(2^n)$  which is exponential in the input size.
- These two ways of encoding  $k$  correspond to two different problems.

## Which encoding to pick?

- In this lecture and the next, we consider problems with concise encodings.
- In particular, numbers are represented in binary, not unary.
- We use the notation  $\langle x \rangle$  to refer to a chosen encoding of an instance  $x$  of a problem.
- Encodings are always binary strings in our setting.

# Languages

- *Alphabet*: finite set  $\Sigma$  of symbols.
- *Language*  $L$  over  $\Sigma$ : a set of strings of symbols from  $\Sigma$ .
- Example:  $\Sigma = \{a, b, c\}$  and  $L = \{a, ba, cab, bbac, \dots\}$ .
- We also allow an empty string and denote it by  $\epsilon$ .
- The empty language is denoted  $\emptyset$  (it does not contain  $\epsilon$ ).
- $\Sigma^*$  denotes the language of all strings (including  $\epsilon$ ).
- Any language  $L$  over  $\Sigma$  is a subset of  $\Sigma^*$ .





## Languages and decision problems

- Recall that we encode instances of a decision problem as binary strings.
- Also recall that we may view a decision problem as a mapping  $Q(x)$  from instances  $x$  to  $\Sigma = \{0, 1\}$ .
- $Q$  can be specified by the binary strings that encode yes-instances of the problem.
- Thus, we can view  $Q$  as a language  $L$ :

$$L = \{x \in \Sigma^* \mid Q(x) = 1\}.$$

- For instance, PATH is the language of binary strings  $\langle G, u, v, k \rangle$  where  $G$  is a graph,  $u$  and  $v$  are vertices of  $G$ , and there is a  $u$ -to- $v$  path in  $G$  with at most  $k$  edges.

## Language accepted/decided by an algorithm

- Let  $A$  be an algorithm for a decision problem and denote by  $A(x) \in \{0, 1\}$  its output (if any) on input  $x$ .
- $A$  *accepts* a string  $x$  if  $A(x) = 1$ .
- $A$  *rejects* a string  $x$  if  $A(x) = 0$ .
- There may be strings that  $A$  neither accepts nor rejects.
- The language *accepted* by  $A$  is:

$$L = \{x \in \{0, 1\}^* \mid A(x) = 1\}.$$

- Suppose in addition that all strings not in  $L$  are rejected by  $A$ , i.e.,  $A(x) = 0$  for all  $x \in \{0, 1\}^* \setminus L$ .
- Then we say that  $L$  is *decided* by  $A$ .
- Deciding a language is stronger than accepting it.

## Accepting/deciding in polynomial time

- Language  $L$  is *accepted by an algorithm  $A$  in polynomial time* if  $A$  accepts  $L$  and runs in polynomial time on strings from  $L$ .
- $L$  is *decided by  $A$  in polynomial time* if  $A$  decides  $L$  and runs in polynomial time on all strings.
- Example: PATH can both be accepted and decided in polynomial time.
- We can now define the complexity class P:

$$P = \{L \subseteq \{0, 1\}^* \mid \text{there exists an algorithm } A \text{ that} \\ \text{decides } L \text{ in polynomial time}\}.$$

## $P$ in terms of acceptance

- Lemma:

$$\begin{aligned} P &\stackrel{\text{def}}{=} \{L \subseteq \{0, 1\}^* \mid \text{there exists an algorithm that} \\ &\quad \text{decides } L \text{ in polynomial time}\} \\ &= \{L \subseteq \{0, 1\}^* \mid \text{there exists an algorithm that} \\ &\quad \text{accepts } L \text{ in polynomial time}\}. \end{aligned}$$

- $\subseteq$ : straightforward.
- $\supseteq$ : need to show that if  $L$  is accepted by a polynomial-time algorithm  $A$ , it is decided by a polynomial-time algorithm  $A'$ .

## $P$ in terms of acceptance

- Need to show: if  $L$  is accepted by a polynomial-time algorithm  $A$ , it is decided by a polynomial-time algorithm  $A'$ .
- Since  $A$  accepts  $L$ , it runs in at most  $cn^k$  steps before halting on any  $n$ -length string from  $L$ , where  $c$  and  $k$  are constants.
- Now let  $s$  be any string in  $\Sigma^*$ .
- $A'$  simulates  $A$  with input  $s$  for at most  $c|s|^k$  steps.
- If the simulation has not halted after this many steps,  $A'$  halts and outputs 0.
- Otherwise,  $A'$  outputs whatever  $A$  outputs.
- $A'$  decides  $L$  and runs in polynomial time.

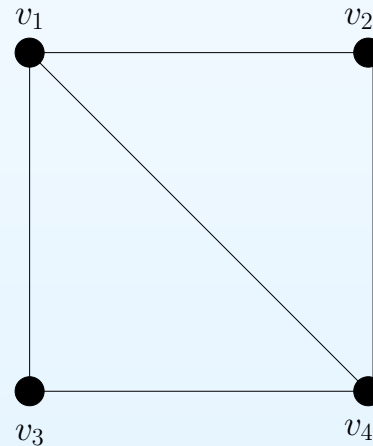
## Verification

- Let  $L$  be a language.
- We might not have an efficient algorithm that accepts  $L$ .
- Consider an algorithm  $A$  taking two parameters,  $x, c \in \Sigma^*$ .
- Instead of trying to find a solution to  $x$  (which may take long time),  $A$  instead *verifies* that  $c$  is a solution to  $x$ .

## The HAM-CYCLE problem

- An undirected graph  $G$  is hamiltonian if it contains a simple cycle containing every vertex of  $G$ .
- We define

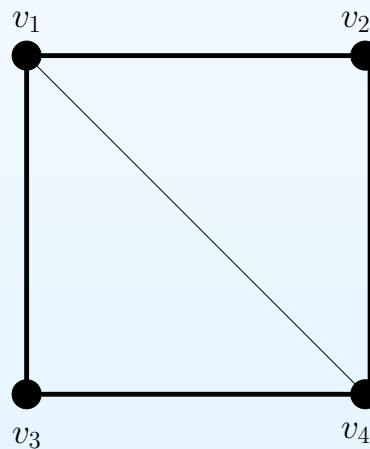
$$\text{HAM-CYCLE} = \{\langle G \rangle \mid G \text{ is Hamiltonian}\}.$$



## The HAM-CYCLE problem

- An undirected graph  $G$  is hamiltonian if it contains a simple cycle containing every vertex of  $G$ .
- We define

$$\text{HAM-CYCLE} = \{\langle G \rangle \mid G \text{ is Hamiltonian}\}.$$





## The HAM-CYCLE problem

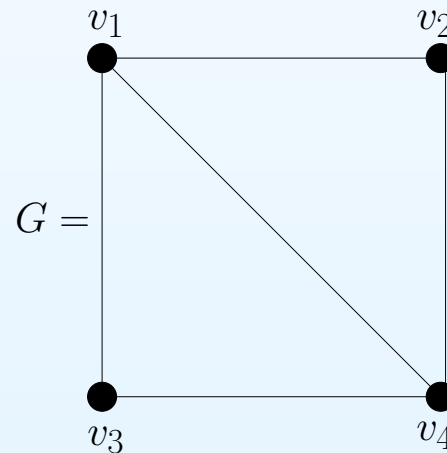
- An undirected graph  $G$  is hamiltonian if it contains a simple cycle containing every vertex of  $G$ .
- We define

$$\text{HAM-CYCLE} = \{\langle G \rangle \mid G \text{ is Hamiltonian}\}.$$

- It is open whether HAM-CYCLE can be decided in polynomial time.
- However, it is easy to show (next slide) that HAM-CYCLE can be verified in polynomial time.

## Verifying HAM-CYCLE

- Consider instead an algorithm  $A_{ham}$  taking two parameters,  $\langle G \rangle$  and  $\langle C \rangle$ .
- $A_{ham}$  checks that  $\langle G \rangle$  defines an undirected graph  $G$  and that  $\langle C \rangle$  encodes a cycle  $C$  containing every vertex of  $G$  exactly once.
- If so,  $A_{ham}$  outputs 1, otherwise 0.

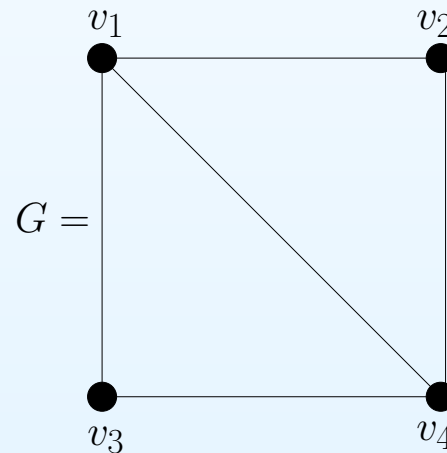


$$C = [v_1, v_2, v_3, v_4]$$

- What is  $A_{ham}(\langle G \rangle, \langle C \rangle)$ ?

## Verifying HAM-CYCLE

- Consider instead an algorithm  $A_{ham}$  taking two parameters,  $\langle G \rangle$  and  $\langle C \rangle$ .
- $A_{ham}$  checks that  $\langle G \rangle$  defines an undirected graph  $G$  and that  $\langle C \rangle$  encodes a cycle  $C$  containing every vertex of  $G$  exactly once.
- If so,  $A_{ham}$  outputs 1, otherwise 0.

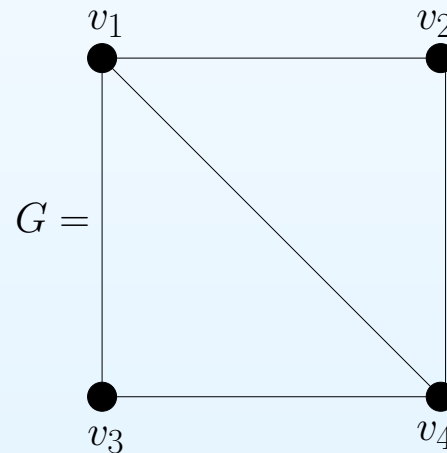


$$C = [v_1, v_2, v_3, v_4]$$

- $A_{ham}(\langle G \rangle, \langle C \rangle) = 0$

## Verifying HAM-CYCLE

- Consider instead an algorithm  $A_{ham}$  taking two parameters,  $\langle G \rangle$  and  $\langle C \rangle$ .
- $A_{ham}$  checks that  $\langle G \rangle$  defines an undirected graph  $G$  and that  $\langle C \rangle$  encodes a cycle  $C$  containing every vertex of  $G$  exactly once.
- If so,  $A_{ham}$  outputs 1, otherwise 0.

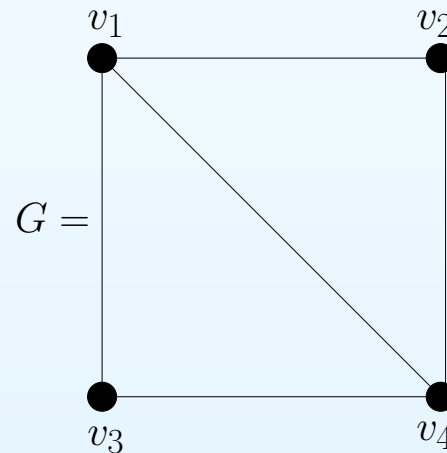


$$C = [v_1, v_2, v_4, v_3]$$

- What is  $A_{ham}(\langle G \rangle, \langle C \rangle)$ ?

## Verifying HAM-CYCLE

- Consider instead an algorithm  $A_{ham}$  taking two parameters,  $\langle G \rangle$  and  $\langle C \rangle$ .
- $A_{ham}$  checks that  $\langle G \rangle$  defines an undirected graph  $G$  and that  $\langle C \rangle$  encodes a cycle  $C$  containing every vertex of  $G$  exactly once.
- If so,  $A_{ham}$  outputs 1, otherwise 0.



$$C = [v_1, v_2, v_4, v_3]$$

- $A_{ham}(\langle G \rangle, \langle C \rangle) = 1$

## Verifying HAM-CYCLE

- Consider instead an algorithm  $A_{ham}$  taking two parameters,  $\langle G \rangle$  and  $\langle C \rangle$ .
- $A_{ham}$  checks that  $\langle G \rangle$  defines an undirected graph  $G$  and that  $\langle C \rangle$  encodes a cycle  $C$  containing every vertex of  $G$  exactly once.
- If so,  $A_{ham}$  outputs 1, otherwise 0.
- Designing  $A_{ham}$  to run in polynomial time is easy.
- Hence we can verify HAM-CYCLE in polynomial time.

## Verifying a language

- A *verification algorithm* is an algorithm  $A$  taking two arguments,  $x, y \in \{0, 1\}^*$ , where  $y$  is the *certificate*.
- $A$  *verifies* a string  $x$  if there is a certificate  $y$  such that  $A(x, y) = 1$ .
- The language verified by  $A$  is

$$L = \{x \in \{0, 1\}^* \mid \text{there is a } y \in \{0, 1\}^* \text{ such that } A(x, y) = 1\}.$$

- Example:

$$\text{HAM-CYCLE} = \{x \in \{0, 1\}^* \mid \text{there is a } y \in \{0, 1\}^* \text{ such that } A_{ham}(x, y) = 1\}.$$

## The complexity class NP

- NP is the class of languages that can be verified in polynomial time.
- More precisely,  $L \in \text{NP}$  if and only if there is a polynomial-time verification algorithm  $A$  and a constant  $c$  such that

$$L = \{x \in \{0, 1\}^* \mid \text{there is a } y \in \{0, 1\}^* \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1\}.$$

- We have seen that  $\text{HAM-CYCLE} \in \text{NP}$ .
- If  $L \in \text{P}$  then  $L \in \text{NP}$ . Why?
- Hence,  $\text{P} \subseteq \text{NP}$ .
- Big open problem: is  $\text{P} = \text{NP}$ ?



## The complexity class co-NP

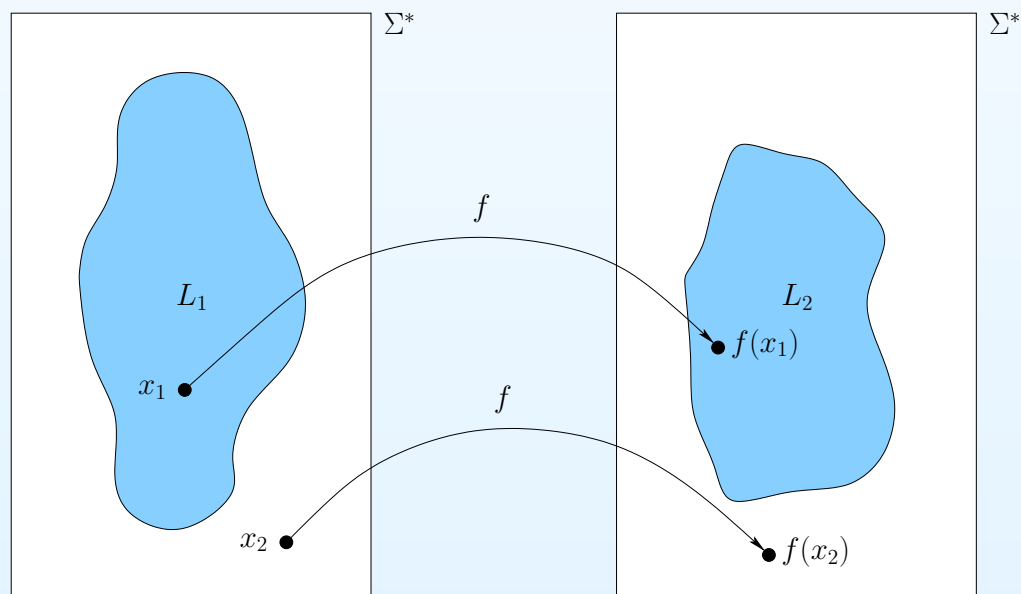
- co-NP is the class of languages  $L$  such that  $\bar{L} \in \text{NP}$ .
- Does  $L \in \text{NP}$  imply  $L \in \text{co-NP}$ ?
- For instance, is  $\text{HAM-CYCLE} \in \text{co-NP}$ ?
- Said differently, is  $\overline{\text{HAM-CYCLE}} \in \text{NP}$ ?
- In words, given a graph, can we easily verify that it does *not* have a simple cycle containing every vertex of  $G$ ?
- What should we use as certificate? Not clear.
- It is open whether  $\text{NP} = \text{co-NP}$ .
- What is known is that  $P \subseteq \text{NP} \cap \text{co-NP}$ .

## NP-complete problems

- There are problems in NP that are “the most difficult” in that class.
- If any one of them can be solved in polynomial time then *every* problem in NP can be solved in polynomial time.
- These difficult problems are called *NP-complete*.
- HAM-CYCLE is NP-complete.
- Hence, if we could show  $\text{HAM-CYCLE} \in P$  then  $P = NP$ .
- We will see examples of several other NP-complete problems.
- To define NP-completeness, we need to first define polynomial-time reducibility.

## Polynomial-time reducibility

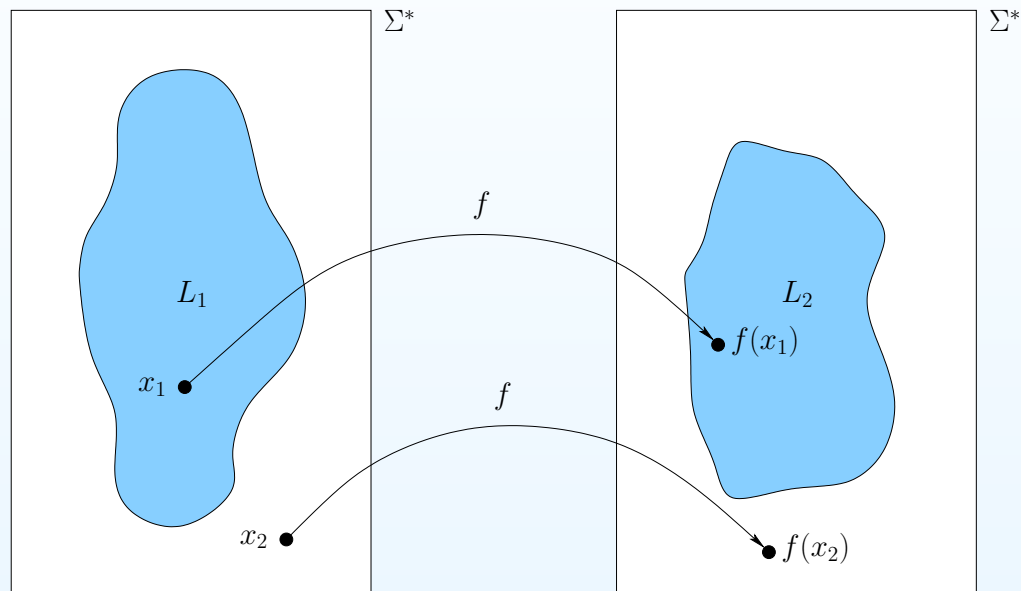
- Language  $L_1$  is polynomial-time *reducible* to language  $L_2$  if there is a polynomial-time computable function  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  such that for all  $x \in \{0, 1\}^*$ ,  
$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$
- In this case, we write  $L_1 \leq_P L_2$ .



- If  $L_1 \leq_P L_2$  then  $L_1$  is in a sense no harder to solve than  $L_2$ .

## Polynomial-time reducibility

- If  $L_1 \leq_P L_2$  then  $L_1$  is in a sense no harder to solve than  $L_2$ .



- More precisely,

$$L_1 \leq_P L_2 \wedge L_2 \in P \Rightarrow L_1 \in P.$$

- This follows since any instance  $x$  of  $L_1$  can be solved by transforming it in polynomial time to an instance  $y = f(x)$  of  $L_2$  and then solving  $y$  with a polynomial-time algorithm for  $L_2$ .

## NP-complete languages

- Language  $L$  is *NP-complete* if
  1.  $L \in \text{NP}$  and
  2.  $L' \leq_P L$  for every  $L' \in \text{NP}$ .
- $L$  is *NP-hard* if property 2 holds (and possibly not property 1).
- The class of NP-complete languages is denoted NPC.
- If some language of NPC belongs to P then  $P = \text{NP}$ . Why?
- It is not immediately clear from the definition that NP-complete languages even exist.
- In practice, why would it be useful to show that a problem is NP-complete?
- We next show that the circuit satisfiability problem is NP-complete.

## An NP-complete problem: Circuit satisfiability

- A *boolean combinational circuit* consists of a collection of logic gates connected together with wires.
- The logic gates allowed are AND, OR, and NOT.
- Each wire has a value which is either 0 or 1.
- Some wires are specified by input values and the rest by the logic gates.
- Other wires specify output values.
- We can represent a circuit as an acyclic graph.

## The circuit satisfiability problem

- Given a boolean combinational circuit  $C$  with one output wire.
- A *satisfying assignment* for  $C$  is an assignment of values to input wires of  $C$  causing an output of 1.
- The *circuit satisfiability problem* CIRCUIT-SAT is to decide if a given circuit has a satisfying assignment:

$$\text{CIRCUIT-SAT} = \{\langle C \rangle \mid C \text{ is a satisfiable boolean combinational circuit}\}.$$

- We will show that CIRCUIT-SAT is NP-complete.

## Showing CIRCUIIT-SAT $\in$ NP

- We construct algorithm  $A$  with inputs  $x$  and  $y$ .
- $A$  checks that  $x$  represents a boolean combinational circuit  $C$  with one output wire and that  $y$  represents an assignment of truth values to the wires of  $C$ .
- If so,  $A$  checks that  $y$  represents a valid truth assignment.
- If so,  $A$  checks that the single output is 1.
- If this is the case,  $A$  returns 1; otherwise it returns 0.
- $A$  is a verification algorithm for CIRCUIIT-SAT and can easily be made to run in polynomial time.
- Thus, CIRCUIIT-SAT  $\in$  NP.



## Showing that CIRCUIT-SAT is NP-hard

- Consider any language  $L \in \text{NP}$ .
- We need to give a polynomial-time reduction from  $L$  to CIRCUIT-SAT.
- In other words, we need to find a polynomial-time algorithm  $A$  computing a function  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  such that

$$x \in L \Leftrightarrow f(x) \in \text{CIRCUIT-SAT}.$$

## Showing that CIRCUIT-SAT is NP-hard

- Since  $L \in \text{NP}$ , there is a polynomial-time algorithm  $A$  such that

$$L = \{x \in \{0, 1\}^* \mid \text{there is a } y \in \{0, 1\}^* \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1\}.$$

- Given string  $x$ ,  $f$  outputs a circuit  $C(x)$  with  $O(|x|^c)$  input wires.
- We ensure that  $C(x)$  has a satisfying assignment of its input wires if and only if  $A(x, y) = 1$  for some  $y$  with  $|y| = O(|x|^c)$ .
- This way,

$$x \in L \Leftrightarrow f(x) = \langle C(x) \rangle \in \text{CIRCUIT-SAT}.$$

- Each  $y$  with  $|y| = O(|x|^c)$  defines an input to  $C(x)$ .
- Intuition: Circuit  $C(x)$  implements algorithm  $A$  on input  $(x, y)$  with  $x$  fixed.
- We ensure that  $A(x, y) = 1$  if and only if  $y$  is a satisfying assignment.

## Showing that CIRCUIT-SAT is NP-hard

- There is a constant  $k$  such that the worst-case running time  $T(n)$  of  $A$  on an input  $(x, y)$  is  $O(n^k)$  where  $n = |x|$ .
- The machine executing  $A$  has a certain *configuration* at each time step.
- The configuration gives a complete specification of the current memory, CPU state, and so on.
- When executing  $A$  on  $(x, y)$ , the machine goes through a series of configurations  $c_0, c_1, \dots, c_{T(n)}$  (assume for simplicity that  $A$  runs for exactly  $T(n)$  steps on  $(x, y)$ ).
- Configuration  $c_0$  specifies inputs  $x$  and  $y$  and the program code for  $A$ .
- One bit of the last configuration  $c_{T(n)}$  specifies the 0/1-output of  $A$ .

## Showing that CIRCUIT-SAT is NP-hard

- Let  $M$  be the circuit implementing the hardware of the machine.
- We feed the initial configuration  $c_0$  as input wires to  $M$ .
- $M$  performs a single step of  $A$  and the new configuration  $c_1$  is stored on output wires.
- These output wires feed into  $M$  which makes another step, forming  $c_2$  as output, and so on.
- In total, we glue  $T(n)$  copies of  $M$  together.
- This gives a BIG circuit representing the entire execution of  $A$  on input  $(x, y)$ .
- The size of the circuit is still polynomial in  $n$ , however.

## Showing that CIRCUIT-SAT is NP-hard

- We modify the circuit by hard-wiring part of the input to that specified by binary string  $x$  and so that the only output wire is that corresponding to the output of  $A$ .
- The circuit now only takes inputs  $y$ .
- The resulting circuit  $C(x)$  has a satisfying assignment  $y$  if and only if  $A(x, y) = 1$ .
- $C(x)$  can be computed from  $x$  in time polynomial in  $|x|$ .
- This shows that  $L \leq_P \text{CIRCUIT-SAT}$ .
- Thus, CIRCUIT-SAT is NP-hard.
- Since also CIRCUIT-SAT  $\in \text{NP}$ , it follows that CIRCUIT-SAT is NP-complete.

## Plan for next lecture

- Showing NP-completeness of other problems using polynomial-time reductions:
  - SAT
  - 3-CNF-SAT
  - CLIQUE
  - VERTEX-COVER
  - (HAM-CYCLE)
  - TSP
  - SUBSET-SUM

## Showing NP-completeness using reductions

- Suppose  $L'$  is an NP-complete language.
- Consider another language  $L$ .
- If  $L' \leq_P L$  then  $L$  is NP-hard. Why?
- If also  $L \in \text{NP}$  then  $L$  is NP-complete.
- Next time we show:

$$\text{CIRCUIT-SAT} \leq_P \text{SAT} \leq_P \text{3-CNF-SAT}$$

$$\leq_P \text{SUBSET-SUM},$$

$$\text{3-CNF-SAT} \leq_P \text{CLIQUE} \leq_P \text{VERTEX-COVER}$$

$$\leq_P \text{HAM-CYCLE} \leq_P \text{TSP}$$

- We also show that all these languages are in NP and hence they are NP-complete.

# **NP-Completeness, part II**

Christian Wulff-Nilsen  
Advanced Algorithms and Data Structures  
DIKU

December 14, 2022



## Overview for today

- NP-completeness and reductions
- NP-completeness of:
  - SAT
  - 3-CNF-SAT
  - CLIQUE
  - VERTEX-COVER
  - (HAM-CYCLE)
  - TSP
  - SUBSET-SUM

## Languages

- *Alphabet*: finite set  $\Sigma$  of symbols.
- *Language*  $L$  over alphabet  $\Sigma$ : a set of strings of symbols from  $\Sigma$ .
- We let  $\Sigma = \{0, 1\}$  so  $L$  is a set of binary strings.
- Example:  $L = \{0, 10, 11001, 11101, \dots\}$ .
- $\Sigma^*$ : set of all binary strings (including  $\epsilon$ ).

## Decision problems and languages

- A *decision problem*  $Q$  consists of yes-instances and no-instances.
- Example,  $Q = \text{HAM-CYCLE}$ :  $\langle G \rangle$  is a yes-instance if  $G$  contains a simple cycle containing all vertices of  $G$ ; otherwise  $\langle G \rangle$  is a no-instance.
- We can view a problem  $Q$  as a mapping of yes-instances to 1 and no-instances to 0.
- We can also view  $Q$  as a language  $L$ :

$$L = \{x \in \{0, 1\}^* \mid Q(x) = 1\}.$$

## Verifying a language

- A *verification algorithm* is an algorithm  $A$  taking two arguments,  $x, y \in \{0, 1\}^*$ , where  $y$  is the *certificate*.
- $A$  *verifies* a string  $x$  if there is a certificate  $y$  such that  $A(x, y) = 1$ .
- The language verified by  $A$  is

$$L = \{x \in \{0, 1\}^* \mid \text{there is a } y \in \{0, 1\}^* \text{ such that } A(x, y) = 1\}.$$

## The complexity class NP

- NP is the class of languages that can be verified in polynomial time.
- In other words,  $L \in \text{NP}$  if and only if there is a polynomial-time verification algorithm  $A$  and a constant  $c$  such that

$$L = \{x \in \{0, 1\}^* \mid \text{there is a } y \in \{0, 1\}^* \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1\}.$$

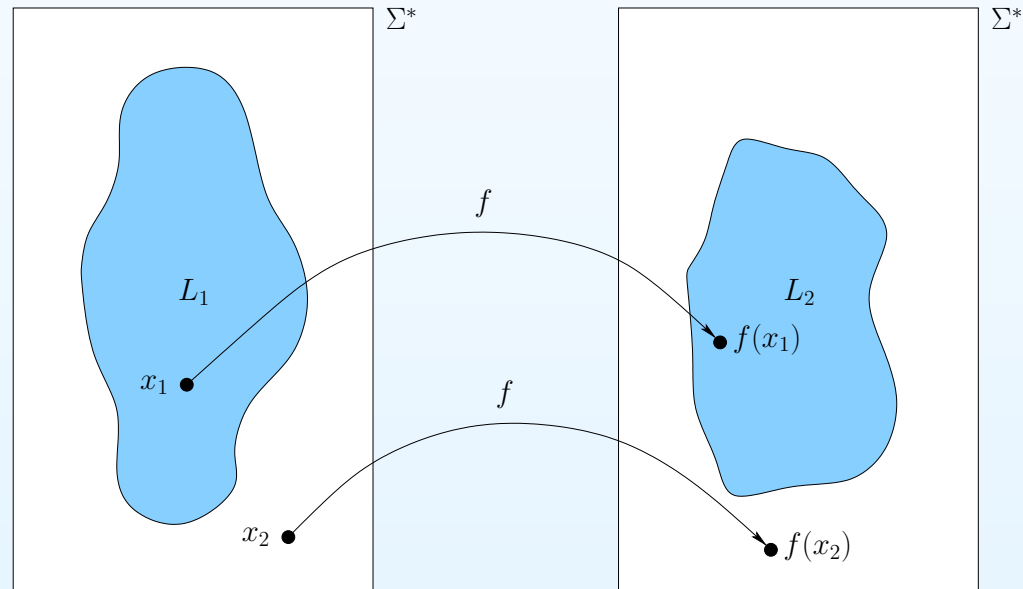
- We saw that  $P \subseteq \text{NP}$ .
- Big open problem: is  $P = \text{NP}$ ?

## Reducibility

- Language  $L_1$  is polynomial-time *reducible* to language  $L_2$  if there is a polynomial-time computable function

$f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  such that for all  $x \in \{0, 1\}^*$ ,

$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$



- We use the notation  $L_1 \leq_P L_2$  for this.
- We saw that

$$L_1 \leq_P L_2 \wedge L_2 \in P \Rightarrow L_1 \in P.$$

## NP-completeness

- Language  $L$  is *NP-complete* if
  1.  $L \in \text{NP}$  and
  2.  $L' \leq_P L$  for every  $L' \in \text{NP}$ .
- $L$  is *NP-hard* if  $L$  satisfies property 2 (and possibly not property 1).
- We saw that if any language of NPC belongs to P then  $P = \text{NP}$ .
- We also showed that CIRCUIT-SAT is NP-complete.

## NP-completeness of other problems via reduction

- Let  $L$  and  $L'$  be two languages with  $L' \in \text{NPC}$ .
- If  $L' \leq_P L$  then  $L$  is NP-hard.
- If in addition  $L \in \text{NP}$  then  $L$  is NP-complete.
- General technique to show NP-completeness of a language  $L$ :
  - Show that  $L \in \text{NP}$ .
  - Pick another language  $L'$  known to be NP-complete (for instance, CIRCUIT-SAT).
  - Show that  $L' \leq_P L$ , i.e., show that there is a polynomial-time computable function  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  such that for all  $x \in \{0, 1\}^*$ ,

$$x \in L' \Leftrightarrow f(x) \in L.$$



## The SAT problem

- A *boolean formula*  $\phi$  consists of boolean variables  $x_1, \dots, x_n$ , boolean connectives  $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$ , and parentheses ( and ).
- Example:  $\phi = (x_1 \vee x_2) \wedge (x_2 \vee x_3 \vee \neg x_4)$ .
- A *satisfying assignment* for a boolean formula  $\phi$  is an assignment of 0/1-values to variables that makes  $\phi$  evaluate to 1.
- $\phi$  is *satisfiable* if there exists a satisfying assignment for  $\phi$ .
- We can now define the problem SAT:

$$\text{SAT} = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable boolean formula} \}.$$

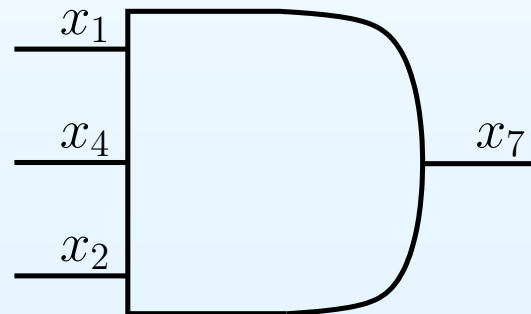
- We will show that SAT is NP-complete.

## Showing that SAT is NP-complete

- To show  $\text{SAT} \in \text{NPC}$ , we follow our recipe:
  - Show that  $\text{SAT} \in \text{NP}$ .
  - Show that  $\text{CIRCUIT-SAT} \leq_P \text{SAT}$ .
- To show that  $\text{SAT} \in \text{NP}$ , we construct a verification algorithm  $A$  taking inputs  $x$  and  $y$ .
- It regards  $x$  as a boolean formula  $\phi$  and  $y$  as an assignment of values to variables of  $\phi$ .
- $A$  returns 1 if  $y$  defines a satisfying assignment for  $\phi$ ; otherwise,  $A$  returns 0.
- We can easily make  $A$  run in polynomial time.
- Thus,  $\text{SAT} \in \text{NP}$ .

## Showing $\text{CIRCUIT-SAT} \leq_P \text{SAT}$

- Given a circuit  $C$ , we transform it into a boolean function  $\phi$  as follows.
- Associate a variable  $x_i$  with each wire of  $C$ ; let  $x_m$  be the output wire variable.
- We can view each gate of  $C$  as a function mapping the values on its input wires to the value on its output wire.
- Construct a sub-formula for each such function.
- Example:



Sub-formula for gate:  $x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)$

## Showing CIRCUI-T-SAT $\leq_P$ SAT

- Example (see blackboard/Figure 34.10 in CLRS):

$$\phi_1 = (x_4 \leftrightarrow \neg x_3)$$

$$\phi_2 = (x_5 \leftrightarrow (x_1 \vee x_2))$$

$$\phi_3 = (x_6 \leftrightarrow \neg x_4)$$

$$\phi_4 = (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))$$

$$\phi_5 = (x_8 \leftrightarrow (x_5 \vee x_6))$$

$$\phi_6 = (x_9 \leftrightarrow (x_6 \vee x_7))$$

$$\phi_7 = (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)).$$

## Showing $\text{CIRCUIT-SAT} \leq_P \text{SAT}$

- If  $\phi_1, \dots, \phi_k$  are the sub-formulas, we define  $\phi$  to be  $x_m \wedge \phi_1 \wedge \phi_2 \wedge \dots \wedge \phi_k$ .

- Example:

$$\begin{aligned}\phi = & x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\ & \wedge (x_6 \leftrightarrow \neg x_4) \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\ & \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\ & \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)).\end{aligned}$$

- $\phi$  can be constructed in polynomial time.
- In words,  $\phi$  is stating that the output wire is 1 and that each gate behaves as it is supposed to.
- Thus,  $C$  is satisfiable if and only if  $\phi$  is satisfiable:

$$\langle C \rangle \in \text{CIRCUIT-SAT} \Leftrightarrow \langle \phi \rangle \in \text{SAT}.$$

- We have now shown that SAT is NP-complete.

## 3-CNF formulas

- Let  $\phi$  be a boolean formula.
- A *literal* in  $\phi$  is an occurrence of a variable or its negation.
- Suppose  $\phi$  is the AND of sub-formulas, called *clauses*.
- Furthermore, suppose that each clause is the OR of exactly 3 distinct literals.
- Then we say that  $\phi$  is in *3-conjunctive normal form*, or *3-CNF*.
- Example:

$$\phi = (x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4).$$

## NP-completeness of 3-CNF-SAT

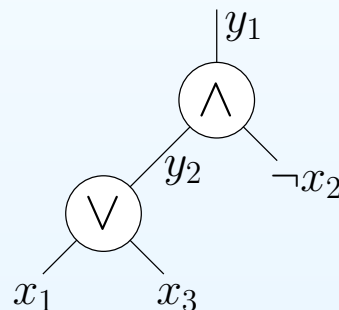
- We define the language 3-CNF-SAT by

$$3\text{-CNF-SAT} = \{\langle \phi \rangle \mid \phi \text{ is in 3-CNF and satisfiable}\}.$$

- We will show that  $3\text{-CNF-SAT} \in \text{NPC}$  in two steps:
  - $3\text{-CNF-SAT} \in \text{NP}$ ,
  - $\text{SAT} \leq_P 3\text{-CNF-SAT}$ .
- Showing  $3\text{-CNF-SAT} \in \text{NP}$  is done using the same argument as for SAT.

## NP-completeness of 3-CNF-SAT

- Remains to show  $\text{SAT} \leq_P \text{3-CNF-SAT}$ .
- Let  $\langle \phi \rangle$  be an instance of SAT.
- We construct a *parse tree* for  $\phi$  where leaves are literals and internal nodes are connectives.
- Parse tree for  $\phi = (x_1 \vee x_3) \wedge \neg x_2$ :

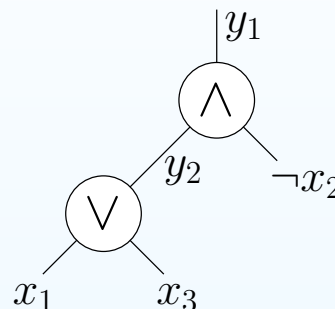


- We may assume that each node in the parse tree has at most two children. (why?)



## NP-completeness of 3-CNF-SAT

- Parse tree for  $\phi = (x_1 \vee x_3) \wedge \neg x_2$ :



- Regard the tree as a circuit with internal nodes as gates.
- We can construct formulas  $\phi'_1, \dots, \phi'_k$  for each of these gates, as before.
- Let  $\phi' = y_1 \wedge \phi'_1 \wedge \phi'_2 \wedge \dots \wedge \phi'_k$ , where  $y_1$  is the output wire.
- $\phi'$  is satisfiable iff  $\phi$  is satisfiable.
- Each clause of  $\phi'$  has at most 3 literals.

## NP-completeness of 3-CNF-SAT

- Consider one clause  $\phi'_i$ .
- Example:  $\phi'_i = y_1 \leftrightarrow (y_2 \wedge \neg x_2)$ .

$y_1$	$y_2$	$x_2$	$\phi'_i$	$\phi''_i \equiv \neg \phi'_i$
0	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	0	1

$$\neg \phi'_i \equiv \phi''_i = (\neg y_1 \wedge y_2 \wedge \neg x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge y_2 \wedge x_2).$$

## NP-completeness of 3-CNF-SAT

- $\phi_i''$  is in *disjunctive normal form*:

$$\begin{aligned}\phi_i'' = & (\neg y_1 \wedge y_2 \wedge \neg x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \\ & \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge y_2 \wedge x_2).\end{aligned}$$

- Negating and applying De Morgan's laws converts  $\phi_i''$  into conjunctive normal form:

$$\begin{aligned}\neg \phi_i'' \equiv & (y_1 \vee \neg y_2 \vee x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \\ & \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee \neg y_2 \vee \neg x_2).\end{aligned}$$

- Recall that  $\phi_i'' \equiv \neg \phi_i'$ .
- It follows that  $\neg \phi_i'' \equiv \neg(\neg \phi_i') \equiv \phi_i'$ .
- In words,  $\neg \phi_i''$  is equivalent to the original  $\phi_i'$ .

## NP-completeness of 3-CNF-SAT

- Applying this technique to each clause converts  $\phi'$  into an equivalent formula which is almost in 3-CNF:
  - each clause has *at most* 3 literals.
- We can extend this to *exactly* 3 (distinct) literals by introducing dummy variables  $p$  and  $q$ :

$$\ell_1 \vee \ell_2 \equiv (\ell_1 \vee \ell_2 \vee p) \wedge (\ell_1 \vee \ell_2 \vee \neg p)$$

$$\ell \equiv (\ell \vee p \vee q) \wedge (\ell \vee p \vee \neg q) \wedge (\ell \vee \neg p \vee q) \wedge (\ell \vee \neg p \vee \neg q)$$

- In polynomial time, we convert  $\phi'$  into a formula  $\phi'''$  in 3-CNF.
- We have

$$\langle \phi \rangle \in \text{SAT} \Leftrightarrow \langle \phi''' \rangle \in \text{3-CNF-SAT}.$$

- Thus,  $\text{SAT} \leq_P \text{3-CNF-SAT}$  so 3-CNF-SAT is NP-complete.

## The subset-sum problem

- Given a set  $S$  of positive integers and given integer target  $t > 0$ .
- Is there a subset  $S'$  of  $S$  summing to  $t$ ?
- As a language:

$$\text{SUBSET-SUM} = \{ \langle S, t \rangle \mid \exists S' \subseteq S \text{ so that } t = \sum_{s \in S'} s \}.$$

- We will show that SUBSET-SUM is NP-complete.
- Clearly, SUBSET-SUM  $\in$  NP.
- To show NP-hardness, we reduce from 3-CNF-SAT:

$$3\text{-CNF-SAT} \leq_P \text{SUBSET-SUM}.$$

## Showing $3\text{-CNF-SAT} \leq_P \text{SUBSET-SUM}$

- Consider a 3-CNF-formula  $\phi$  with  $n$  variables  $x_1, \dots, x_n$  and  $k$  clauses  $C_1, \dots, C_k$ .
- We will construct an instance  $\langle S, t \rangle$  such that:

$$\langle \phi \rangle \in 3\text{-CNF-SAT} \Leftrightarrow \langle S, t \rangle \in \text{SUBSET-SUM}.$$

- In other words, we want that  $\phi$  is satisfiable if and only if  $S$  has a subset summing to  $t$ .

## Constructing $S$ and $t$ (see blackboard/Figure 34.19 in CLRS)

- For each variable  $x_i$ , create two decimal numbers  $v_i$  and  $v'_i$ .
- For each clause  $C_j$ , create two decimal numbers  $s_j$  and  $s'_j$ .
- Finally, create a decimal number  $t$ .
- Each number has  $n + k$  digits.
- The  $n$  most significant digits are associated with  $x_1, \dots, x_n$ .
- The  $k$  least significant digits are associated with  $C_1, \dots, C_k$ .
- $S$  consists of numbers  $v_1, v'_1, v_2, v'_2, \dots, v_n, v'_n$  and  $s_1, s'_1, s_2, s'_2, \dots, s_k, s'_k$ .

## Constructing $S$ and $t$ (see blackboard/Figure 34.19 in CLRS)

- Digit  $x_i$  of  $v_i$  and digit  $x_i$  of  $v'_i$  is 1.
- Digit  $C_j$  of  $v_i$  is 1 if  $x_i \in C_j$ .
- Digit  $C_j$  of  $v'_i$  is 1 if  $\neg x_i \in C_j$ .
- Digit  $C_i$  of  $s_i$  is 1 and digit  $C_i$  of  $s'_i$  is 2.
- Target  $t$  is defined to be:

$$t = \overbrace{11 \dots 1}^n \overbrace{44 \dots 4}^k.$$

- All digits not specified above are 0.
- Simplifying assumptions: no clause contains both a variable and its negation and each variable appears somewhere. This ensures unique numbers (otherwise,  $S$  might be a multiset).



## Showing $\langle \phi \rangle \in 3\text{-CNF-SAT} \Rightarrow \langle S, t \rangle \in \text{SUBSET-SUM}$

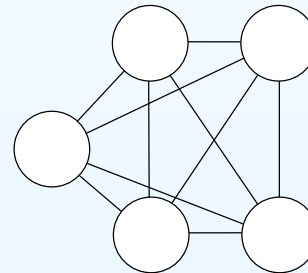
- Assume  $\langle \phi \rangle \in 3\text{-CNF-SAT}$ .
- In other words, assume that  $\phi$  is in 3-CNF and satisfiable.
- We will find a subset  $S'$  of  $S$  with  $\sum_{s \in S'} s = t$ .
- Assign values to  $x_1, \dots, x_n$  that satisfy  $\phi$ .
- For  $i = 1, \dots, n$ , if  $x_i = 1$  then include  $v_i$  in  $S'$ ; otherwise include  $v'_i$ .
- Include additional numbers from  $\{s_1, s'_1, s_2, s'_2, \dots, s_k, s'_k\}$  to reach target  $t$ .
- Why is this possible?

## **Showing** $\langle S, t \rangle \in \text{SUBSET-SUM} \Rightarrow \langle \phi \rangle \in \text{3-CNF-SAT}$

- Let  $S'$  be a subset of  $S$  summing to  $t$ .
- We need to find a satisfying assignment for  $\phi$ .
- We set  $x_i$  to 1 if and only if  $v_i \in S'$ .
- Why is this a satisfying assignment?

## The CLIQUE-problem

- Let  $G = (V, E)$  be an undirected graph.
- A *clique* in  $G$  is a subset  $V' \subseteq V$  such that  $(u, v) \in E$  for all distinct  $u, v \in V'$ .
- The size of the clique is  $|V'|$ .
- Example of a clique of size 5:



- The CLIQUE problem is the problem of determining if  $G$  contains a clique of a given size  $k$ :

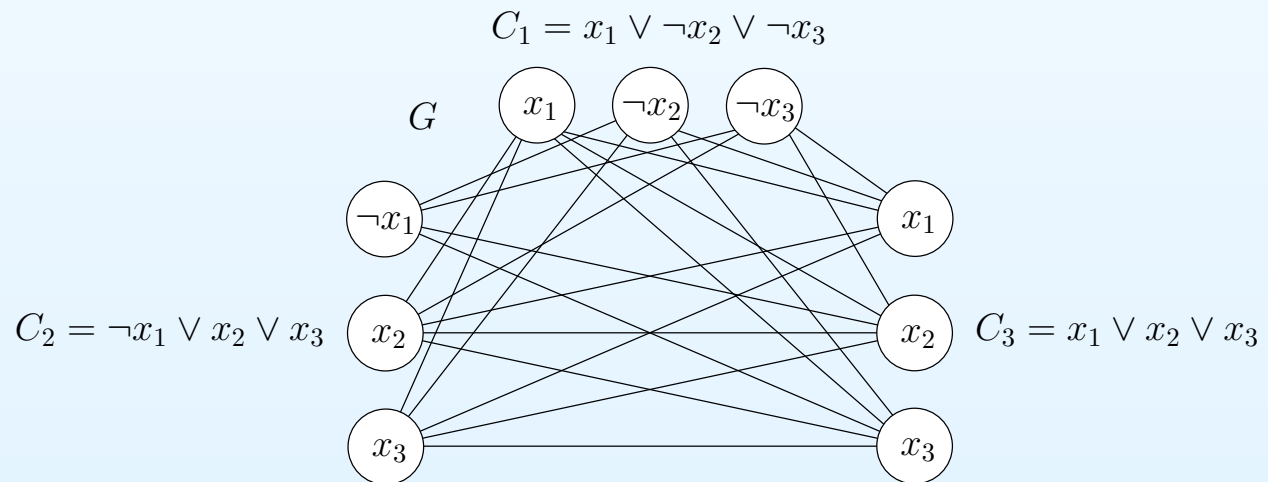
$$\text{CLIQUE} = \{ \langle G, k \rangle \mid G \text{ is a graph containing a clique of size } k \}.$$

## NP-completeness of CLIQUE

- We will show that CLIQUE is NP-complete as follows:
  - $\text{CLIQUE} \in \text{NP}$ ,
  - $3\text{-CNF-SAT} \leq_P \text{CLIQUE}$ .
- To show  $\text{CLIQUE} \in \text{NP}$ , consider an algorithm  $A$  taking two inputs,  $\langle G, k \rangle$  and a certificate  $y$ .
- $y$  specifies a subset  $V'$  of vertices of  $G$ .
- $A$  checks that  $|V'| = k$  and that  $V'$  is a clique in  $G$ .
- This can easily be done in polynomial time.
- Thus,  $\text{CLIQUE} \in \text{NP}$ .

## Showing $3\text{-CNF-SAT} \leq_P \text{CLIQUE}$

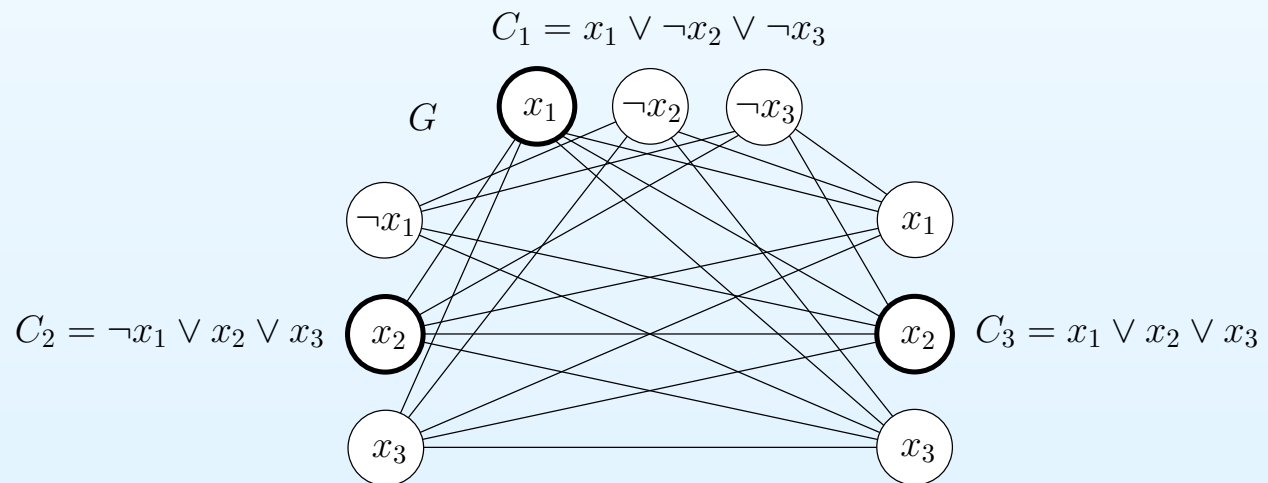
- Given a formula  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_k$  in 3-CNF.
- We will construct a graph  $G$  such that  $\phi$  is satisfiable if and only if  $G$  has a clique of size  $k$ .
- For each  $C_r = \ell_1^r \vee \ell_2^r \vee \ell_3^r$ , we include three vertices  $v_1^r, v_2^r$ , and  $v_3^r$  to  $G$ .
- There is an edge  $(v_i^r, v_j^s)$  in  $G$  if and only if  $r \neq s$  and  $\ell_i^r$  is not the negation of  $\ell_j^s$ .
- Example with  $\phi = C_1 \wedge C_2 \wedge C_3$ :



- $G$  can be constructed in polynomial time from  $\phi$ .
- We will show  $\langle \phi \rangle \in 3\text{-CNF-SAT} \Leftrightarrow \langle G, k \rangle \in \text{CLIQUE}$ .

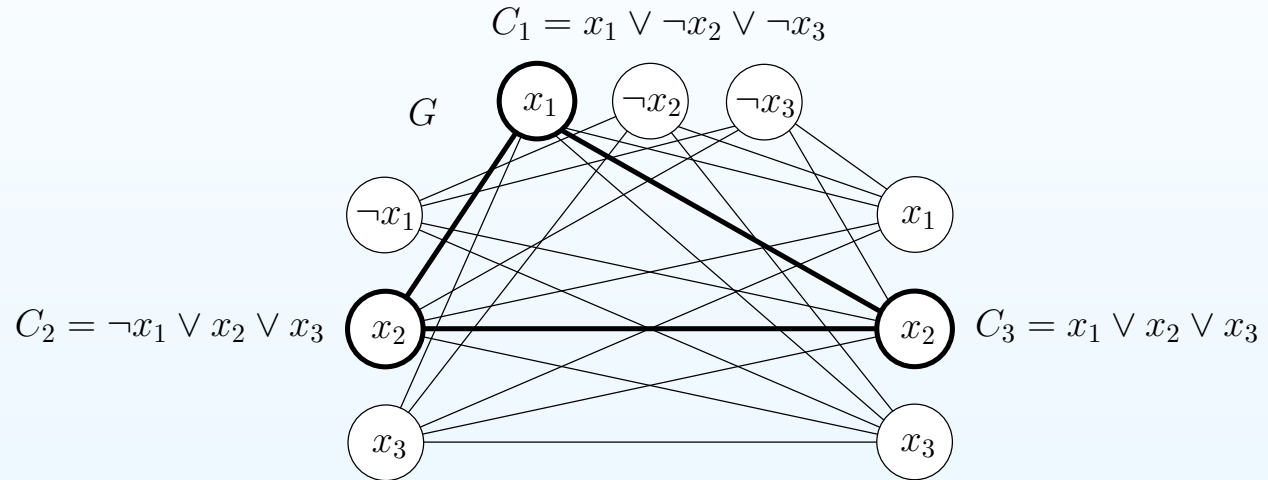
## Showing $3\text{-CNF-SAT} \leq_P \text{CLIQUE}$

- Need to show that  $\phi$  is satisfiable if and only if  $G$  has a clique of size  $k$ .
- Assume first that  $\langle \phi \rangle \in 3\text{-CNF-SAT}$ .
- In a satisfying assignment for  $\phi$ , each clause  $C_i$  of  $\phi$  has at least one true literal.
- Pick the corresponding vertex in  $G$ .
- This gives a total of  $k$  vertices.
- Example with satisfying assignment  $x_1 = 1, x_2 = 1, x_3 = 0$ :



## Showing $3\text{-CNF-SAT} \leq_P \text{CLIQUE}$

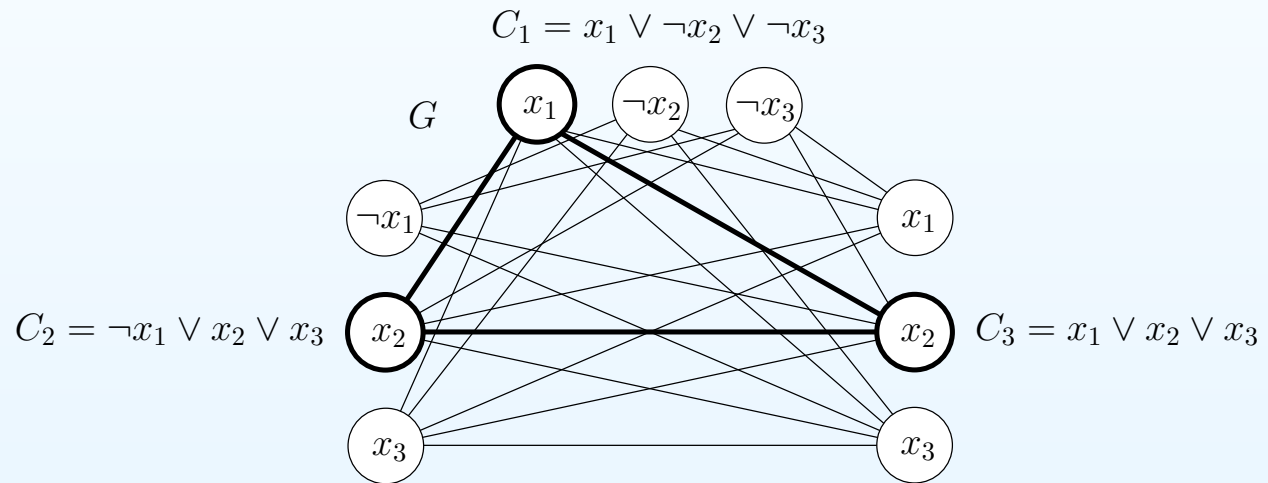
- Why is this a clique in  $G$  of size  $k$ ?



- There must be an edge between each pair of these vertices since no picked literal can be the negation of another picked literal (we only picked true literals).
- Hence,  $G$  has a clique of size  $k$  so  $\langle G, k \rangle \in \text{CLIQUE}$ .

## Showing $3\text{-CNF-SAT} \leq_P \text{CLIQUE}$

- It remains to show that if  $G$  has a clique of size  $k$  then  $\phi$  is satisfiable.
- Let  $V'$  be a clique of  $G$  of size  $k$ .

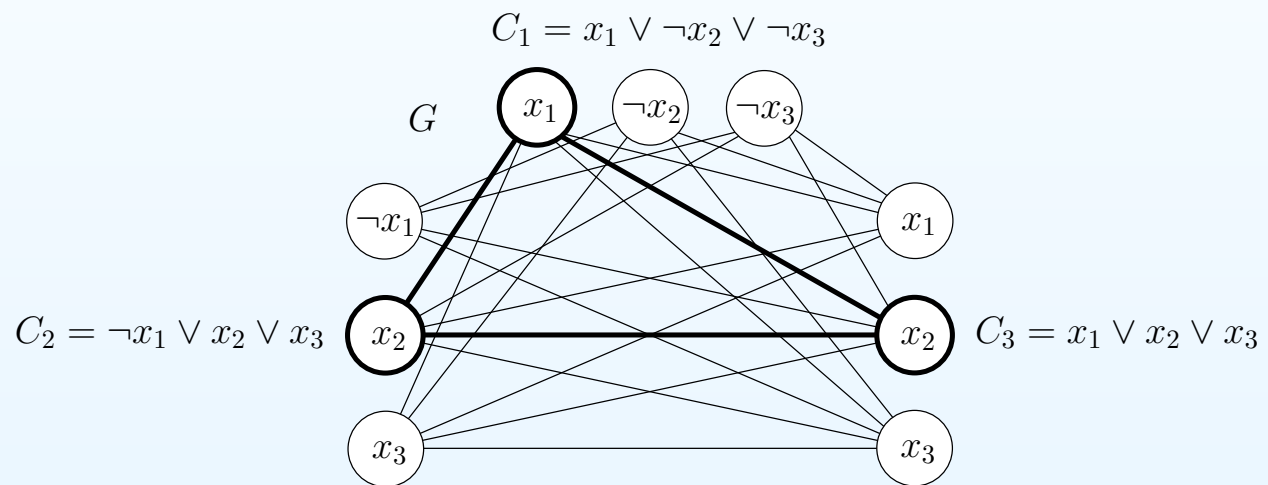


- Each vertex triple of  $G$  has exactly one vertex in  $V'$ .
- We assign 1 to the literal of  $\phi$  corresponding to that vertex.
- In the example above,  $x_1 = 1$  and  $x_2 = 1$ .



## Showing $3\text{-CNF-SAT} \leq_P \text{CLIQUE}$

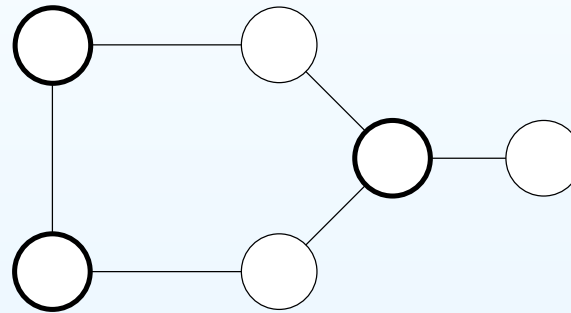
- It remains to show that if  $G$  has a clique of size  $k$  then  $\phi$  is satisfiable.
- Let  $V'$  be a clique of  $G$  of size  $k$ .



- No variable of  $\phi$  is assigned both 0 and 1 in this way since there is no edge between a variable and its negation in  $G$ .
- This gives a legal assignment of values to some of the variables.
- This assignment satisfies  $\phi$  as it makes each clause true.
- What about variables that have not been assigned a value? ( $x_3$  in the example) Pick an arbitrary value for each of them.

## The VERTEX-COVER problem

- A *vertex cover* of  $G = (V, E)$  is a subset  $V' \subseteq V$  such that every edge of  $E$  has at least one endpoint in  $V'$ .
- Vertex cover problem: find a minimum-size vertex cover of  $G$ .



- Restating as a decision problem: does  $G$  have a vertex cover of a given size  $k$ ?
- Stated as a language:

$$\text{VERTEX-COVER} = \{ \langle G, k \rangle \mid G \text{ has a vertex cover of size } k \}.$$

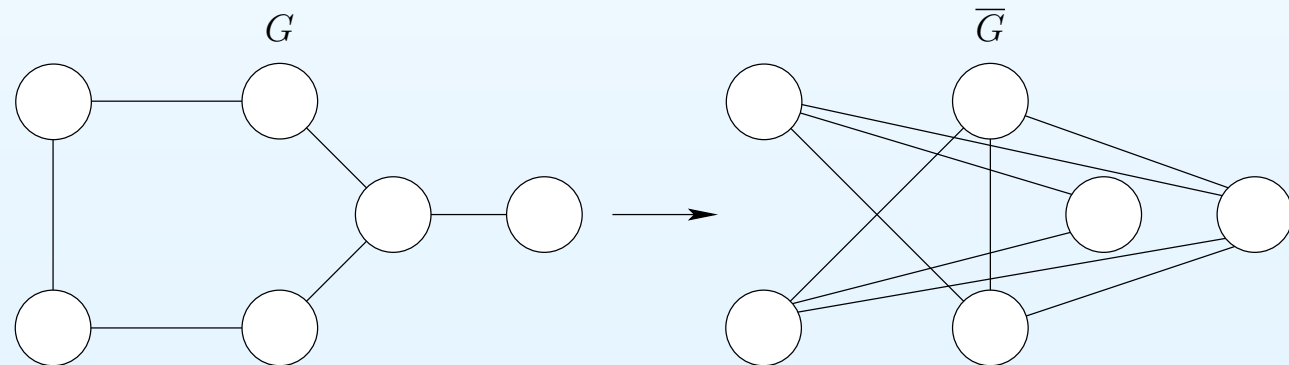
- We will show that VERTEX-COVER is NP-complete.

## Showing that VERTEX-COVER $\in$ NP

- The verification algorithm takes an instance  $\langle G, k \rangle$  and a certificate denoting a subset  $V'$  of vertices of  $G$ .
- It checks that  $V'$  has size  $k$  and that every edge of  $G$  is incident to at least one vertex of  $V'$ .
- This can easily be done in polynomial time.
- Hence, VERTEX-COVER  $\in$  NP.

## Showing that VERTEX-COVER is NP-hard

- We show  $\text{CLIQUE} \leq_P \text{VERTEX-COVER}$ .
- Given an instance  $\langle G, k \rangle$  of the clique problem.
- Let  $n$  denote the number of vertices of  $G$ .
- We transform  $\langle G, k \rangle$  in polynomial time to the instance  $\langle \overline{G}, n - k \rangle$  of the vertex cover problem.
- Here,  $\overline{G}$  is the *complement* of  $G$  which has the same vertex set as  $G$  and has an edge between two vertices  $u$  and  $v$  if and only if there is no edge between  $u$  and  $v$  in  $G$ .

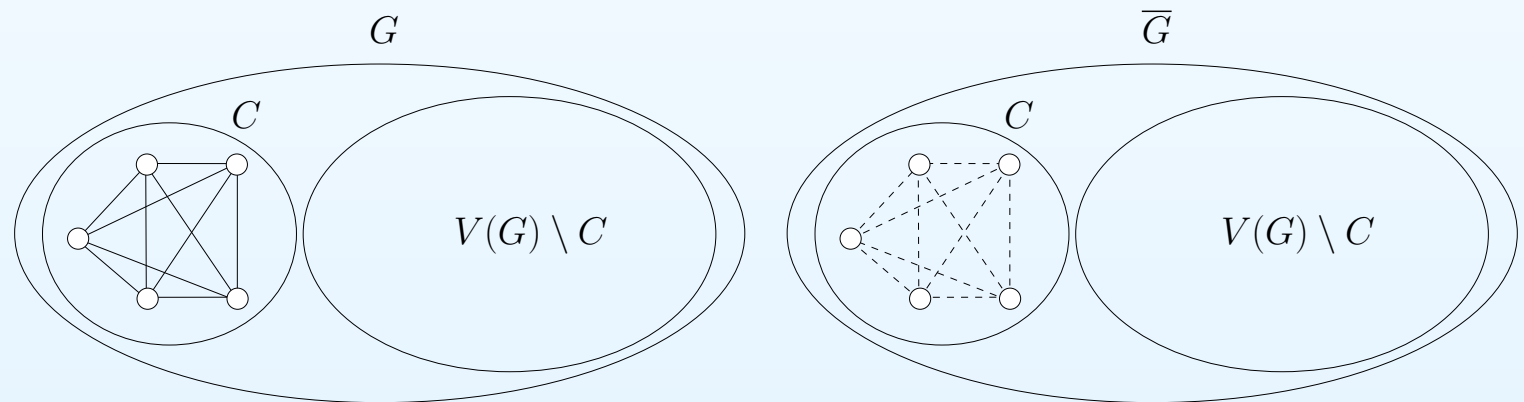


## Showing that VERTEX-COVER is NP-hard

- Need to show:

$$\langle G, k \rangle \in \text{CLIQUE} \Leftrightarrow \langle \overline{G}, n - k \rangle \in \text{VERTEX-COVER}.$$

- Observe that any vertex set  $C$  is a clique in  $G$  if and only if  $C$  contains no edges in  $\overline{G}$ .



- Hence,  $C$  is a clique of size  $k$  in  $G$  if and only if  $V(G) \setminus C$  is a vertex cover of size  $n - k$  in  $\overline{G}$ .
- We have shown that VERTEX-COVER is NP-hard.

## The travelling-salesman problem

- Let  $G = (V, E)$  be a complete undirected graph and let  $c$  be a non-negative integer cost function on the edge set of  $G$ .
- A *tour* of  $G$  is a Hamilton cycle of  $G$ .
- The travelling salesman problem is that of finding a minimum-cost tour of  $G$ .
- Stated as a decision problem/language:

$$\text{TSP} = \{ \langle G, c, k \rangle \mid G \text{ has a tour of cost at most } k \}.$$

- Clearly,  $\text{TSP} \in \text{NP}$ .
- Since HAM-CYCLE is NP-complete, we show that TSP is NP-hard by:

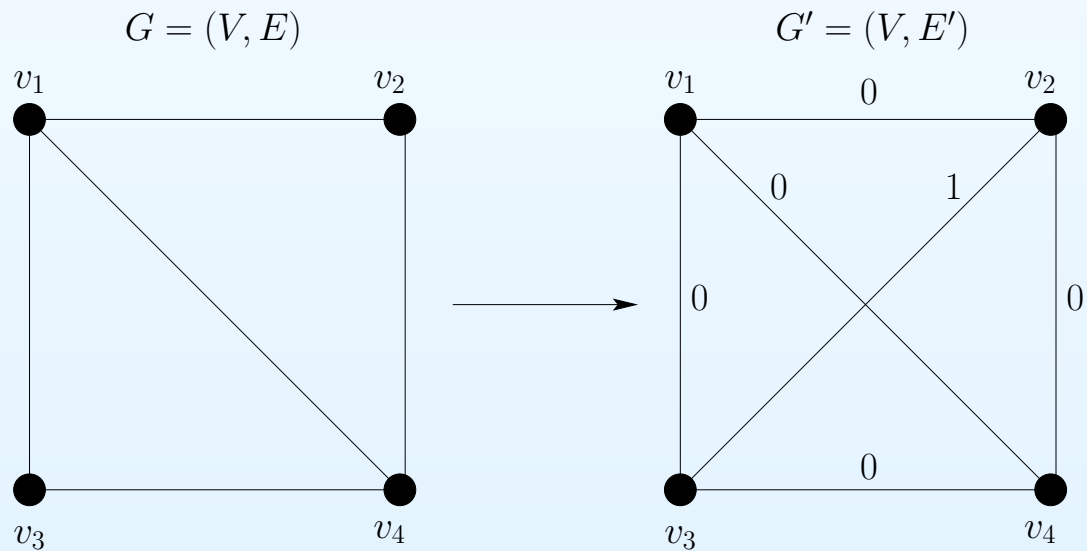
$$\text{HAM-CYCLE} \leq_P \text{TSP}.$$

- This will imply that TSP is NP-complete.

## HAM-CYCLE $\leq_P$ TSP

- Let  $G = (V, E)$  be an instance of the Hamilton-cycle problem.
- We construct a complete graph  $G' = (V, E')$  on vertex set  $V$ .
- Define a cost function  $c$  on  $E'$  by

$$c(u, v) = \begin{cases} 0 & \text{if } (u, v) \in E, \\ 1 & \text{if } (u, v) \notin E. \end{cases}$$



- Show that:  $\langle G \rangle \in \text{HAM-CYCLE} \Leftrightarrow \langle G', c, 0 \rangle \in \text{TSP}$ .

# Advanced algorithms and data structures

## Exact exponential algorithms and parameterized complexity

Jacob Holm ([jaho@di.ku.dk](mailto:jaho@di.ku.dk))

December 16th 2024



# Today's Lecture

## Exact exponential algorithms and parameterized complexity

### Introduction

### Exact exponential algorithms

- Exact TSP via Dynamic Programming

- Dynamic Programming in general

- Exact MIS via Branching

### Parameterized problems

- "Bar fight prevention" aka  $k$ -Vertex Cover

- Kernelization

- Bounded search tree

### FPT vs XP

- Example: Vertex  $k$ -Coloring

- Example:  $k$ -Clique

- Example: Clique parameterized by  $\Delta$

### Summary

# Introduction

We usually want algorithms that

- 1) in polynomial time,
- 2) for all inputs,
- 3) find an exact solution.

Unfortunately some problems are hard, and we may have to settle for (at best) 2 out of 3. We call such algorithms

**Exact exponential algorithms**

if we relax 1) to allow using exponential time.

**Parameterized algorithms**

if we relax 2) to only inputs with small fixed values of some parameter.

**Approximation algorithms**

if we relax 3) to allow approximate solutions (final 2 lectures).

## AADS Lecture on EE/FPT, Part 2

### Exact exponential algorithms

# Exact exponential algorithms

Recall that a *decision problem* is in NP if and only if there exists:

- ▶ A polynomial-time verifier  $R(x, y)$ ; and
- ▶ a function  $m(x) \in \mathcal{O}(\text{poly}|x|)$ ; such that
- ▶ for every problem instance  $x$ :  $x$  is a yes-instance if and only if there exists a certificate  $y$  of size  $|y| \leq m(x)$  such that  $R(x, y)$  is true.

**Note:** A certificate  $y$  is proof that  $x$  “has a solution”, but  $y$  does not have to *be* a solution. However, a solution is often the most natural certificate.

**Note:** Every optimization problem has a decision version. What is it?

Instead of asking for the “best” value  $z$  with some property, ask whether a value  $z$  given as part of the input has that property.

Every problem in NP has a simple brute-force algorithm of the following form: Given problem instance  $x$ , try all potential certificates  $y$  with  $|y| \leq m(x)$  and check if  $R(x, y)$  for any of them.

Since a potential certificate is just a bit string of length at most  $m(x)$  there are at most  $2^{m(x)}$  potential certificates to check, and each check takes  $\mathcal{O}(\text{poly}|x|)$  time. Thus, if we assume  $m(x)$  can be computed in  $\mathcal{O}(\text{poly}|x|)$  time, the brute force running time is  $\mathcal{O}(2^{m(x)} \cdot \text{poly}|x|)$ .

## Notation: $\mathcal{O}^*(\cdot)$

For any fixed  $0 < a < b$ , and  $c, d \in \mathbb{R}$ , we have  $\mathcal{O}(a^n \cdot n^c) \subset \mathcal{O}(b^n \cdot n^d)$ .

So when comparing exact exponential algorithms, the bases of the exponentials are much more important than the polynomial factors.

Define

$$f(n) \in \mathcal{O}^*(g(n)) \iff \exists c \in \mathbb{R} : f(n) \in \mathcal{O}(n^c \cdot g(n))$$

In other words,  $\mathcal{O}^*(\cdot)$  is the same as  $\mathcal{O}(\cdot)$  but ignores polynomial factors.

Notice that for all  $0 < a < b$ :  $\mathcal{O}(a^n) \subset \mathcal{O}^*(a^n) \subset \mathcal{O}(b^n)$ .

Using this notation, what is the running time for the simple brute-force algorithm?  $\mathcal{O}^*(2^{m(x)})$

# Size of a problem

What do we mean by the “size”  $n$  of a problem? Typically:

$n$ , or  $m + n$  for graphs with  $n$  vertices and  $m$  edges.

$|S|$  for problems involving some set  $S$ .

$\#variables$  for SAT-type problems.

This measure of “size” is usually sufficient to describe the running time of the natural brute-force algorithm and to show improvements in better algorithms.

Problem	certificate size	brute-force time	this lecture
SAT, MIS	$m(x) = n$	$T(n) \in \mathcal{O}^*(2^n)$	$\mathcal{O}^*(1.45^n)$
TSP	$m(x) = \log_2(n!)$	$T(n) \in \mathcal{O}^*(n!)$	$\mathcal{O}^*(2^n)$
$k$ -Vertex Cover	$m(x) = k \log_2(n)$	$T(n) \in \mathcal{O}(n^k \cdot \text{poly} x )$	$\mathcal{O}_k(m + n)$
Vertex $k$ -coloring	$m(x) = \log_2(k^n)$	$T(n) \in \mathcal{O}^*(k^n)$	?

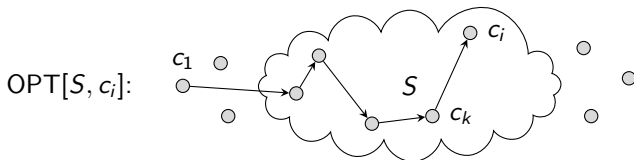
# TSP via Dynamic Programming (Bellman-Held-Karp)

**Problem:** Given cities  $c_1, \dots, c_n$ , and distances  $d_{ij} = d(c_i, c_j)$ , find tour of minimal length, visiting all cities exactly once. Equivalently, find permutation  $\pi$  minimizing  $d(c_{\pi(n)}, c_{\pi(1)}) + \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$ .

**Idea:** For all  $S \subseteq \{c_2, \dots, c_n\}$  and  $c_i \in S$  define

$\text{OPT}[S, c_i] :=$  minimum length of any path that starts in  $c_1$ , visits all of  $S$  once without leaving  $S$ , and ends in  $c_i$ .

Then  $\min\{\text{OPT}[\{c_2, \dots, c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \dots, c_n\}\}$  is the length of the minimal tour.



## Lemma

$$\text{OPT}[S, c_i] = \begin{cases} d(c_1, c_i) & \text{if } S = \{c_i\} \\ \min\{\text{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \mid c_k \in S \setminus \{c_i\}\} & \text{if } \{c_i\} \subset S \end{cases}$$

## Proof.

Let  $e = (c_k, c_i)$  be the last edge on such a path. If  $k = 1$  we are done. If  $k \neq 1$  the shortest length through  $e$  must be  $\text{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i)$ . The shortest such path must use the minimum over all  $c_k \in S \setminus \{c_i\}$ .  $\square$

# TSP via Dynamic Programming (Bellman-Held-Karp)

We can solve TSP by computing all  $\text{OPT}[S, c_i]$  values in order of increasing size of  $S$ .

```
1: function TSP( $\{c_1, \dots, c_n\}, d$ )
2:   for  $i \leftarrow 2 \dots n$  do
3:      $\text{OPT}[\{c_i\}, c_i] \leftarrow d(c_1, c_i)$ 
4:   for  $j \leftarrow 2 \dots n-1$  do
5:     for  $S \subseteq \{c_2, \dots, c_n\}$  with  $|S| = j$  do
6:       for  $c_i \in S$  do
7:          $\text{OPT}[S, c_i] \leftarrow \min\{\text{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \mid c_k \in S \setminus \{c_i\}\}$ 
8:   return  $\min\{\text{OPT}[\{c_2, \dots, c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \dots, c_n\}\}$ 
```

## Lemma

*The above procedure solves TSP by computing  $\mathcal{O}(n^2 \cdot 2^n)$  shortest paths.*

## Proof.

The number of path lengths computed in line 7 is

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \sum_{i=1}^j (j-1) \leq n^2 \sum_{j=0}^n \binom{n}{j} = n^2 \cdot 2^n$$

And lines 3 and 8 compute only  $n-1$  more path lengths each. □

What is the running time of the algorithm?  $\mathcal{O}^*(2^n)$  if we assume additions take at most polynomial time in  $n$ . Much better than  $\mathcal{O}^*(n!)$ .



# Dynamic Programming in general

Similar to “Divide and Conquer” in that it requires “Optimal Substructure” but subproblems may be overlapping.

Instead of recursively solving smaller disjoint subproblems, “Dynamic Programming” solves all smaller subproblems in order of increasing size.

A hybrid idea called “Memoization” (not “Memorization”!) does the same by using recursion, but caching results so each subproblem is only solved once.

In our TSP example the original problem does not have the optimal substructure property (A piece of an optimal tour does not have to be an optimal tour of some subgraph). *The trick is to notice that the problem of computing  $\text{OPT}[\{c_2, \dots, c_n\}, c_i]$  does have the property, and that TSP can be solved once we know that for all  $c_i$ .*

## Exact MIS via Branching

**Problem:** Given undirected graph  $(V, E)$ , find the maximum cardinality of  $I \subseteq V$  so each edge has at most one endpoint in  $I$ .

Such a set  $I$  is called a *Maximum Independent Set (MIS)* for the graph.

**Naive:** Try all  $2^n$  subsets (where  $n = |V|$ ). This takes  $\mathcal{O}^*(2^n)$  time.

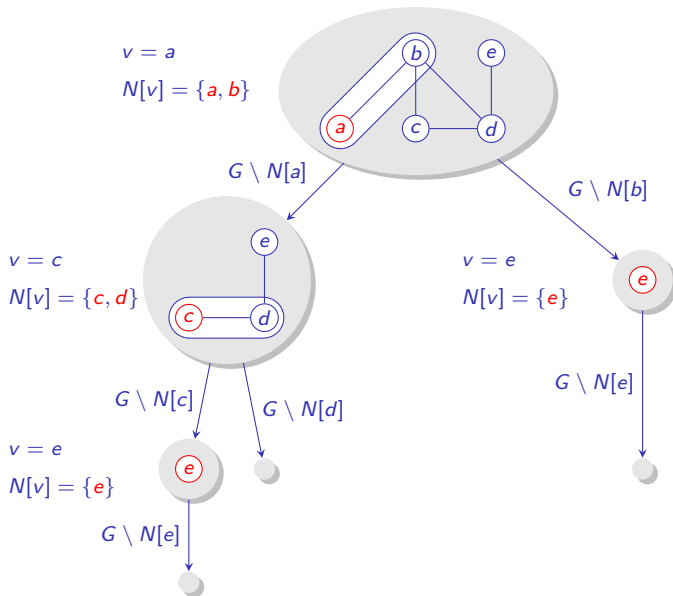
For  $v \in V$  define  $N[v] := \{v\} \cup \{w \in V \mid (v, w) \in E\}$ . This is called the *closed neighborhood* of  $v$ .

**Observation:**  $N[v] \cap I \neq \emptyset$  for all  $v \in V$  and all MIS  $I$ .

Why? If  $N[v] \cap I = \emptyset$  for some  $v \in V$ ,  $I \cup \{v\}$  would be a larger solution.

- 1: **function** MISsize( $G = (V, E)$ )
- 2:     **if**  $V = \emptyset$  **then return** 0
- 3:      $v \leftarrow$  vertex in  $V$  of minimum degree.
- 4:     **return**  $1 + \max\{\text{MISsize}(G \setminus N[w]) \mid w \in N[v]\}$

# Exact MIS via Branching



# Exact MIS via Branching

Let  $T(n)$  be the maximum number of subproblems considered by the branching algorithm on a graph with  $n$  vertices, then (very loosely):

$$T(0) = 1$$

$$T(n) \leq 1 + \sum_{w \in N[v]} T(n - (d(w) + 1))$$

$$\leq 1 + (d(v) + 1) \cdot T(n - (d(v) + 1)) \quad \text{since } T(\cdot) \text{ is nondecreasing}$$

$$= 1 + s \cdot T(n - s)$$

for some  $v \in V$  of minimum degree

where  $s = d(v) + 1$  and thus  $s \in \{1, \dots, n\}$

## Lemma

$$T(n) \in \mathcal{O}(3^{n/3}) \subset \mathcal{O}(1.44225^n)$$

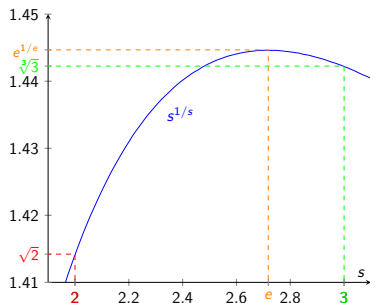
“Proof” (Error:  $s$  depends on  $v$ ).

$$T(n) \leq 1 + s \cdot T(n - s)$$

$$\leq 1 + s + s^2 + \dots + s^{n/s}$$

$$= \frac{s^{1+n/s} - 1}{s - 1} < 2s^{n/s} \quad (\text{for } s \geq 2)$$

$$\in \mathcal{O}(s^{n/s}) \subseteq \mathcal{O}(3^{n/3}) \quad \square$$



## AADS Lecture on EE/FPT, Part 3

### Parameterized problems

## “Bar fight prevention” aka $k$ -Vertex Cover

**Problem:** Bouncer in a small city wants to block people at the door to prevent fights. Assume he knows everyone and knows which pairs of people would fight if they were both let in. Management only allows him to block  $\leq k$  of the  $n$  people who wants in. Is that enough to prevent fights, and if so, who should be blocked?

**Equivalent Problem:** Given a graph  $(V, E)$  with  $n = |V|$  vertices, is there a subset  $C \subseteq V$  of size  $|C| \leq k$  such that every edge has at least one endpoint in  $C$ ? Such a set  $C$  is called a  *$k$ -Vertex Cover* in the graph, and its complement  $V \setminus C$  is an *Independent Set* of size  $n - k$ .

For concreteness in the following, suppose  $n = 1000$  and  $k = 10$ .

**Naive 1:** Try all  $2^n$  subsets of people. ( $2^{1000} \approx 1.07 \cdot 10^{301}$  cases).

**Naive 2:** Use Exact MIS algorithm. ( $2 \cdot 3^{1000/3} - 1 \approx 2.195 \cdot 10^{159}$  cases).

**Better 1:** Try all  $\binom{n}{k}$  subsets of  $k$  people. ( $\binom{1000}{10} \approx 2.63 \cdot 10^{23}$  cases).

## “Bar fight prevention” via Kernelization

Consider the conflict graph  $G = (V, E)$ .

**Idea:** If  $d(v) = 0$ : let  $v$  in and drop  $v$  from  $G$ .

Why? **Safe because no conflicts.**

**Idea:** If  $d(v) > k$ : reject  $v$ , drop  $v$  from  $G$ , and decrease  $k$ .

Why? **Not rejecting  $v$  means rejecting  $d(v) > k$  people.**

**Note:** If  $d(v) \leq k$  for all  $v$  and  $|E| > k^2$ , there is no solution.

Why? **Each rejection resolves at most  $k$  conflicts.**

**Better 2:** The above ideas reduce to a graph  $H$  with  $|V| \leq 2k^2$  vertices.

Why?  $|V| = \sum_{v \in V} 1 \leq \sum_{v \in V} d(v) = 2|E| \leq 2k^2$

Now try all  $\binom{2k^2}{k}$  subsets of  $k$  people.  $\left(\binom{2 \cdot 10^2}{10}\right) \approx 2.24 \cdot 10^{16}$ .

**Idea:** If  $N[v] = \{v, w\}$ : let  $v$  in, reject  $w$ , drop  $N[v]$  from  $G$ , and decrease  $k$ .

Why? **In any solution that lets  $w$  in, we can let  $v$  in instead. Never worse.**

**Better 3:** The above ideas reduce to a graph  $H$  with  $|V| \leq k^2$  vertices.

Why?  $|V| = \sum_{v \in V} 1 = \frac{1}{2} \sum_{v \in V} 2 \leq \frac{1}{2} \sum_{v \in V} d(v) = |E| \leq k^2$

Now try all  $\binom{k^2}{k}$  subsets of  $k$  people.  $\left(\binom{10^2}{10}\right) \approx 1.73 \cdot 10^{13}$ .

## “Bar fight prevention” via Kernelization

```
1: function BarFightPrevention( $k, G$ )
2:    $k', H, C \leftarrow \text{BFP-Kernel}(k, G)$ 
3:   if  $H$  has  $\leq (k')^2$  edges and  $\text{BFP-Brute-Force}(k', H)$  returns a solution  $C'$  then
4:     return  $C \cup C'$ 
5:   return “No solution”

6: function BFP-Kernel( $k, G$ )
7:    $k' \leftarrow k, H \leftarrow G, C \leftarrow \emptyset$ 
8:   loop
9:     if Some  $v$  has  $d(v) = 0$  then
10:       $H \leftarrow H \setminus \{v\}$ 
11:     elseif  $k' > 0$  and some  $v$  has  $d(v) > k'$  then
12:       $H \leftarrow H \setminus \{v\}, C \leftarrow C \cup \{v\}, k' \leftarrow k' - 1$ 
13:     elseif  $k' > 0$  and some  $v$  has  $N[v] = \{v, w\}$  for some  $w$  then
14:       $H \leftarrow H \setminus N[v], C \leftarrow C \cup \{w\}, k' \leftarrow k' - 1$ 
15:     else
16:       return  $k', H, C$ 

17: function BFP-Brute-Force( $k, G = (V, E)$ )
18:   for every subset  $C \subseteq V$  of size  $k$  do
19:     if  $C$  is a vertex cover of  $G$  then
20:       return  $C$ 
21:   return “No solution”
```



## Kernelization

The subgraph  $H$  we reduced to before brute-forcing is called a *Kernel* for the Bar Fight Prevention problem, and the process of finding such a kernel is called *Kernelization*.

The general idea is to use the parameter  $k$  to quickly reduce to a smaller problem, whose size ideally depends only on  $k$  and not on  $n$ .

For the bar fight prevention problem we have just shown that:

- ▶ If there is a solution for a given  $k$  and a given graph  $G$  with  $n$  vertices and  $m$  edges, then we can find a kernel  $H$  with at most  $k^2$  vertices.
- ▶ Furthermore, such a kernel can be found in  $\mathcal{O}(m + n)$  time, and checking if a given subset of size at most  $k$  is a solution can be done in  $\mathcal{O}(k^2)$  time.
- ▶ Thus, for any fixed  $k$ , the total running time of this algorithm is  $\mathcal{O}(m + n + \binom{k^2}{k} k^2) \subseteq \mathcal{O}(m + n + k^{2k+2}) \subseteq \mathcal{O}_k(m + n)$ .

## “Bar fight prevention” via Bounded Search Tree

**Note:** For each edge  $(u, v) \in E$ , at least one of  $u, v$  must be rejected.

**Idea:** Pick arbitrary edge  $(u, v)$ , and recursively try with  $u$  rejected and with  $v$  rejected.

```
1: function BFP-Bounded-Search( $k, G$ )
2:   if  $G$  has no edges then
3:     return  $\emptyset$ 
4:   if  $k > 0$  then
5:     Let  $(u, v)$  be an arbitrary edge of  $G$ 
6:     for  $w \in \{u, v\}$  do
7:       if BFP-Bounded-Search( $k - 1, G \setminus \{w\}$ ) returns a solution  $C$  then
8:         return  $C \cup \{w\}$ 
9:   return “No solution”
```

This recursive procedure has depth at most  $k$ .

Thus the total number of subproblems considered at most  $2^k$ .

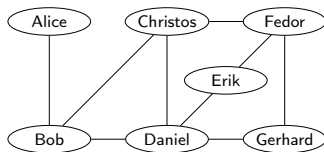
If we start by rejecting all vertices of degree  $d(v) > k$  (like in the kernelization approach), the resulting graph has at most

$|E| = \frac{1}{2} \sum_{v \in V} d(v) \leq \frac{1}{2} nk$  edges, so constructing each subproblem can be done in  $\mathcal{O}(nk)$  time.

The total running time is then  $\mathcal{O}(m + nk \cdot 2^k)$ .  $(1000 \cdot 10 \cdot 2^{10} \approx 10^7)$

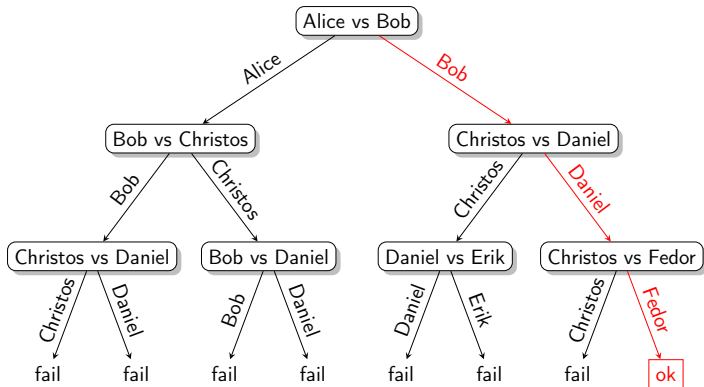
Part of Assignment 5 asks you to improve this.

# "Bar fight prevention" via Bounded Search Tree



$k = 3$

$C = \{\text{Fedor, Daniel, Bob}\}$



## AADS Lecture on EE/FPT, Part 4

FPT vs XP

## FPT vs XP

An important feature of the Bar Fight Prevention problem is the existence of the *parameter*  $k$ . The problem of finding the minimum  $k$  that works is NP-complete, but for any fixed constant  $k$  we have just seen two linear-time algorithms!

We say the problem is *parameterized* by the *parameter*  $k$ . In this case  $k$  is the maximum solution size, but other problems may have different parameters (and may have more than one).

**Definition:** A parameterized problem is *Fixed Parameter Tractable (FPT)* if it has an algorithm with running time  $f(k) \cdot n^c$  for some function  $f$  and some constant  $c \in \mathbb{R}$ .

**Definition:** A parameterized problem is *Slice-wise Polynomial (XP)* if it has an algorithm with running time  $f(k) \cdot n^{g(k)}$  for some functions  $f, g$ .

**Note:**  $\text{FPT} \subset \text{XP}$ , why? Simply set  $g(k) = c$ .

## Example: Vertex $k$ -Coloring

**Problem:** For an integer  $k$ , given a graph  $G$  does  $G$  have a proper vertex coloring with  $k$  colors?

### Lemma

*Unless  $P = NP$ , this problem is not  $XP$  and therefore not  $FPT$ .*

### Proof sketch.

The problem is NP-hard even for  $k = 5$ , so unless  $P = NP$  there can be no algorithm for general  $k$  with running time  $f(k) \cdot n^{g(k)}$ . □

## Example: $k$ -Clique

**Problem:** For an integer  $k$ , given a graph  $G$  does  $G$  have a clique of size  $k$ ?

### Lemma

*$k$ -clique is XP.*

### Proof.

A simple brute-force algorithm is to check every  $k$ -subset of the vertices. There are  $\binom{n}{k} \leq n^k$  such subsets, and we can check in  $\mathcal{O}(k^2)$  time whether a given subset forms a clique. Thus the running time of this algorithm is  $\mathcal{O}(k^2 \cdot n^k)$  which proves the problem is in XP.  $\square$

It is unknown whether  $k$ -clique is FPT, but it is widely believed that  $\mathcal{O}(n^k)$  is optimal which would prove it is not.

## Example: Clique parameterized by $\Delta$

**Problem:** For integer  $\Delta$ , given a graph  $G$  with maximum degree  $\Delta$  and an integer  $k$ , does  $G$  have a clique of size  $k$ ?

