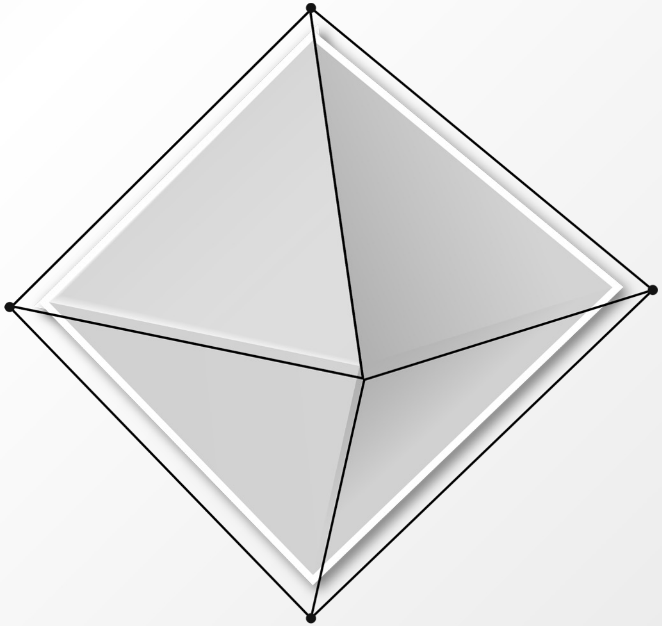
A Simple Aggregative Algorithm for Counting Triangulations of Planar Point Sets

Lorin Urbantat

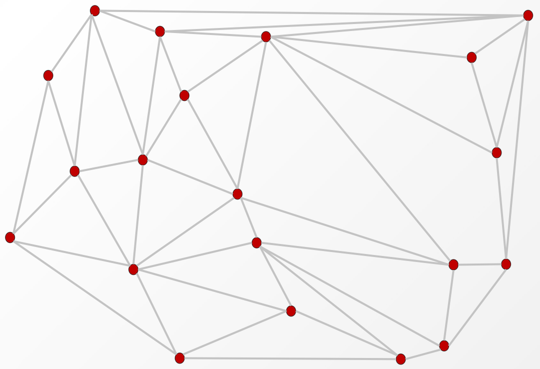
TABLE OF CONTENTS

01
Motivation & Context



Objective

Count number of Triangulations



Algorithm runs in $O(n^2 2^n)$ time and $O(n2^n)$ space

First algorithm to be provably faster than enumeration $O(2.43^n)$

Can also compute optimal Triangulation, and generate Triangulations uniformly at random

Context

Paper published in 2013 by Victor Alvarez and Raimund Seidel

Was the first algorithm to achieve counting number of triangulations faster then enumeration

Dániel Marx and Tillmann Miltzow since achieved counting triangulations subexponentially in $O(n^{(11+O(1))\sqrt{n}})$, 2016

It's unlikely that a polynomial time counting method will be found because related problems are NP-hard 1, 2

Related Work: Counting triangulations and other crossing-free structures approximately, Victor Alvarez, Karl Bringmann, Saurabh Ray, Raimund Seidel, 2014

Applications

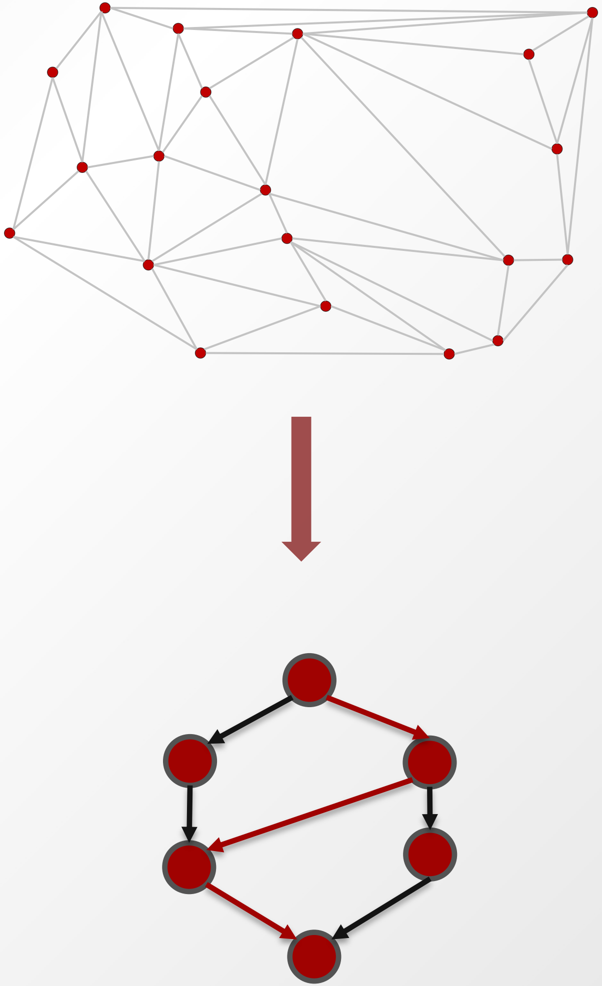
Computer Graphics, Geo-information-Systems, etc.

02
Conceptual
Overview

Idea

1) Create an Isomorphism from triangulations to source sink paths in a DAG

2) Count number of source-sink paths and thus triangulations and hope that we’ll accomplish resource bounds



03

Detailed Walkthrough

Setup

We consider a set of n points in the plane $S = \{p_1, p_1, \dots, p_n\}$

Assumptions

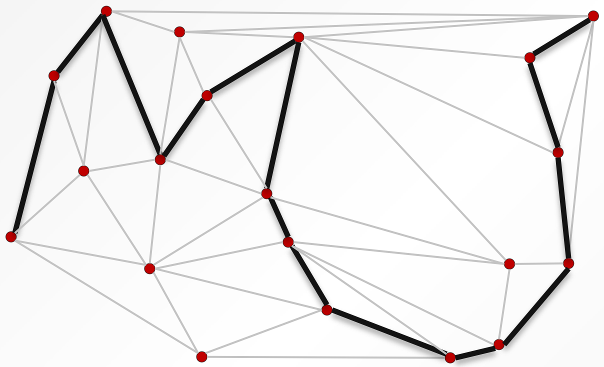
No 3 points in S lie on a straight line

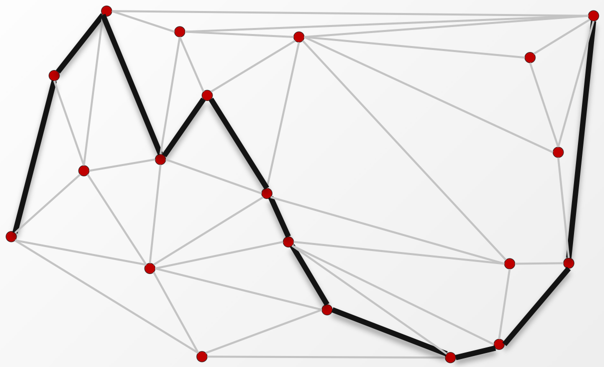
No 2 points lie on a common vertical line → Points in S can be sorted by x-coordinate

Monotone Chain

Definition

A monotone chain C for S is a polygonal chain that connects p_1 with p_n , contains only points of S as vertices and intersects every vertical line at most once

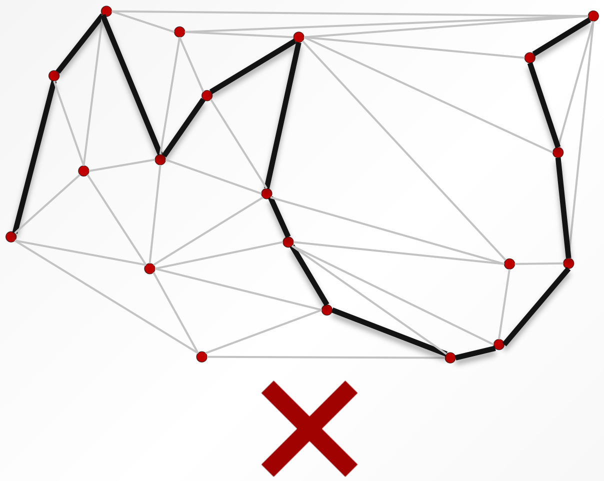




Monotone Chain

Definition

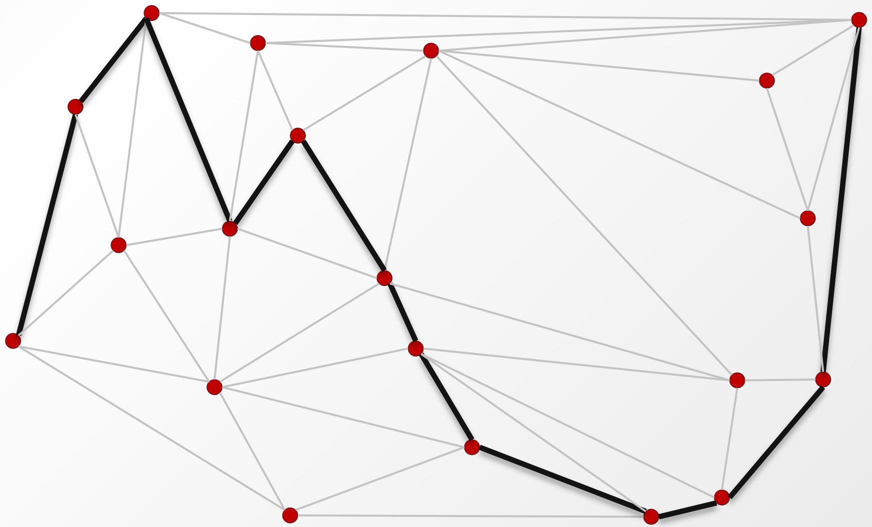
A monotone chain C for S is a polygonal chain that connects p_1 with p_n , contains only points of S as vertices and intersects every vertical line at most once



