Random Spanning Forests on Graphs for Fast Laplacian-Based Computations

Nicolas Tremblay

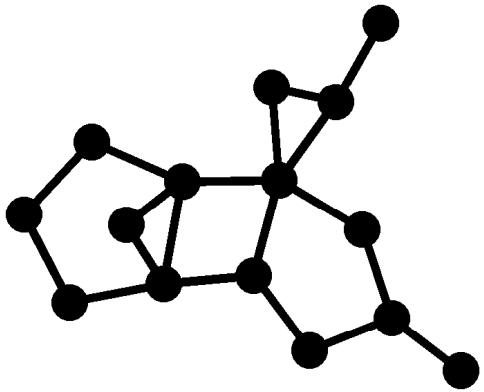
(joint work with P.-O. Amblard, L. Avena, S. Barthelmé, A. Gaudillière, Y. Pilavci)

CNRS, GIPSA-lab, Univ. Grenoble-Alpes, Grenoble-INP, France

Contents

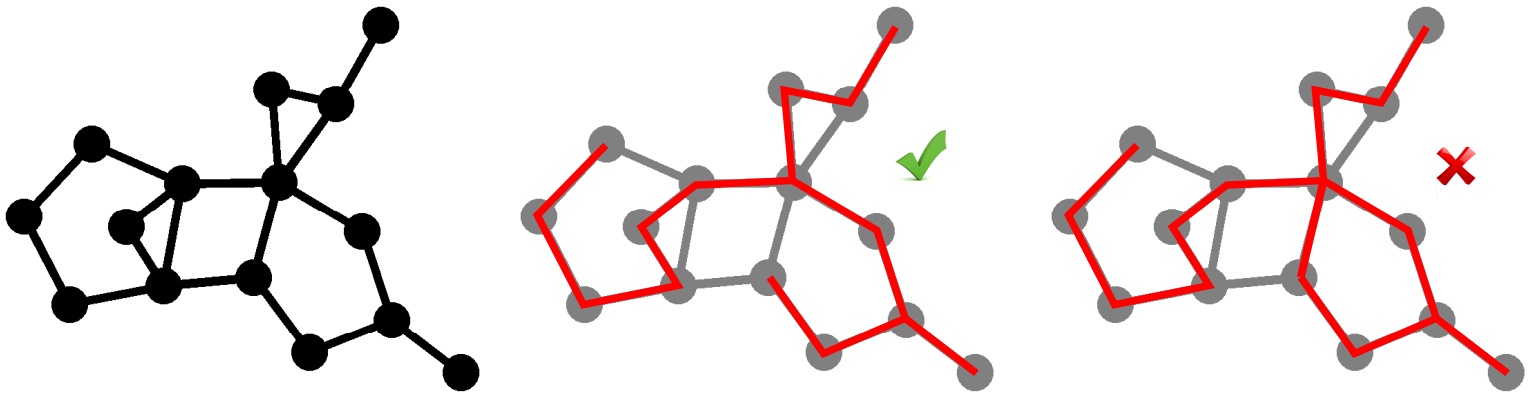
DPPs and USTs

- Consider a connected graph with n nodes and m edges:



DPPs and USTs

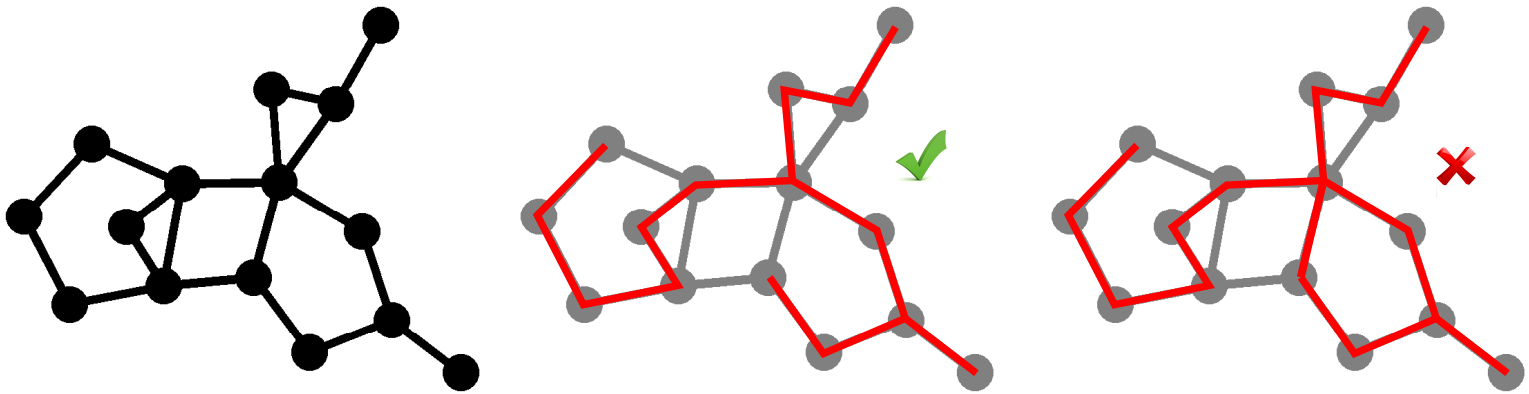
- Consider a connected graph with n nodes and m edges:



- A spanning tree is a subset of edges that i/ are connected, ii/ do not contain cycles, iii/ span all nodes.

DPPs and USTs

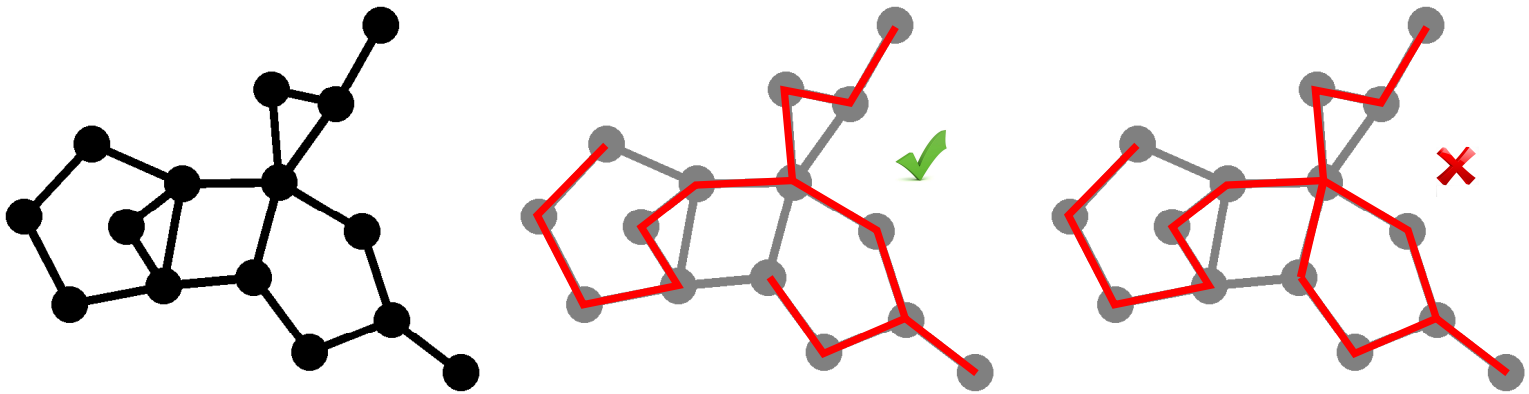
- Consider a connected graph with n nodes and m edges:



- A spanning tree is a subset of edges that i/ are connected, ii/ do not contain cycles, iii/ span all nodes.
- A graph of size n contains between 1 and n^{n-2} different spanning trees