### Lemma

*Clique is FPT when parameterized by the maximum degree  $\Delta$ .*

### Proof.

If  $k > \Delta + 1$  the answer is just no. Otherwise a naive algorithm is for each vertex to try all subsets of its neighbors. There are at most  $n \cdot 2^\Delta$  such subsets and each can be checked in  $\mathcal{O}(\Delta^2)$  time. The total time is thus  $\mathcal{O}((2^\Delta \cdot \Delta^2) \cdot n)$ , which proves the problem is FPT.  $\square$

In fact, we can easily improve this algorithm to run in  $\mathcal{O}(\binom{\Delta}{k-1} \cdot k^2 \cdot n)$  time when  $k \leq \Delta + 1$ .

There are often many possible choices of parameter. Choosing the right one for a specific problem is an art.



# Summary

Today's topics were “Exact exponential algorithms” and “Parameterized Complexity”. We have covered

- ▶ The natural brute force algorithm for problems in NP.
- ▶ An exact  $\mathcal{O}^*(2^n)$ -time dynamic programming algorithm for TSP.
- ▶ An exact  $\mathcal{O}^*(3^{n/3})$ -time branching algorithm for MIS.
- ▶ A kernelization for the “Bar Fight Prevention” problem, a.k.a.  $k$ -vertex cover.
- ▶ A bounded search tree algorithm for  $k$ -vertex cover.
- ▶ Definitions of parameterized complexity, FPT and XP.
- ▶ Examples of problems in FPT, XP but not FPT, and not XP.
- ▶ Next time: van Emde Boas trees / predecessor search.

# Advanced algorithms and data structures

Lecture: van Emde Boas Trees

Jacob Holm ([jaho@di.ku.dk](mailto:jaho@di.ku.dk))

Dec 18th 2024

# Today's Lecture

## van Emde Boas Trees

- Predecessor search/ordered sets

- Naive

- Twolevel

- Recursive

- vEB: worst case  $\mathcal{O}(\log \log |U|)$  time

- RS-vEB: expected  $\mathcal{O}(\log \log |U|)$  time,  $\mathcal{O}(n \log \log |U|)$  space

- $R^2S$ -vEB: expected  $\mathcal{O}(\log \log |U|)$  time,  $\mathcal{O}(n)$  space

- Bonus: vEB is optimal for  $w = \Theta(\log n)$

- Bonus: Integer sorting in expected  $\mathcal{O}(n \log \log |U|)$  time.

# Predecessor search/ordered sets

## Problem:

Given a universe  $U = [u]$  where  $u = 2^w$ ,  
maintain subset  $S \subseteq U$ ,  $|S| = n$  under:

**member**( $x, S$ ): Return  $[x \in S]$ .

**insert**( $x, S$ ): Add  $x$  to  $S$  (assumes  $x \notin S$ ).

**delete**( $x, S$ ): Remove  $x$  from  $S$  (assumes  $x \in S$ ).

**empty**( $S$ ): Return  $[S = \emptyset]$ .

**min**( $S$ ): Return  $\min S$  (assumes  $S \neq \emptyset$ ).

**max**( $S$ ): Return  $\max S$  (assumes  $S \neq \emptyset$ ).

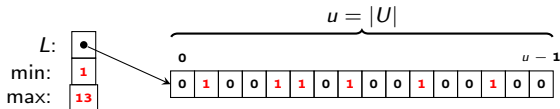
**predecessor**( $x, S$ ): Return  $\max\{y \in S \mid y < x\}$   
(assumes  $\{y \in S \mid y < x\}$  is nonempty,  
i.e.  $S \neq \emptyset$  and  $x > \min(S)$ ).

**successor**( $x, S$ ): Return  $\min\{y \in S \mid y > x\}$   
(assumes  $\{y \in S \mid y > x\}$  is nonempty,  
i.e.  $S \neq \emptyset$  and  $x < \max(S)$ ).

## Naive

**Idea:** If we are willing to spend  $\mathcal{O}(|U|)$  space...

Store  $S$  as a bit-array  $L$  of length  $|U|$  such that  $L[x] = [x \in S]$ , and keep track of the min and max values explicitly.



$$\{1, 4, 5, 7, 10, 13\}$$

How fast is:

$\text{empty}(S)$ ,  $\text{min}(S)$ , and  $\text{max}(S)$ ? **worst case  $\mathcal{O}(1)$  —  $\text{empty}(S)$  is tricky.**

$\text{member}(x, S)$ ? **worst case  $\mathcal{O}(1)$ .**

$\text{predecessor}(x, S)$  and  $\text{successor}(x, S)$ ? **worst case  $\Theta(|U|)$ .**

$\text{delete}(x, S)$ ? **worst case  $\mathcal{O}(|U|)$  — need to update min / max.**

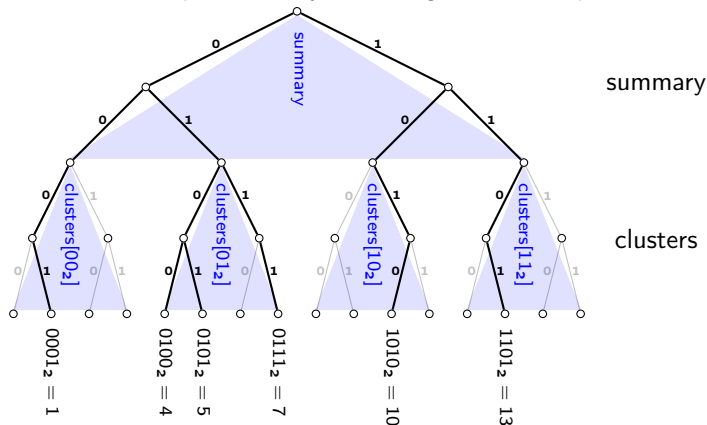
$\text{insert}(x, S)$ ? **worst case  $\mathcal{O}(1)$ .**

# Bit-Trie

**Idea:** Think of each key as a  $w$ -bit string describing a path in a binary *trie* (= a special kind of tree).

The naive structure can be thought of as just storing which leaves exist. An alternative naive choice would be to store the actual trie, which we could maintain in  $\mathcal{O}(w) = \mathcal{O}(\log|U|)$  time.

What if we instead split each key into a high and a low part?



## Twolevel

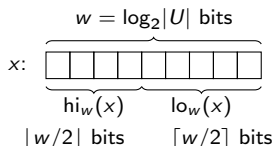
**Idea:** Split each key into *high* and *low* parts. Use naive for each.

Recall that  $U = [2^w]$ , and define:

$$\text{hi}_w(x) := \left\lfloor \frac{x}{2^{\lceil w/2 \rceil}} \right\rfloor$$

$$\text{lo}_w(x) := x \bmod 2^{\lceil w/2 \rceil}$$

$$\text{index}_w(h, \ell) := h \cdot 2^{\lceil w/2 \rceil} + \ell$$



note that  $x = \text{index}_w(\text{hi}_w(x), \text{lo}_w(x))$ .

Now let the structure directly store the values:

$$\text{min} := \min(S) \text{ if } S \neq \emptyset, \text{ else } 1$$

$$\text{max} := \max(S) \text{ if } S \neq \emptyset, \text{ else } 0$$

and use the naive structure to store

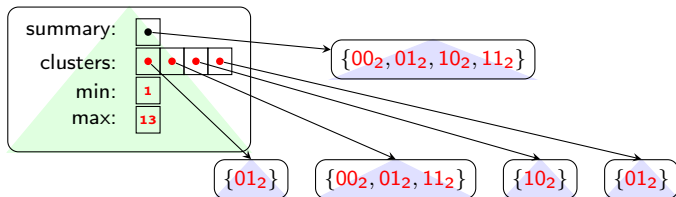
$$\text{summary} := \{\text{hi}_w(x) \mid x \in S\}$$

$$\text{clusters}[h] := \{\ell \in [2^{\lceil w/2 \rceil}] \mid \text{index}_w(h, \ell) \in S\} \quad \forall h \in [2^{\lfloor w/2 \rfloor}]$$

note that  $S = \bigcup_{h \in \text{summary}} \{\text{index}_w(h, \ell) \mid \ell \in \text{clusters}[h]\}$ .

## Twolevel

We can draw the structure for the set  $S = \{1, 4, 5, 7, 10, 13\} = \{0001_2, 0100_2, 0101_2, 0111_2, 1010_2, 1101_2\} \subseteq [2^4]$  as:



How fast (worst case) is:

$\text{empty}(S)$ ,  $\text{min}(S)$ ,  $\text{max}(S)$ ?

$\mathcal{O}(1)$

$\text{member}(x, S)$ ?

$\mathcal{O}(1) + 1 \times \text{naive} = \mathcal{O}(1)$

$\text{predecessor}(x, S)$ ,  $\text{successor}(x, S)$ ?

$\mathcal{O}(1) + 1 \times \text{naive}$

$= \Theta(2^{\lceil w/2 \rceil}) = \Theta(\sqrt{|U|})$

$\text{delete}(x, S)$ ?

$\mathcal{O}(1) + 2 \times \text{naive} = \Theta(\sqrt{|U|})$

$\text{insert}(x, S)$ ?

$\mathcal{O}(1) + 2 \times \text{naive} = \mathcal{O}(1)$

```
function empty(S)
```

```
    return S.min > S.max
```

```
function min(S)
```

```
    ▷ Assumes  $S \neq \emptyset$ 
```

```
    return S.min
```

```
function max(S)
```

```
    ▷ Assumes  $S \neq \emptyset$ 
```

```
    return S.max
```

```
function memberw(x, S)
```

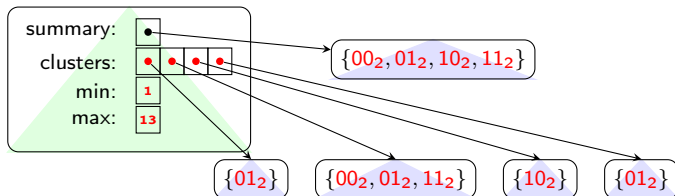
```
    return member $\lceil w/2 \rceil$ (loww(x), S.clusters[hiw(x)])
```

How can we improve the time further?



# Twollevel

We can draw the structure for the set  $S = \{1, 4, 5, 7, 10, 13\} = \{0001_2, 0100_2, 0101_2, 0111_2, 1010_2, 1101_2\} \subseteq [2^4]$  as:



How fast (worst case) is:

$\text{empty}(S)$ ,  $\text{min}(S)$ ,  $\text{max}(S)$ ?

$\mathcal{O}(1)$

$\text{member}(x, S)$ ?

$\mathcal{O}(1) + 1 \times \text{naive} = \mathcal{O}(1)$

$\text{predecessor}(x, S)$ ,  $\text{successor}(x, S)$ ?

$\mathcal{O}(1) + 1 \times \text{naive}$

$= \Theta(2^{\lceil w/2 \rceil}) = \Theta(\sqrt{|U|})$

$\text{delete}(x, S)$ ?

$\mathcal{O}(1) + 2 \times \text{naive} = \Theta(\sqrt{|U|})$

$\text{insert}(x, S)$ ?

$\mathcal{O}(1) + 2 \times \text{naive} = \mathcal{O}(1)$

**function**  $\text{predecessor}_w(x, S)$

▷ Assumes  $S \neq \emptyset$  and  $\text{min}(S) < x$

**if**  $x > S.\text{max}$  **then**

**return**  $S.\text{max}$

$p \leftarrow \text{hi}_w(x)$ ,  $s \leftarrow \text{lo}_w(x)$ ,  $C \leftarrow S.\text{clusters}[p]$

**if not**  $\text{empty}(C)$  **and**  $C.\text{min} < s$  **then**

**return**  $\text{index}_w(p, \text{predecessor}_{\lceil w/2 \rceil}(s, C))$

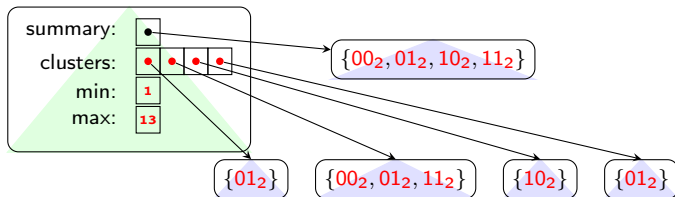
$p \leftarrow \text{predecessor}_{\lfloor w/2 \rfloor}(p, S.\text{summary})$

**return**  $\text{index}_w(p, S.\text{clusters}[p].\text{max})$

How can we improve the time further?

## Twollevel

We can draw the structure for the set  $S = \{1, 4, 5, 7, 10, 13\} = \{0001_2, 0100_2, 0101_2, 0111_2, 1010_2, 1101_2\} \subseteq [2^4]$  as:



How fast (worst case) is:

$\text{empty}(S)$ ,  $\text{min}(S)$ ,  $\text{max}(S)$ ?

$\mathcal{O}(1)$

$\text{member}(x, S)$ ?

$\mathcal{O}(1) + 1 \times \text{naive} = \mathcal{O}(1)$

$\text{predecessor}(x, S)$ ,  $\text{successor}(x, S)$ ?

$\mathcal{O}(1) + 1 \times \text{naive}$

$= \Theta(2^{\lceil w/2 \rceil}) = \Theta(\sqrt{|U|})$

$\text{delete}(x, S)$ ?

$\mathcal{O}(1) + 2 \times \text{naive} = \Theta(\sqrt{|U|})$

$\text{insert}(x, S)$ ?

$\mathcal{O}(1) + 2 \times \text{naive} = \mathcal{O}(1)$

**function**  $\text{delete}_w(x, S)$

▷ Assumes  $x \in S$

$p \leftarrow \text{hi}_w(x)$ ,  $s \leftarrow \text{lo}_w(x)$ ,  $C \leftarrow S.\text{clusters}[p]$

**delete** $_{\lceil w/2 \rceil}(s, C)$

**if**  $\text{empty}(C)$  **then**

**delete** $_{\lfloor w/2 \rfloor}(p, S.\text{summary})$

**if**  $S.\text{min} = S.\text{max}$  **then**

$S.\text{min} \leftarrow 1$ ,  $S.\text{max} \leftarrow 0$

**else if**  $x = S.\text{min}$  **then**

$p \leftarrow S.\text{summary}.min$ ,  $S.\text{min} \leftarrow \text{index}_w(p, S.\text{clusters}[p].min)$

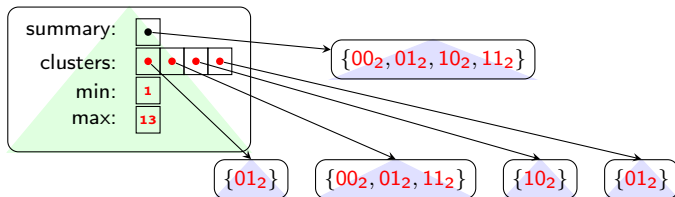
**else if**  $x = S.\text{max}$  **then**

$p \leftarrow S.\text{summary}.max$ ,  $S.\text{max} \leftarrow \text{index}_w(p, S.\text{clusters}[p].max)$

How can we improve the time further?

## Twolevel

We can draw the structure for the set  $S = \{1, 4, 5, 7, 10, 13\} = \{0001_2, 0100_2, 0101_2, 0111_2, 1010_2, 1101_2\} \subseteq [2^4]$  as:



How fast (worst case) is:

$\text{empty}(S)$ ,  $\text{min}(S)$ ,  $\text{max}(S)$ ?

$\mathcal{O}(1)$

$\text{member}(x, S)$ ?

$\mathcal{O}(1) + 1 \times \text{naive} = \mathcal{O}(1)$

$\text{predecessor}(x, S)$ ,  $\text{successor}(x, S)$ ?

$\mathcal{O}(1) + 1 \times \text{naive}$

$= \Theta(2^{\lceil w/2 \rceil}) = \Theta(\sqrt{|U|})$

$\text{delete}(x, S)$ ?

$\mathcal{O}(1) + 2 \times \text{naive} = \Theta(\sqrt{|U|})$

$\text{insert}(x, S)$ ?

$\mathcal{O}(1) + 2 \times \text{naive} = \mathcal{O}(1)$

**function**  $\text{insert}_w(x, S)$

▷ Assumes  $x \notin S$

**if**  $\text{empty}(S)$  **then**

$S.\text{min} \leftarrow x$ ,  $S.\text{max} \leftarrow x$

**if**  $x < S.\text{min}$  **then**  $S.\text{min} \leftarrow x$

**if**  $x > S.\text{max}$  **then**  $S.\text{max} \leftarrow x$

$p \leftarrow \text{hi}_w(x)$ ,  $s \leftarrow \text{lo}_w(x)$

**if**  $\text{empty}(S.\text{clusters}[p])$  **then**

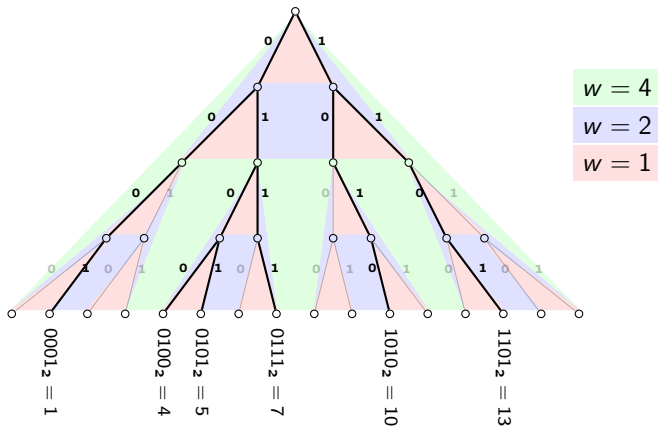
$\text{insert}_{\lfloor w/2 \rfloor}(p, S.\text{summary})$

$\text{insert}_{\lceil w/2 \rceil}(s, S.\text{clusters}[p])$

How can we improve the time further?

## Recursive

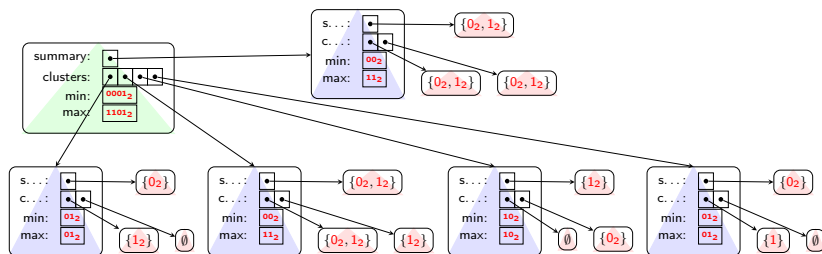
**Idea:** Instead of using the naive structure for summary and clusters[ $h$ ], recursively use the same type of structure (stop recursion when  $w = 1$ ).



# Recursive

**Idea:** Instead of using the naive structure for summary and clusters[ $h$ ], recursively use the same type of structure (stop recursion when  $w = 1$ ).

We can draw the recursive structure for the set  $S = \{1, 4, 5, 7, 10, 13\} = \{0001_2, 0100_2, 0101_2, 0111_2, 1010_2, 1101_2\} \subseteq [2^4]$  as:



This structure is (essentially) what the book calls “proto-vEB”. Note that the naive structure we started with is almost completely gone.

# Recursive

## Theorem

*The recursion depth of this structure, when used on the universe  $U = [2^w]$  is  $\lceil \log_2 w \rceil = \mathcal{O}(\log \log |U|)$ .*

## Proof.

Let  $d(w)$  be the recursion depth when working with a universe of size  $2^w$ . We will prove by strong induction that  $d(w) = \lceil \log_2 w \rceil$ .

The base case is  $w = 1$  where there is no recursion, so  $d(1) = 0 = \lceil \log_2(1) \rceil$ .

For the induction case, suppose  $w > 1$  and that  $d(w') = \lceil \log_2(w') \rceil$  for all  $w' \in [w]_+$ . Now let  $w' = \lceil w/2 \rceil \in [w]_+$ , then the largest universe size used in the recursion is  $2^{w'}$ , and (by induction)

$d(w) = 1 + d(w') = 1 + \lceil \log_2(w') \rceil = \lceil \log_2(2w') \rceil$ . What remains is to show  $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$ .

If  $w$  is even,  $2w' = w$  and we are done.

Otherwise  $w > 1$  is odd so  $w \geq 3$  and the smallest integer  $k = \lceil \log_2(w) \rceil$  such that  $2^k \geq w$  satisfies  $k \geq 1$ . Thus  $2^k$  is even and so must satisfy  $2^k \geq w + 1 = 2w'$  and therefore also  $k = \lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$ .  $\square$

# Recursive

Using the recursive structure, how fast is:

$\text{empty}(S)$ ,  $\text{min}(S)$ , and  $\text{max}(S)$ ? worst case  $\mathcal{O}(1)$ .

$\text{member}(x, S)$ ?  $\mathcal{O}(1) + 1 \times \text{recursion} = \mathcal{O}(d(w)) = \mathcal{O}(\log \log |U|)$ .

$\text{predecessor}(x, S)$  and  $\text{successor}(x, S)$ ?  $\mathcal{O}(1) + 1 \times \text{recursion}$   
 $= \mathcal{O}(d(w)) = \mathcal{O}(\log \log |U|)$ .

$\text{insert}(x, S)$  and  $\text{delete}(x, S)$ ?  $\mathcal{O}(1) + 2 \times \text{recursion}$   
 $= \Theta(2^{d(w)}) = \Theta(w) = \Theta(\log |U|)$ .

How can we improve the update time? Somehow make insert and delete recurse in only one substructure.

## vEB: worst case $\mathcal{O}(\log \log |U|)$ time

**Idea:** Exclude  $\min(S)$  and/or  $\max(S)$  from the set of keys stored in summary and clusters. (CLRS excludes only  $\min(S)$ , I exclude both).

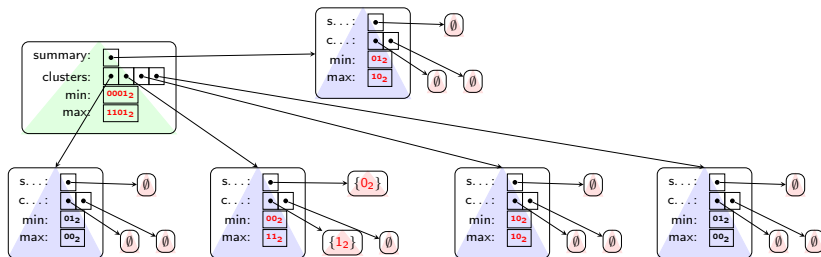
Specifically, redefine summary and clusters to recursively store the sets:

$$\text{summary} := \{\text{hi}_w(x) \mid x \in S \setminus \{\min, \max\}\}$$

$$\text{clusters}[h] := \{\ell \in [2^{\lceil w/2 \rceil}] \mid \text{index}_w(h, \ell) \in S \setminus \{\min, \max\}\} \quad \forall h \in [2^{\lfloor w/2 \rfloor}]$$

We can draw the van Emde Boas tree for the set  $S =$

$$\{1, 4, 5, 7, 10, 13\} = \{0001_2, 0100_2, 0101_2, 0111_2, 1010_2, 1101_2\} \subseteq [2^4] \text{ as:}$$





vEB: worst case  $\mathcal{O}(\log \log |U|)$  time,  $\mathcal{O}(|U|)$  space

```
1: function predecessorw(x, S)           ▷ Assumes  $S \neq \emptyset$  and  $x > S.\text{min}$ 
2:   if  $x > S.\text{max}$  then
3:     return  $S.\text{max}$ 
4:   if  $w = 1$  then
5:     return  $S.\text{min}$ 
6:    $p \leftarrow \text{hi}_w(x)$ ,  $s \leftarrow \text{lo}_w(x)$ ,  $C \leftarrow S.\text{clusters}[p]$ 
7:   if not empty( $C$ ) and  $C.\text{min} < s$  then
8:     return indexw( $p$ , predecessor $\lceil w/2 \rceil$ ( $s$ ,  $C$ ))
9:   if empty( $S.\text{summary}$ ) or  $p \leq S.\text{summary}.\text{min}$  then
10:    return  $S.\text{min}$ 
11:    $p \leftarrow$  predecessor $\lfloor w/2 \rfloor$ ( $p$ ,  $S.\text{summary}$ )
12:   return indexw( $p$ ,  $S.\text{clusters}[p].\text{max}$ )
```

## Theorem

predecessor<sub>w</sub>( $x$ ,  $S$ ) takes worst case  $\mathcal{O}(d(w)) = \mathcal{O}(\log \log |U|)$  time.

## Proof.

It makes at most one recursive call.



vEB: worst case  $\mathcal{O}(\log \log |U|)$  time,  $\mathcal{O}(|U|)$  space

```
1: function insertw(x, S) ▷ Assumes  $x \notin S$ 
2:   if empty(S) then
3:     S.min  $\leftarrow$  x, S.max  $\leftarrow$  x, return
4:   if S.min = S.max then
5:     if x < S.min then S.min  $\leftarrow$  x
6:     if x > S.max then S.max  $\leftarrow$  x
7:     return
8:   if x < S.min then S.min  $\leftrightarrow$  x
9:   if x > S.max then S.max  $\leftrightarrow$  x
10:  p  $\leftarrow$  hiw(x), s  $\leftarrow$  low(x)
11:  if empty(S.clusters[p]) then
12:    insert[w/2](p, S.summary)
13:  insert[w/2](s, S.clusters[p])
```

## Theorem

insert<sub>w</sub>(x, S) takes worst case  $\mathcal{O}(d(w)) = \mathcal{O}(\log \log |U|)$  time.

## Proof.

It makes at most one recursive call on a non-empty substructure, and inserting in an empty substructure takes constant time. □

RS-vEB: expected  $\mathcal{O}(\log \log |U|)$  time,  $\mathcal{O}(n \log \log |U|)$  space

**Idea:** Use a hash table instead of an array to store clusters $[\cdot]$ , and don't store empty substructures.

Why does this change updates from worst case  $\mathcal{O}(\log \log |U|)$  to expected  $\mathcal{O}(\log \log |U|)$  time? Because updates to a hash table take expected  $\mathcal{O}(1)$  time rather than worst case.

Why does this only use  $\mathcal{O}(n \cdot d(w)) = \mathcal{O}(n \log \log |U|)$  space? Because the empty structure uses  $\mathcal{O}(1)$  space and  $\text{insert}_w$  only creates or updates  $\mathcal{O}(d(w))$  substructures in the worst case. Each of these costs at most an additional  $\mathcal{O}(1)$  space.

$R^2S$ -vEB: expected  $\mathcal{O}(\log \log |U|)$  time,  $\mathcal{O}(n)$  space

**Idea:** Partition  $S$  into “chunks” of size  $\Theta(\min\{n, \log \log |U|\})$ , and store only one element representing each chunk in the RS-vEB structure. I.e. RS-vEB stores only  $\mathcal{O}(\max\{1, n/\log \log |U|\})$  elements, using  $\mathcal{O}(n)$  space.

We can store each chunk as a sorted linked list, and keep a hash table mapping each representative to its chunk. This also takes  $\mathcal{O}(n)$  space.

predecessor and successor uses the RS-vEB structure to find the nearest two representatives in  $\mathcal{O}(\log \log |U|)$  time, and can then spend linear time in the size of the two chunks to find the result.

insert may have to split a chunk that becomes too large and insert a new representative in the RS-vEB structure. Splitting the chunk can take linear time in the size of the chunk, and together with inserting the new representative into the RS-vEB structure, this still only takes expected  $\mathcal{O}(\log \log |U|)$  time.

Similarly, delete may have to join two chunks and delete a representative, but again this only takes expected  $\mathcal{O}(\log \log |U|)$  time.

## Bonus: vEB is optimal for $w = \Theta(\log n)$

In fact,  $\mathcal{O}(\log w)$  is the optimal query time when  $w \in \Theta(\log n)$ , or equivalently when  $n \in 2^{\Theta(w)}$ . This was shown in:

*Time-space trade-offs for predecessor search,*

Mihai Pătraşcu and Mikkel Thorup,

STOC'06: Proceedings of the thirty-eighth annual ACM symposium on Theory of Computing, pages 232–240.

<https://doi.org/10.1145/1132516.1132551>

## Bonus: Integer sorting in expected $\mathcal{O}(n \log \log |U|)$ time.

**Idea:** Insert all elements in  $R^2S$ -vEB structure, use  $\min(S)$  then repeatedly  $\text{successor}(x, S)$ .

This takes expected  $\mathcal{O}(n \log \log |U|) = \mathcal{O}(n \log w)$  time.

Compare this to other famous sorting methods:

**QuickSort** Takes expected  $\mathcal{O}(n \log n)$  time.

**CountingSort** Takes  $\mathcal{O}(n + |U|)$  time.

**RadixSort** Takes  $\mathcal{O}((n + r) \log_r |U|) = \mathcal{O}((n + r) \frac{w}{\log r})$  time for radix  $r$ , e.g.  $\mathcal{O}(\frac{n}{\log n} \log |U|) = \mathcal{O}(\frac{nw}{\log n})$  for  $r = n$ .

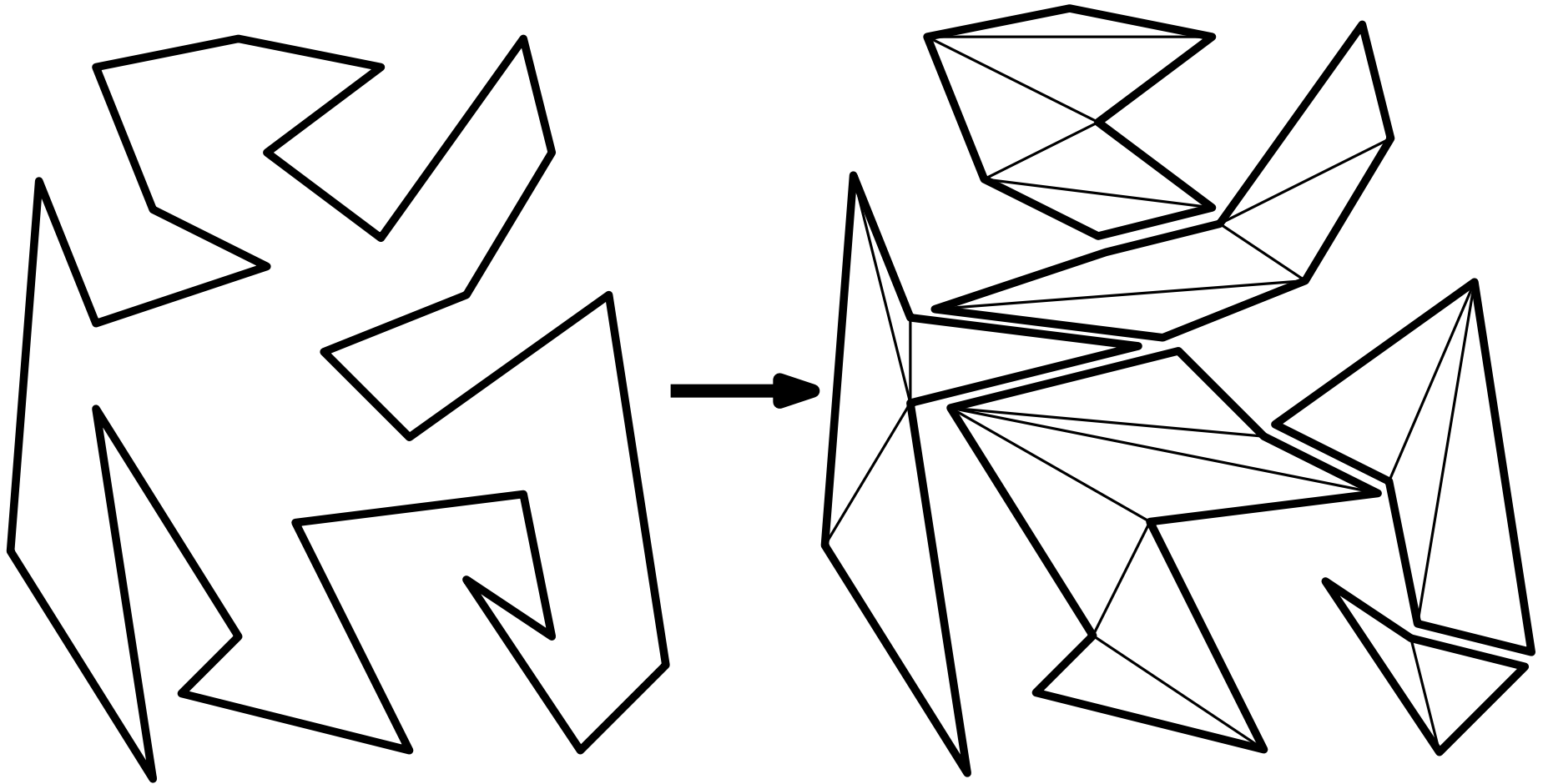
We observe that for  $\log n \in \Omega(\log w) \cap \mathcal{O}(\frac{w}{\log w})$ , or equivalently  $n \in w^{\Omega(1)} \cap 2^{\mathcal{O}(\frac{w}{\log w})}$ , sorting using vEB should be better than these alternatives.

# Summary

Today's topic was van Emde Boas Trees, which is a data structure for sets of bounded integers. We have covered

- ▶ A naive implementation of ordered sets.
- ▶ A slightly less naive two-level implementation.
- ▶ A recursive implementation with good query time (proto-vEB).
- ▶ The actual van Emde Boas Tree (vEB).
- ▶ An extension to vEB using hashing to save space (RS-vEB).
- ▶ A further space-saving extension using “indirection” ( $R^2S$ -vEB).
- ▶ Some notes about the optimality of vEB and on integer sorting.
- ▶ Next time: Polygonal Triangulation

# Polygon Triangulation



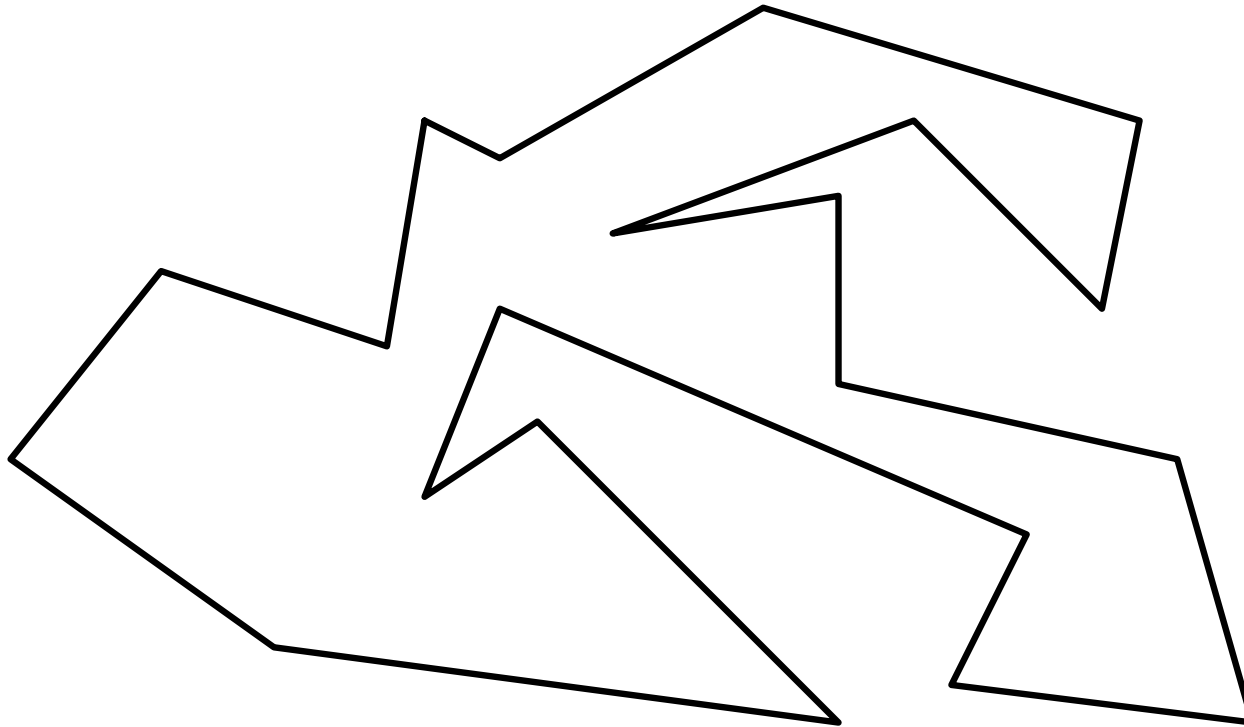
Mikkel Abrahamsen



# Motivation: The Art Gallery Problem



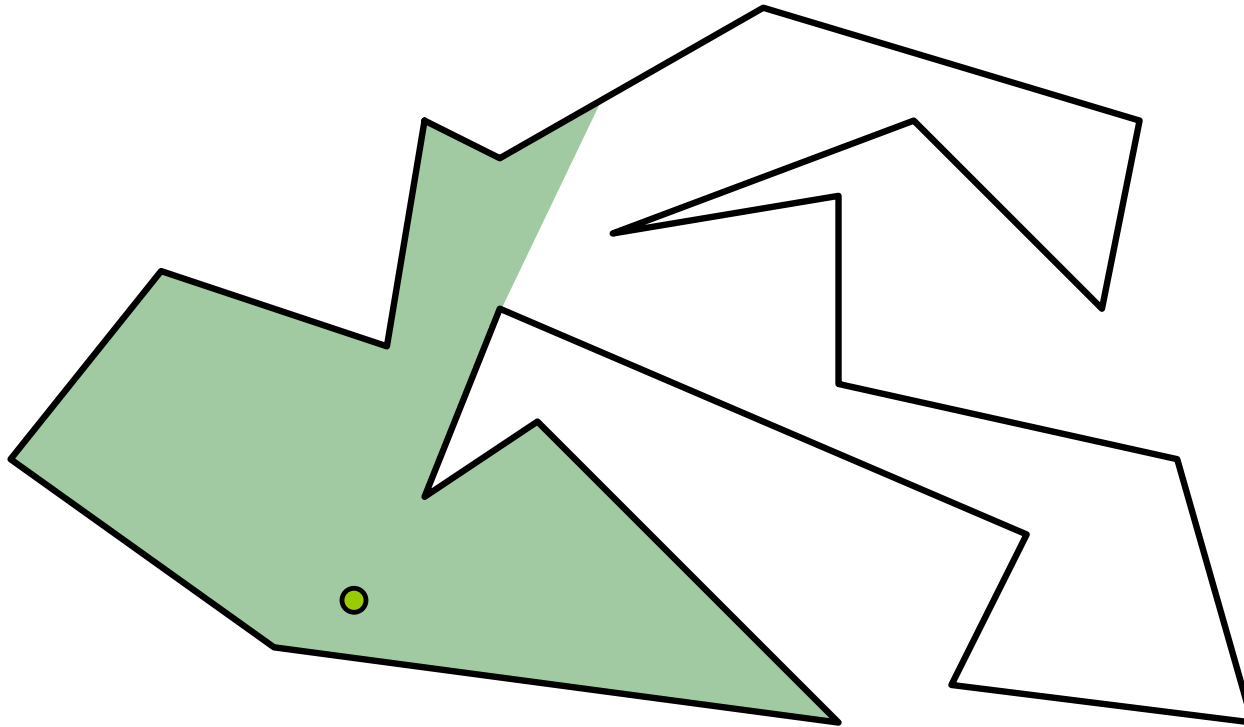
Viktor Klee 1973: How many guards are needed to cover a given art gallery (polygon) with  $n$  vertices?



# Motivation: The Art Gallery Problem



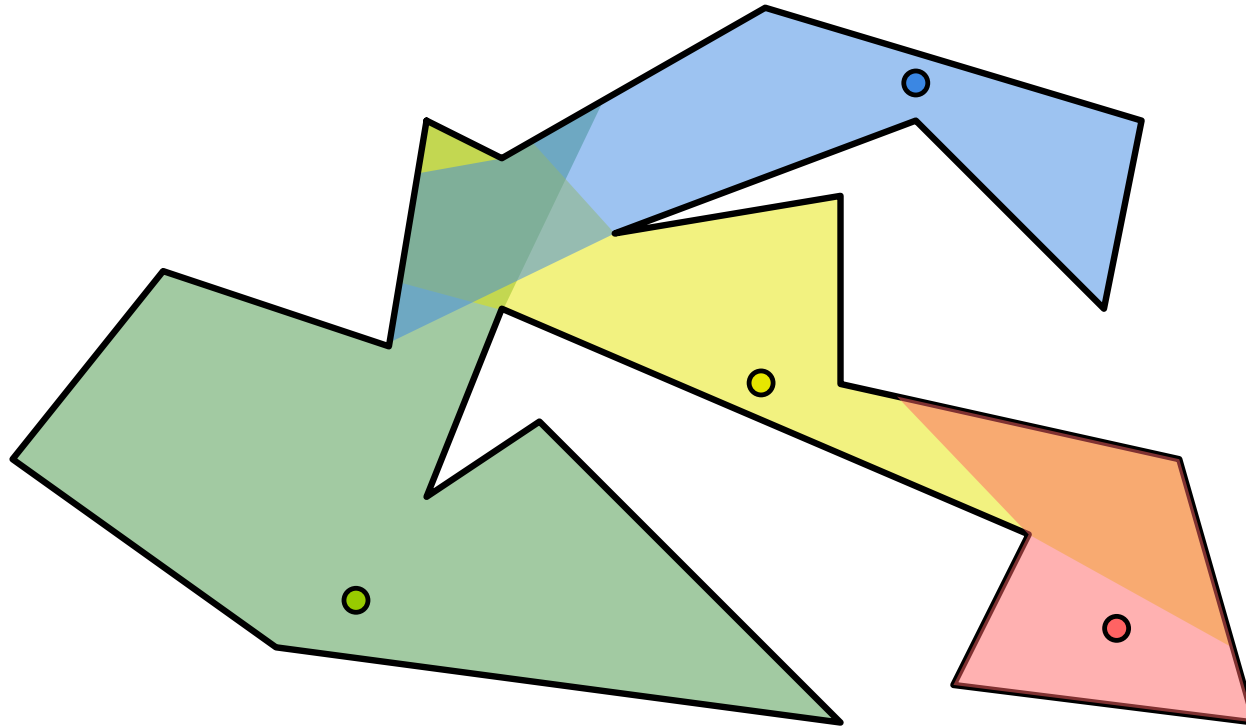
Viktor Klee 1973: How many guards are needed to cover a given art gallery (polygon) with  $n$  vertices?



# Motivation: The Art Gallery Problem



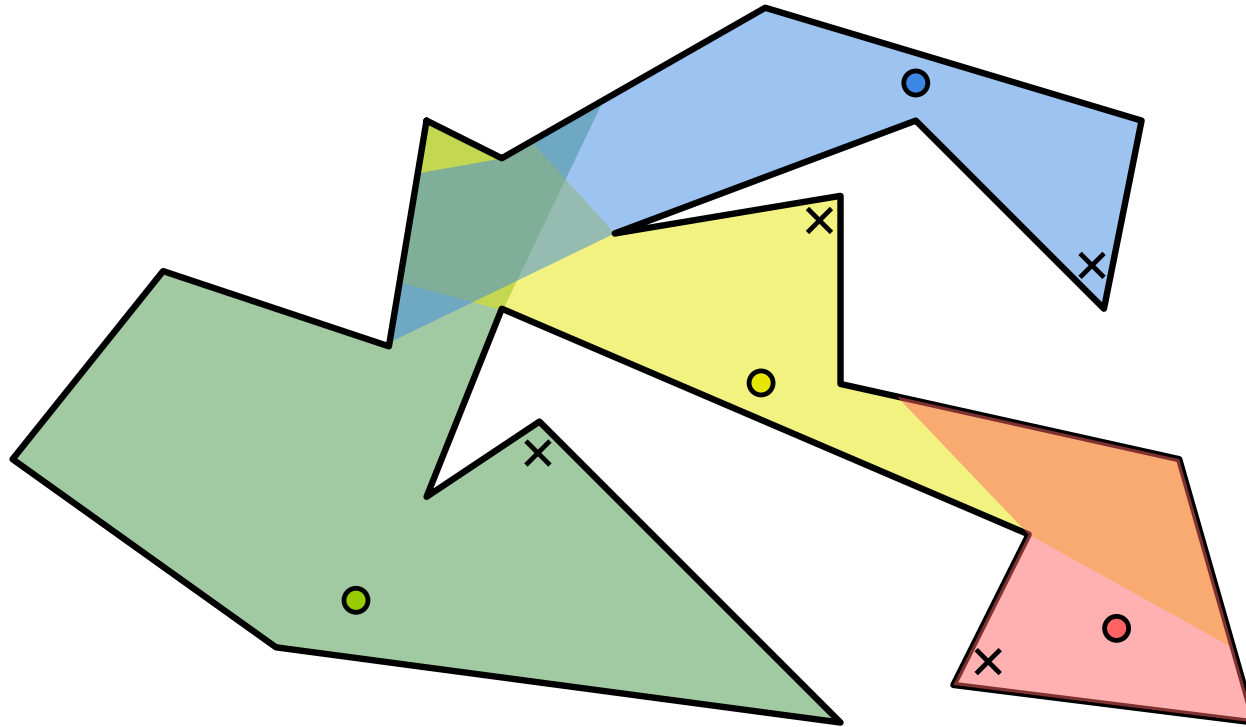
Viktor Klee 1973: How many guards are needed to cover a given art gallery (polygon) with  $n$  vertices?



# Motivation: The Art Gallery Problem



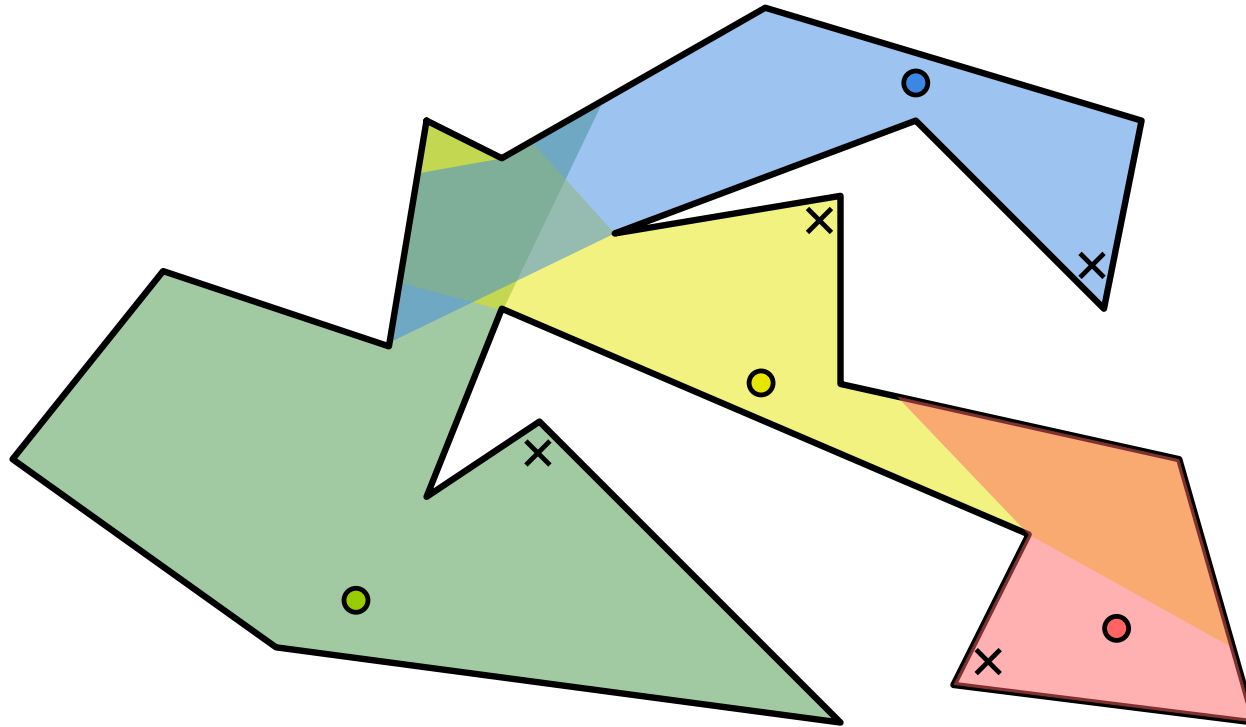
Viktor Klee 1973: How many guards are needed to cover a given art gallery (polygon) with  $n$  vertices?



# Motivation: The Art Gallery Problem



Viktor Klee 1973: How many guards are needed to cover a given art gallery (polygon) with  $n$  vertices?



**Input.** Array of points  $(x_1, y_1), \dots, (x_n, y_n)$ : the corners of the art gallery in cyclic order, where  $x_i, y_i \in \mathbb{Z}$ . An integer  $k$ .  
**Output.** Can we place  $k$  guards that see the entire art gallery?

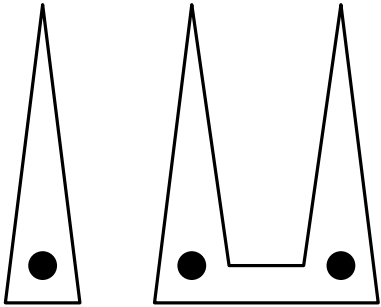
$\lfloor \frac{n}{3} \rfloor$  guards are sometimes needed

If  $n = 3k$ , then  $k$  guards can be needed!



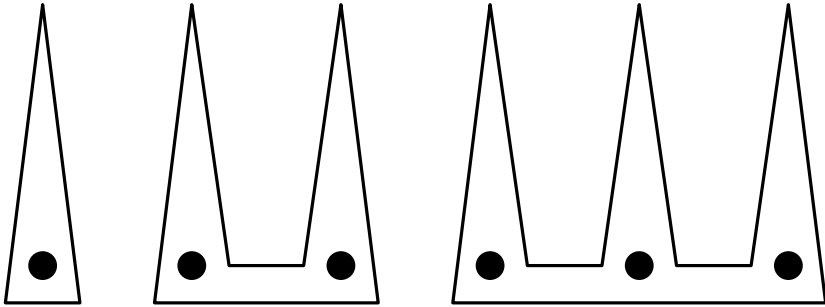
$\lfloor \frac{n}{3} \rfloor$  guards are sometimes needed

If  $n = 3k$ , then  $k$  guards can be needed!



$\lfloor \frac{n}{3} \rfloor$  guards are sometimes needed

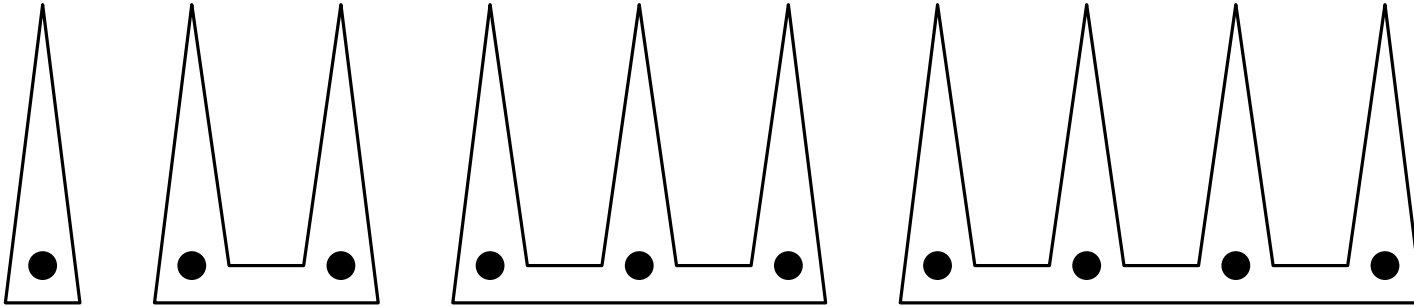
If  $n = 3k$ , then  $k$  guards can be needed!





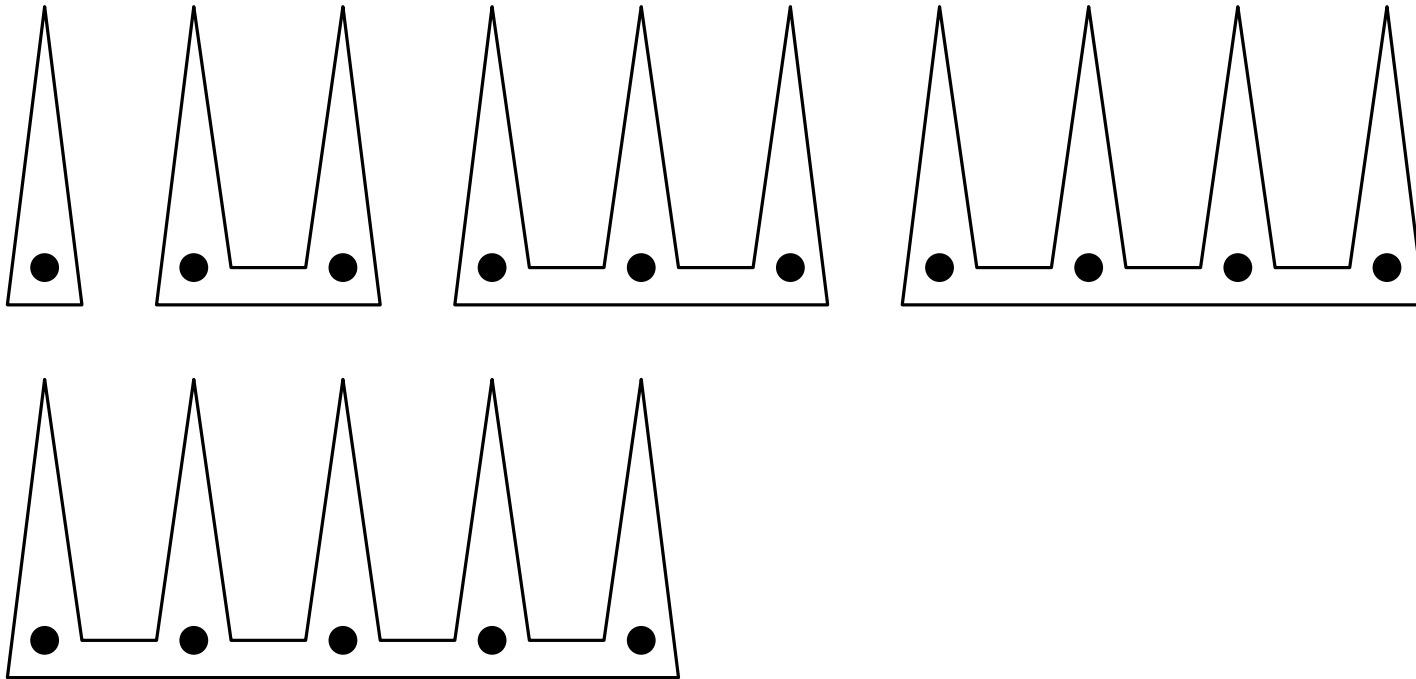
$\lfloor \frac{n}{3} \rfloor$  guards are sometimes needed

If  $n = 3k$ , then  $k$  guards can be needed!



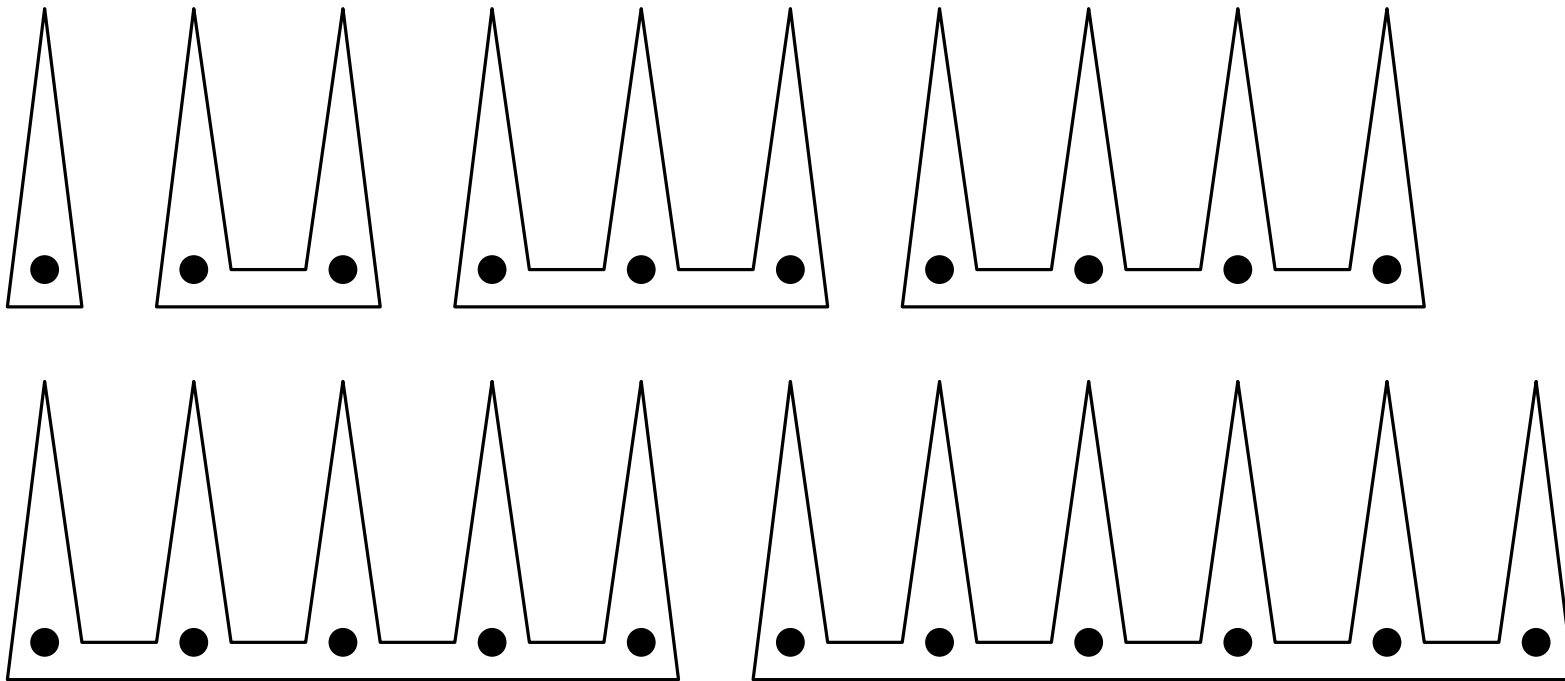
$\lfloor \frac{n}{3} \rfloor$  guards are sometimes needed

If  $n = 3k$ , then  $k$  guards can be needed!



$\lfloor \frac{n}{3} \rfloor$  guards are sometimes needed

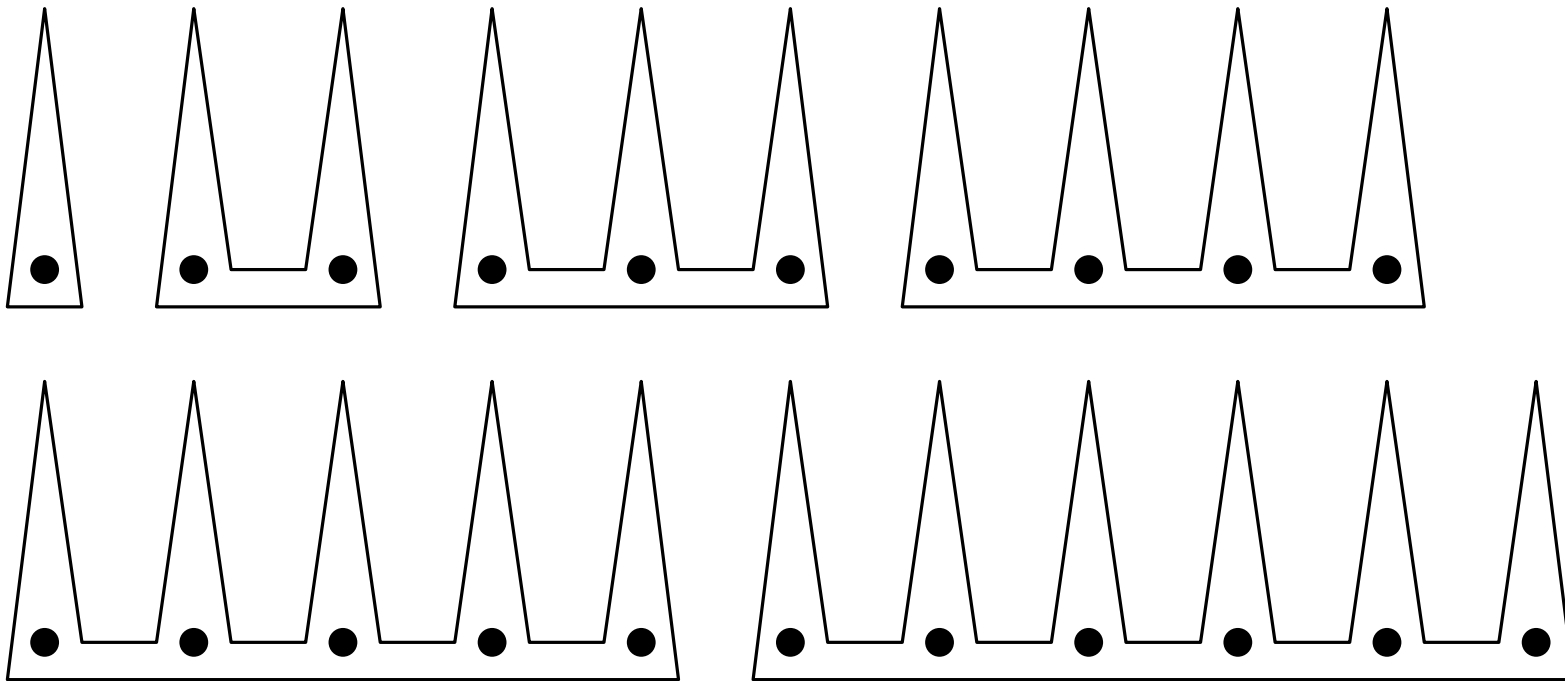
If  $n = 3k$ , then  $k$  guards can be needed!



Conclusion:  $\lfloor \frac{n}{3} \rfloor$  guards are sometimes needed.

$\lfloor \frac{n}{3} \rfloor$  guards are sometimes needed

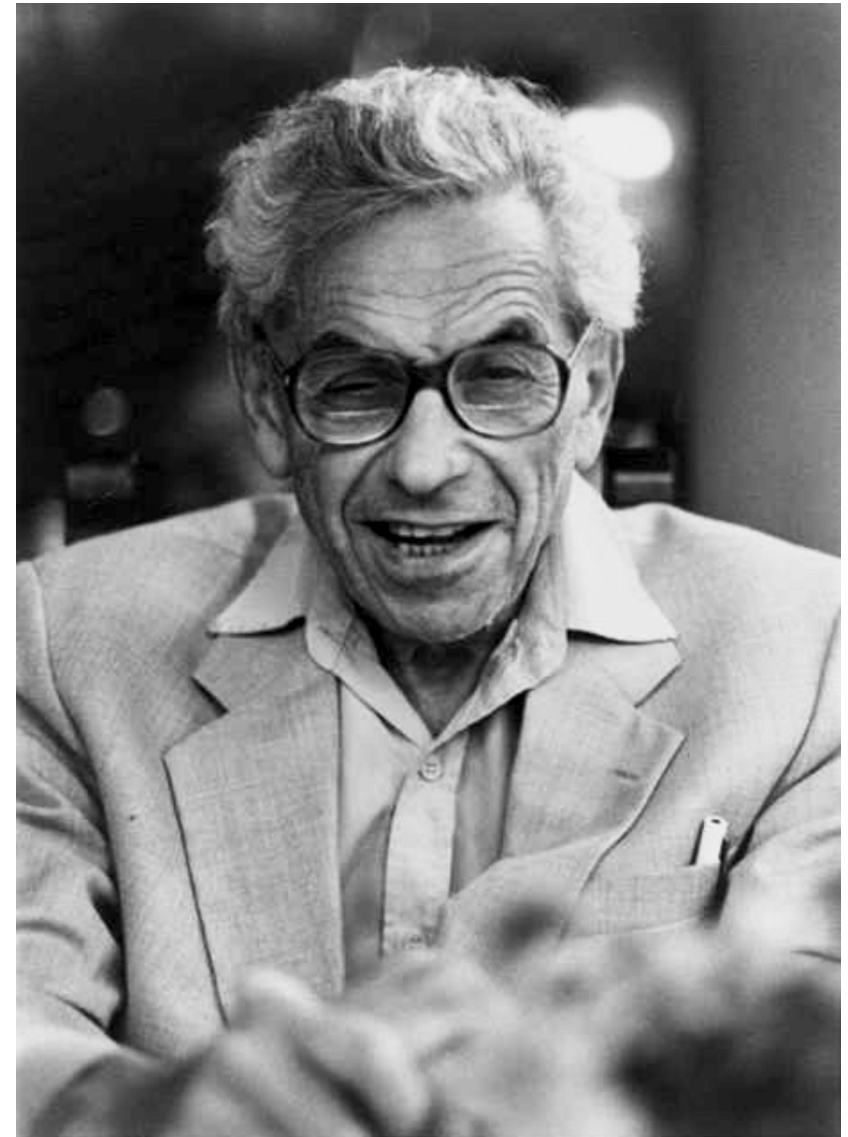
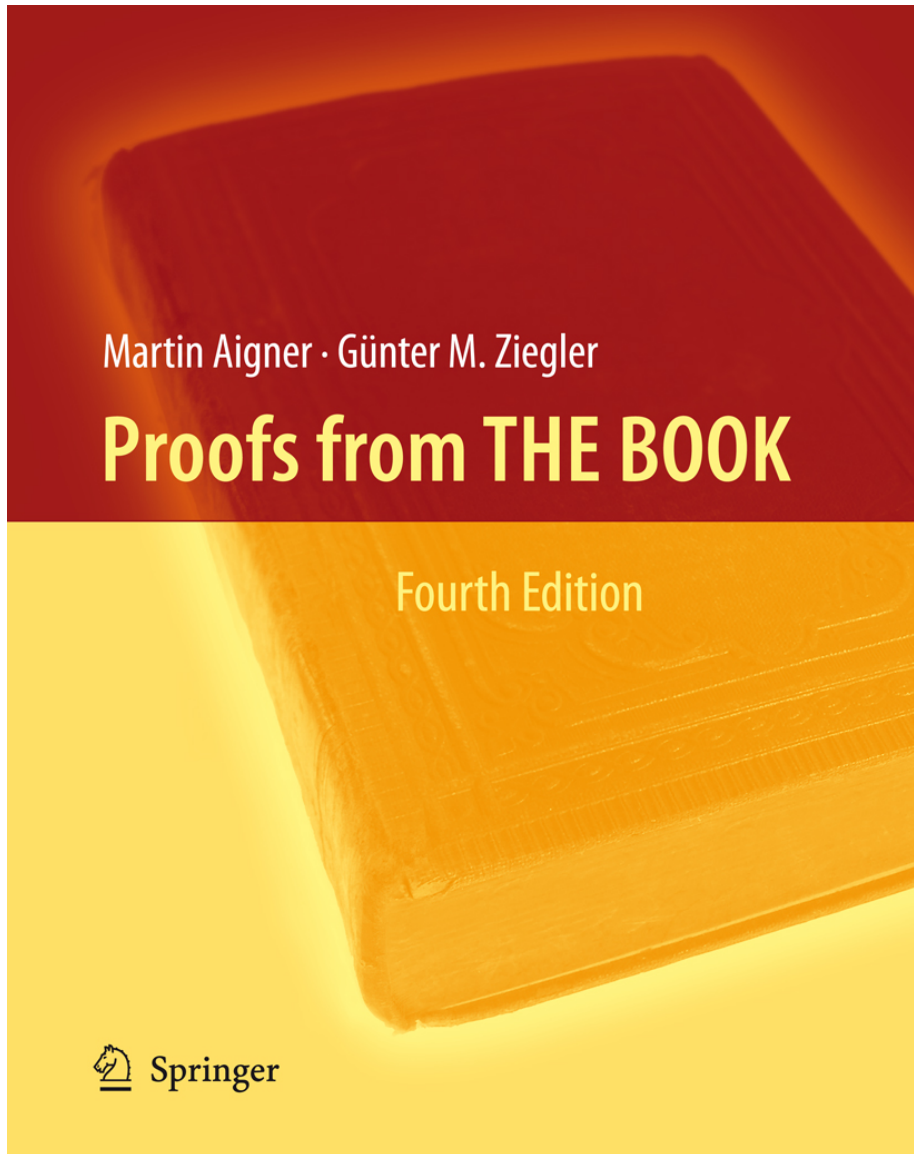
If  $n = 3k$ , then  $k$  guards can be needed!



Conclusion:  $\lfloor \frac{n}{3} \rfloor$  guards are sometimes needed.

Question: Is it always enough?

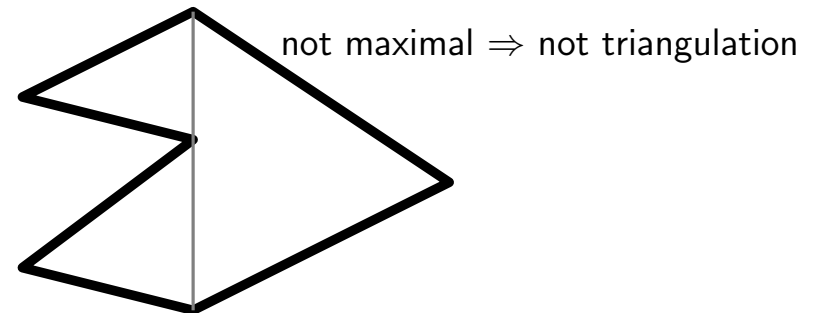
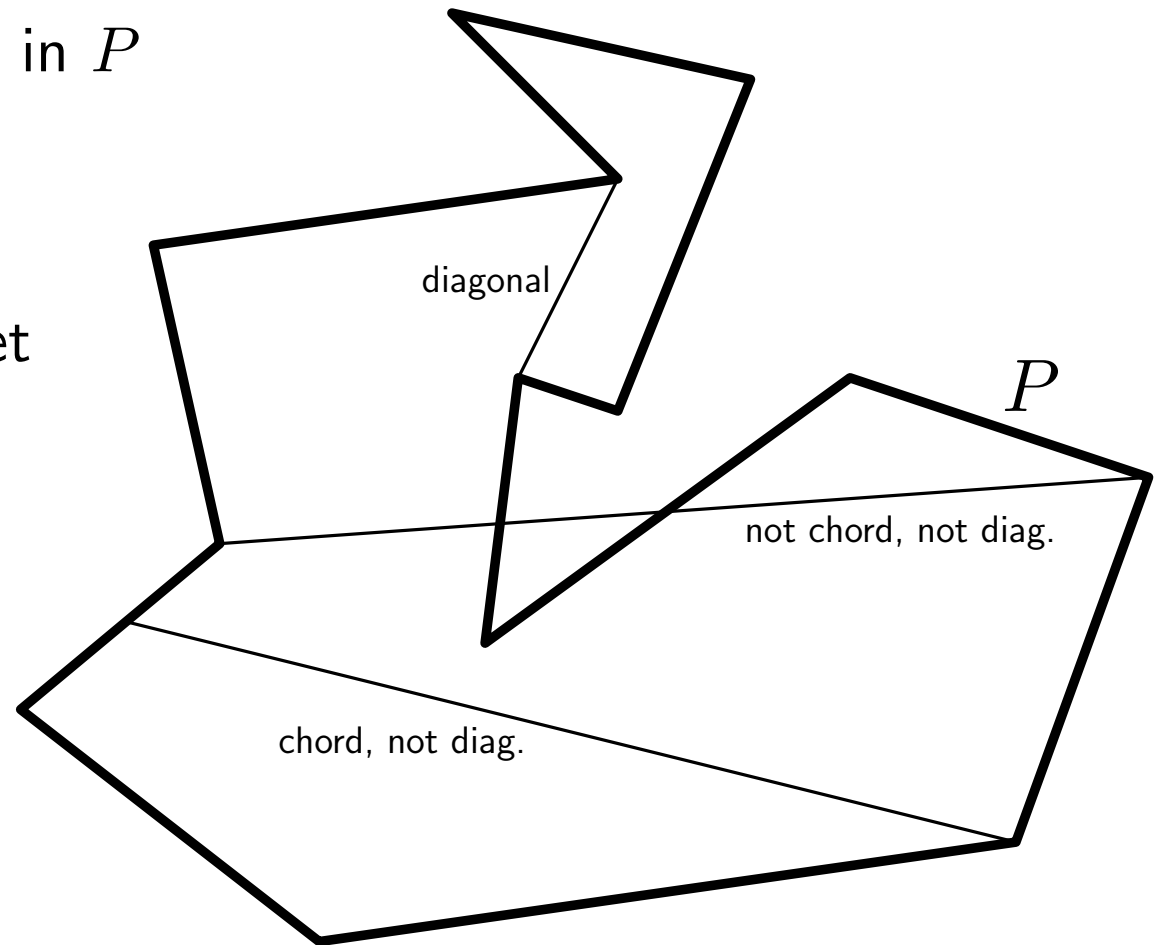
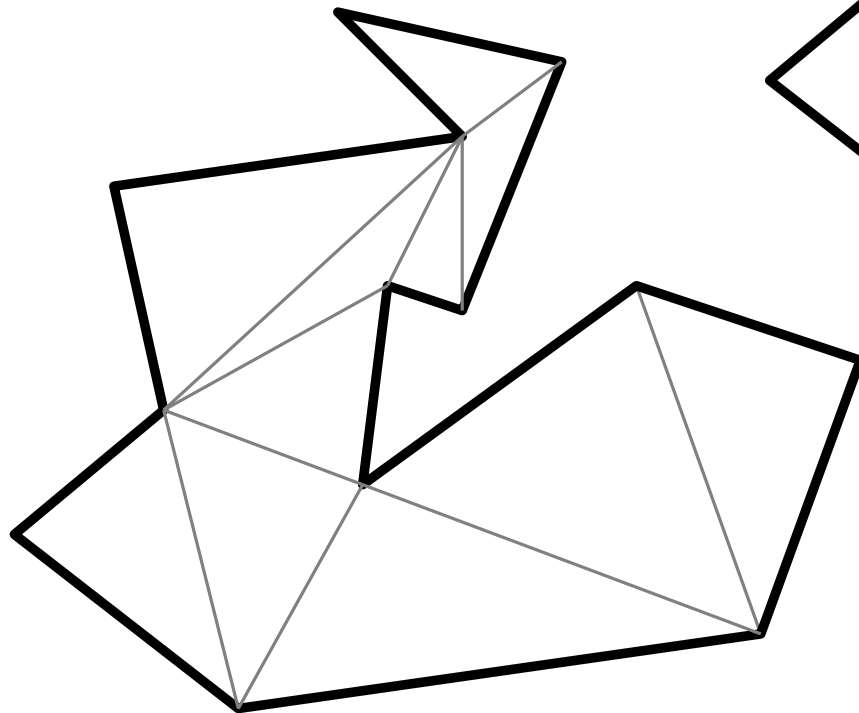
Proof that  $\lfloor \frac{n}{3} \rfloor$  are sufficient



# What is a triangulation?

*Diagonal:* Segment contained in  $P$  between two vertices.

*Triangulation:* Partition of  $P$  into triangles by a maximal set of non-intersecting diagonals.



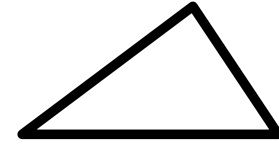
# Any polygon can be triangulated

**Lemma:** A polygon  $P$  with  $n$  vertices can be triangulated, and any triangulation has  $n - 2$  triangles using  $n - 3$  diagonals.

# Any polygon can be triangulated

**Lemma:** A polygon  $P$  with  $n$  vertices can be triangulated, and any triangulation has  $n - 2$  triangles using  $n - 3$  diagonals.

**Proof:** Induction on  $n$ . Base case  $n = 3$  is trivial.



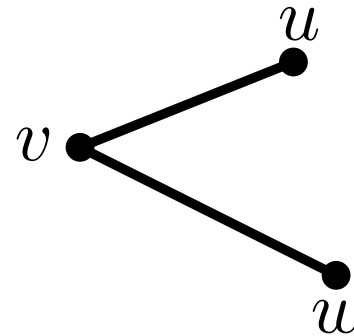
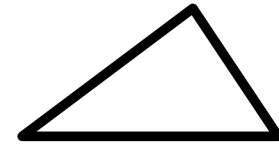


# Any polygon can be triangulated

**Lemma:** A polygon  $P$  with  $n$  vertices can be triangulated, and any triangulation has  $n - 2$  triangles using  $n - 3$  diagonals.

**Proof:** Induction on  $n$ . Base case  $n = 3$  is trivial.

Induction step:  $v$  leftmost vertex,  $u$  and  $w$  neighbours.



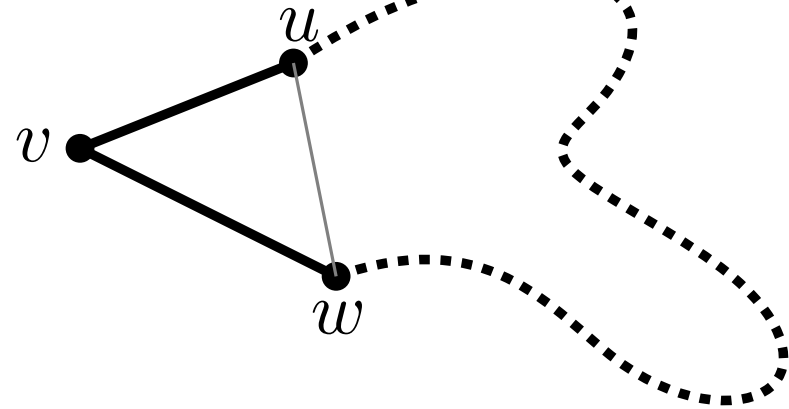
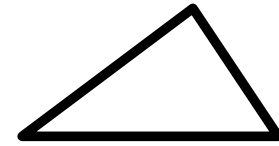
# Any polygon can be triangulated

**Lemma:** A polygon  $P$  with  $n$  vertices can be triangulated, and any triangulation has  $n - 2$  triangles using  $n - 3$  diagonals.

**Proof:** Induction on  $n$ . Base case  $n = 3$  is trivial.

Induction step:  $v$  leftmost vertex,  $u$  and  $w$  neighbours.

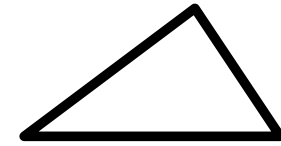
**Case 1:**  $uw$  is a diagonal.



# Any polygon can be triangulated

**Lemma:** A polygon  $P$  with  $n$  vertices can be triangulated, and any triangulation has  $n - 2$  triangles using  $n - 3$  diagonals.

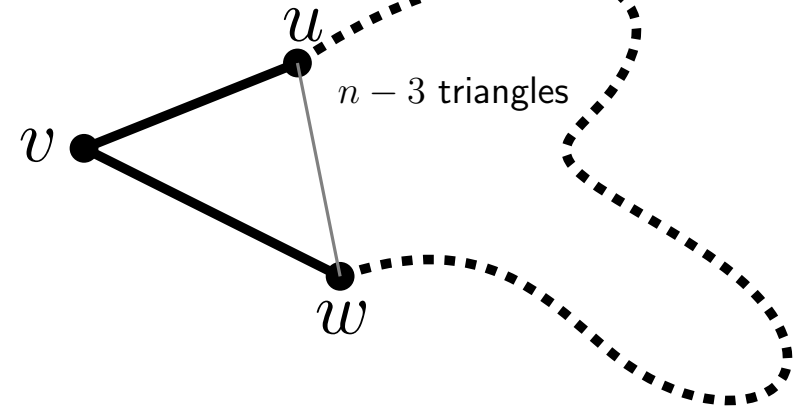
**Proof:** Induction on  $n$ . Base case  $n = 3$  is trivial.



Induction step:  $v$  leftmost vertex,  $u$  and  $w$  neighbours.

**Case 1:**  $uw$  is a diagonal.

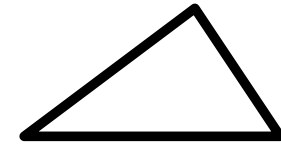
Induction hypothesis  $\Rightarrow$   
triangulation with  $n - 3$  triangles on  
the other side of  $uw$ .



# Any polygon can be triangulated

**Lemma:** A polygon  $P$  with  $n$  vertices can be triangulated, and any triangulation has  $n - 2$  triangles using  $n - 3$  diagonals.

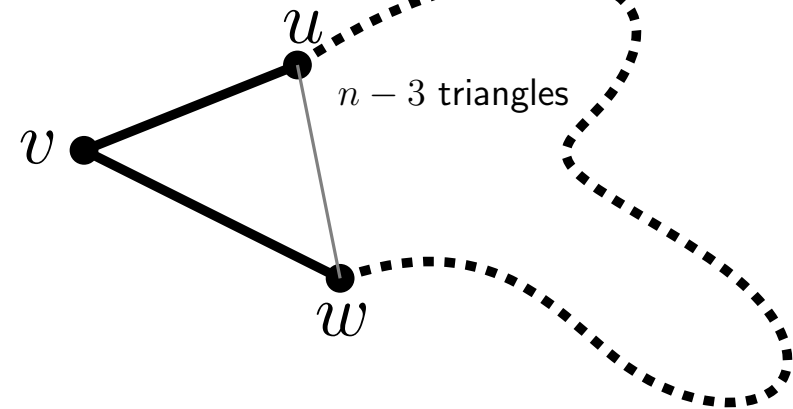
**Proof:** Induction on  $n$ . Base case  $n = 3$  is trivial.



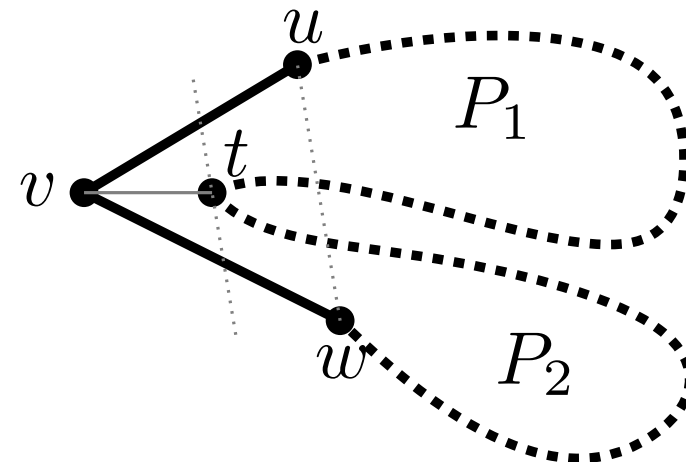
Induction step:  $v$  leftmost vertex,  $u$  and  $w$  neighbours.

**Case 1:**  $uw$  is a diagonal.

Induction hypothesis  $\Rightarrow$   
triangulation with  $n - 3$  triangles on  
the other side of  $uw$ .



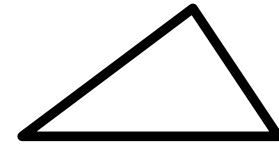
**Case 2:**  $uw$  is no diagonal. Let  $t$  be  
corner in  $uvw$  farthest from  $uw$ .  $vt$  is  
a diagonal, splits  $P$  into  $P_1$  and  $P_2$   
with  $m_1 < n$  and  $m_2 < n$  vertices,  
 $m_1 + m_2 = n + 2$ .



# Any polygon can be triangulated

**Lemma:** A polygon  $P$  with  $n$  vertices can be triangulated, and any triangulation has  $n - 2$  triangles using  $n - 3$  diagonals.

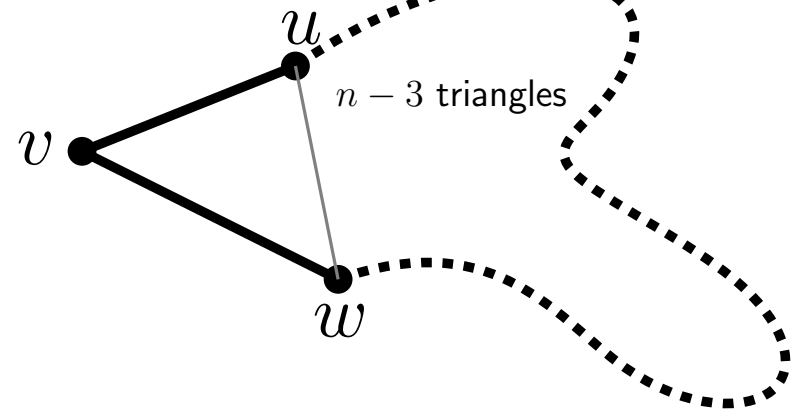
**Proof:** Induction on  $n$ . Base case  $n = 3$  is trivial.



Induction step:  $v$  leftmost vertex,  $u$  and  $w$  neighbours.

**Case 1:**  $uw$  is a diagonal.

Induction hypothesis  $\Rightarrow$   
triangulation with  $n - 3$  triangles on  
the other side of  $uw$ .



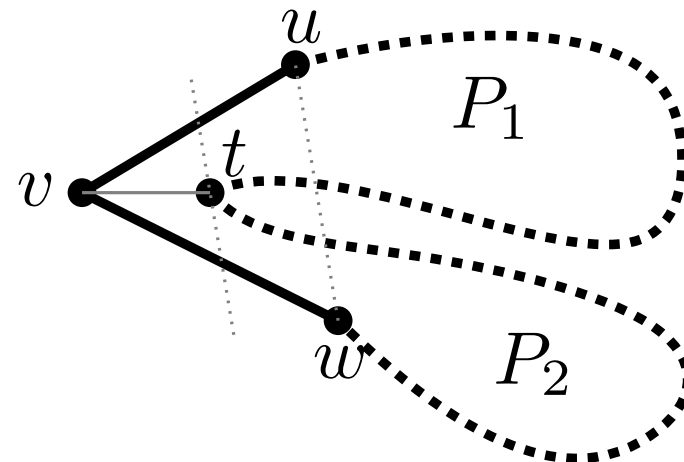
**Case 2:**  $uw$  is no diagonal. Let  $t$  be  
corner in  $uvw$  farthest from  $uw$ .  $vt$  is  
a diagonal, splits  $P$  into  $P_1$  and  $P_2$   
with  $m_1 < n$  and  $m_2 < n$  vertices,  
 $m_1 + m_2 = n + 2$ .

Induction hypothesis  $\Rightarrow$

$P_1$ :  $m_1 - 2$  triangles.

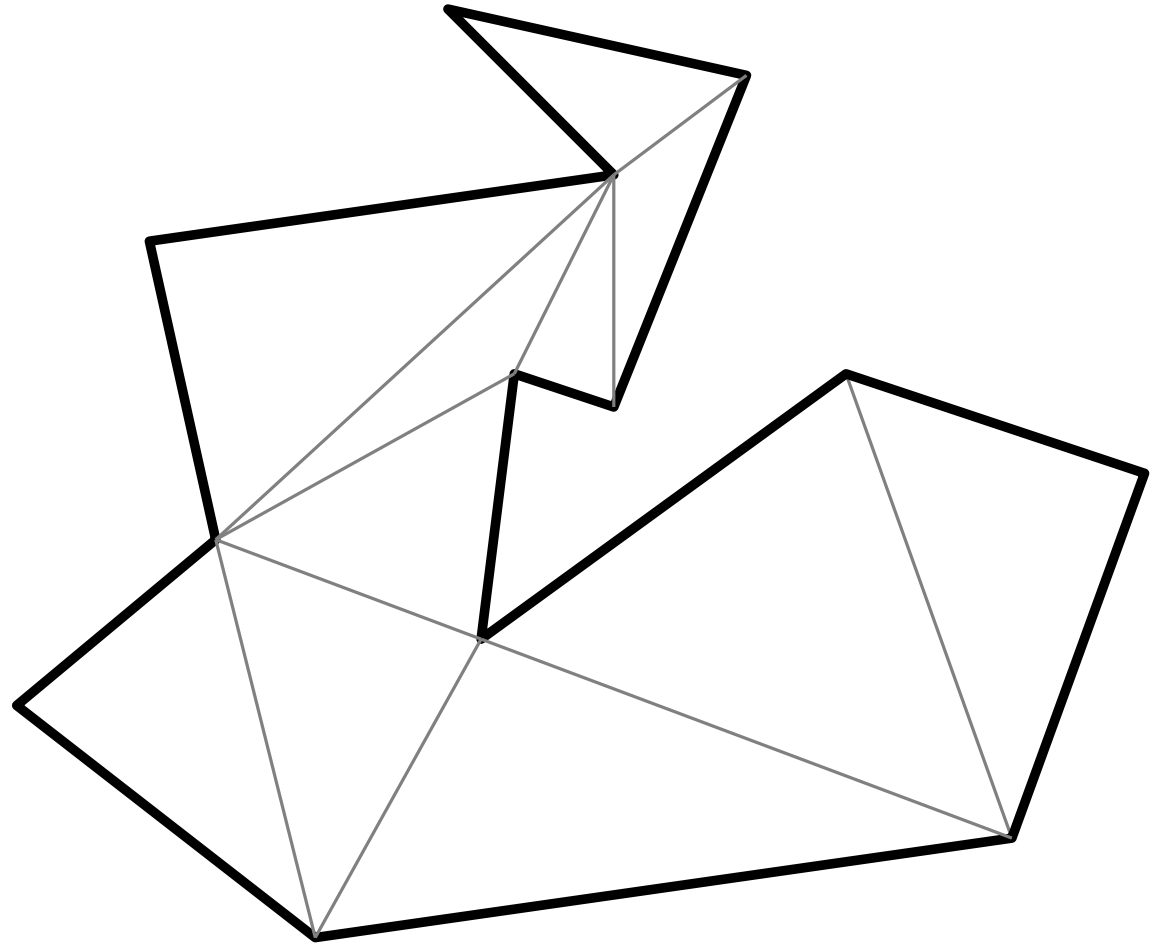
$P_2$ :  $m_2 - 2$  triangles.

In total:  $m_1 + m_2 - 4 = n + 2 - 4 = n - 2$ .



# Proof that $\lfloor \frac{n}{3} \rfloor$ are sufficient

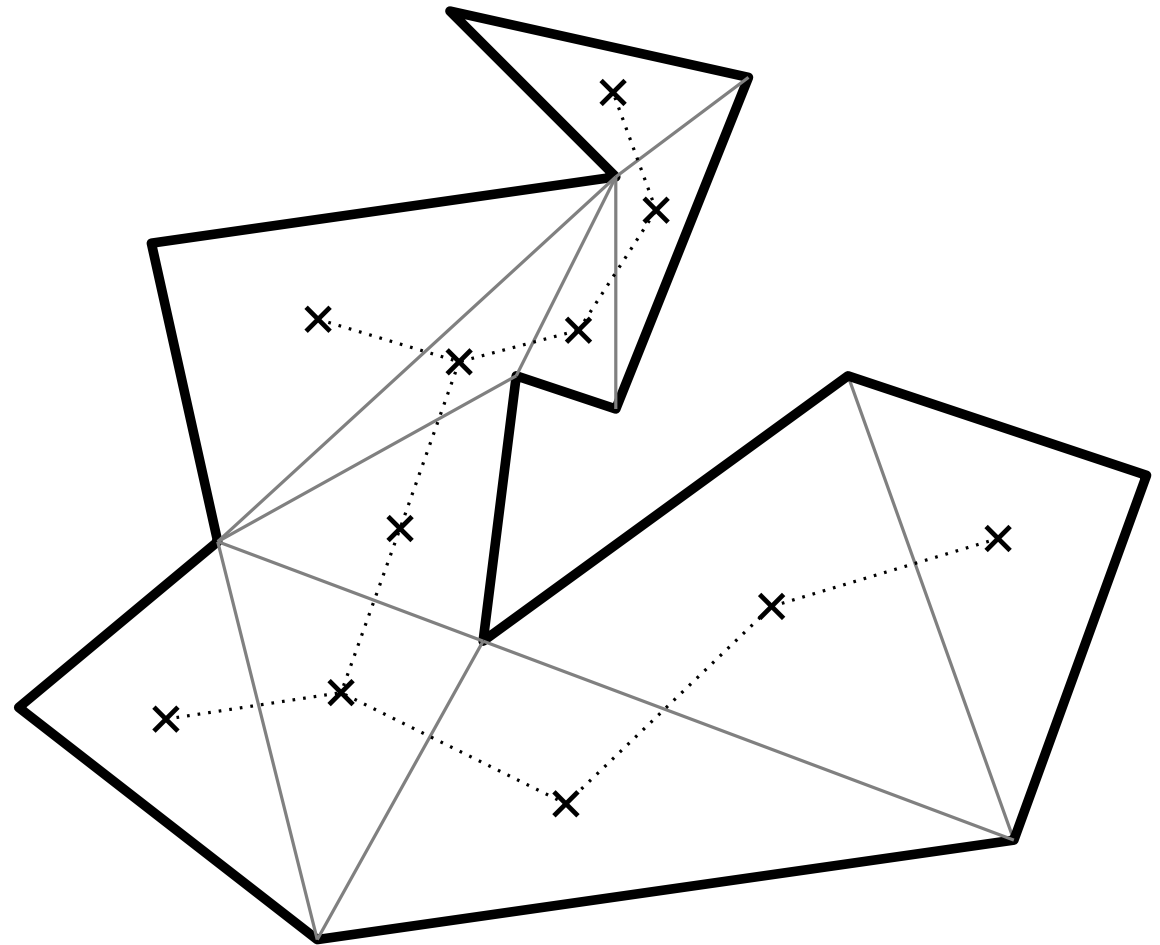
Consider a triangulation.