Advancing Monotone Chain

Definition

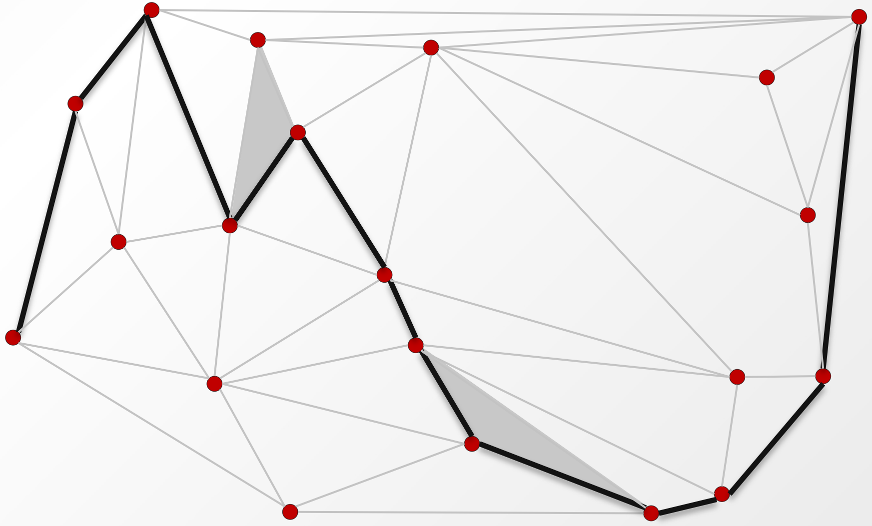
We call a triangle T an advance for the monotone chain C if it touches C from above and if we add T to the set of triangles below C we get a new monotone chain



Advancing Monotone Chain

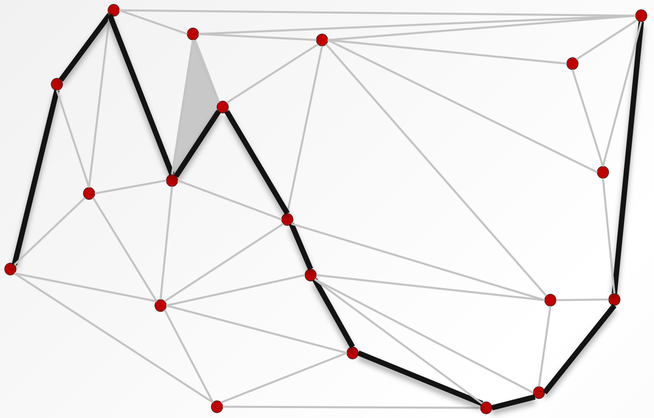
Definition

We call a triangle T an advance for the monotone chain C if it touches C from above and if we add T to the set of triangles below C we get a new monotone chain

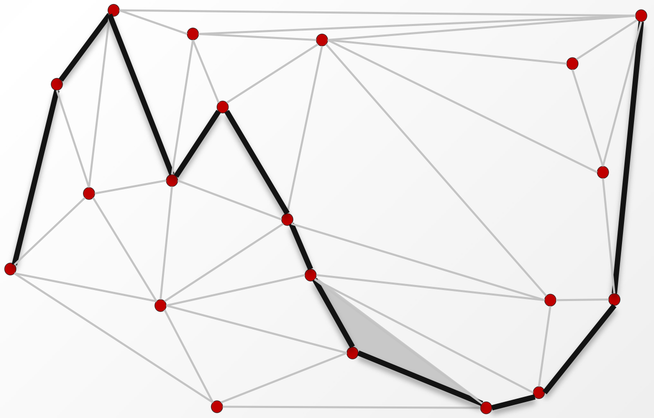


Advancing Monotone Chain

Case 1



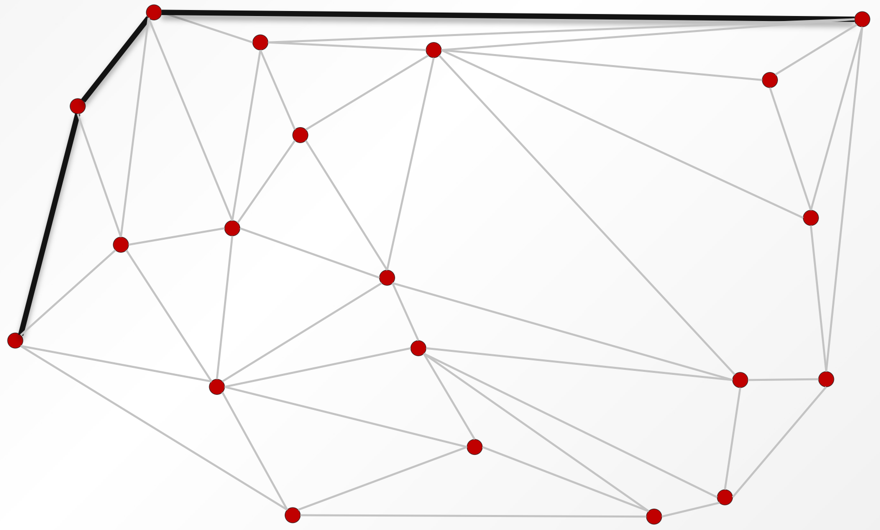
Case 2



Can you always advance?

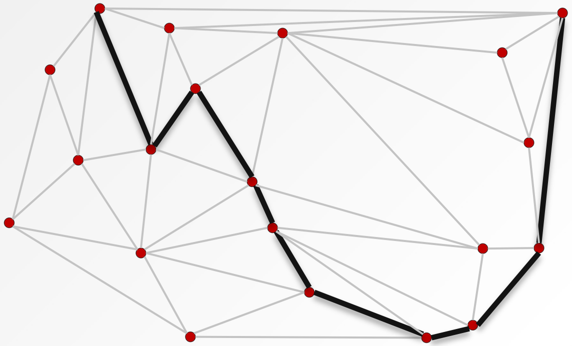
Can you always advance?

There always is an advance unless C contains only upper hull edges

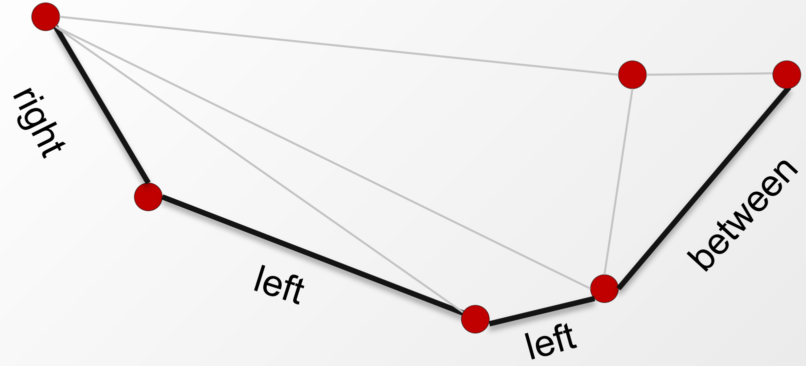


Can you always advance?