DPPs and USTs

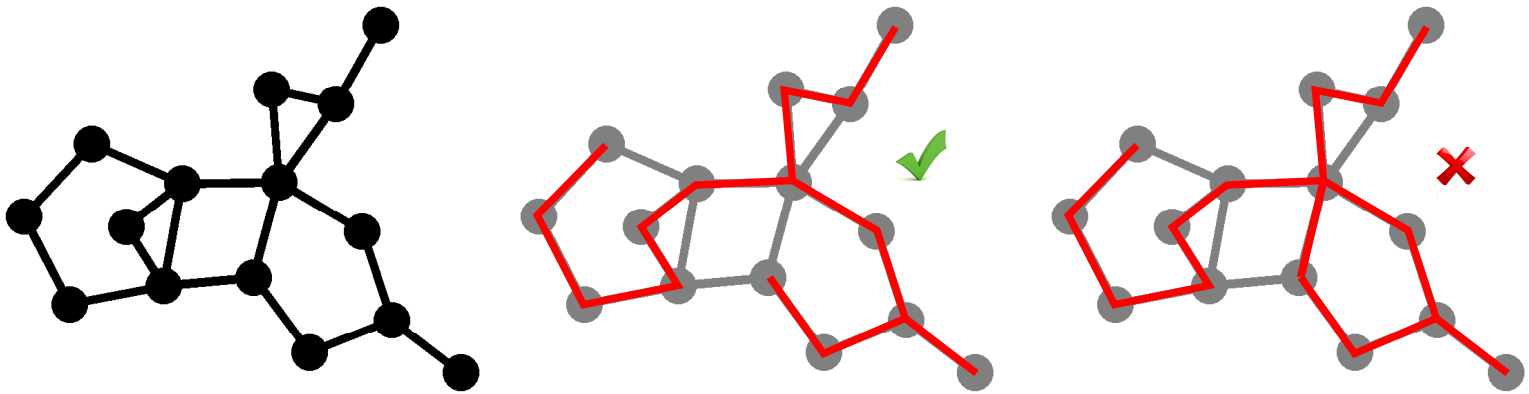
- Consider a connected graph with n nodes and m edges:



- A spanning tree is a subset of edges that i/ are connected, ii/ do not contain cycles, iii/ span all nodes.
- A graph of size n contains between 1 and n^{n-2} different spanning trees
- A uniform spanning tree (UST) is a spanning tree, drawn uniformly at random from the set of all spanning trees.

DPPs and USTs

- Consider a connected graph with n nodes and m edges:



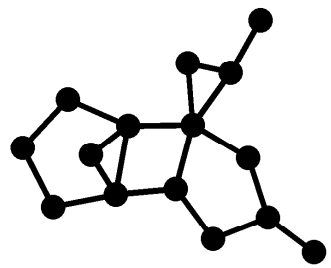
- A spanning tree is a subset of edges that i/ are connected, ii/ do not contain cycles, iii/ span all nodes.
- A graph of size n contains between 1 and n^{n-2} different spanning trees
- A uniform spanning tree (UST) is a spanning tree, drawn uniformly at random from the set of all spanning trees.
- Classical result: *A UST is a (projective) DPP defined over the set of edges*

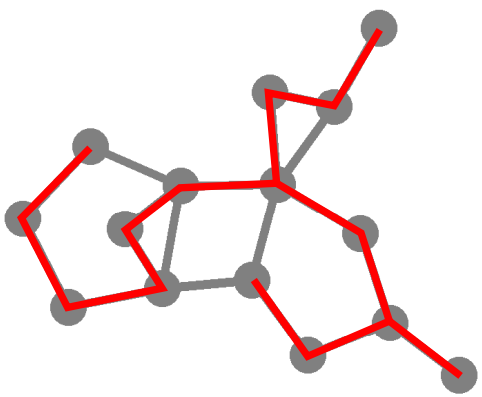
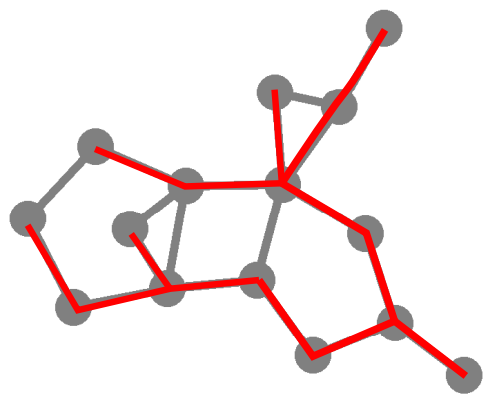
To give you the flavor of this talk

- DPPs can be understood as (weakly) repulsive summaries of a ground set of elements
- A general ML/SP question is: what information is kept by the sampling process? What can be estimated about the original set from this summary?

To give you the flavor of this talk

- DPPs can be understood as (weakly) repulsive summaries of a ground set of elements
- A general ML/SP question is: what information is kept by the sampling process? What can be estimated about the original set from this summary?
- Given that USTs (and their generalisation Random Spanning Forests) are DPPs, the question translates as:

What can we learn about a graph  given a collection of USTs

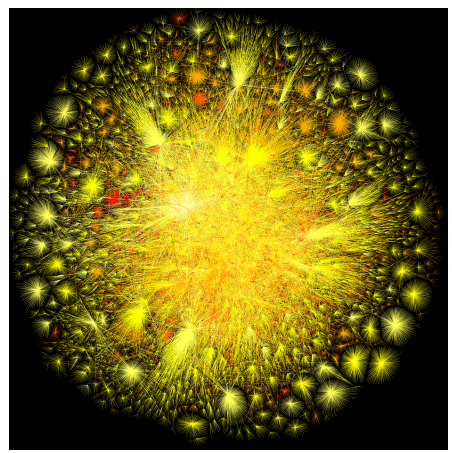

,

,. . . ?