# Proof that $\lfloor \frac{n}{3} \rfloor$ are sufficient

Consider a triangulation.

Consider dual tree.

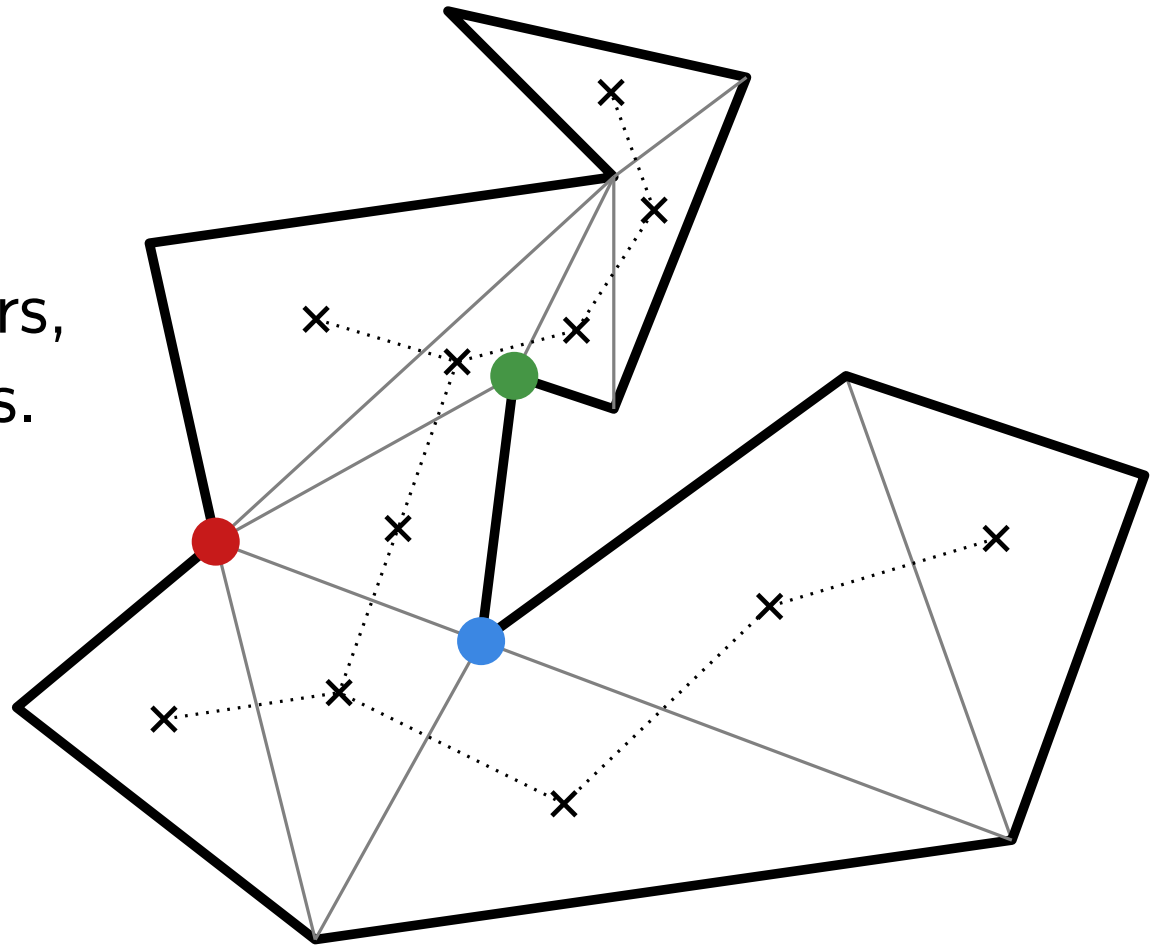


# Proof that $\lfloor \frac{n}{3} \rfloor$ are sufficient

Consider a triangulation.

Consider dual tree.

Color vertices with 3 colors,  
each triangle gets 3 colors.



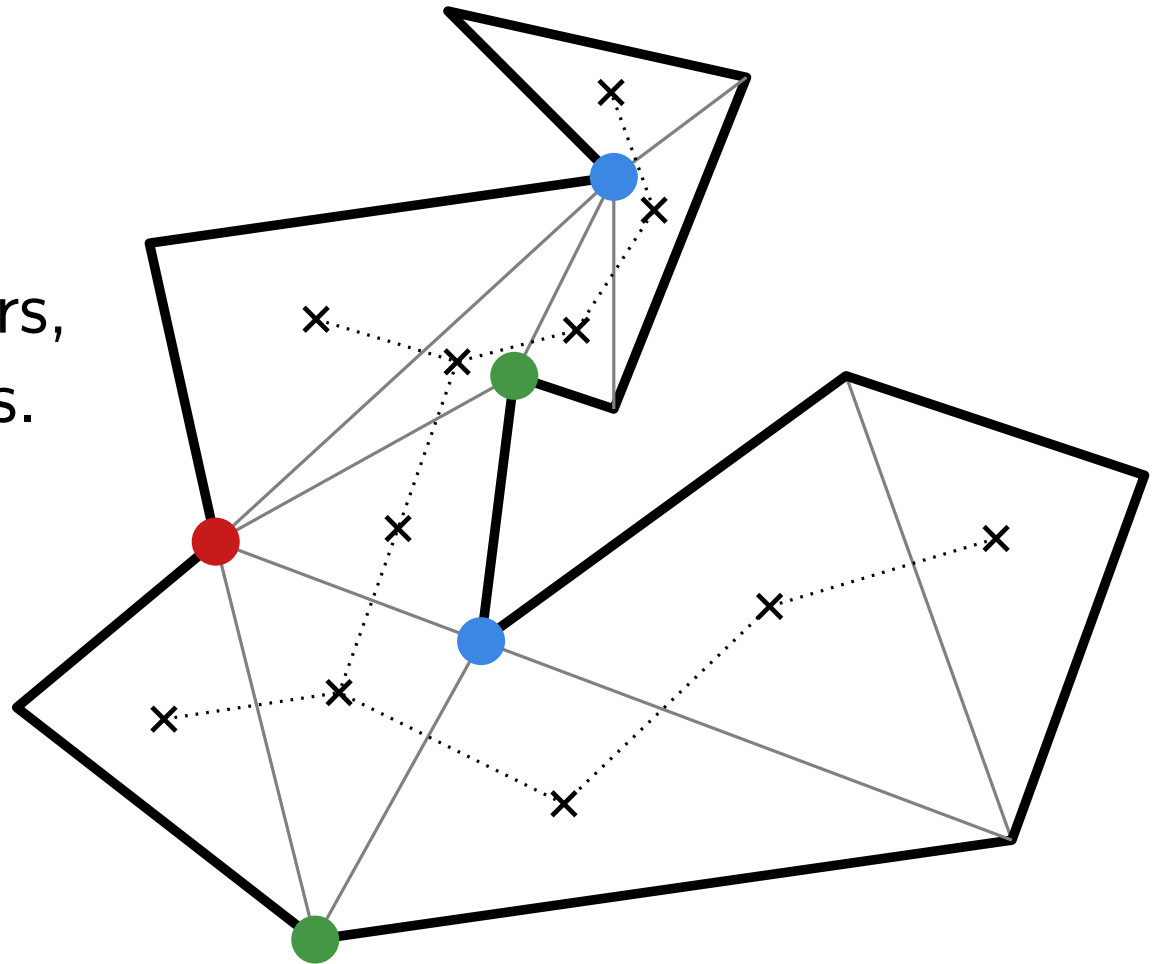


# Proof that $\lfloor \frac{n}{3} \rfloor$ are sufficient

Consider a triangulation.

Consider dual tree.

Color vertices with 3 colors,  
each triangle gets 3 colors.

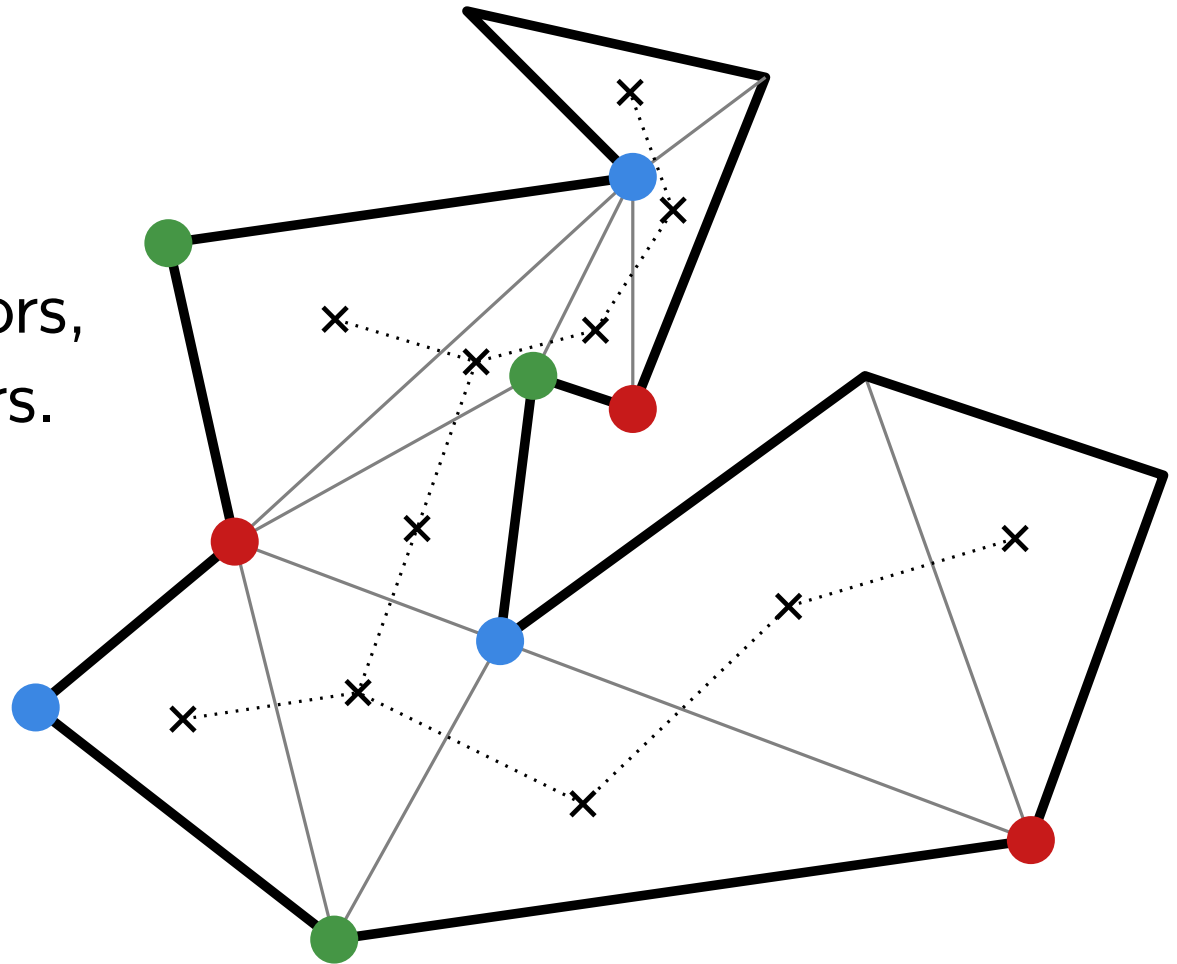


# Proof that $\lfloor \frac{n}{3} \rfloor$ are sufficient

Consider a triangulation.

Consider dual tree.

Color vertices with 3 colors,  
each triangle gets 3 colors.

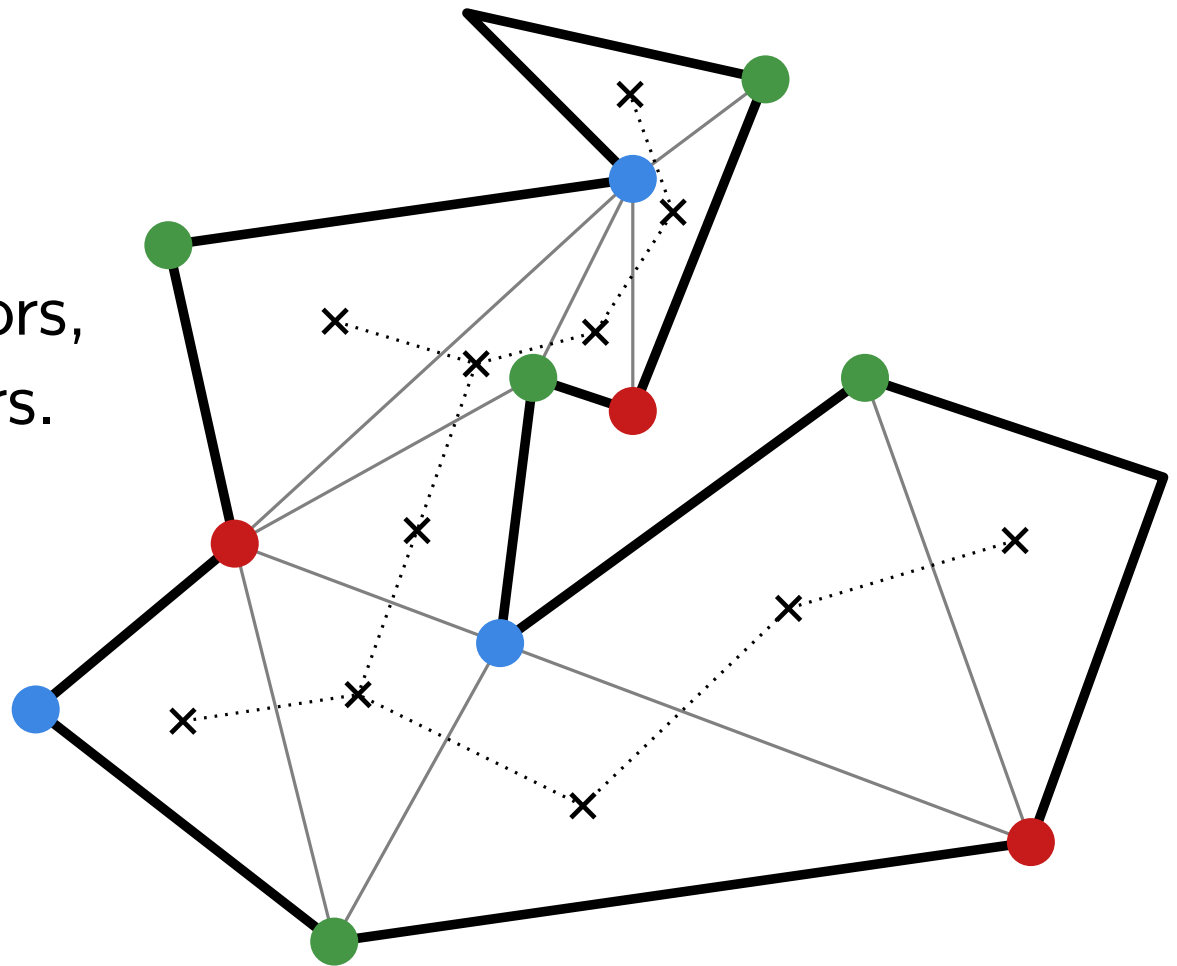


# Proof that $\lfloor \frac{n}{3} \rfloor$ are sufficient

Consider a triangulation.

Consider dual tree.

Color vertices with 3 colors,  
each triangle gets 3 colors.

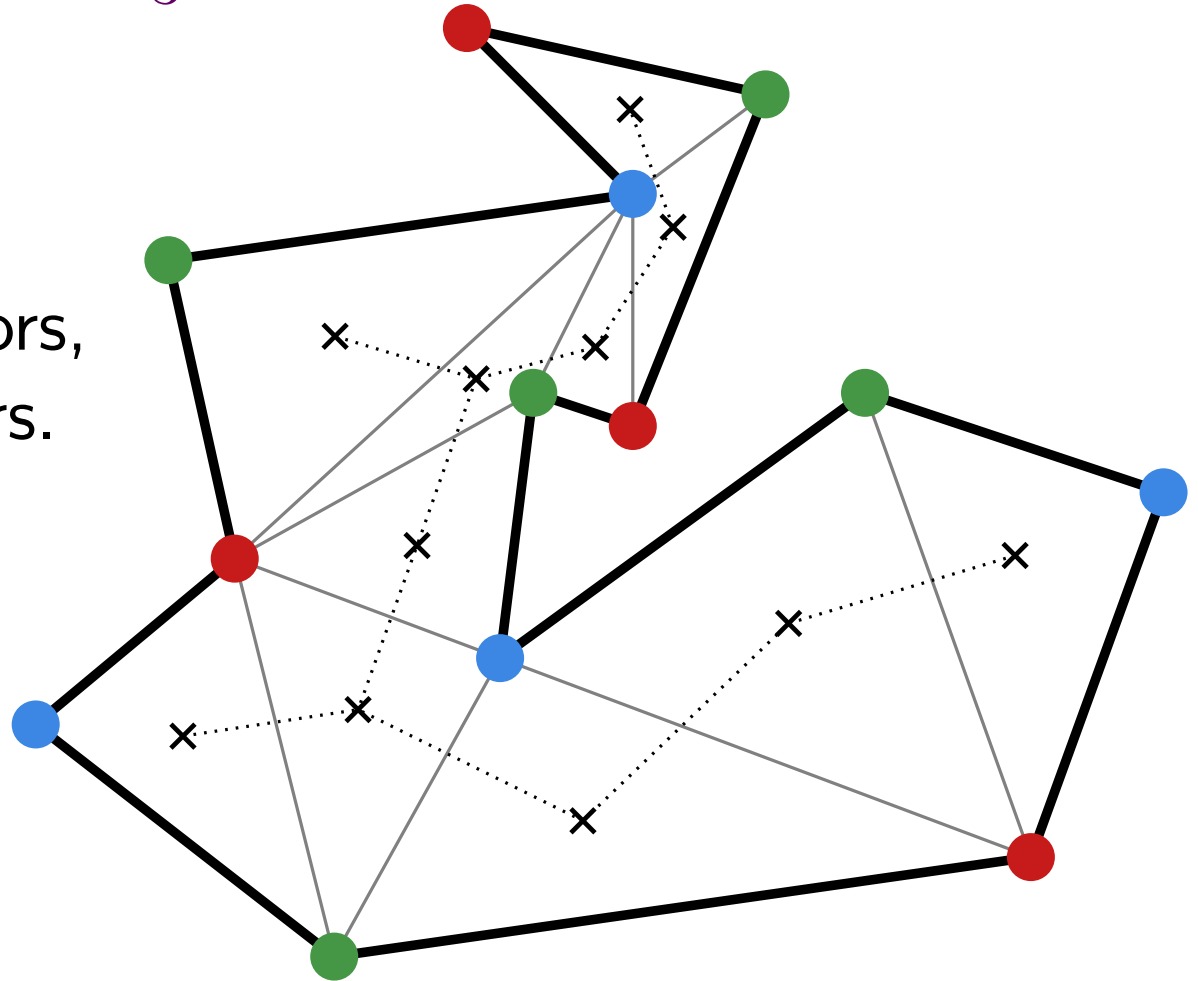


# Proof that $\lfloor \frac{n}{3} \rfloor$ are sufficient

Consider a triangulation.

Consider dual tree.

Color vertices with 3 colors,  
each triangle gets 3 colors.



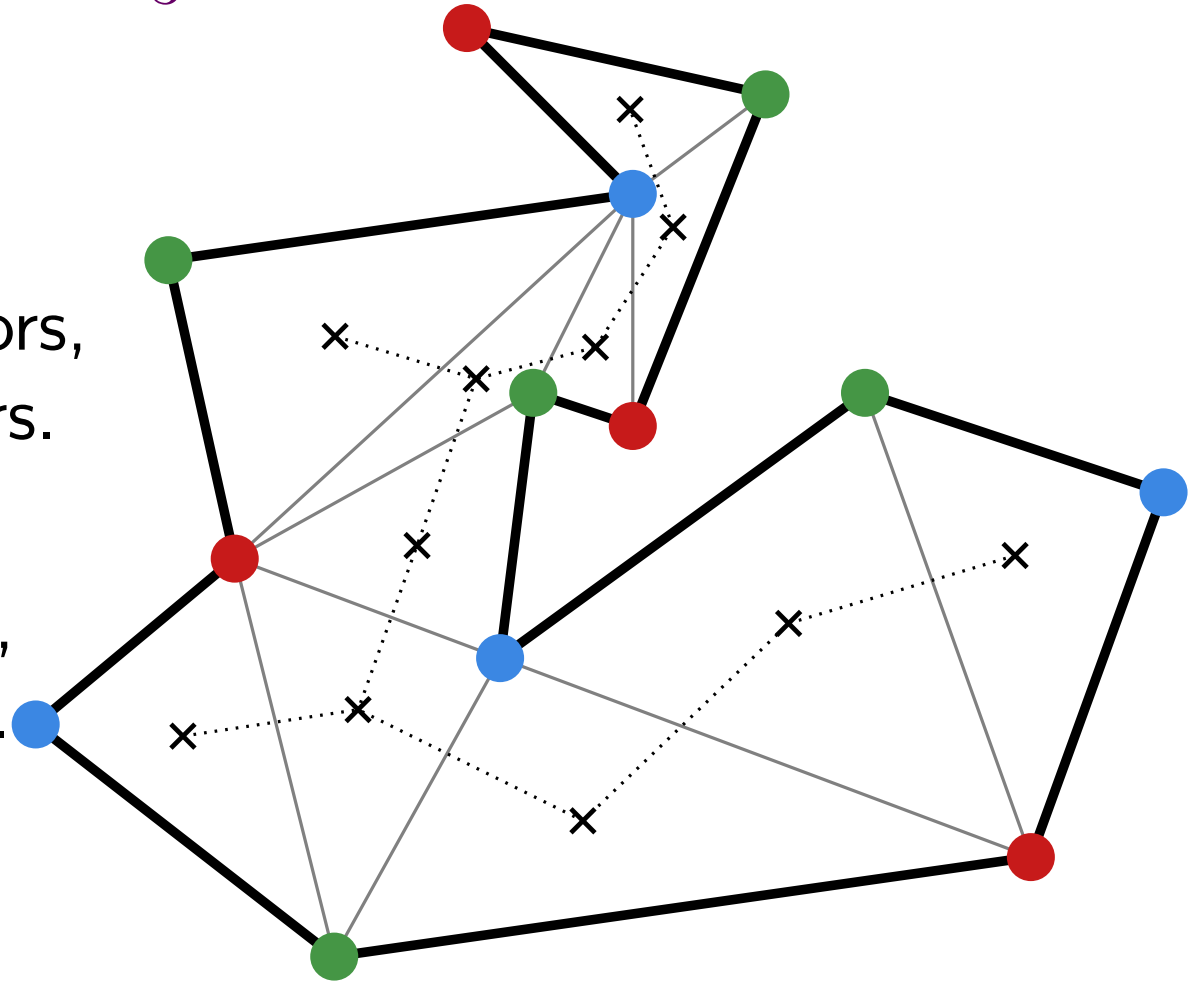
# Proof that $\lfloor \frac{n}{3} \rfloor$ are sufficient

Consider a triangulation.

Consider dual tree.

Color vertices with 3 colors,  
each triangle gets 3 colors.

Observe that the **red**  
vertices guard the gallery,  
as do the **green** and **blue**.



# Proof that $\lfloor \frac{n}{3} \rfloor$ are sufficient

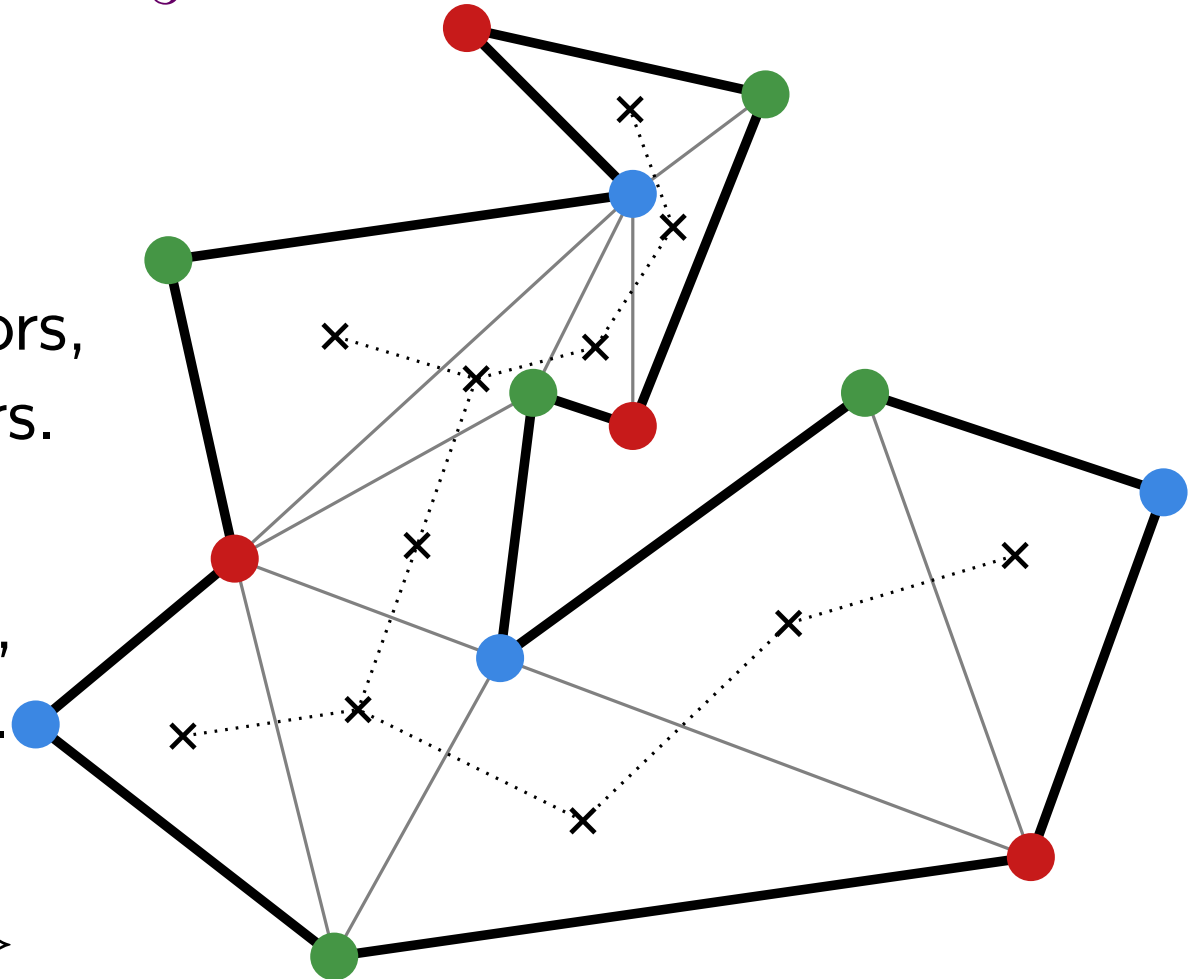
Consider a triangulation.

Consider dual tree.

Color vertices with 3 colors,  
each triangle gets 3 colors.

Observe that the **red**  
vertices guard the gallery,  
as do the **green** and **blue**.

$$\begin{aligned} n &= n_r + n_g + n_b \implies \\ \min\{n_r, n_g, n_b\} &\leq \frac{n}{3} \implies \\ \min\{n_r, n_g, n_b\} &\leq \left\lfloor \frac{n}{3} \right\rfloor \end{aligned}$$



# Proof that $\lfloor \frac{n}{3} \rfloor$ are sufficient

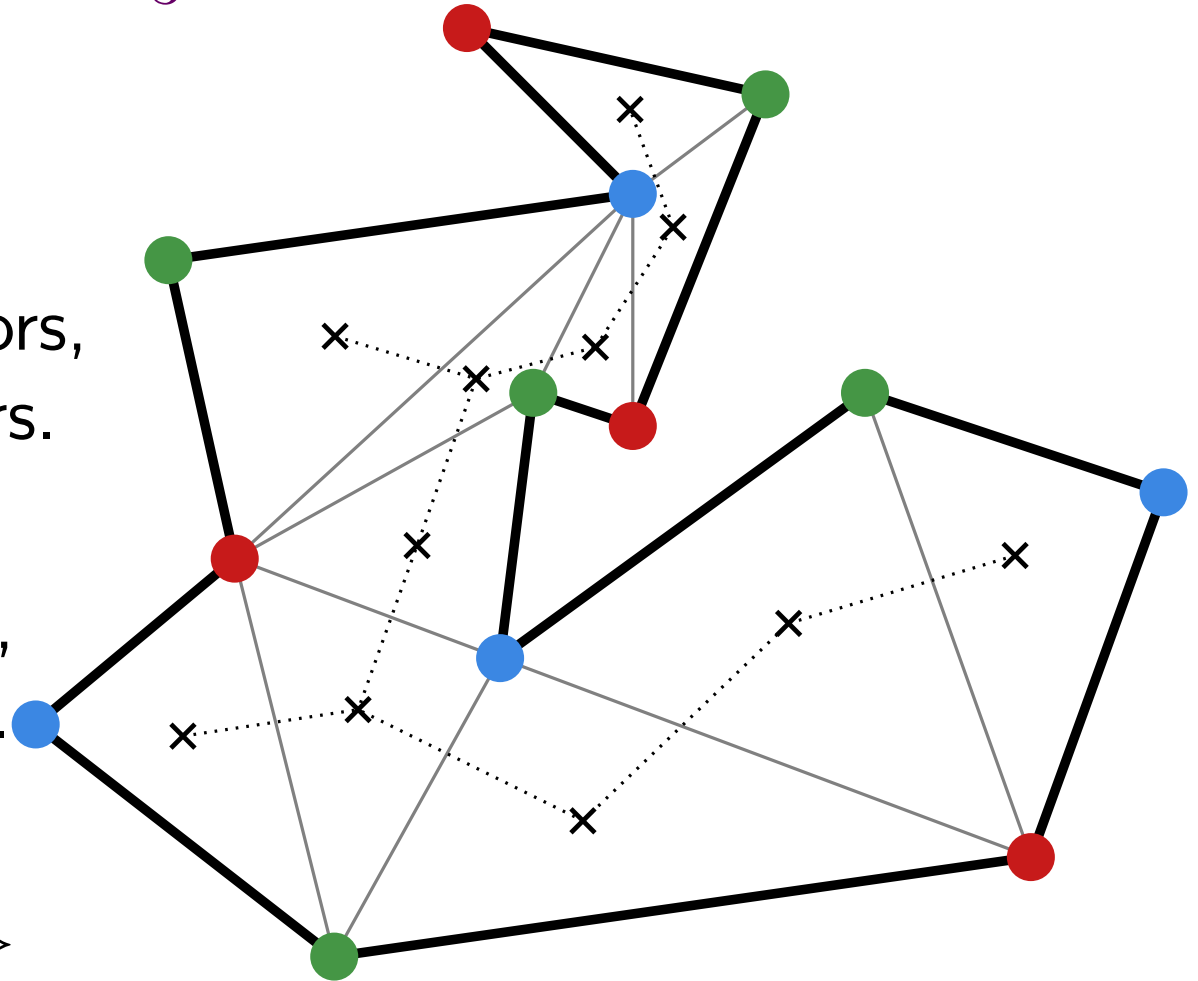
Consider a triangulation.

Consider dual tree.

Color vertices with 3 colors,  
each triangle gets 3 colors.

Observe that the **red**  
vertices guard the gallery,  
as do the **green** and **blue**.

$$\begin{aligned} n &= n_r + n_g + n_b \implies \\ \min\{n_r, n_g, n_b\} &\leq \frac{n}{3} \implies \\ \min\{n_r, n_g, n_b\} &\leq \left\lfloor \frac{n}{3} \right\rfloor \end{aligned}$$



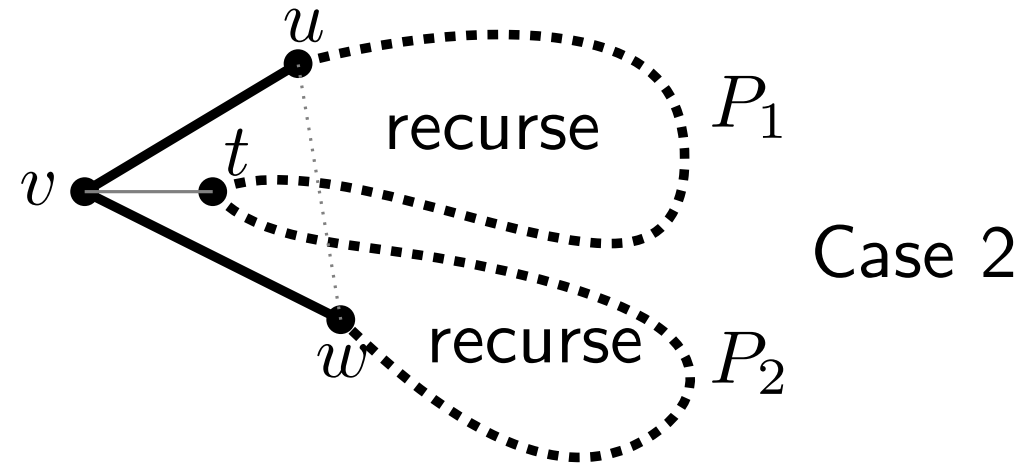
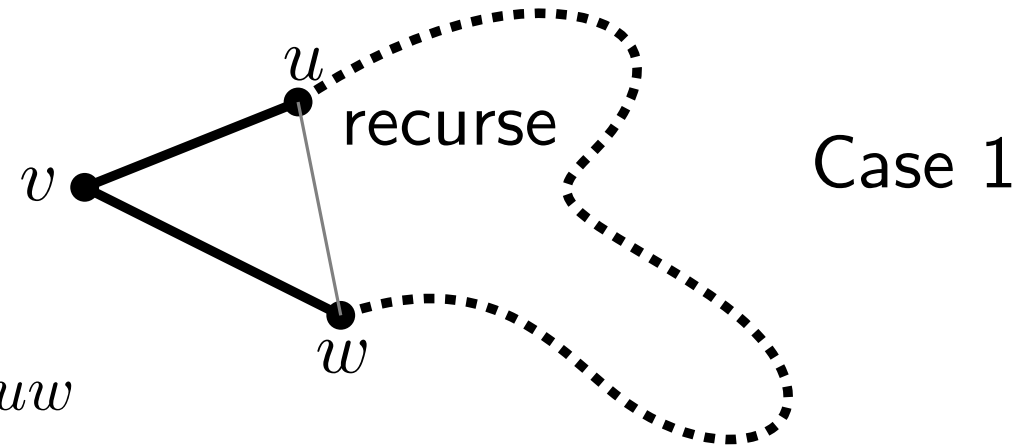
Why does the proof fail if  $P$  has holes?

# Simple triangulation algorithm

Use proof that a triangulation exists!

triangulate polygon  $P$

1. find points  $u, v, w$
2. if  $uw$  is a diagonal // case 1
3.     add diagonal  $uw$
4.     recurse on the other side of  $uw$
5. else // case 2
6.     find point  $t$
7.     add diagonal  $vt$
8.     recurse on  $P_1$  and  $P_2$



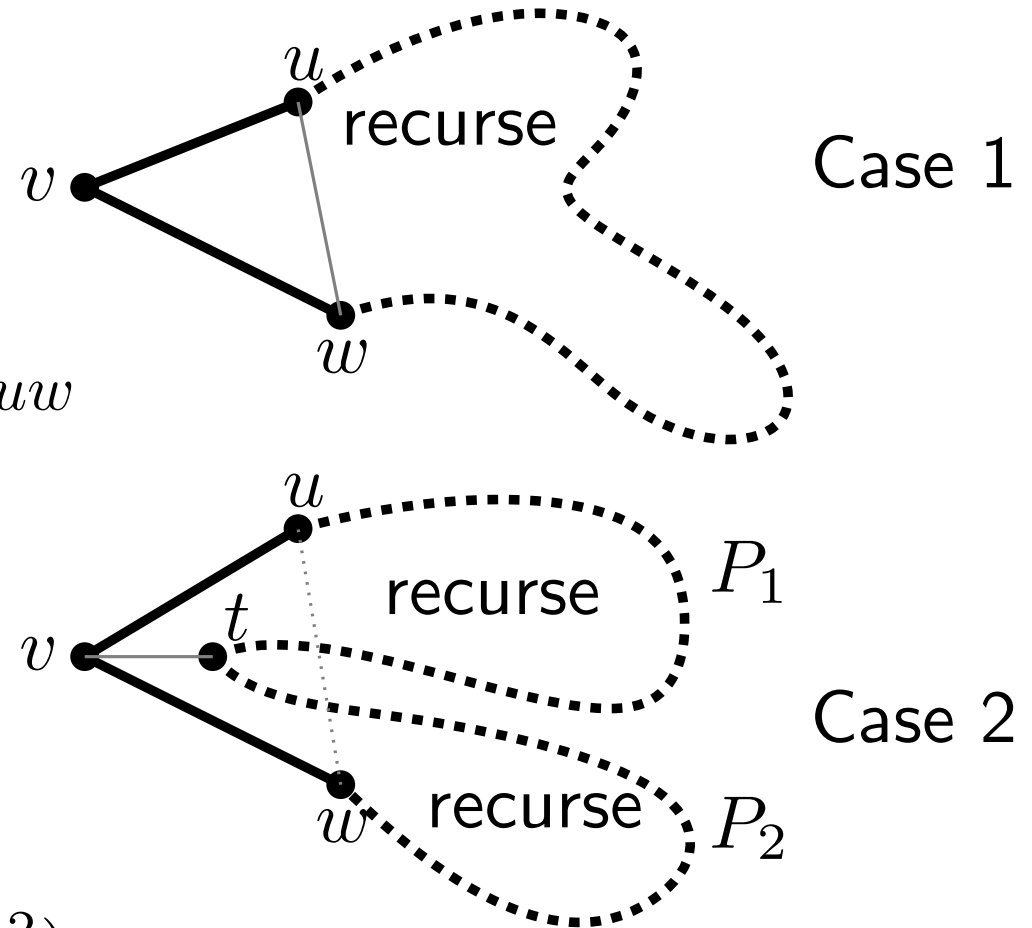


# Simple triangulation algorithm

Use proof that a triangulation exists!

triangulate polygon  $P$

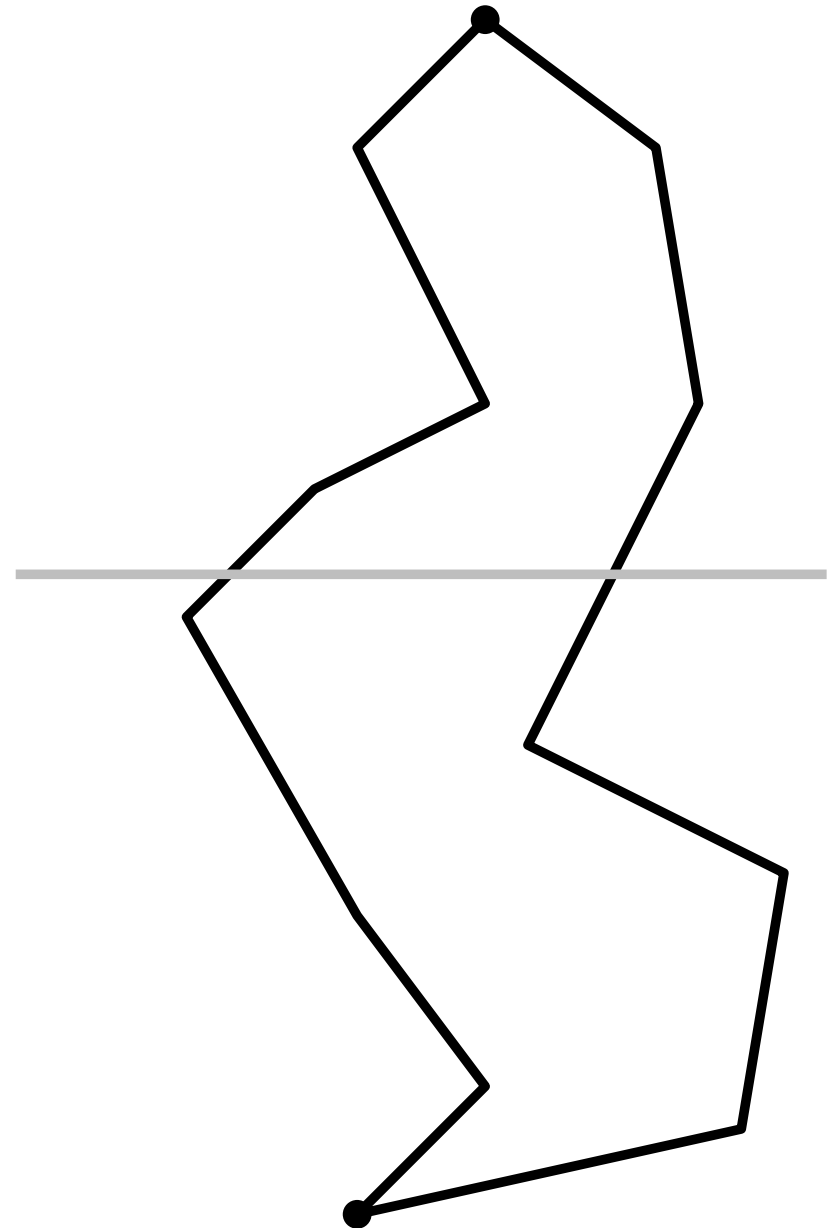
1. find points  $u, v, w$
2. if  $uw$  is a diagonal // case 1
3.     add diagonal  $uw$
4.     recurse on the other side of  $uw$
5. else // case 2
6.     find point  $t$
7.     add diagonal  $vt$
8.     recurse on  $P_1$  and  $P_2$



Worst-case running time:  $O(n^2)$ .

# Faster algorithm

$y$ -monotone polygon: Intersection with any horizontal line is connected.  
Equivalent: One vertex with both edges going down and one with both edges going up.



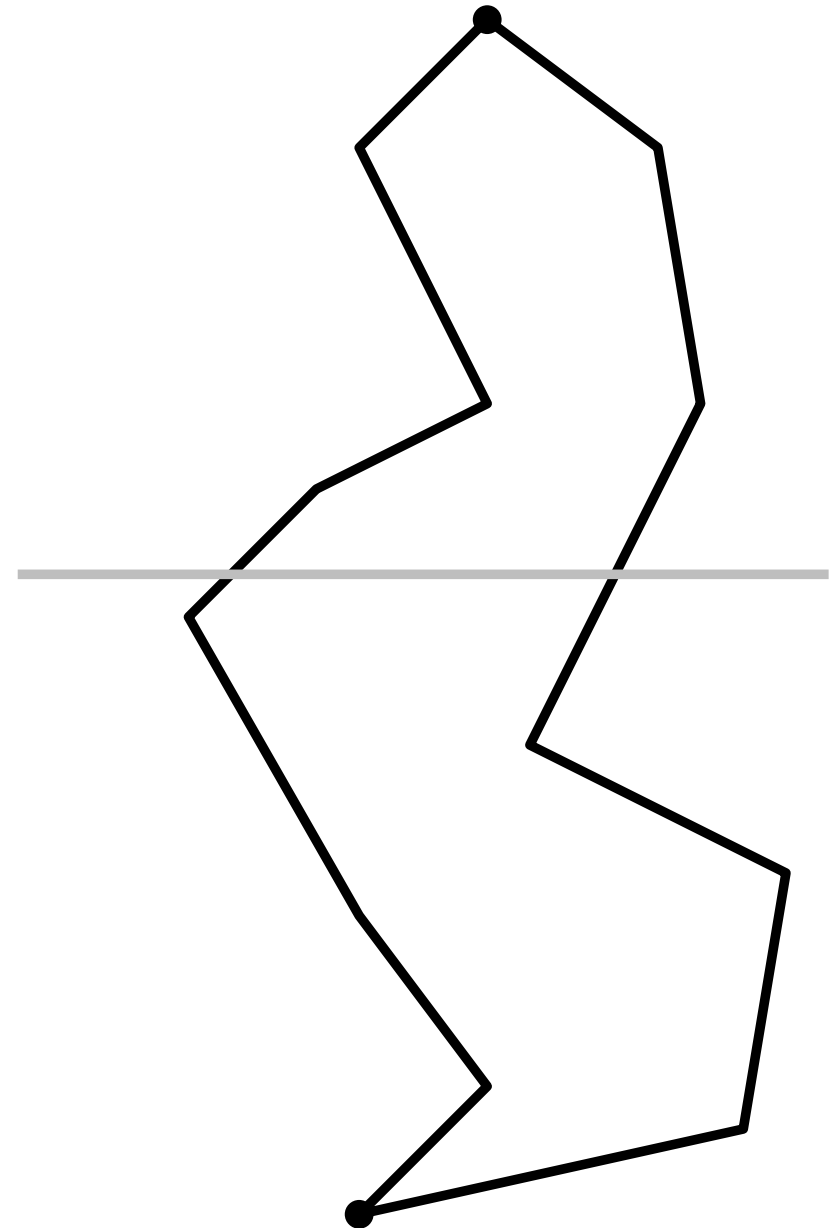
# Faster algorithm

$y$ -monotone polygon: Intersection with any horizontal line is connected.  
Equivalent: One vertex with both edges going down and one with both edges going up.

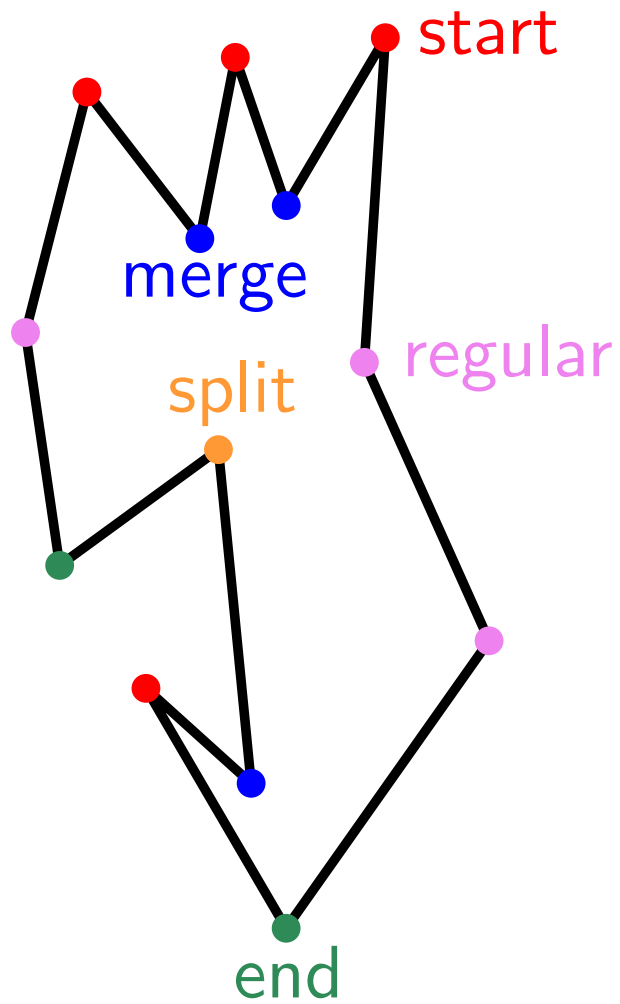
Idea:

Phase 1: Split polygon  $P$  into  $y$ -monotone polygons.

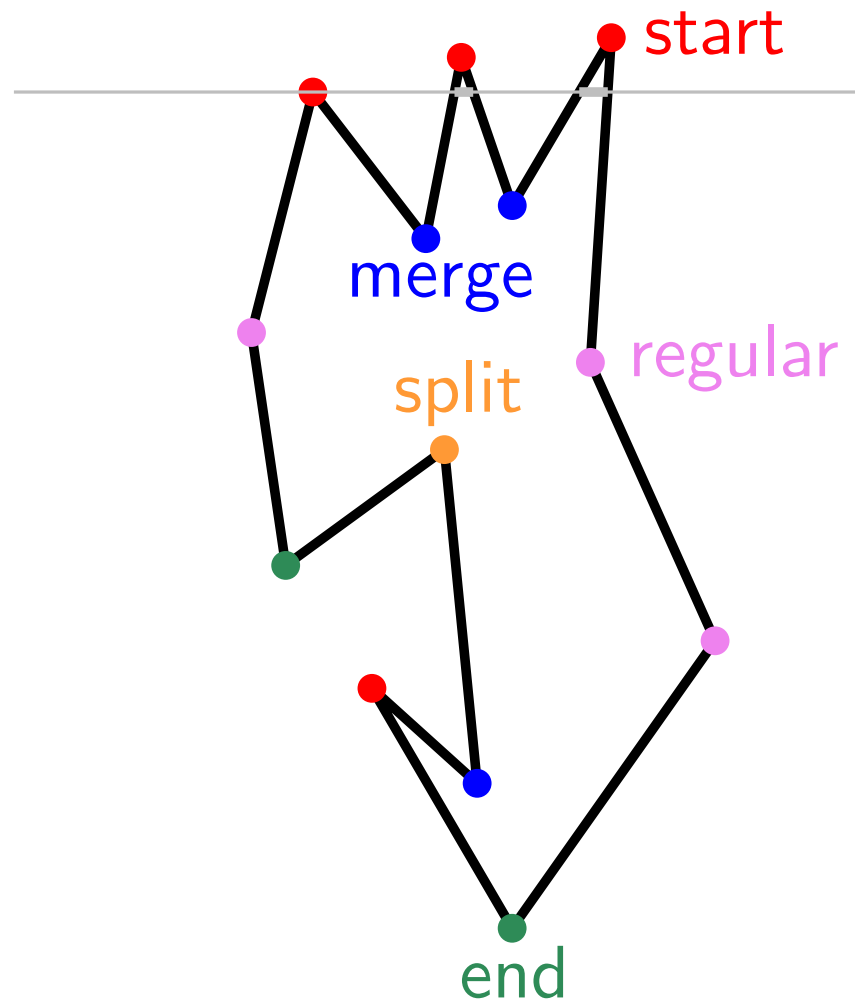
Phase 2: Triangulate each  $y$ -monotone polygon.



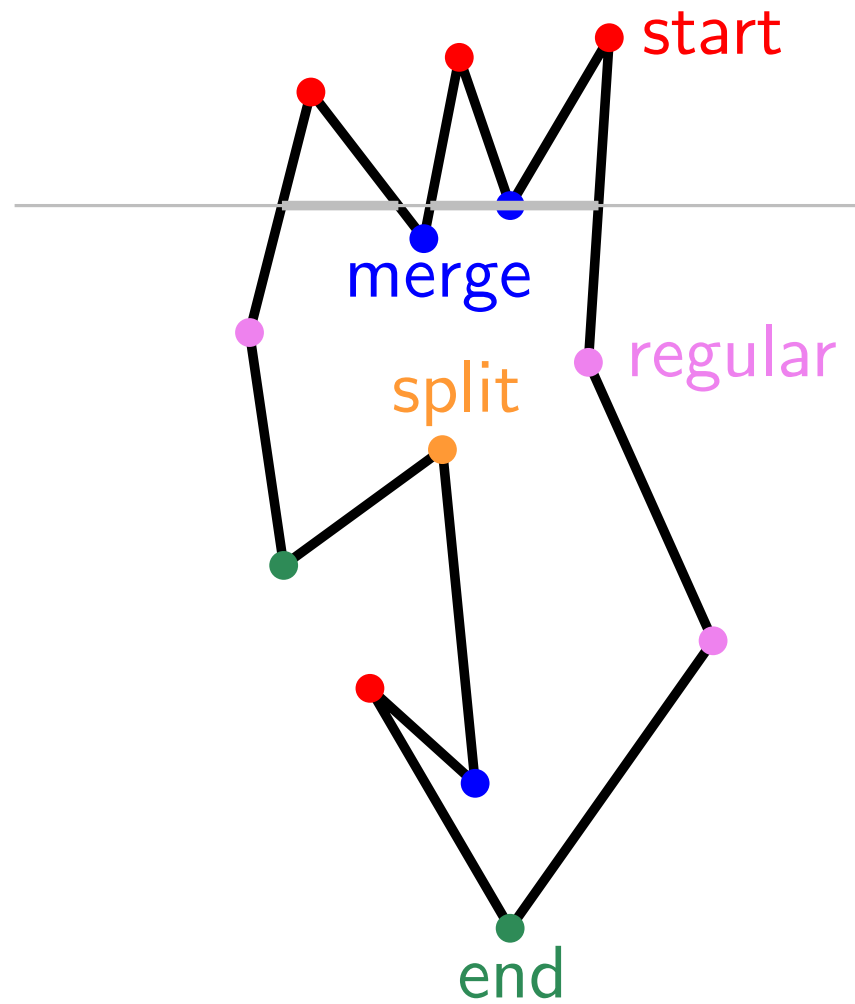
# 4 types of vertices



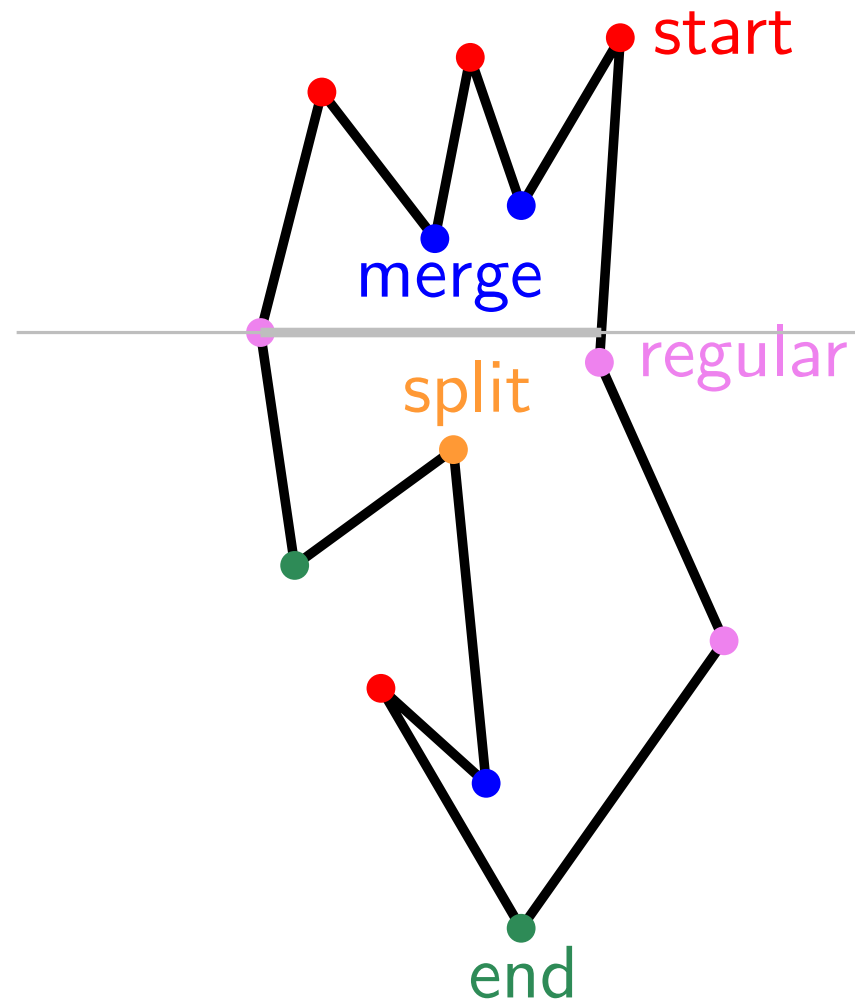
# 4 types of vertices



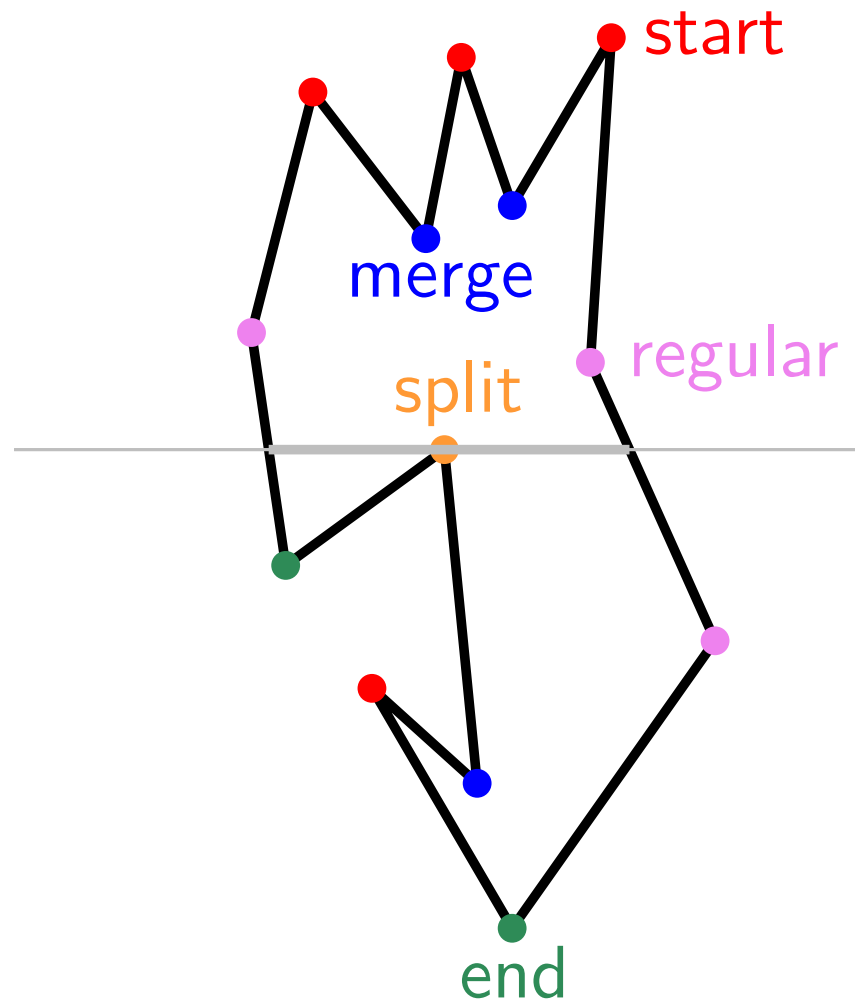
# 4 types of vertices



# 4 types of vertices

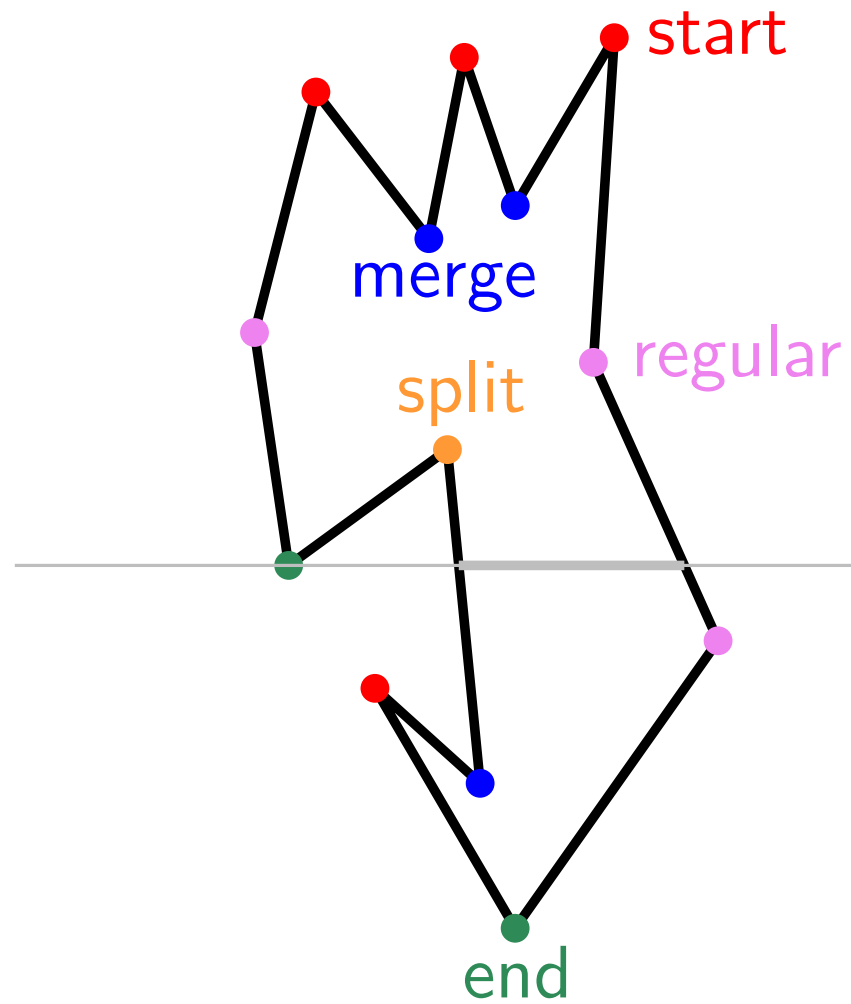


# 4 types of vertices





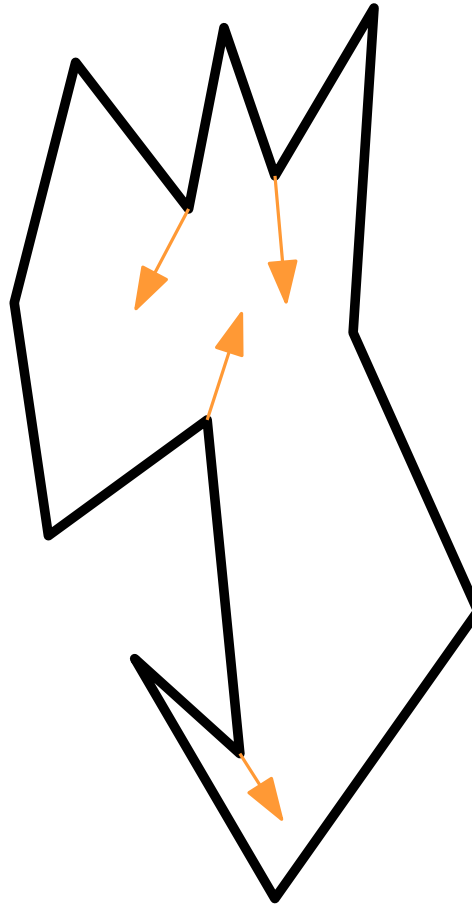
# 4 types of vertices



## Idea

Idea: “Shoot” a diagonal from every  
split and merge vertex.

No split and merge  $\Rightarrow$   
monotone polygon!

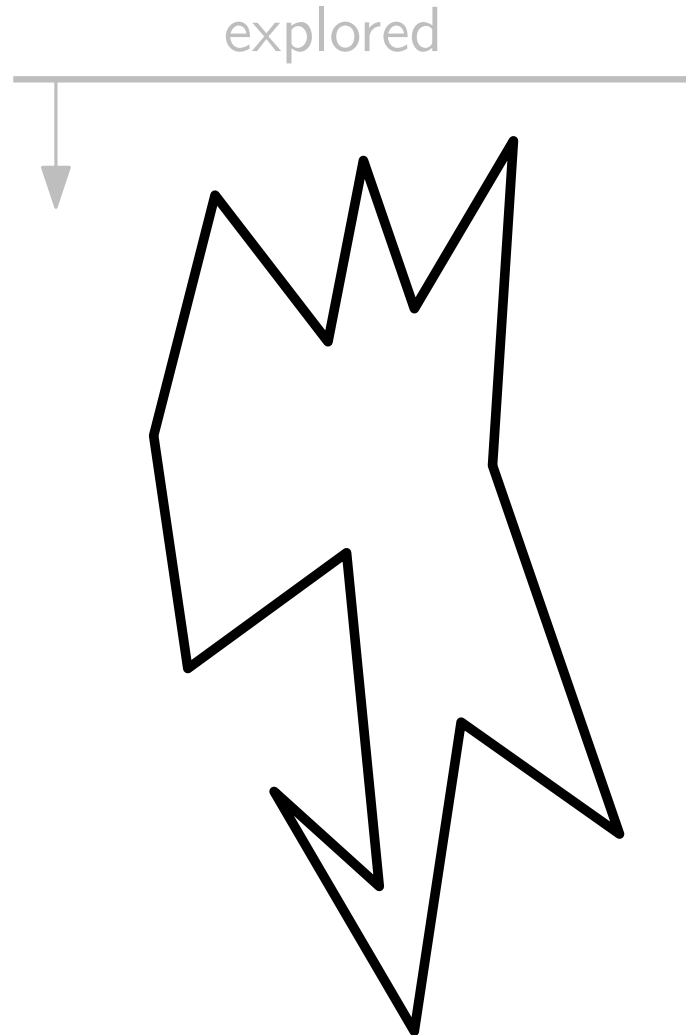


# Idea

Use horizontal sweep line going down.

Fix split vertices first.

Then go up and fix merge vertices.

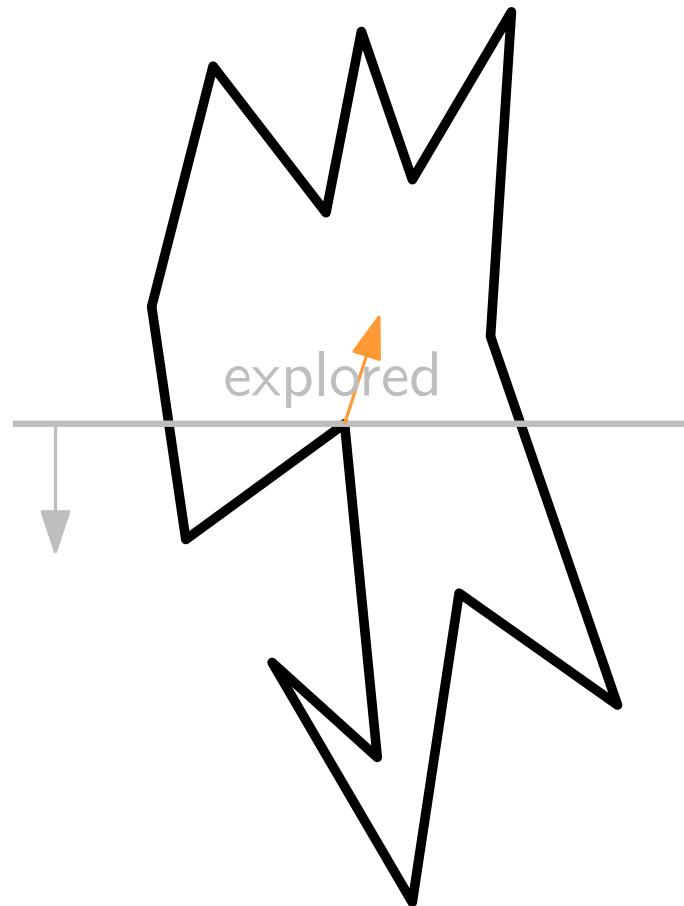


# Idea

Use horizontal sweep line going down.

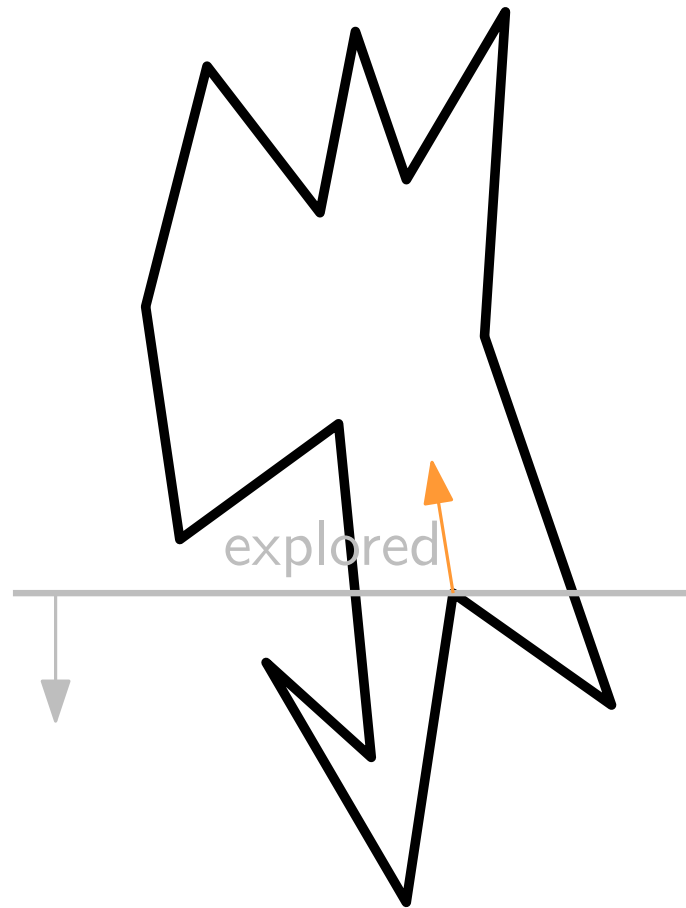
Fix split vertices first.

Then go up and fix merge vertices.



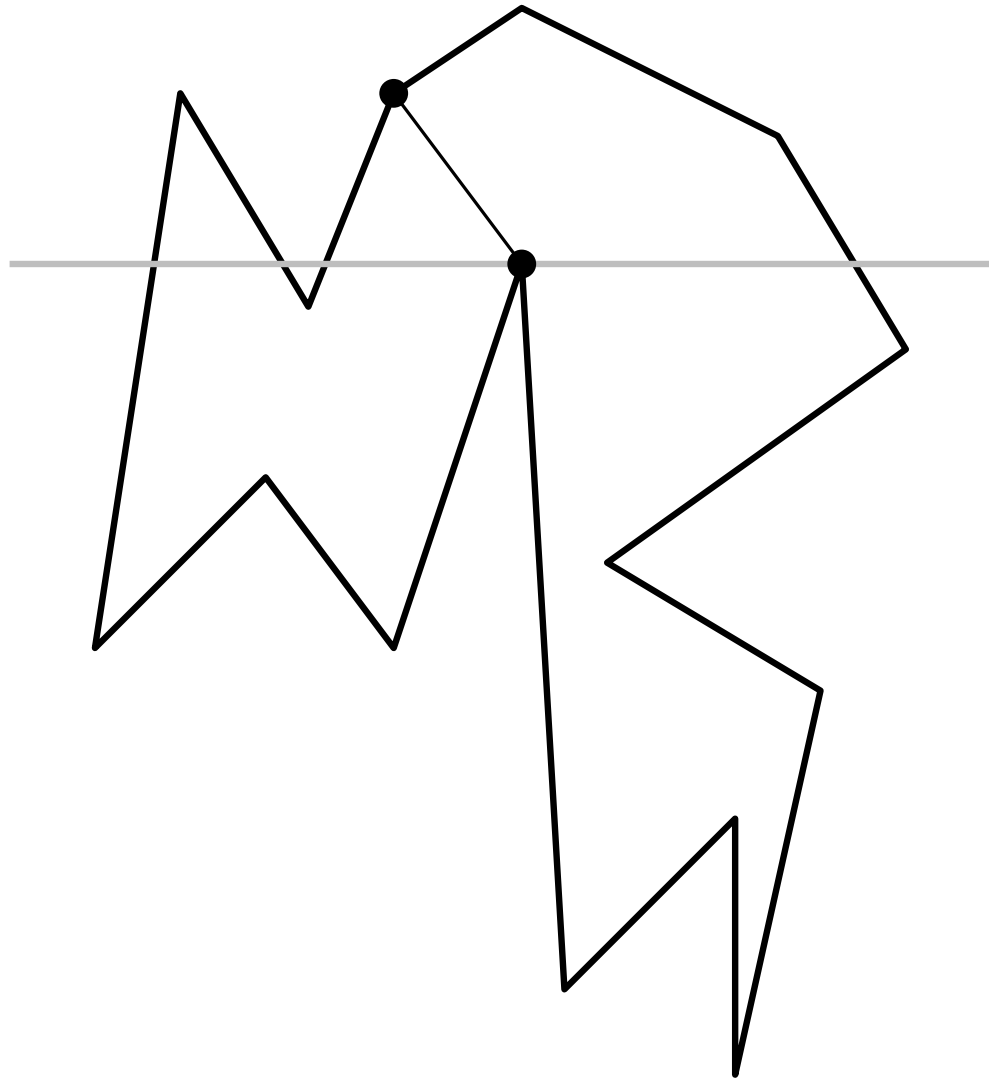
## Idea

Use horizontal sweep line going down.  
Fix split vertices first.  
Then go up and fix merge vertices.



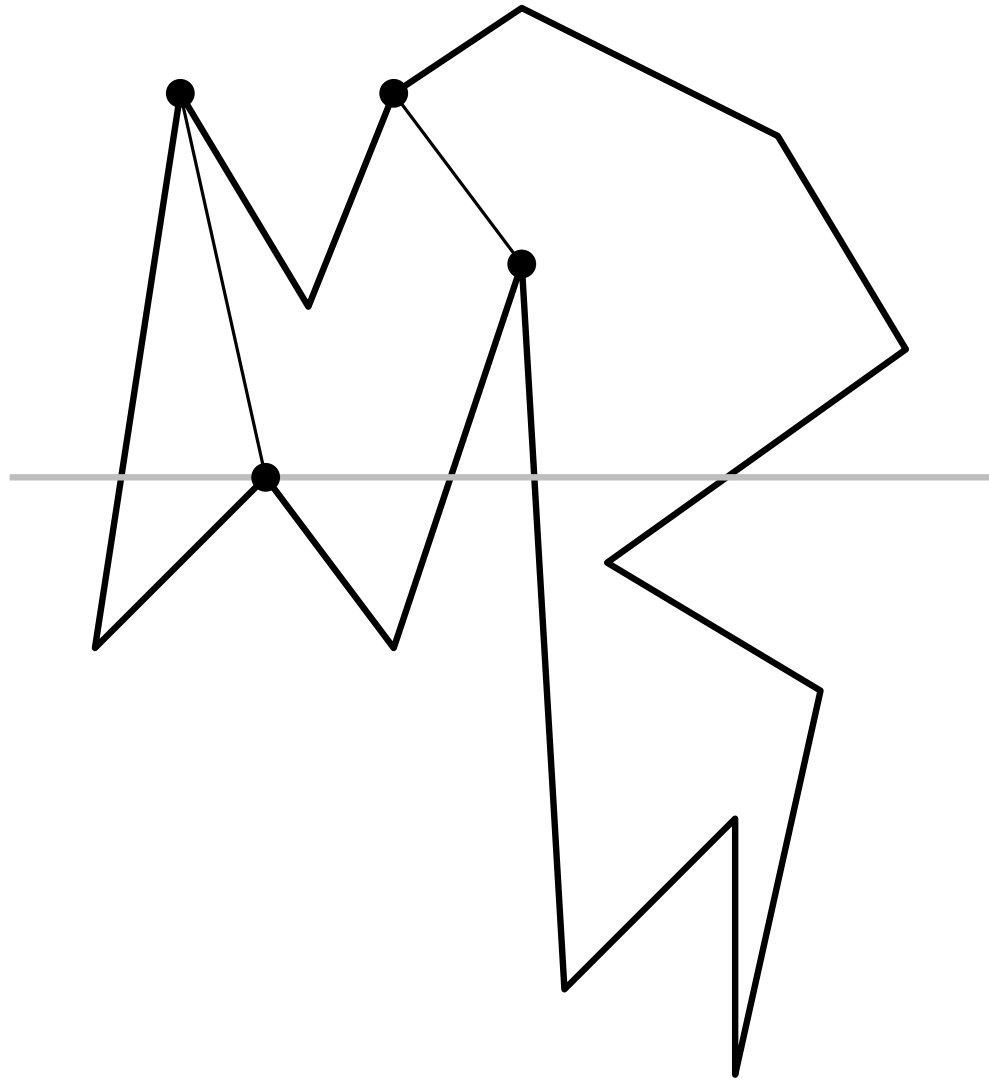
## First idea

Make segment to top  
vertex of line segment  
to the left?



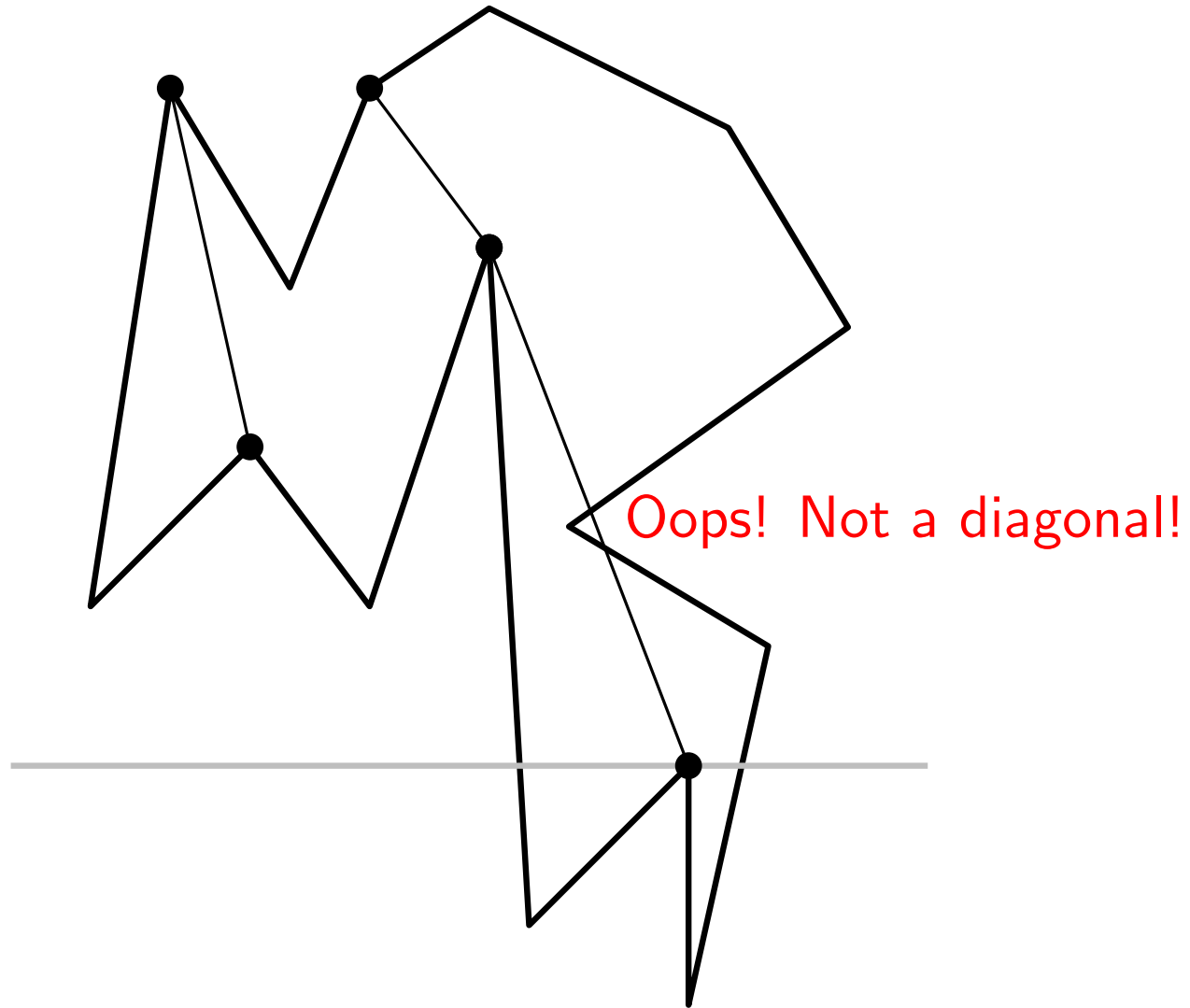
## First idea

Make segment to top  
vertex of line segment  
to the left?



## First idea

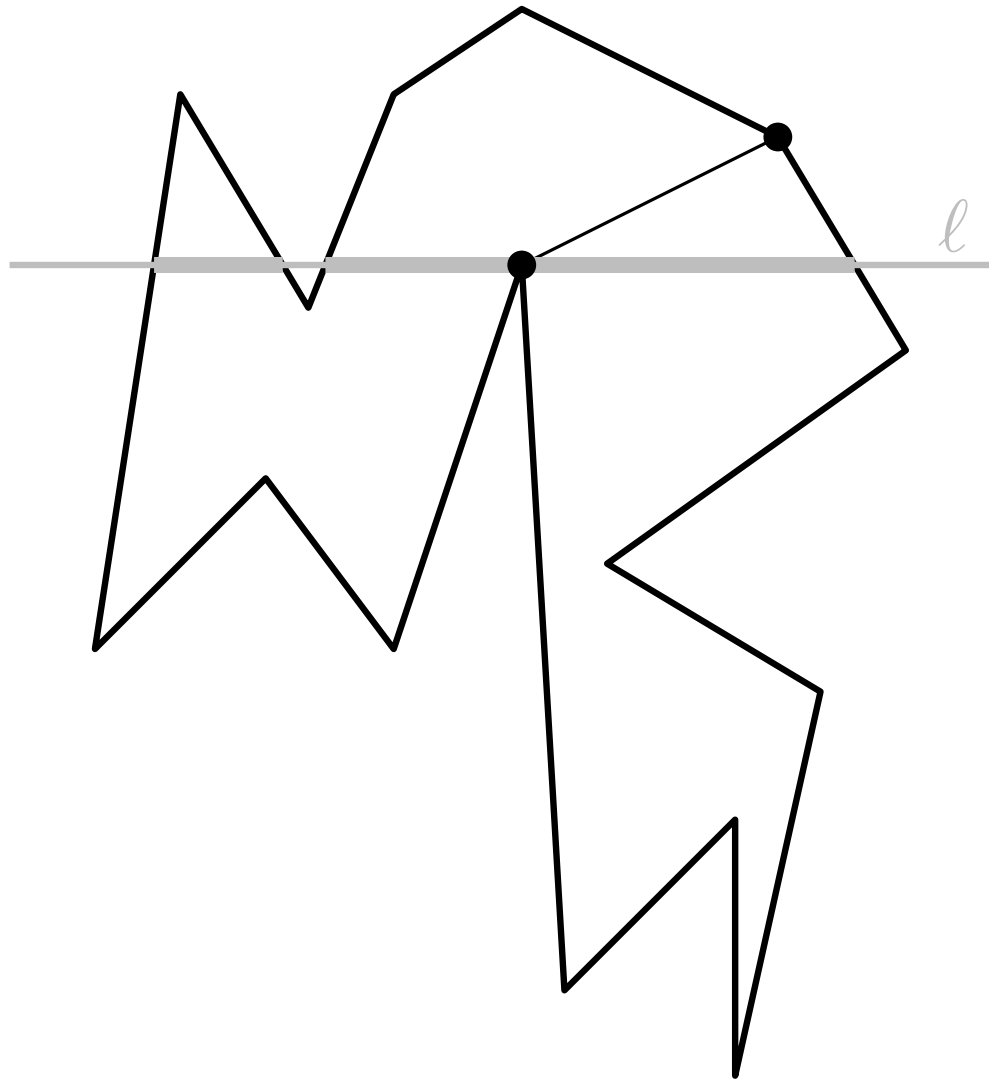
Make segment to top  
vertex of line segment  
to the left?





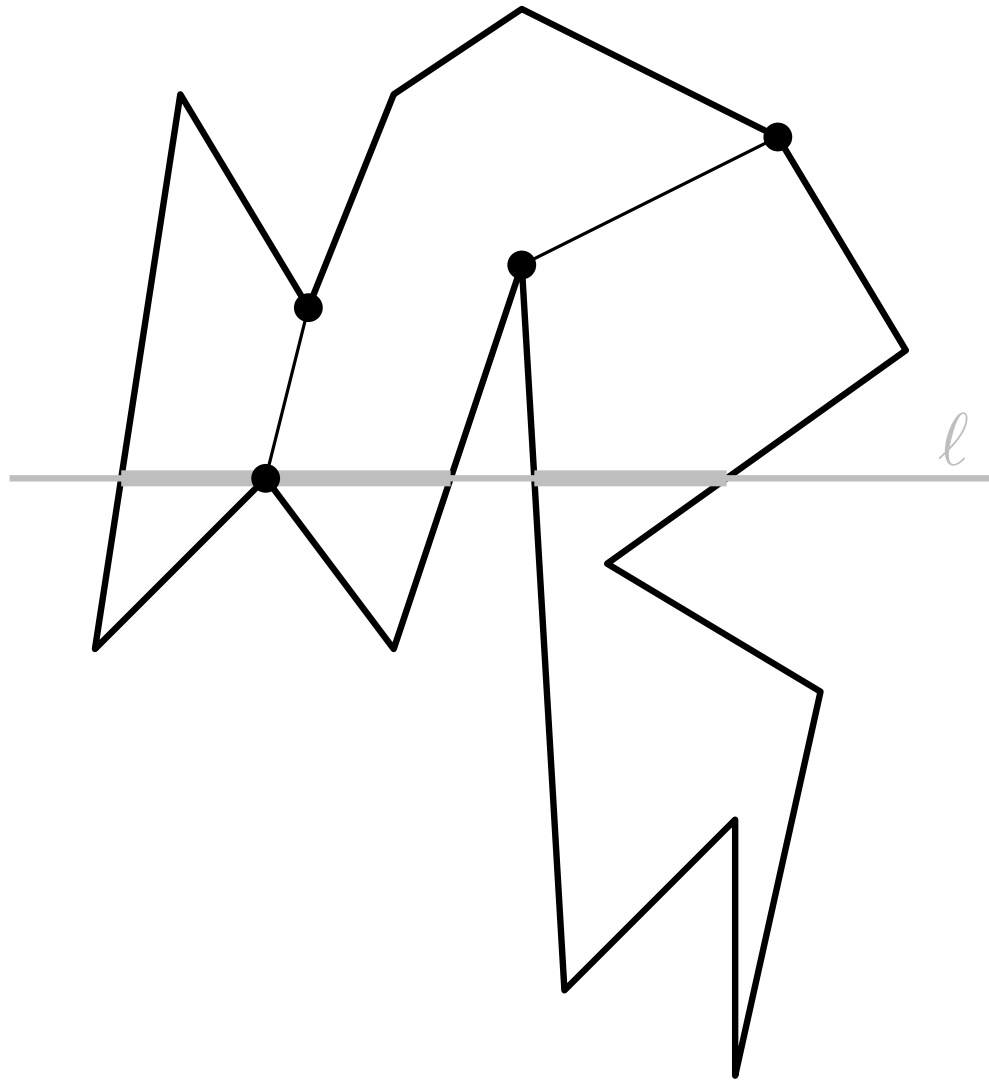
## Refined idea

Make segment to  
previous vertex visited  
by the component of  
 $\ell \cap P$ .



## Refined idea

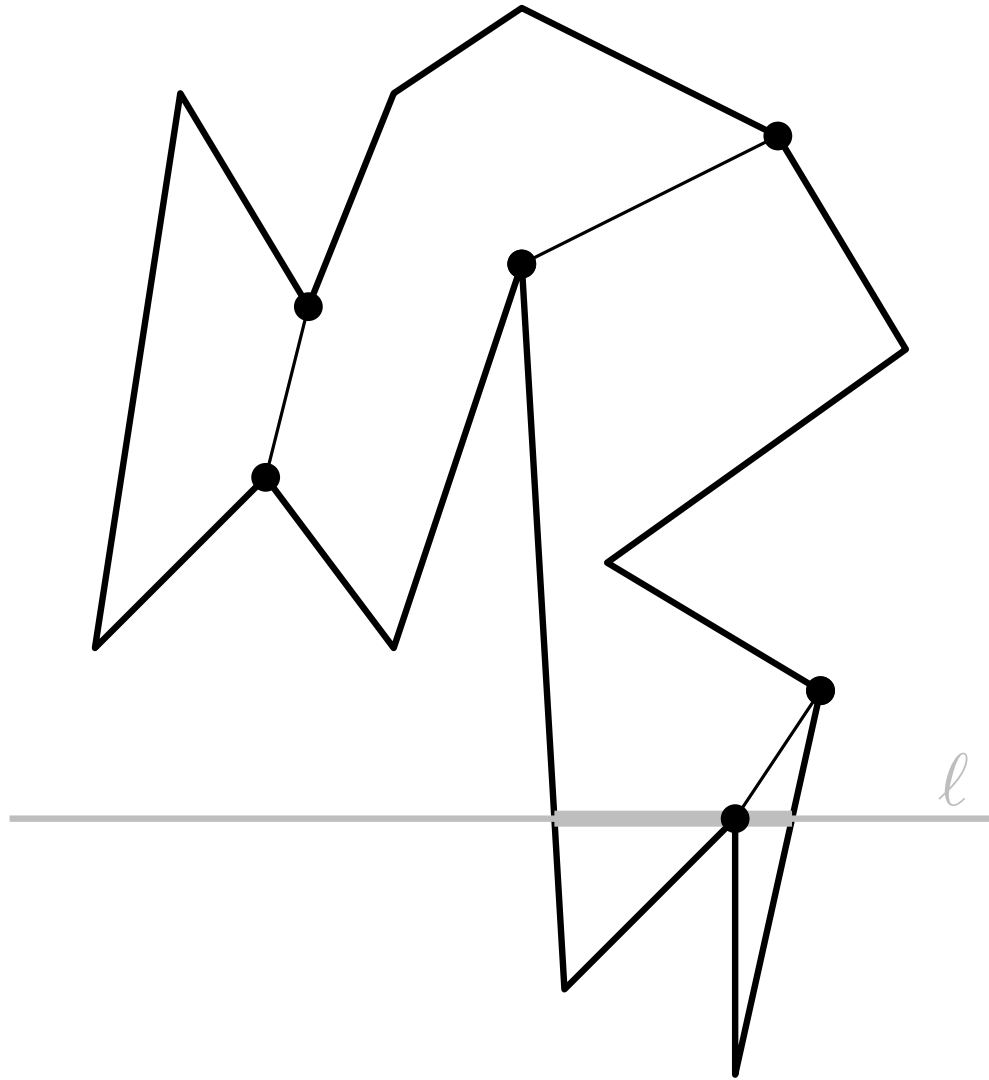
Make segment to  
previous vertex visited  
by the component of  
 $\ell \cap P$ .