sub-chain containing no upper hull edges

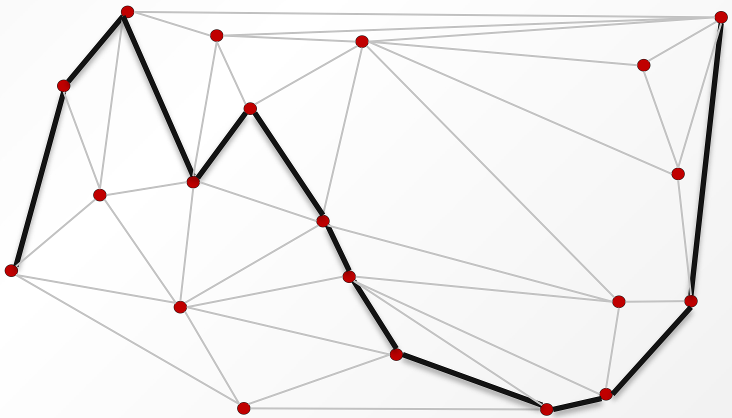


Zoomed in



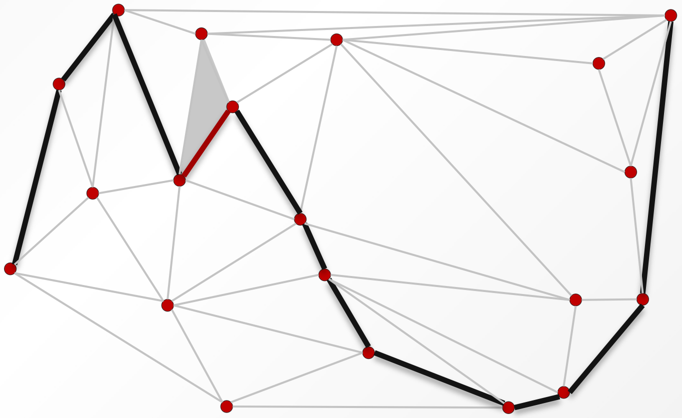
Unique Leftmost Advance

Every montone chain has a unique leftmost advance



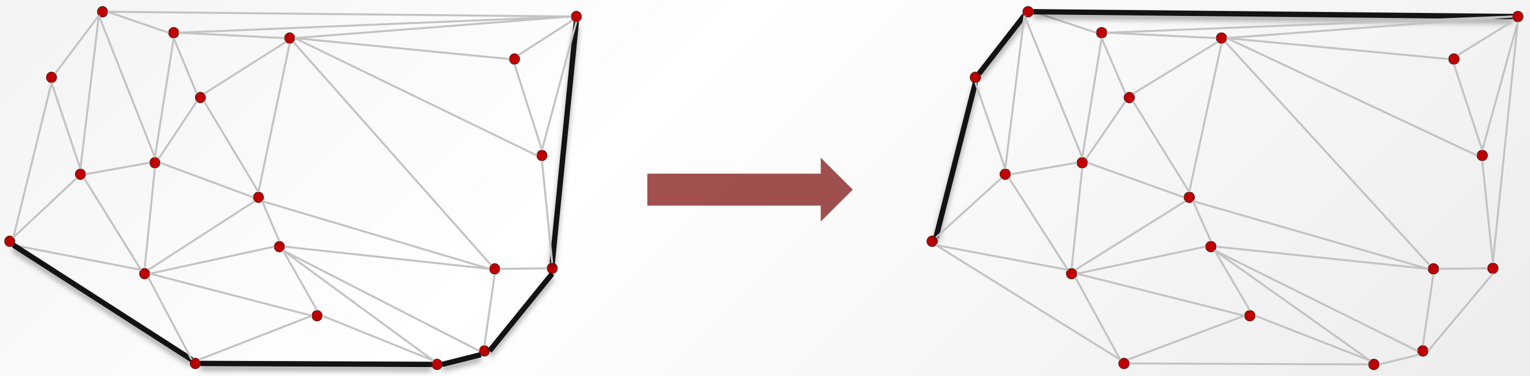
Unique Leftmost Advance

Every montone chain has a unique leftmost advance



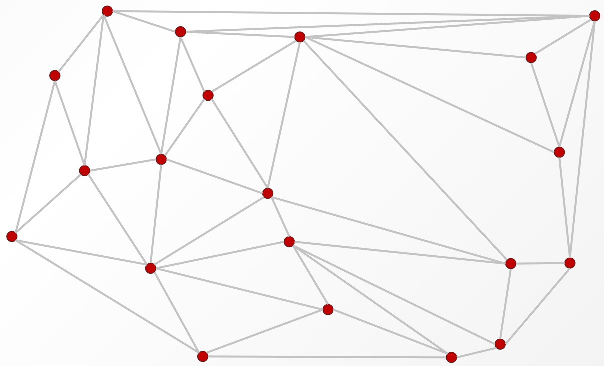
Leftmost sweeping Advance

There is a unique sequence $C_0, \ldots, C_M$ for each triangulation

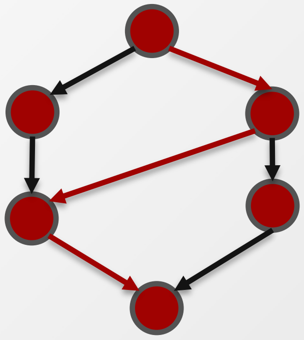


Idea Revisited

1) Create an Isomorphism from triangulations to source sink paths in a DAG



2) Count number of source-sink paths and thus triangulations and hope that we'll accomplish resource bounds



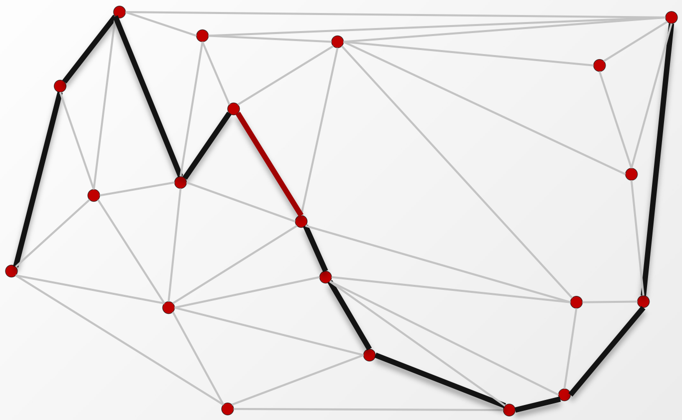
Constructing the DAG

Nodes

are *marked monotone chains* (C,*k*) where C is a monotone chain with its *kth* edge marked

Edges

next, we define the successor relation on the marked monotone chains

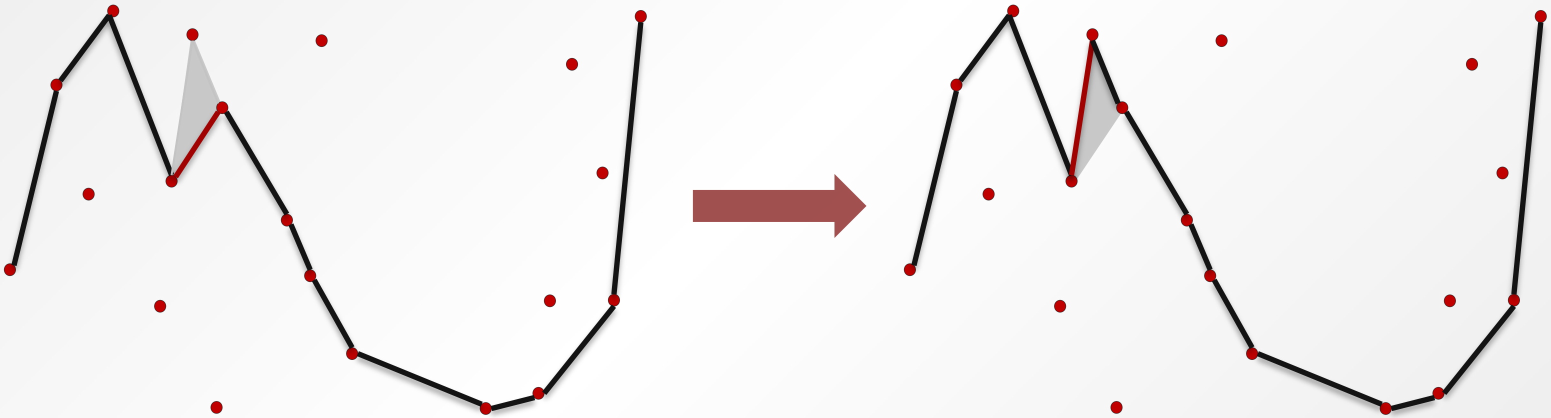


(C,5)

Successor Relation

Case 1

Advancing increases length by 1

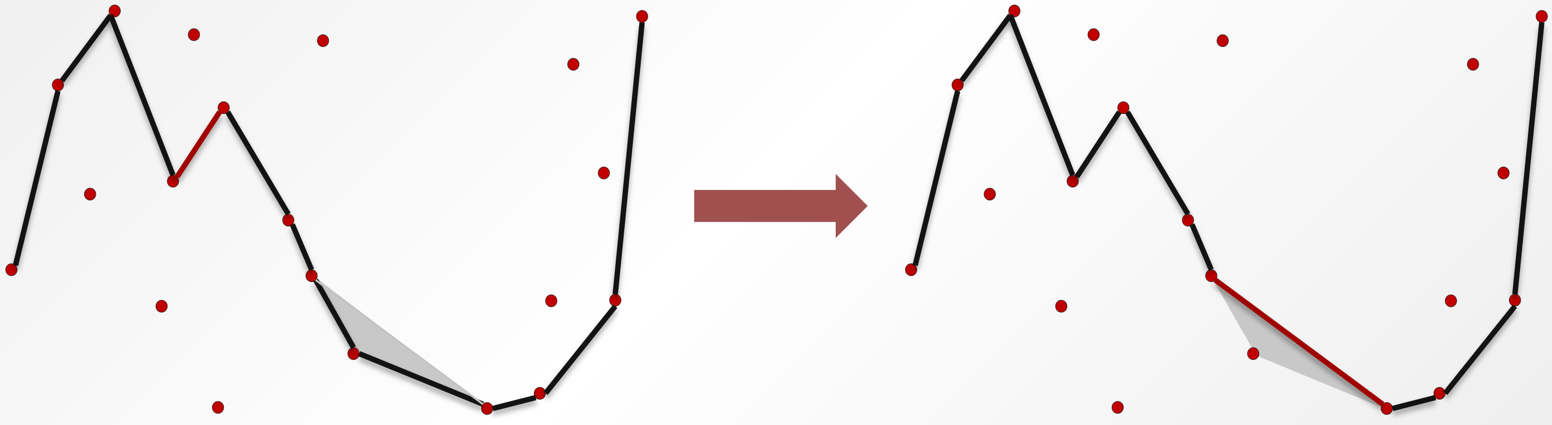


(C,4) (C′,4)

Successor Relation

Case 2

Advancing decreases length by 1

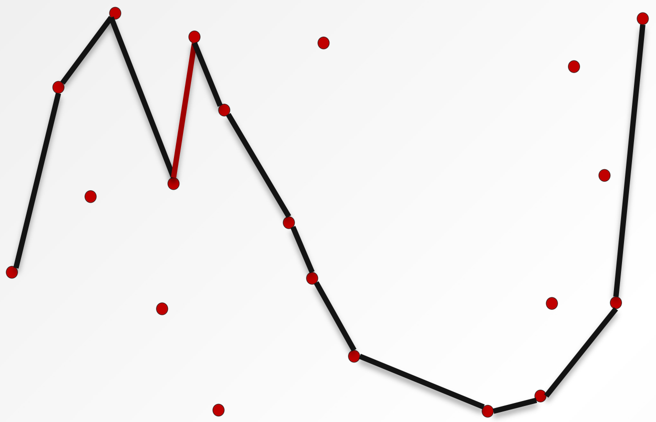


(C,4)