(trivial example: lower bound on maximum degree)

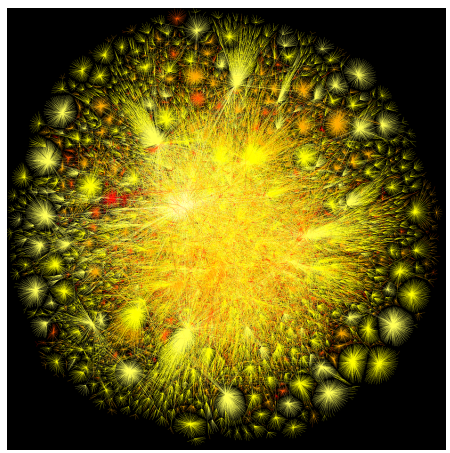
Contents

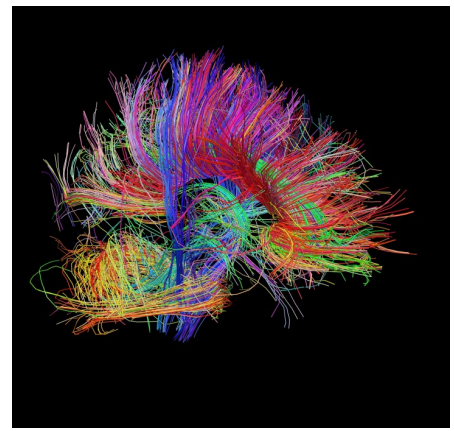
Graph-based data?

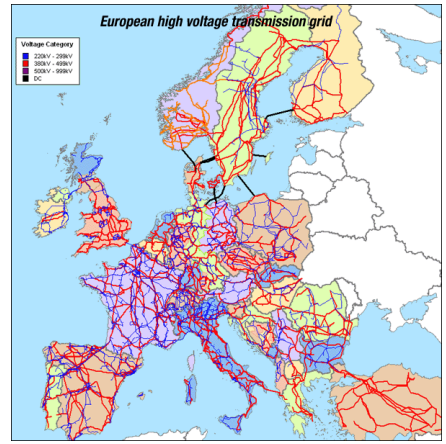
Graph-based data?



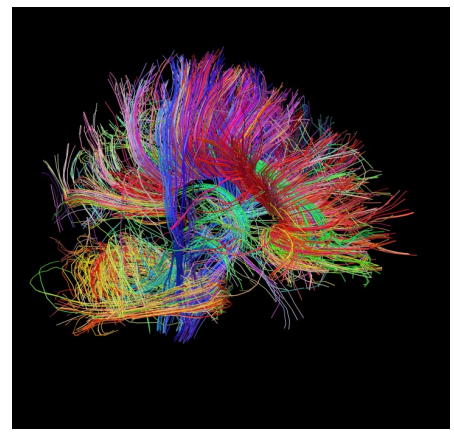
Graph-based data?