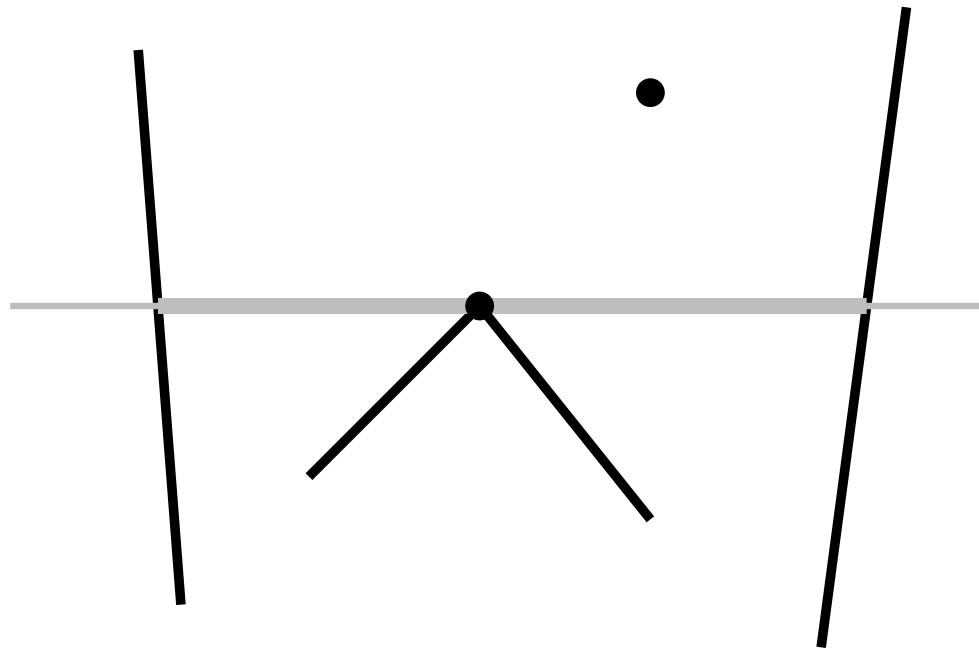
## Refined idea

Make segment to  
previous vertex visited  
by the component of  
 $\ell \cap P$ .

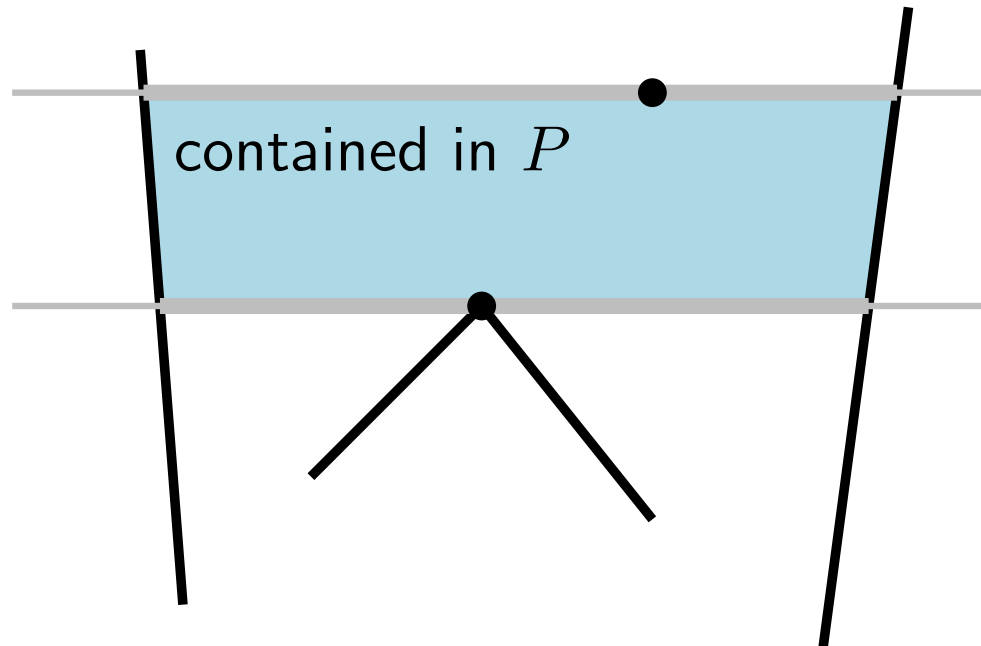
Always a diagonal?  
Non-intersecting  
diagonals?



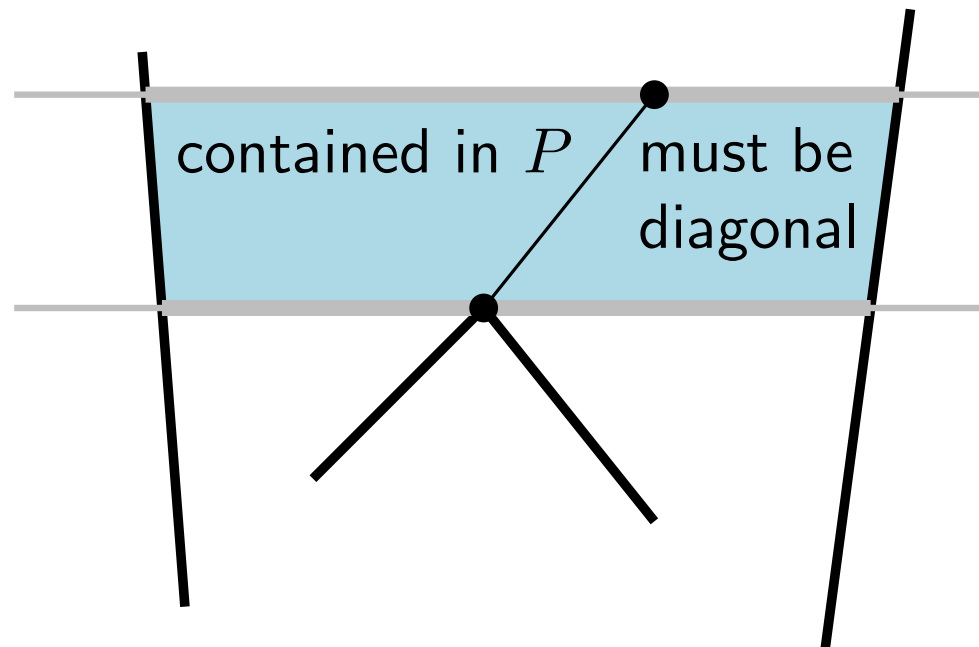
Idea works!



Idea works!

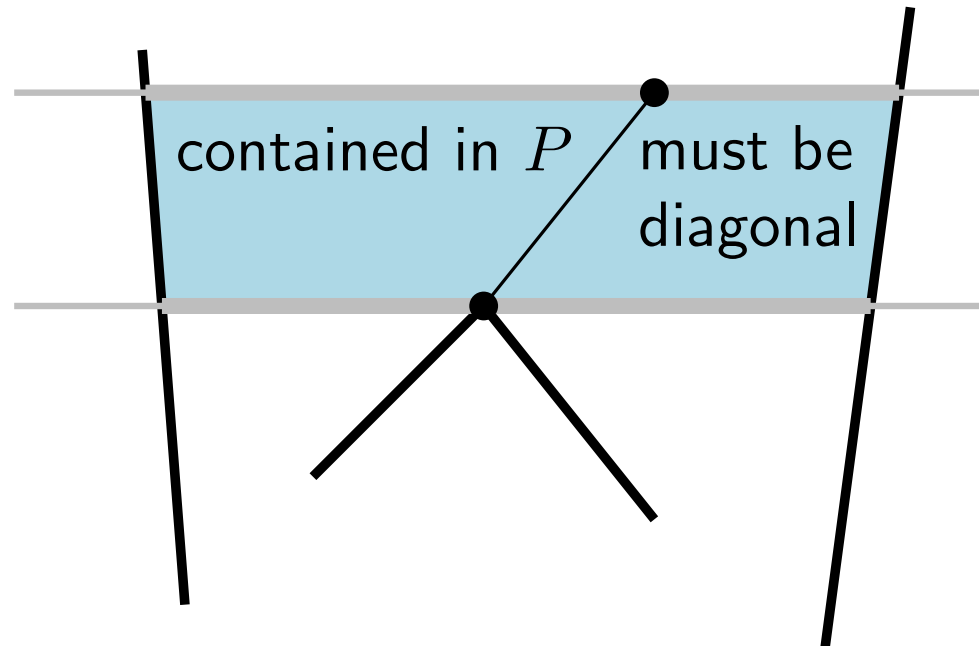


Idea works!



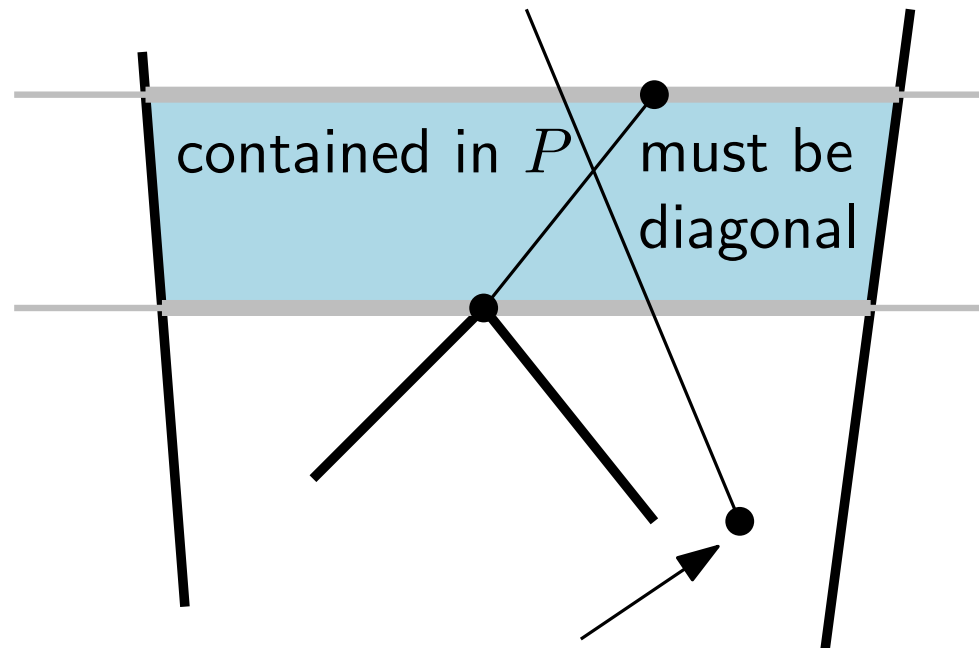
Idea works!

Can new diagonal  
intersect a  
previously added?



Idea works!

Can new diagonal  
intersect a  
previously added?

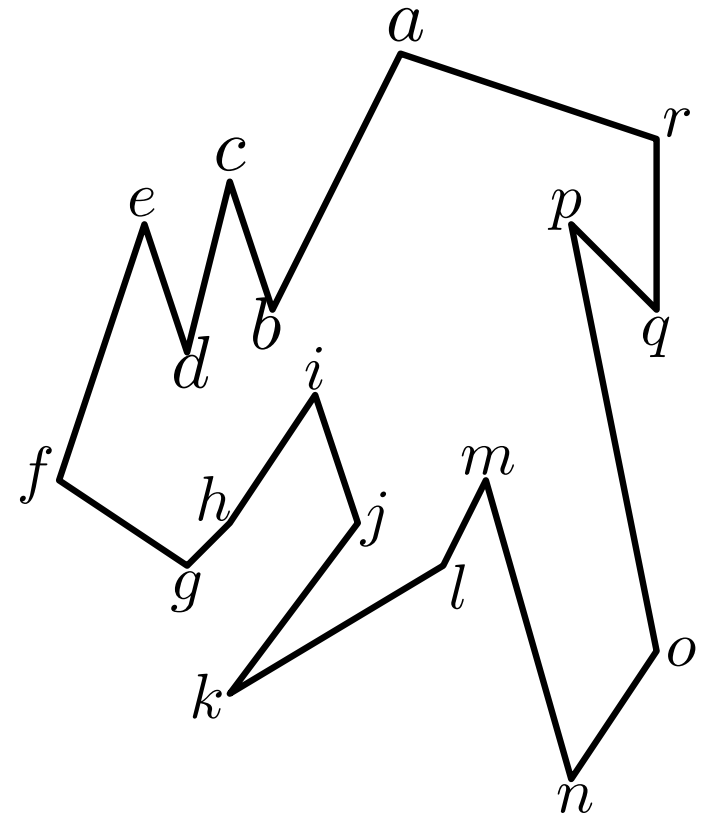
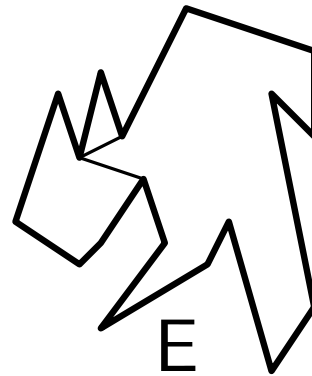
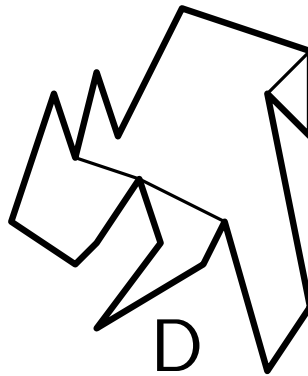
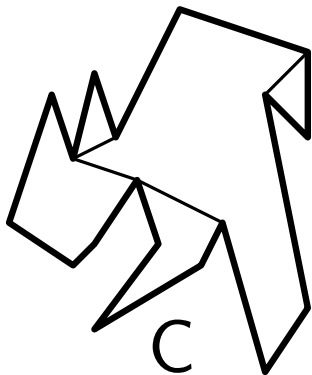
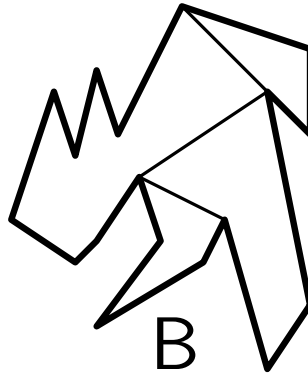
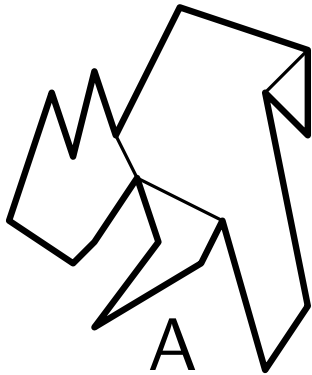


Vertex not yet  
visited.  
Contradiction.



# What diagonals are introduced after sweeping down?

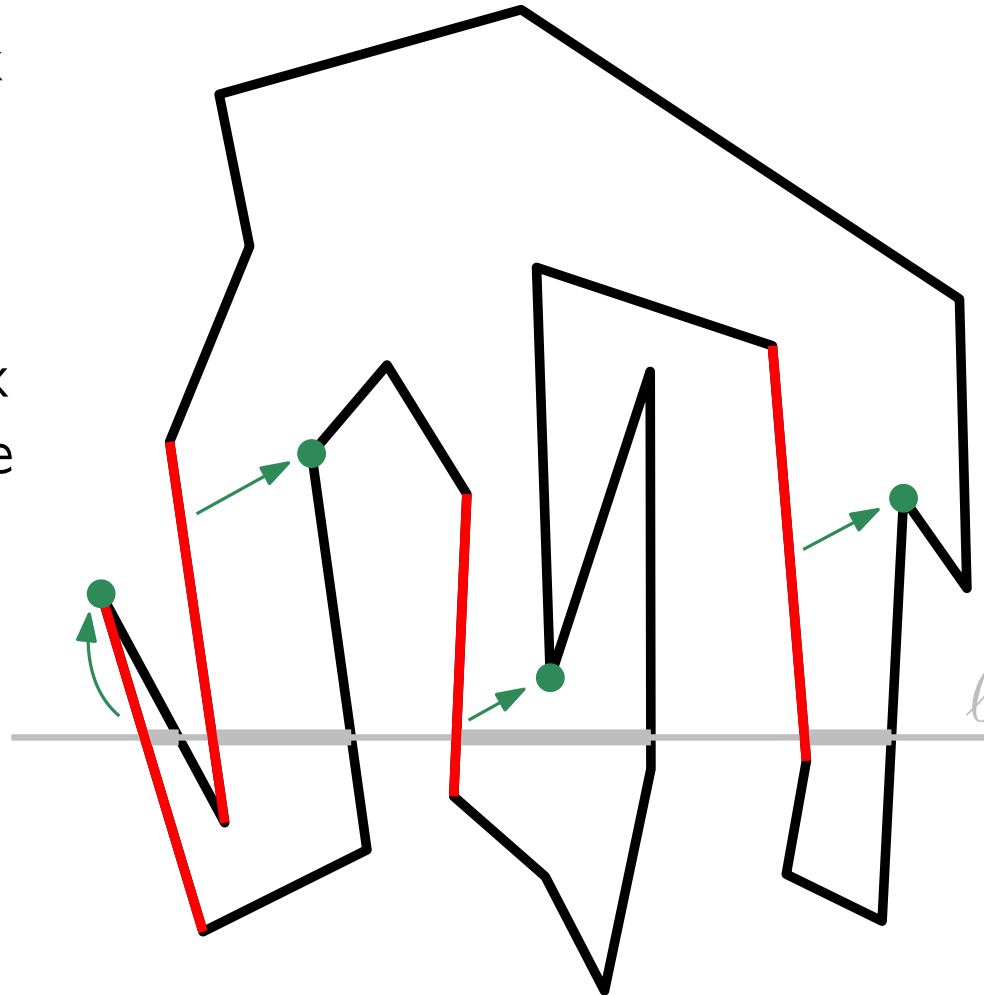
socrative.com → Student login,  
Room name: ABRAHAMSEN3464



# Helpers

Helper of edge  $e$  intersected by  $\ell$  with interior of  $P$  to the right: Previous vertex visited by this connected component of  $\ell \cap P$ .

Equivalent: Lowest vertex above  $\ell$  that sees  $e$  to the left.

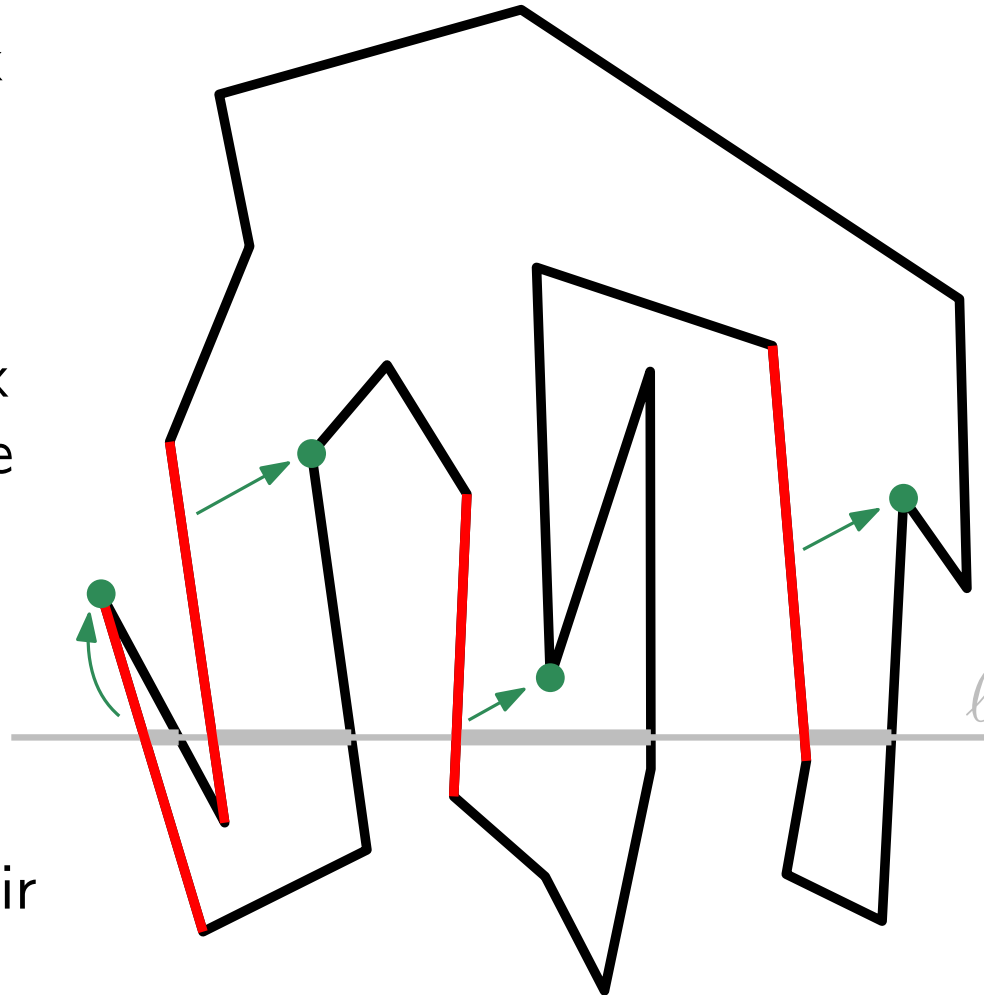


# Helpers

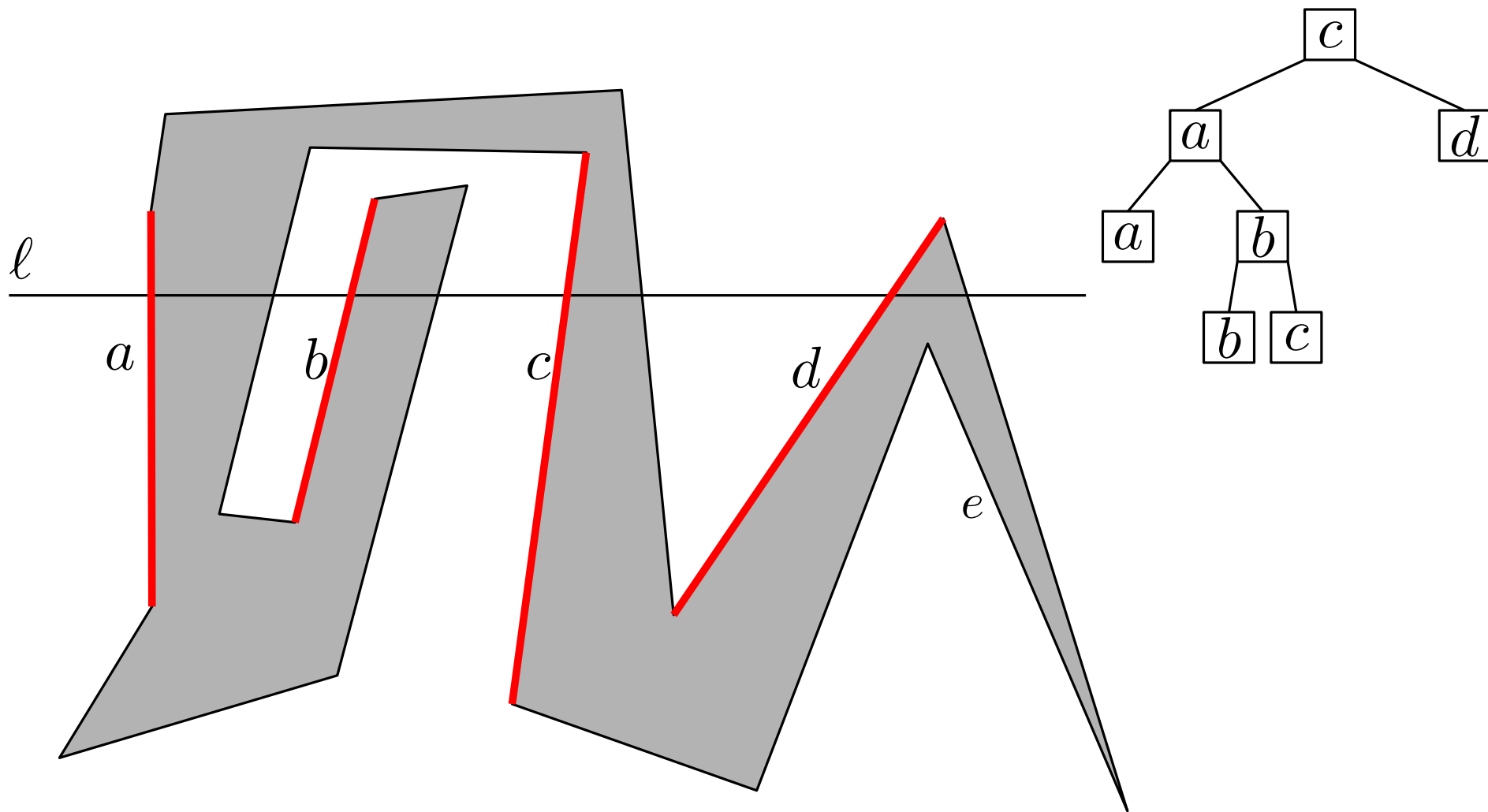
Helper of edge  $e$  intersected by  $\ell$  with interior of  $P$  to the right: Previous vertex visited by this connected component of  $\ell \cap P$ .

Equivalent: Lowest vertex above  $\ell$  that sees  $e$  to the left.

Intersected edges and their helpers are stored in the *status* structure  $T$ .

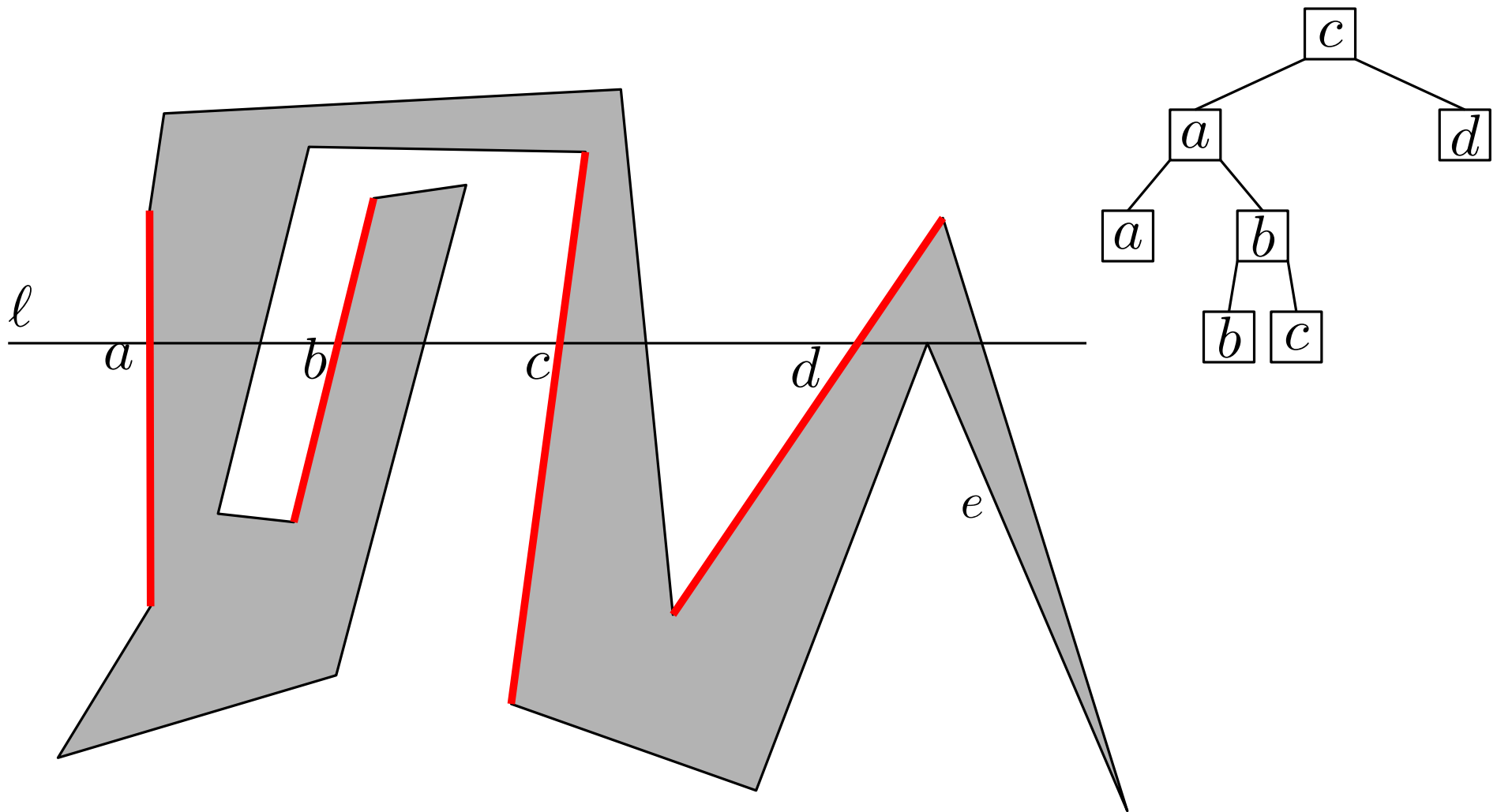


# Status



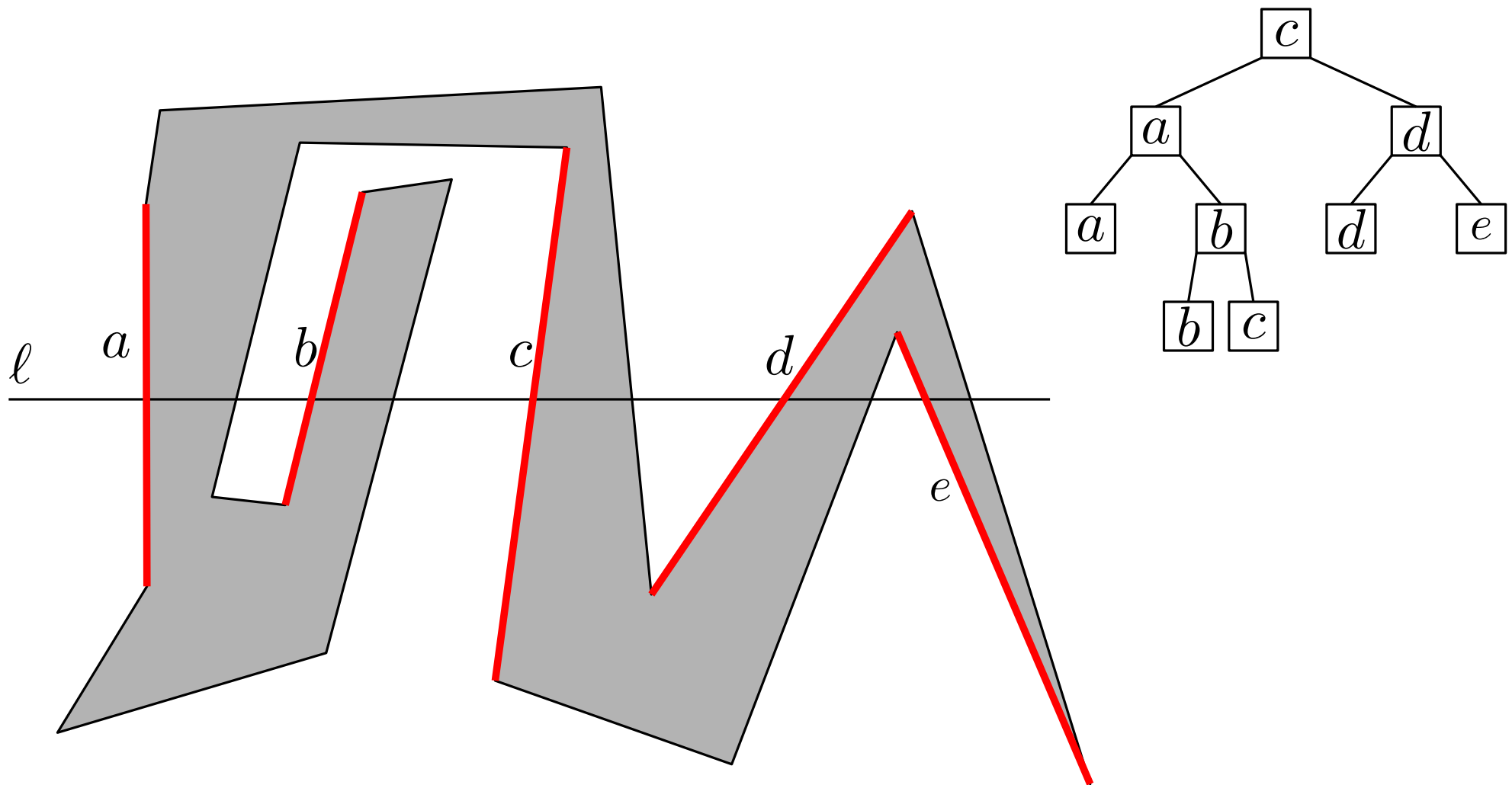
Implement status with balanced binary search tree  $T$ .  
Sorting order: intersection points with  $\ell$  from left to right.

# Status



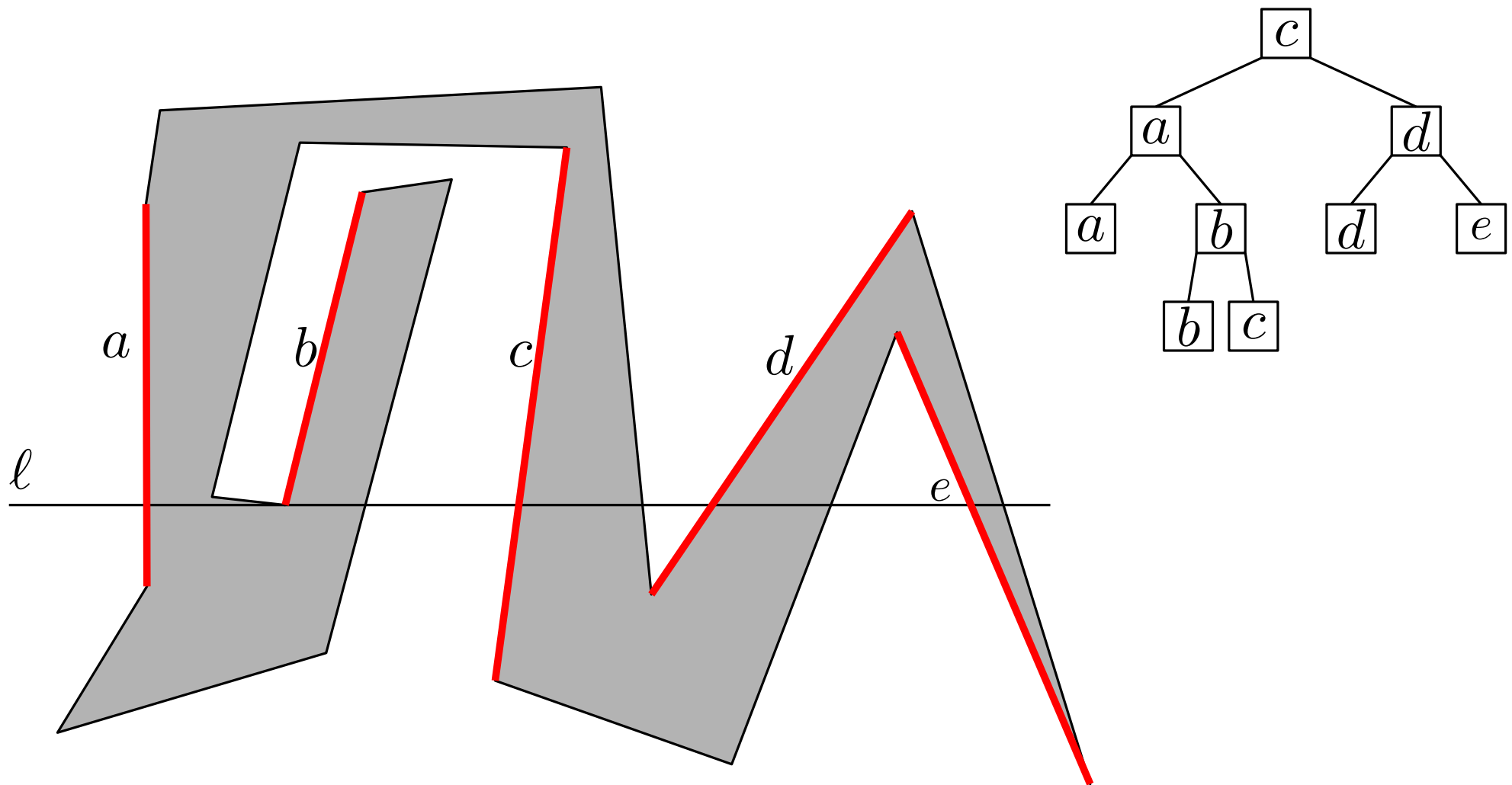
Implement status with balanced binary search tree  $T$ .  
Sorting order: intersection points with  $\ell$  from left to right.

# Status



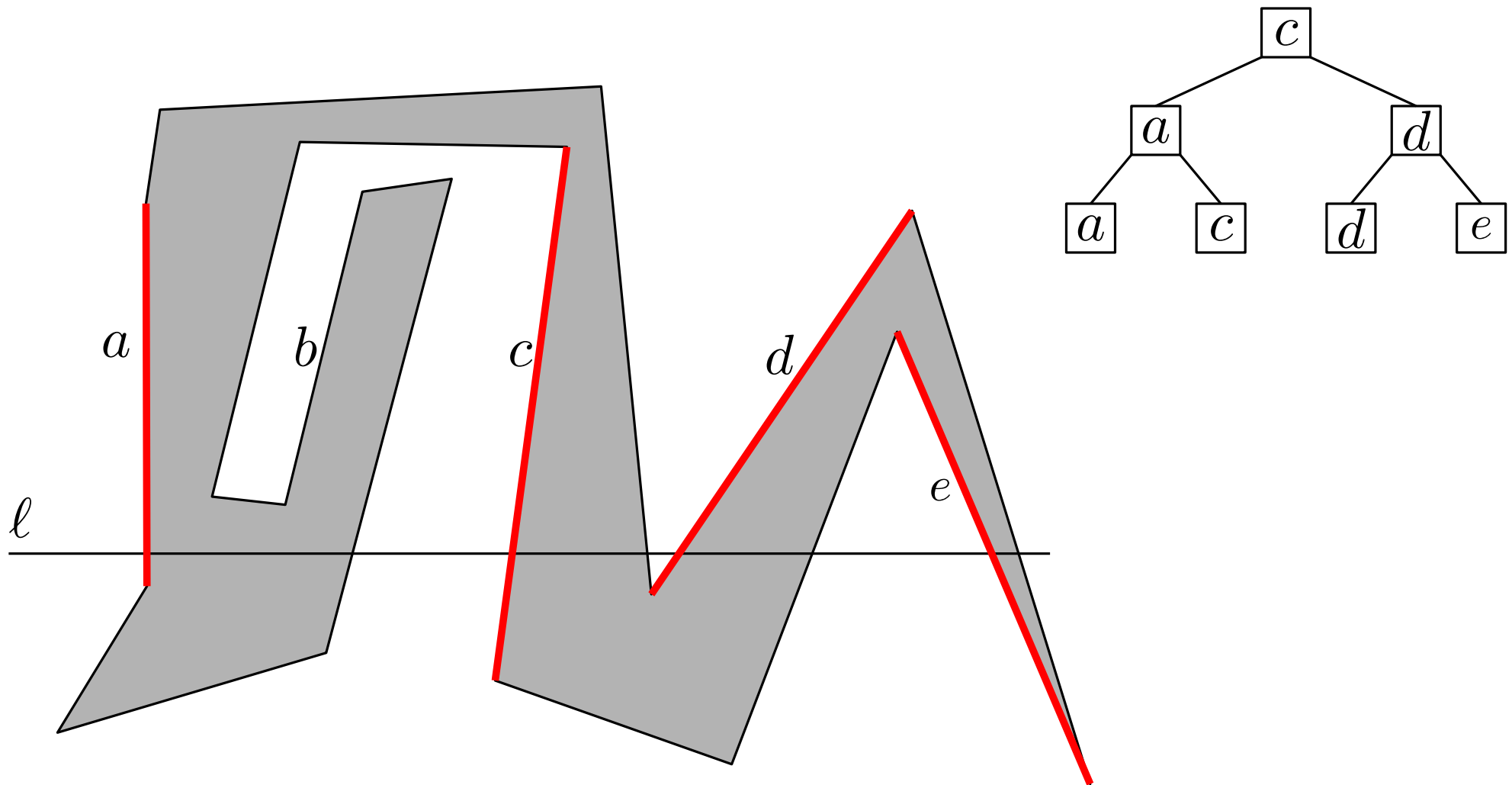
Implement status with balanced binary search tree  $T$ .  
Sorting order: intersection points with  $\ell$  from left to right.

# Status



Implement status with balanced binary search tree  $T$ .  
Sorting order: intersection points with  $\ell$  from left to right.

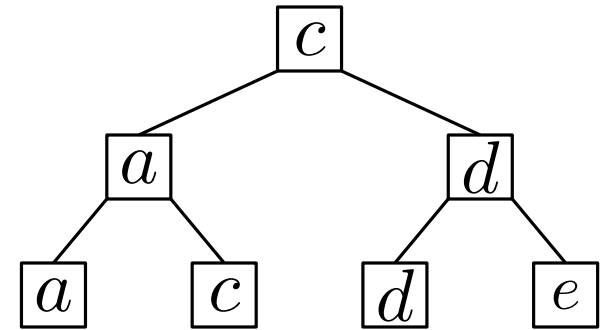
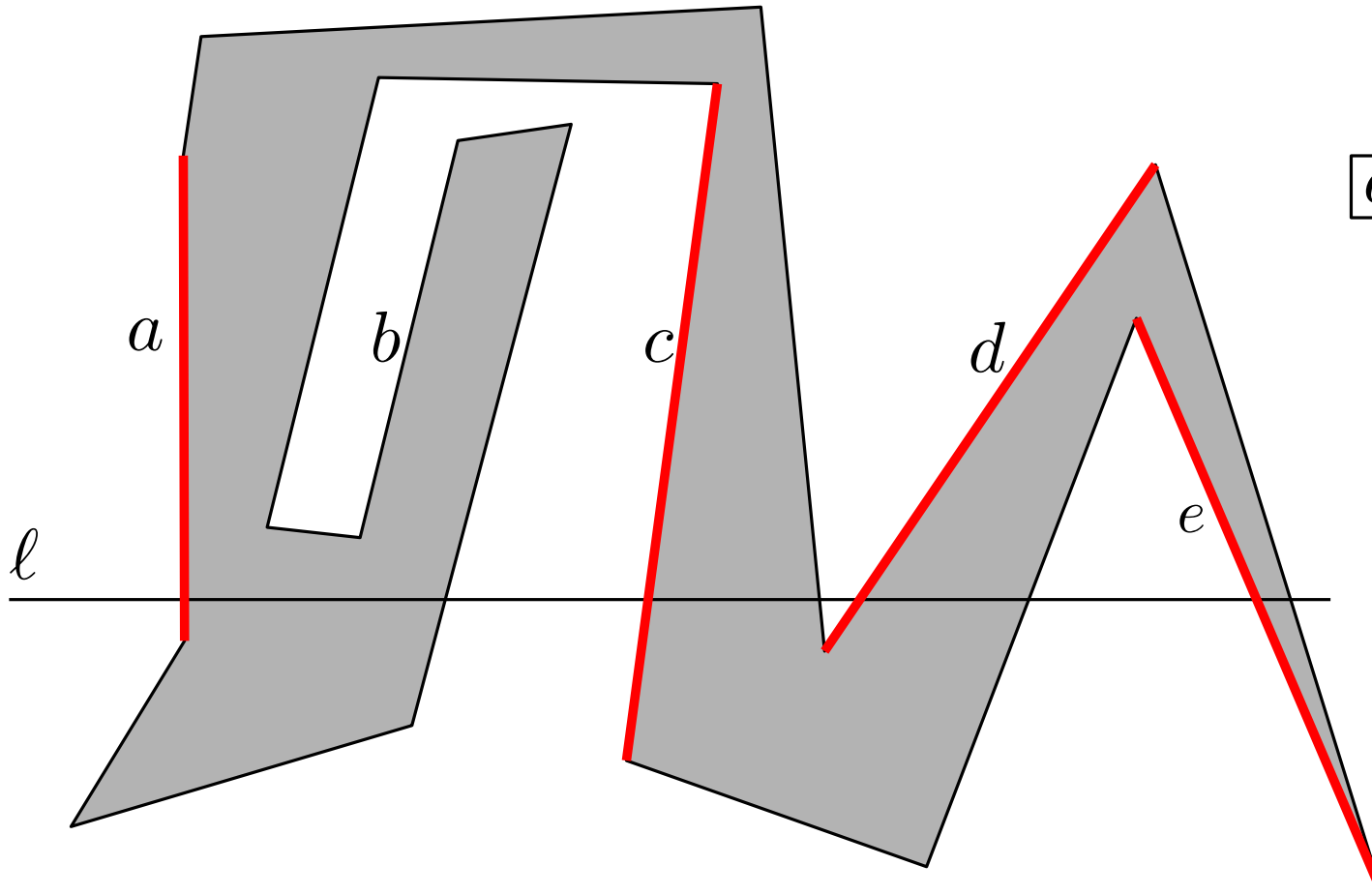
# Status



Implement status with balanced binary search tree  $T$ .  
Sorting order: intersection points with  $\ell$  from left to right.



# Status

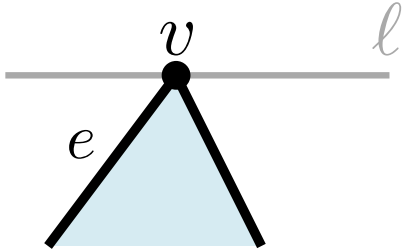


When reaching a new vertex, use  $O(\log n)$  time to find the edge in  $T$  immediately to the left.

Implement status with balanced binary search tree  $T$ .  
Sorting order: intersection points with  $\ell$  from left to right.

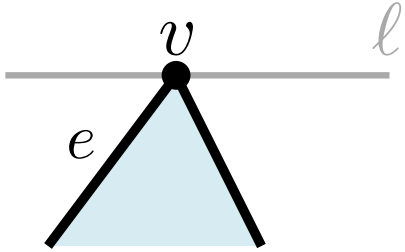
# Events

Start vertex: Insert  $e$  in  $T$  with helper  $v$ .

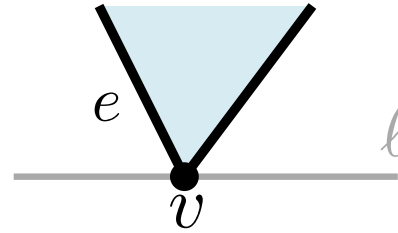


# Events

Start vertex: Insert  $e$  in  $T$  with helper  $v$ .

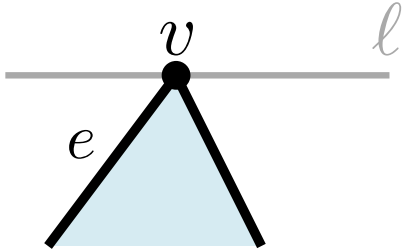


End vertex: Remove  $e$  from  $T$ .

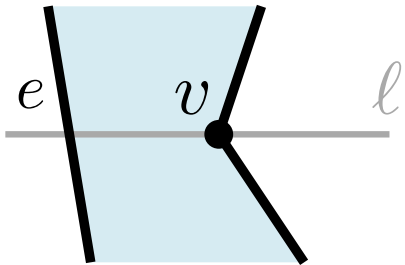


# Events

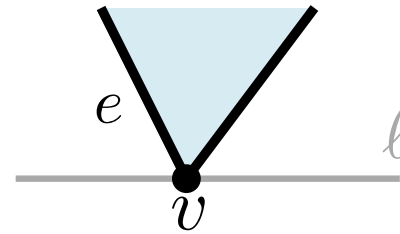
Start vertex: Insert  $e$  in  $T$  with helper  $v$ .



Regular vertex with  $P$  to the left:  
Update helper of  $e$  to  $v$ .

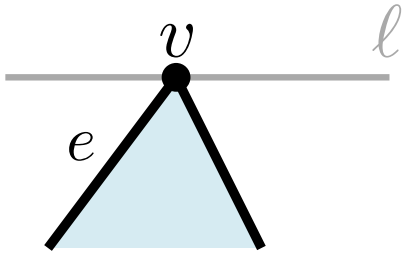


End vertex: Remove  $e$  from  $T$ .

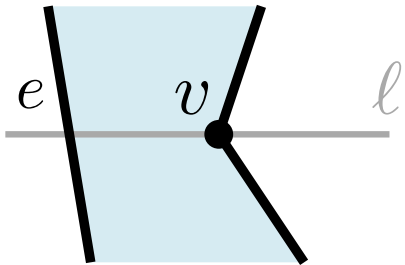


# Events

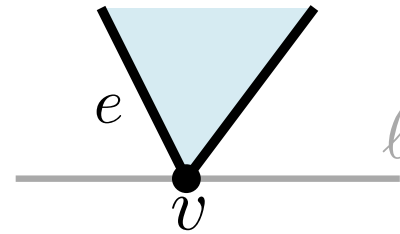
Start vertex: Insert  $e$  in  $T$  with helper  $v$ .



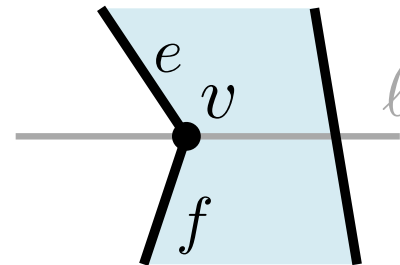
Regular vertex with  $P$  to the left:  
Update helper of  $e$  to  $v$ .



End vertex: Remove  $e$  from  $T$ .

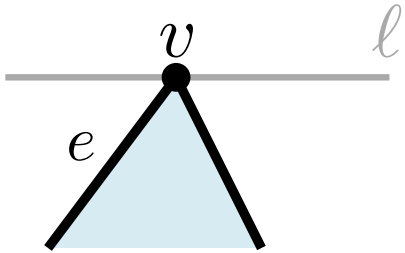


Regular vertex with  $P$  to the right:  
Replace  $e$  by  $f$  in  $T$  with helper  $v$ .

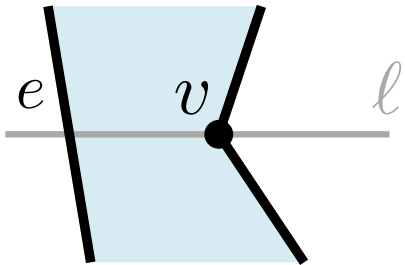


# Events

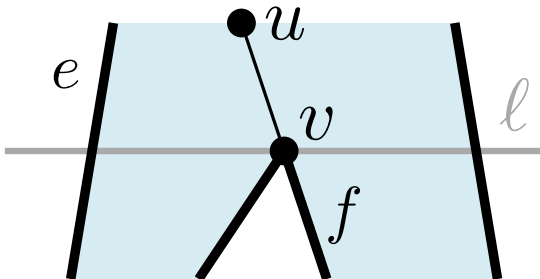
Start vertex: Insert  $e$  in  $T$  with helper  $v$ .



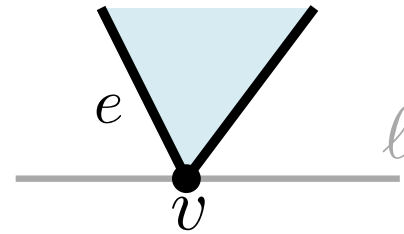
Regular vertex with  $P$  to the left:  
Update helper of  $e$  to  $v$ .



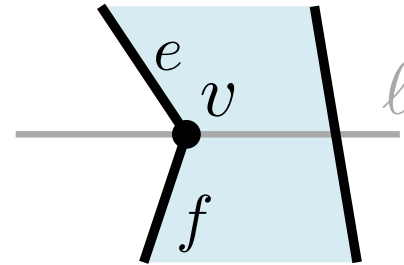
Split vertex: Add diagonal to helper  $u$  of  $e$ . Update helper of  $e$  to  $v$ . Add  $f$  to  $T$  with helper  $v$ .



End vertex: Remove  $e$  from  $T$ .

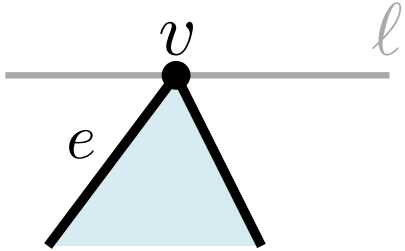


Regular vertex with  $P$  to the right:  
Replace  $e$  by  $f$  in  $T$  with helper  $v$ .

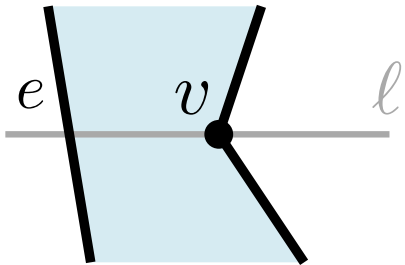


# Events

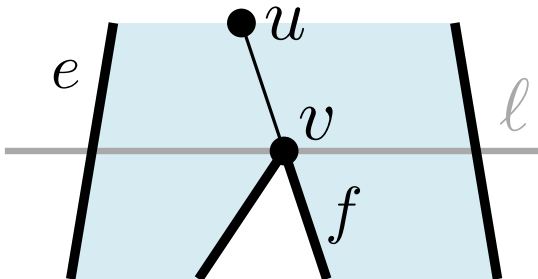
Start vertex: Insert  $e$  in  $T$  with helper  $v$ .



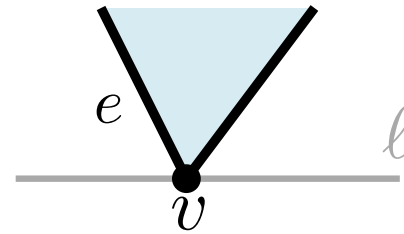
Regular vertex with  $P$  to the left:  
Update helper of  $e$  to  $v$ .



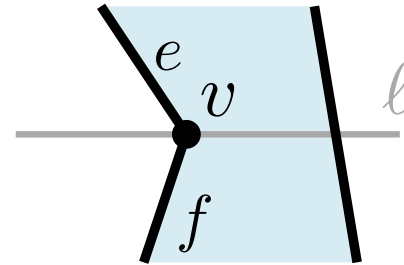
Split vertex: Add diagonal to helper  $u$  of  $e$ . Update helper of  $e$  to  $v$ . Add  $f$  to  $T$  with helper  $v$ .



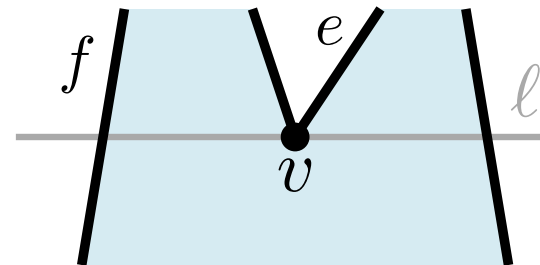
End vertex: Remove  $e$  from  $T$ .



Regular vertex with  $P$  to the right:  
Replace  $e$  by  $f$  in  $T$  with helper  $v$ .

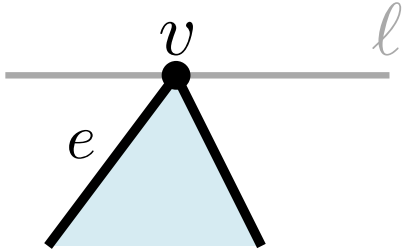


Merge vertex: Remove  $e$  from  $T$ .  
Update helper of  $f$  to  $v$ .

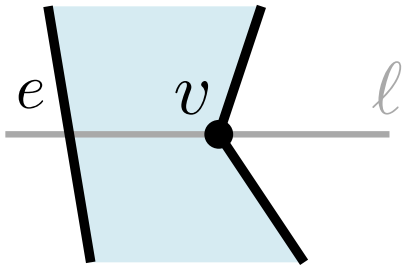


# Events

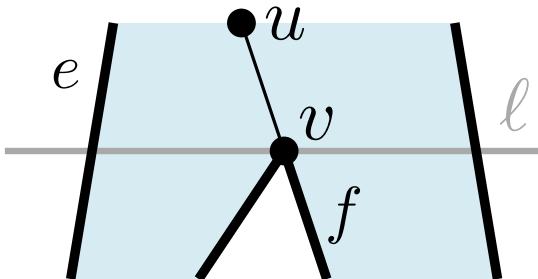
Start vertex: Insert  $e$  in  $T$  with helper  $v$ .



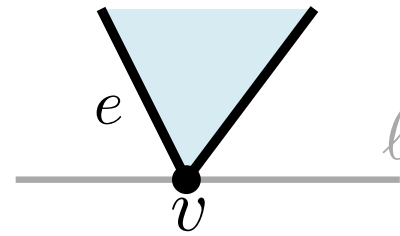
Regular vertex with  $P$  to the left:  
Update helper of  $e$  to  $v$ .



Split vertex: Add diagonal to helper  $u$  of  $e$ . Update helper of  $e$  to  $v$ . Add  $f$  to  $T$  with helper  $v$ .

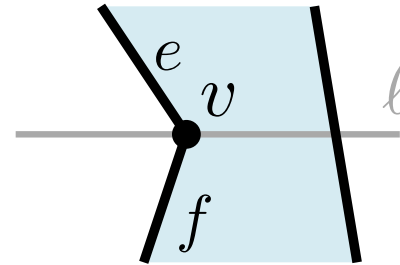


End vertex: Remove  $e$  from  $T$ .

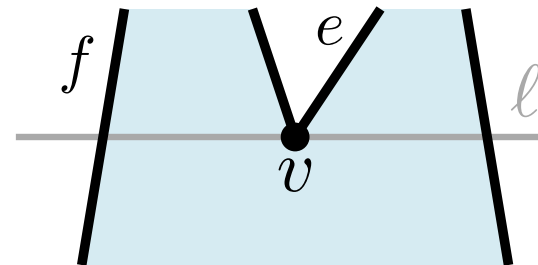


**Each event takes time  $O(\log n)$ .**

Regular vertex with  $P$  to the right:  
Replace  $e$  by  $f$  in  $T$  with helper  $v$ .

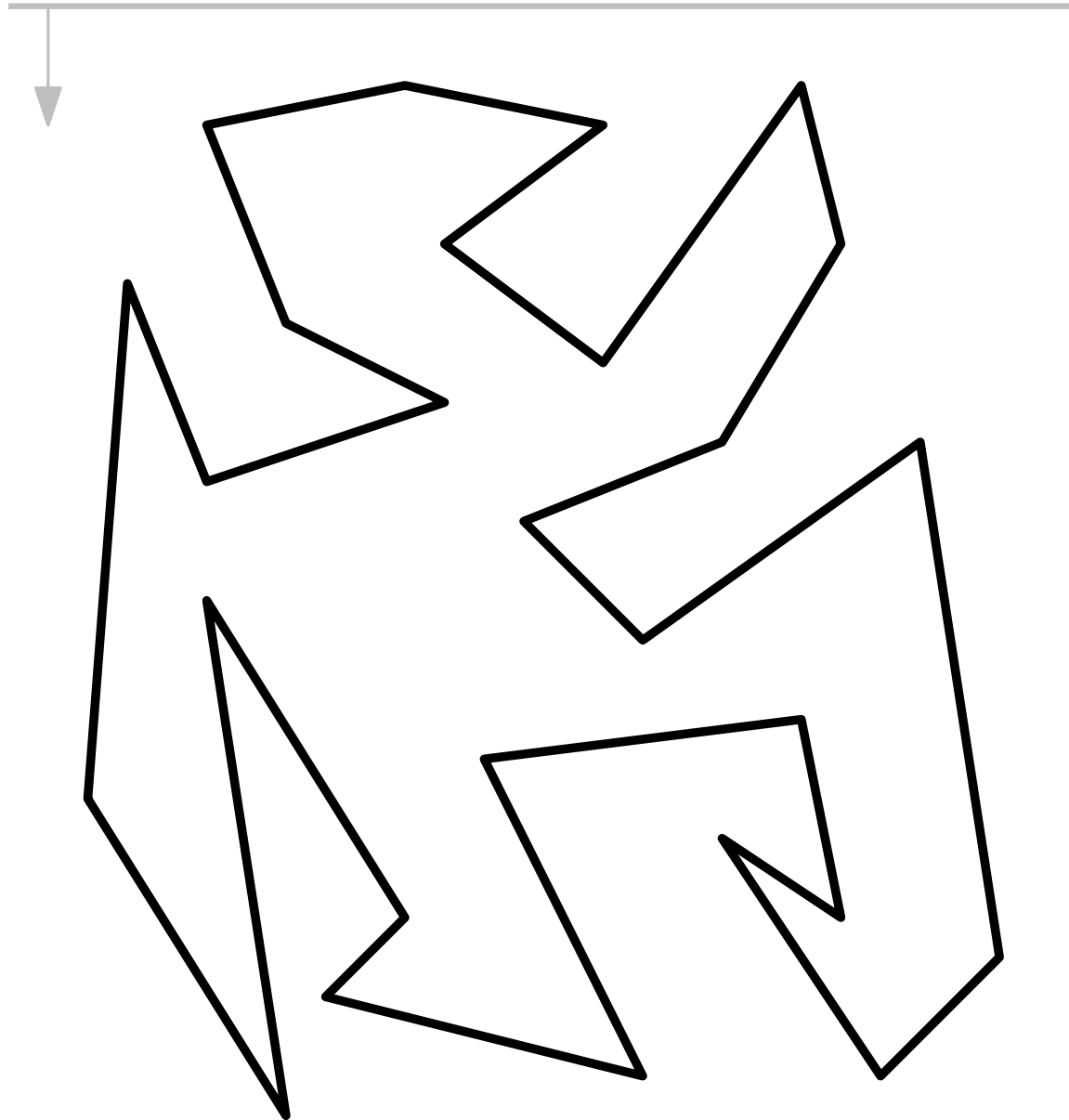


Merge vertex: Remove  $e$  from  $T$ .  
Update helper of  $f$  to  $v$ .

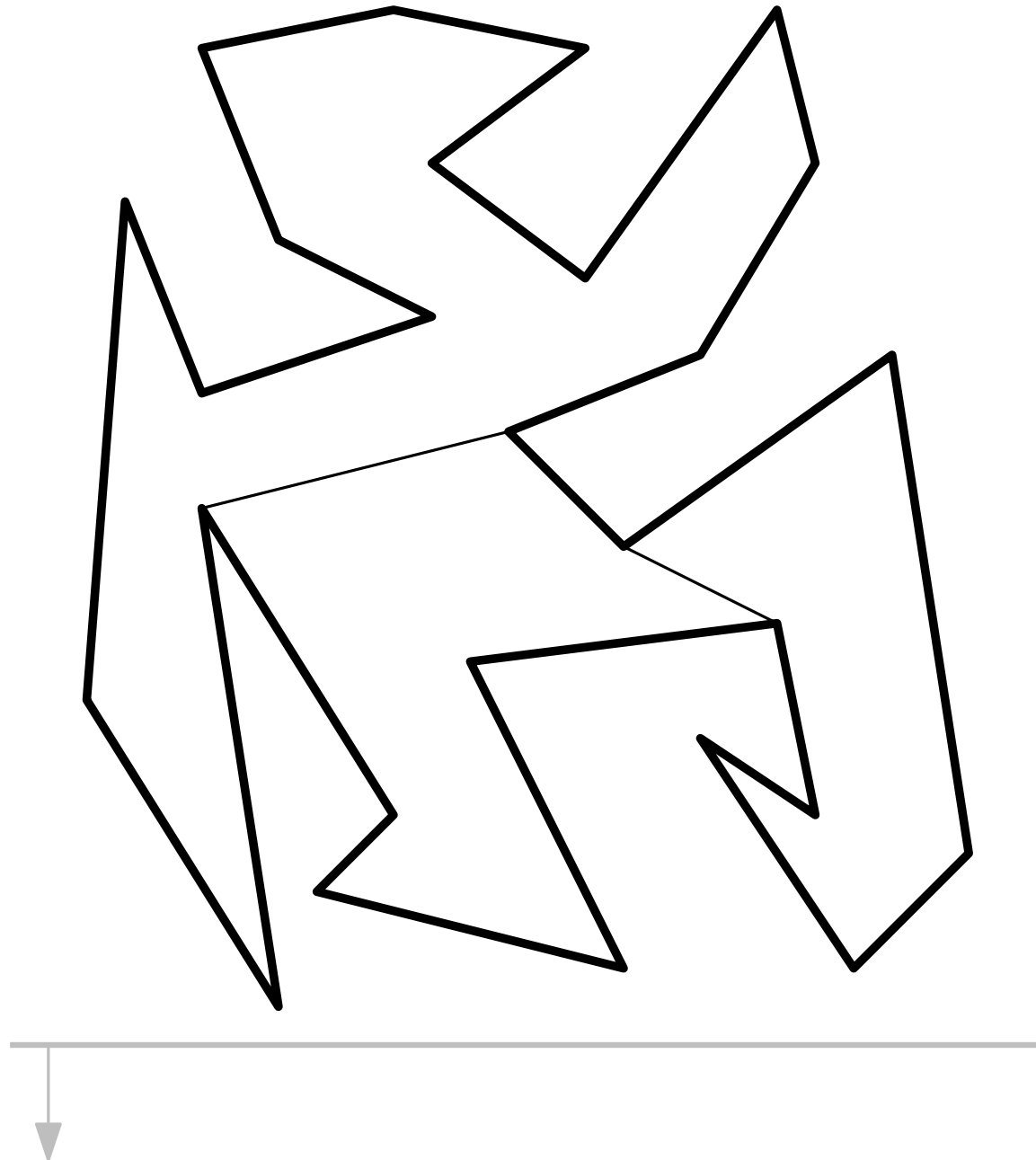




Sweep down, then up

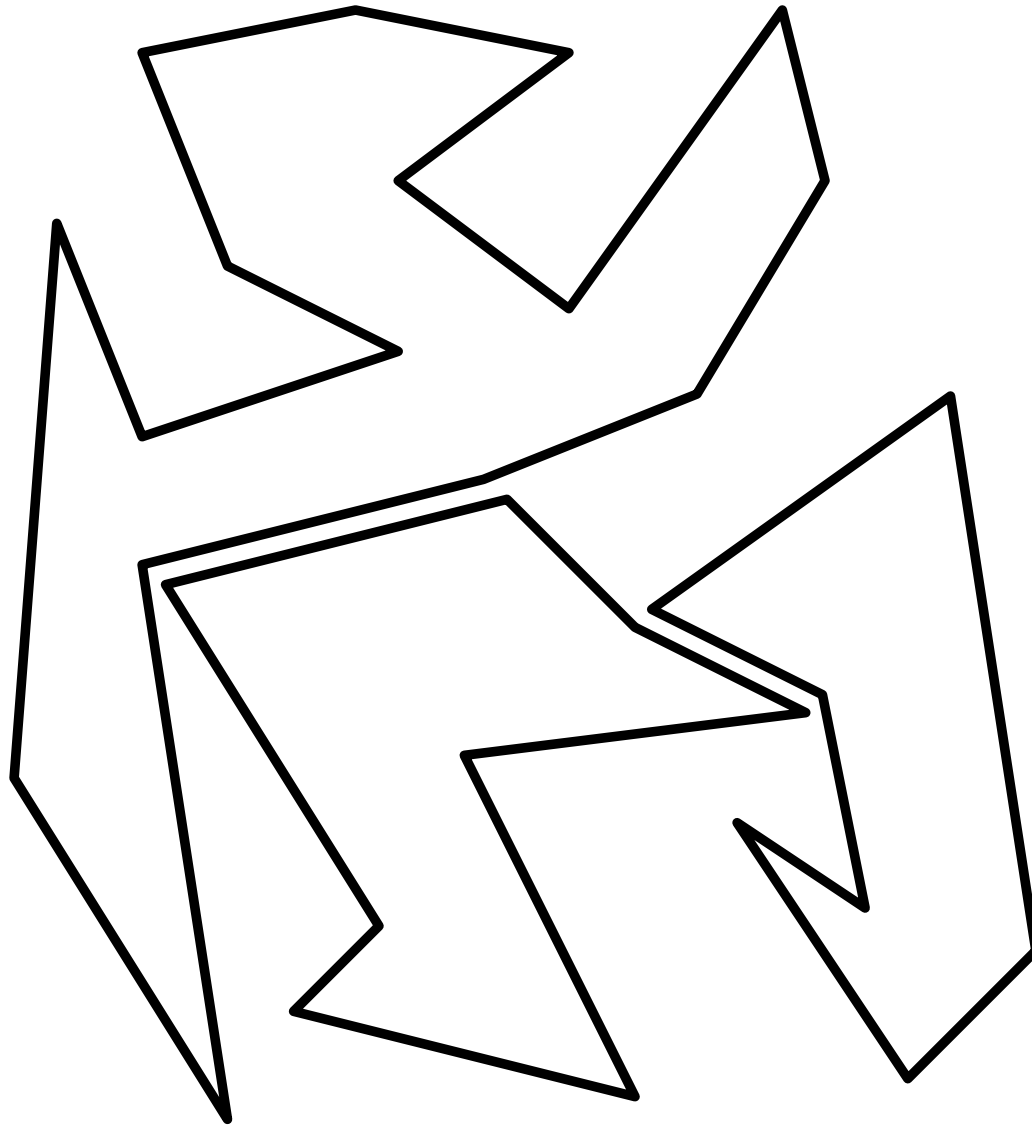


Sweep down, then up



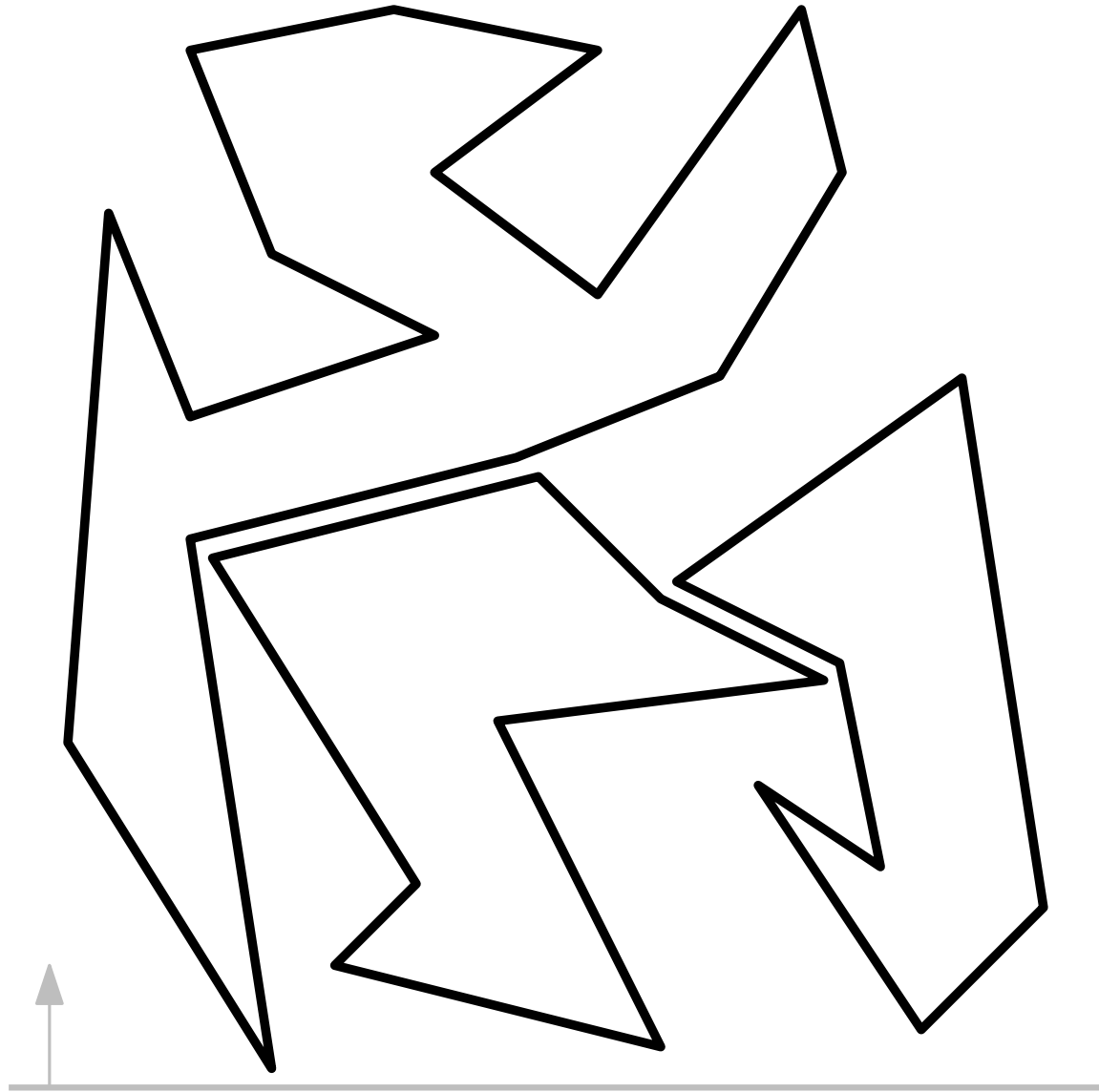
## Sweep down, then up

From below:  
Each polygon  
separately!



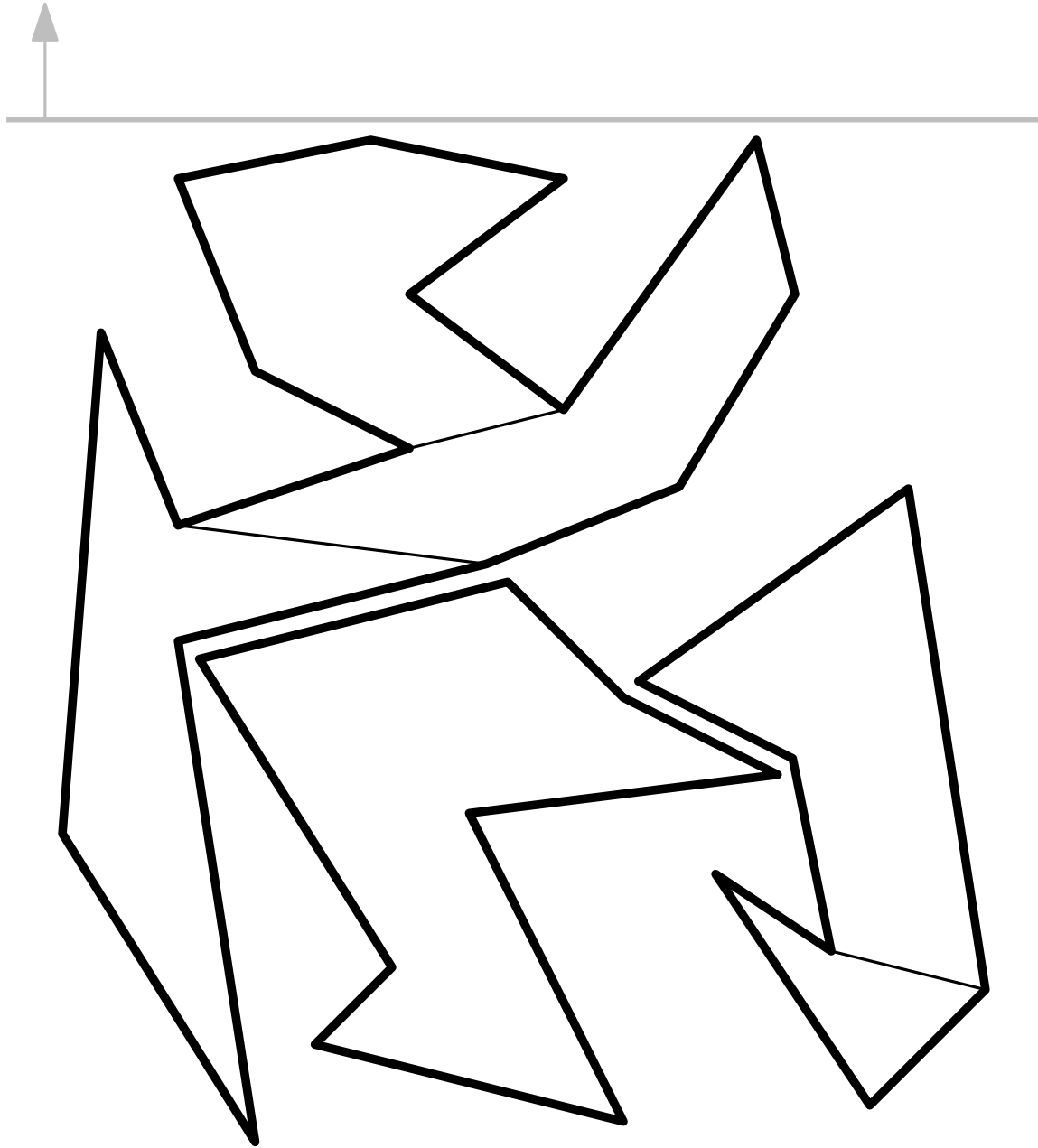
## Sweep down, then up

From below:  
Each polygon  
separately!

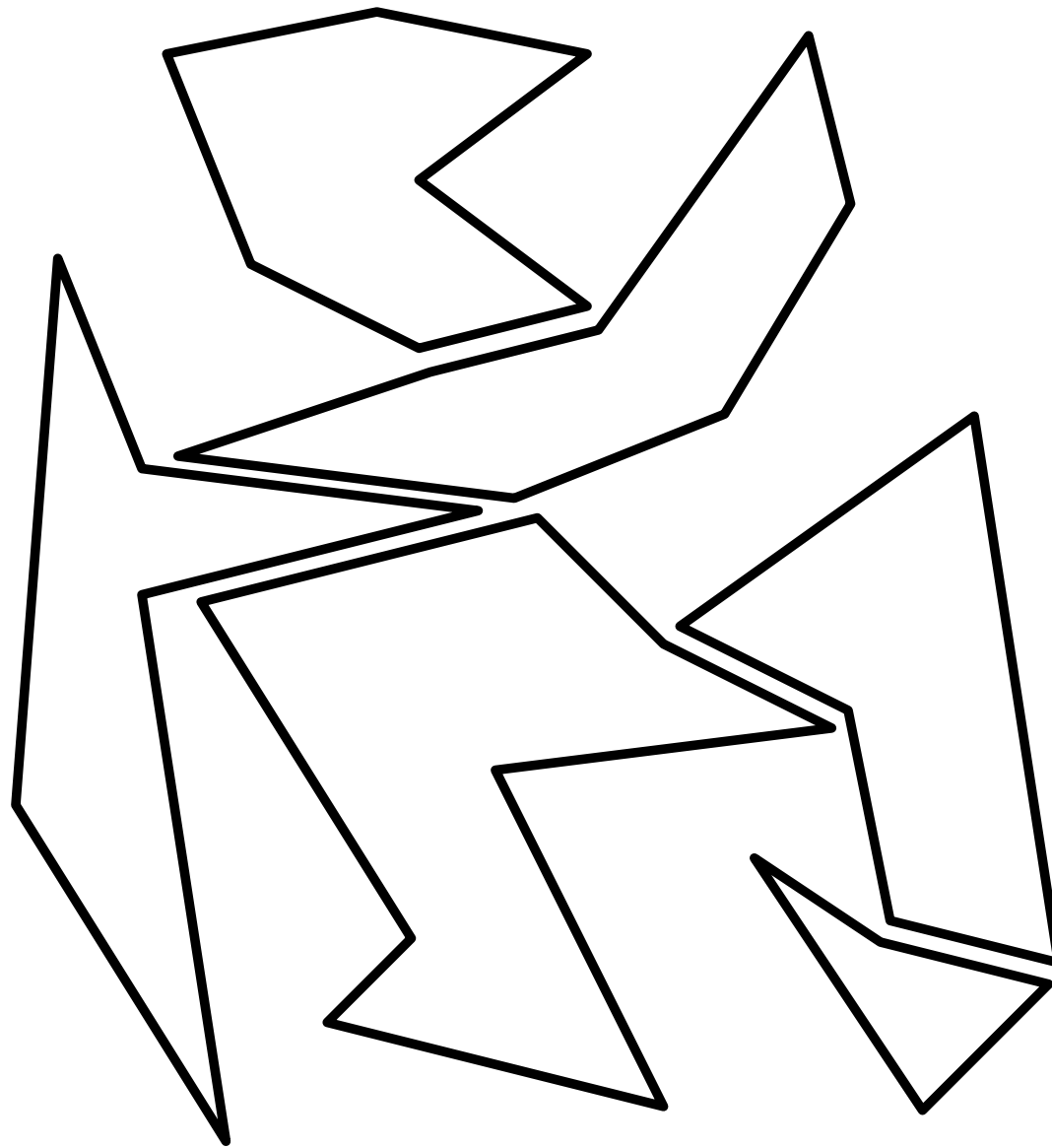


Sweep down, then up

From below:  
Each polygon  
separately!

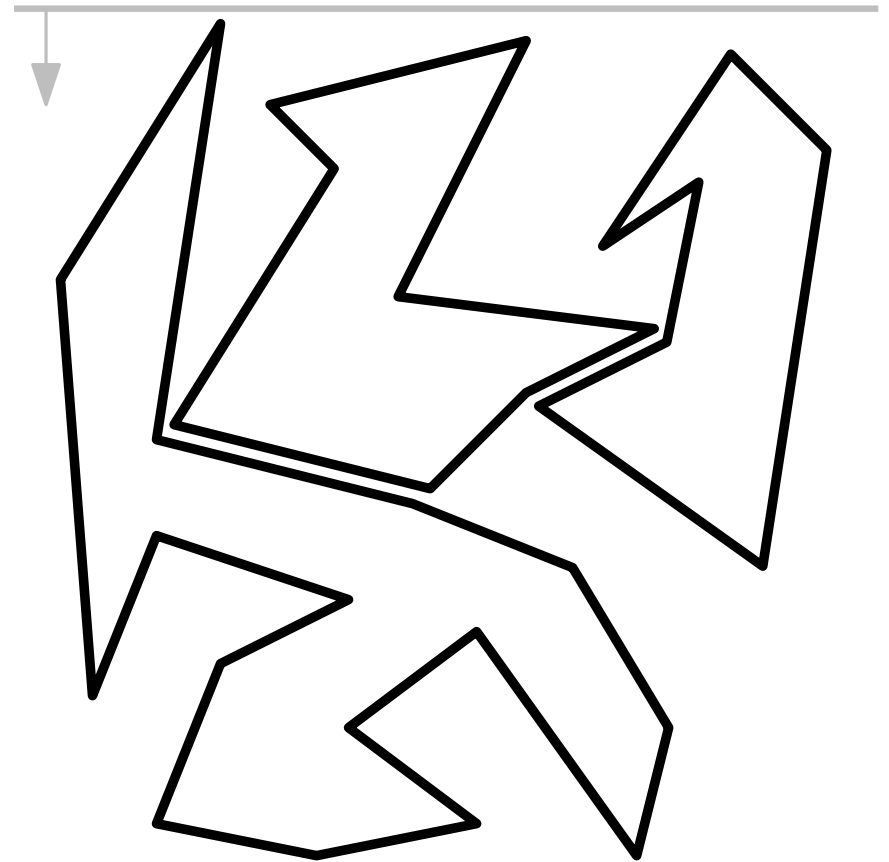
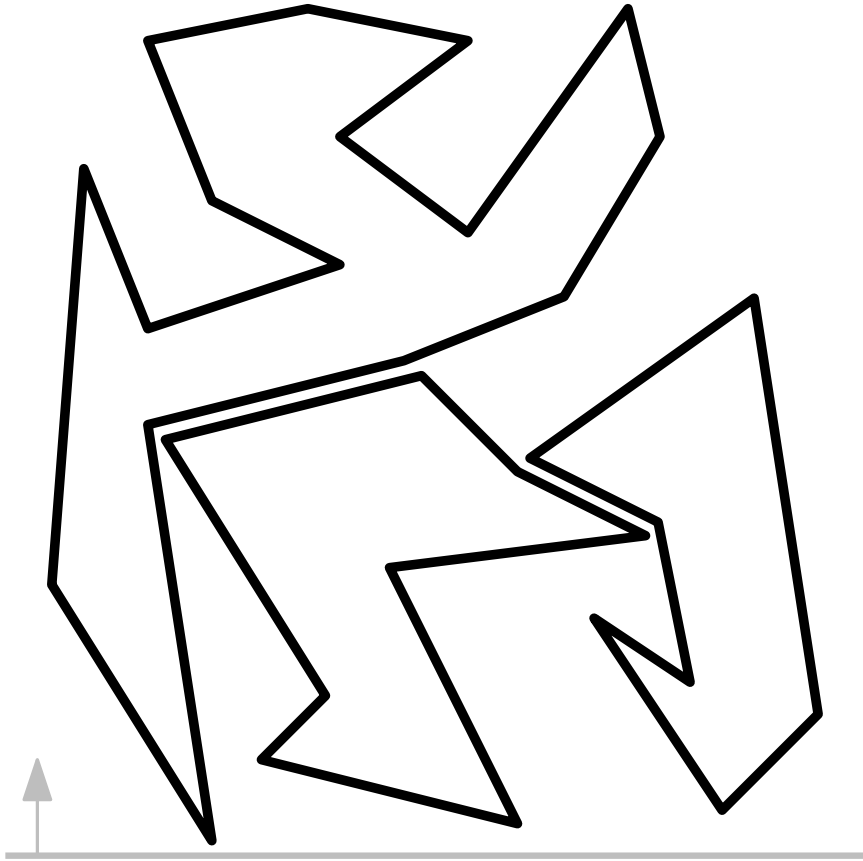


Sweep down, then up



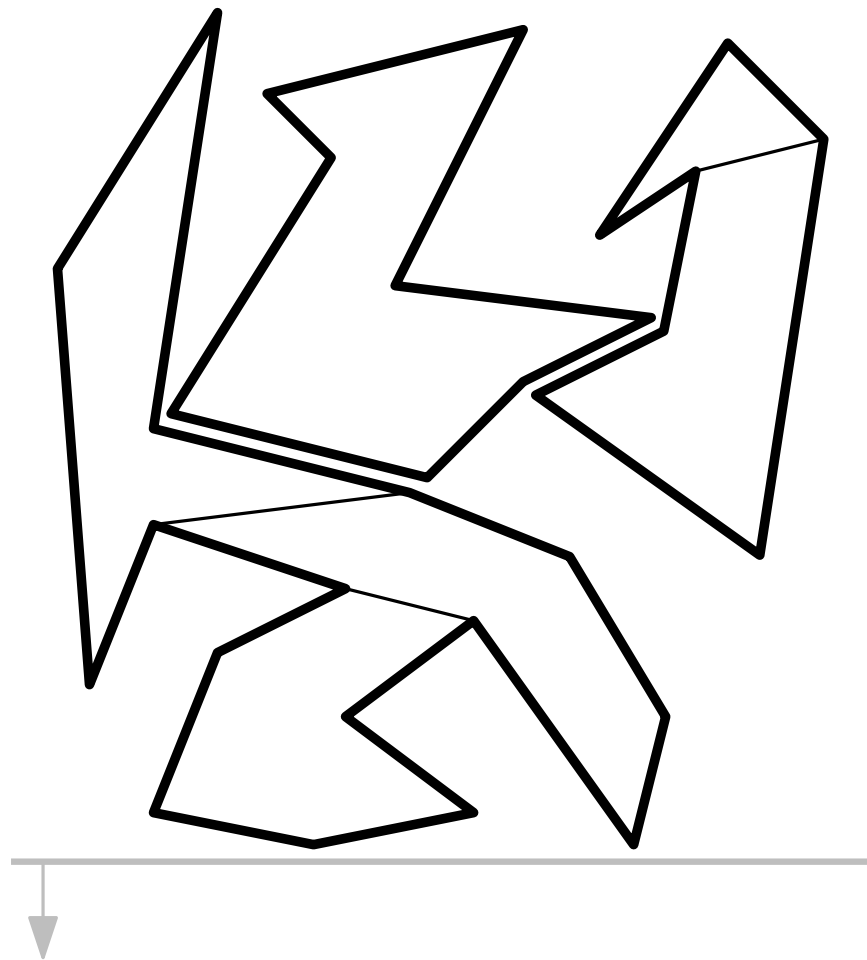
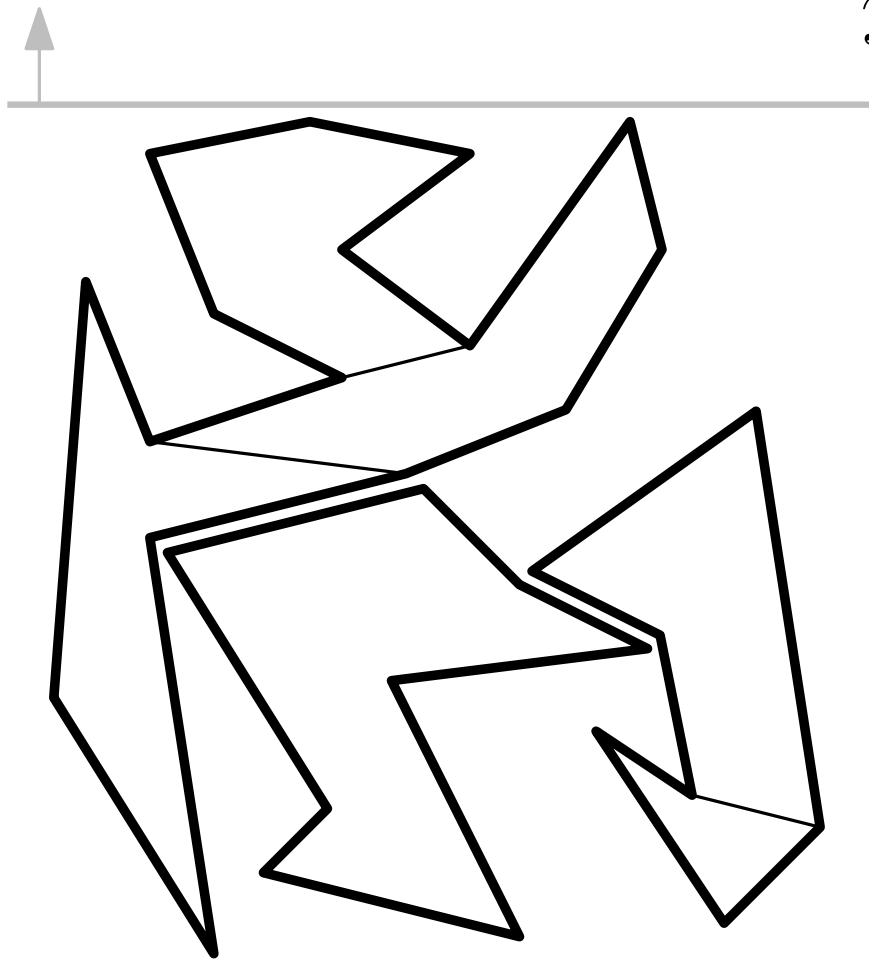
Sweep down, then up

$$y \mapsto -y$$



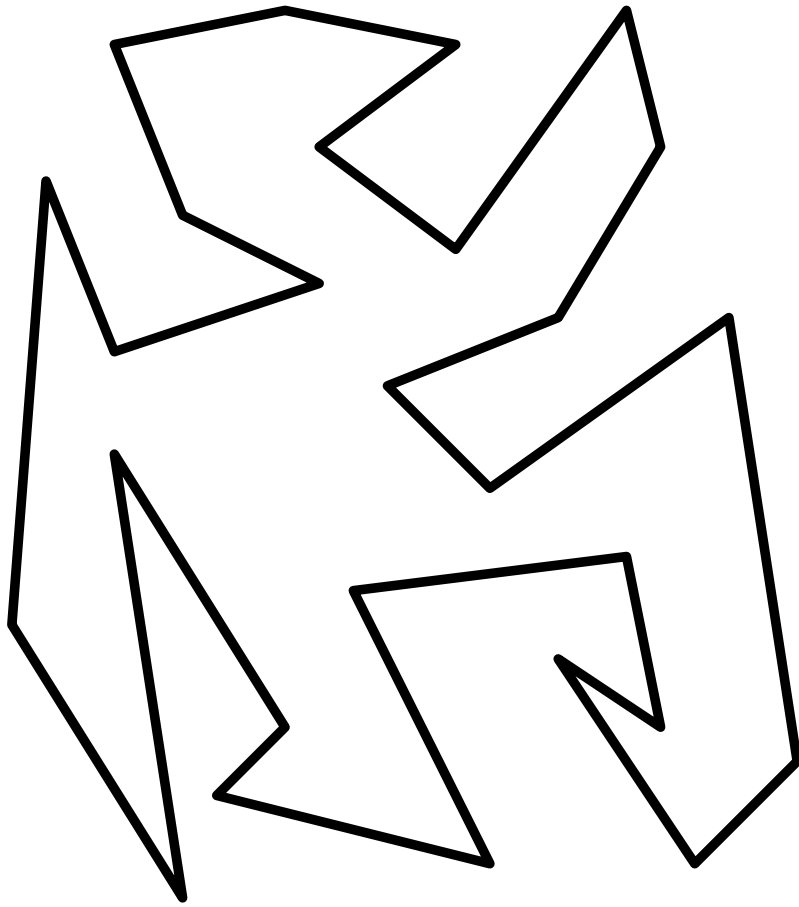
Sweep down, then up

$$y \mapsto -y$$

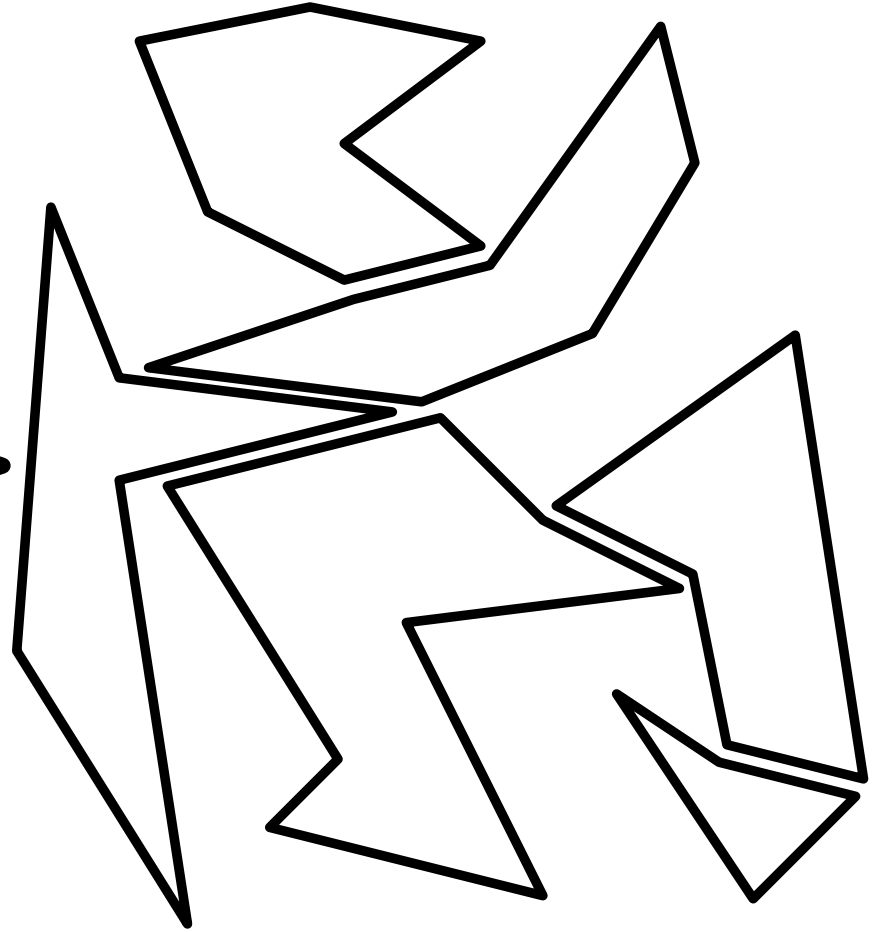




How many vertices?