(C”,7)

Successor Relation

Another Example

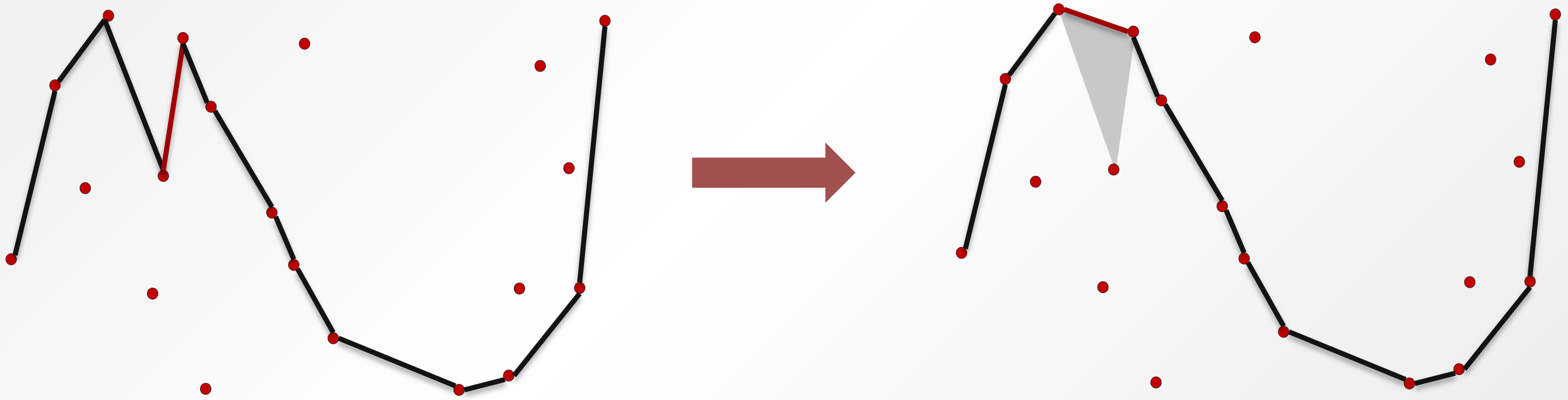


(C’,4)

?

Successor Relation

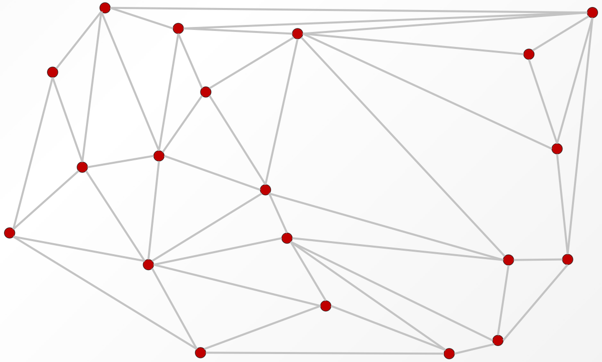
Another Example

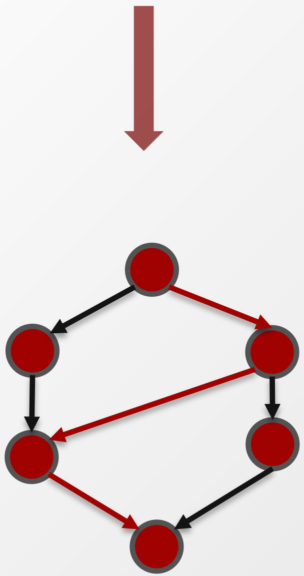


(C’,4) *(C’’’,3)*

Idea

1) Create an Isomorphism from triangulations to source sink paths in a DAG





2) Count number of source-sink paths and thus triangulations and hope that we'll accomplish resource bounds

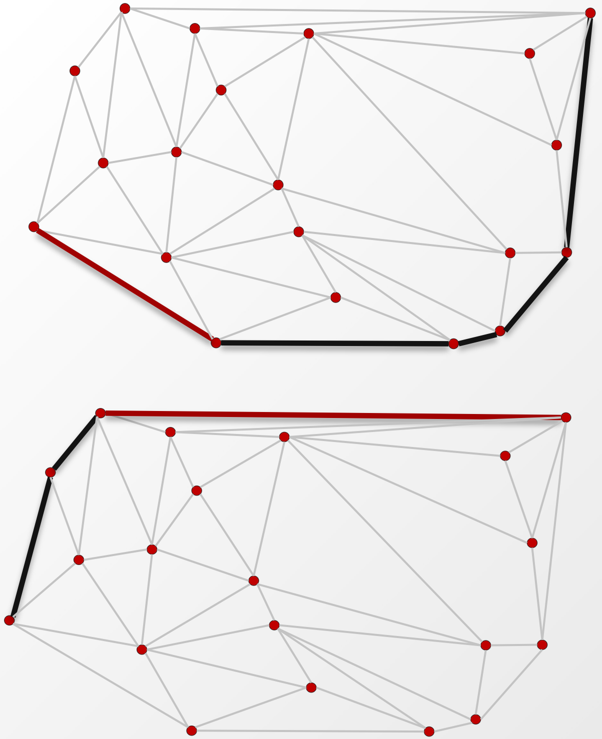
Counting Triangulations

Set *(B,1)* as source

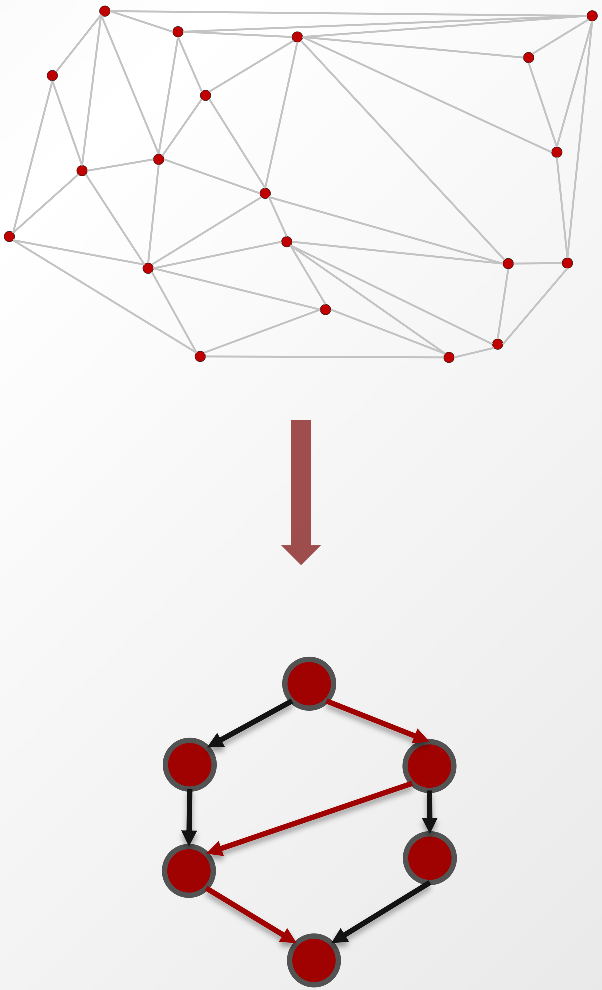
Where *B* is the lower hull chain

Create Node *T* set it as sink

Connect *T* to every Node *(U,k)*, where *U* is the upper hull chain and *k* is any number



Idea



1) Create an Isomorphism from triangulations to source sink paths in a DAG

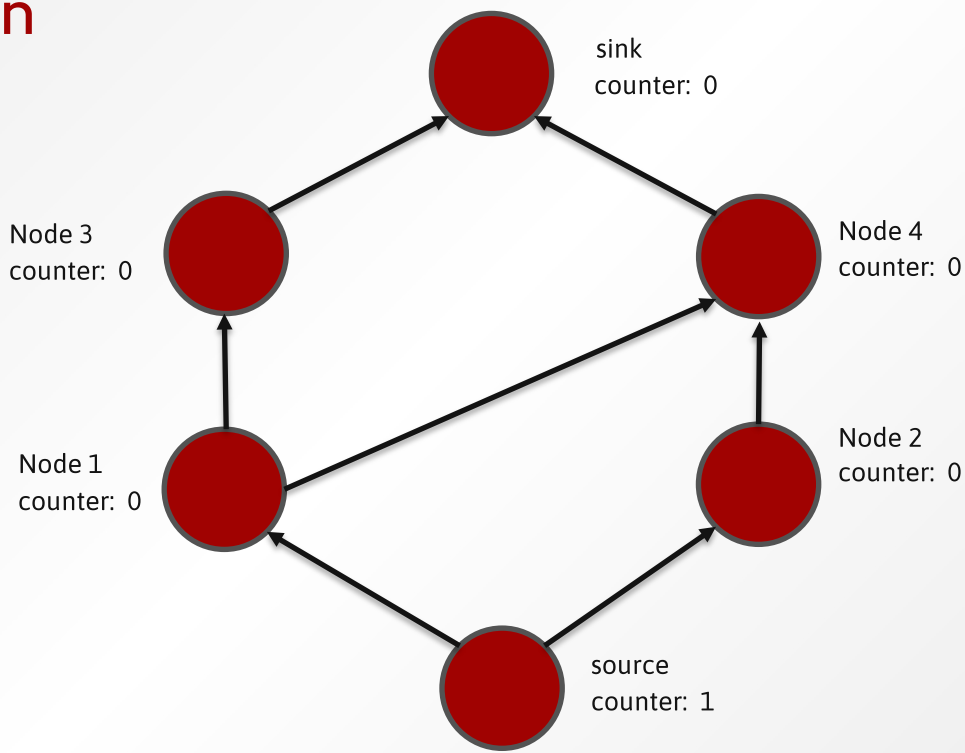
✓

2) Count number of source-sink paths and thus triangulations and hope that we'll accomplish resource bounds