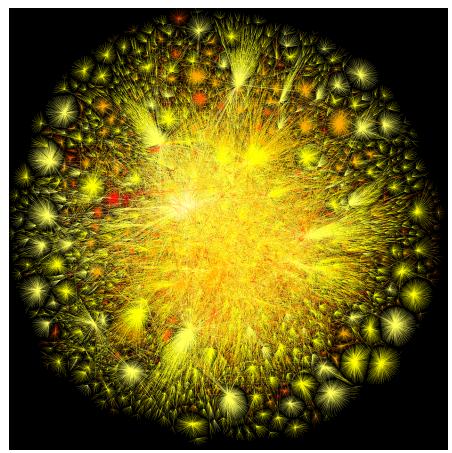


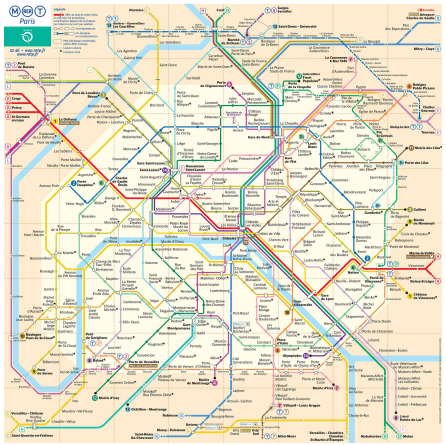


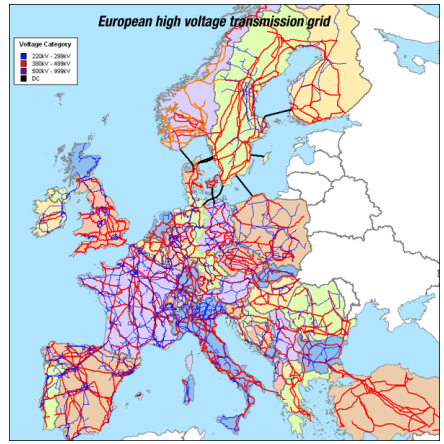


Graph-based data?

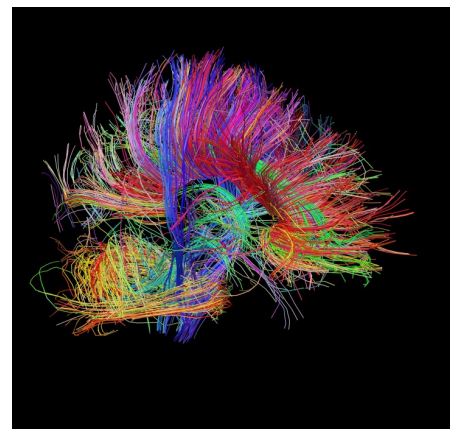


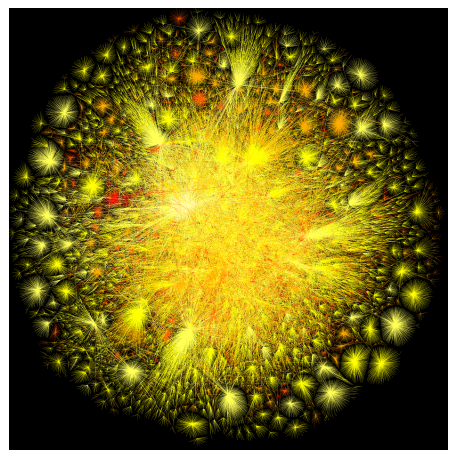


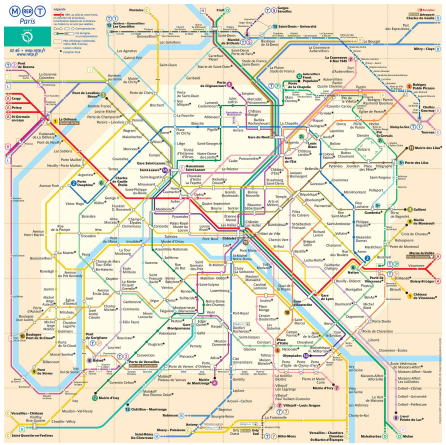


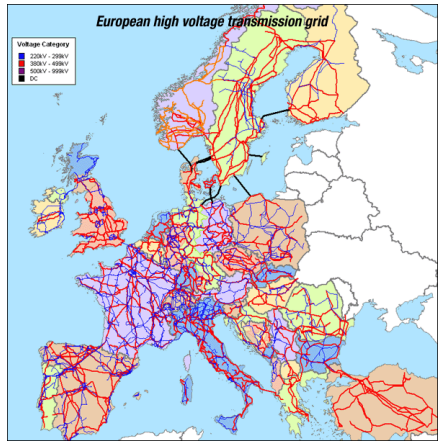


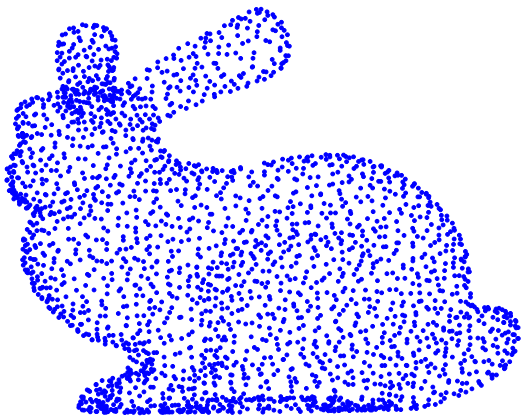
Graph-based data?