$n$  vertices



2 extra vertices per diagonal and  
 $\leq n - 3$  diagonals  $\Rightarrow$   
 $\leq n + 2(n - 3) = 3n - 6$  vertices

# Efficiency

Sorting vertices by  $y$ -coordinates:  $O(n \log n)$ .

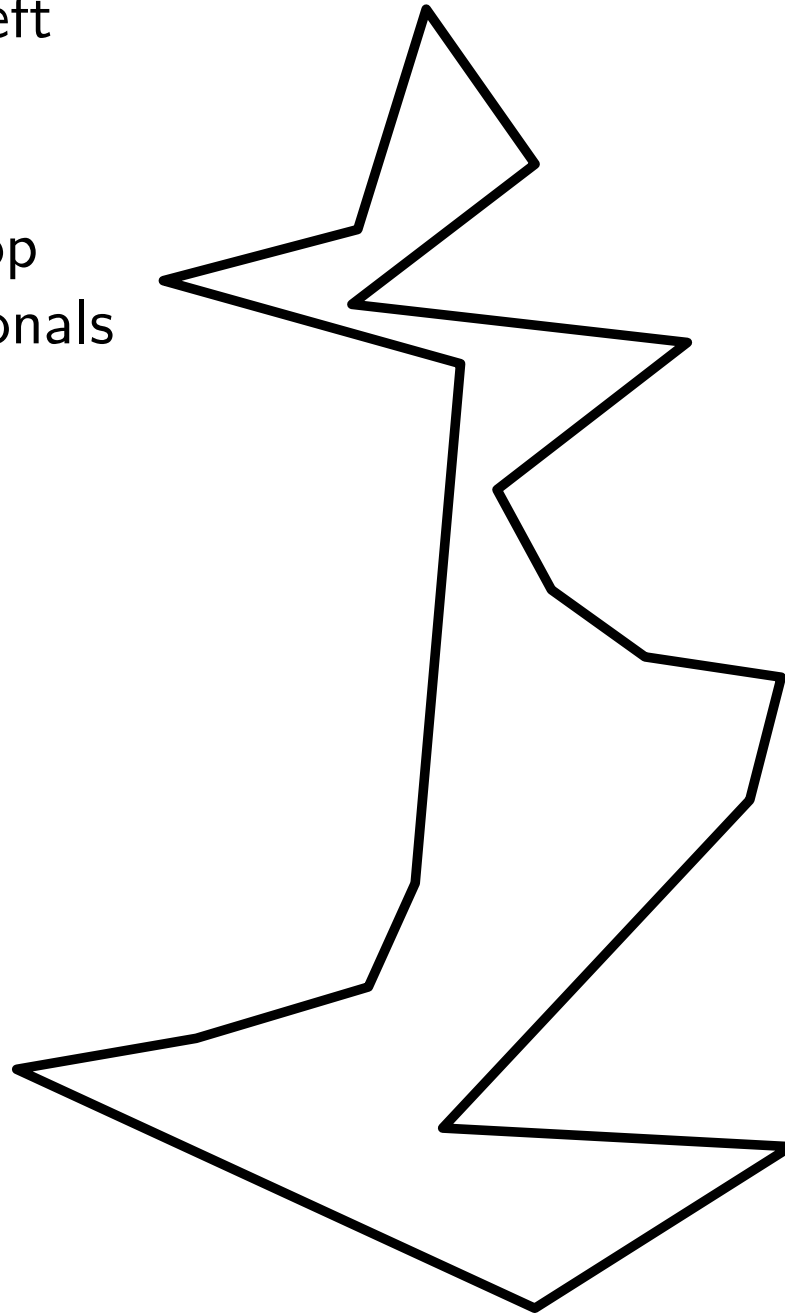
Handling  $n$  events:  $O(n \log n)$  time.

Doing it again from below (at most  $3n - 6$  vertices now).

In total  $O(n \log n)$ .

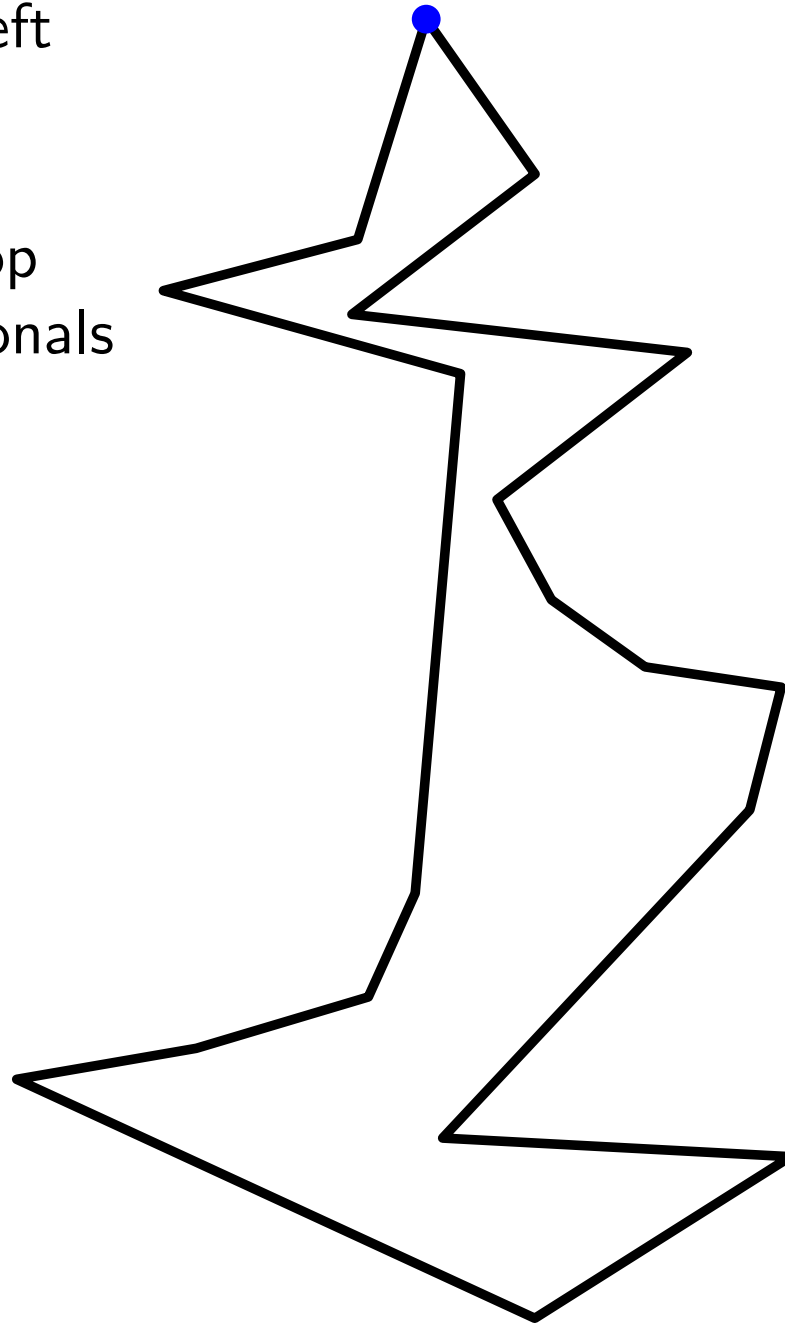
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



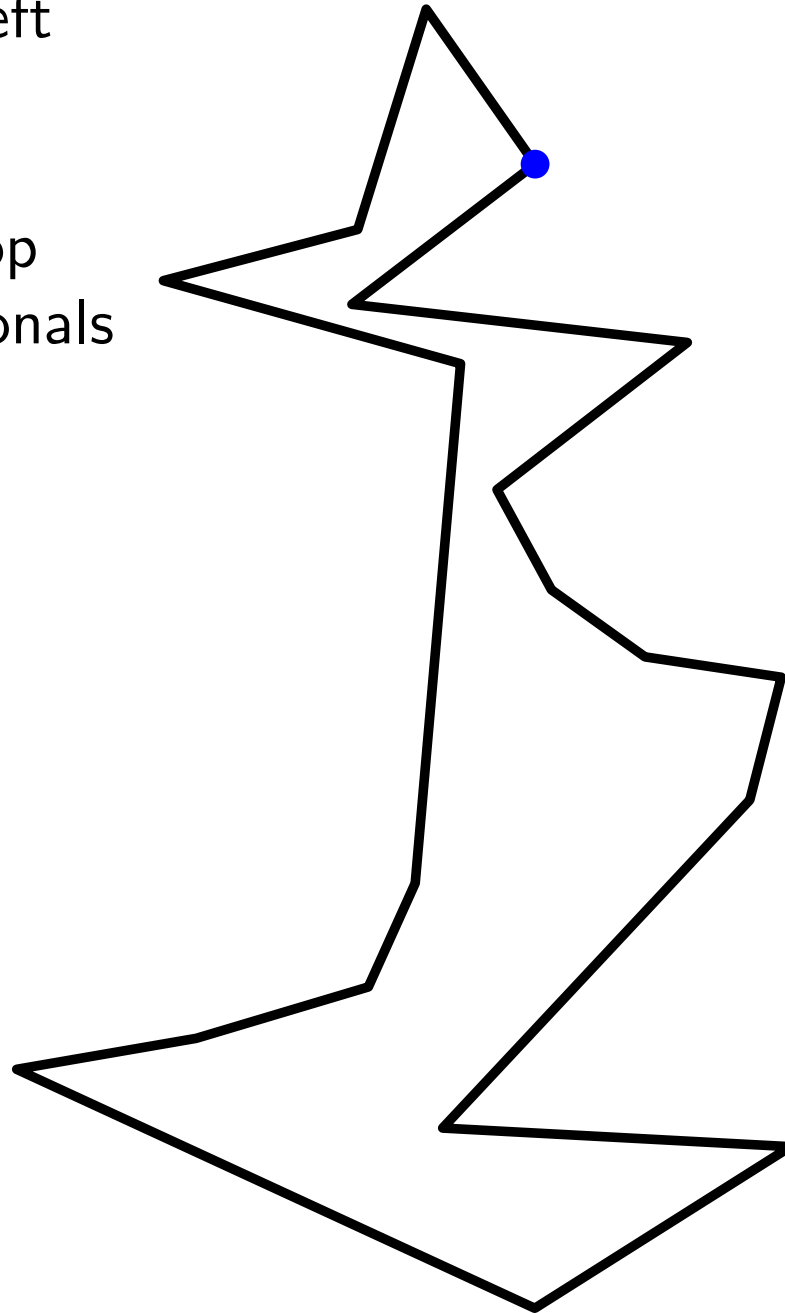
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



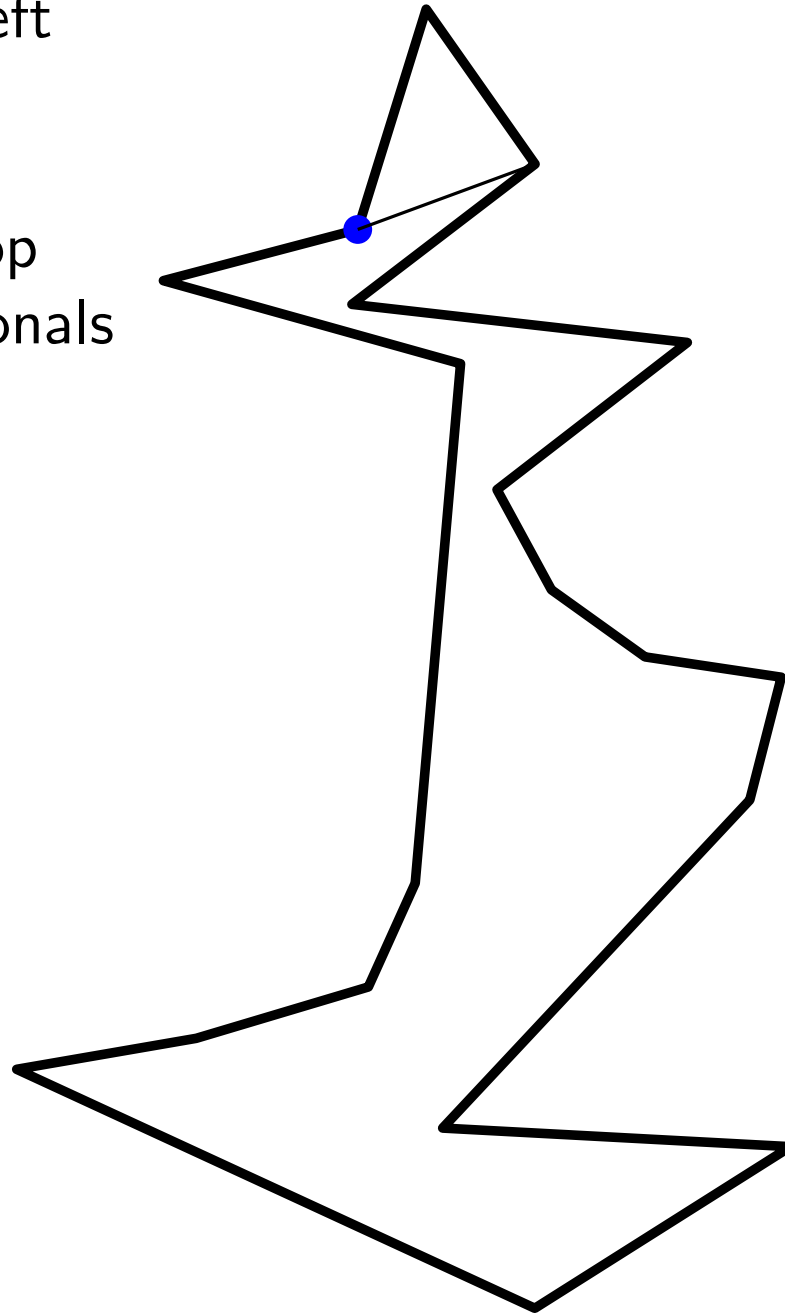
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



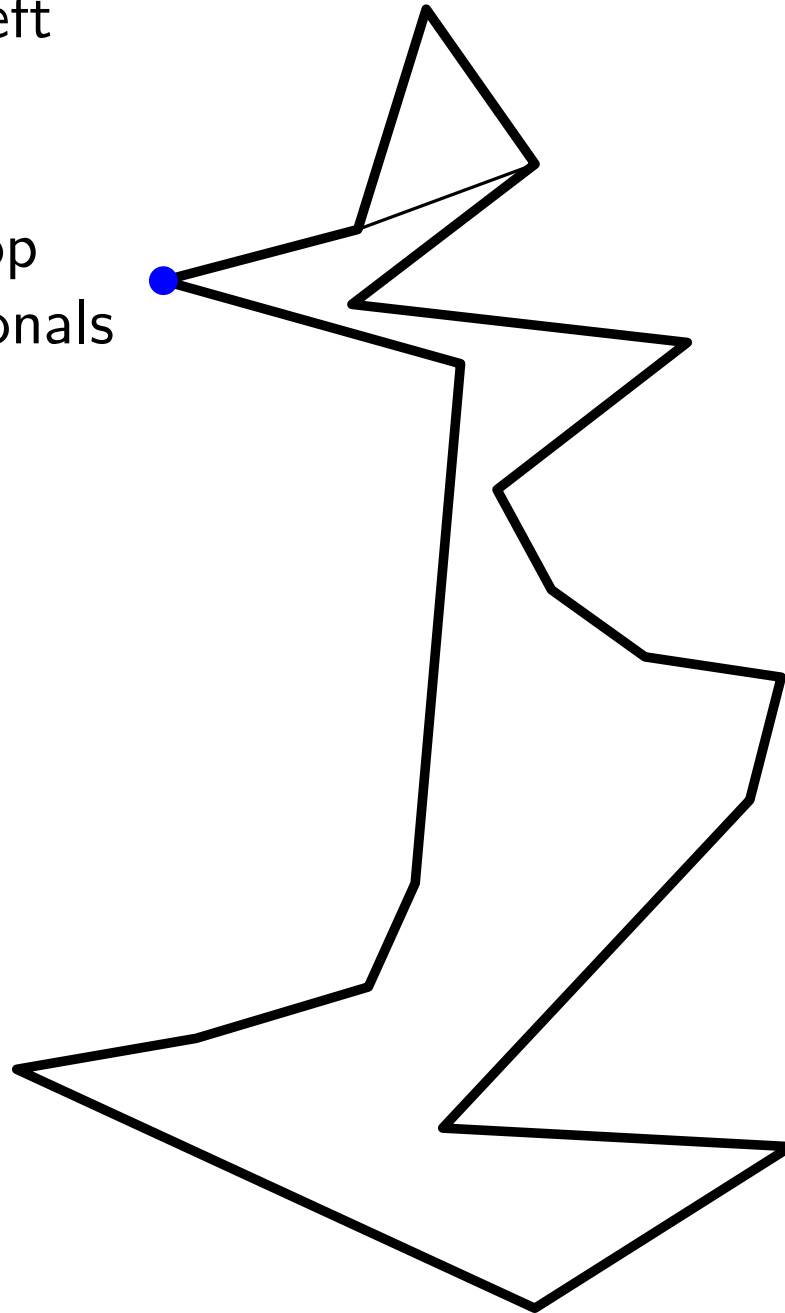
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



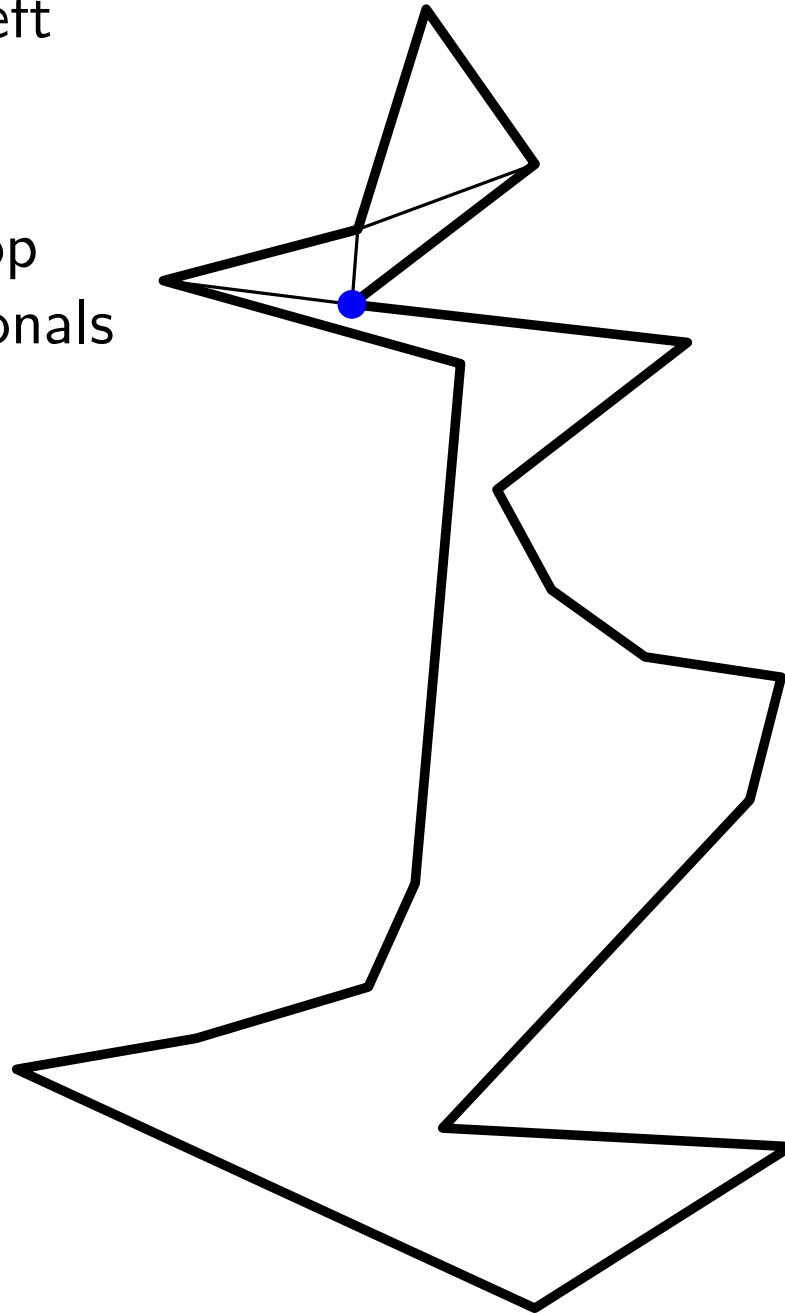
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



# Triangulating a $y$ -monotone polygon

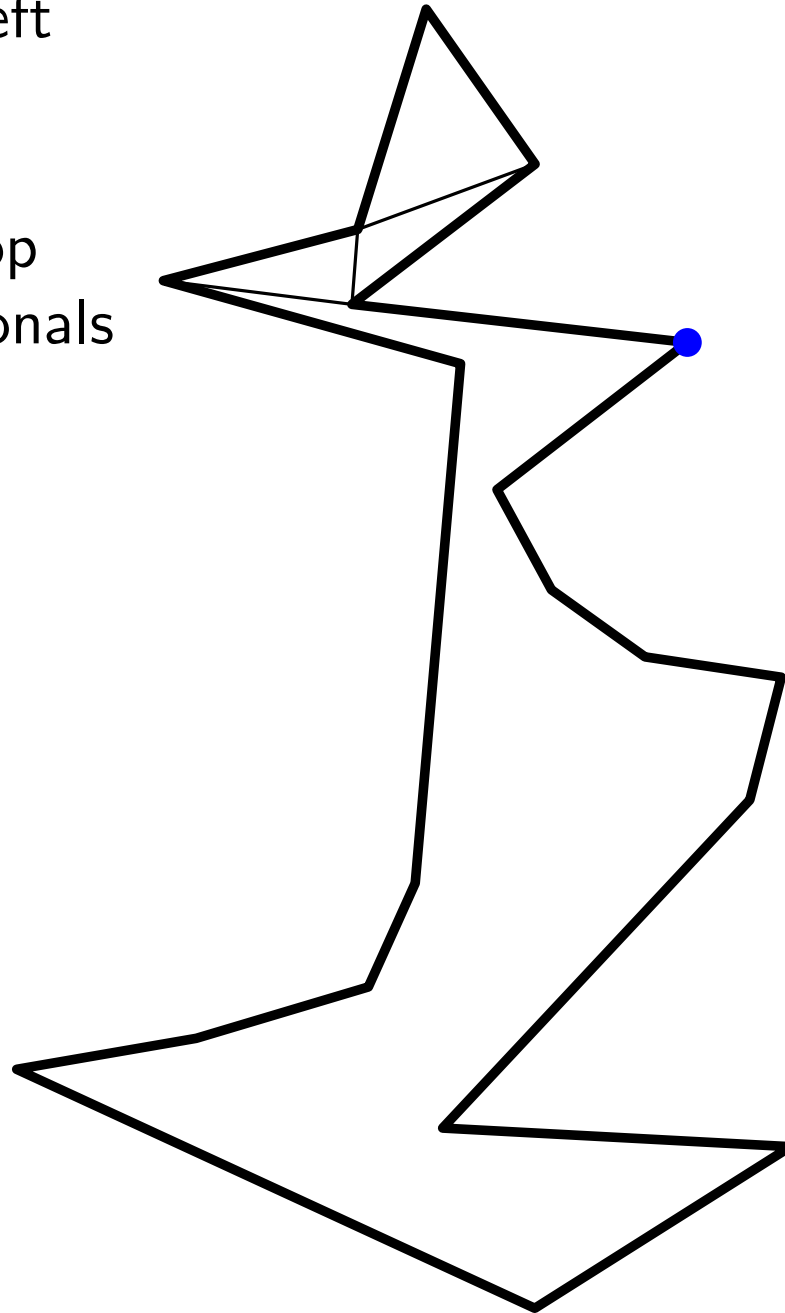
- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above





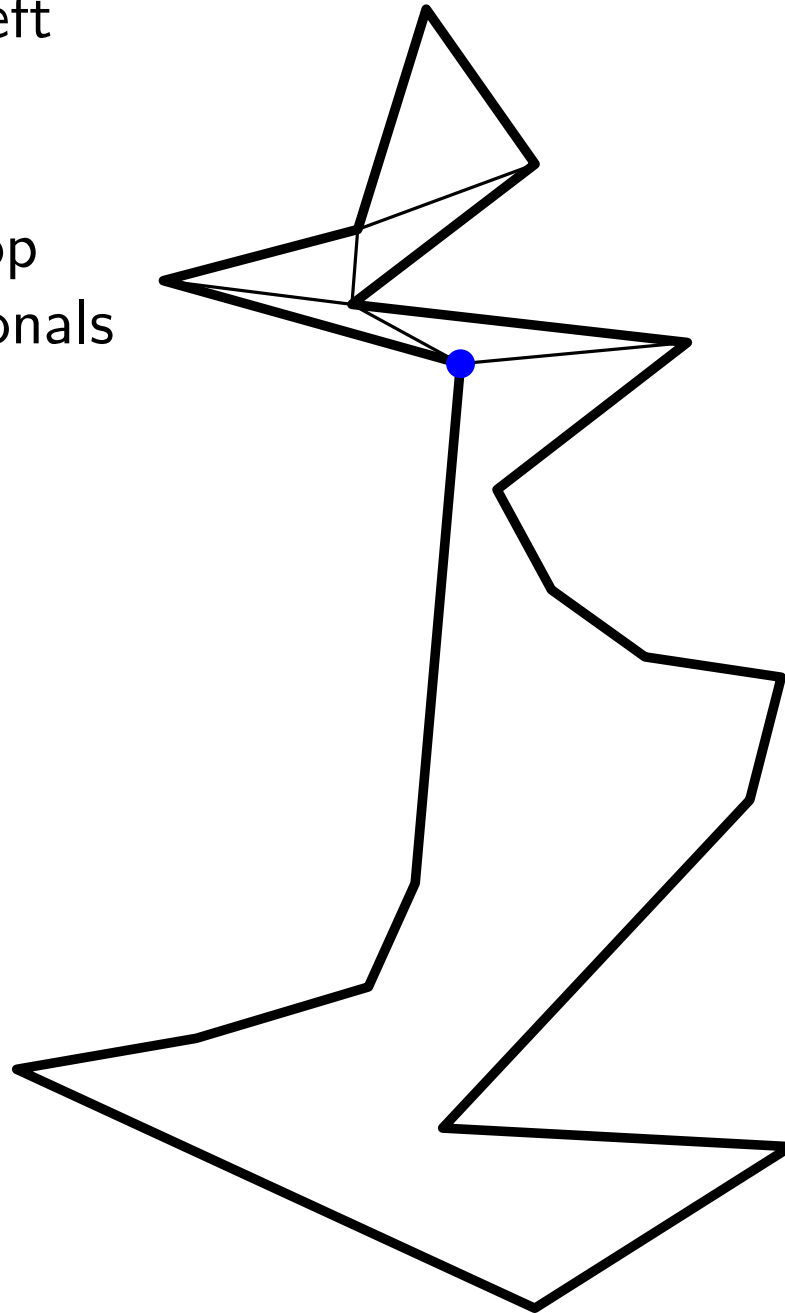
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



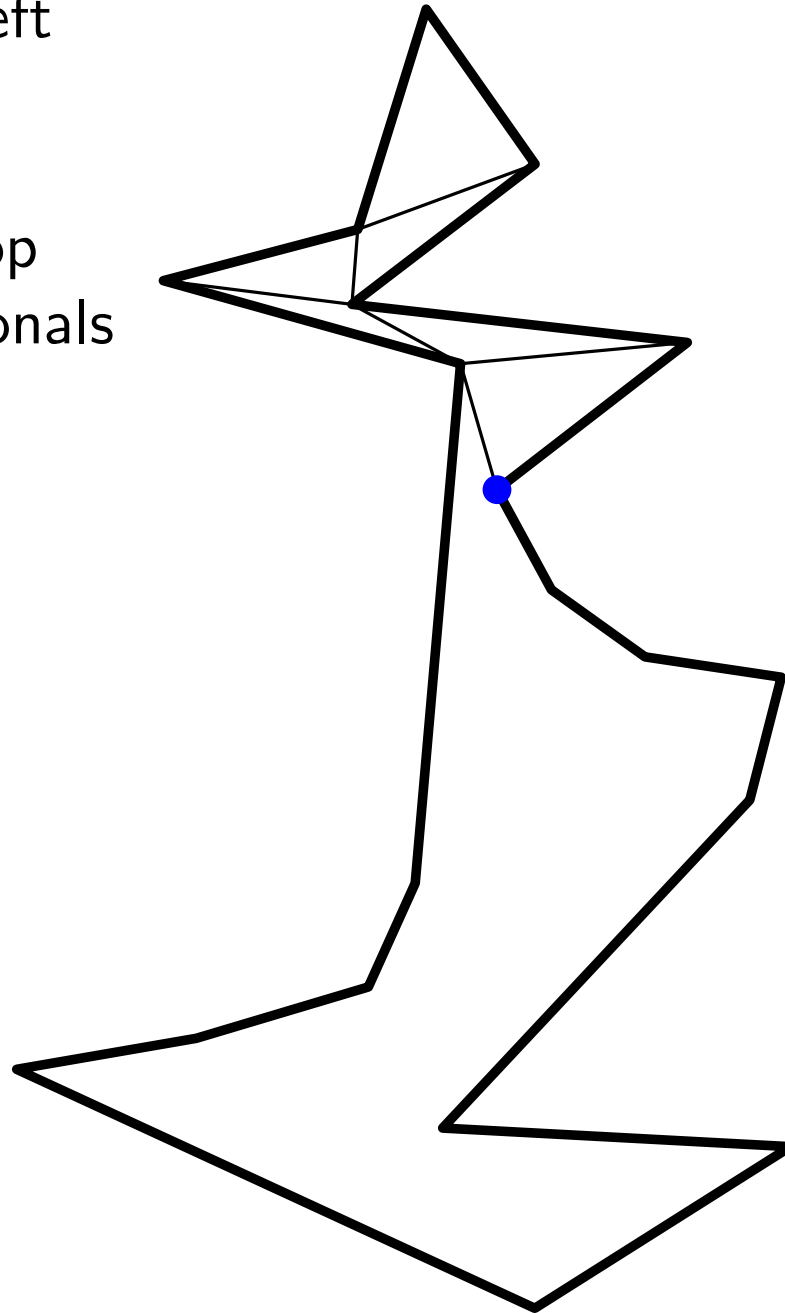
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



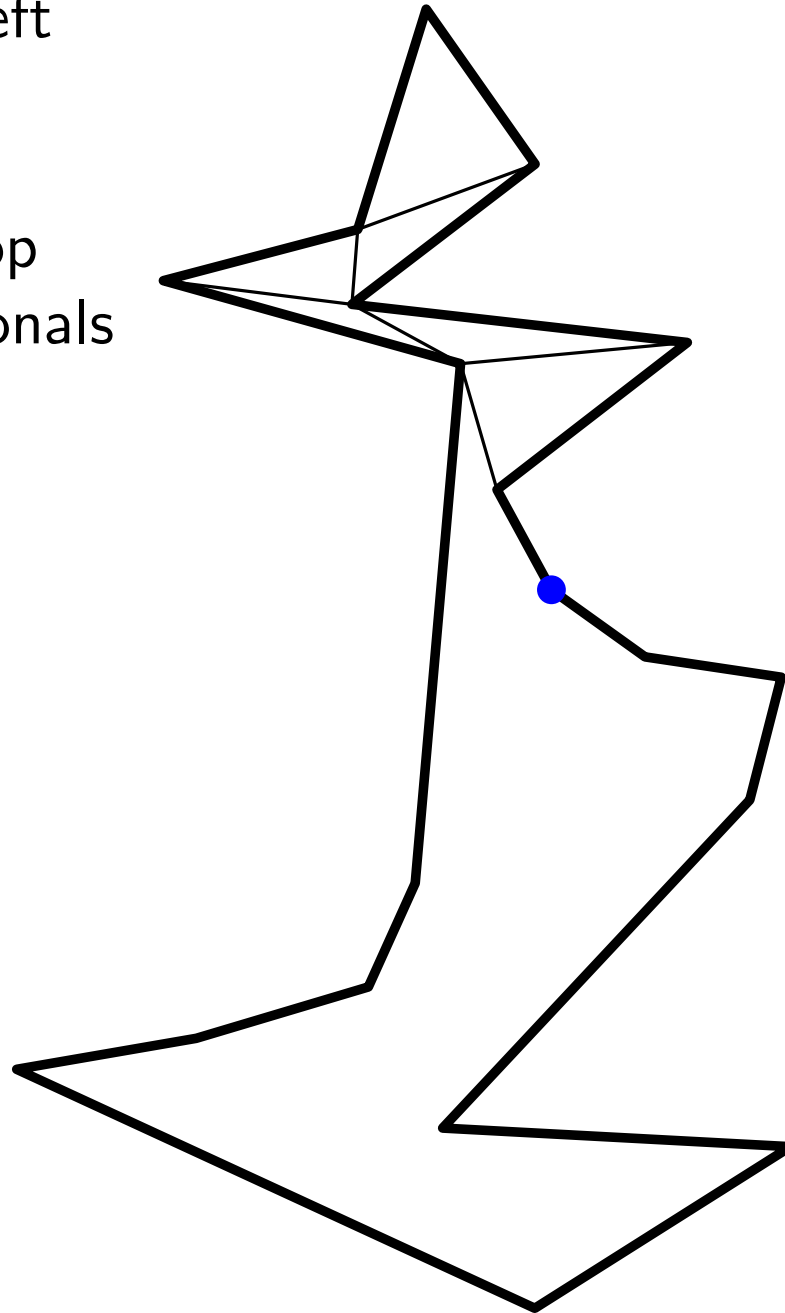
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



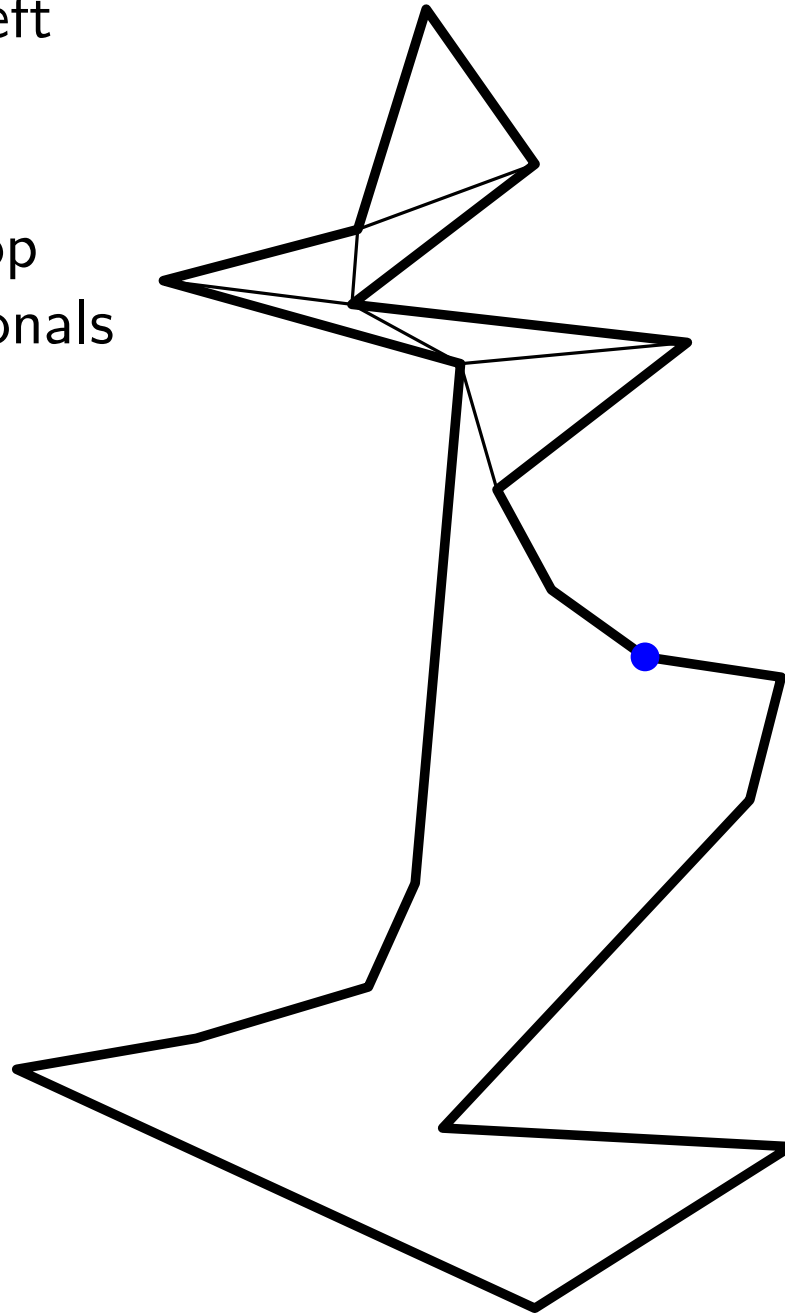
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



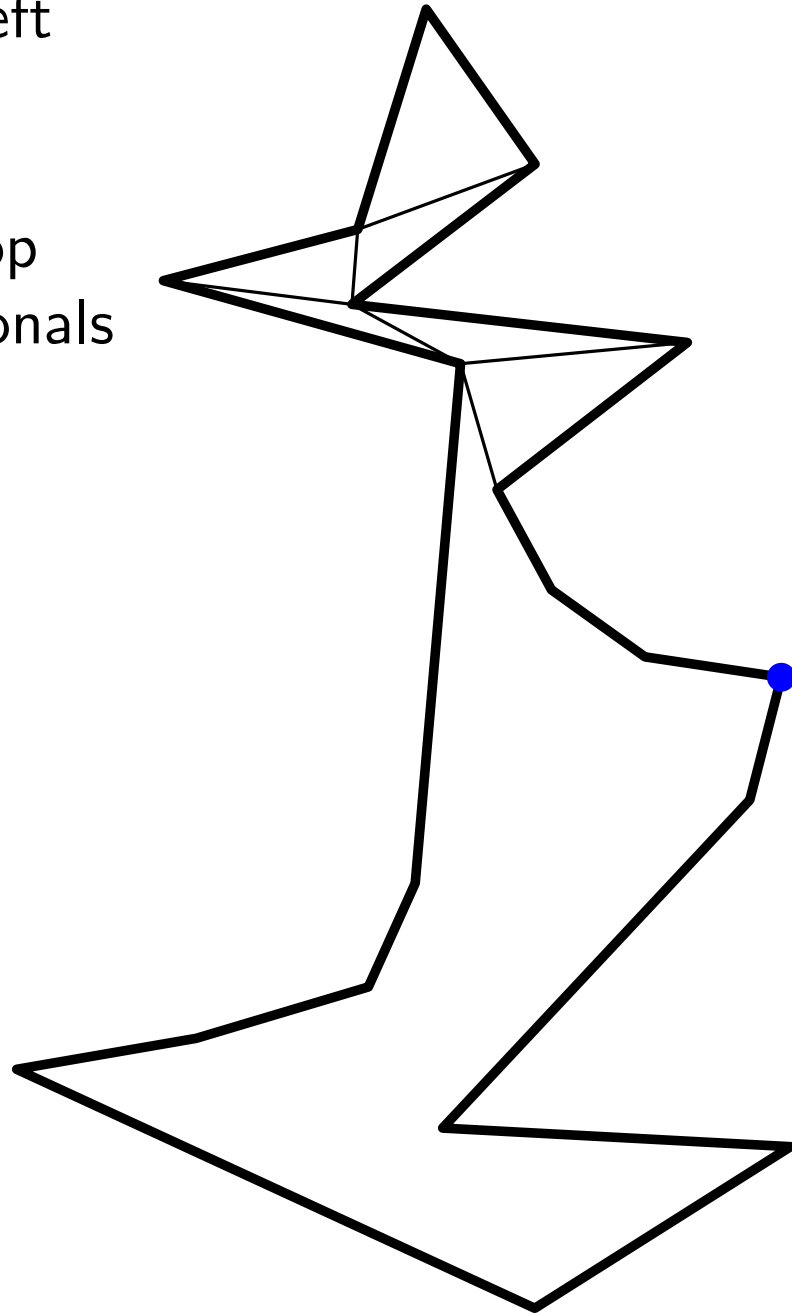
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



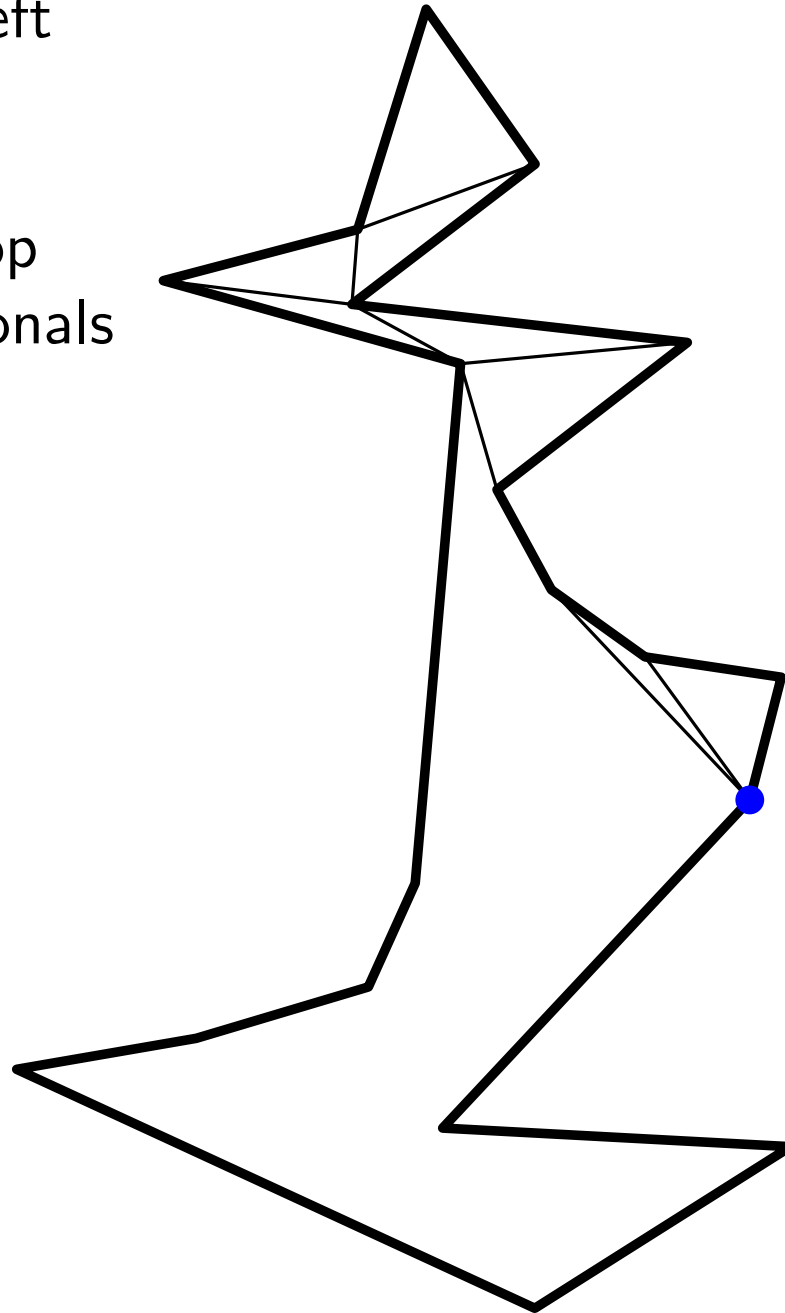
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



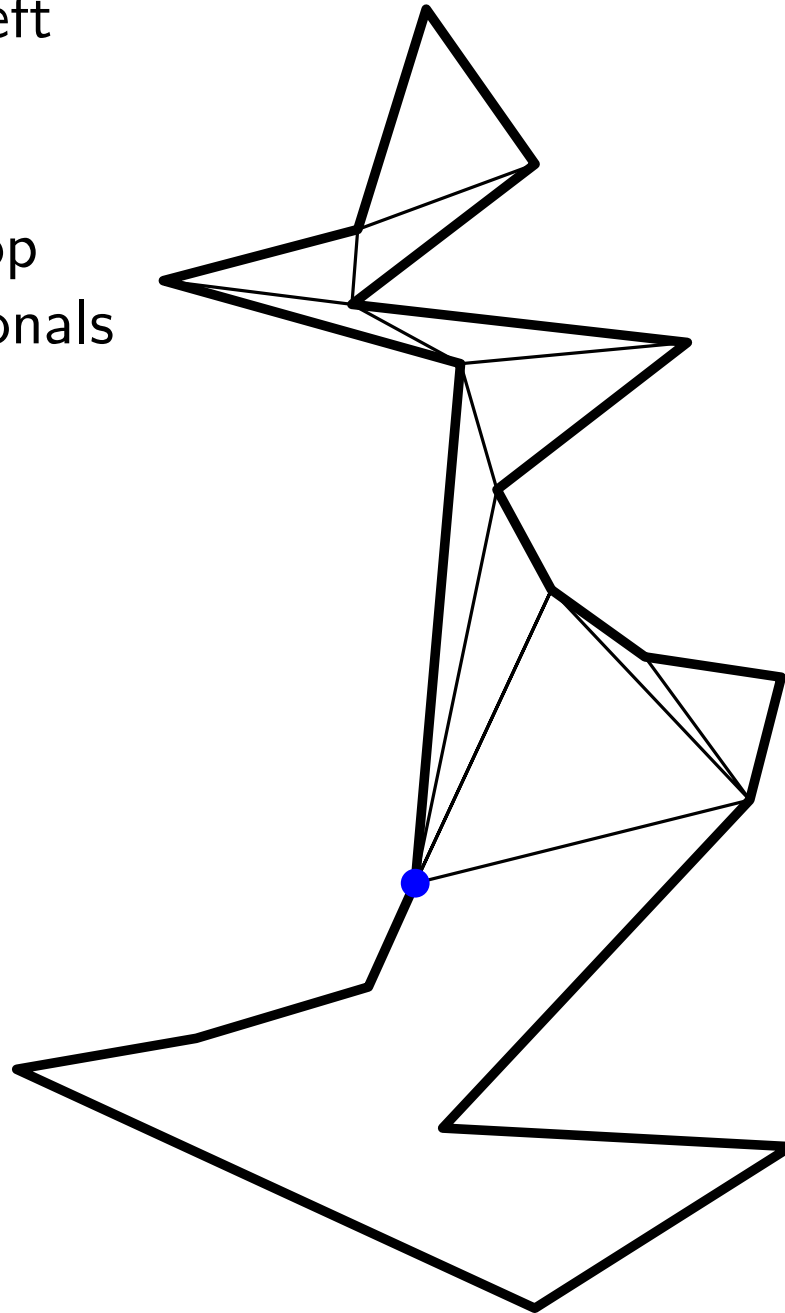
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



# Triangulating a $y$ -monotone polygon

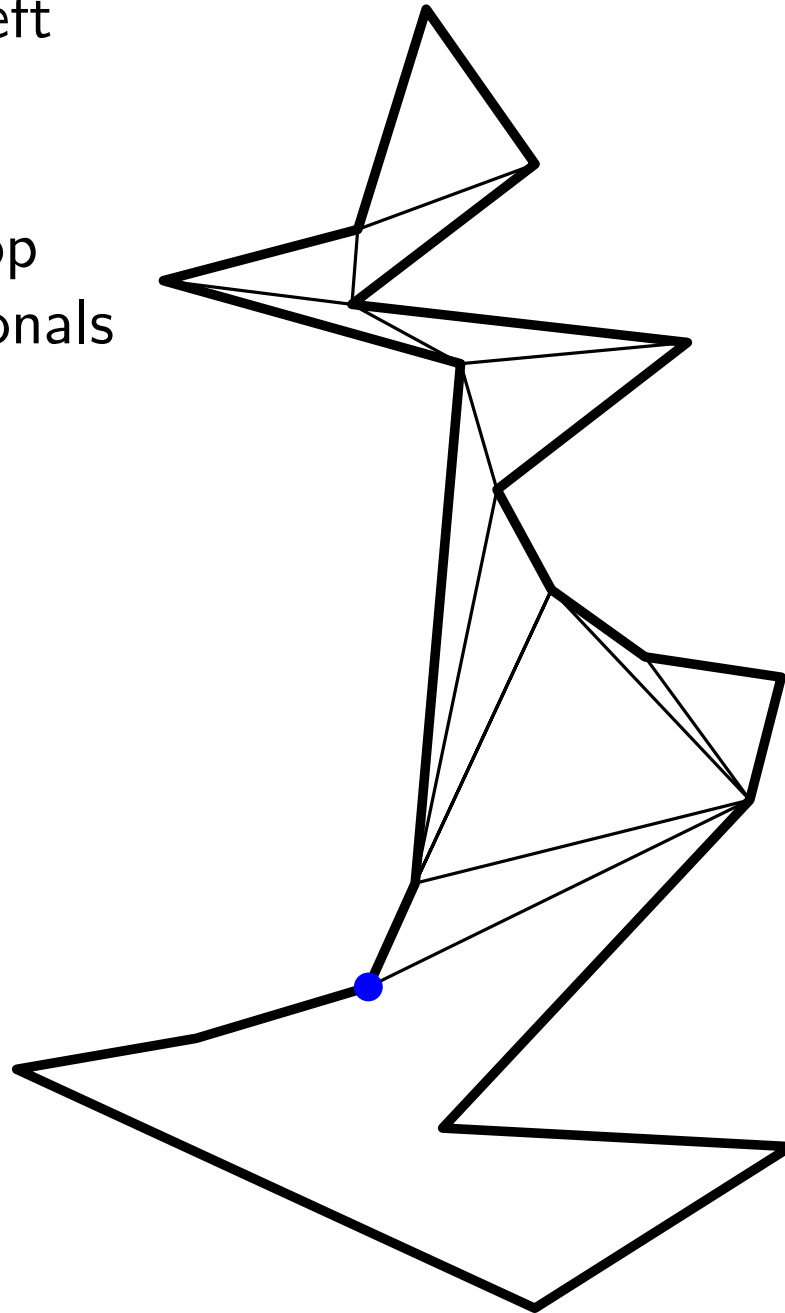
- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above





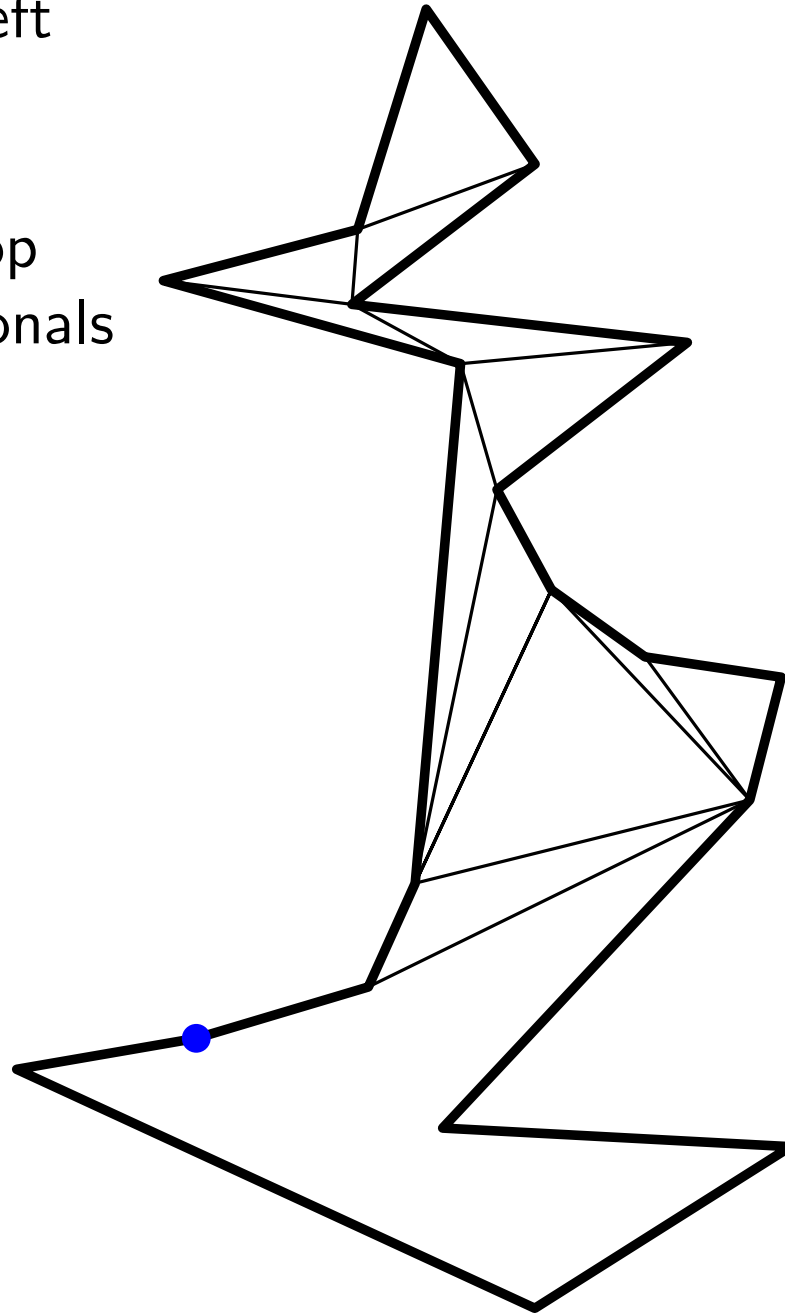
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



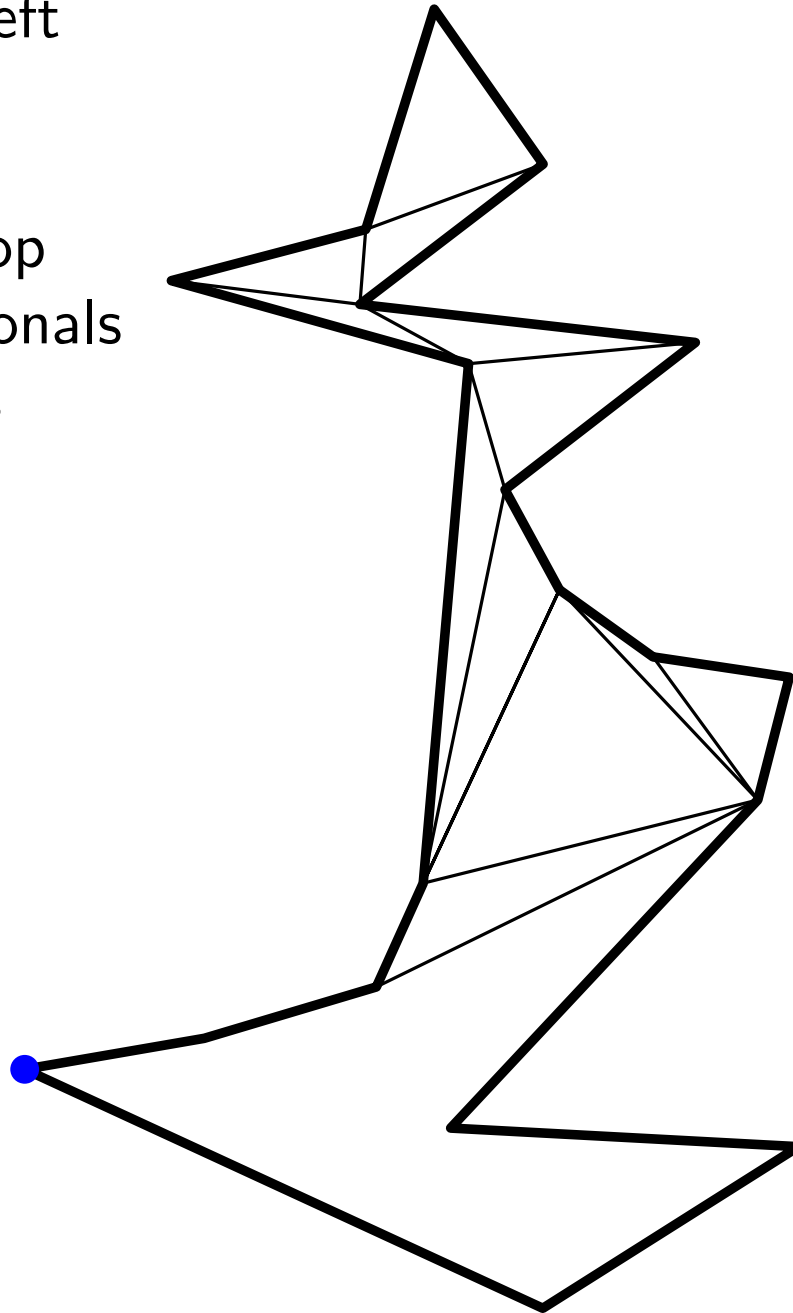
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



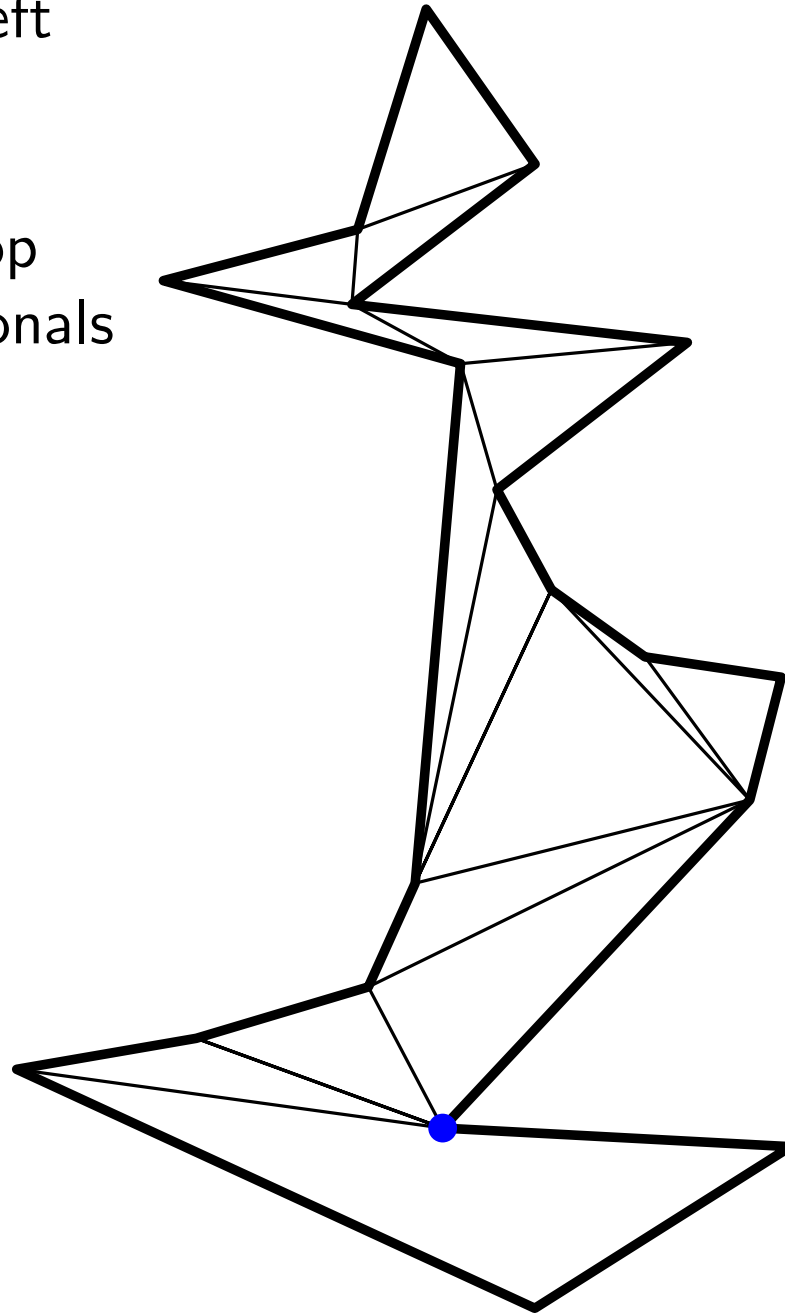
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



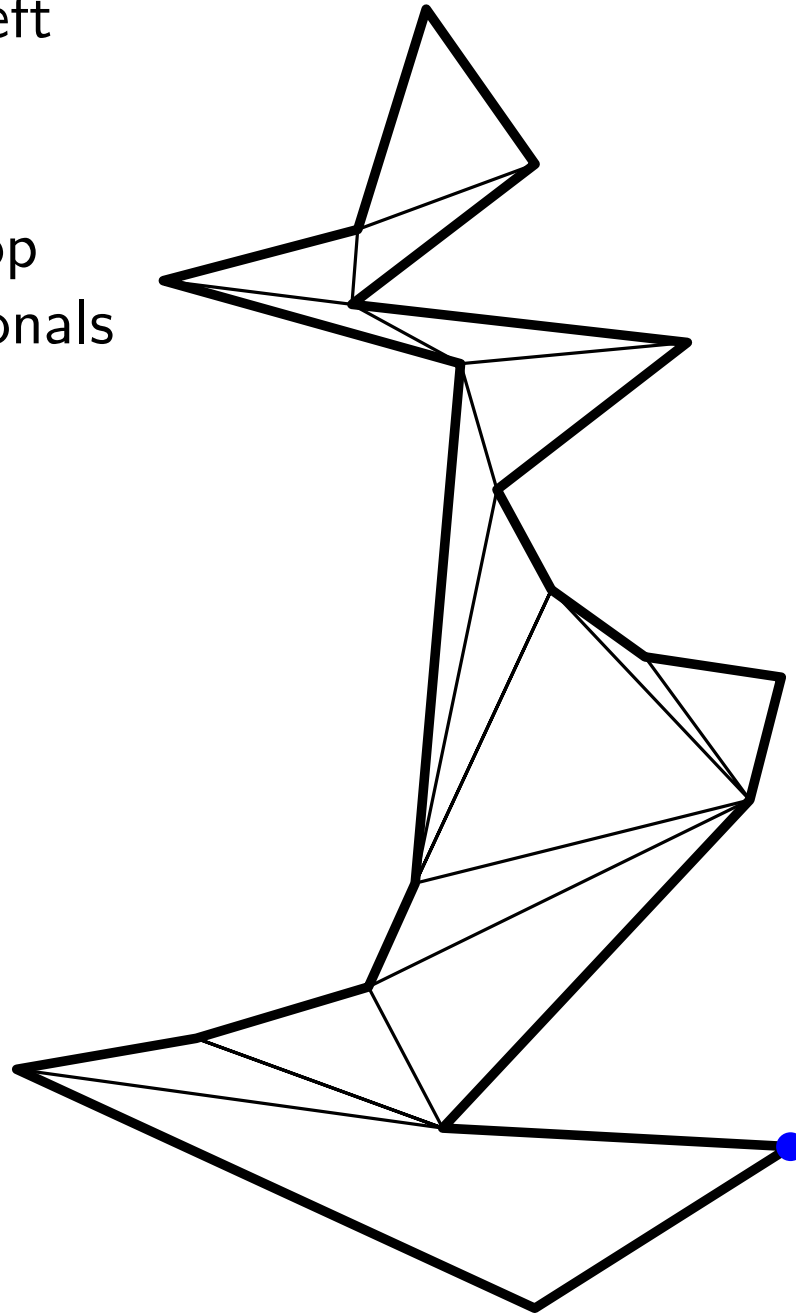
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



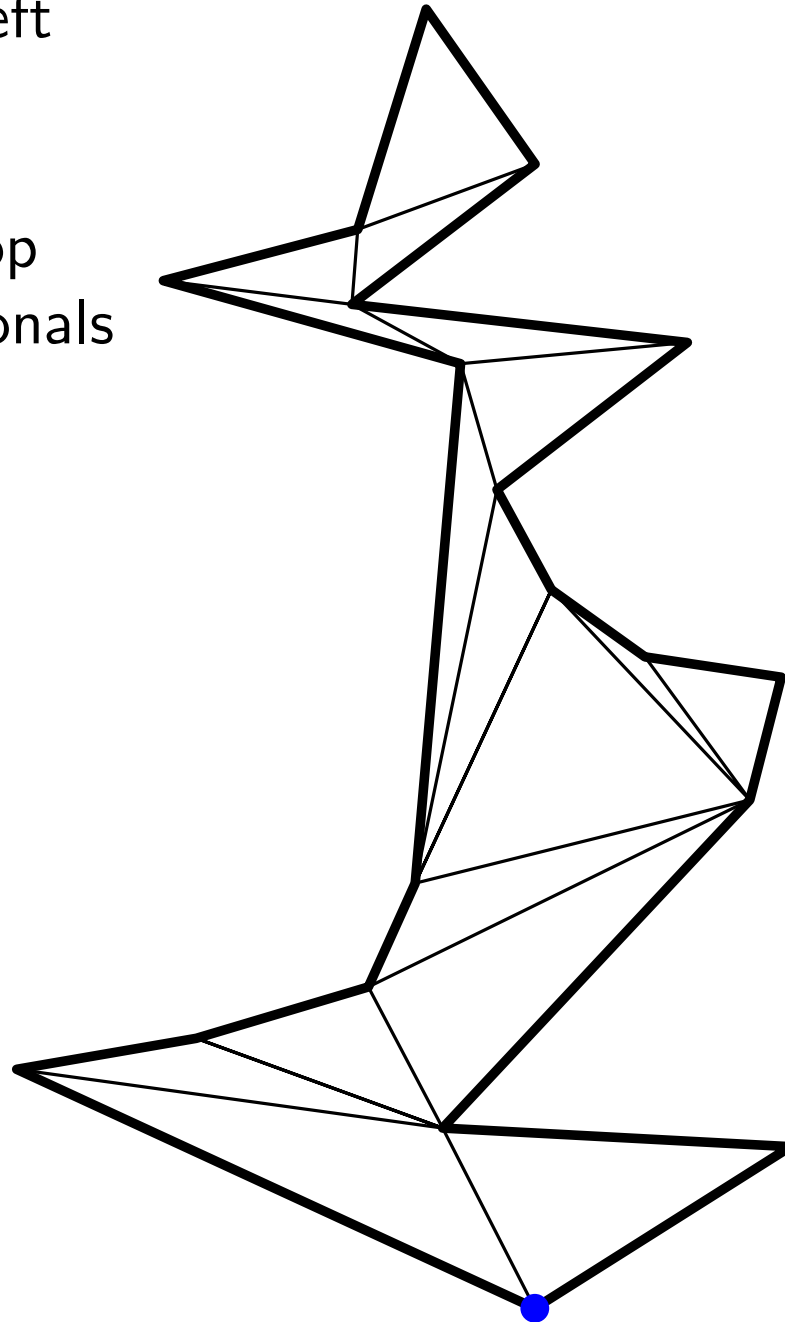
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above



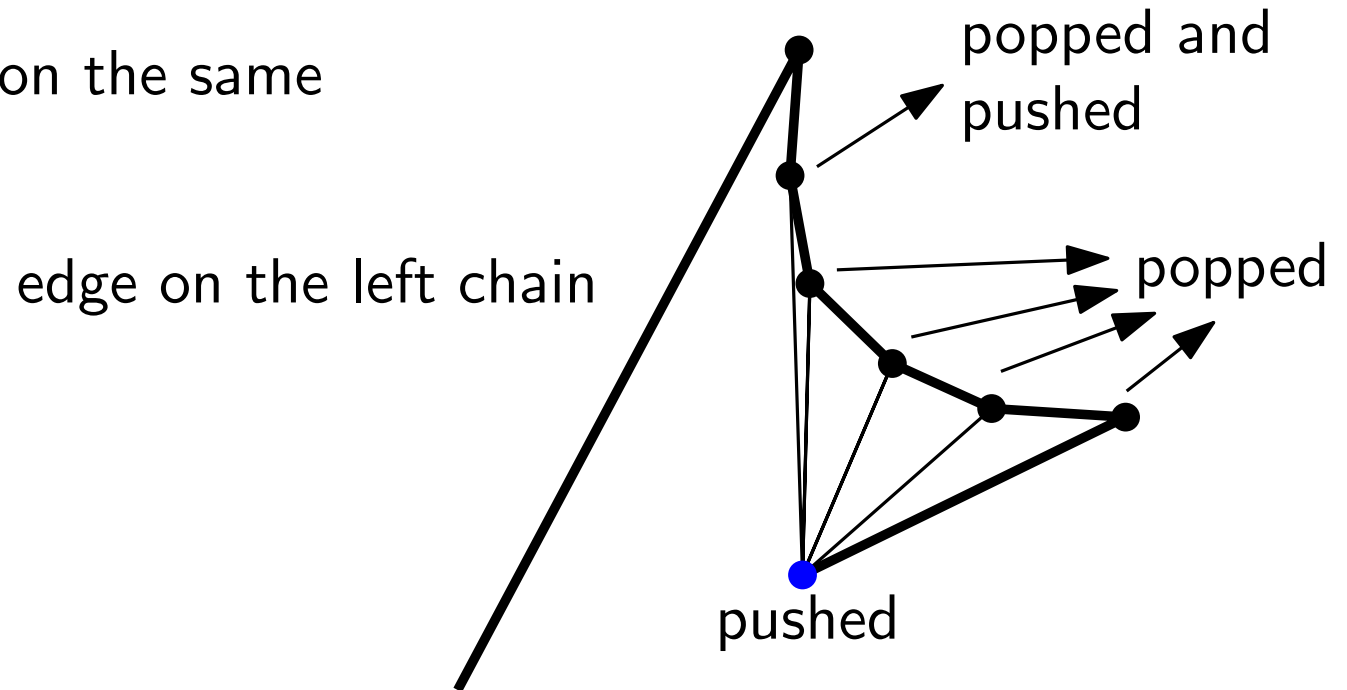
# Triangulating a $y$ -monotone polygon

- 1: Merge vertices of left and right chain to get sorted order
- 2: Traverse vertices top down and create diagonals to all possible vertices above

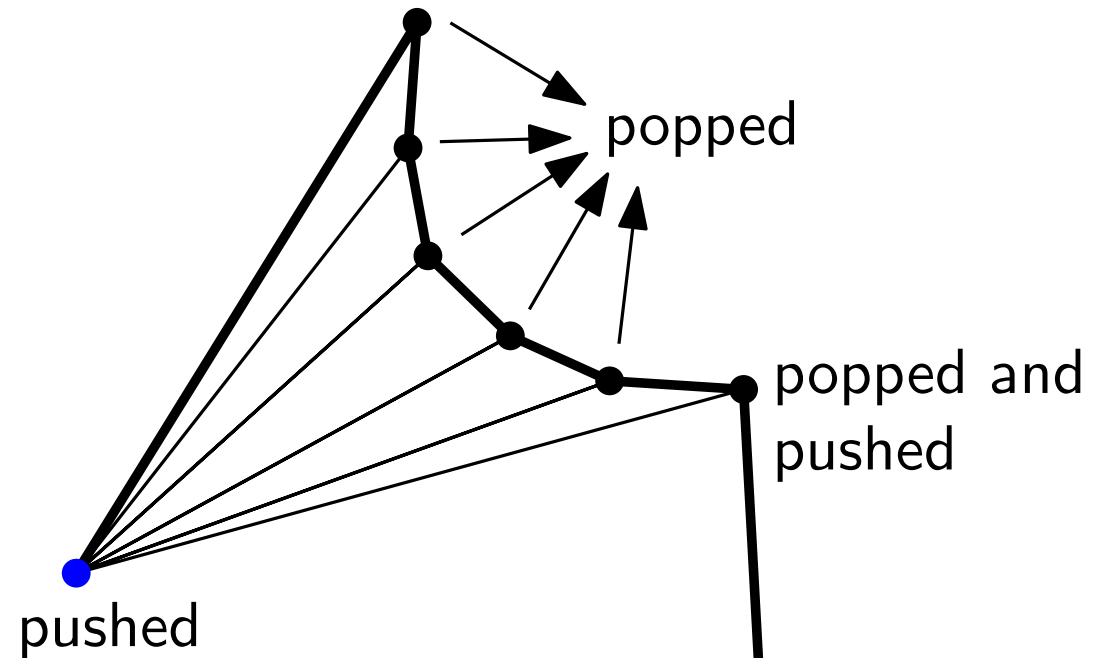


# Using a stack to make diagonals

**Case 1:** New vertex on the same chain (here right).

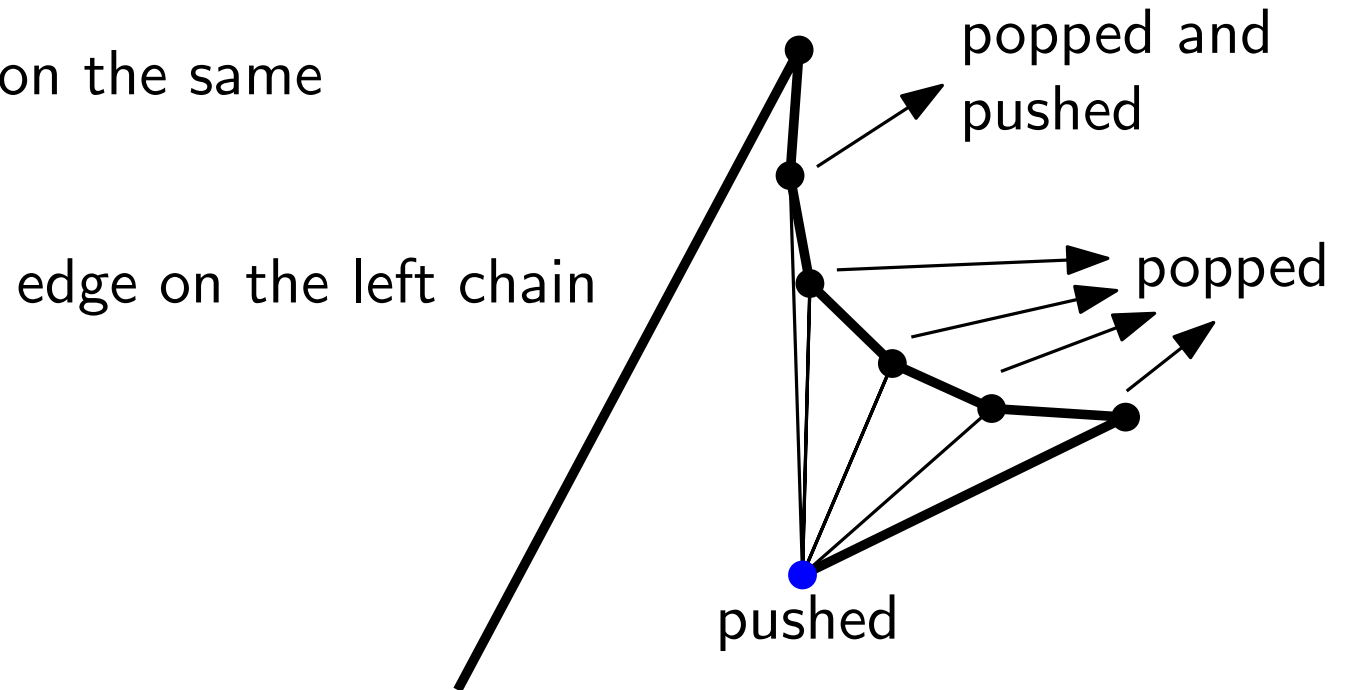


**Case 2:** New vertex on the other chain.



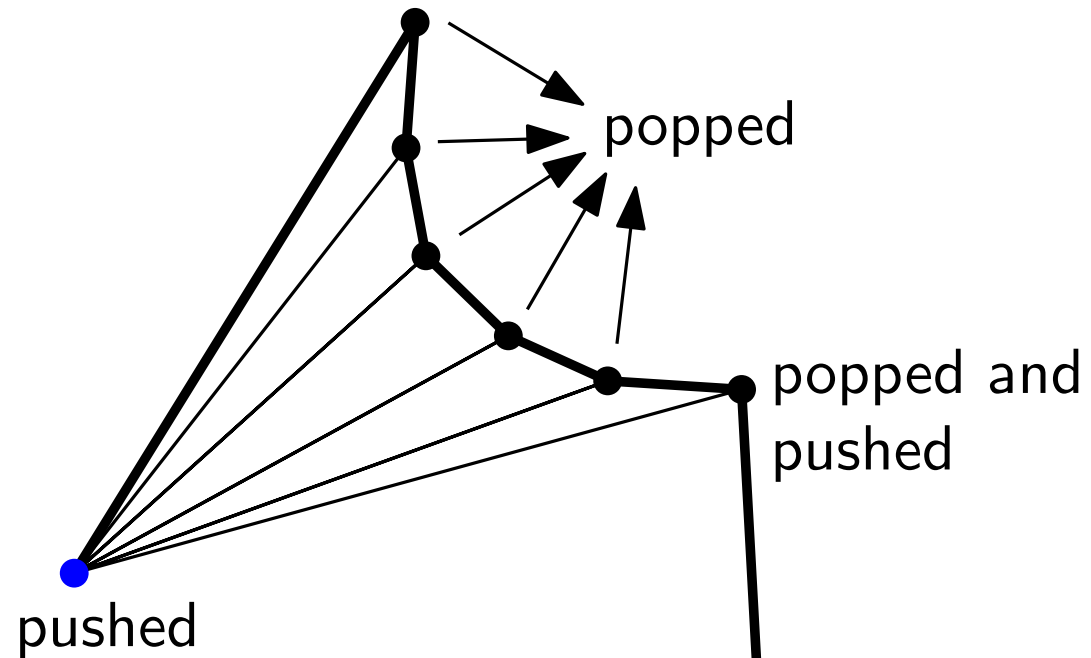
# Using a stack to make diagonals

**Case 1:** New vertex on the same chain (here right).



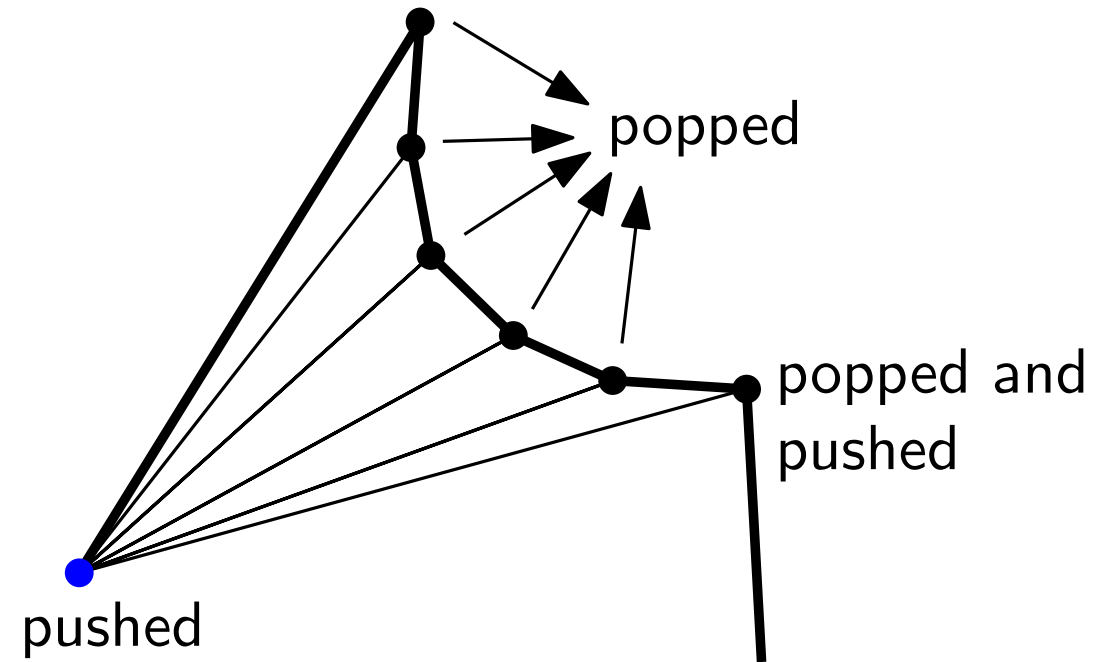
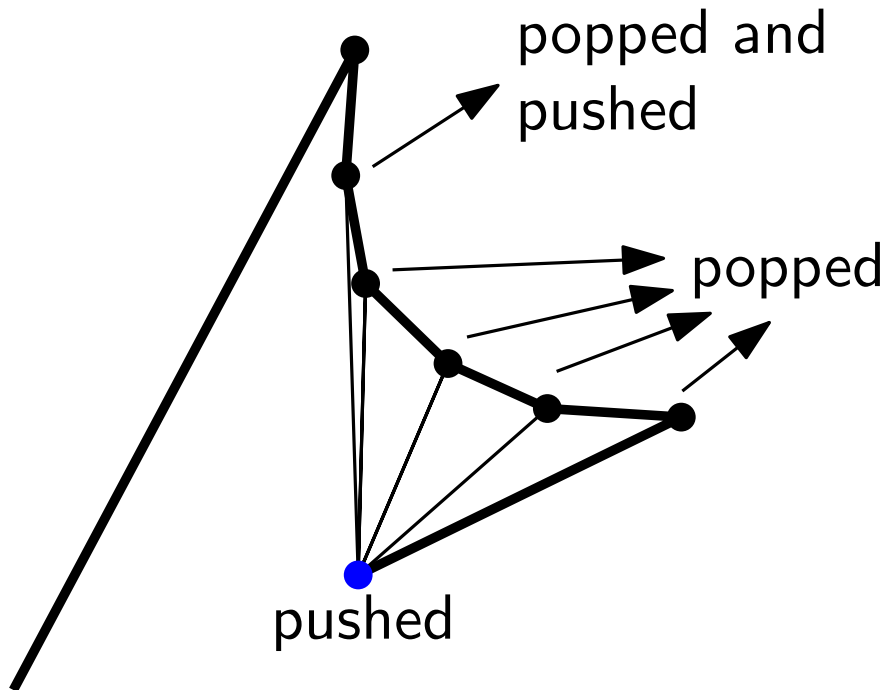
**Case 2:** New vertex on the other chain.

Why are all diagonals OK?





## Time complexity

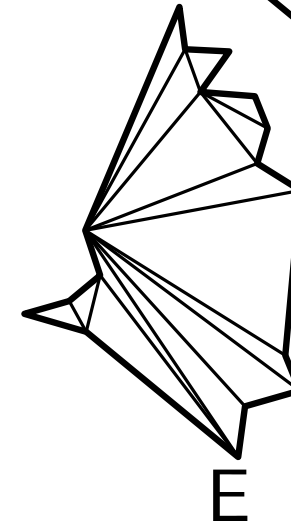
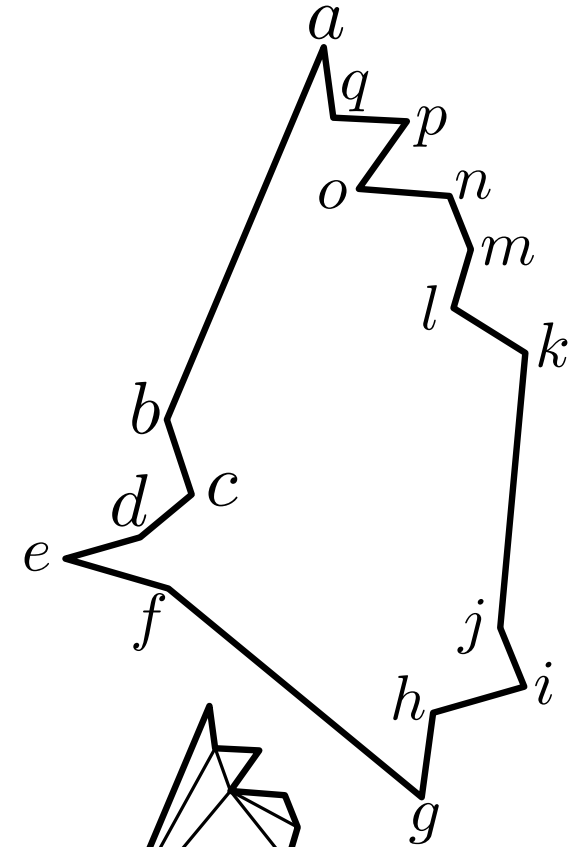
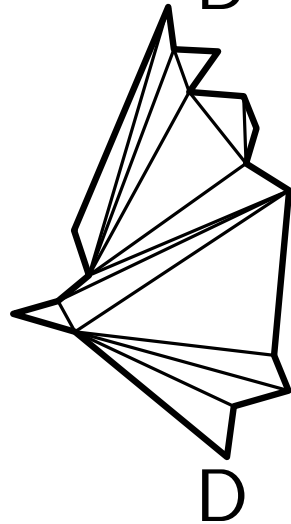
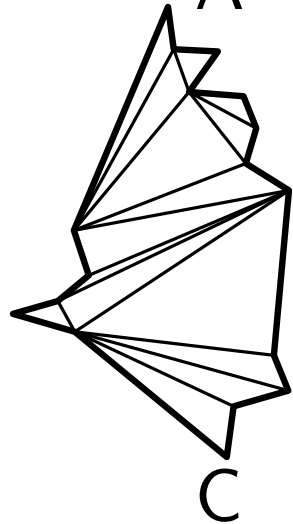
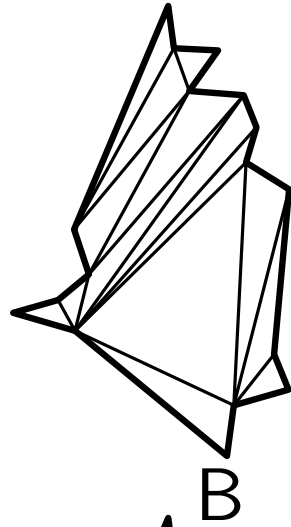
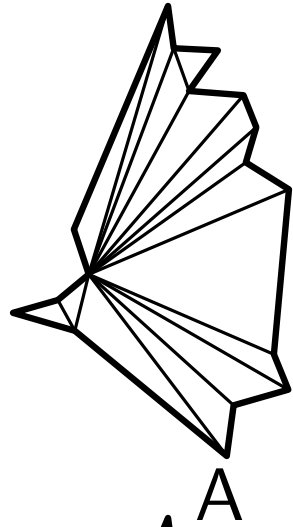


**Note:** In either case, we push two vertices  $\Rightarrow \leq 2n$  pushes, pops and diagonal checks.

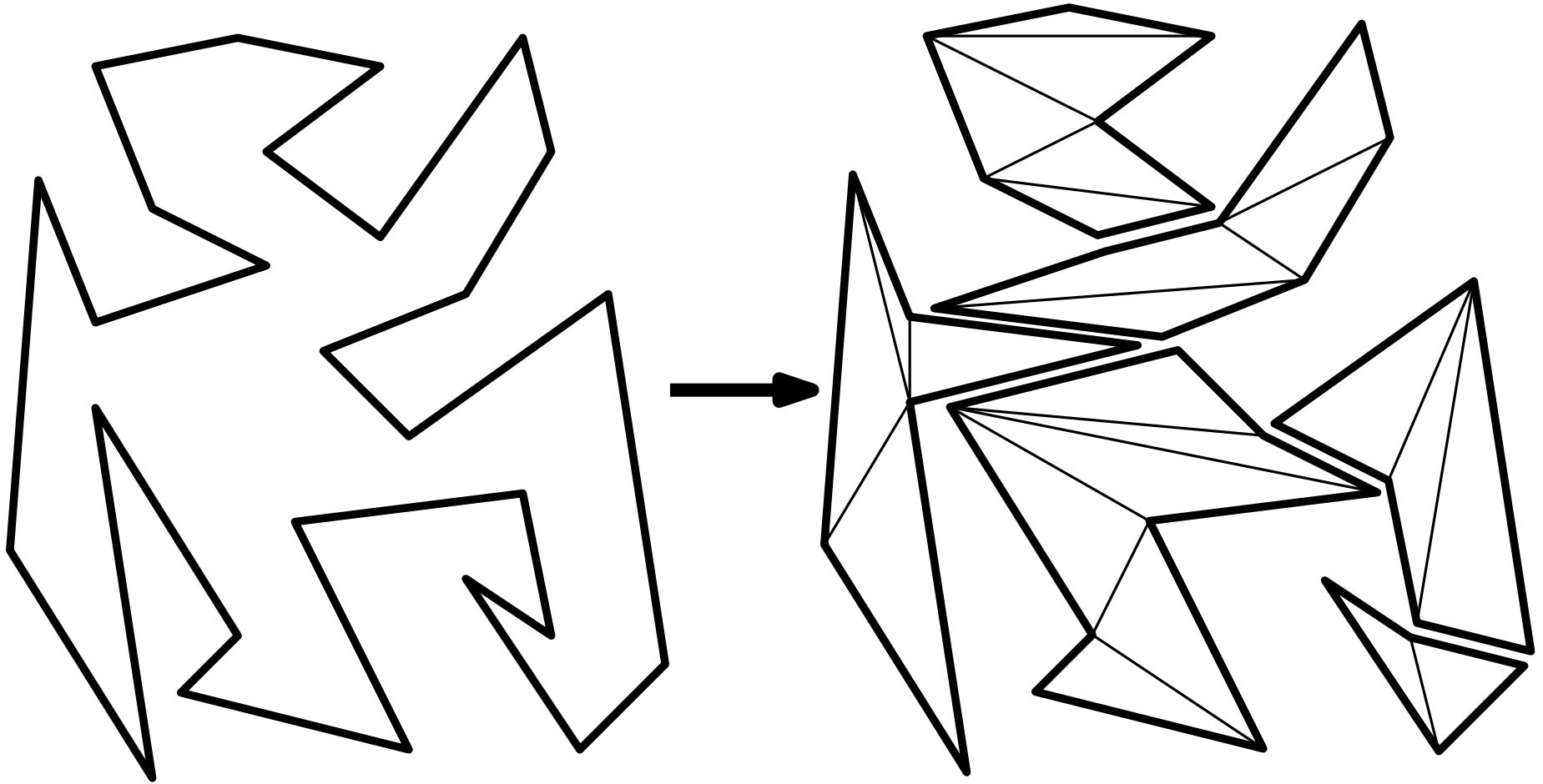
**Time complexity:**  $O(n)$  for merging and traversing.  
 $n$ : number of vertices of this  $y$ -monotone polygon

# How does the triangulation look?

socrative.com → Student login,  
Room name: ABRAHAMSEN3464



## Total time complexity



$\leq 3n - 6$  vertices  $\Rightarrow O(n)$  time to  
triangulate monotone polygons.  
In total  $O(n \log n)$  time  
(monotone partition dominates).