✓

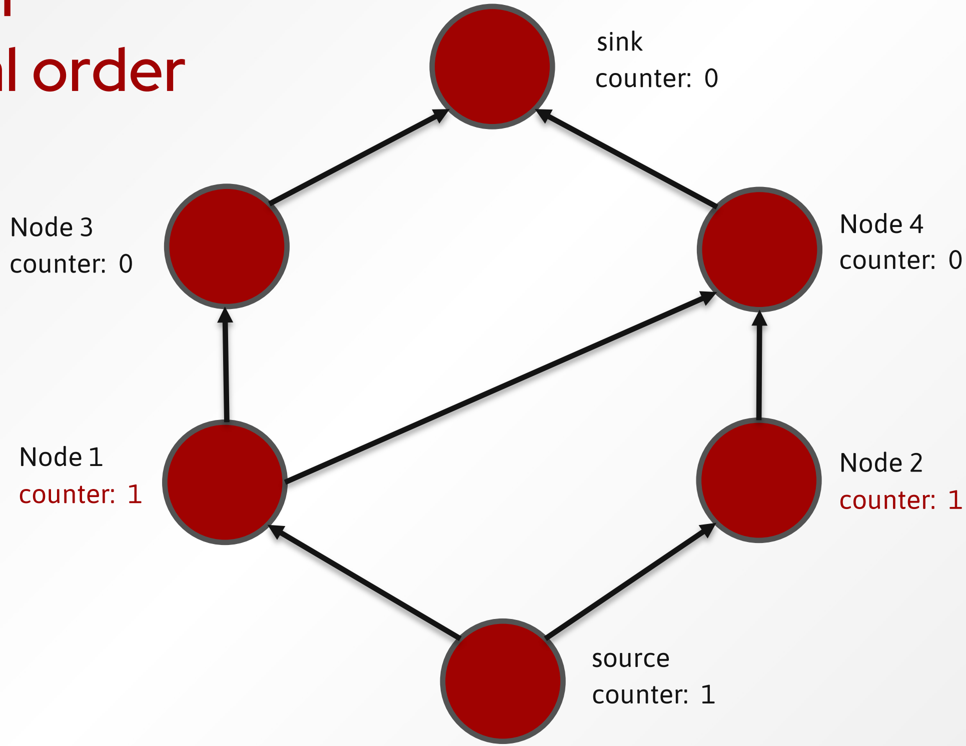
Counting in Detail

Initialization



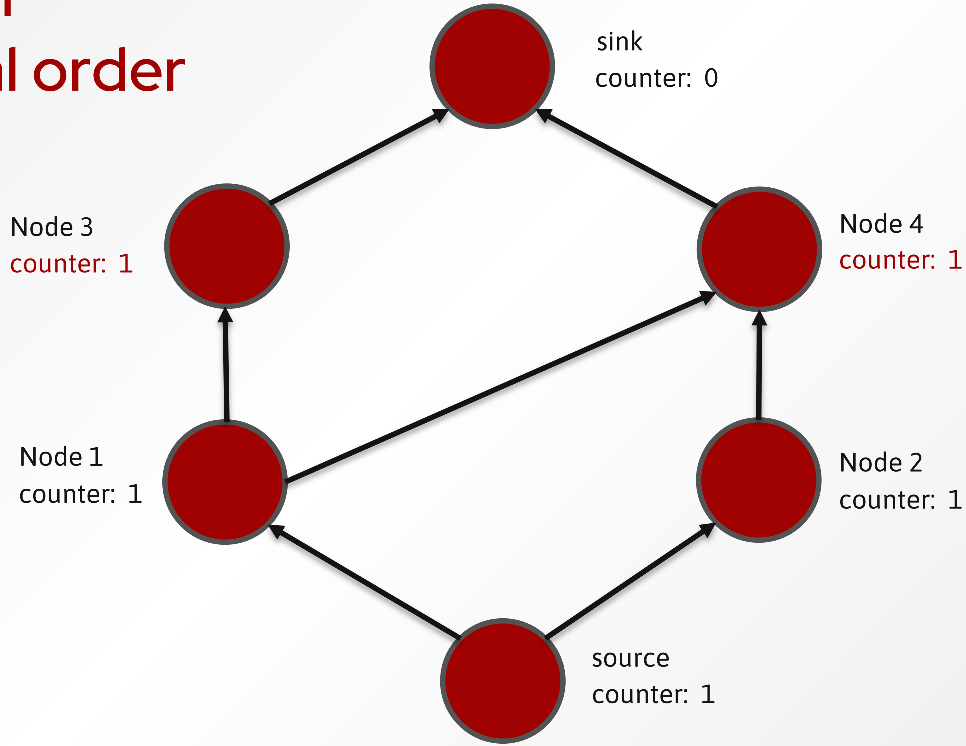
Counting in Detail

Traverse in topological order



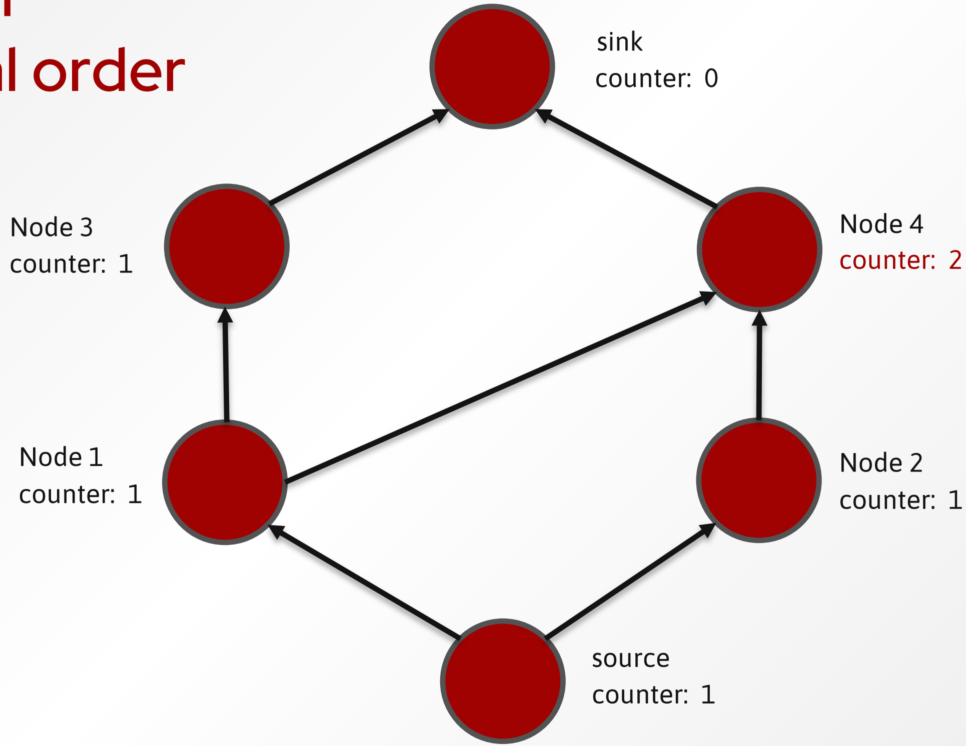
Counting in Detail

Traverse in topological order



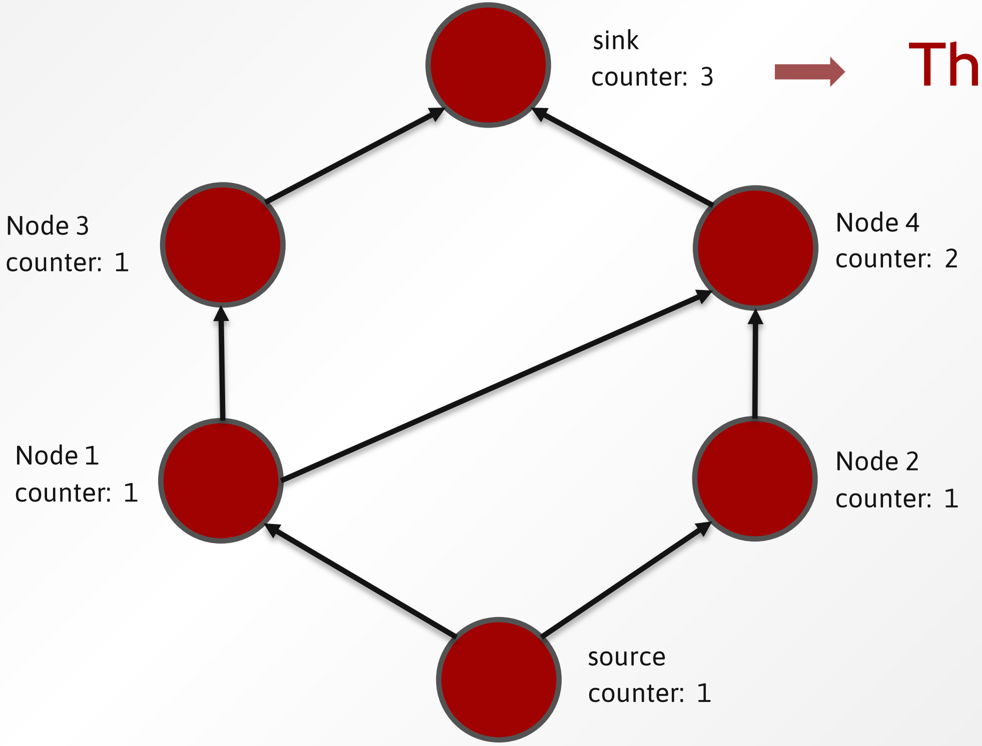
Counting in Detail

Traverse in topological order



Counting in Detail

Done



There are 3 paths

Resource bounds

Time

The time is proportional to the number of edges in the Graph

Space

We must store a counter for each node

Why do we not need to store edges ?