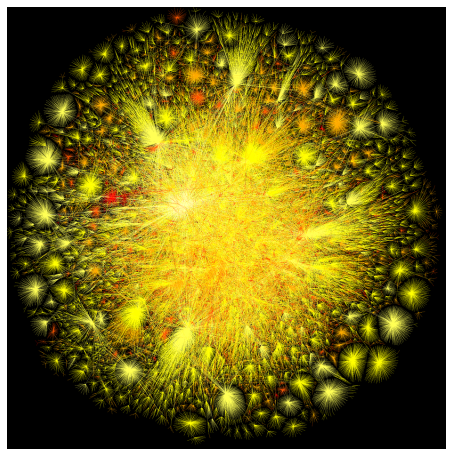


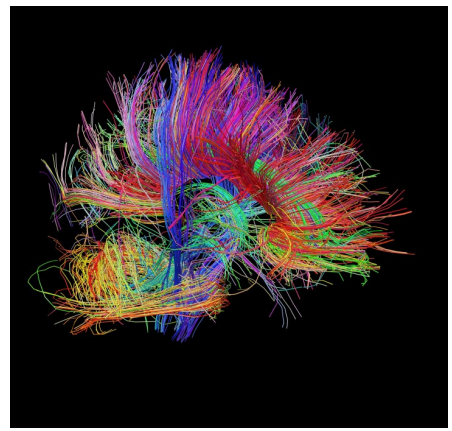






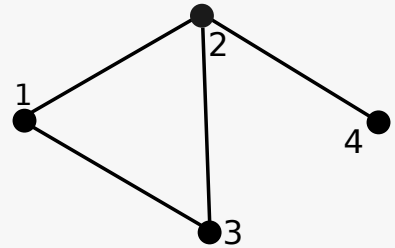
Graph-based data?





Graph-based data =

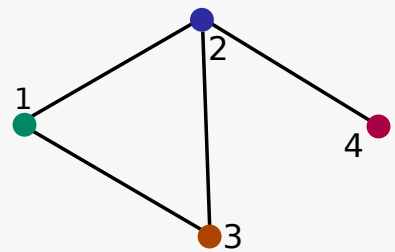
a connectivity structure



$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \qquad D = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad L = D - A = \begin{bmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 2 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

+

possibly a signal on the nodes
(and/or edges)



$$\boldsymbol{y} = [y_1, y_2, y_3, y_4]^\top \in \mathbb{R}^4$$

Different operations on graphs or signals defined on graphs

on graphs

- community detection, un-, semi-, supervised learning
- control of speed of diffusion (information, disease)
- graph simplification (sparsification, node merging, etc.)
- graph visualization