# History of Triangulation Algorithms

1979: Garey, Johnson, Preparata, Tarjan.  $O(n \log n)$  sweep-line algorithm, similar to this.

1982: Chazelle.  $O(n \log n)$  divide-and-conquer algorithm.

1986: Tarjan and Van Wyk.  $O(n \log \log n)$  algorithm.

1988: Clarkson, Tarjan, and Van Wyk. Randomized  $O(n \log^* n)$  algorithm. Two other algorithms with same running time around the same time.

1990: Chazelle. Optimal  $O(n)$  algorithm.

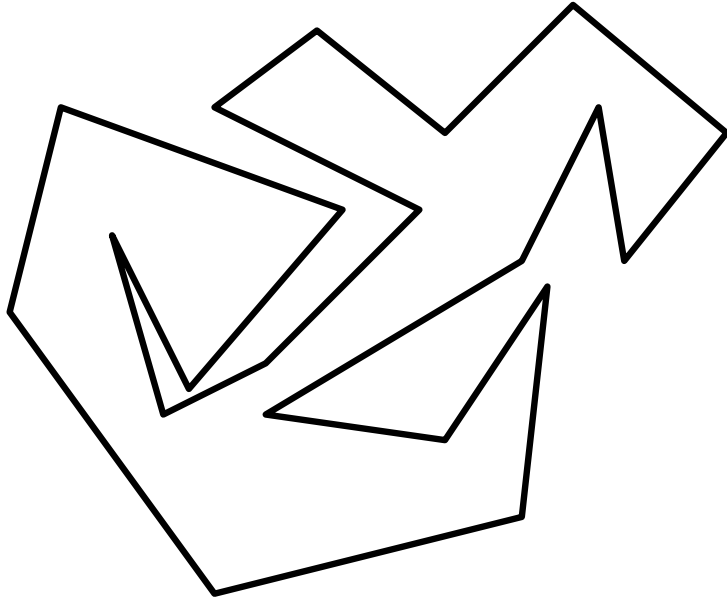
$\log^* n$ : number of times to apply  $\log$  before we get to  $\leq 1$ .

$$\log^* n = \begin{cases} 0, & \text{if } n \leq 1, \\ 1 + \log^*(\log n), & \text{if } n > 1. \end{cases}$$

$n$	$\log^* n$
$(-\infty, 1]$	0
$(1, 2]$	1
$(2, 4]$	2
$(4, 16]$	3
$(16, 65536]$	4
$(65536, 2^{65536}]$	5

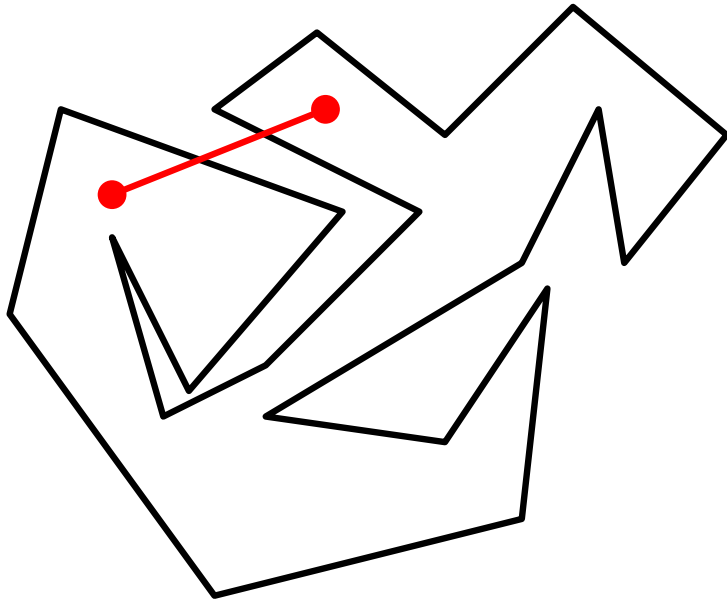
# Use of Triangulations

Visibility problems.



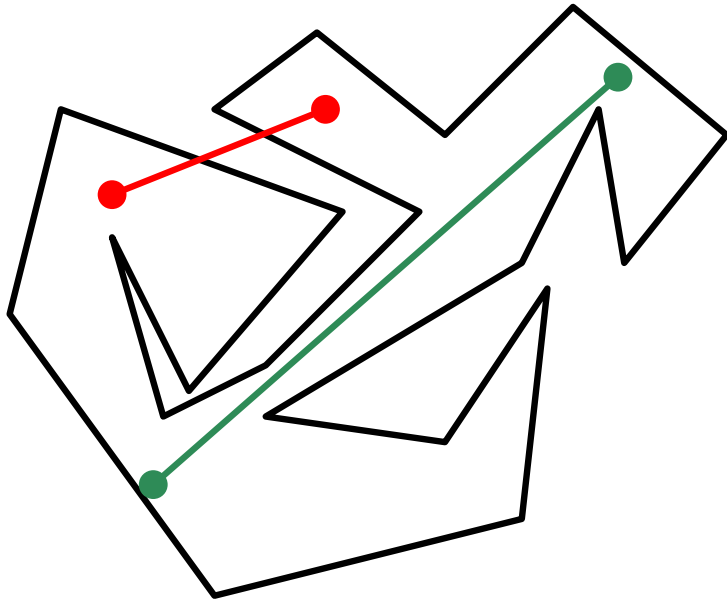
# Use of Triangulations

Visibility problems.



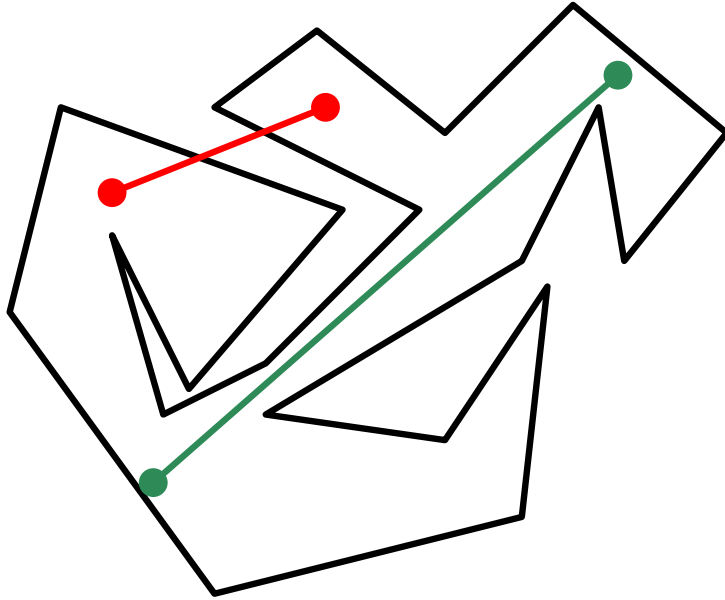
# Use of Triangulations

Visibility problems.

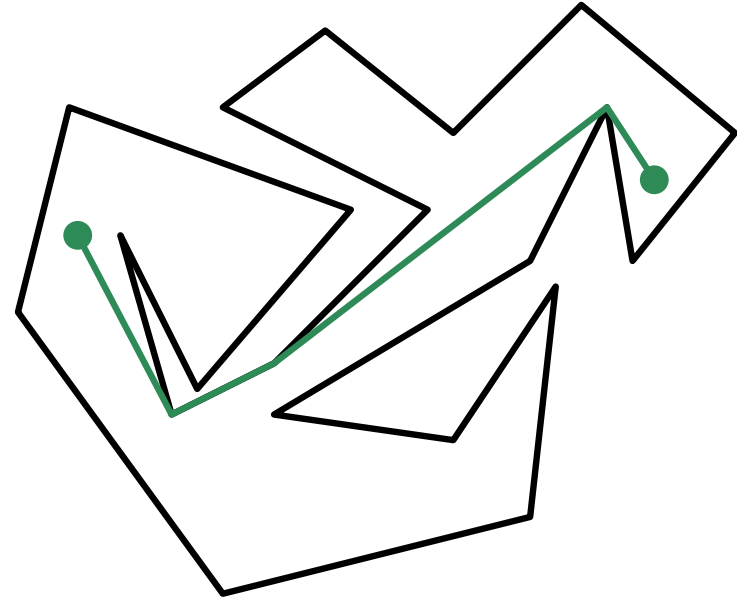


# Use of Triangulations

Visibility problems.



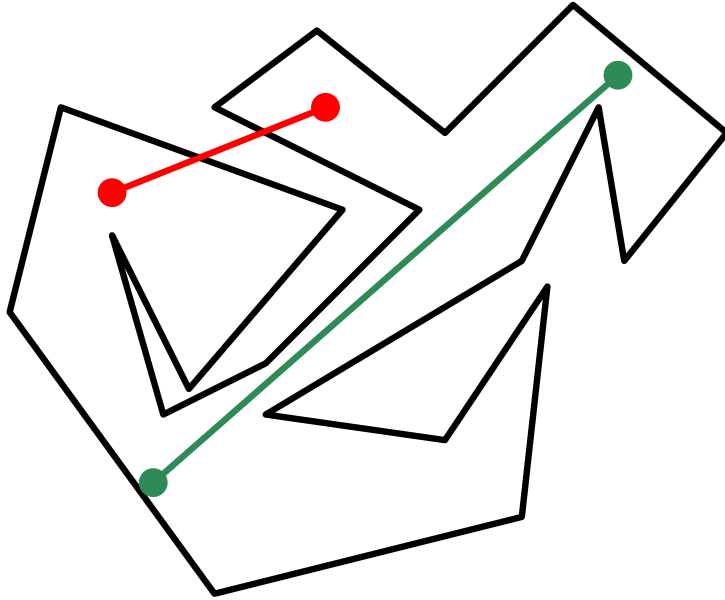
Shortest paths.



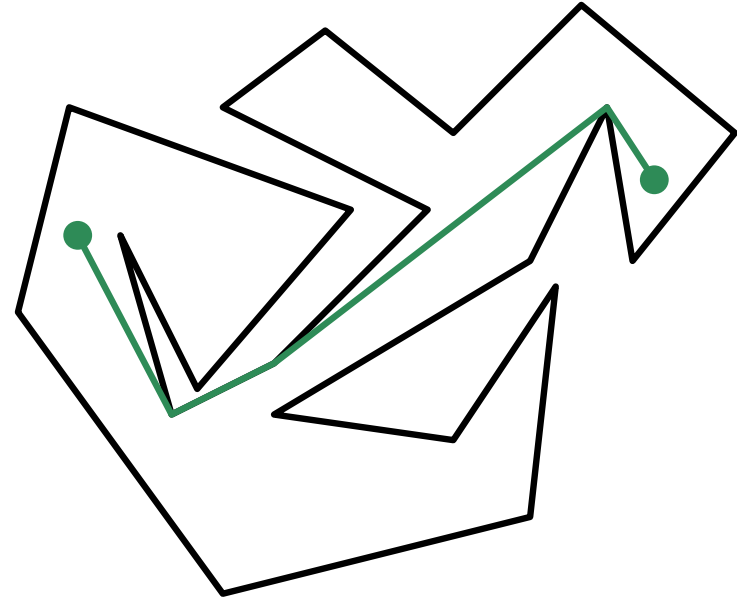


# Use of Triangulations

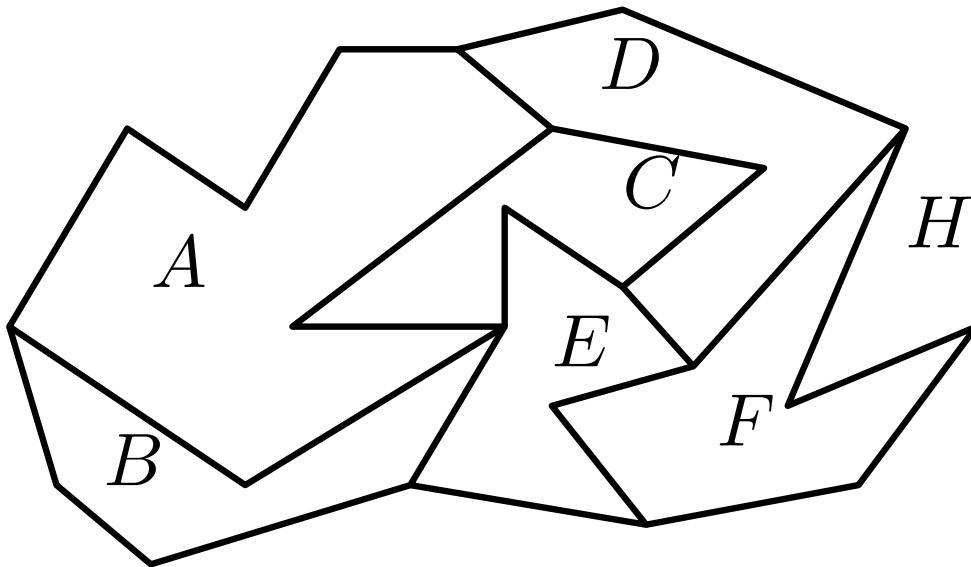
Visibility problems.



Shortest paths.

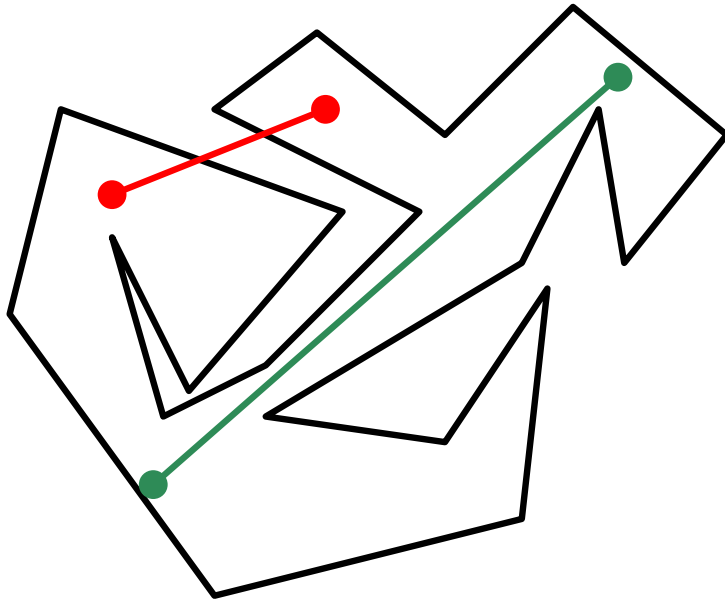


Point location.

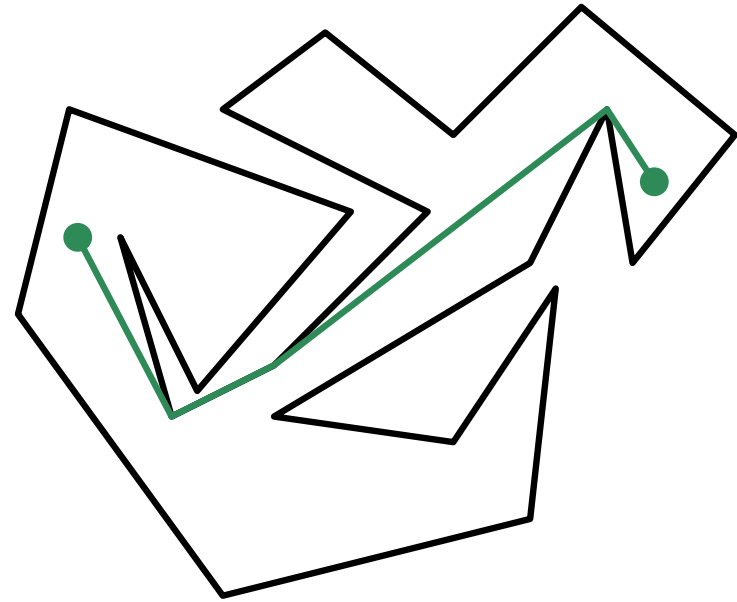


# Use of Triangulations

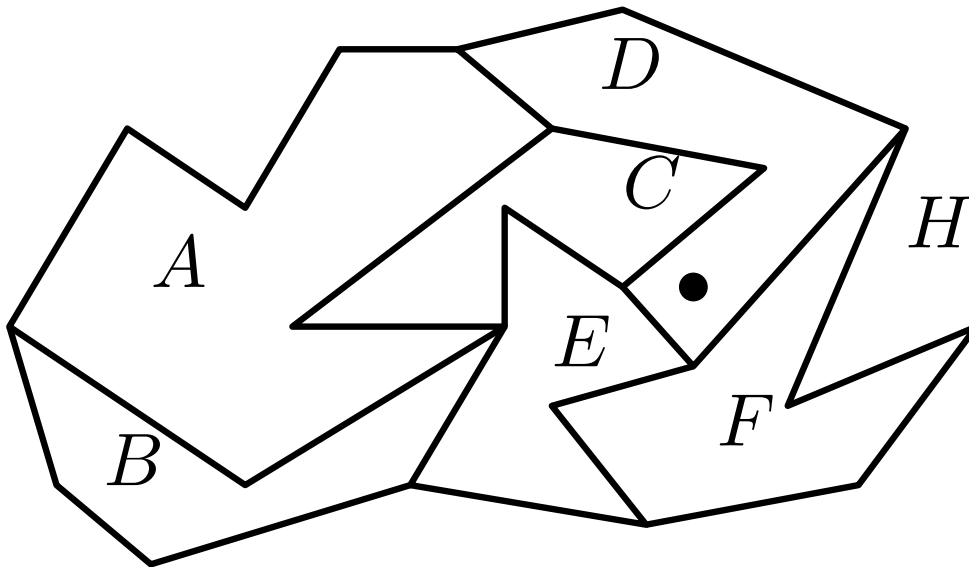
Visibility problems.



Shortest paths.

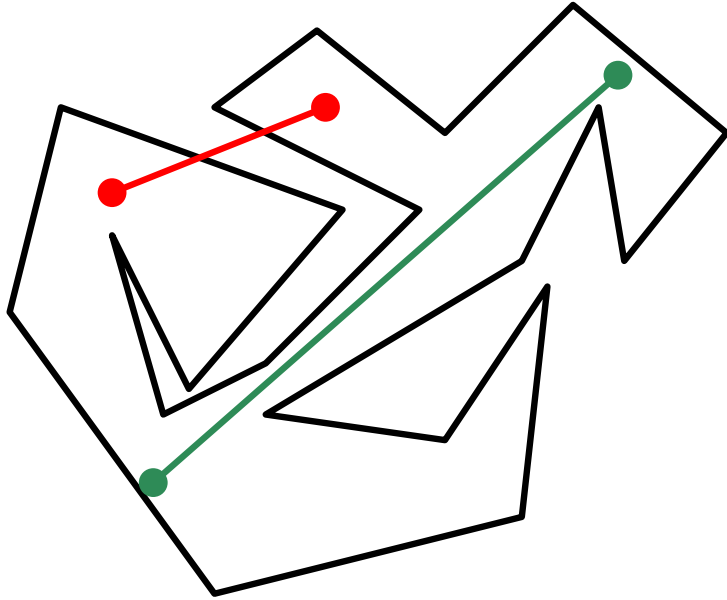


Point location.

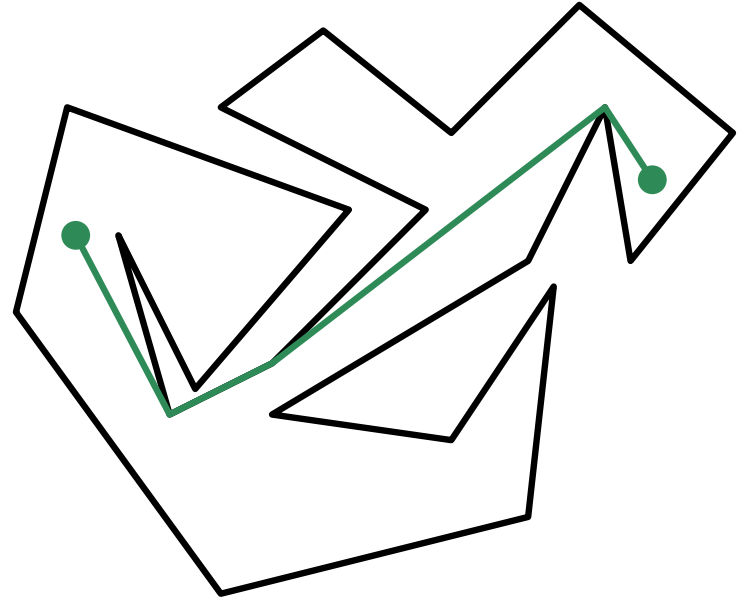


# Use of Triangulations

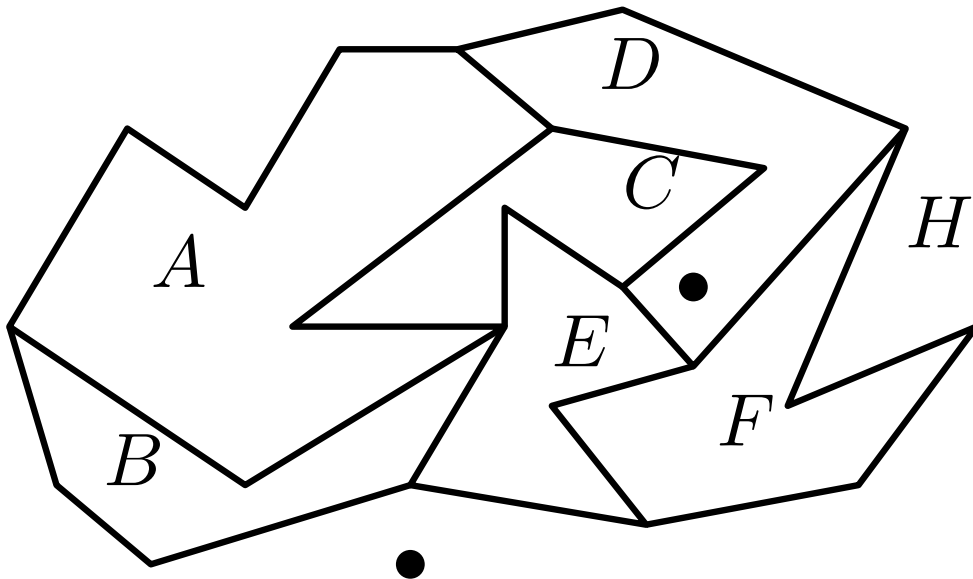
Visibility problems.



Shortest paths.

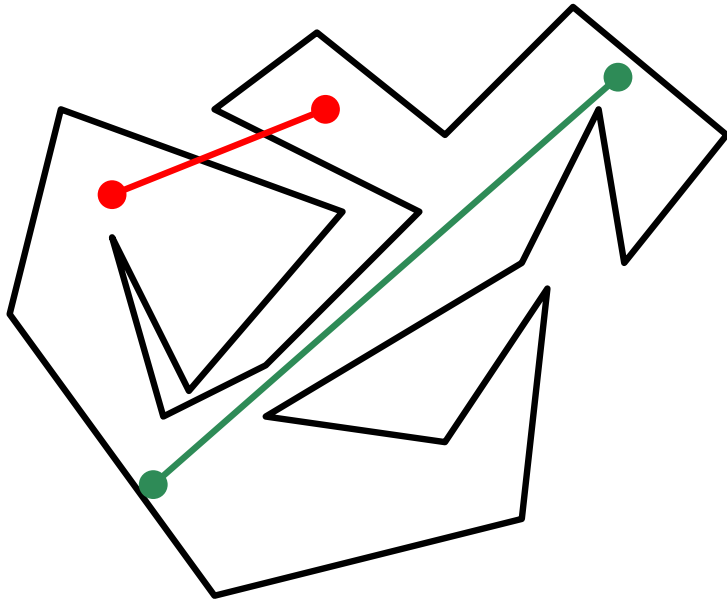


Point location.

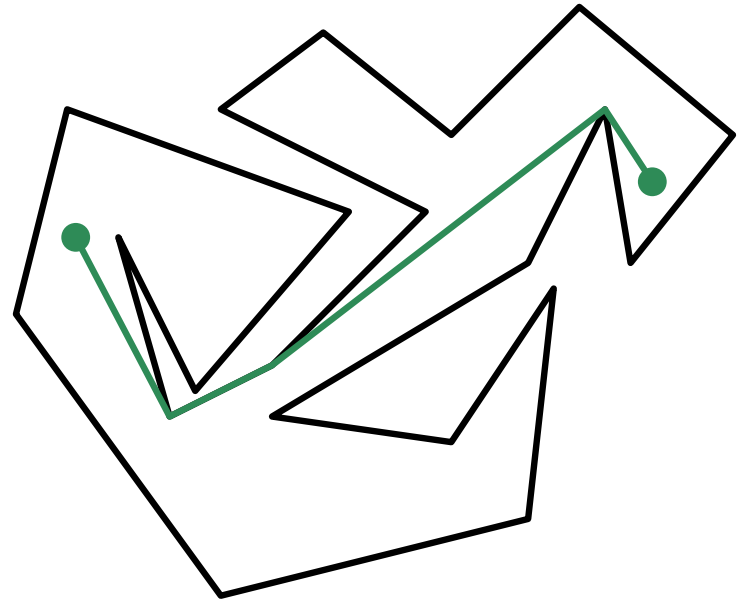


# Use of Triangulations

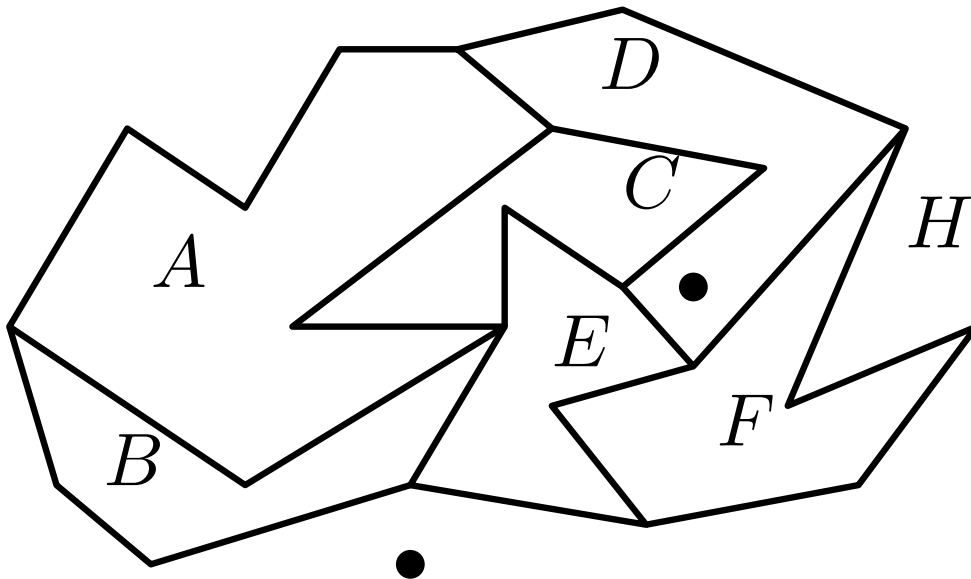
Visibility problems.



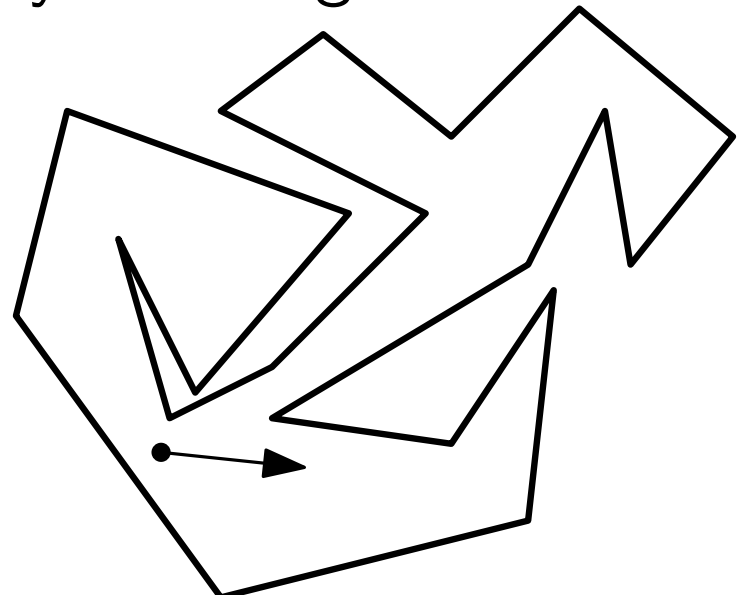
Shortest paths.



Point location.

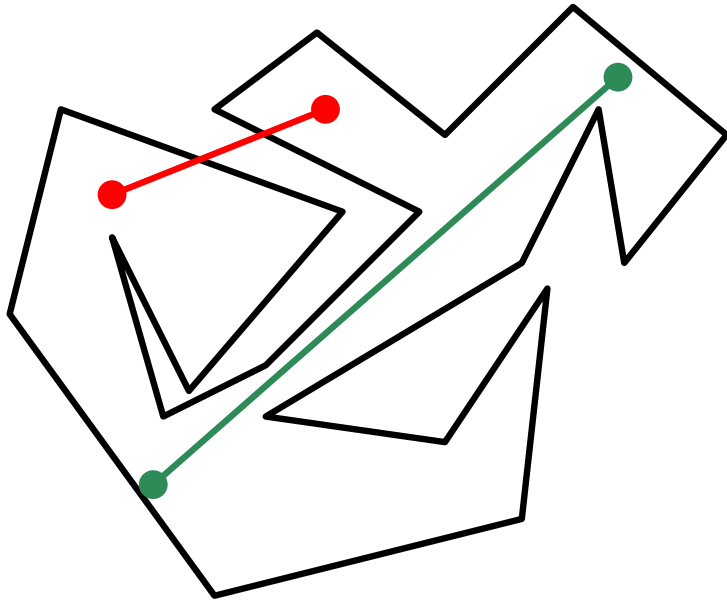


Ray shooting.

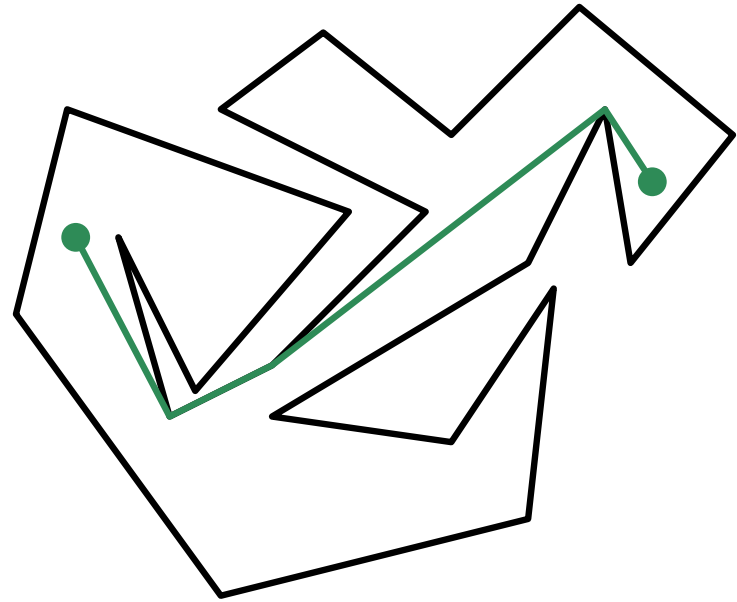


# Use of Triangulations

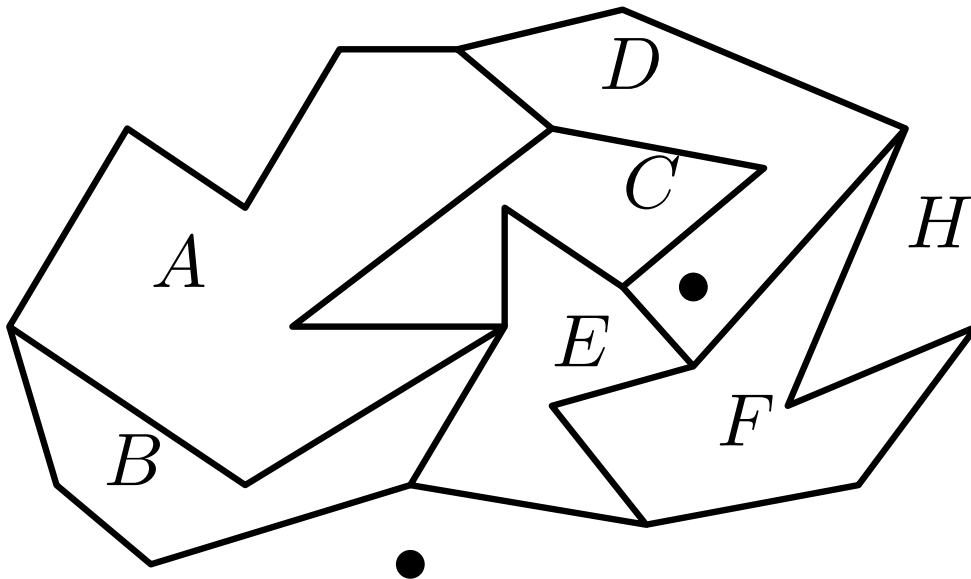
Visibility problems.



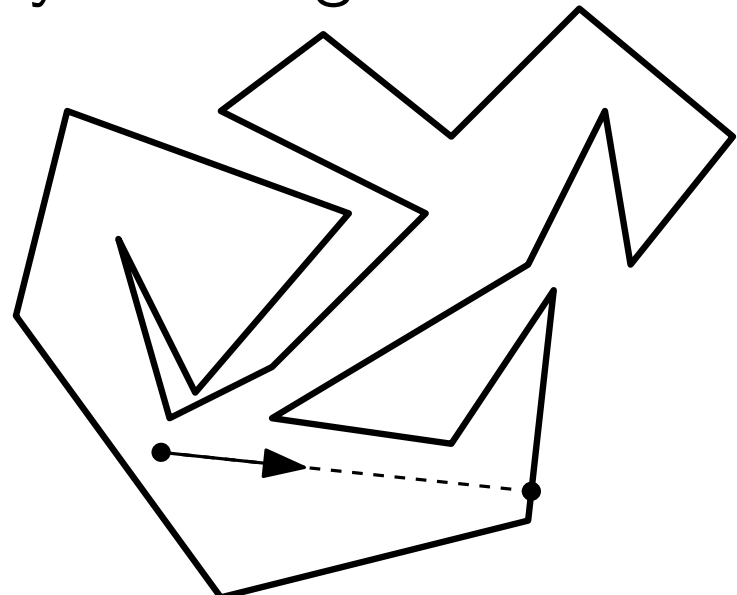
Shortest paths.



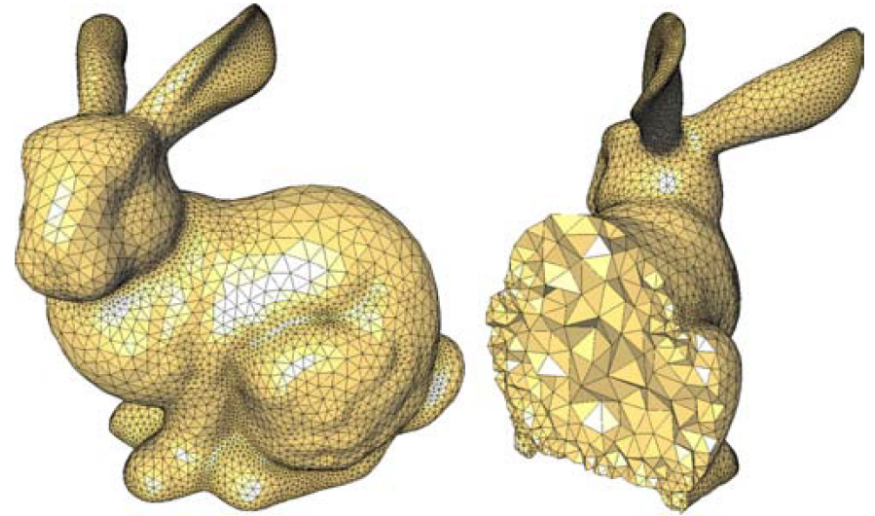
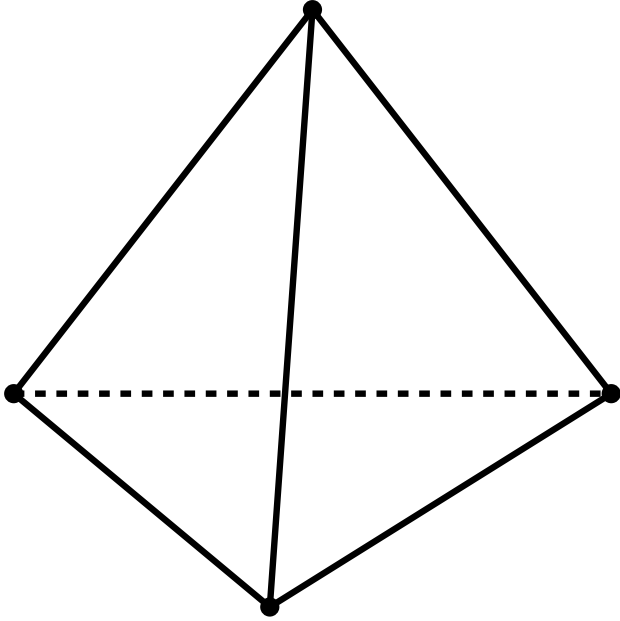
Point location.



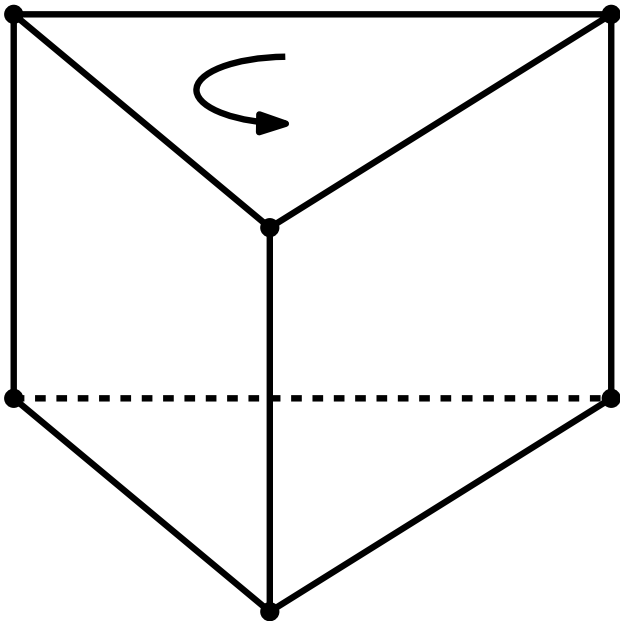
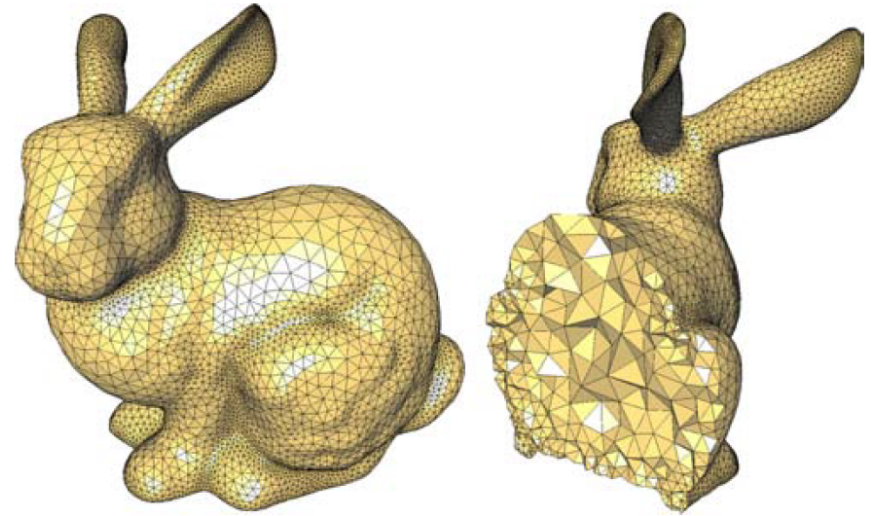
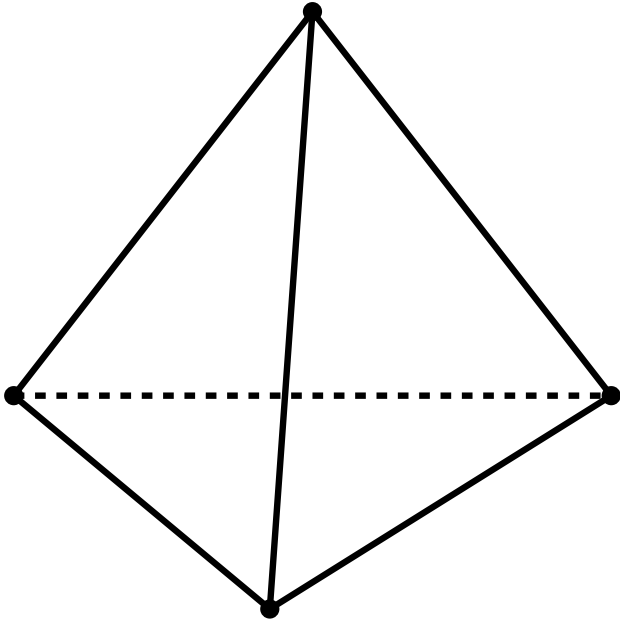
Ray shooting.



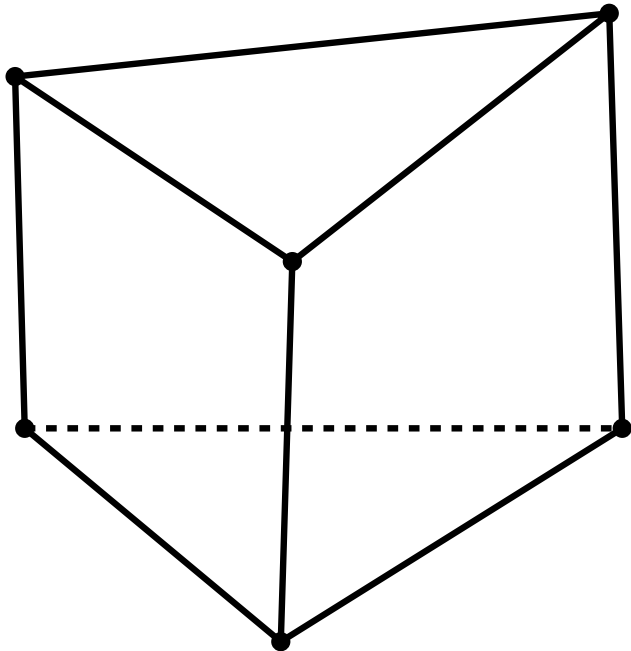
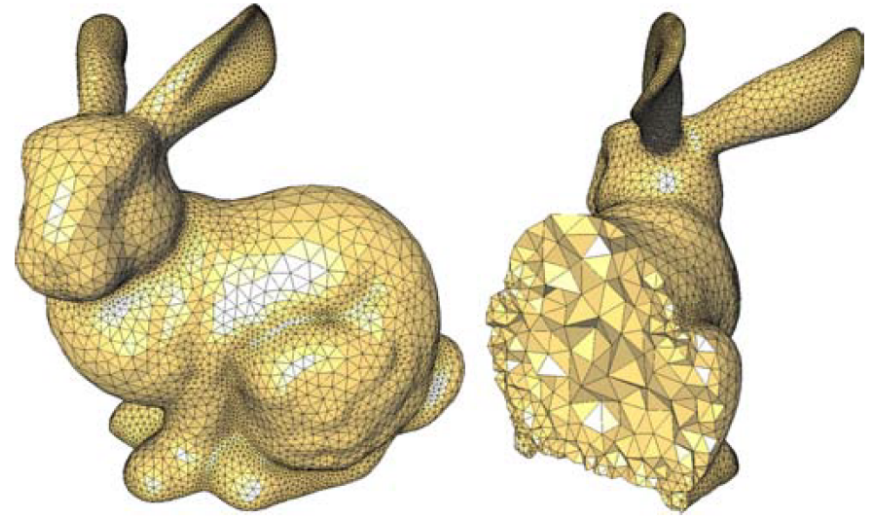
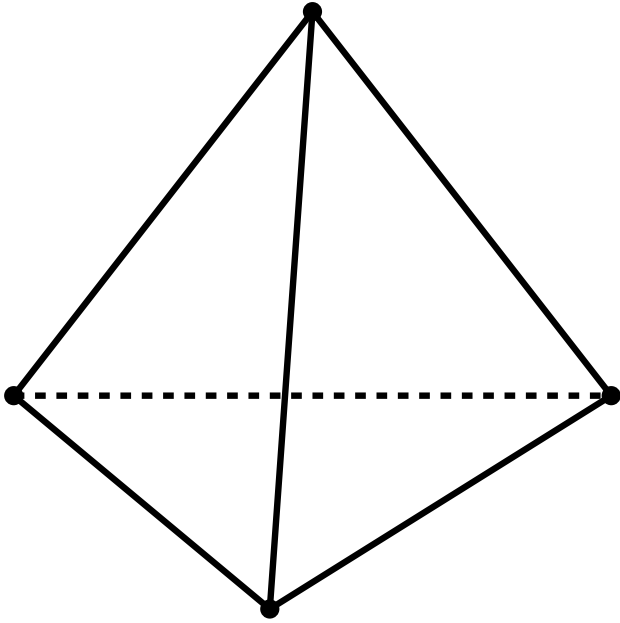
3D



3D

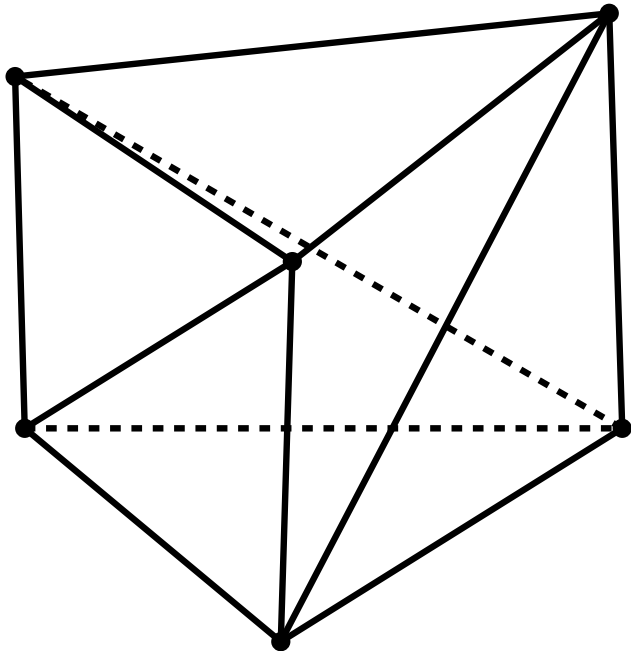
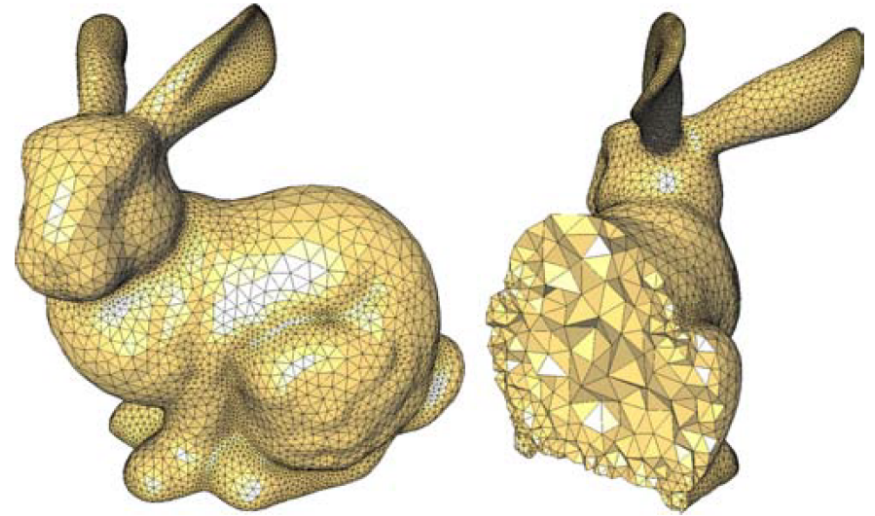
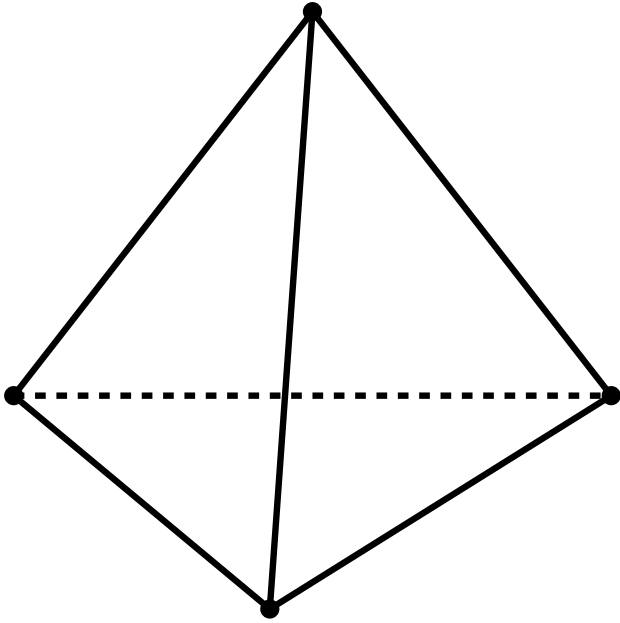


3D

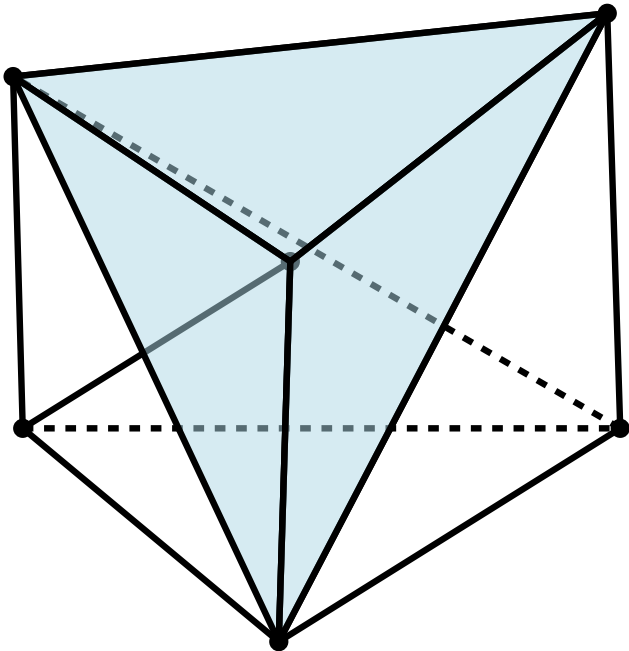
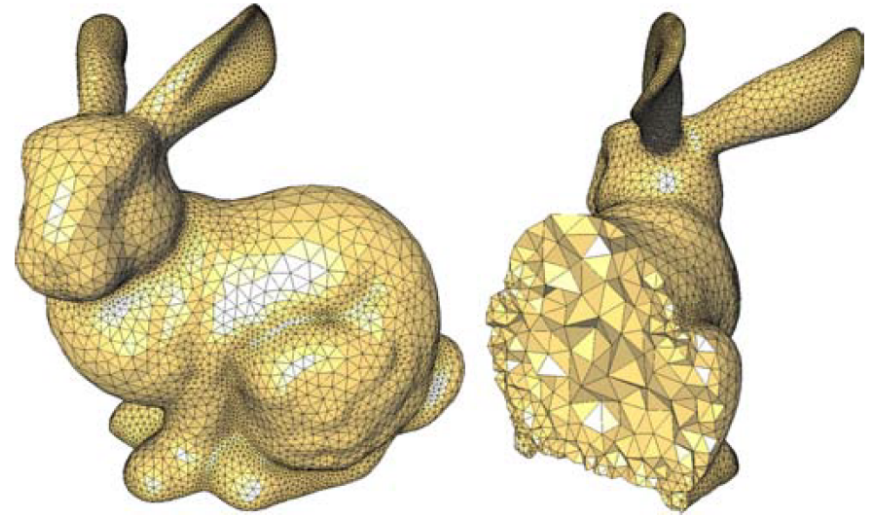
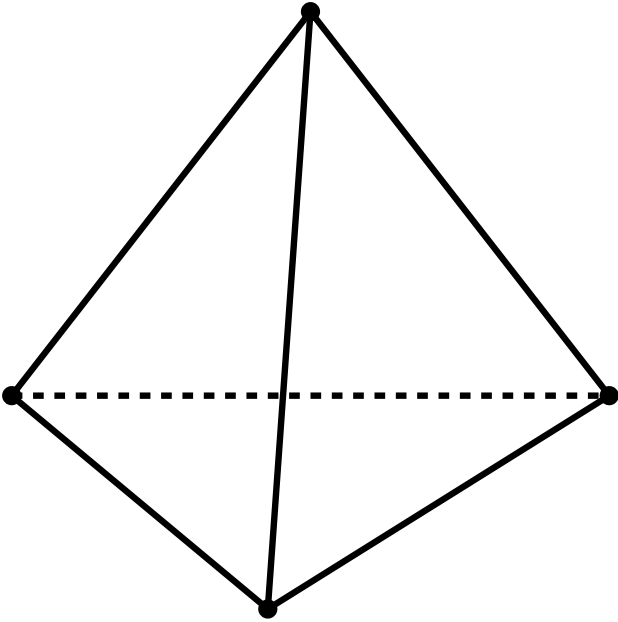




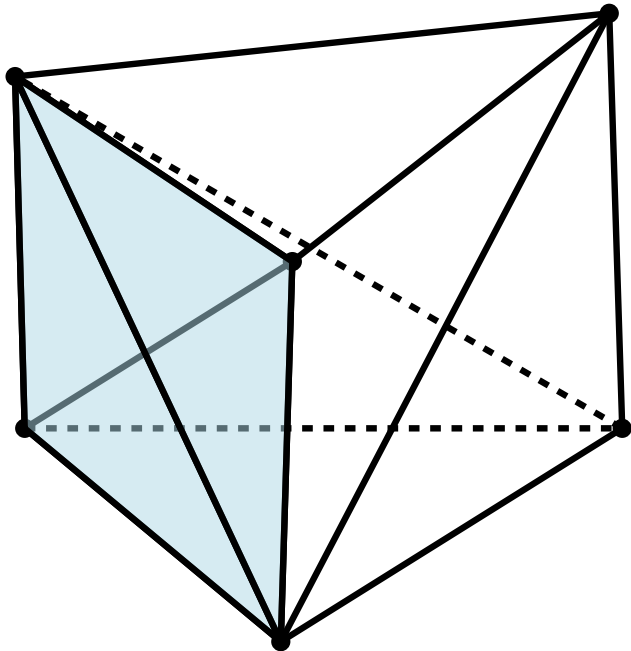
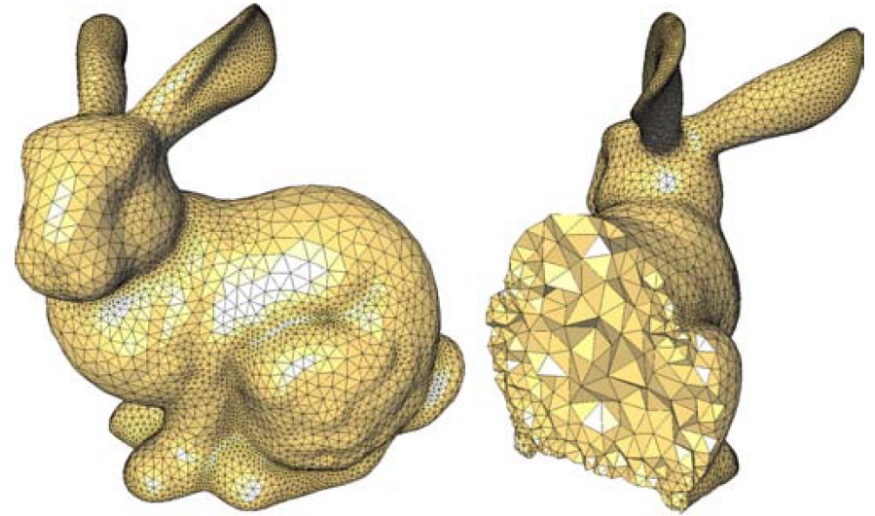
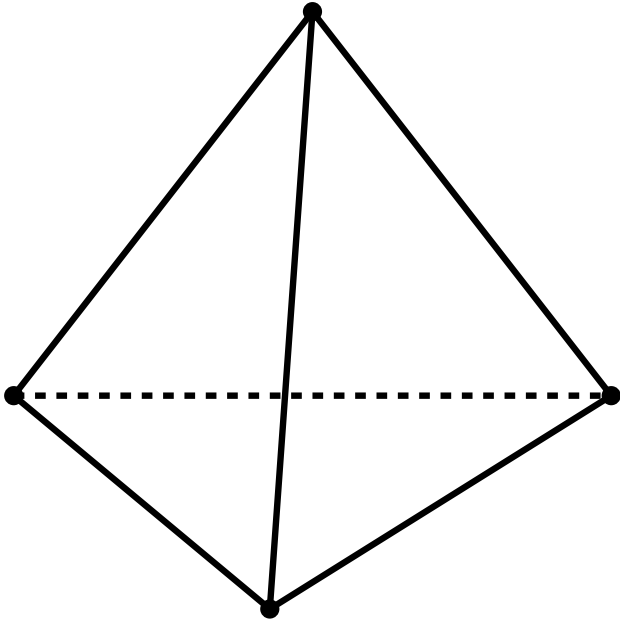
3D



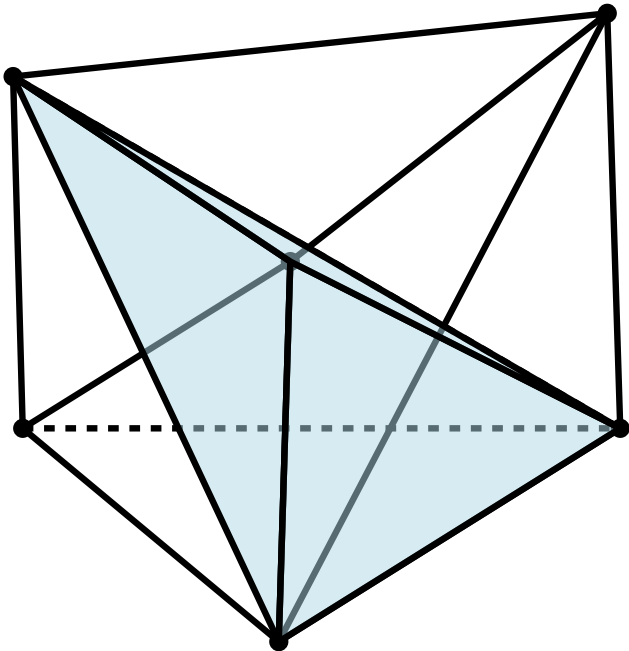
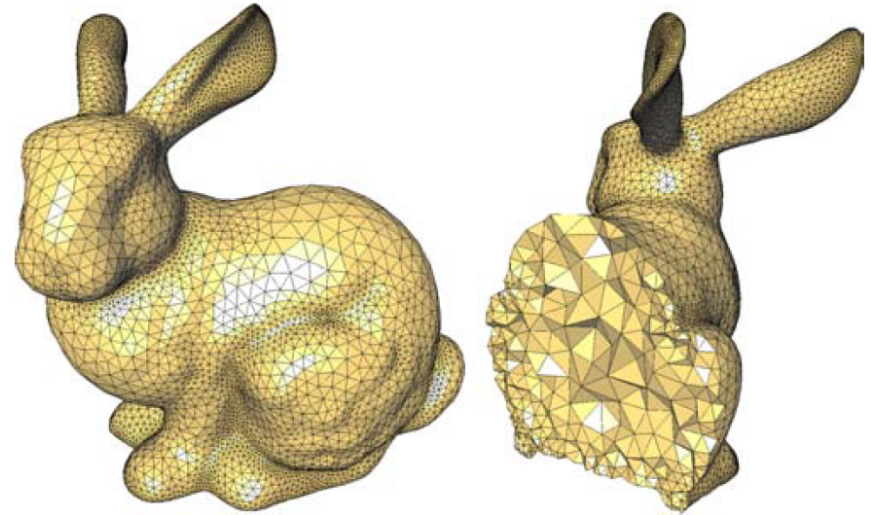
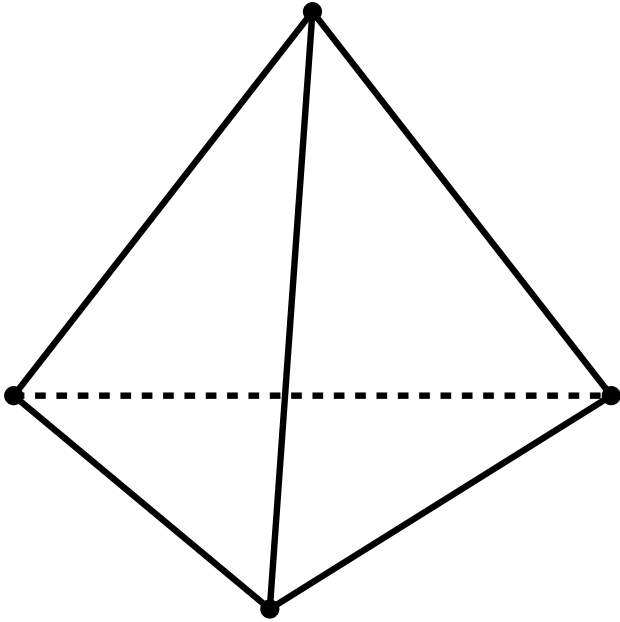
3D



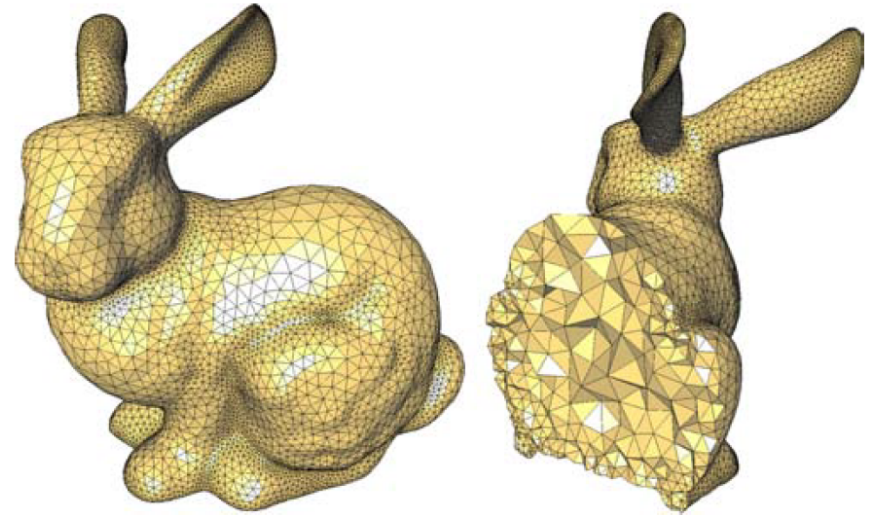
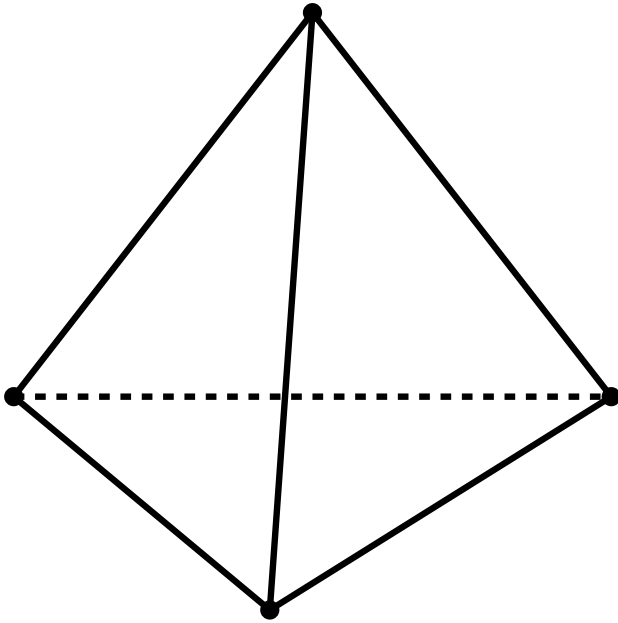
3D



3D



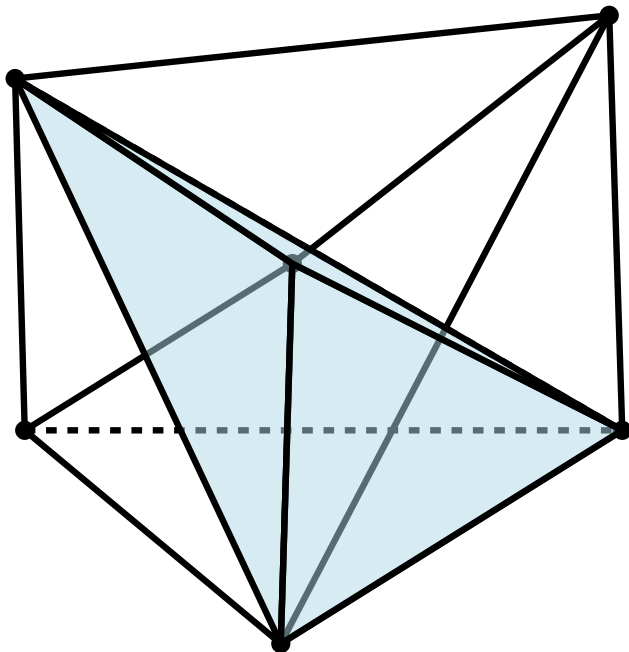
## 3D



Sometimes  $\Theta(n^2)$  Steiner points are needed. NP-complete to decide if possible without Steiner points.

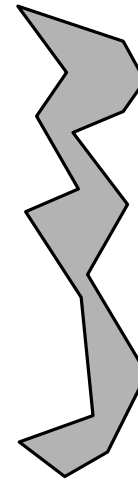
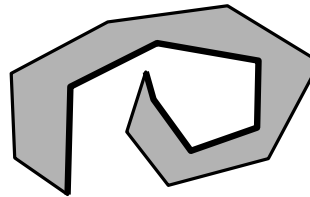
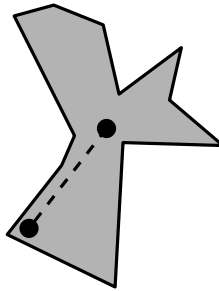
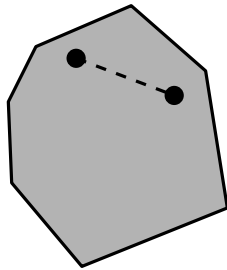
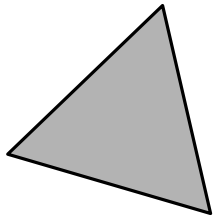
Even when possible without Steiner points, it differs how many tetrahedra are needed.

For some convex polyhedra with  $n$  corners, there exist different tetrahedralizations without Steiner points of sizes  $\Theta(n)$  and  $\Theta(n^2)$ .



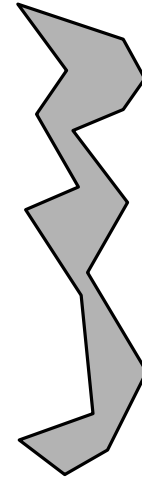
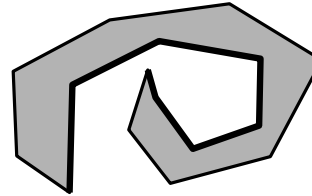
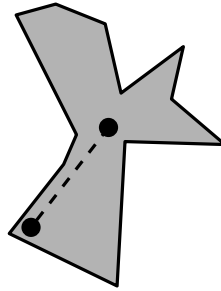
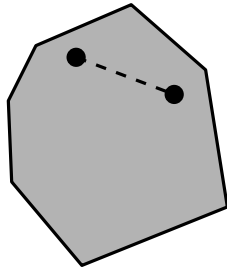
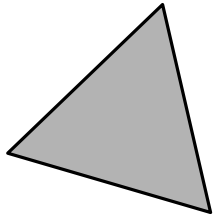
# Other decomposition problems

Type of pieces

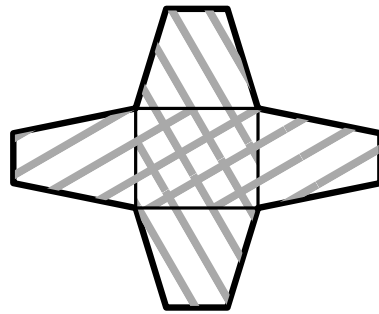
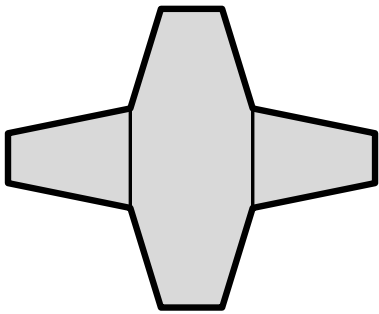


# Other decomposition problems

Type of pieces

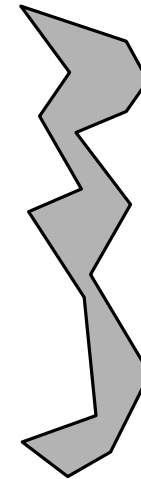
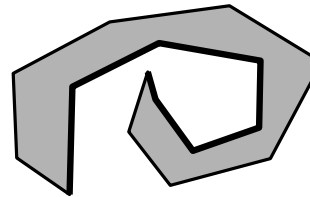
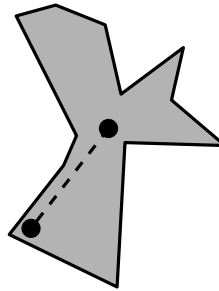
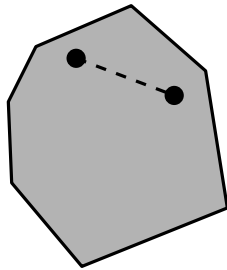
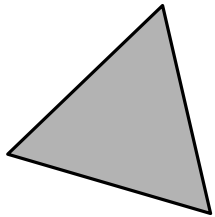


Partition vs. covering

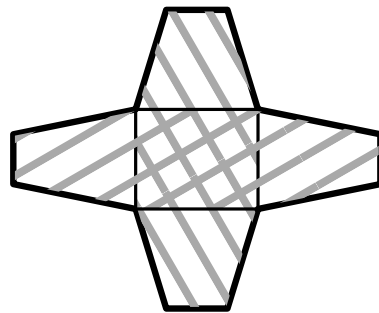
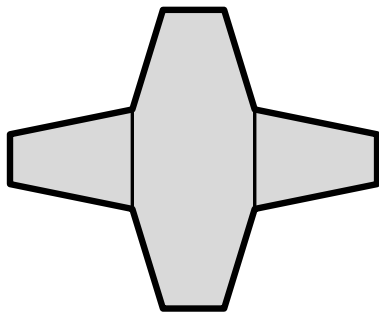


# Other decomposition problems

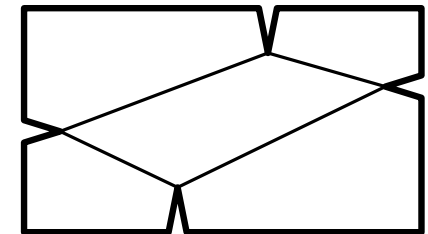
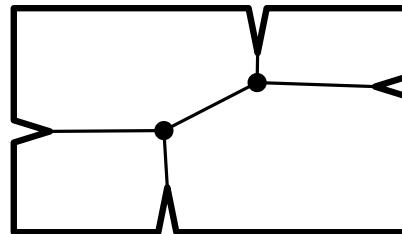
Type of pieces



Partition vs. covering



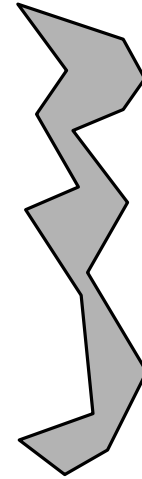
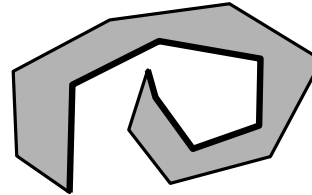
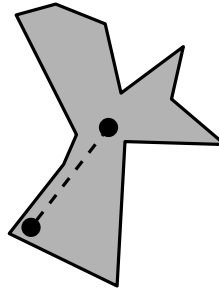
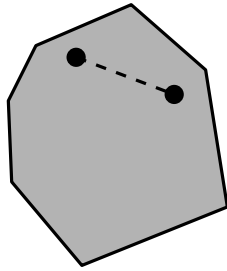
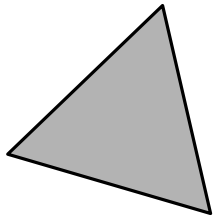
With or without Steiner points



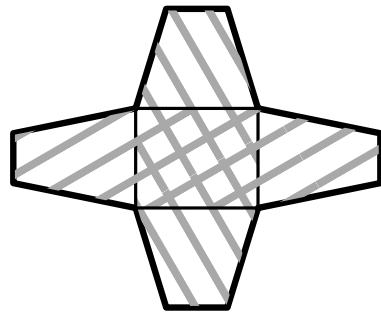
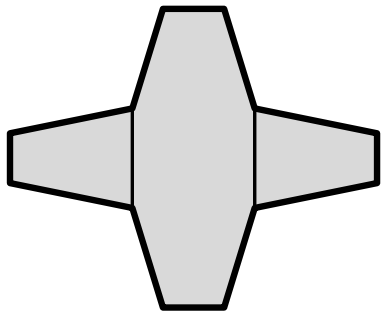


# Other decomposition problems

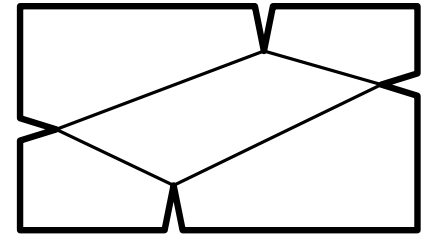
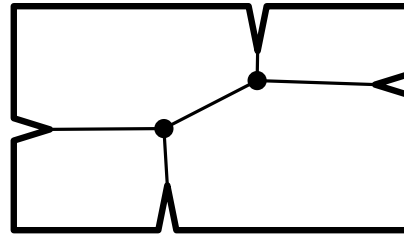
Type of pieces



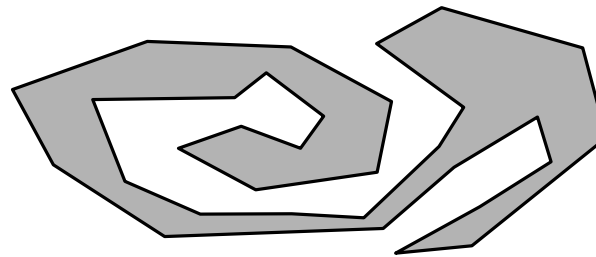
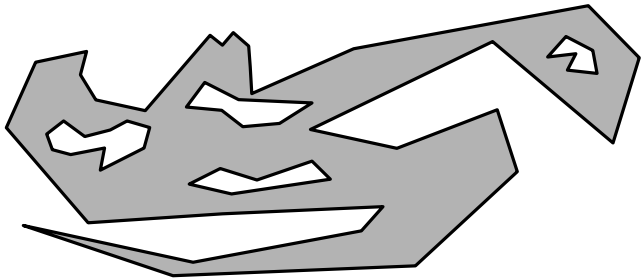
Partition vs. covering



With or without Steiner points

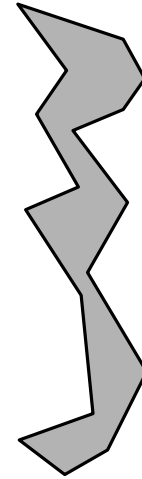
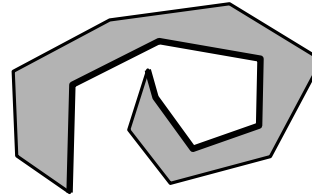
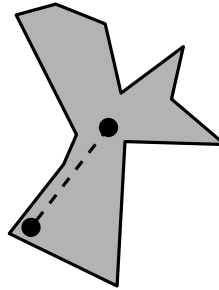
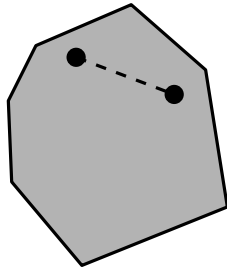
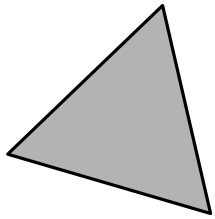


With or without holes

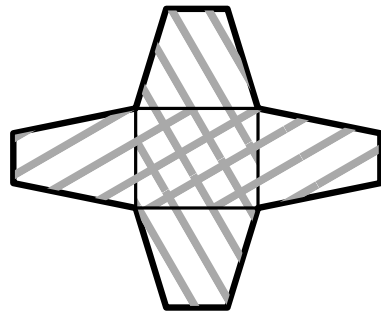
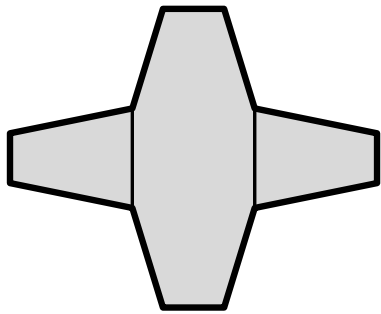


# Other decomposition problems

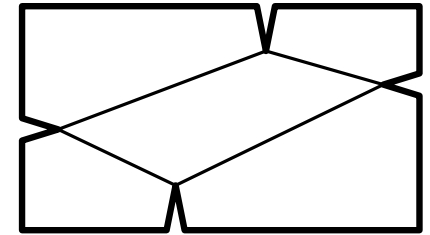
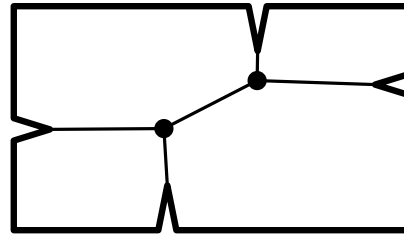
Type of pieces



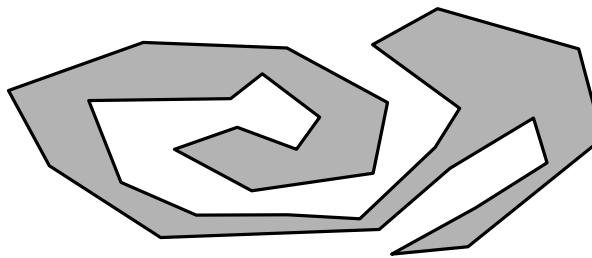
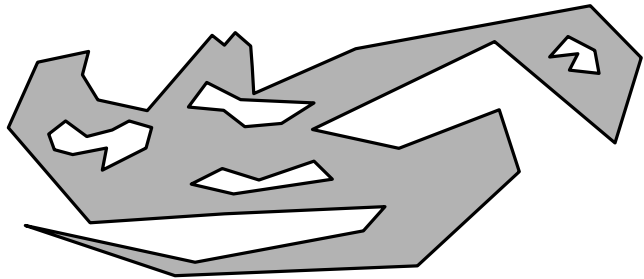
Partition vs. covering



With or without Steiner points



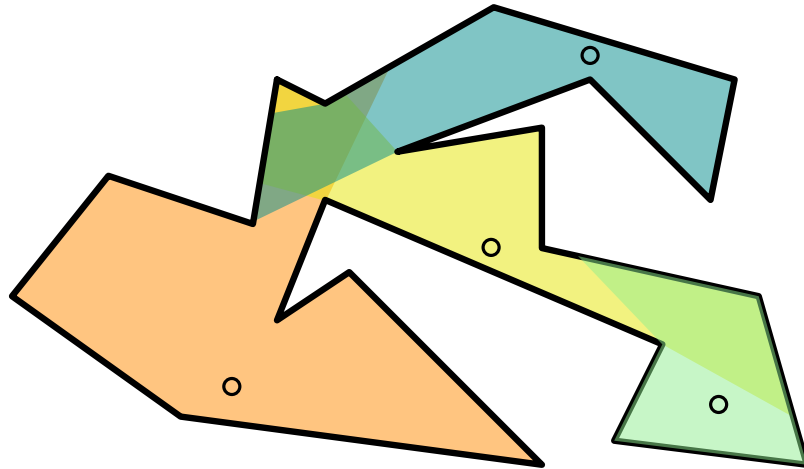
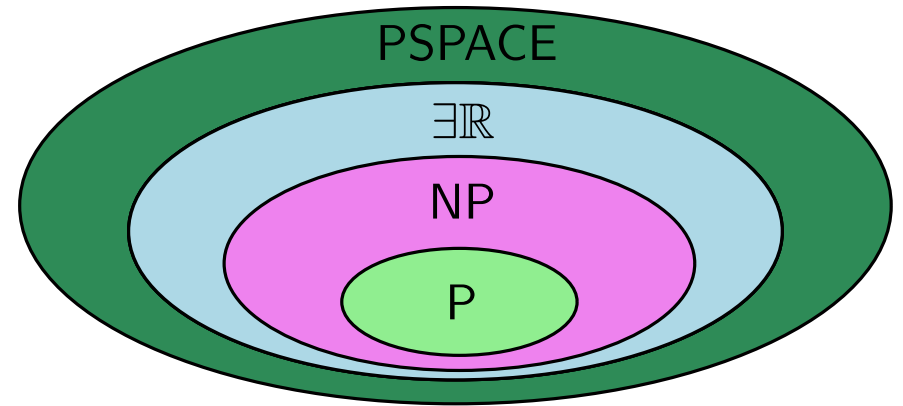
With or without holes



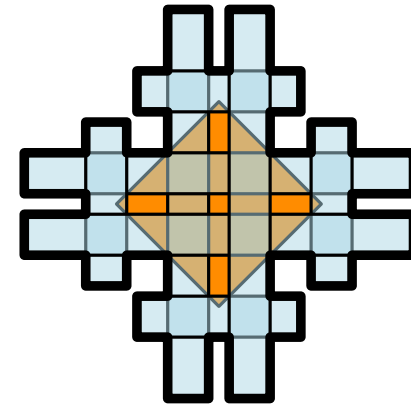
$5 \cdot 2 \cdot 2 \cdot 2 = 40$  problems!

# Some of my own work

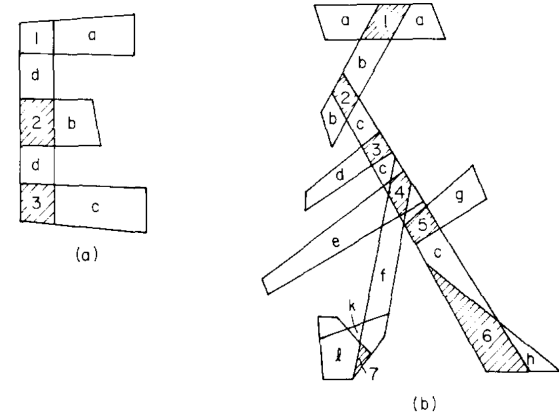
$\exists\mathbb{R}$ : Given polynomial  $P(x_1, \dots, x_k)$  with integer coefficients, exists real solution to  $P(x_1, \dots, x_k) = 0$ ?



Art Gallery Problem is  $\exists\mathbb{R}$ -complete  
(**A.**, Adamaszek, Miltzow, 2018)

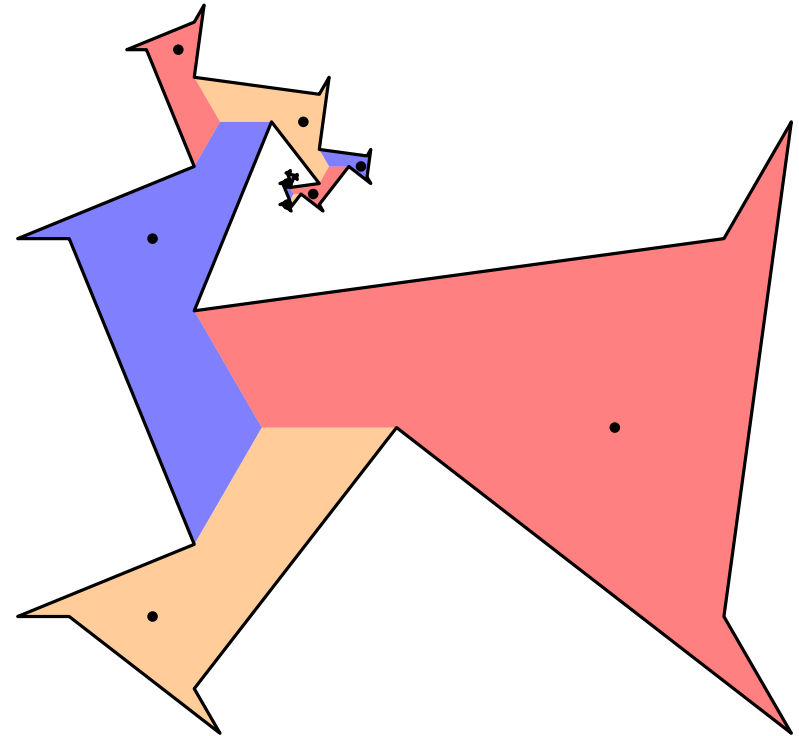
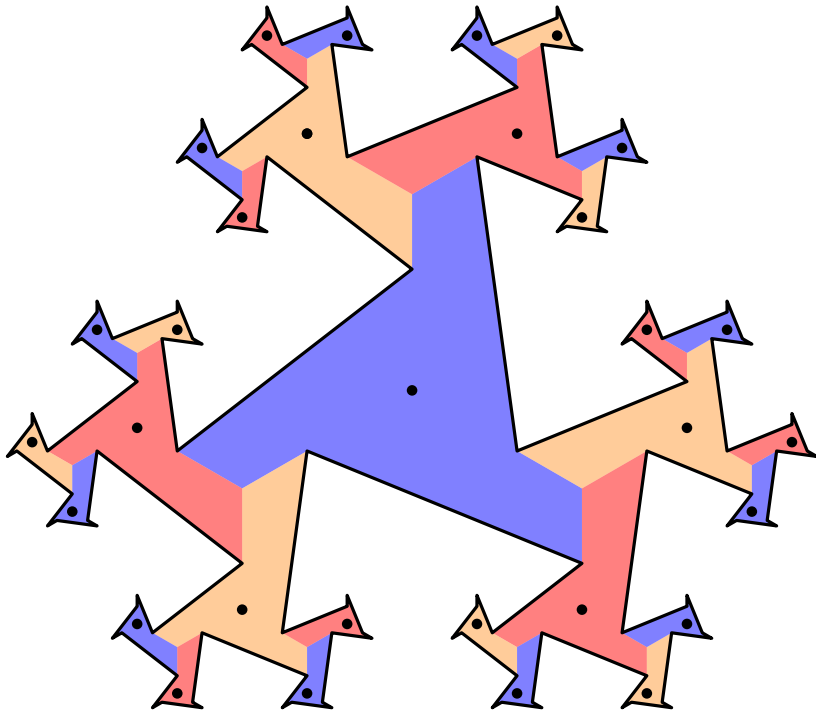


Minimum Convex Cover is  $\exists\mathbb{R}$ -complete  
(**A.**, 2021)



# Recent work on decomposition problems

Minimum star partitions in polynomial time ( $O(n^{107})$ )



**A.**, Blikstad, Nusser, Zhang, 2023

# Introduction to Approximation Algorithms, part I

08-1 2025, Srikanth Srinivasan,  
DIKU (Slides: Mikkel Abrahamsen)

APPROX-VERTEX-COVER( $G$ )

$C := \emptyset$

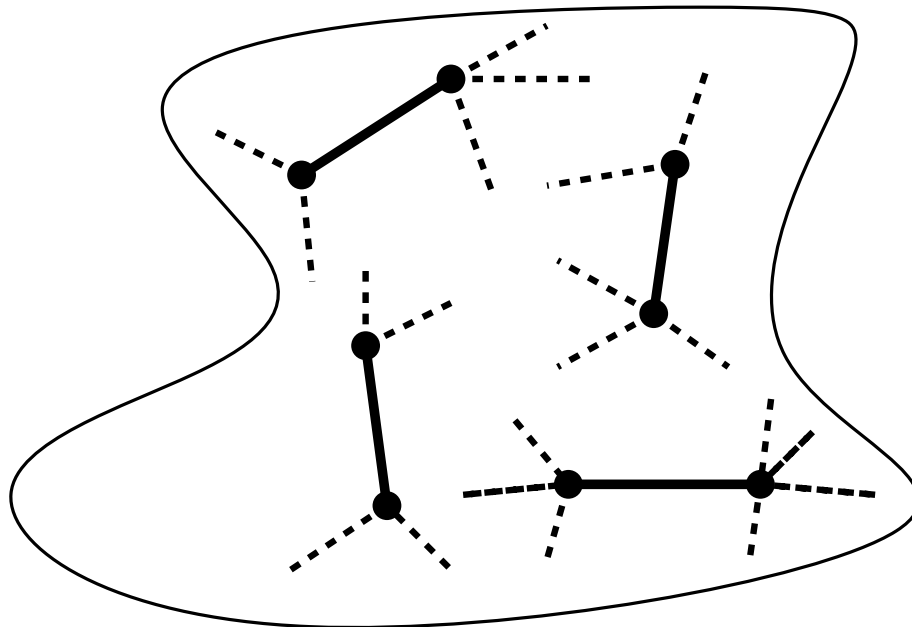
while  $E(G) \neq \emptyset$

  choose  $uv \in E(G)$

$C := C \cup \{u, v\}$

  remove all edges incident on  $u$  or  $v$  from  $E(G)$

return  $C$



# The big picture

**Previously:** Fast exponential algorithms (good for small instances) and parameterized algorithms (good for special cases).