Resource bounds

Number of Nodes

there can be no more than 2^n *monotone chains*

each monotone chain can have at most n-1 markings

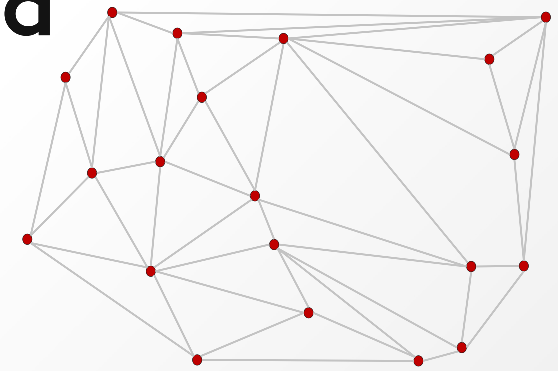
$O(n2^n)$ number of Nodes

Number of edges

each node can have at most n successors

$O(n^2 2^n)$ number of edges

Objective Revisited



Count number of Triangulations

Algorithm runs in $O(n^2 2^n)$ time and $O(n 2^n)$ space

First algorithm to be provably faster than enumeration $O(2.43^n)$

Can also compute optimal Triangulation, and generate Triangulations uniformly at random

04

Generalizations

Generating Triangulations uniformly at random

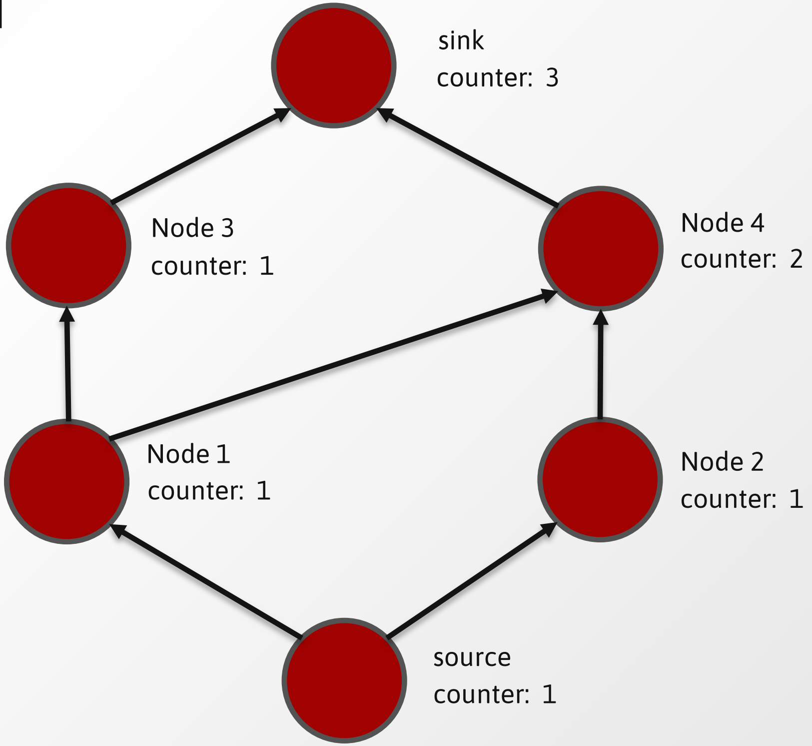
Algorithm

compute the Graph

remove nodes unreachable from source or sink

choose a random number *r* between 1 and number of triangulations

traverse the graph backwards subtracting the count from *r*



Generating Triangulations uniformly at random

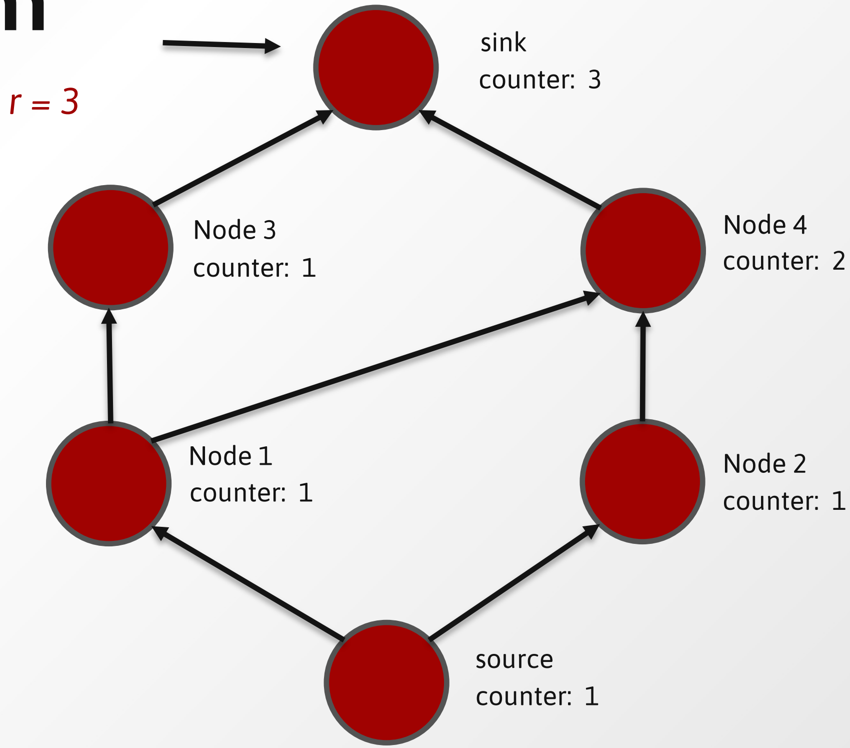
Algorithm

compute the Graph

remove nodes unreachable from source or sink

choose a random number *r* between 1 and number of triangulations

traverse the graph backwards subtracting the count from *r*