# The big picture

**Previously:** Fast exponential algorithms (good for small instances) and parameterized algorithms (good for special cases).

**Today:** Approximation algorithms (good when suboptimal solutions are acceptable).

## Definition

**Def.:** An algorithm for an optimization problem has *approximation ratio*  $\rho(n)$  if for every input of size  $n$ ,

$$\max \left\{ \frac{C}{C^*}, \frac{C^*}{C} \right\} \leq \rho(n).$$



## Definition

**Def.:** An algorithm for an optimization problem has *approximation ratio*  $\rho(n)$  if for every input of size  $n$ ,

$$\max \left\{ \frac{C}{C^*}, \frac{C^*}{C} \right\} \leq \rho(n).$$

$C^* := \text{cost}(\text{opt. sol.})$    $C := \text{cost}(\text{produced sol.})$  

# Definition

**Def.:** An algorithm for an optimization problem has *approximation ratio*  $\rho(n)$  if for every input of size  $n$ ,

$$\max \left\{ \frac{C}{C^*}, \frac{C^*}{C} \right\} \leq \rho(n).$$

$C^* := \text{cost}(\text{opt. sol.})$

$C := \text{cost}(\text{produced sol.})$

minimization problem

maximization problem

# Vertex Cover

**Def.:** Let  $G = (V, E)$  be a graph. A set  $V' \subseteq V$  of vertices is a *vertex cover* if for all  $uv \in E$ , we have  $u \in V'$  or  $v \in V'$ .

# Vertex Cover

**Def.:** Let  $G = (V, E)$  be a graph. A set  $V' \subseteq V$  of vertices is a *vertex cover* if for all  $uv \in E$ , we have  $u \in V'$  or  $v \in V'$ .

NP-hard!

# Vertex Cover

**Def.:** Let  $G = (V, E)$  be a graph. A set  $V' \subseteq V$  of vertices is a *vertex cover* if for all  $uv \in E$ , we have  $u \in V'$  or  $v \in V'$ .

NP-hard!

APPROX-VERTEX-COVER( $G$ )

$C := \emptyset$

while  $E(G) \neq \emptyset$

    choose  $uv \in E(G)$

$C := C \cup \{u, v\}$

    remove all edges incident on  $u$  or  $v$  from  $E(G)$

return  $C$

# Vertex Cover

**Def.:** Let  $G = (V, E)$  be a graph. A set  $V' \subseteq V$  of vertices is a *vertex cover* if for all  $uv \in E$ , we have  $u \in V'$  or  $v \in V'$ .

NP-hard!

APPROX-VERTEX-COVER( $G$ )

$C := \emptyset$

while  $E(G) \neq \emptyset$

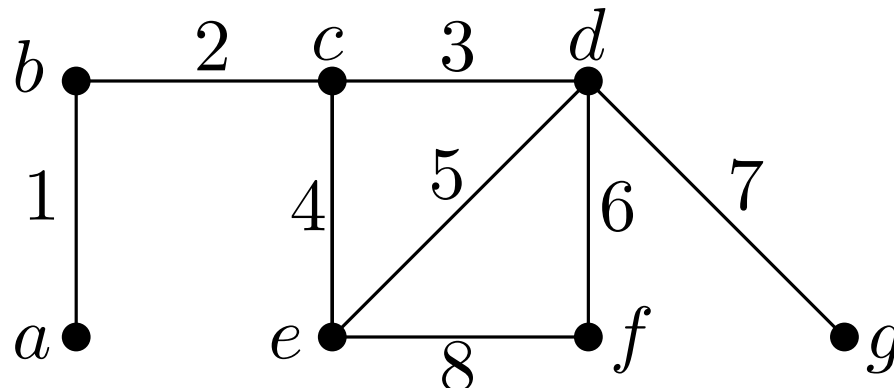
  choose  $uv \in E(G)$

$C := C \cup \{u, v\}$

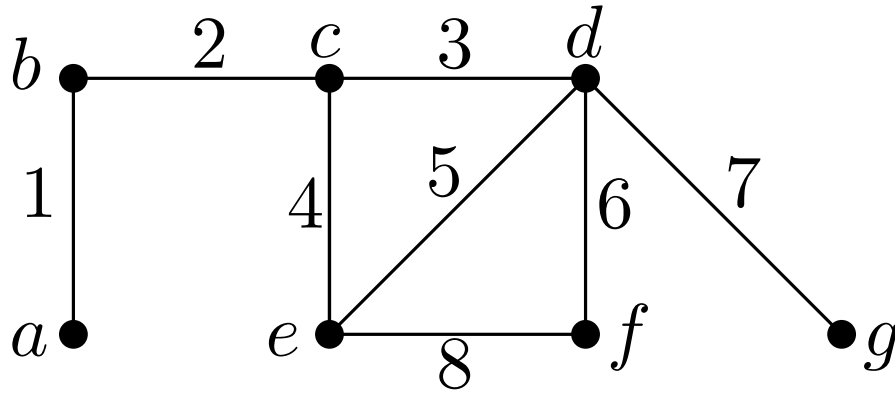
  remove all edges incident on  $u$  or  $v$  from  $E(G)$

return  $C$

**Exercise:**



# Implementation



Adjacency lists:

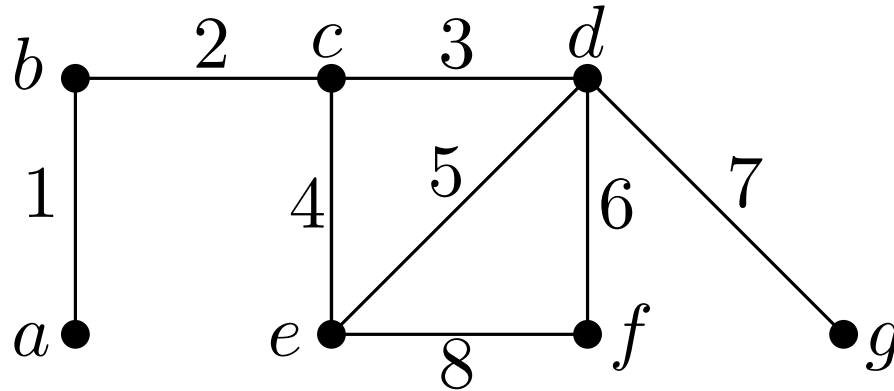
$$L[a] = \{b\}$$

$$L[b] = \{a, c\}$$

$$L[c] = \{b, d, e\}$$

$\vdots$

# Implementation



Adjacency lists:

$$\begin{aligned} L[a] &= \{b\} & E[a] &= [1] \\ L[b] &= \{a, c\} & E[b] &= [1, 2] \\ L[c] &= \{b, d, e\} & E[c] &= [2, 3, 4] \end{aligned}$$

$\vdots$

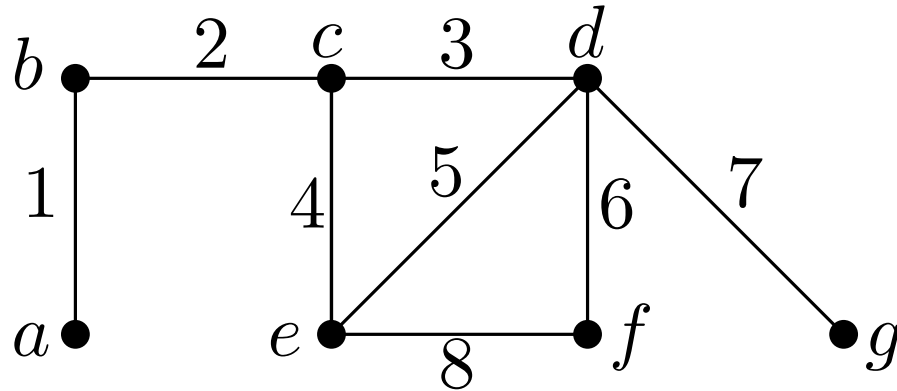
$\vdots$

Array of edges

$[(a, b, 1), (b, c, 1), (c, d, 1), (c, e, 1), (d, e, 1), \dots]$



# Implementation



Adjacency lists:

$$\begin{aligned} L[a] &= \{b\} & E[a] &= [1] \\ L[b] &= \{a, c\} & E[b] &= [1, 2] \\ L[c] &= \{b, d, e\} & E[c] &= [2, 3, 4] \end{aligned}$$

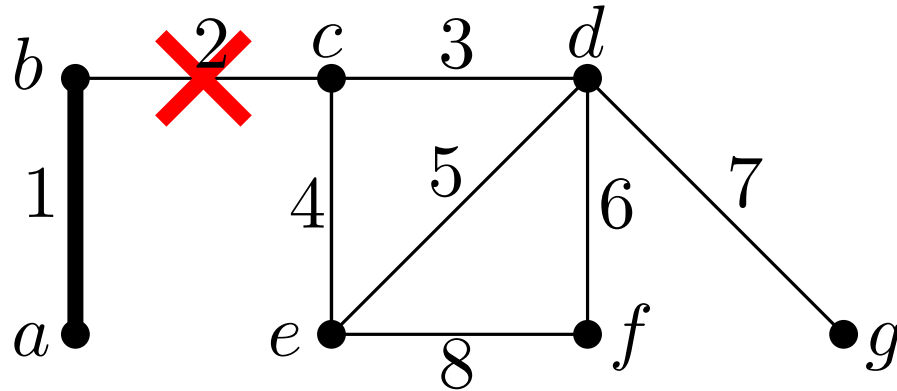
$\vdots$

$\vdots$

Array of edges

$[(\underline{a, b, 1}), (b, c, 1), (c, d, 1), (c, e, 1), (d, e, 1), \dots]$

# Implementation



Adjacency lists:

$L[a] = \{b\}$        $E[a] = [1]$   
 $L[b] = \{a, c\}$      $E[b] = [1, 2]$   
 $L[c] = \{b, d, e\}$   $E[c] = [2, 3, 4]$

⋮

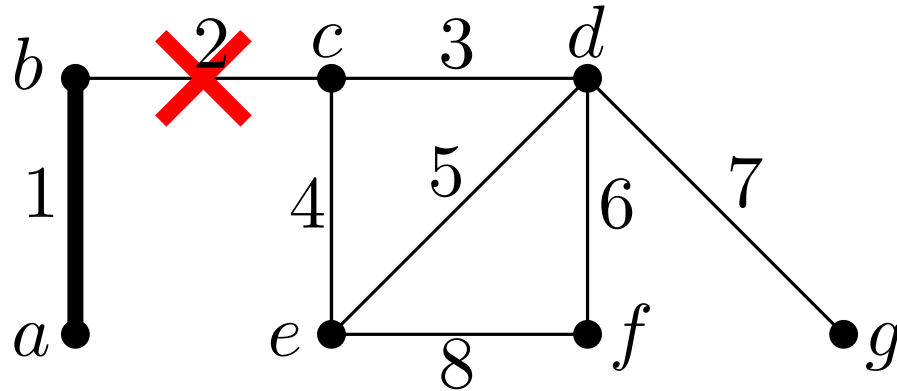
⋮

Array of edges

$[(a, b, 1), (b, c, 1), (c, d, 1), (c, e, 1), (d, e, 1), \dots]$

$[(a, b, 0), (b, c, 0), (c, d, 1), (c, e, 1), (d, e, 1), \dots]$

# Implementation



Adjacency lists:

$L[a] = \{b\}$        $E[a] = [1]$   
 $L[b] = \{a, c\}$      $E[b] = [1, 2]$   
 $L[c] = \{b, d, e\}$   $E[c] = [2, 3, 4]$

⋮

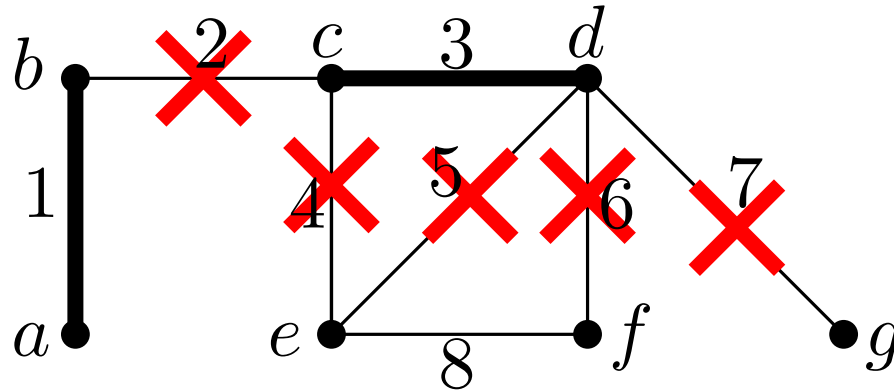
⋮

Array of edges

$[(a, b, 1), (b, c, 1), (c, d, 1), (c, e, 1), (d, e, 1), \dots]$

$[(a, b, 0), (b, c, 0), \underline{(c, d, 1)}, (c, e, 1), (d, e, 1), \dots]$

# Implementation



Adjacency lists:

$$\begin{aligned}
 L[a] &= \{b\} & E[a] &= [1] \\
 L[b] &= \{a, c\} & E[b] &= [1, 2] \\
 L[c] &= \{b, d, e\} & E[c] &= [2, 3, 4]
 \end{aligned}$$

$\vdots$

$\vdots$

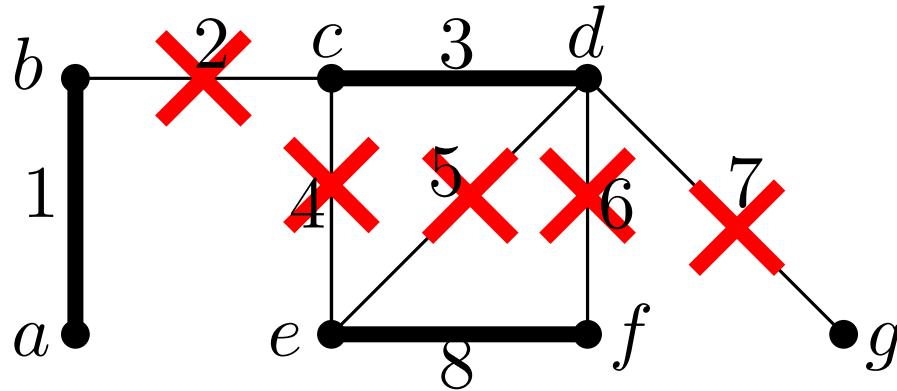
Array of edges

$[(a, b, 1), (b, c, 1), (c, d, 1), (c, e, 1), (d, e, 1), \dots]$

$[(a, b, 0), (b, c, 0), (c, d, 1), (c, e, 1), (d, e, 1), \dots]$

$[(a, b, 0), (b, c, 0), (c, d, 0), (c, e, 0), (d, e, 0), \dots]$

# Implementation



Adjacency lists:

$$\begin{aligned} L[a] &= \{b\} & E[a] &= [1] \\ L[b] &= \{a, c\} & E[b] &= [1, 2] \\ L[c] &= \{b, d, e\} & E[c] &= [2, 3, 4] \end{aligned}$$

$\vdots$

$\vdots$

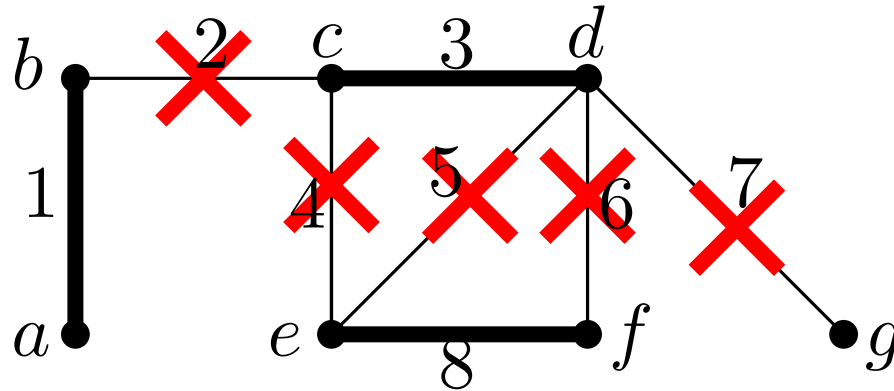
Array of edges

$[(a, b, 1), (b, c, 1), (c, d, 1), (c, e, 1), (d, e, 1), \dots]$

$[(a, b, 0), (b, c, 0), (c, d, 1), (c, e, 1), (d, e, 1), \dots]$

$[(a, b, 0), (b, c, 0), (c, d, 0), (c, e, 0), (d, e, 0), \dots]$

# Implementation



Adjacency lists:

$L[a] = \{b\}$        $E[a] = [1]$   
 $L[b] = \{a, c\}$      $E[b] = [1, 2]$   
 $L[c] = \{b, d, e\}$   $E[c] = [2, 3, 4]$

⋮

⋮

Array of edges

$[(a, b, 1), (b, c, 1), (c, d, 1), (c, e, 1), (d, e, 1), \dots]$

$[(a, b, 0), (b, c, 0), (c, d, 1), (c, e, 1), (d, e, 1), \dots]$

$[(a, b, 0), (b, c, 0), (c, d, 0), (c, e, 0), (d, e, 0), \dots]$

Running time:  $O(|V| + |E|)$

# Theorem

**Thm.:** APPROX-VERTEX-COVER is a 2-approximation algorithm.

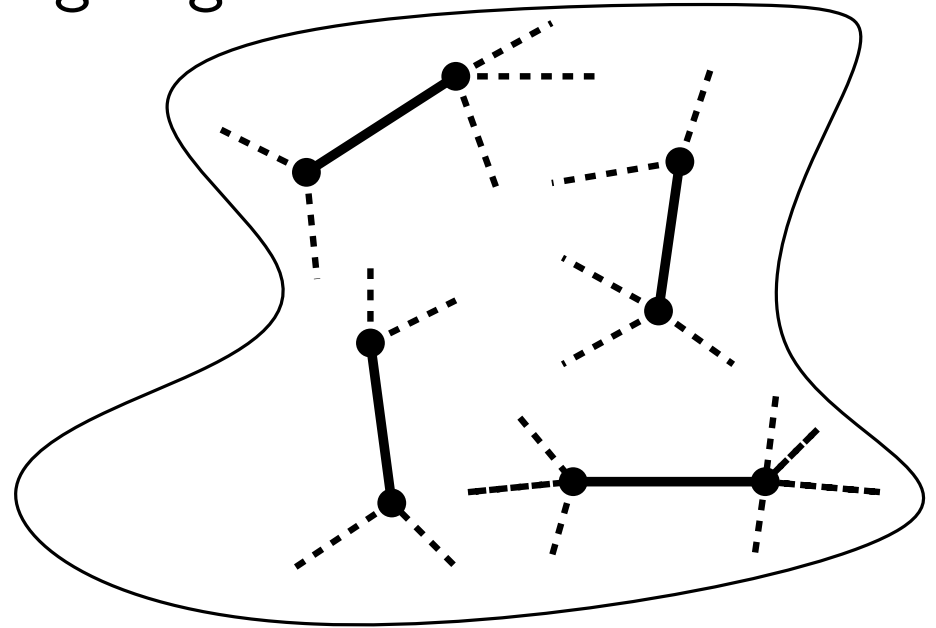
# Theorem

**Thm.:** APPROX-VERTEX-COVER is a 2-approximation algorithm.

*Proof:* Let  $C^*$  be an optimal cover.

Let  $A \subset E$  be the edges chosen by the algorithm.

Obs: No shared endpoints among edges in  $A$





# Theorem

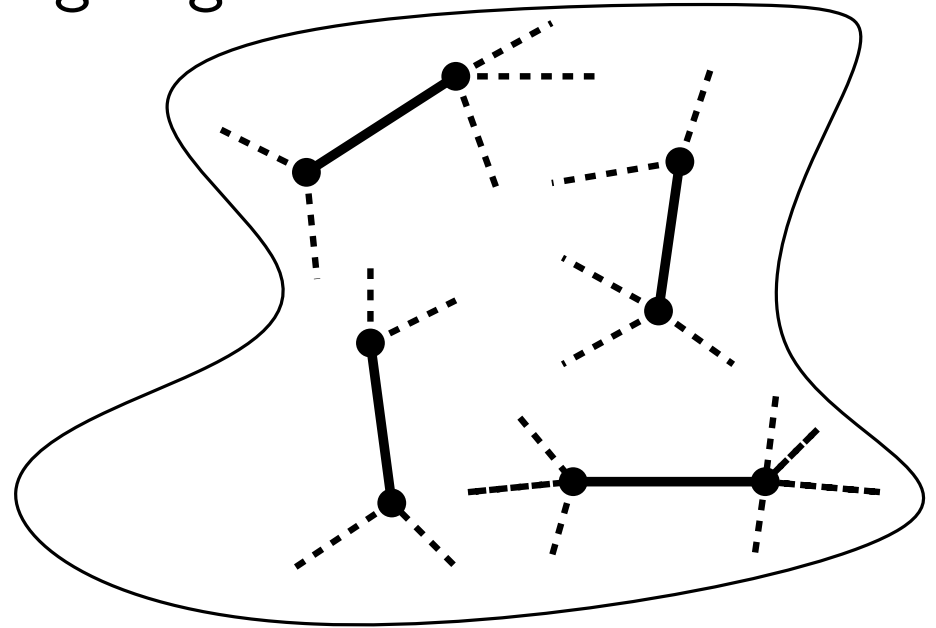
**Thm.:** APPROX-VERTEX-COVER is a 2-approximation algorithm.

*Proof:* Let  $C^*$  be an optimal cover.

Let  $A \subset E$  be the edges chosen by the algorithm.

Obs: No shared endpoints among edges in  $A$

An endpoint of each  
 $uv \in A$  must be in  $C^*$ .



# Theorem

**Thm.:** APPROX-VERTEX-COVER is a 2-approximation algorithm.

*Proof:* Let  $C^*$  be an optimal cover.

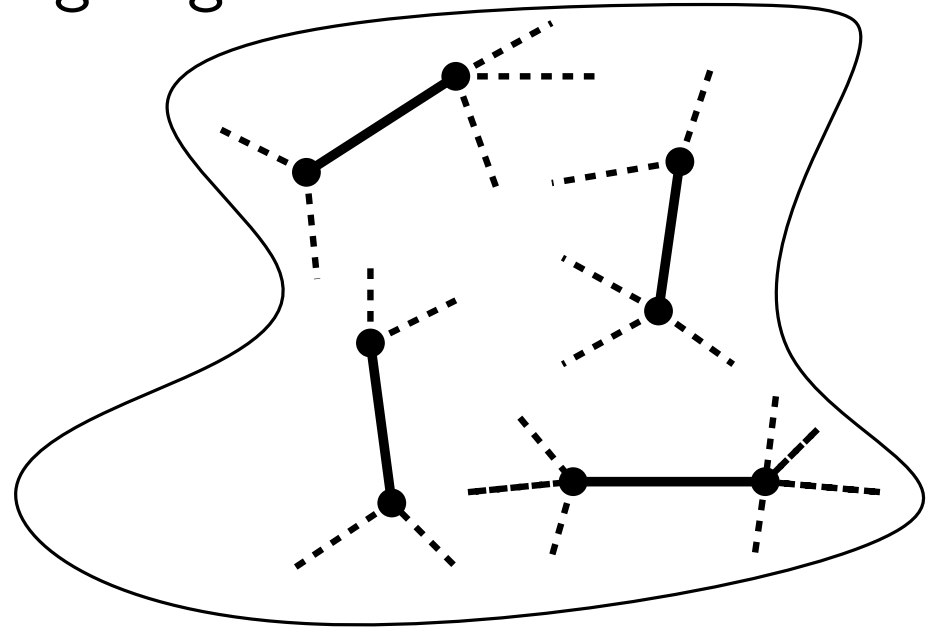
Let  $A \subset E$  be the edges chosen by the algorithm.

Obs: No shared endpoints among edges in  $A$

An endpoint of each  $uv \in A$  must be in  $C^*$ .

Hence,

$$|C^*| \geq |A| = |C|/2 \implies \frac{|C|}{|C^*|} \leq 2.$$



## Reflection and methodology

How can we prove  $C/C^* \leq 2$  when we don't know  $C^*$ ?

Answer: By proving  $C \leq 2|A|$  and  $|A| \leq C^*$ .

## Reflection and methodology

How can we prove  $C/C^* \leq 2$  when we don't know  $C^*$ ?

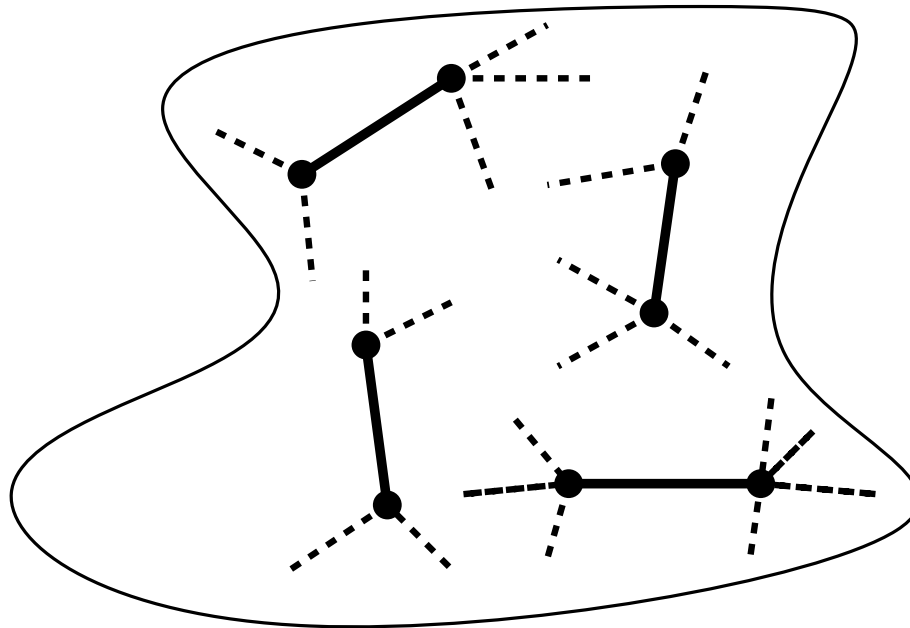
Answer: By proving  $C \leq 2|A|$  and  $|A| \leq C^*$ .

General technique: Find a parameter  $\square$  such that  $C \leq \rho \cdot \square$  and  $\square \leq C^*$ .

For vertex cover:  $\square = |A|$  and  $\rho = 2$ .

# Question

Try to guess: Is there an approximation algorithm with a better approximation ratio?



# History

1972: Karp's 21 NP-complete problems  
(including vertex cover, set cover, Hamiltonian cycle and subset sum)



Karp

Turing Award

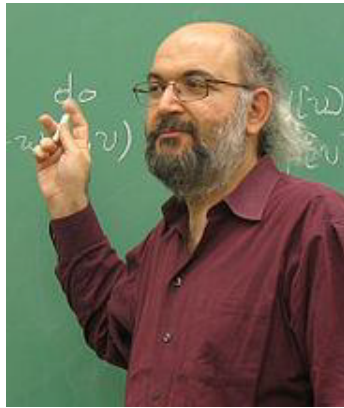


# History

19xx:  $\text{Many} \leq 2 - o(1)$ .



Gavril



Yannakakis



...

# History

Assuming  $P \neq NP$ :

1999: Håstad,  $\geq 7/6$

2005: Dinur & Safra,  $\geq 1.38$

2018: Khot, Minzer, Safra,  $\geq 1.41$



Håstad



Dinur



Safra



Khot



Minzer



# History

2008: Khot & Regev,  $\geq 2 - \varepsilon$  assuming the  
Unique Game Conjecture.

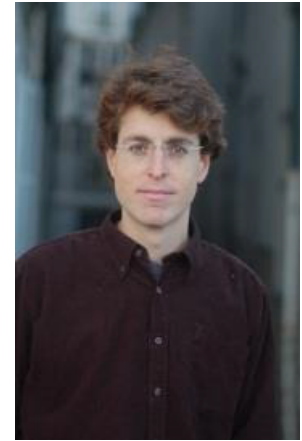
Some, but not all people believe it.



Khot



Nevanlinna prize 2016



Regev

# Traveling Salesperson

Given a complete undirected graph  $G = (V, E)$ .

For all  $u, v \in V$ , we are given  $c(uv) \in \{0, 1, \dots\}$ .

**Goal:** Find minimum weight cycle through all vertices.

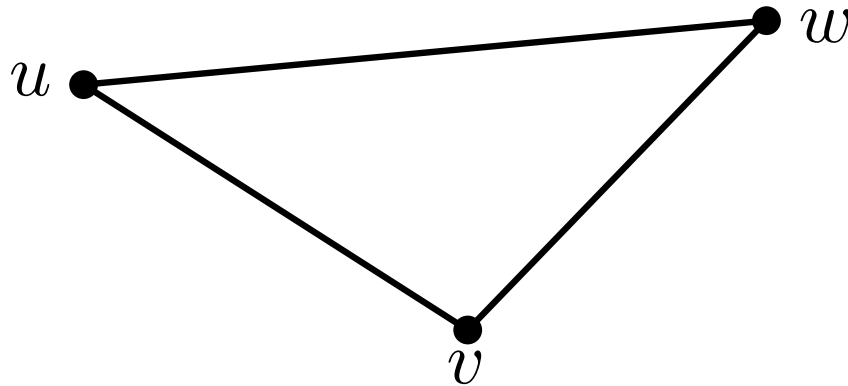
# Traveling Salesperson

Given a complete undirected graph  $G = (V, E)$ .

For all  $u, v \in V$ , we are given  $c(uv) \in \{0, 1, \dots\}$ .

**Goal:** Find minimum weight cycle through all vertices.

**Assume:** Triangle inequality:  $c(uw) \leq c(uv) + c(vw)$ .



# Traveling Salesperson

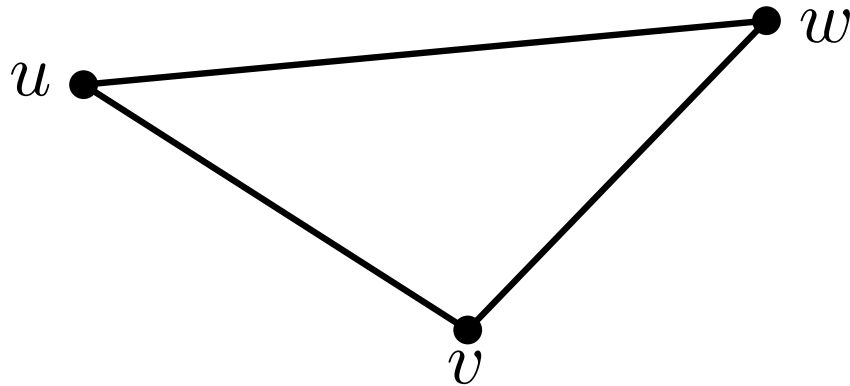
Given a complete undirected graph  $G = (V, E)$ .

For all  $u, v \in V$ , we are given  $c(uv) \in \{0, 1, \dots\}$ .

**Goal:** Find minimum weight cycle through all vertices.

**Assume:** Triangle inequality:  $c(uw) \leq c(uv) + c(vw)$ .

Still NP-hard!



# Algorithm

APPROX-TSP( $G, c$ )

Find MST  $T$

Make Euler tour  $W$  using each edge of  $T$  twice

Shortcut  $W$  to  $H$  by skipping duplicates

Return  $H$

# Algorithm

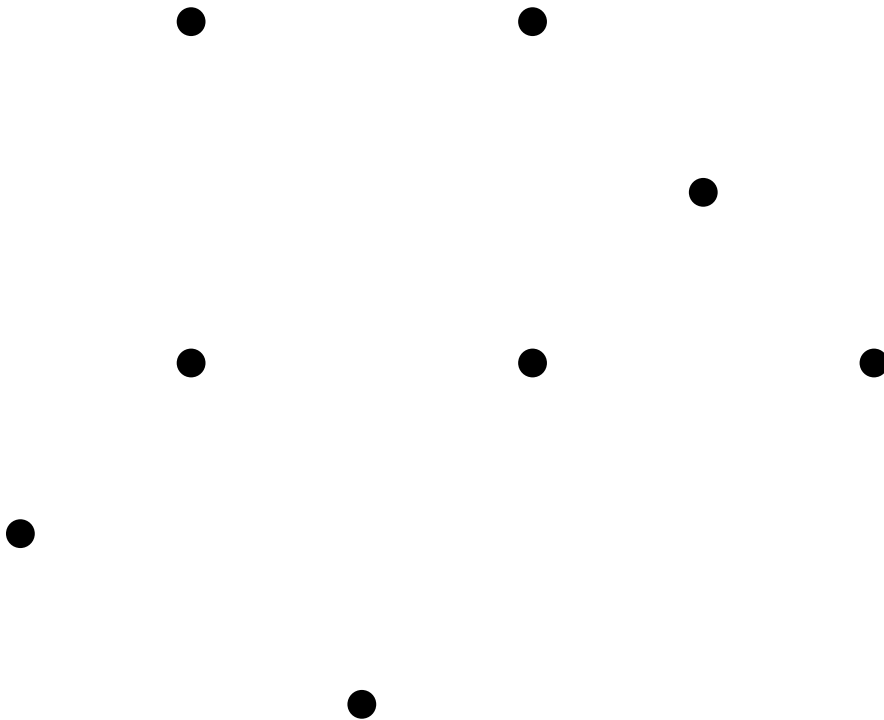
APPROX-TSP( $G, c$ )

Find MST  $T$

Make Euler tour  $W$  using each edge of  $T$  twice

Shortcut  $W$  to  $H$  by skipping duplicates

Return  $H$



# Algorithm

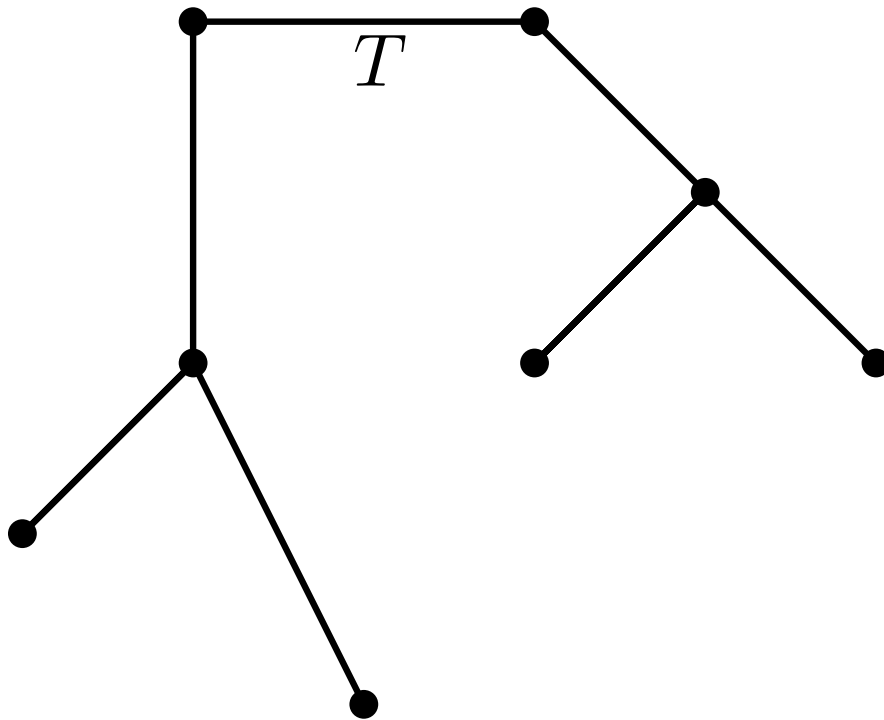
APPROX-TSP( $G, c$ )

Find MST  $T$

Make Euler tour  $W$  using each edge of  $T$  twice

Shortcut  $W$  to  $H$  by skipping duplicates

Return  $H$



# Algorithm

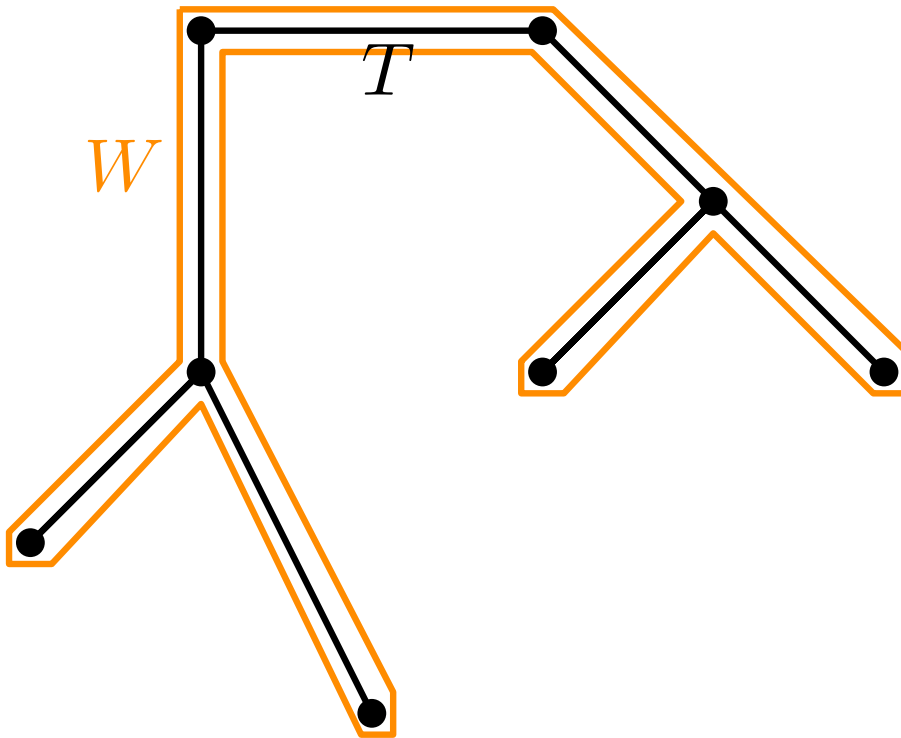
APPROX-TSP( $G, c$ )

Find MST  $T$

Make Euler tour  $W$  using each edge of  $T$  twice

Shortcut  $W$  to  $H$  by skipping duplicates

Return  $H$





# Algorithm

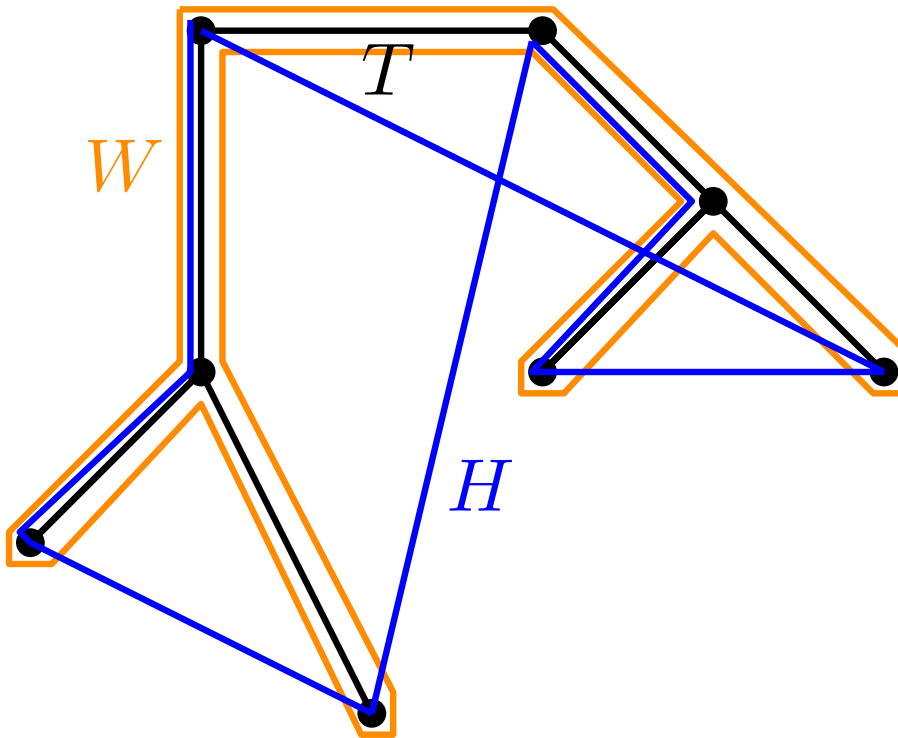
APPROX-TSP( $G, c$ )

Find MST  $T$

Make Euler tour  $W$  using each edge of  $T$  twice

Shortcut  $W$  to  $H$  by skipping duplicates

Return  $H$



# Algorithm

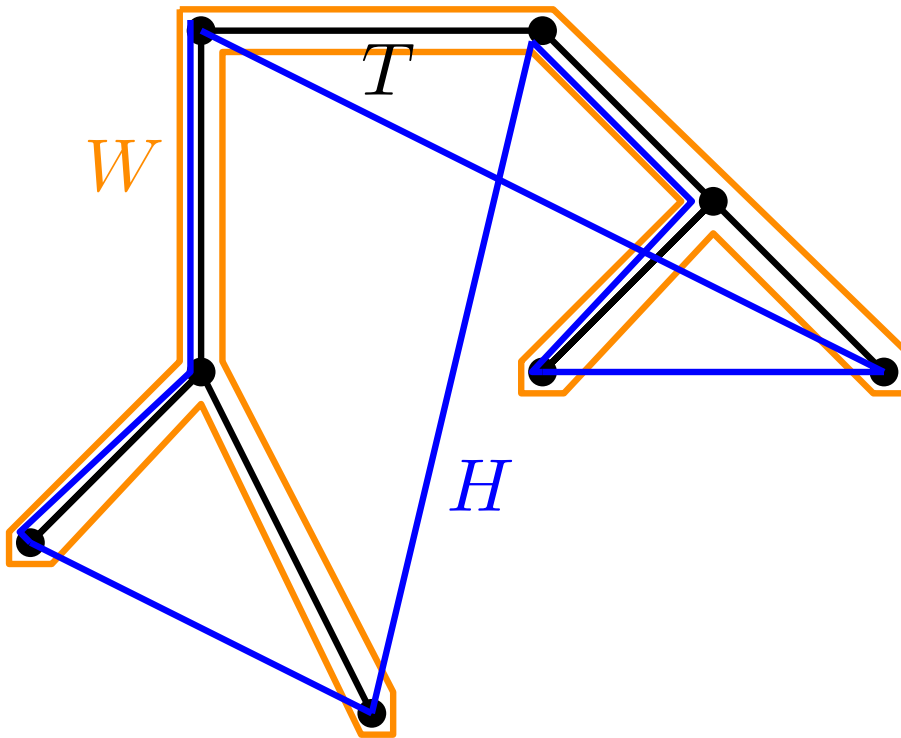
APPROX-TSP( $G, c$ )

Find MST  $T$

Make Euler tour  $W$  using each edge of  $T$  twice

Shortcut  $W$  to  $H$  by skipping duplicates

Return  $H$



Exercise: Run the algorithm on this instance.

$a$

$b$

$d$

$c$

# Theorem

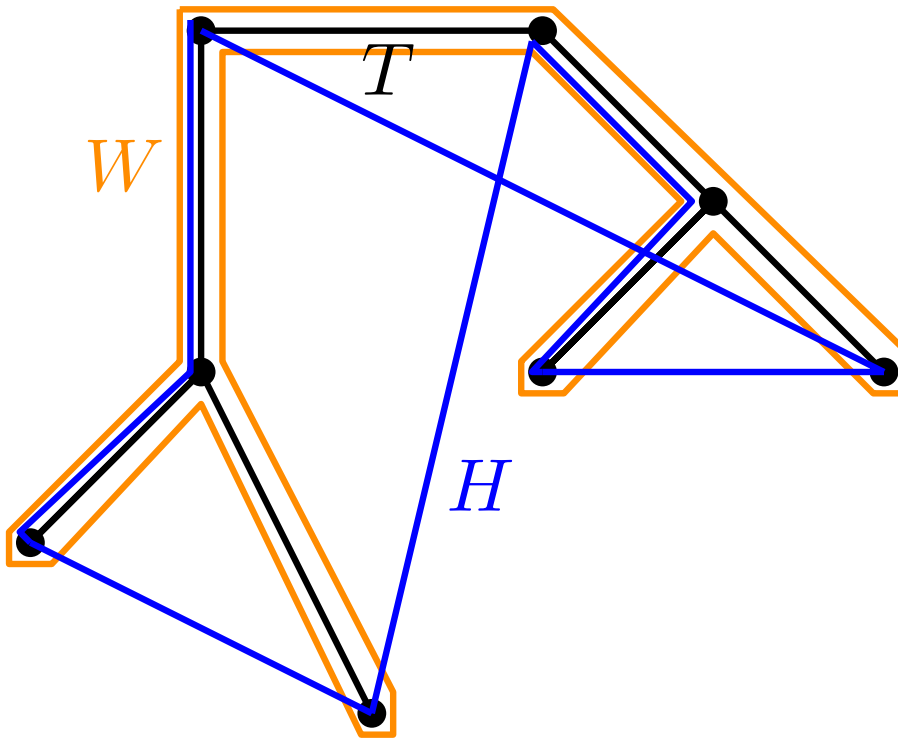
APPROX-TSP( $G, c$ )

Find MST  $T$

Make Euler tour  $W$  using each edge of  $T$  twice

Shortcut  $W$  to  $H$  by skipping duplicates

Return  $H$



**Thm.:** APPROX-TSP is a  
poly-time 2-approx. alg.

# Theorem

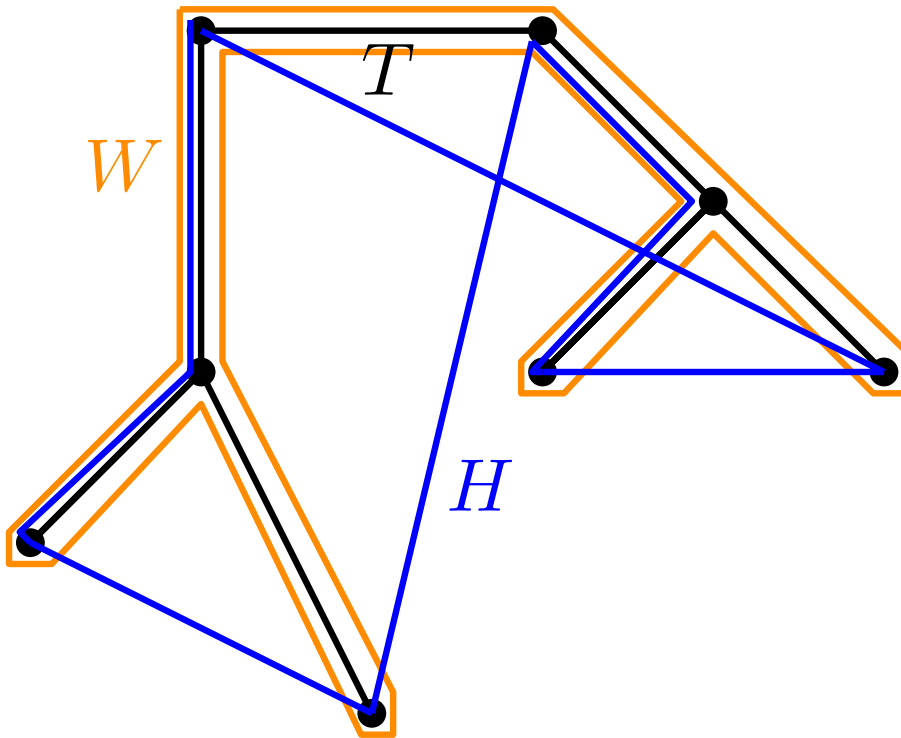
APPROX-TSP( $G, c$ )

Find MST  $T$

Make Euler tour  $W$  using each edge of  $T$  twice

Shortcut  $W$  to  $H$  by skipping duplicates

Return  $H$



**Thm.:** APPROX-TSP is a poly-time 2-approx. alg.

*Proof:* Poly-time?

Let  $H^*$  be an opt. sol.

# Theorem

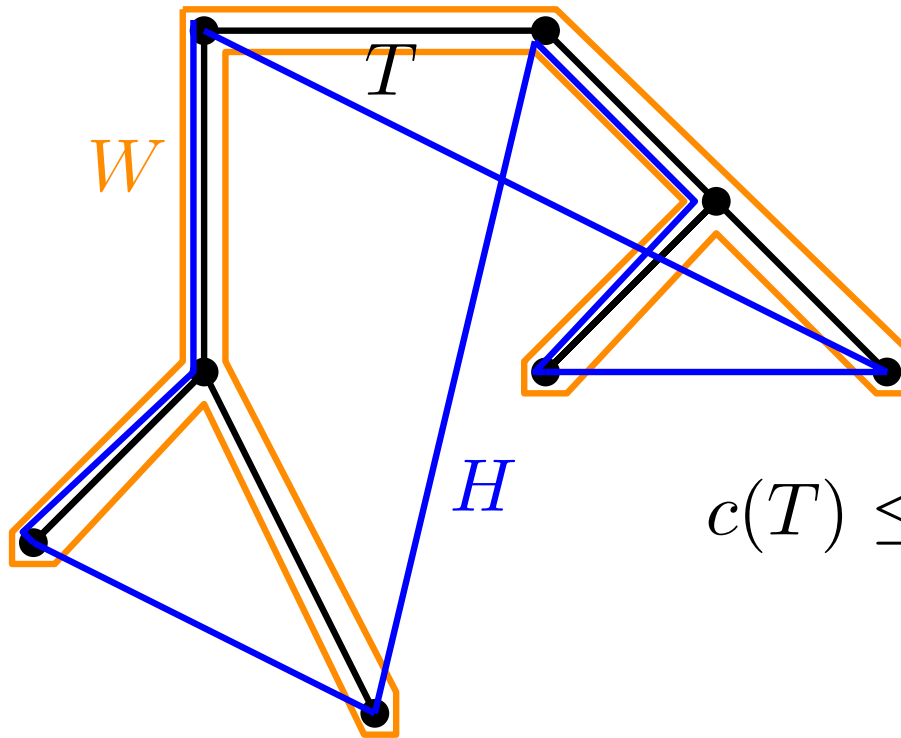
APPROX-TSP( $G, c$ )

Find MST  $T$

Make Euler tour  $W$  using each edge of  $T$  twice

Shortcut  $W$  to  $H$  by skipping duplicates

Return  $H$



**Thm.:** APPROX-TSP is a poly-time 2-approx. alg.

*Proof:* Poly-time?

Let  $H^*$  be an opt. sol.

$$c(T) \leq c(H^*)$$

# Theorem

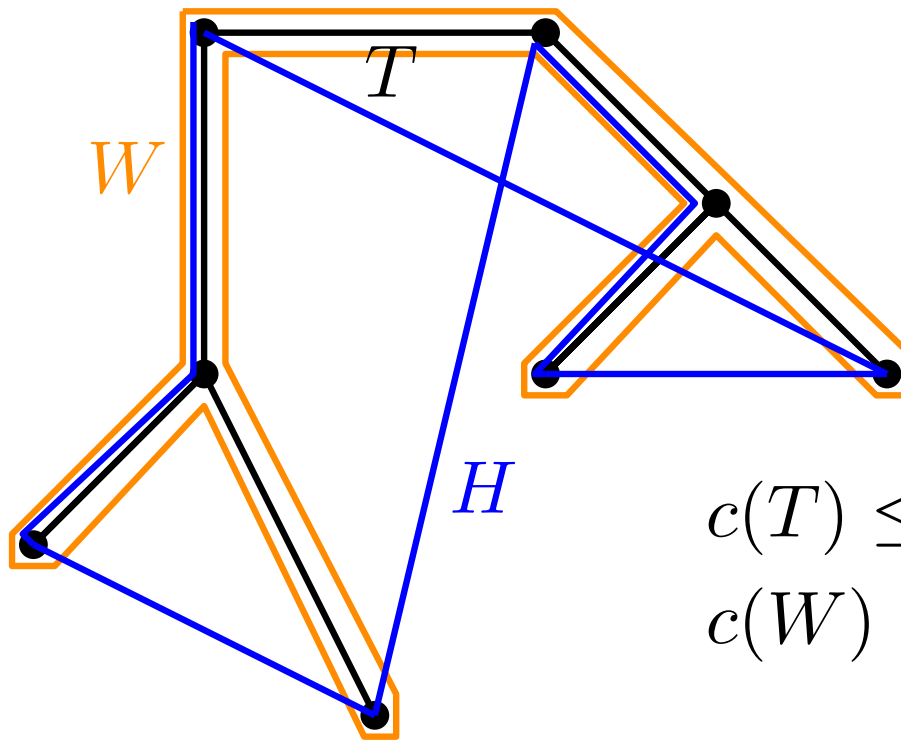
APPROX-TSP( $G, c$ )

Find MST  $T$

Make Euler tour  $W$  using each edge of  $T$  twice

Shortcut  $W$  to  $H$  by skipping duplicates

Return  $H$



**Thm.:** APPROX-TSP is a poly-time 2-approx. alg.

*Proof:* Poly-time?

Let  $H^*$  be an opt. sol.

$$c(T) \leq c(H^*)$$

$$c(W) = 2c(T)$$

# Theorem

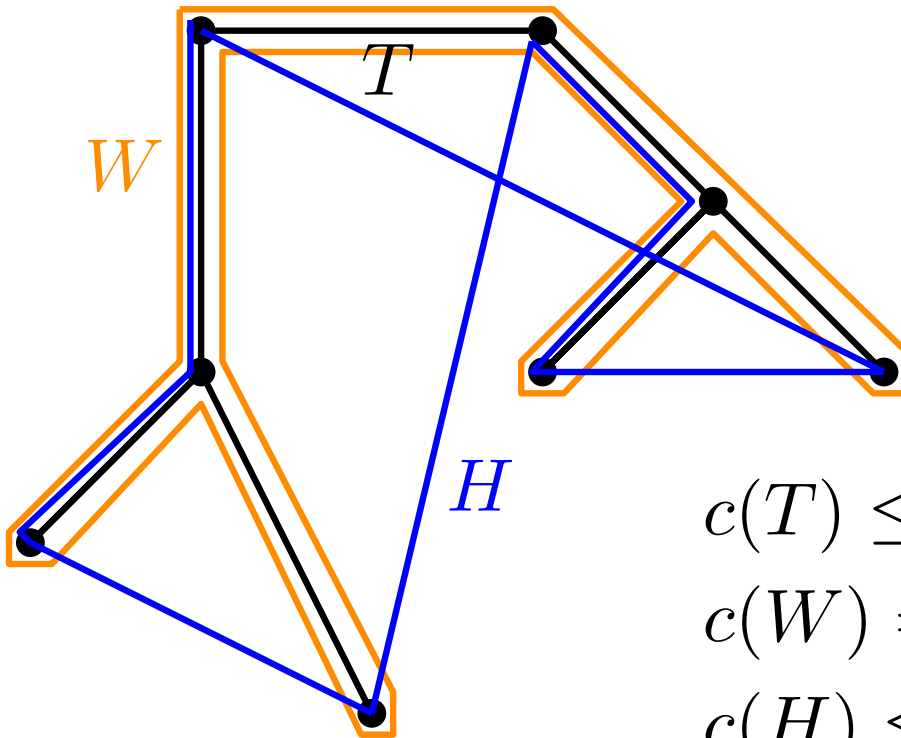
APPROX-TSP( $G, c$ )

Find MST  $T$

Make Euler tour  $W$  using each edge of  $T$  twice

Shortcut  $W$  to  $H$  by skipping duplicates

Return  $H$



**Thm.:** APPROX-TSP is a poly-time 2-approx. alg.

*Proof:* Poly-time?

Let  $H^*$  be an opt. sol.

$$c(T) \leq c(H^*)$$

$$c(W) = 2c(T)$$

$$c(H) \leq c(W)$$

# Theorem

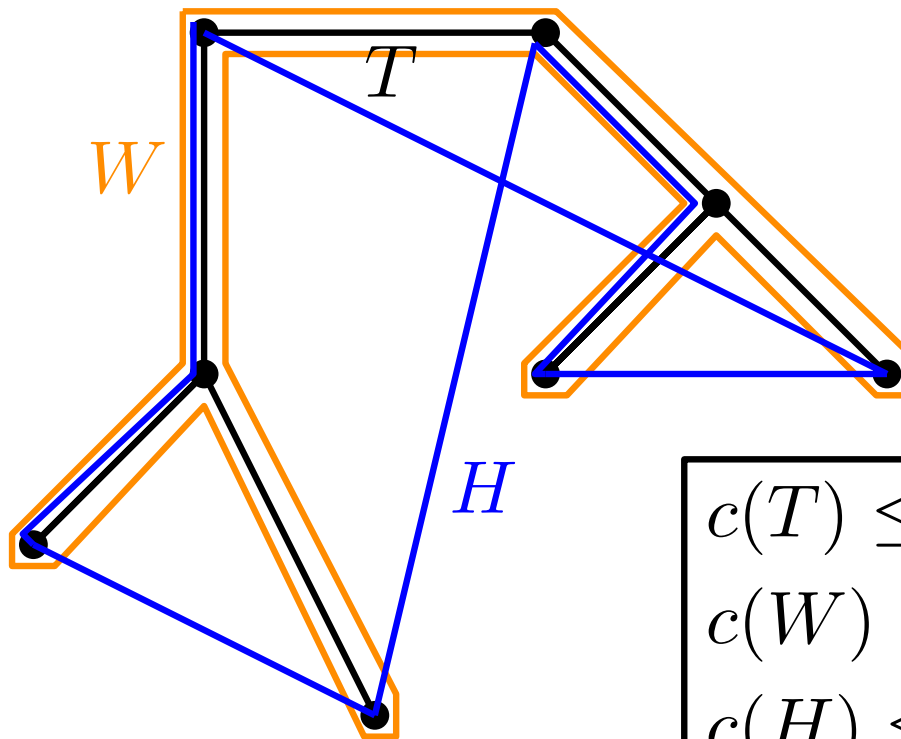
APPROX-TSP( $G, c$ )

Find MST  $T$

Make Euler tour  $W$  using each edge of  $T$  twice

Shortcut  $W$  to  $H$  by skipping duplicates

Return  $H$



**Thm.:** APPROX-TSP is a poly-time 2-approx. alg.

*Proof:* Poly-time?

Let  $H^*$  be an opt. sol.

$$c(T) \leq c(H^*)$$

$$c(W) = 2c(T) \implies c(H) \leq 2c(H^*)$$

$$c(H) \leq c(W)$$



## Reflection and methodology

How can we prove  $c(H)/c(H^*) \leq 2$  when we don't know  $H^*$ ?

Answer: By proving  $c(H) \leq 2c(T)$  and  $c(T) \leq c(H^*)$ .

# Reflection and methodology

How can we prove  $c(H)/c(H^*) \leq 2$  when we don't know  $H^*$ ?

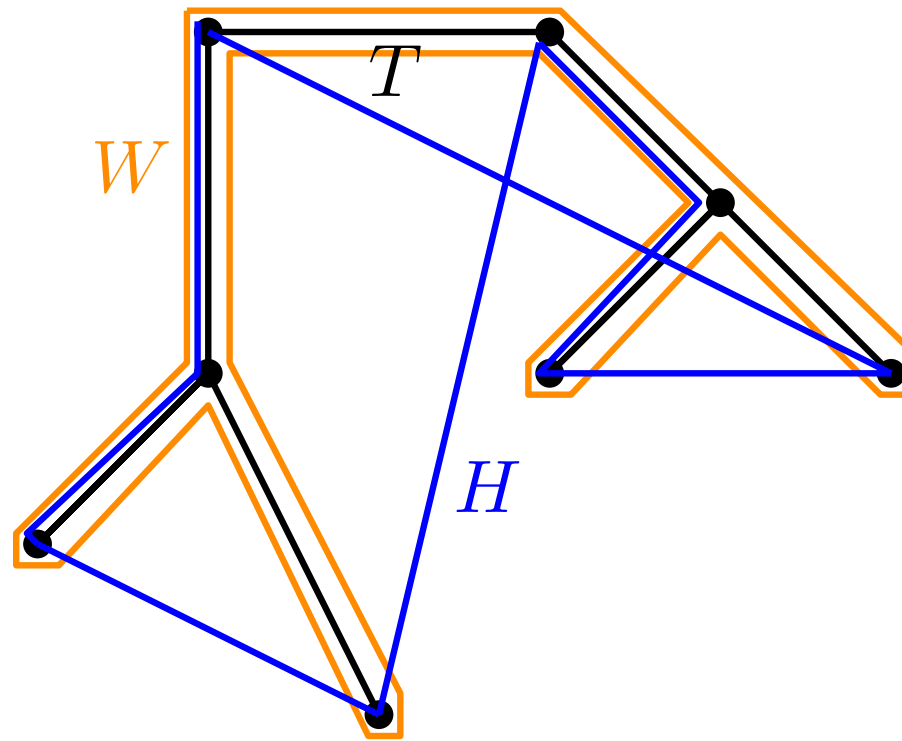
Answer: By proving  $c(H) \leq 2c(T)$  and  $c(T) \leq c(H^*)$ .

General technique: Find a parameter  $\square$  such that  $C \leq \rho \cdot \square$  and  $\square \leq C^*$ .

For TSP:  $\square = c(T)$  and  $\rho = 2$ .

# Question

Try to guess: Is there an approximation algorithm with a better approximation ratio?



# History

1976: Christofides, Serdyukov, 1.5-apx algorithm

It's simple! See, e.g., Wikipedia. No improvement for decades

2021: Karlin, Klein, Gharan,  $(1.5 - \varepsilon)$ -apx algorithm for some  $\varepsilon > 10^{-36}$



## Computer Scientists Break Traveling Salesperson Record

24 |

*After 44 years, there's finally a better way to find approximate solutions to the notoriously difficult traveling salesperson problem.*



# Set Cover

**Input:** Pair  $(X, \mathcal{F})$ , where  $X$  is a finite set and  $\mathcal{F} \subseteq \mathcal{P}(X)$  is a family of subsets of  $X$ .

# Set Cover

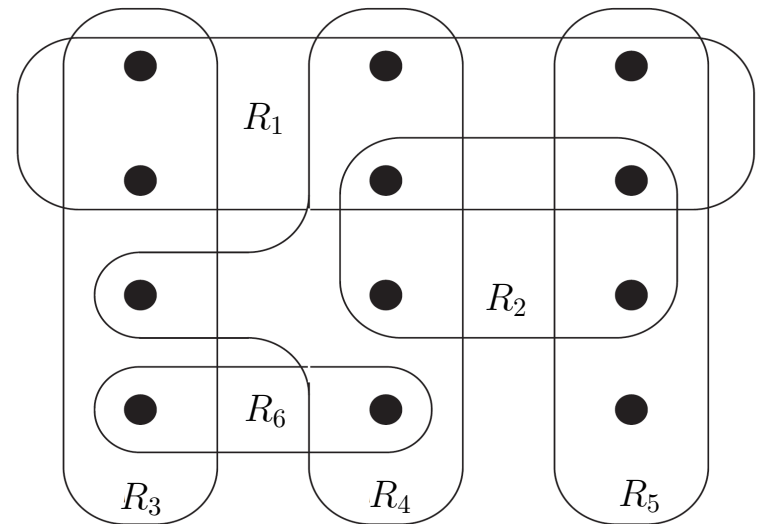
**Input:** Pair  $(X, \mathcal{F})$ , where  $X$  is a finite set and  $\mathcal{F} \subseteq \mathcal{P}(X)$  is a family of subsets of  $X$ .

**Goal:** Find  $\mathcal{C} \subseteq \mathcal{F}$  covering  $X$ , i.e.,  $\bigcup_{S \in \mathcal{C}} S = X$ , with  $|\mathcal{C}|$  minimum.

# Set Cover

**Input:** Pair  $(X, \mathcal{F})$ , where  $X$  is a finite set and  $\mathcal{F} \subseteq \mathcal{P}(X)$  is a family of subsets of  $X$ .

**Goal:** Find  $\mathcal{C} \subseteq \mathcal{F}$  covering  $X$ , i.e.,  $\bigcup_{S \in \mathcal{C}} S = X$ , with  $|\mathcal{C}|$  minimum.

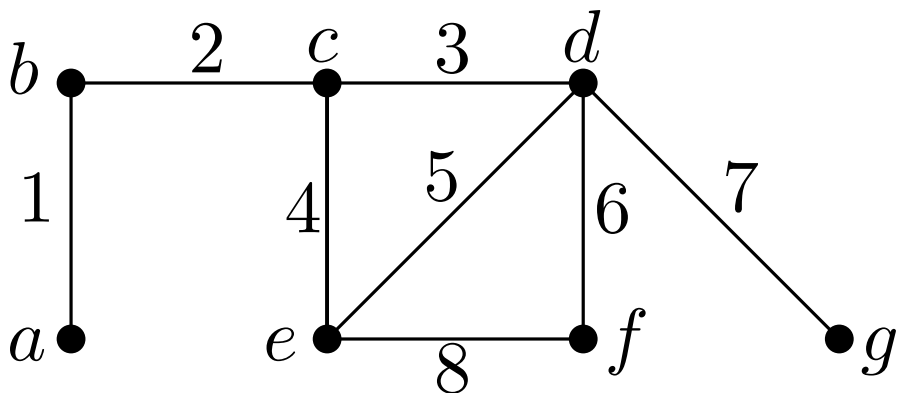
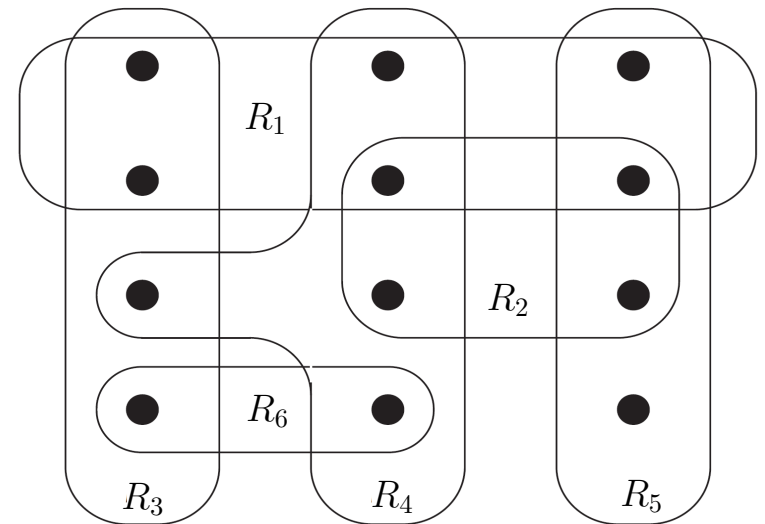


# Set Cover

**Input:** Pair  $(X, \mathcal{F})$ , where  $X$  is a finite set and  $\mathcal{F} \subseteq \mathcal{P}(X)$  is a family of subsets of  $X$ .

**Goal:** Find  $\mathcal{C} \subseteq \mathcal{F}$  covering  $X$ , i.e.,  $\bigcup_{S \in \mathcal{C}} S = X$ , with  $|\mathcal{C}|$  minimum.

**Exercise:** Show that vertex cover is a special case.





# Set Cover

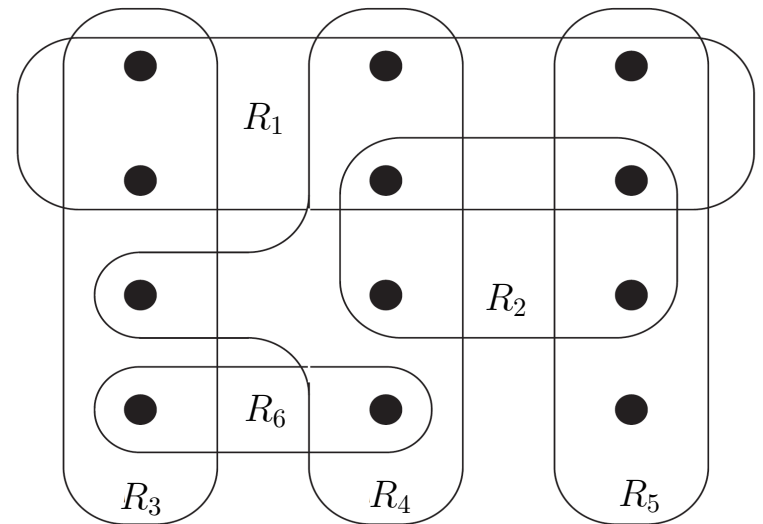
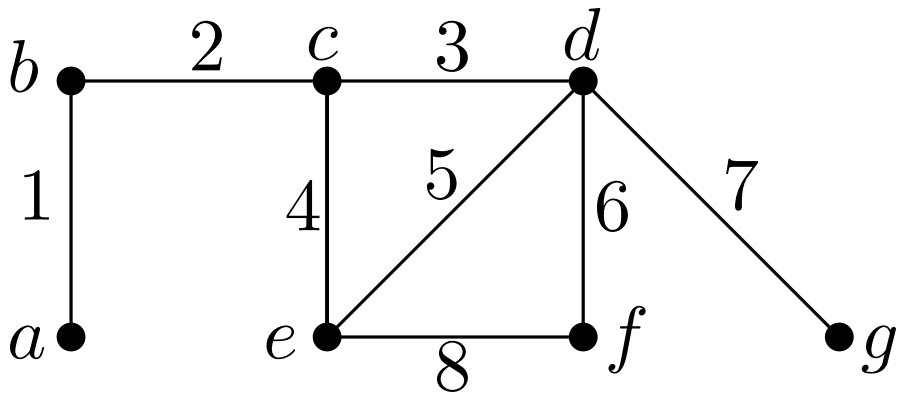
**Input:** Pair  $(X, \mathcal{F})$ , where  $X$  is a finite set and  $\mathcal{F} \subseteq \mathcal{P}(X)$  is a family of subsets of  $X$ .

**Goal:** Find  $\mathcal{C} \subseteq \mathcal{F}$  covering  $X$ , i.e.,  $\bigcup_{S \in \mathcal{C}} S = X$ , with  $|\mathcal{C}|$  minimum.

**Exercise:** Show that vertex cover is a special case.

$$X := \{1, 2, \dots, 8\}$$

$$\mathcal{F} := \{\{1\}, \{1, 2\}, \{2, 3, 4\}, \\ \{3, 5, 6, 7\}, \{4, 5, 8\}, \{6, 8\}, \{7\}\}$$



# Set Cover

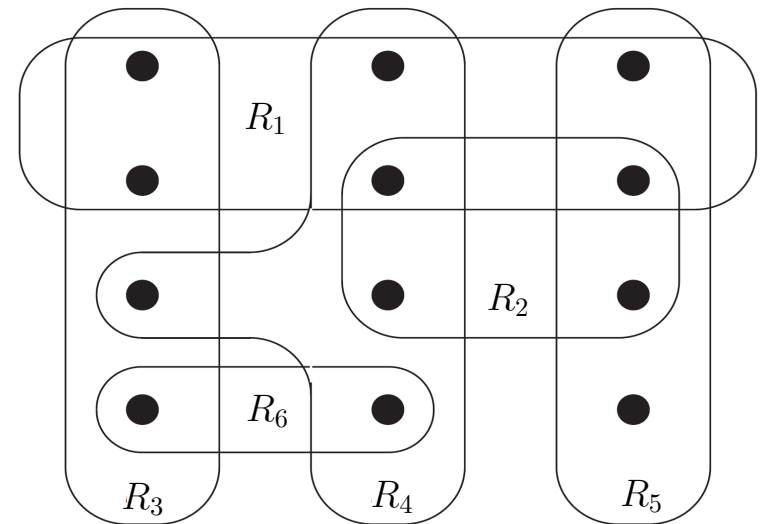
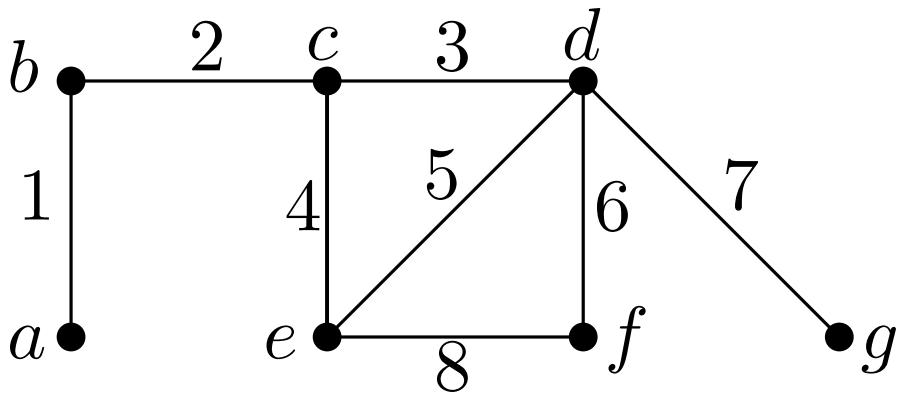
**Input:** Pair  $(X, \mathcal{F})$ , where  $X$  is a finite set and  $\mathcal{F} \subseteq \mathcal{P}(X)$  is a family of subsets of  $X$ .

**Goal:** Find  $\mathcal{C} \subseteq \mathcal{F}$  covering  $X$ , i.e.,  $\bigcup_{S \in \mathcal{C}} S = X$ , with  $|\mathcal{C}|$  minimum.

**Exercise:** Show that vertex cover is a special case.

$$X := \{1, 2, \dots, 8\}$$

$$\mathcal{F} := \{\{1\}, \{1, 2\}, \{2, 3, 4\}, \\ \{3, 5, 6, 7\}, \{4, 5, 8\}, \{6, 8\}, \{7\}\}$$



$$X := E$$

$$\mathcal{F} := \{E(v) \mid v \in V\}$$

$$E(v) := \{uv \in E \mid u \in V\}$$

# Greedy Algorithm

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

Here,  $S_{<i} := \bigcup_{j=1}^{i-1} S_j$ .

# Greedy Algorithm

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

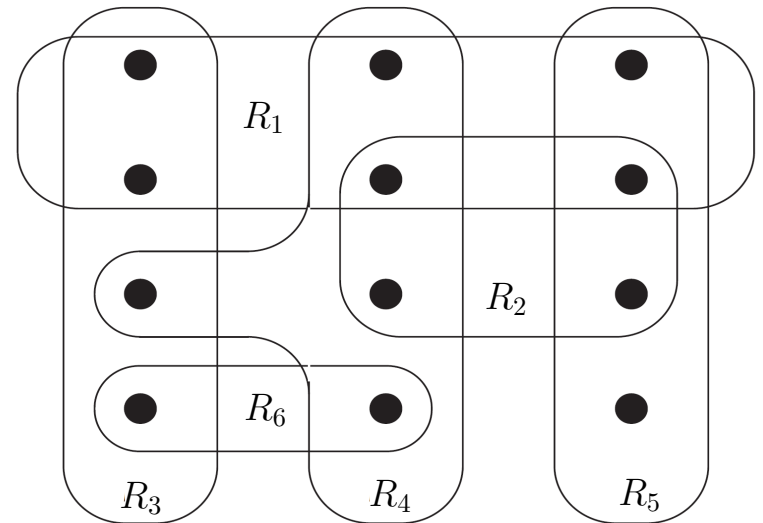
$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

Here,  $S_{<i} := \bigcup_{j=1}^{i-1} S_j$ .

**Exercise:** Run the algorithm on this instance.



# Greedy Algorithm

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

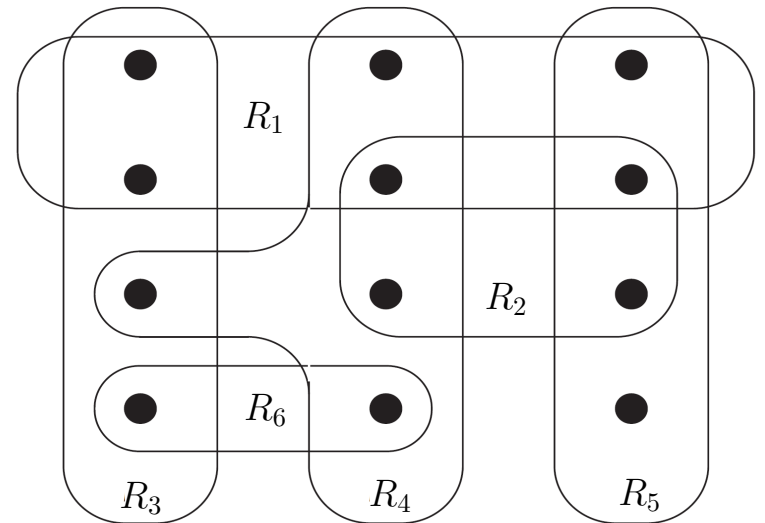
Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

Here,  $S_{<i} := \bigcup_{j=1}^{i-1} S_j$ .

**Exercise:** Run the algorithm on this instance.

$S_1 := R_1$



# Greedy Algorithm

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

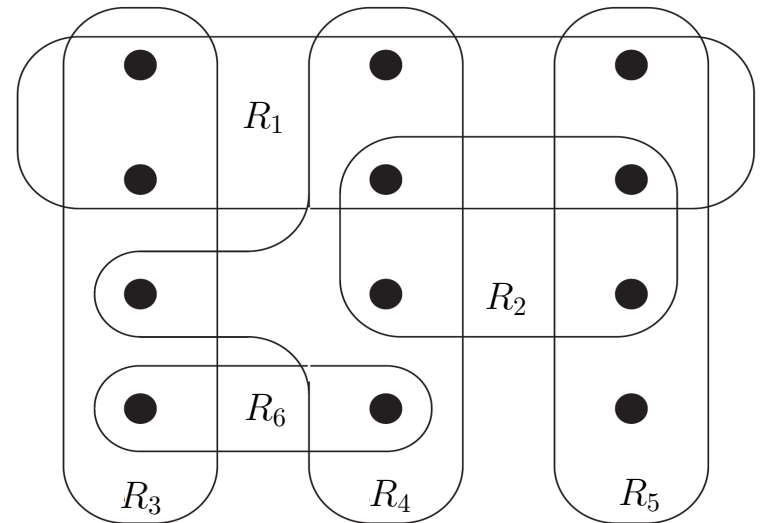
Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

Here,  $S_{<i} := \bigcup_{j=1}^{i-1} S_j$ .

**Exercise:** Run the algorithm on this instance.

$S_1 := R_1$

$S_2 := R_4$



# Greedy Algorithm

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

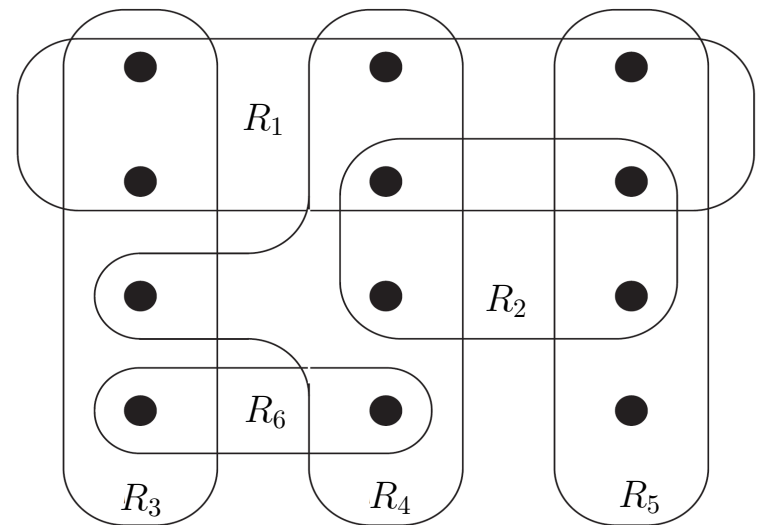
Here,  $S_{<i} := \bigcup_{j=1}^{i-1} S_j$ .

**Exercise:** Run the algorithm on this instance.

$S_1 := R_1$

$S_2 := R_4$

$S_3 := R_5$



# Greedy Algorithm

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

Here,  $S_{<i} := \bigcup_{j=1}^{i-1} S_j$ .

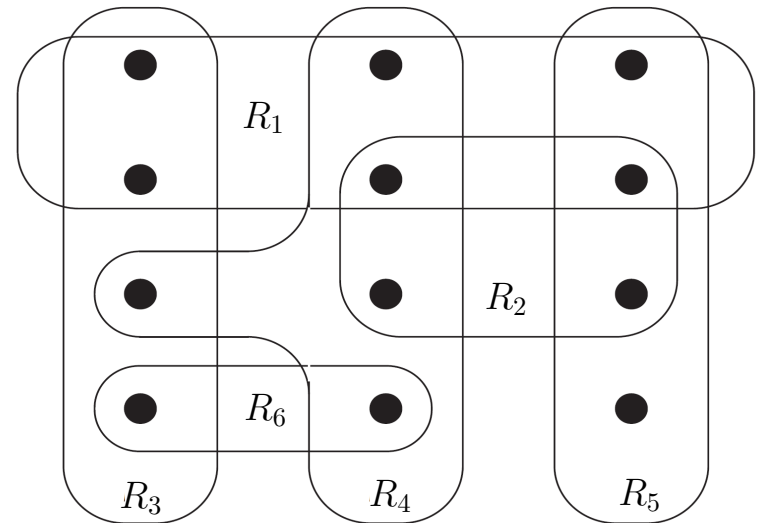
**Exercise:** Run the algorithm on this instance.

$S_1 := R_1$

$S_2 := R_4$

$S_3 := R_5$

$S_4 := R_3$  or  $S_4 := R_6$





# Theorem

**Thm.:** For opt. sol.  $\mathcal{C}^*$ , we have

$$|\mathcal{C}| \leq H_{|X|} \cdot |\mathcal{C}^*|,$$

where

$$H_n := \sum_{i=1}^n 1/i \leq \ln n + 1.$$

Hence, GREEDY-SET-COVER is a  $O(\log n)$ -approx. alg.

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

# Theorem

$$\text{Thm.}: |\mathcal{C}| \leq H_{|X|} \cdot |\mathcal{C}^*|.$$

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

# Theorem

$$\text{Thm.}: |\mathcal{C}| \leq H_{|X|} \cdot |\mathcal{C}^*|.$$

For  $x \in S_i \setminus S_{<i}$ , define  $c_x := \frac{1}{|S_i \setminus S_{<i}|}$ .

For  $Y \subset X$ , define  $c(Y) := \sum_{x \in Y} c_x$ .

GREEDY-SET-COVER( $X, \mathcal{F}$ )

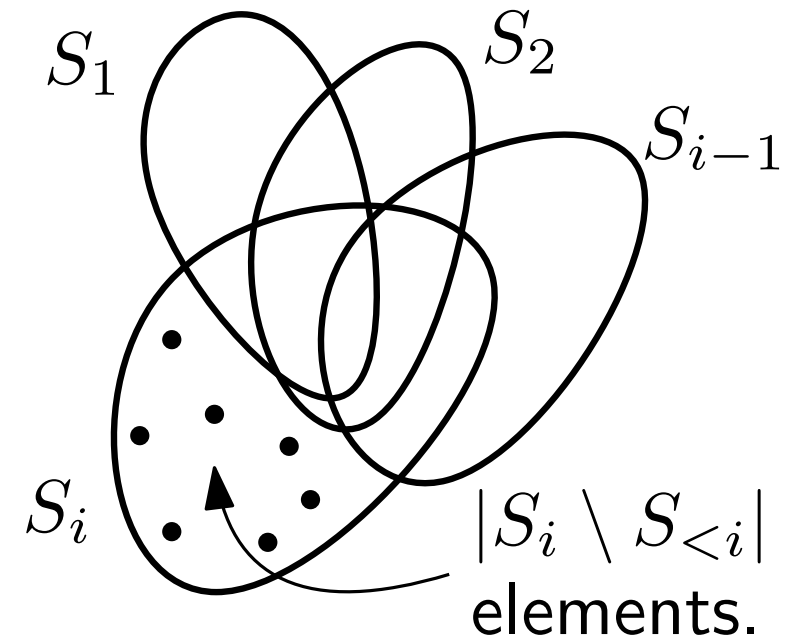
$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$



# Theorem

$$\text{Thm.}: |\mathcal{C}| \leq H_{|X|} \cdot |\mathcal{C}^*|.$$

For  $x \in S_i \setminus S_{<i}$ , define  $c_x := \frac{1}{|S_i \setminus S_{<i}|}$ .

For  $Y \subset X$ , define  $c(Y) := \sum_{x \in Y} c_x$ .

**Observation:**

$$c(X) = \sum_{i=1}^{|\mathcal{C}|} \sum_{x \in S_i \setminus S_{<i}} c_x = \sum_{i=1}^{|\mathcal{C}|} 1 = |\mathcal{C}|.$$

GREEDY-SET-COVER( $X, \mathcal{F}$ )

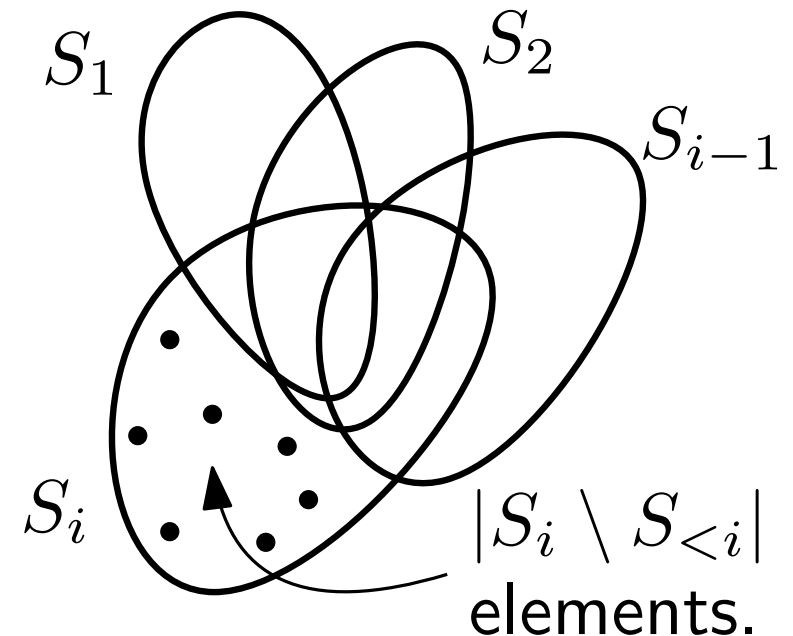
$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$



# Theorem

$$\text{Thm.: } |\mathcal{C}| \leq H_{|X|} \cdot |\mathcal{C}^*|.$$

For  $x \in S_i \setminus S_{<i}$ , define  $c_x := \frac{1}{|S_i \setminus S_{<i}|}$ .

For  $Y \subset X$ , define  $c(Y) := \sum_{x \in Y} c_x$ .

**Observation:**

$$c(X) = \sum_{i=1}^{|\mathcal{C}|} \sum_{x \in S_i \setminus S_{<i}} c_x = \sum_{i=1}^{|\mathcal{C}|} 1 = |\mathcal{C}|.$$

**Lemma:** For all  $S \in \mathcal{F}$ :

$$c(S) \leq \sum_{i=1}^{|S|} \frac{1}{i} = H_{|S|}.$$

GREEDY-SET-COVER( $X, \mathcal{F}$ )

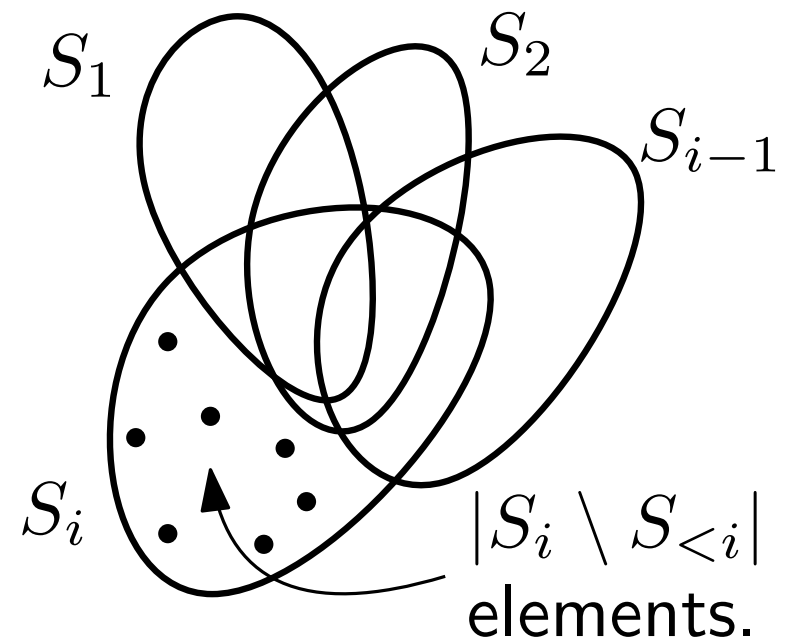
$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$



# Theorem

$$\text{Thm.: } |\mathcal{C}| \leq H_{|X|} \cdot |\mathcal{C}^*|.$$

For  $x \in S_i \setminus S_{<i}$ , define  $c_x := \frac{1}{|S_i \setminus S_{<i}|}$ .

For  $Y \subset X$ , define  $c(Y) := \sum_{x \in Y} c_x$ .

**Observation:**

$$c(X) = \sum_{i=1}^{|\mathcal{C}|} \sum_{x \in S_i \setminus S_{<i}} c_x = \sum_{i=1}^{|\mathcal{C}|} 1 = |\mathcal{C}|.$$

**Lemma:** For all  $S \in \mathcal{F}$ :

$$c(S) \leq \sum_{i=1}^{|S|} \frac{1}{i} = H_{|S|}.$$

*Proof of Thm.:*

$$|\mathcal{C}| = c(X) \leq \sum_{S \in \mathcal{C}^*} c(S) \leq \sum_{S \in \mathcal{C}^*} H_{|S|} \leq \sum_{S \in \mathcal{C}^*} H_{|X|} = |\mathcal{C}^*| \cdot H_{|X|}.$$

GREEDY-SET-COVER( $X, \mathcal{F}$ )

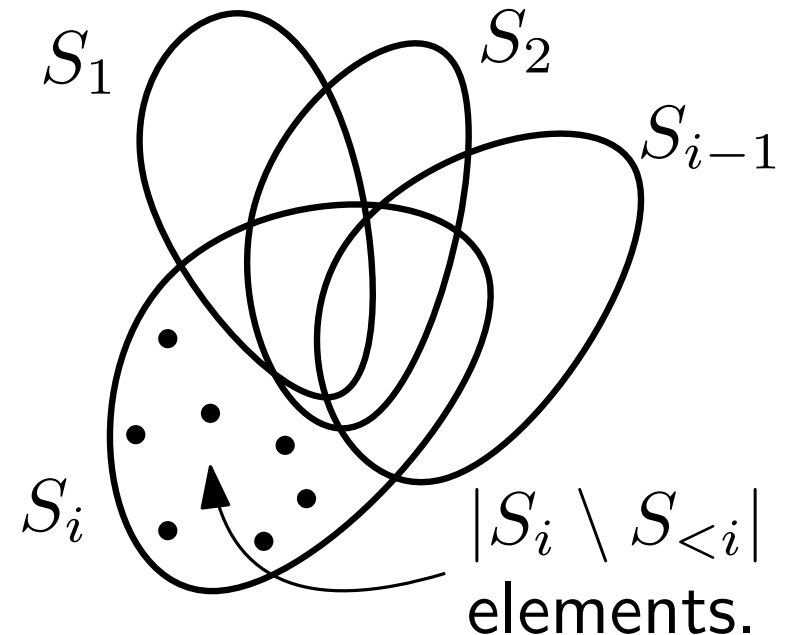
$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$



# Lemma: Idea and Example

**Lemma:** For all  $S \in \mathcal{F}$ :

$$c(S) \leq \sum_{i=1}^{|S|} \frac{1}{i} = H_{|S|}.$$

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

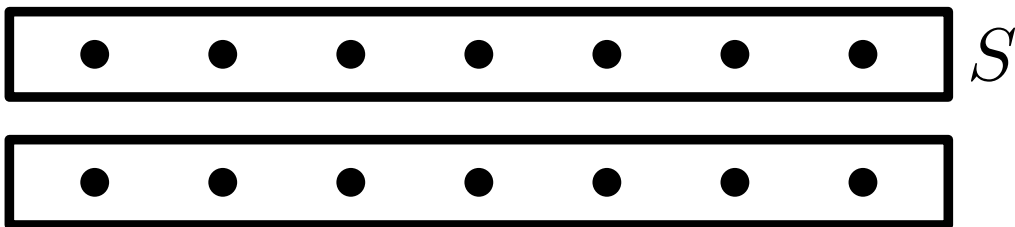
For  $x \in S_i \setminus S_{<i}$ , define  $c_x := \frac{1}{|S_i \setminus S_{<i}|}$ .

For  $Y \subset X$ , define  $c(Y) := \sum_{x \in Y} c_x$ .

**Idea:** 1st element in  $S$  to be covered has  $c_x \leq \frac{1}{|S|}$ , 2nd has  $c_x \leq \frac{1}{|S|-1}$ ,

...

**Example:**



# Lemma: Idea and Example

**Lemma:** For all  $S \in \mathcal{F}$ :

$$c(S) \leq \sum_{i=1}^{|S|} \frac{1}{i} = H_{|S|}.$$

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

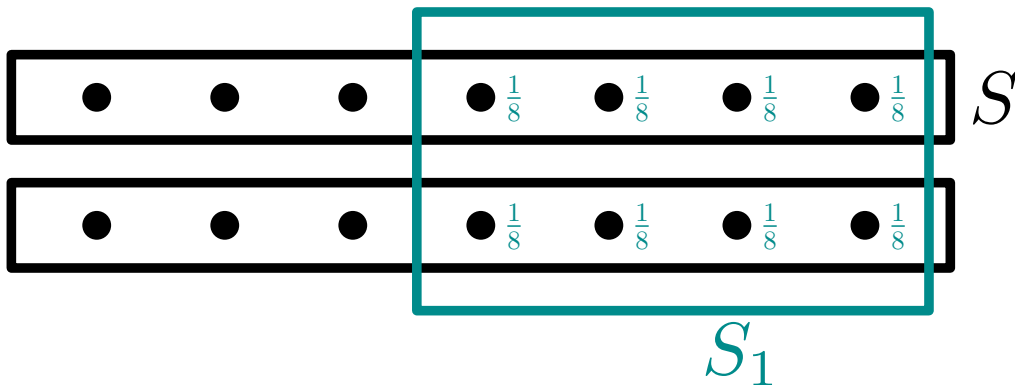
For  $x \in S_i \setminus S_{<i}$ , define  $c_x := \frac{1}{|S_i \setminus S_{<i}|}$ .

For  $Y \subset X$ , define  $c(Y) := \sum_{x \in Y} c_x$ .

**Idea:** 1st element in  $S$  to be covered has  $c_x \leq \frac{1}{|S|}$ , 2nd has  $c_x \leq \frac{1}{|S|-1}$ ,

...

**Example:**





# Lemma: Idea and Example

**Lemma:** For all  $S \in \mathcal{F}$ :

$$c(S) \leq \sum_{i=1}^{|S|} \frac{1}{i} = H_{|S|}.$$

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

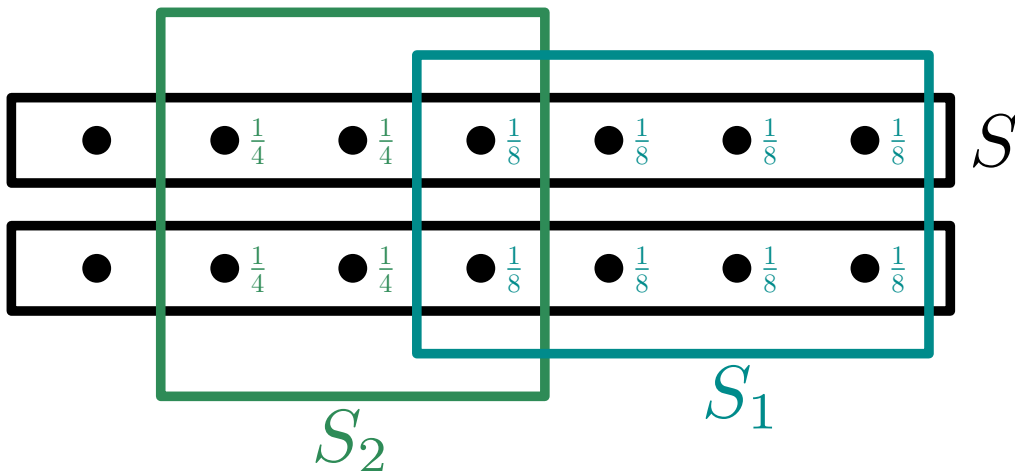
For  $x \in S_i \setminus S_{<i}$ , define  $c_x := \frac{1}{|S_i \setminus S_{<i}|}$ .

For  $Y \subset X$ , define  $c(Y) := \sum_{x \in Y} c_x$ .

**Idea:** 1st element in  $S$  to be covered has  $c_x \leq \frac{1}{|S|}$ , 2nd has  $c_x \leq \frac{1}{|S|-1}$ ,

...

**Example:**



# Lemma: Idea and Example

**Lemma:** For all  $S \in \mathcal{F}$ :

$$c(S) \leq \sum_{i=1}^{|S|} \frac{1}{i} = H_{|S|}.$$

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

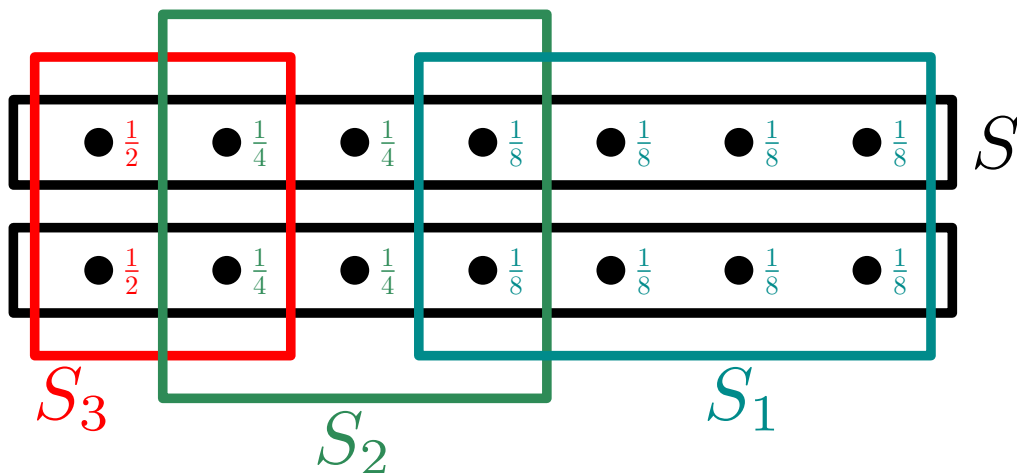
For  $x \in S_i \setminus S_{<i}$ , define  $c_x := \frac{1}{|S_i \setminus S_{<i}|}$ .

For  $Y \subset X$ , define  $c(Y) := \sum_{x \in Y} c_x$ .

**Idea:** 1st element in  $S$  to be covered has  $c_x \leq \frac{1}{|S|}$ , 2nd has  $c_x \leq \frac{1}{|S|-1}$ ,

...

**Example:**



# Lemma: Idea and Example

**Lemma:** For all  $S \in \mathcal{F}$ :

$$c(S) \leq \sum_{i=1}^{|S|} \frac{1}{i} = H_{|S|}.$$

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

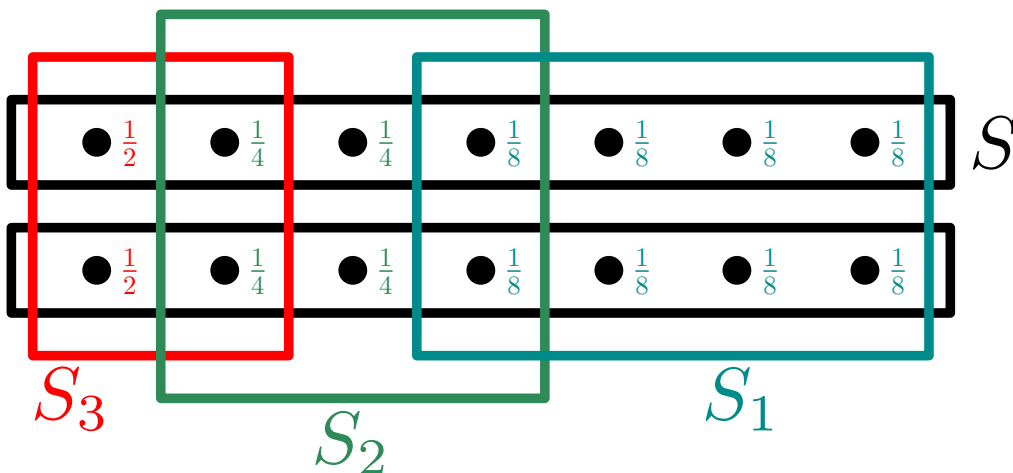
For  $x \in S_i \setminus S_{<i}$ , define  $c_x := \frac{1}{|S_i \setminus S_{<i}|}$ .

For  $Y \subset X$ , define  $c(Y) := \sum_{x \in Y} c_x$ .

**Idea:** 1st element in  $S$  to be covered has  $c_x \leq \frac{1}{|S|}$ , 2nd has  $c_x \leq \frac{1}{|S|-1}$ ,

...

**Example:**



$$\begin{aligned} c(S) &= \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} \\ &\leq 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} \\ &= H_{|S|}. \end{aligned}$$

# Proof of Lemma

**Lemma:** For all  $S \in \mathcal{F}$ :

$$c(S) \leq \sum_{i=1}^{|S|} \frac{1}{i} = H_{|S|}.$$

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

For  $x \in S_i \setminus S_{<i}$ , define  $c_x := \frac{1}{|S_i \setminus S_{<i}|}$ .

For  $Y \subset X$ , define  $c(Y) := \sum_{x \in Y} c_x$ .

**Proof:** Let  $S = \{x_k, x_{k-1}, \dots, x_1\}$ , where  $x_k$  covered first, then  $x_{k-1}$ , etc. (break ties arbitrarily).

# Proof of Lemma

**Lemma:** For all  $S \in \mathcal{F}$ :

$$c(S) \leq \sum_{i=1}^{|S|} \frac{1}{i} = H_{|S|}.$$

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

For  $x \in S_i \setminus S_{<i}$ , define  $c_x := \frac{1}{|S_i \setminus S_{<i}|}$ .

For  $Y \subset X$ , define  $c(Y) := \sum_{x \in Y} c_x$ .

**Proof:** Let  $S = \{x_k, x_{k-1}, \dots, x_1\}$ , where  $x_k$  covered first, then  $x_{k-1}$ , etc. (break ties arbitrarily).

$x_j$  covered first by  $S_i \implies |S \setminus S_{<i}| \geq j$   
(since  $S \setminus S_{<i}$  contains  $x_j, x_{j-1}, \dots, x_1$ )

# Proof of Lemma

**Lemma:** For all  $S \in \mathcal{F}$ :

$$c(S) \leq \sum_{i=1}^{|S|} \frac{1}{i} = H_{|S|}.$$

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

For  $x \in S_i \setminus S_{<i}$ , define  $c_x := \frac{1}{|S_i \setminus S_{<i}|}$ .

For  $Y \subset X$ , define  $c(Y) := \sum_{x \in Y} c_x$ .

**Proof:** Let  $S = \{x_k, x_{k-1}, \dots, x_1\}$ , where  $x_k$  covered first, then  $x_{k-1}$ , etc. (break ties arbitrarily).

$x_j$  covered first by  $S_i \implies |S \setminus S_{<i}| \geq j$   
(since  $S \setminus S_{<i}$  contains  $x_j, x_{j-1}, \dots, x_1$ )

$$|S_i \setminus S_{<i}| \geq |S \setminus S_{<i}| \geq j \implies c_{x_j} = \frac{1}{|S_i \setminus S_{<i}|} \leq \frac{1}{j}.$$

 by greedy choice of  $S_i$

# Proof of Lemma

**Lemma:** For all  $S \in \mathcal{F}$ :

$$c(S) \leq \sum_{i=1}^{|S|} \frac{1}{i} = H_{|S|}.$$

GREEDY-SET-COVER( $X, \mathcal{F}$ )

$i := 0$

while  $X \setminus S_{<i+1} \neq \emptyset$

$i := i + 1$

Pick  $S_i \in \mathcal{F}$  with  $\max |S_i \setminus S_{<i}|$

Return  $\mathcal{C} := \{S_1, \dots, S_i\}$

For  $x \in S_i \setminus S_{<i}$ , define  $c_x := \frac{1}{|S_i \setminus S_{<i}|}$ .

For  $Y \subset X$ , define  $c(Y) := \sum_{x \in Y} c_x$ .

**Proof:** Let  $S = \{x_k, x_{k-1}, \dots, x_1\}$ , where  $x_k$  covered first, then  $x_{k-1}$ , etc. (break ties arbitrarily).

$x_j$  covered first by  $S_i \implies |S \setminus S_{<i}| \geq j$   
(since  $S \setminus S_{<i}$  contains  $x_j, x_{j-1}, \dots, x_1$ )

$$|S_i \setminus S_{<i}| \geq |S \setminus S_{<i}| \geq j \implies c_{x_j} = \frac{1}{|S_i \setminus S_{<i}|} \leq \frac{1}{j}.$$

 by greedy choice of  $S_i$

$$c(S) = c_{x_1} + c_{x_2} + \dots + c_{x_k} \leq 1 + \frac{1}{2} + \dots + \frac{1}{k} = H_{|S|}$$

## Using greedy algorithm for vertex cover

GREEDY-VERTEX-COVER( $G$ )

$C := \emptyset$

while  $E \neq \emptyset$

    Choose  $v \in V$  of maximum degree

$C := C \cup \{u\}$

    Remove edges incident to  $u$  from  $E$

return  $C$



## Using greedy algorithm for vertex cover

GREEDY-VERTEX-COVER( $G$ )

$C := \emptyset$

while  $E \neq \emptyset$

    Choose  $v \in V$  of maximum degree

$C := C \cup \{u\}$

    Remove edges incident to  $u$  from  $E$

return  $C$

**Exercise:** Find graph  $G$  where GREEDY-VERTEX-COVER does not produce optimal solution.

## Using greedy algorithm for vertex cover

GREEDY-VERTEX-COVER( $G$ )

$C := \emptyset$

while  $E \neq \emptyset$

    Choose  $v \in V$  of maximum degree

$C := C \cup \{u\}$

    Remove edges incident to  $u$  from  $E$

return  $C$

**Exercise:** Find graph  $G$  where GREEDY-VERTEX-COVER does not produce optimal solution.

The algorithm only gives a  $\Theta(\log |E|)$ -approximation.

# Introduction to Approximation Algorithms, part II

13-1 2025, Srikanth Srinivasan,  
DIKU (Slides: Mikkel Abrahamsen)



APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return last( $L'_n$ )

# Definition

**Def.:** An algorithm for an optimization problem has *approximation ratio*  $\rho(n)$  if for every input of size  $n$ ,

$$\max \left\{ \frac{C}{C^*}, \frac{C^*}{C} \right\} \leq \rho(n).$$

## Definition

**Def.:** An algorithm for an optimization problem has *approximation ratio*  $\rho(n)$  if for every input of size  $n$ ,

$$\max \left\{ \frac{C}{C^*}, \frac{C^*}{C} \right\} \leq \rho(n).$$

$C^* := \text{cost}(\text{opt. sol.})$    $C := \text{cost}(\text{produced sol.})$  

# Definition

**Def.:** An algorithm for an optimization problem has *approximation ratio*  $\rho(n)$  if for every input of size  $n$ ,

$$\max \left\{ \frac{C}{C^*}, \frac{C^*}{C} \right\} \leq \rho(n).$$

$C^* := \text{cost}(\text{opt. sol.})$

$C := \text{cost}(\text{produced sol.})$

minimization problem

maximization problem

# Definition

**Def.:** An algorithm for an optimization problem has *approximation ratio*  $\rho(n)$  if for every input of size  $n$ ,

$$\max \left\{ \frac{C}{C^*}, \frac{C^*}{C} \right\} \leq \rho(n).$$

$C^* := \text{cost}(\text{opt. sol.})$   $C := \text{cost}(\text{produced sol.})$

minimization problem

maximization problem

**Today:** Examples of use of randomization, linear programming, and a fully polynomial time approximation scheme (FPTAS).

# Definition

**Def.:** An algorithm for an optimization problem has *approximation ratio*  $\rho(n)$  if for every input of size  $n$ ,

$$\max \left\{ \frac{C}{C^*}, \frac{C^*}{C} \right\} \leq \rho(n).$$

$C^* := \text{cost}(\text{opt. sol.})$

$C := \text{cost}(\text{produced sol.})$

minimization problem

maximization problem

**Today:** Examples of use of randomization, linear programming, and a fully polynomial time approximation scheme (FPTAS).

$C := \mathbf{E} [\text{cost}(\text{produced sol.})]$



# 3-SAT

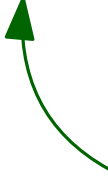
$$\begin{aligned} & (x_1 \vee x_7 \vee \neg x_9) \\ & \wedge (\neg x_7 \vee x_8 \vee x_9) \\ & \wedge (\neg x_2 \vee x_3 \vee \neg x_4) \\ & \vdots \end{aligned}$$

# 3-SAT

$$\left. \begin{array}{l} (x_1 \vee x_7 \vee \neg x_9) \\ \wedge (\neg x_7 \vee x_8 \vee x_9) \\ \wedge (\neg x_2 \vee x_3 \vee \neg x_4) \\ \vdots \end{array} \right] n \text{ clauses}$$

# 3-SAT

Each clause has three *literals* involving three distinct variables.


$$\begin{array}{l} (x_1 \vee x_7 \vee \neg x_9) \\ \wedge (\neg x_7 \vee x_8 \vee x_9) \\ \wedge (\neg x_2 \vee x_3 \vee \neg x_4) \\ \vdots \end{array} \quad \left. \vphantom{\begin{array}{l} (x_1 \vee x_7 \vee \neg x_9) \\ \wedge (\neg x_7 \vee x_8 \vee x_9) \\ \wedge (\neg x_2 \vee x_3 \vee \neg x_4) \\ \vdots \end{array}} \right] n \text{ clauses}$$

# 3-SAT

Each clause has three *literals*  
involving three distinct variables.

Decision version:  
NP-complete!


$$(x_1 \vee x_7 \vee \neg x_9)$$

$$\wedge (\neg x_7 \vee x_8 \vee x_9)$$

$$\wedge (\neg x_2 \vee x_3 \vee \neg x_4)$$

$$\vdots$$

$n$  clauses

# 3-SAT


Each clause has three *literals* involving three distinct variables.

Decision version:  
NP-complete!

$$\left[ \begin{array}{l} (x_1 \vee x_7 \vee \neg x_9) \\ \wedge (\neg x_7 \vee x_8 \vee x_9) \\ \wedge (\neg x_2 \vee x_3 \vee \neg x_4) \\ \vdots \end{array} \right] n \text{ clauses}$$

MAX-3-SAT: Find assignment that maximizes the number of true clauses.

# Randomly assigning values

  $\Phi$  is a MAX-3-SAT instance

RANDOM-ASSIGNMENT( $\Phi$ )  
  for each variable  $x_i$  of  $\Phi$   
    choose  $x_i \in \{0, 1\}$  by flipping fair coin  
  return assignment

# Randomly assigning values

RANDOM-ASSIGNMENT( $\Phi$ )  $\Phi$  is a MAX-3-SAT instance

for each variable  $x_i$  of  $\Phi$

choose  $x_i \in \{0, 1\}$  by flipping fair coin

return assignment

**Thm.:** RANDOM-ASSIGNMENT is a  $8/7$ -approximation algorithm.

# Randomly assigning values

  $\Phi$  is a MAX-3-SAT instance

RANDOM-ASSIGNMENT( $\Phi$ )

  for each variable  $x_i$  of  $\Phi$

    choose  $x_i \in \{0, 1\}$  by flipping fair coin

  return assignment

**Thm.:** RANDOM-ASSIGNMENT is a  $8/7$ -approximation algorithm.

*Proof:* Let  $\Phi = C_1 \wedge \dots \wedge C_n$ . Consider  $C_i = \ell_1 \vee \ell_2 \vee \ell_3$ .



# Randomly assigning values

  $\Phi$  is a MAX-3-SAT instance

RANDOM-ASSIGNMENT( $\Phi$ )

  for each variable  $x_i$  of  $\Phi$

    choose  $x_i \in \{0, 1\}$  by flipping fair coin

  return assignment

**Thm.:** RANDOM-ASSIGNMENT is a  $8/7$ -approximation algorithm.

*Proof:* Let  $\Phi = C_1 \wedge \dots \wedge C_n$ . Consider  $C_i = \ell_1 \vee \ell_2 \vee \ell_3$ .

Clause  $C_i$  not satisfied  $\iff$

# Randomly assigning values

  $\Phi$  is a MAX-3-SAT instance

RANDOM-ASSIGNMENT( $\Phi$ )

for each variable  $x_i$  of  $\Phi$

choose  $x_i \in \{0, 1\}$  by flipping fair coin

return assignment

**Thm.:** RANDOM-ASSIGNMENT is a  $8/7$ -approximation algorithm.

*Proof:* Let  $\Phi = C_1 \wedge \dots \wedge C_n$ . Consider  $C_i = \ell_1 \vee \ell_2 \vee \ell_3$ .

Clause  $C_i$  not satisfied  $\iff \neg \ell_1 \wedge \neg \ell_2 \wedge \neg \ell_3$ .

# Randomly assigning values

  $\Phi$  is a MAX-3-SAT instance

RANDOM-ASSIGNMENT( $\Phi$ )

for each variable  $x_i$  of  $\Phi$

choose  $x_i \in \{0, 1\}$  by flipping fair coin

return assignment

**Thm.:** RANDOM-ASSIGNMENT is a  $8/7$ -approximation algorithm.

*Proof:* Let  $\Phi = C_1 \wedge \dots \wedge C_n$ . Consider  $C_i = \ell_1 \vee \ell_2 \vee \ell_3$ .

Clause  $C_i$  not satisfied  $\iff \neg \ell_1 \wedge \neg \ell_2 \wedge \neg \ell_3$ .

$$\mathbf{Pr} [\neg C_i] = \mathbf{Pr} [\neg \ell_1] \cdot \mathbf{Pr} [\neg \ell_2] \cdot \mathbf{Pr} [\neg \ell_3]$$

# Randomly assigning values

RANDOM-ASSIGNMENT( $\Phi$ )  $\Phi$  is a MAX-3-SAT instance

for each variable  $x_i$  of  $\Phi$

choose  $x_i \in \{0, 1\}$  by flipping fair coin


return assignment

**Thm.:** RANDOM-ASSIGNMENT is a  $8/7$ -approximation algorithm.

*Proof:* Let  $\Phi = C_1 \wedge \dots \wedge C_n$ . Consider  $C_i = \ell_1 \vee \ell_2 \vee \ell_3$ .

Clause  $C_i$  not satisfied  $\iff \neg \ell_1 \wedge \neg \ell_2 \wedge \neg \ell_3$ .

$$\Pr[\neg C_i] = \Pr[\neg \ell_1] \cdot \Pr[\neg \ell_2] \cdot \Pr[\neg \ell_3]$$

variables in  $C_i$  chosen independently

# Randomly assigning values

RANDOM-ASSIGNMENT( $\Phi$ )  $\Phi$  is a MAX-3-SAT instance

for each variable  $x_i$  of  $\Phi$

choose  $x_i \in \{0, 1\}$  by flipping fair coin


return assignment

**Thm.:** RANDOM-ASSIGNMENT is a  $8/7$ -approximation algorithm.

*Proof:* Let  $\Phi = C_1 \wedge \dots \wedge C_n$ . Consider  $C_i = \ell_1 \vee \ell_2 \vee \ell_3$ .

Clause  $C_i$  not satisfied  $\iff \neg \ell_1 \wedge \neg \ell_2 \wedge \neg \ell_3$ .

$$\Pr[\neg C_i] = \Pr[\neg \ell_1] \cdot \Pr[\neg \ell_2] \cdot \Pr[\neg \ell_3] = \left(\frac{1}{2}\right)^3 = 1/8$$

variables in  $C_i$  chosen independently

# Randomly assigning values

  $\Phi$  is a MAX-3-SAT instance

RANDOM-ASSIGNMENT( $\Phi$ )

for each variable  $x_i$  of  $\Phi$

choose  $x_i \in \{0, 1\}$  by flipping fair coin


return assignment

**Thm.:** RANDOM-ASSIGNMENT is a  $8/7$ -approximation algorithm.

*Proof:* Let  $\Phi = C_1 \wedge \dots \wedge C_n$ . Consider  $C_i = \ell_1 \vee \ell_2 \vee \ell_3$ .

Clause  $C_i$  not satisfied  $\iff \neg \ell_1 \wedge \neg \ell_2 \wedge \neg \ell_3$ .

$$\Pr[\neg C_i] = \Pr[\neg \ell_1] \cdot \Pr[\neg \ell_2] \cdot \Pr[\neg \ell_3] = \left(\frac{1}{2}\right)^3 = 1/8$$

 variables in  $C_i$  chosen independently

$$\Pr[C_i] = 1 - \Pr[\neg C_i] = 1 - 1/8 = 7/8$$

# Randomly assigning values

RANDOM-ASSIGNMENT( $\Phi$ )   $\Phi$  is a MAX-3-SAT instance

for each variable  $x_i$  of  $\Phi$

choose  $x_i \in \{0, 1\}$  by flipping fair coin


return assignment

**Thm.:** RANDOM-ASSIGNMENT is a  $8/7$ -approximation algorithm.

*Proof:* Let  $\Phi = C_1 \wedge \dots \wedge C_n$ . Consider  $C_i = \ell_1 \vee \ell_2 \vee \ell_3$ .

Clause  $C_i$  not satisfied  $\iff \neg \ell_1 \wedge \neg \ell_2 \wedge \neg \ell_3$ .

$$\mathbf{Pr}[\neg C_i] = \mathbf{Pr}[\neg \ell_1] \cdot \mathbf{Pr}[\neg \ell_2] \cdot \mathbf{Pr}[\neg \ell_3] = \left(\frac{1}{2}\right)^3 = 1/8$$

 variables in  $C_i$  chosen independently

$$\mathbf{Pr}[C_i] = 1 - \mathbf{Pr}[\neg C_i] = 1 - 1/8 = 7/8$$

$$X := \sum_{i=1}^n [C_i] = \# \text{satisfied clauses}$$

# Randomly assigning values

RANDOM-ASSIGNMENT( $\Phi$ )

for each variable  $x_i$  of  $\Phi$

choose  $x_i \in \{0, 1\}$  by flipping fair coin

return assignment

**Thm.:** RANDOM-ASSIGNMENT is a  $8/7$ -approximation algorithm.

*Proof:* Let  $\Phi = C_1 \wedge \dots \wedge C_n$ . Consider  $C_i = \ell_1 \vee \ell_2 \vee \ell_3$ .

Clause  $C_i$  not satisfied  $\iff \neg \ell_1 \wedge \neg \ell_2 \wedge \neg \ell_3$ .

$$\Pr[\neg C_i] = \Pr[\neg \ell_1] \cdot \Pr[\neg \ell_2] \cdot \Pr[\neg \ell_3] = \left(\frac{1}{2}\right)^3 = 1/8$$

variables in  $C_i$  chosen independently

$$\Pr[C_i] = 1 - \Pr[\neg C_i] = 1 - 1/8 = 7/8$$

$X := \sum_{i=1}^n [C_i] = \# \text{satisfied clauses}$

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^n [C_i]\right] = \sum_{i=1}^n \mathbf{E}[C_i] = \sum_{i=1}^n \frac{7}{8} = 7n/8$$

Linearity of  
expectation



# Randomly assigning values

$\Phi$  is a MAX-3-SAT instance  
RANDOM-ASSIGNMENT( $\Phi$ )

for each variable  $x_i$  of  $\Phi$

choose  $x_i \in \{0, 1\}$  by flipping fair coin

return assignment

**Thm.:** RANDOM-ASSIGNMENT is a  $8/7$ -approximation algorithm.

*Proof:* Let  $\Phi = C_1 \wedge \dots \wedge C_n$ . Consider  $C_i = \ell_1 \vee \ell_2 \vee \ell_3$ .

Clause  $C_i$  not satisfied  $\iff \neg \ell_1 \wedge \neg \ell_2 \wedge \neg \ell_3$ .

$$\Pr[\neg C_i] = \Pr[\neg \ell_1] \cdot \Pr[\neg \ell_2] \cdot \Pr[\neg \ell_3] = \left(\frac{1}{2}\right)^3 = 1/8$$

variables in  $C_i$  chosen independently

$$\Pr[C_i] = 1 - \Pr[\neg C_i] = 1 - 1/8 = 7/8$$

$X := \sum_{i=1}^n [C_i] = \# \text{satisfied clauses}$

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^n [C_i]\right] = \sum_{i=1}^n \mathbf{E}[C_i] = \sum_{i=1}^n \frac{7}{8} = 7n/8$$

Linearity of expectation

$$\text{Approximation ratio: } \frac{C^*}{C} = \frac{C^*}{7n/8} \leq$$

# Randomly assigning values

$\Phi$  is a MAX-3-SAT instance  
RANDOM-ASSIGNMENT( $\Phi$ )

for each variable  $x_i$  of  $\Phi$

choose  $x_i \in \{0, 1\}$  by flipping fair coin

return assignment

**Thm.:** RANDOM-ASSIGNMENT is a  $8/7$ -approximation algorithm.

*Proof:* Let  $\Phi = C_1 \wedge \dots \wedge C_n$ . Consider  $C_i = \ell_1 \vee \ell_2 \vee \ell_3$ .

Clause  $C_i$  not satisfied  $\iff \neg \ell_1 \wedge \neg \ell_2 \wedge \neg \ell_3$ .

$$\Pr[\neg C_i] = \Pr[\neg \ell_1] \cdot \Pr[\neg \ell_2] \cdot \Pr[\neg \ell_3] = \left(\frac{1}{2}\right)^3 = 1/8$$

variables in  $C_i$  chosen independently

$$\Pr[C_i] = 1 - \Pr[\neg C_i] = 1 - 1/8 = 7/8$$

$X := \sum_{i=1}^n [C_i] = \# \text{satisfied clauses}$

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^n [C_i]\right] = \sum_{i=1}^n \mathbf{E}[C_i] = \sum_{i=1}^n \frac{7}{8} = 7n/8$$

Linearity of expectation

$$\text{Approximation ratio: } \frac{C^*}{C} = \frac{C^*}{7n/8} \leq \frac{n}{7n/8} = 8/7$$

# Bonus: Derandomization

**Goal:** Find deterministic alg. that satisfies  $7n/8$  clauses.

## Bonus: Derandomization

**Goal:** Find deterministic alg. that satisfies  $7n/8$  clauses.

**Recall:**  $\mathbf{E}[X] = 7n/8$ , where  $X := \sum_{i=1}^n [C_i] = \# \text{satisfied clauses}$

## Bonus: Derandomization

**Goal:** Find deterministic alg. that satisfies  $7n/8$  clauses.

**Recall:**  $\mathbf{E}[X] = 7n/8$ , where  $X := \sum_{i=1}^n [C_i] = \# \text{satisfied clauses}$

Assignment guaranteed to exist by *the probabilistic method*.

## Bonus: Derandomization

**Goal:** Find deterministic alg. that satisfies  $7n/8$  clauses.

**Recall:**  $\mathbf{E}[X] = 7n/8$ , where  $X := \sum_{i=1}^n [C_i] = \# \text{satisfied clauses}$

Assignment guaranteed to exist by *the probabilistic method*.

Clause  $C$  with 3 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^3 = 7/8$ .

## Bonus: Derandomization

**Goal:** Find deterministic alg. that satisfies  $7n/8$  clauses.

**Recall:**  $\mathbf{E}[X] = 7n/8$ , where  $X := \sum_{i=1}^n [C_i] = \# \text{satisfied clauses}$

Assignment guaranteed to exist by *the probabilistic method*.

Clause  $C$  with 3 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^3 = 7/8$ .

Clause  $C$  with 2 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^2 = 3/4$ .

## Bonus: Derandomization

**Goal:** Find deterministic alg. that satisfies  $7n/8$  clauses.

**Recall:**  $\mathbf{E}[X] = 7n/8$ , where  $X := \sum_{i=1}^n [C_i] = \# \text{satisfied clauses}$

Assignment guaranteed to exist by *the probabilistic method*.

Clause  $C$  with 3 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^3 = 7/8$ .

Clause  $C$  with 2 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^2 = 3/4$ .

Clause  $C$  with 1 literal:  $\mathbf{Pr}[C] = 1 - \frac{1}{2} = 1/2$ .



## Bonus: Derandomization

**Goal:** Find deterministic alg. that satisfies  $7n/8$  clauses.

**Recall:**  $\mathbf{E}[X] = 7n/8$ , where  $X := \sum_{i=1}^n [C_i] = \# \text{satisfied clauses}$

Assignment guaranteed to exist by *the probabilistic method*.

Clause  $C$  with 3 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^3 = 7/8$ .

Clause  $C$  with 2 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^2 = 3/4$ .

Clause  $C$  with 1 literal:  $\mathbf{Pr}[C] = 1 - \frac{1}{2} = 1/2$ .

Clause  $C$  with 0 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^0 = 0$ .

## Bonus: Derandomization

**Goal:** Find deterministic alg. that satisfies  $7n/8$  clauses.

**Recall:**  $\mathbf{E}[X] = 7n/8$ , where  $X := \sum_{i=1}^n [C_i] = \# \text{satisfied clauses}$

Assignment guaranteed to exist by *the probabilistic method*.

Clause  $C$  with 3 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^3 = 7/8$ .

Clause  $C$  with 2 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^2 = 3/4$ .

Clause  $C$  with 1 literal:  $\mathbf{Pr}[C] = 1 - \frac{1}{2} = 1/2$ .

Clause  $C$  with 0 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^0 = 0$ .

DETERMINISTIC-ASSIGNMENT( $\Phi$ )

for  $i = 1, \dots, \boxed{m}$    $m = \# \text{variables in } \Phi$

$x_i := 0$

    compute  $D := \mathbf{E}[X \mid \text{chosen values of } x_1, \dots, x_i]$

    if  $D < 7n/8$

$x_i := 1$

return assignment

## Bonus: Derandomization

**Goal:** Find deterministic alg. that satisfies  $7n/8$  clauses.

**Recall:**  $\mathbf{E}[X] = 7n/8$ , where  $X := \sum_{i=1}^n [C_i] = \# \text{satisfied clauses}$

Assignment guaranteed to exist by *the probabilistic method*.

Clause  $C$  with 3 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^3 = 7/8$ .

Clause  $C$  with 2 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^2 = 3/4$ .

Clause  $C$  with 1 literal:  $\mathbf{Pr}[C] = 1 - \frac{1}{2} = 1/2$ .

Clause  $C$  with 0 literals:  $\mathbf{Pr}[C] = 1 - (\frac{1}{2})^0 = 0$ .

DETERMINISTIC-ASSIGNMENT( $\Phi$ )

for  $i = 1, \dots, m$    $m = \# \text{variables in } \Phi$

$x_i := 0$

    compute  $D := \mathbf{E}[X \mid \text{chosen values of } x_1, \dots, x_i]$

    if  $D < 7n/8$

$x_i := 1$

return assignment

Method of conditional  
probabilities

# Example

DETERMINISTIC-ASSIGNMENT( $\Phi$ )

for  $i = 1, \dots, m$

$x_i := 0$

compute  $D := \mathbf{E}[X \mid \text{chosen values of } x_1, \dots, x_i]$

if  $D < 7n/8$

$x_i := 1$

return assignment

Method of conditional probabilities

$$\Phi = (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee \neg x_4 \vee x_5) \wedge (x_1 \vee \neg x_5 \vee x_6) \wedge \dots$$

# Example

DETERMINISTIC-ASSIGNMENT( $\Phi$ )

for  $i = 1, \dots, m$

$x_i := 0$

compute  $D := \mathbf{E}[X \mid \text{chosen values of } x_1, \dots, x_i]$

if  $D < 7n/8$

$x_i := 1$

return assignment

Method of conditional probabilities

$$\Phi = \overbrace{(\neg x_1 \vee \neg x_2 \vee x_4)}^{7/8} \wedge \overbrace{(x_1 \vee \neg x_4 \vee x_5)}^{7/8} \wedge \overbrace{(x_1 \vee \neg x_5 \vee x_6)}^{7/8} \wedge \dots$$

# Example

DETERMINISTIC-ASSIGNMENT( $\Phi$ )

for  $i = 1, \dots, m$

$x_i := 0$

compute  $D := \mathbf{E}[X \mid \text{chosen values of } x_1, \dots, x_i]$

if  $D < 7n/8$

$x_i := 1$

return assignment

Method of conditional probabilities

$$\Phi = \overbrace{(\neg x_1 \vee \neg x_2 \vee x_4)}^{7/8} \wedge \overbrace{(x_1 \vee \neg x_4 \vee x_5)}^{7/8} \wedge \overbrace{(x_1 \vee \neg x_5 \vee x_6)}^{7/8} \wedge \dots$$

$x_1 := 0$ :

$$\Phi = (1 \vee \neg x_2 \vee x_4) \wedge (0 \vee \neg x_4 \vee x_5) \wedge (0 \vee \neg x_5 \vee x_6) \wedge \dots$$

# Example

DETERMINISTIC-ASSIGNMENT( $\Phi$ )

for  $i = 1, \dots, m$

$x_i := 0$

compute  $D := \mathbf{E}[X \mid \text{chosen values of } x_1, \dots, x_i]$

if  $D < 7n/8$

$x_i := 1$

return assignment

Method of conditional probabilities

$$\Phi = \overbrace{(\neg x_1 \vee \neg x_2 \vee x_4)}^{7/8} \wedge \overbrace{(x_1 \vee \neg x_4 \vee x_5)}^{7/8} \wedge \overbrace{(x_1 \vee \neg x_5 \vee x_6)}^{7/8} \wedge \dots$$

$x_1 := 0$ :

$$\Phi = \overbrace{(1 \vee \neg x_2 \vee x_4)}^1 \wedge \overbrace{(0 \vee \neg x_4 \vee x_5)}^{3/4} \wedge \overbrace{(0 \vee \neg x_5 \vee x_6)}^{3/4} \wedge \dots$$

# Example

DETERMINISTIC-ASSIGNMENT( $\Phi$ )

for  $i = 1, \dots, m$

$x_i := 0$

compute  $D := \mathbf{E}[X \mid \text{chosen values of } x_1, \dots, x_i]$

if  $D < 7n/8$

$x_i := 1$

return assignment

Method of conditional probabilities

$$\Phi = \overbrace{(\neg x_1 \vee \neg x_2 \vee x_4)}^{7/8} \wedge \overbrace{(x_1 \vee \neg x_4 \vee x_5)}^{7/8} \wedge \overbrace{(x_1 \vee \neg x_5 \vee x_6)}^{7/8} \wedge \dots$$

$x_1 := 0$ :

$$\Phi = \overbrace{(1 \vee \neg x_2 \vee x_4)}^1 \wedge \overbrace{(0 \vee \neg x_4 \vee x_5)}^{3/4} \wedge \overbrace{(0 \vee \neg x_5 \vee x_6)}^{3/4} \wedge \dots$$

$x_1 := 1$

$$\Phi = (0 \vee \neg x_2 \vee x_4) \wedge (1 \vee \neg x_4 \vee x_5) \wedge (1 \vee \neg x_5 \vee x_6) \wedge \dots$$



# Example

DETERMINISTIC-ASSIGNMENT( $\Phi$ )

for  $i = 1, \dots, m$

$x_i := 0$

compute  $D := \mathbf{E}[X \mid \text{chosen values of } x_1, \dots, x_i]$

if  $D < 7n/8$

$x_i := 1$

return assignment

Method of conditional probabilities

$$\Phi = \overbrace{(\neg x_1 \vee \neg x_2 \vee x_4)}^{7/8} \wedge \overbrace{(x_1 \vee \neg x_4 \vee x_5)}^{7/8} \wedge \overbrace{(x_1 \vee \neg x_5 \vee x_6)}^{7/8} \wedge \dots$$

$x_1 := 0:$

$$\Phi = \overbrace{(1 \vee \neg x_2 \vee x_4)}^1 \wedge \overbrace{(0 \vee \neg x_4 \vee x_5)}^{3/4} \wedge \overbrace{(0 \vee \neg x_5 \vee x_6)}^{3/4} \wedge \dots$$

$x_1 := 1$

$$\Phi = \overbrace{(0 \vee \neg x_2 \vee x_4)}^{3/4} \wedge \overbrace{(1 \vee \neg x_4 \vee x_5)}^1 \wedge \overbrace{(1 \vee \neg x_5 \vee x_6)}^1 \wedge \dots$$

# Example

DETERMINISTIC-ASSIGNMENT( $\Phi$ )

for  $i = 1, \dots, m$

$x_i := 0$

compute  $D := \mathbf{E}[X \mid \text{chosen values of } x_1, \dots, x_i]$

if  $D < 7n/8$

$x_i := 1$

return assignment

Method of conditional probabilities

$$\Phi = \overbrace{(\neg x_1 \vee \neg x_2 \vee x_4)}^{7/8} \wedge \overbrace{(x_1 \vee \neg x_4 \vee x_5)}^{7/8} \wedge \overbrace{(x_1 \vee \neg x_5 \vee x_6)}^{7/8} \wedge \dots$$

$$x_1 := 0: \quad \Phi = \overbrace{(1 \vee \neg x_2 \vee x_4)}^1 \wedge \overbrace{(0 \vee \neg x_4 \vee x_5)}^{3/4} \wedge \overbrace{(0 \vee \neg x_5 \vee x_6)}^{3/4} \wedge \dots$$

$$x_1 := 1 \quad \Phi = \overbrace{(0 \vee \neg x_2 \vee x_4)}^{3/4} \wedge \overbrace{(1 \vee \neg x_4 \vee x_5)}^1 \wedge \overbrace{(1 \vee \neg x_5 \vee x_6)}^1 \wedge \dots$$

$$x_2 := 0 \quad \Phi = \overbrace{(0 \vee 1 \vee x_4)}^1 \wedge \overbrace{(1 \vee \neg x_4 \vee x_5)}^1 \wedge \overbrace{(1 \vee \neg x_5 \vee x_6)}^1 \wedge \dots$$

## Bonus info

DETERMINISTIC-ASSIGNMENT( $\Phi$ )

for  $i = 1, \dots, m$

$x_i := 0$

compute  $D := \mathbf{E}[X \mid \text{chosen values of } x_1, \dots, x_i]$

if  $D < 7n/8$

$x_i := 1$

return assignment

By work of Håstad, it is NP-hard to approximate within  $8/7 - \varepsilon$  for all  $\varepsilon > 0$ , so this very simple algorithm is essentially optimal, unless  $P=NP$ .

# Vertex Cover

**Def.:** Let  $G = (V, E)$  be a graph. A set  $V' \subseteq V$  of vertices is a *vertex cover* if for all  $uv \in E$ , we have  $u \in V'$  or  $v \in V'$ .

# Vertex Cover

**Def.:** Let  $G = (V, E)$  be a graph. A set  $V' \subseteq V$  of vertices is a *vertex cover* if for all  $uv \in E$ , we have  $u \in V'$  or  $v \in V'$ .

NP-hard!

# Vertex Cover

**Def.:** Let  $G = (V, E)$  be a graph. A set  $V' \subseteq V$  of vertices is a *vertex cover* if for all  $uv \in E$ , we have  $u \in V'$  or  $v \in V'$ .

NP-hard!

APPROX-VERTEX-COVER( $G$ )

$C := \emptyset$

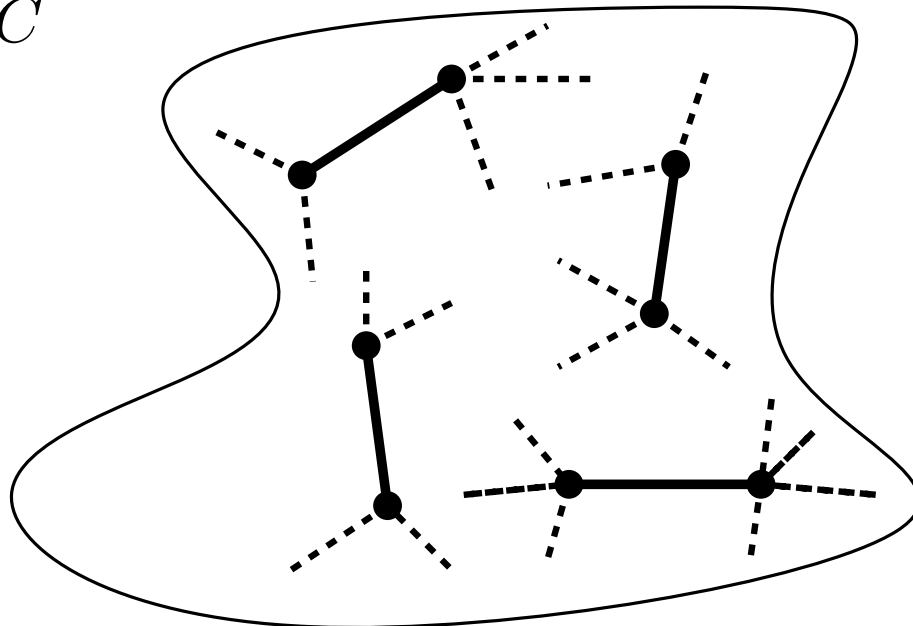
while  $E(G) \neq \emptyset$

  choose  $uv \in E(G)$

$C := C \cup \{u, v\}$

  remove all edges incident on  $u$  or  $v$  from  $E(G)$

return  $C$



$$\frac{C}{C^*} \leq 2$$

# Weighted Vertex Cover

**Def.:** Let  $G = (V, E)$  be a graph. A set  $V' \subseteq V$  of vertices is a *vertex cover* if for all  $uv \in E$ , we have  $u \in V'$  or  $v \in V'$ .

# Weighted Vertex Cover

**Def.:** Let  $G = (V, E)$  be a graph. A set  $V' \subseteq V$  of vertices is a *vertex cover* if for all  $uv \in E$ , we have  $u \in V'$  or  $v \in V'$ .

**Now:** We are given weight  $w(v) > 0$  for each  $v \in V$ .

**Goal:** Find vertex cover  $C$  with minimum

$$w(C) = \sum_{v \in C} w(v).$$



# Weighted Vertex Cover

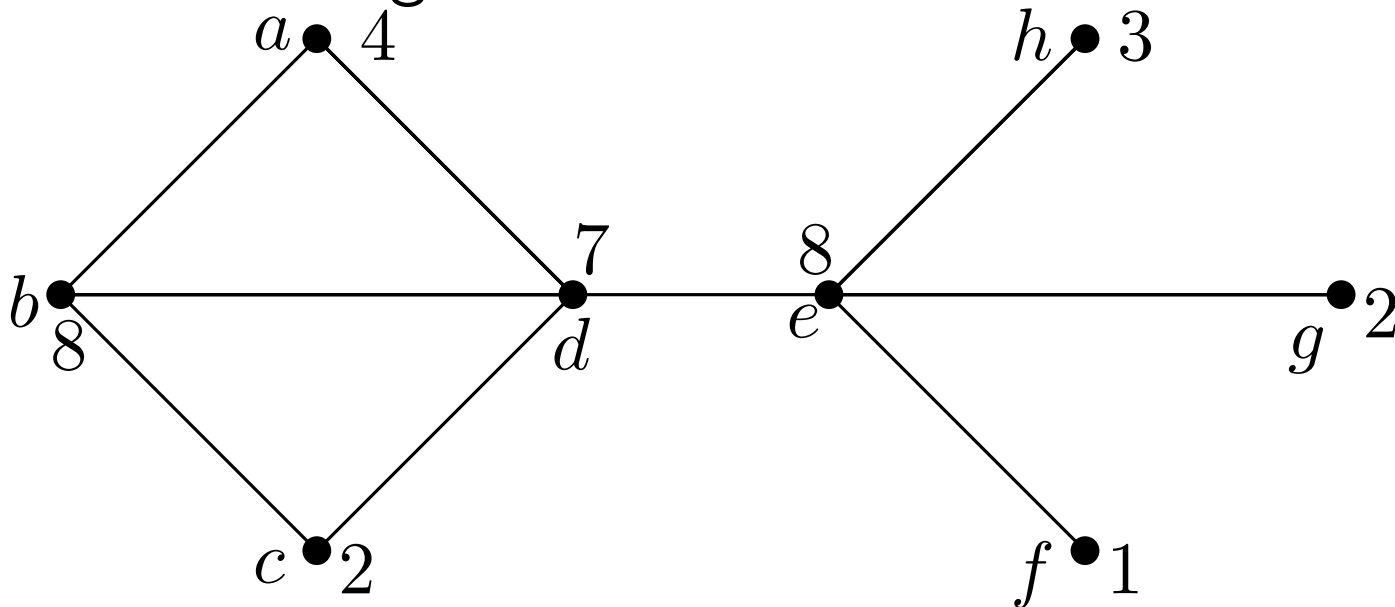
**Def.:** Let  $G = (V, E)$  be a graph. A set  $V' \subseteq V$  of vertices is a *vertex cover* if for all  $uv \in E$ , we have  $u \in V'$  or  $v \in V'$ .

**Now:** We are given weight  $w(v) > 0$  for each  $v \in V$ .

**Goal:** Find vertex cover  $C$  with minimum

$$w(C) = \sum_{v \in C} w(v).$$

**Exercise:** Find minimum (unweighted) vertex cover and then minimum weighted vertex cover.



# Weighted Vertex Cover

**Def.:** Let  $G = (V, E)$  be a graph. A set  $V' \subseteq V$  of vertices is a *vertex cover* if for all  $uv \in E$ , we have  $u \in V'$  or  $v \in V'$ .

**Now:** We are given weight  $w(v) > 0$  for each  $v \in V$ .

**Goal:** Find vertex cover  $C$  with minimum

$$w(C) = \sum_{v \in C} w(v).$$

0-1-integer program (IP):

$$\begin{array}{ll} x_v \in \{0, 1\}, \forall v \in V & (x_v = 1 \Leftrightarrow v \in C) \\ x_u + x_v \geq 1, \forall uv \in E & (\text{edge } uv \text{ covered}) \end{array}$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

# LP-relaxation

0-1-integer program (IP):

$$\begin{aligned} x_v &\in \{0, 1\}, \forall v \in V && (x_v = 1 \Leftrightarrow v \in C) \\ x_u + x_v &\geq 1, \forall uv \in E && (\text{edge } uv \text{ covered}) \end{aligned}$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

IP is NP-complete!

# LP-relaxation

0-1-integer program (IP):

$$\begin{aligned} x_v &\in \{0, 1\}, \forall v \in V && (x_v = 1 \Leftrightarrow v \in C) \\ x_u + x_v &\geq 1, \forall uv \in E && (\text{edge } uv \text{ covered}) \end{aligned}$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

IP is NP-complete!

**LP-relaxation:** replace  $x_v \in \{0, 1\}$  with  $0 \leq x_v \leq 1$ . Result:

# LP-relaxation

0-1-integer program (IP):

$$\begin{aligned}x_v &\in \{0, 1\}, \forall v \in V && (x_v = 1 \Leftrightarrow v \in C) \\x_u + x_v &\geq 1, \forall uv \in E && (\text{edge } uv \text{ covered})\end{aligned}$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

IP is NP-complete!

**LP-relaxation:** replace  $x_v \in \{0, 1\}$  with  $0 \leq x_v \leq 1$ . Result:

$$\begin{aligned}0 &\leq x_v \leq 1, \forall v \in V \\x_u + x_v &\geq 1, \forall uv \in E\end{aligned}$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

# LP-relaxation

0-1-integer program (IP):

$$\begin{aligned} x_v &\in \{0, 1\}, \forall v \in V && (x_v = 1 \Leftrightarrow v \in C) \\ x_u + x_v &\geq 1, \forall uv \in E && (\text{edge } uv \text{ covered}) \end{aligned}$$

IP is NP-complete!

**LP-relaxation:** replace  $x_v \in \{0, 1\}$  with  $0 \leq x_v \leq 1$ . Result:

$$\begin{aligned} 0 &\leq x_v \leq 1, \forall v \in V \\ x_u + x_v &\geq 1, \forall uv \in E \end{aligned}$$


$$\text{minimize } \sum_{v \in V} w(v)x_v$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

Relaxed solution can be smaller, not larger

# Algorithm

**LP:**

$$0 \leq x_v \leq 1, \forall v \in V$$

$$x_u + x_v \geq 1, \forall uv \in E$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

APPROX-MIN-WEIGHT-VC( $G, W$ ):

Compute opt. sol.  $\bar{x}$  to LP

return  $C := \{v \in V \mid \bar{x}_v \geq 1/2\}$

# Algorithm

**LP:**

$$0 \leq x_v \leq 1, \forall v \in V$$

$$x_u + x_v \geq 1, \forall uv \in E$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

**APPROX-MIN-WEIGHT-VC( $G, W$ ):**

Compute opt. sol.  $\bar{x}$  to LP

return  $C := \{v \in V \mid \bar{x}_v \geq 1/2\}$

**Theorem:** Alg. is a polynomial-time 2-approximation algorithm for minimum-weight vertex cover.



# Algorithm

**LP:**

$$0 \leq x_v \leq 1, \forall v \in V$$

$$x_u + x_v \geq 1, \forall uv \in E$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

**APPROX-MIN-WEIGHT-VC( $G, W$ ):**

Compute opt. sol.  $\bar{x}$  to LP

return  $C := \{v \in V \mid \bar{x}_v \geq 1/2\}$

**Theorem:** Alg. is a polynomial-time 2-approximation algorithm for minimum-weight vertex cover.

**Need to prove:** 1) Polynomial time. 2) Alg. produces feasible solution (i.e.,  $C$  is a vertex cover). 3)  $\frac{w(C)}{w(C^*)} \leq 2$ .

# Algorithm

**LP:**

$$0 \leq x_v \leq 1, \forall v \in V$$

$$x_u + x_v \geq 1, \forall uv \in E$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

**APPROX-MIN-WEIGHT-VC( $G, W$ ):**

Compute opt. sol.  $\bar{x}$  to LP

return  $C := \{v \in V \mid \bar{x}_v \geq 1/2\}$

**Theorem:** Alg. is a polynomial-time 2-approximation algorithm for minimum-weight vertex cover.

**Need to prove:** 1) Polynomial time. 2) Alg. produces feasible solution (i.e.,  $C$  is a vertex cover). 3)  $\frac{w(C)}{w(C^*)} \leq 2$ .

1) ✓

# Algorithm

**LP:**

$$0 \leq x_v \leq 1, \forall v \in V$$

$$x_u + x_v \geq 1, \forall uv \in E$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

**APPROX-MIN-WEIGHT-VC( $G, W$ ):**

Compute opt. sol.  $\bar{x}$  to LP

return  $C := \{v \in V \mid \bar{x}_v \geq 1/2\}$

**Theorem:** Alg. is a polynomial-time 2-approximation algorithm for minimum-weight vertex cover.

**Need to prove:** 1) Polynomial time. 2) Alg. produces feasible solution (i.e.,  $C$  is a vertex cover). 3)  $\frac{w(C)}{w(C^*)} \leq 2$ .

1) ✓

2)  $uv \in E \Rightarrow \bar{x}_u + \bar{x}_v \geq 1 \Rightarrow \bar{x}_u \geq \frac{1}{2} \vee \bar{x}_v \geq \frac{1}{2} \Rightarrow u \in C \vee v \in C$ . ✓

# Algorithm

**LP:**

$$0 \leq x_v \leq 1, \forall v \in V$$

$$x_u + x_v \geq 1, \forall uv \in E$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

**APPROX-MIN-WEIGHT-VC( $G, W$ ):**

Compute opt. sol.  $\bar{x}$  to LP

return  $C := \{v \in V \mid \bar{x}_v \geq 1/2\}$

**Need to prove:** 3)  $\frac{w(C)}{w(C^*)} \leq 2$ .

# Algorithm

**LP:**

$$0 \leq x_v \leq 1, \forall v \in V$$

$$x_u + x_v \geq 1, \forall uv \in E$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

**APPROX-MIN-WEIGHT-VC( $G, W$ ):**

Compute opt. sol.  $\bar{x}$  to LP

return  $C := \{v \in V \mid \bar{x}_v \geq 1/2\}$

**Need to prove:** 3)  $\frac{w(C)}{w(C^*)} \leq 2$ .

Let  $z^* := \sum_{v \in V} \bar{x}_v w(v)$ , recall  $z^* \leq w(C^*)$ .

# Algorithm

**LP:**

$$0 \leq x_v \leq 1, \forall v \in V$$

$$x_u + x_v \geq 1, \forall uv \in E$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

**APPROX-MIN-WEIGHT-VC( $G, W$ ):**

Compute opt. sol.  $\bar{x}$  to LP

return  $C := \{v \in V \mid \bar{x}_v \geq 1/2\}$

**Need to prove:** 3)  $\frac{w(C)}{w(C^*)} \leq 2$ .

Let  $z^* := \sum_{v \in V} \bar{x}_v w(v)$ , recall  $z^* \leq w(C^*)$ .

$$w(C) = \sum_{v \in C} w(v) = \sum_{v \in V} w(v)[v \in C] = \sum_{v \in V} w(v)[\bar{x}_v \geq \tfrac{1}{2}]$$

# Algorithm

**LP:**

$$0 \leq x_v \leq 1, \forall v \in V$$

$$x_u + x_v \geq 1, \forall uv \in E$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

**APPROX-MIN-WEIGHT-VC( $G, W$ ):**

Compute opt. sol.  $\bar{x}$  to LP

return  $C := \{v \in V \mid \bar{x}_v \geq 1/2\}$

**Need to prove:** 3)  $\frac{w(C)}{w(C^*)} \leq 2$ .

Let  $z^* := \sum_{v \in V} \bar{x}_v w(v)$ , recall  $z^* \leq w(C^*)$ .

$$w(C) = \sum_{v \in C} w(v) = \sum_{v \in V} w(v)[v \in C] = \sum_{v \in V} w(v) \boxed{\bar{x}_v \geq \frac{1}{2}} \stackrel{\leq 2\bar{x}_v}{\leq}$$

# Algorithm

**LP:**

$$0 \leq x_v \leq 1, \forall v \in V$$

$$x_u + x_v \geq 1, \forall uv \in E$$

$$\text{minimize } \sum_{v \in V} w(v)x_v$$

**APPROX-MIN-WEIGHT-VC( $G, W$ ):**

Compute opt. sol.  $\bar{x}$  to LP

return  $C := \{v \in V \mid \bar{x}_v \geq 1/2\}$

**Need to prove:** 3)  $\frac{w(C)}{w(C^*)} \leq 2$ .

Let  $z^* := \sum_{v \in V} \bar{x}_v w(v)$ , recall  $z^* \leq w(C^*)$ .

$$w(C) = \sum_{v \in C} w(v) = \sum_{v \in V} w(v)[v \in C] = \sum_{v \in V} w(v) \overset{\leq 2\bar{x}_v}{\boxed{[\bar{x}_v \geq \frac{1}{2}]}}$$

$$\leq 2 \sum_{v \in V} w(v)\bar{x}_v = 2z^* \leq 2w(C^*) \Rightarrow \frac{w(C)}{w(C^*)} \leq 2. \quad \checkmark$$



## Reflection and methodology

How can we prove  $w(C)/w(C^*) \leq 2$  when we don't know  $w(C^*)$ ?

Answer: By proving  $w(C) \leq 2z^*$  and  $|z^*| \leq w(C^*)$ .

## Reflection and methodology

How can we prove  $w(C)/w(C^*) \leq 2$  when we don't know  $w(C^*)$ ?

Answer: By proving  $w(C) \leq 2z^*$  and  $|z^*| \leq w(C^*)$ .

General technique: Find a parameter  $\square$  such that  $C \leq \rho \cdot \square$  and  $\square \leq C^*$ .

For weighted vertex cover:  $\square = z^*$  and  $\rho = 2$ .

# Approximation schemes

## **Polynomial-time approximation scheme (PTAS):**

Approximation algorithm that takes instance  $I$  of an optimization problem  $P$  and  $\varepsilon > 0$  as input. For any fixed  $\varepsilon$  works as  $(1 + \varepsilon)$ -approximation algorithm for  $P$ .

# Approximation schemes

## **Polynomial-time approximation scheme (PTAS):**

Approximation algorithm that takes instance  $I$  of an optimization problem  $P$  and  $\varepsilon > 0$  as input. For any fixed  $\varepsilon$  works as  $(1 + \varepsilon)$ -approximation algorithm for  $P$ .

**Ex:** Runtime  $O(2^{1/\varepsilon} \cdot n^3)$  or  $O(n^{1/\varepsilon})$  or  $O(n \log n / \varepsilon^2)$ .

# Approximation schemes

## **Polynomial-time approximation scheme (PTAS):**

Approximation algorithm that takes instance  $I$  of an optimization problem  $P$  and  $\varepsilon > 0$  as input. For any fixed  $\varepsilon$  works as  $(1 + \varepsilon)$ -approximation algorithm for  $P$ .

**Ex:** Runtime  $O(2^{1/\varepsilon} \cdot n^3)$  or  $O(n^{1/\varepsilon})$  or  $O(n \log n / \varepsilon^2)$ .

## **Fully polynomial-time approximation scheme (FPTAS):**

PTAS with runtime polynomial in  $1/\varepsilon$  and the size of  $I$ .

# Approximation schemes

## **Polynomial-time approximation scheme (PTAS):**

Approximation algorithm that takes instance  $I$  of an optimization problem  $P$  and  $\varepsilon > 0$  as input. For any fixed  $\varepsilon$  works as  $(1 + \varepsilon)$ -approximation algorithm for  $P$ .

**Ex:** Runtime  $O(2^{1/\varepsilon} \cdot n^3)$  or  $O(n^{1/\varepsilon})$  or  $O(n \log n / \varepsilon^2)$ .

## **Fully polynomial-time approximation scheme (FPTAS):**

PTAS with runtime polynomial in  $1/\varepsilon$  and the size of  $I$ .

**Ex:** Runtime  $O(n \log n / \varepsilon^2)$ .

# SUBSET-SUM

**Input:** Set  $S = \{x_1, \dots, x_n\} \subset \mathbb{N}$ , and  $t \in \mathbb{N}$ .

**Goal:** Find  $U \subset S$  s.t.  $\sum_{x \in U} x \leq t$  with maximum  $\sum_{x \in U} x$ .

# SUBSET-SUM

**Input:** Set  $S = \{x_1, \dots, x_n\} \subset \mathbb{N}$ , and  $t \in \mathbb{N}$ .

**Goal:** Find  $U \subset S$  s.t.  $\sum_{x \in U} x \leq t$  with maximum  $\sum_{x \in U} x$ .

**Example:**  $S = \{1, 4, 5\}$ ,  $t = 8$ .



# SUBSET-SUM

**Input:** Set  $S = \{x_1, \dots, x_n\} \subset \mathbb{N}$ , and  $t \in \mathbb{N}$ .

**Goal:** Find  $U \subset S$  s.t.  $\sum_{x \in U} x \leq t$  with maximum  $\sum_{x \in U} x$ .

**Example:**  $S = \{1, 4, 5\}$ ,  $t = 8$ .

NP-complete to decide if  $\exists U \subset S : \sum_{x \in U} x = t$ .

# SUBSET-SUM

**Input:** Set  $S = \{x_1, \dots, x_n\} \subset \mathbb{N}$ , and  $t \in \mathbb{N}$ .

**Goal:** Find  $U \subset S$  s.t.  $\sum_{x \in U} x \leq t$  with maximum  $\sum_{x \in U} x$ .

**Example:**  $S = \{1, 4, 5\}$ ,  $t = 8$ .

NP-complete to decide if  $\exists U \subset S : \sum_{x \in U} x = t$ .

**Abstract exact alg.:**

for  $k = 1, 2, \dots, n$

    compute  $L_k := \left\{ \sum_{x \in U} x \mid U \subset \{x_1, \dots, x_k\} \wedge \sum_{x \in U} x \leq t \right\}$ .

return  $\max L_n$

# SUBSET-SUM

**Input:** Set  $S = \{x_1, \dots, x_n\} \subset \mathbb{N}$ , and  $t \in \mathbb{N}$ .

**Goal:** Find  $U \subset S$  s.t.  $\sum_{x \in U} x \leq t$  with maximum  $\sum_{x \in U} x$ .

**Example:**  $S = \{1, 4, 5\}$ ,  $t = 8$ .

NP-complete to decide if  $\exists U \subset S : \sum_{x \in U} x = t$ .

**Abstract exact alg.:**

for  $k = 1, 2, \dots, n$

    compute  $L_k := \{\sum_{x \in U} x \mid U \subset \{x_1, \dots, x_k\} \wedge \sum_{x \in U} x \leq t\}$ .

return  $\max L_n$

**Note:**  $L_k \subset L_{k-1} \cup (L_{k-1} + x_k)$ .

# SUBSET-SUM

**Input:** Set  $S = \{x_1, \dots, x_n\} \subset \mathbb{N}$ , and  $t \in \mathbb{N}$ .

**Goal:** Find  $U \subset S$  s.t.  $\sum_{x \in U} x \leq t$  with maximum  $\sum_{x \in U} x$ .

**Example:**  $S = \{1, 4, 5\}$ ,  $t = 8$ .

NP-complete to decide if  $\exists U \subset S : \sum_{x \in U} x = t$ .

**Abstract exact alg.:**

```
for  $k = 1, 2, \dots, n$ 
  compute  $L_k := \{ \sum_{x \in U} x \mid U \subset \{x_1, \dots, x_k\} \wedge \sum_{x \in U} x \leq t \}$ .
return  $\max L_n$ 
```

**Note:**  $L_k \subset L_{k-1} \cup (L_{k-1} + x_k)$ .

EXACT-SUBSET-SUM( $S, t$ )

```
 $L_0 = [0]$ 
for  $k = 1, \dots, n$ 
   $L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$ 
  remove from  $L_k$  duplicates and elements  $> t$ 
return last( $L_n$ )
```

# SUBSET-SUM

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Example:**  $S = \{1, 4, 5\}$ ,  $t = 8$ .

# SUBSET-SUM

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Example:**  $S = \{1, 4, 5\}$ ,  $t = 8$ .

$L_0 = [0]$

# SUBSET-SUM

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Example:**  $S = \{1, 4, 5\}$ ,  $t = 8$ .

$L_0 = [0]$

$L_1 = L_0 \cup (L_0 + 1) = [0] \cup [1] = [0, 1]$

# SUBSET-SUM

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Example:**  $S = \{1, 4, 5\}$ ,  $t = 8$ .

$L_0 = [0]$

$L_1 = L_0 \cup (L_0 + 1) = [0] \cup [1] = [0, 1]$

$L_2 = L_1 \cup (L_1 + 4) = [0, 1] \cup [4, 5] = [0, 1, 4, 5]$



# SUBSET-SUM

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Example:**  $S = \{1, 4, 5\}$ ,  $t = 8$ .

$L_0 = [0]$

$L_1 = L_0 \cup (L_0 + 1) = [0] \cup [1] = [0, 1]$

$L_2 = L_1 \cup (L_1 + 4) = [0, 1] \cup [4, 5] = [0, 1, 4, 5]$

$L_3 = L_2 \cup (L_2 + 5) = [0, 1, 4, 5] \cup [\cancel{5}, \cancel{6}, \cancel{9}, \cancel{10}] = [0, 1, 4, 5, \boxed{6}]$

# SUBSET-SUM

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Example:**  $S = \{1, 4, 5\}$ ,  $t = 8$ .

$L_0 = [0]$

$L_1 = L_0 \cup (L_0 + 1) = [0] \cup [1] = [0, 1]$

$L_2 = L_1 \cup (L_1 + 4) = [0, 1] \cup [4, 5] = [0, 1, 4, 5]$

$L_3 = L_2 \cup (L_2 + 5) = [0, 1, 4, 5] \cup [\cancel{5}, \cancel{6}, \cancel{9}, \cancel{10}] = [0, 1, 4, 5, \boxed{6}]$

**Running time:** Computing  $L_k$ :  $O(|L_{k-1}|)$ .

Total:  $O(\sum_{k=1}^n |L_k|)$

# SUBSET-SUM

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Example:**  $S = \{1, 4, 5\}$ ,  $t = 8$ .

$L_0 = [0]$

$L_1 = L_0 \cup (L_0 + 1) = [0] \cup [1] = [0, 1]$

$L_2 = L_1 \cup (L_1 + 4) = [0, 1] \cup [4, 5] = [0, 1, 4, 5]$

$L_3 = L_2 \cup (L_2 + 5) = [0, 1, 4, 5] \cup [\cancel{5}, \cancel{6}, \cancel{9}, \cancel{10}] = [0, 1, 4, 5, \boxed{6}]$

**Running time:** Computing  $L_k$ :  $O(|L_{k-1}|)$ .

Total:  $O(\sum_{k=1}^n |L_k|) = O(nt)$

# SUBSET-SUM

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Example:**  $S = \{1, 4, 5\}$ ,  $t = 8$ .

$L_0 = [0]$

$L_1 = L_0 \cup (L_0 + 1) = [0] \cup [1] = [0, 1]$

$L_2 = L_1 \cup (L_1 + 4) = [0, 1] \cup [4, 5] = [0, 1, 4, 5]$

$L_3 = L_2 \cup (L_2 + 5) = [0, 1, 4, 5] \cup [\cancel{5}, \cancel{6}, \cancel{9}, \cancel{10}] = [0, 1, 4, 5, \boxed{6}]$

**Running time:** Computing  $L_k$ :  $O(|L_{k-1}|)$ .

Total:  $O(\sum_{k=1}^n |L_k|) = O(nt) = O(n2^{\log t})$  **EXPONENTIAL !!**

# Trimming

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Idea:** Trim list

$L \subset \{0, 1, \dots, t\}$  with  
parameter  $\delta > 0$ : if we keep  
 $s \in L$ , then remove  
 $(s, (1 + \delta)s]$ .

# Trimming

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Idea:** Trim list

$L \subset \{0, 1, \dots, t\}$  with  
parameter  $\delta > 0$ : if we keep  
 $s \in L$ , then remove  
 $(s, (1 + \delta)s]$ .

**Example:**  $L = [0, 9, 10, 11, 12, 13, 16]$ ,  $\delta = 0.1$ .

# Trimming

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Idea:** Trim list

$L \subset \{0, 1, \dots, t\}$  with  
parameter  $\delta > 0$ : if we keep  
 $s \in L$ , then remove  
 $(s, (1 + \delta)s]$ .

**Example:**  $L = [0, 9, 10, \text{~~11~~, 12, 13, 16}]$ ,  $\delta = 0.1$ .

# Trimming

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Idea:** Trim list

$L \subset \{0, 1, \dots, t\}$  with  
parameter  $\delta > 0$ : if we keep  
 $s \in L$ , then remove  
 $(s, (1 + \delta)s]$ .

**Example:**  $L = [0, 9, 10, \text{~~11~~, 12, \text{~~13~~, 16}]$ ,  $\delta = 0.1$ .



# Trimming

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Idea:** Trim list

$L \subset \{0, 1, \dots, t\}$  with  
parameter  $\delta > 0$ : if we keep  
 $s \in L$ , then remove  
 $(s, (1 + \delta)s]$ .

**Example:**  $L = [0, 9, 10, \text{~~11~~, 12, \text{~~13~~, 16}]$ ,  $\delta = 0.1$ .

TRIM( $L = [s_1, \dots, s_m], \delta$ )

$L' = [s_1]$

for  $i = 2, \dots, m$

if  $s_i > \text{last}(L') \cdot (1 + \delta)$

$L' = L' \cup [s_i]$

return  $L'$

# Trimming

EXACT-SUBSET-SUM( $S, t$ )

$L_0 = [0]$

for  $k = 1, \dots, n$

$L_k = \text{MERGE-LISTS}(L_{k-1}, L_{k-1} + x_k)$

remove duplicates and elm.s  $> t$

return last( $L_n$ )

**Idea:** Trim list

$L \subset \{0, 1, \dots, t\}$  with  
parameter  $\delta > 0$ : if we keep  
 $s \in L$ , then remove  
 $(s, (1 + \delta)s]$ .

**Example:**  $L = [0, 9, 10, \text{~~11~~, 12, \text{~~13~~, 16}]$ ,  $\delta = 0.1$ .

TRIM( $L = [s_1, \dots, s_m], \delta$ )

$L' = [s_1]$

for  $i = 2, \dots, m$

if  $s_i > \text{last}(L') \cdot (1 + \delta)$

$L' = L' \cup [s_i]$

return  $L'$

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return last( $L'_n$ )

# Theorem

**Thm.:** The alg. is an FPTAS.

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return last( $L'_n$ )

# Theorem

**Thm.:** The alg. is an FPTAS.

*Proof:* Feasibility: Opt. is  $s_{\max} = \text{last}(L_n) \leq t$  and  $\text{last}(L'_n) \leq \text{last}(L_n)$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return  $\text{last}(L'_n)$

# Theorem

**Thm.:** The alg. is an FPTAS.

*Proof:* Feasibility: Opt. is  $s_{\max} = \text{last}(L_n) \leq t$  and  $\text{last}(L'_n) \leq \text{last}(L_n)$ .

Approx. ratio: Assume we trim with  $\delta$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return  $\text{last}(L'_n)$

# Theorem

**Thm.:** The alg. is an FPTAS.

*Proof:* Feasibility: Opt. is  $s_{\max} = \text{last}(L_n) \leq t$  and  $\text{last}(L'_n) \leq \text{last}(L_n)$ .

Approx. ratio: Assume we trim with  $\delta$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return  $\text{last}(L'_n)$

From exercise:  $\forall s \in L_k \exists s' \in L'_k : s' \leq s \leq (1 + \delta)^k s'$ .

# Theorem

**Thm.:** The alg. is an FPTAS.

*Proof:* Feasibility: Opt. is  $s_{\max} = \text{last}(L_n) \leq t$  and  $\text{last}(L'_n) \leq \text{last}(L_n)$ .

Approx. ratio: Assume we trim with  $\delta$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return  $\text{last}(L'_n)$

From exercise:  $\forall s \in L_k \exists s' \in L'_k : s' \leq s \leq (1 + \delta)^k s'$

$$\Rightarrow \frac{s}{s'} \leq (1 + \delta)^k$$

# Theorem

**Thm.:** The alg. is an FPTAS.

*Proof:* Feasibility: Opt. is  $s_{\max} = \text{last}(L_n) \leq t$  and  $\text{last}(L'_n) \leq \text{last}(L_n)$ .

Approx. ratio: Assume we trim with  $\delta$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return  $\text{last}(L'_n)$

From exercise:  $\forall s \in L_k \exists s' \in L'_k : s' \leq s \leq (1 + \delta)^k s'$

$$\Rightarrow \frac{s}{s'} \leq (1 + \delta)^k$$

From exercise, there is  $s' \in L'_n$  such that  $\frac{s_{\max}}{s'} \leq (1 + \delta)^n$ .



# Theorem

**Thm.:** The alg. is an FPTAS.

*Proof:* Feasibility: Opt. is  $s_{\max} = \text{last}(L_n) \leq t$  and  $\text{last}(L'_n) \leq \text{last}(L_n)$ .

Approx. ratio: Assume we trim with  $\delta$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return  $\text{last}(L'_n)$

From exercise:  $\forall s \in L_k \exists s' \in L'_k : s' \leq s \leq (1 + \delta)^k s'$   
 $\Rightarrow \frac{s}{s'} \leq (1 + \delta)^k$

From exercise, there is  $s' \in L'_n$  such that  $\frac{s_{\max}}{s'} \leq (1 + \delta)^n$ .

Approximation ratio:  $\frac{s_{\max}}{\text{last}(L'_n)} \leq \frac{s_{\max}}{s'} \leq (1 + \delta)^n$ .

# Theorem

**Thm.:** The alg. is an FPTAS.

*Proof:* Feasibility: Opt. is  $s_{\max} = \text{last}(L_n) \leq t$  and  $\text{last}(L'_n) \leq \text{last}(L_n)$ .

Approx. ratio: Assume we trim with  $\delta$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return  $\text{last}(L'_n)$

From exercise:  $\forall s \in L_k \exists s' \in L'_k : s' \leq s \leq (1 + \delta)^k s'$

$$\Rightarrow \frac{s}{s'} \leq (1 + \delta)^k$$

want  $\leq 1 + \varepsilon$  !

From exercise, there is  $s' \in L'_n$  such that  $\frac{s_{\max}}{s'} \leq (1 + \delta)^n$ .

Approximation ratio:  $\frac{s_{\max}}{\text{last}(L'_n)} \leq \frac{s_{\max}}{s'} \leq (1 + \delta)^n$

# Theorem

**Thm.:** The alg. is an FPTAS.

*Proof:* Feasibility: Opt. is  $s_{\max} = \text{last}(L_n) \leq t$  and  $\text{last}(L'_n) \leq \text{last}(L_n)$ .

Approx. ratio: Assume we trim with  $\delta$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return  $\text{last}(L'_n)$

From exercise:  $\forall s \in L_k \exists s' \in L'_k : s' \leq s \leq (1 + \delta)^k s'$

$$\Rightarrow \frac{s}{s'} \leq (1 + \delta)^k$$

want  $\leq 1 + \varepsilon$  !

From exercise, there is  $s' \in L'_n$  such that  $\frac{s_{\max}}{s'} \leq (1 + \delta)^n$ .

Approximation ratio:  $\frac{s_{\max}}{\text{last}(L'_n)} \leq \frac{s_{\max}}{s'} \leq (1 + \delta)^n$

**Claim:**  $(1 + \delta)^n \leq 1 + 2n\delta$  if  $2n\delta \leq 1$ .

# Theorem

**Thm.:** The alg. is an FPTAS.

*Proof:* Feasibility: Opt. is  $s_{\max} = \text{last}(L_n) \leq t$  and  $\text{last}(L'_n) \leq \text{last}(L_n)$ .

Approx. ratio: Assume we trim with  $\delta$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return  $\text{last}(L'_n)$

From exercise:  $\forall s \in L_k \exists s' \in L'_k : s' \leq s \leq (1 + \delta)^k s'$

$$\Rightarrow \frac{s}{s'} \leq (1 + \delta)^k$$

want  $\leq 1 + \varepsilon$  !

From exercise, there is  $s' \in L'_n$  such that  $\frac{s_{\max}}{s'} \leq (1 + \delta)^n$ .

Approximation ratio:  $\frac{s_{\max}}{\text{last}(L'_n)} \leq \frac{s_{\max}}{s'} \leq (1 + \delta)^n$

**Claim:**  $(1 + \delta)^n \leq 1 + 2n\delta$  if  $2n\delta \leq 1$ .

Induction:  $(1 + \delta)^0 = 1 = 1 + 2 \cdot 0 \cdot \delta$ . ✓

# Theorem

**Thm.:** The alg. is an FPTAS.

*Proof:* Feasibility: Opt. is  $s_{\max} = \text{last}(L_n) \leq t$  and  $\text{last}(L'_n) \leq \text{last}(L_n)$ .

Approx. ratio: Assume we trim with  $\delta$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return  $\text{last}(L'_n)$

From exercise:  $\forall s \in L_k \exists s' \in L'_k : s' \leq s \leq (1 + \delta)^k s'$

$$\Rightarrow \frac{s}{s'} \leq (1 + \delta)^k$$

want  $\leq 1 + \varepsilon$  !

From exercise, there is  $s' \in L'_n$  such that  $\frac{s_{\max}}{s'} \leq (1 + \delta)^n$ .

$$\text{Approximation ratio: } \frac{s_{\max}}{\text{last}(L'_n)} \leq \frac{s_{\max}}{s'} \leq (1 + \delta)^n$$

**Claim:**  $(1 + \delta)^n \leq 1 + 2n\delta$  if  $2n\delta \leq 1$ .

Induction:  $(1 + \delta)^0 = 1 = 1 + 2 \cdot 0 \cdot \delta$ . ✓

$$(1 + \delta)^n = (1 + \delta)^{n-1}(1 + \delta) \leq (1 + 2(n-1)\delta)(1 + \delta)$$

$$= 1 + 2n\delta - 2\delta + \delta + \delta \cdot 2(n-1)\delta$$

use induction hypothesis

# Theorem

**Thm.:** The alg. is an FPTAS.

*Proof:* Feasibility: Opt. is  $s_{\max} = \text{last}(L_n) \leq t$  and  $\text{last}(L'_n) \leq \text{last}(L_n)$ .

Approx. ratio: Assume we trim with  $\delta$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return  $\text{last}(L'_n)$

From exercise:  $\forall s \in L_k \exists s' \in L'_k : s' \leq s \leq (1 + \delta)^k s'$

$$\Rightarrow \frac{s}{s'} \leq (1 + \delta)^k$$

want  $\leq 1 + \varepsilon$  !

From exercise, there is  $s' \in L'_n$  such that  $\frac{s_{\max}}{s'} \leq (1 + \delta)^n$ .

$$\text{Approximation ratio: } \frac{s_{\max}}{\text{last}(L'_n)} \leq \frac{s_{\max}}{s'} \leq (1 + \delta)^n$$

**Claim:**  $(1 + \delta)^n \leq 1 + 2n\delta$  if  $2n\delta \leq 1$ .

Induction:  $(1 + \delta)^0 = 1 = 1 + 2 \cdot 0 \cdot \delta$ . ✓

$$(1 + \delta)^n = (1 + \delta)^{n-1} (1 + \delta) \leq (1 + 2(n-1)\delta)(1 + \delta)$$

$$= 1 + 2n\delta - 2\delta + \delta + \delta \cdot 2(n-1)\delta$$

$< 1$

use induction hypothesis

# Theorem

**Thm.:** The alg. is an FPTAS.

*Proof:* Feasibility: Opt. is  $s_{\max} = \text{last}(L_n) \leq t$  and  $\text{last}(L'_n) \leq \text{last}(L_n)$ .

Approx. ratio: Assume we trim with  $\delta$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return  $\text{last}(L'_n)$

From exercise:  $\forall s \in L_k \exists s' \in L'_k : s' \leq s \leq (1 + \delta)^k s'$

$$\Rightarrow \frac{s}{s'} \leq (1 + \delta)^k$$

want  $\leq 1 + \varepsilon$  !

From exercise, there is  $s' \in L'_n$  such that  $\frac{s_{\max}}{s'} \leq (1 + \delta)^n$ .

$$\text{Approximation ratio: } \frac{s_{\max}}{\text{last}(L'_n)} \leq \frac{s_{\max}}{s'} \leq (1 + \delta)^n$$

**Claim:**  $(1 + \delta)^n \leq 1 + 2n\delta$  if  $2n\delta \leq 1$ .

Induction:  $(1 + \delta)^0 = 1 = 1 + 2 \cdot 0 \cdot \delta$ . ✓

$$(1 + \delta)^n = (1 + \delta)^{n-1} (1 + \delta) \leq (1 + 2(n-1)\delta) (1 + \delta)$$

$$= 1 + 2n\delta - 2\delta + \delta + \delta \cdot 2(n-1)\delta < 1 + 2n\delta.$$

$< 1$

use induction hypothesis

# Theorem

**Thm.:** The alg. is an FPTAS.

*Proof:* Feasibility: Opt. is  $s_{\max} = \text{last}(L_n) \leq t$  and  $\text{last}(L'_n) \leq \text{last}(L_n)$ .

Approx. ratio: Assume we trim with  $\delta$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return  $\text{last}(L'_n)$

From exercise:  $\forall s \in L_k \exists s' \in L'_k : s' \leq s \leq (1 + \delta)^k s'$

$$\Rightarrow \frac{s}{s'} \leq (1 + \delta)^k$$

want  $\leq 1 + \varepsilon$  !

From exercise, there is  $s' \in L'_n$  such that  $\frac{s_{\max}}{s'} \leq (1 + \delta)^n$ .

Approximation ratio:  $\frac{s_{\max}}{\text{last}(L'_n)} \leq \frac{s_{\max}}{s'} \leq (1 + \delta)^n$

**Claim:**  $(1 + \delta)^n \leq 1 + 2n\delta$  if  $2n\delta \leq 1$ .

$$\delta := \varepsilon/2n \Rightarrow 1 + 2n\delta \leq 1 + \varepsilon \checkmark$$

Induction:  $(1 + \delta)^0 = 1 = 1 + 2 \cdot 0 \cdot \delta$ .  $\checkmark$

$$(1 + \delta)^n = (1 + \delta)^{n-1}(1 + \delta) \leq (1 + 2(n-1)\delta)(1 + \delta)$$

$$= 1 + 2n\delta - 2\delta + \delta + \delta \cdot 2(n-1)\delta < 1 + 2n\delta.$$

$< 1$

use induction hypothesis



# Running time

**Thm.:** The alg. is an FPTAS.

Running time:

$$O\left(\sum_{k=1}^n |L'_k|\right).$$

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return last( $L'_n$ )

# Running time

**Thm.:** The alg. is an FPTAS.

Running time:

$$O\left(\sum_{k=1}^n |L'_k|\right).$$

Claim:  $|L'_k| = O\left(\frac{n \log t}{\varepsilon}\right)$ .

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return last( $L'_n$ )

# Running time

**Thm.:** The alg. is an FPTAS.

Running time:

$$O\left(\sum_{k=1}^n |L'_k|\right).$$

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return last( $L'_n$ )

Claim:  $|L'_k| = O\left(\frac{n \log t}{\varepsilon}\right)$ .

Let  $L'_k = [0, s_0, s_1, \dots, s_m]$ . Then

$t \geq s_m > (1 + \delta)s_{m-1} > \dots > (1 + \delta)^m s_0 \geq (1 + \delta)^m$ .

Recall  $\delta = \varepsilon/2n$ .

# Running time

**Thm.:** The alg. is an FPTAS.

Running time:

$$O\left(\sum_{k=1}^n |L'_k|\right).$$

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return last( $L'_n$ )

Claim:  $|L'_k| = O\left(\frac{n \log t}{\varepsilon}\right)$ .

Let  $L'_k = [0, s_0, s_1, \dots, s_m]$ . Then

$t \geq s_m > (1 + \delta)s_{m-1} > \dots > (1 + \delta)^m s_0 \geq (1 + \delta)^m$ .

So  $m < \log_{1+\delta} t = \frac{\ln t}{\ln(1+\delta)}$ .

Recall  $\delta = \varepsilon/2n$ .

# Running time

**Thm.:** The alg. is an FPTAS.

Running time:

$$O\left(\sum_{k=1}^n |L'_k|\right).$$

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return last( $L'_n$ )

Claim:  $|L'_k| = O\left(\frac{n \log t}{\varepsilon}\right)$ .

Let  $L'_k = [0, s_0, s_1, \dots, s_m]$ . Then

$t \geq s_m > (1 + \delta)s_{m-1} > \dots > (1 + \delta)^m s_0 \geq (1 + \delta)^m$ .

So  $m < \log_{1+\delta} t = \frac{\ln t}{\ln(1+\delta)}$ .

CLRS eq. (3.17): if  $\delta > -1$ :  $\delta \geq \ln(1 + \delta) \geq \frac{\delta}{1+\delta}$ .

Recall  $\delta = \varepsilon/2n$ .

# Running time

**Thm.:** The alg. is an FPTAS.

Running time:

$$O\left(\sum_{k=1}^n |L'_k|\right).$$

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return last( $L'_n$ )

Claim:  $|L'_k| = O\left(\frac{n \log t}{\varepsilon}\right)$ .

Let  $L'_k = [0, s_0, s_1, \dots, s_m]$ . Then

$$t \geq s_m > (1 + \delta)s_{m-1} > \dots > (1 + \delta)^m s_0 \geq (1 + \delta)^m.$$

$$\text{So } m < \log_{1+\delta} t = \frac{\ln t}{\ln(1+\delta)}.$$

CLRS eq. (3.17): if  $\delta > -1$ :  $\delta \geq \ln(1 + \delta) \geq \frac{\delta}{1+\delta}$ .

$$\text{So } m < \frac{\ln t}{\ln(1+\delta)} \leq \frac{\ln t}{\frac{\delta}{1+\delta}} = \frac{(1+\delta) \ln t}{\delta} \leq \frac{2 \ln t}{\delta} = \frac{4n \ln t}{\varepsilon}.$$

Recall  $\delta = \varepsilon/2n$ .

# Running time

**Thm.:** The alg. is an FPTAS.

Running time:

$$O\left(\sum_{k=1}^n |L'_k|\right).$$

APPROX-SUBSET-SUM( $S, t, \varepsilon$ )

$L'_0 = [0]$

for  $k = 1, \dots, n$

$L'_k = \text{MERGE-LISTS}(L'_{k-1}, L'_{k-1} + x_k)$

$L'_k = \text{TRIM}(L'_k, \varepsilon/2n)$

remove duplicates and elm.s  $> t$

return last( $L'_n$ )

Claim:  $|L'_k| = O\left(\frac{n \log t}{\varepsilon}\right)$ .

Let  $L'_k = [0, s_0, s_1, \dots, s_m]$ . Then

$t \geq s_m > (1 + \delta)s_{m-1} > \dots > (1 + \delta)^m s_0 \geq (1 + \delta)^m$ .

So  $m < \log_{1+\delta} t = \frac{\ln t}{\ln(1+\delta)}$ .

CLRS eq. (3.17): if  $\delta > -1$ :  $\delta \geq \ln(1 + \delta) \geq \frac{\delta}{1+\delta}$ .

So  $m < \frac{\ln t}{\ln(1+\delta)} \leq \frac{\ln t}{\frac{\delta}{1+\delta}} = \frac{(1+\delta) \ln t}{\delta} \leq \frac{2 \ln t}{\delta} = \frac{4n \ln t}{\varepsilon}$ .

Total running time:  $O\left(\sum_{k=1}^n |L'_k|\right) = O\left(\frac{n^2 \ln t}{\varepsilon}\right)$ .

Recall  $\delta = \varepsilon/2n$ .