Generating Triangulations uniformly at random

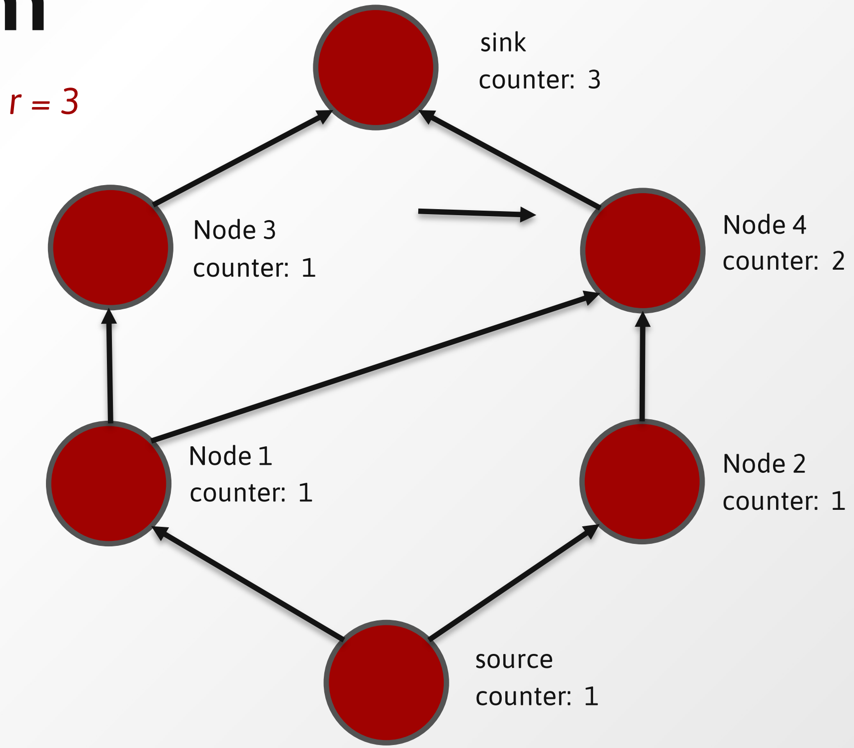
Algorithm

compute the Graph

remove nodes unreachable from source or sink

choose a random number *r* between 1 and number of triangulations

traverse the graph backwards subtracting the count from *r*



Generating Triangulations uniformly at random

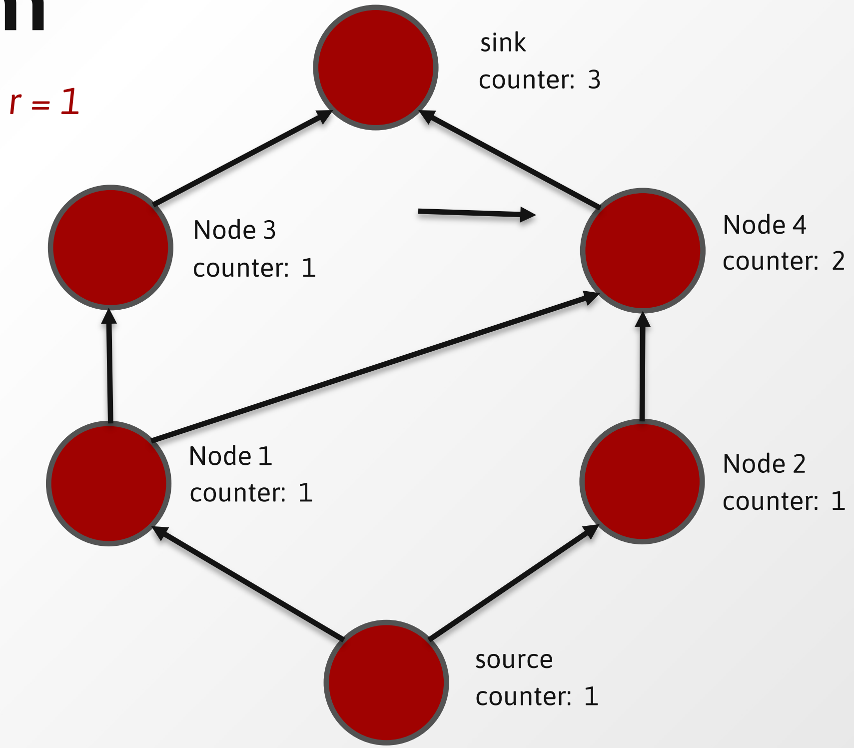
Algorithm

compute the Graph

remove nodes unreachable from source or sink

choose a random number *r* between 1 and number of triangulations

traverse the graph backwards subtracting the count from *r*



Generating Triangulations uniformly at random

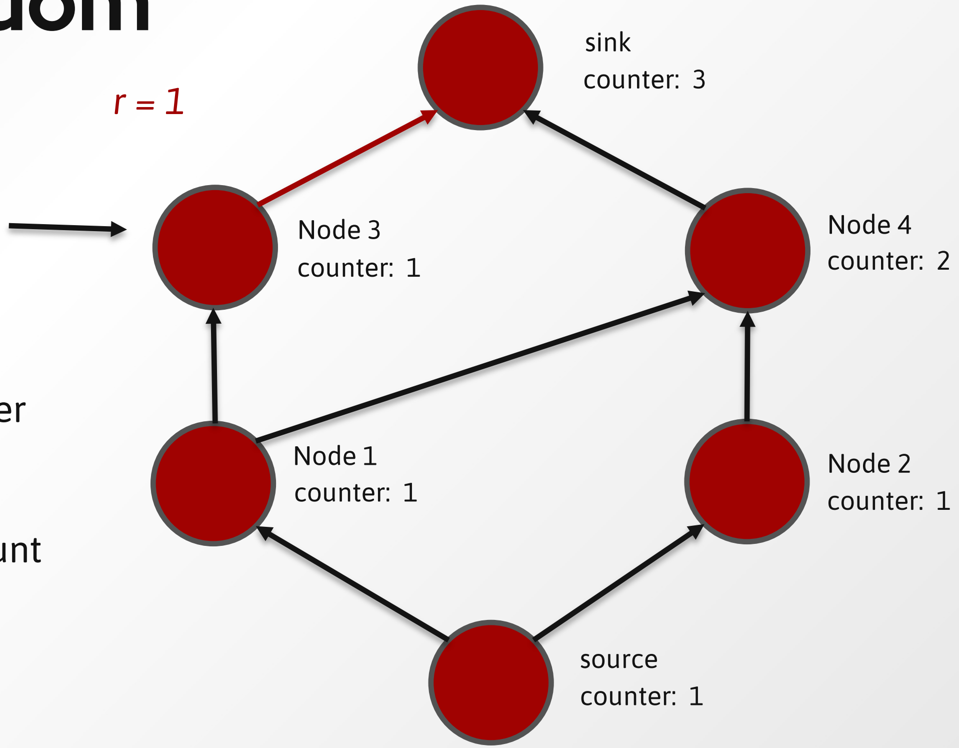
Algorithm

compute the Graph

remove nodes unreachable from source or sink

choose a random number *r* between 1 and number of triangulations

traverse the graph backwards subtracting the count from *r*



Generating Triangulations uniformly at random

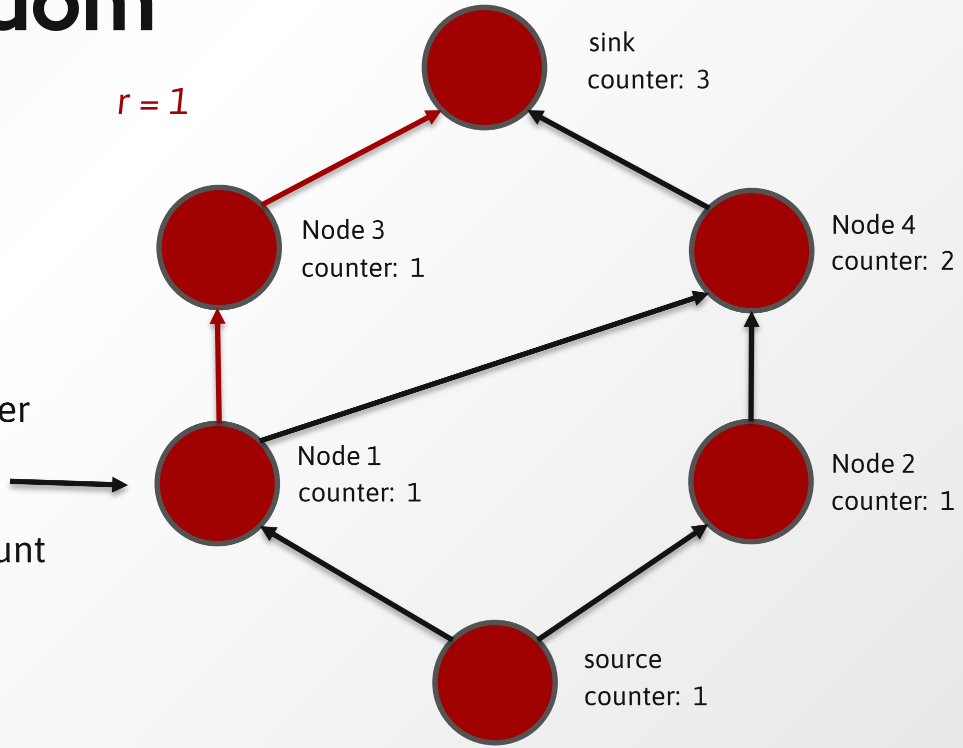
Algorithm

compute the Graph

remove nodes unreachable from source or sink

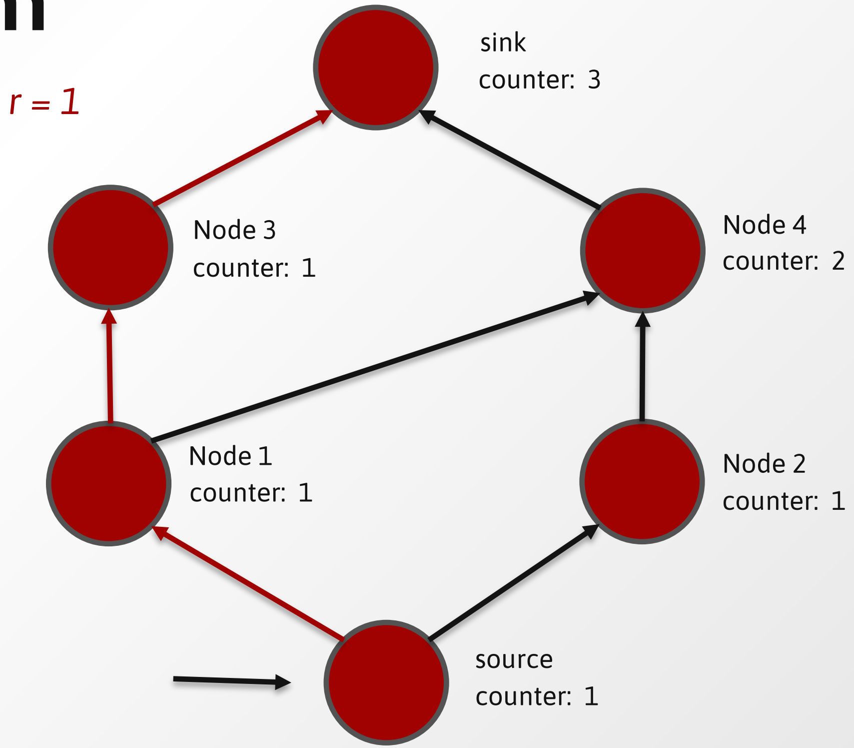
choose a random number *r* between 1 and number of triangulations

traverse the graph backwards subtracting the count from *r*



Generating Triangulations uniformly at random

Done



Generating Triangulations uniformly at random

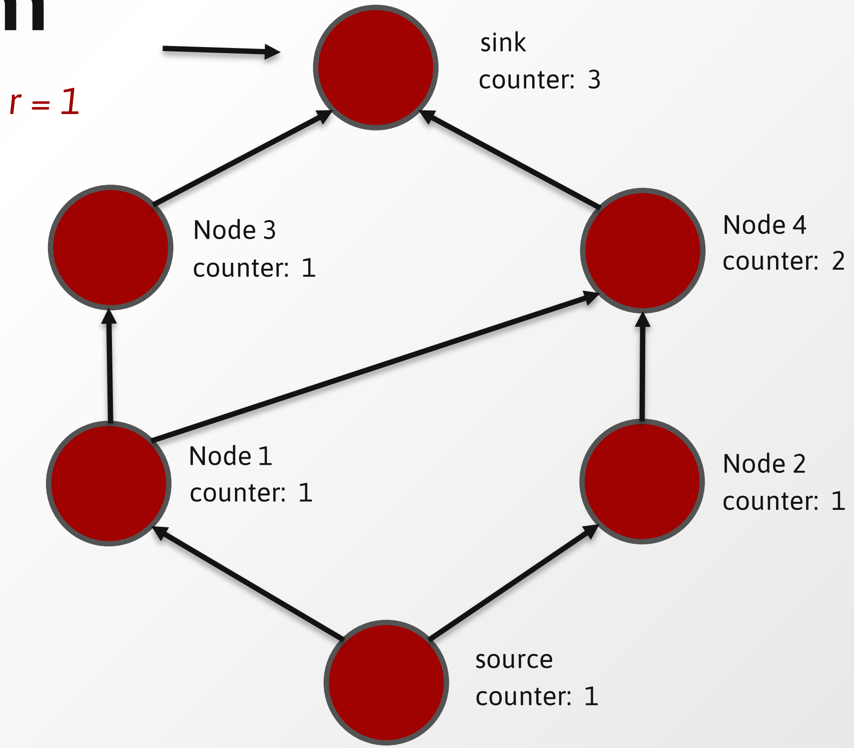
Algorithm

compute the Graph

remove nodes unreachable from source or sink

choose a random number *r* between 1 and number of triangulations

traverse the graph backwards subtracting the count from *r*



Generating Triangulations uniformly at random

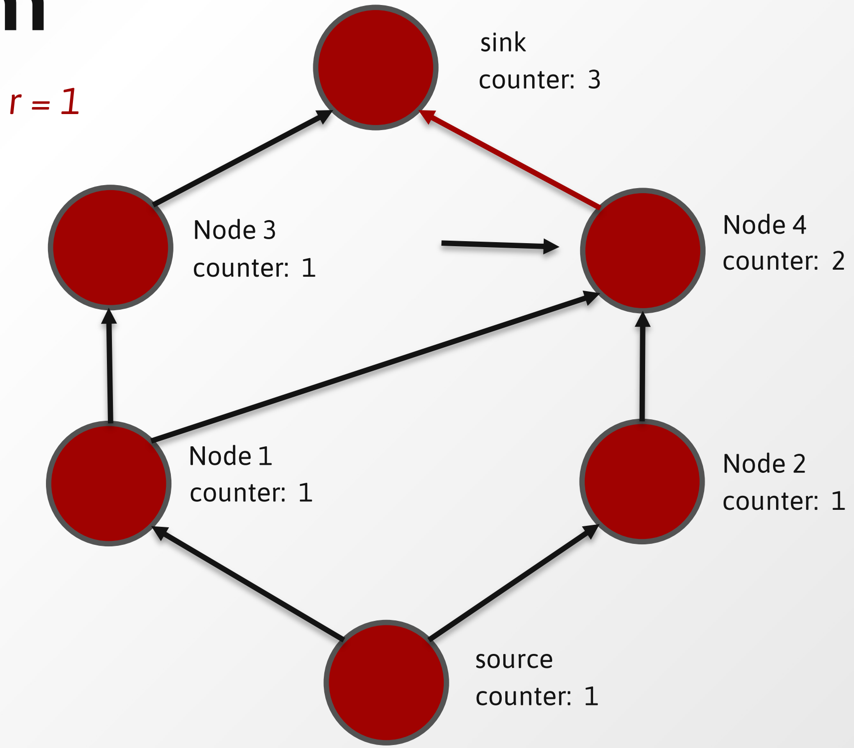
Algorithm

compute the Graph

remove nodes unreachable from source or sink

choose a random number *r* between 1 and number of triangulations

traverse the graph backwards subtracting the count from *r*



Generating Triangulations uniformly at random

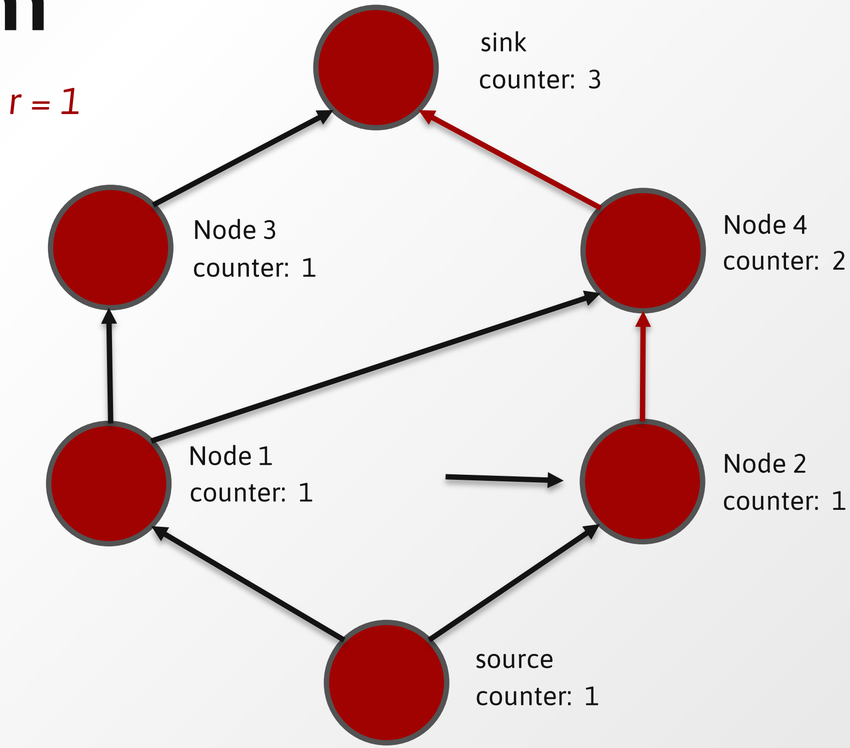
Algorithm

compute the Graph

remove nodes unreachable from source or sink

choose a random number *r* between 1 and number of triangulations

traverse the graph backwards subtracting the count from *r*



Generating Triangulations uniformly at random

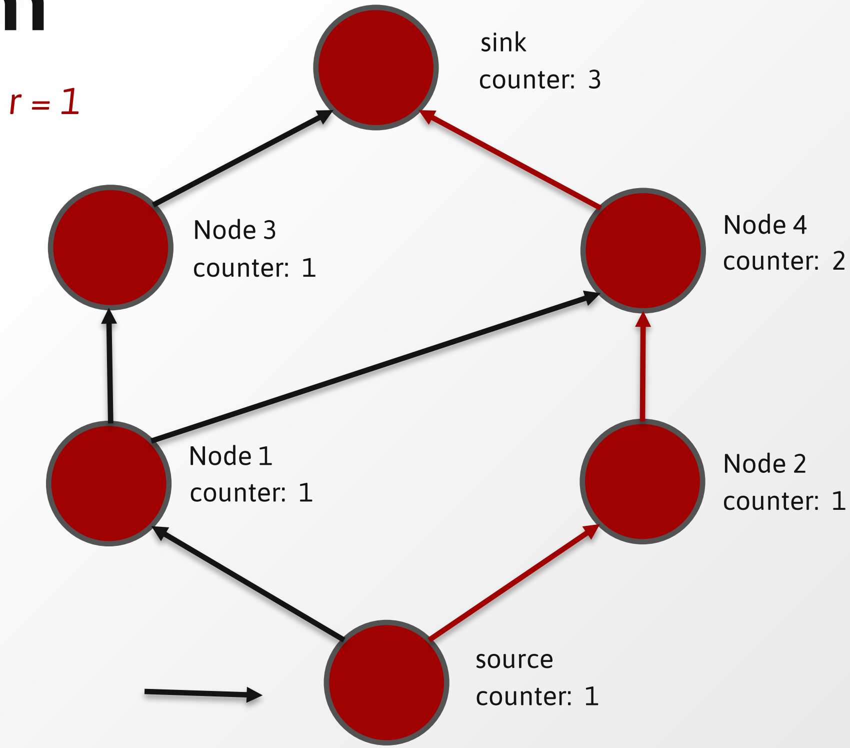
Algorithm

compute the Graph

remove nodes unreachable from source or sink

choose a random number *r* between 1 and number of triangulations

traverse the graph backwards subtracting the count from *r*



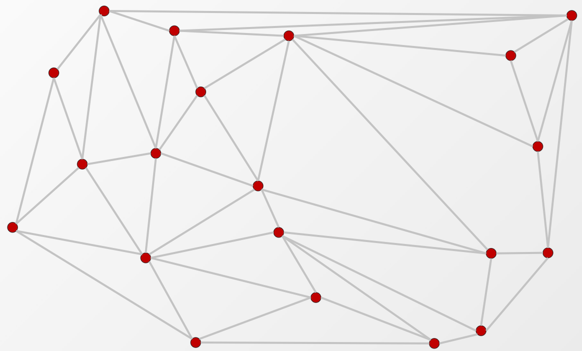
Find triangulation that fulfils specific optimality criteria

Limitations

let f be the function to be optimized. Let *D* be a set of points and *t* be a triangle. Then the optimum f over *D U t* must be obtained from the optimum of *D* and *t*

Example – minimize triangle weights

When traversing the Graph choose advance that minimally increases triangle weights



THANKS!

Slides in PDF format *https://bruol.me/other/agp-presentation.pdf*

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon** and infographics & images by **Freepik**