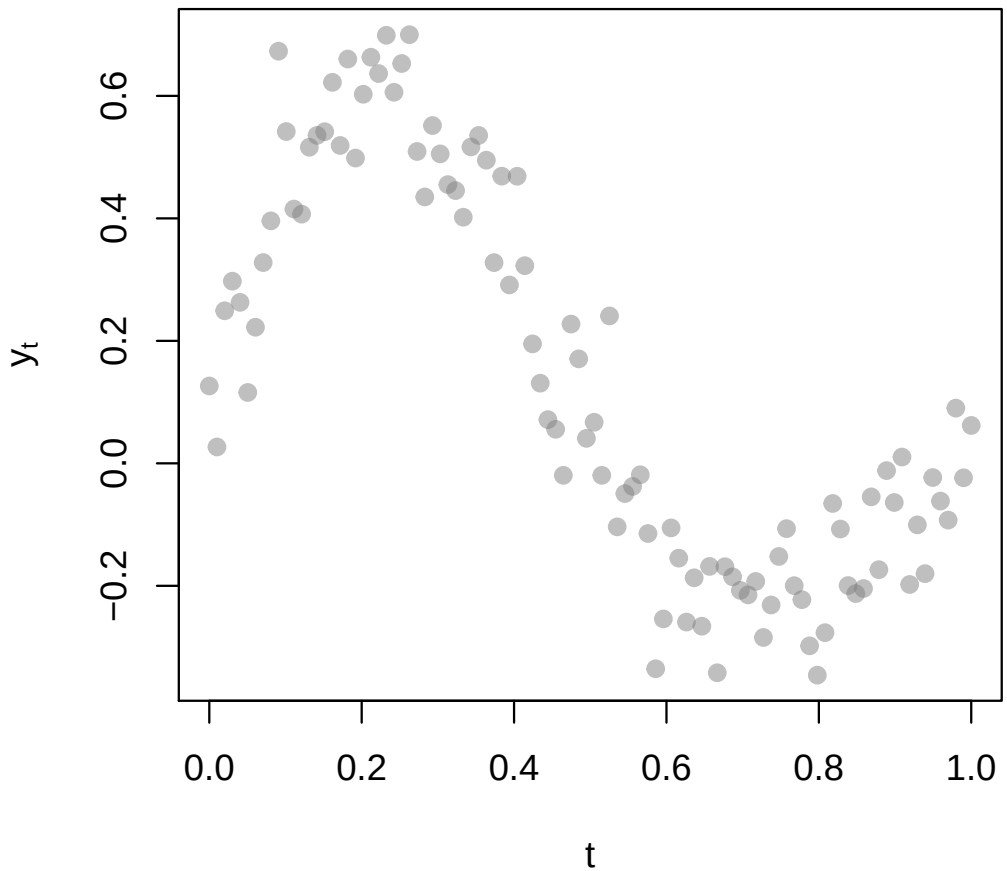
on signals defined on graphs

- Tikhonov/wavelet denoising
- signal compression / sampling
- inpainting / interpolation
- graph filtering (*e.g.* in GNNs)

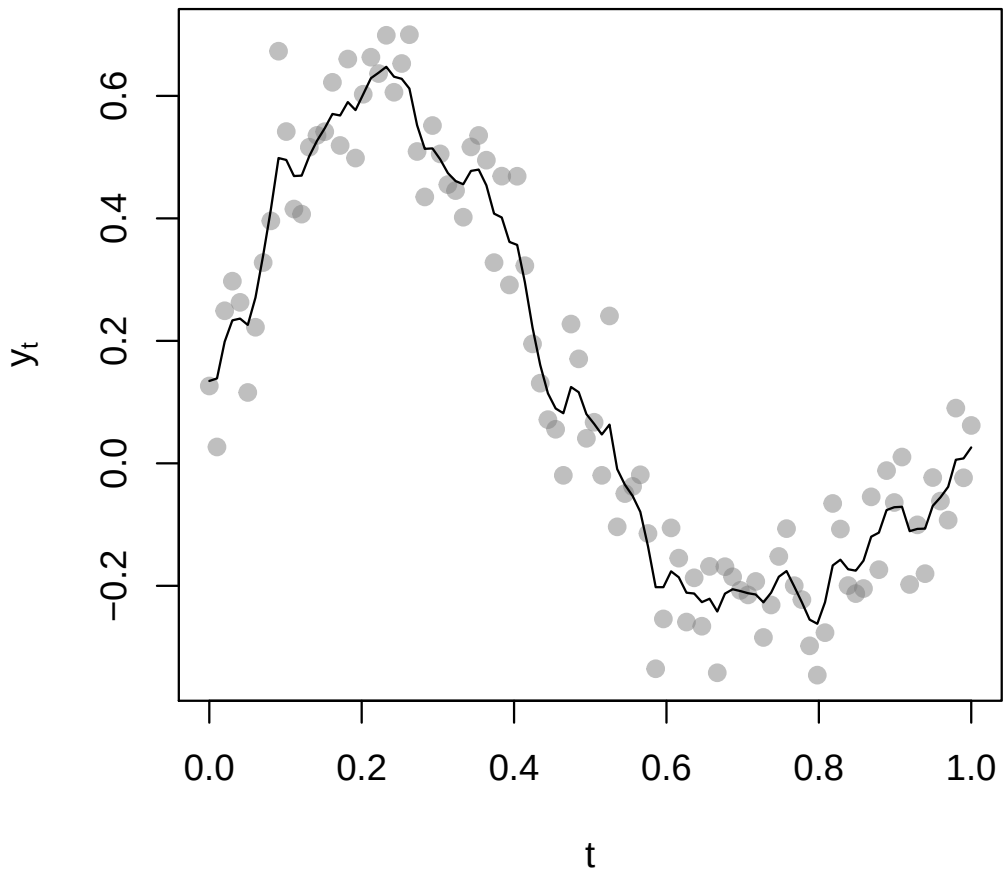
+ many other examples from many different fields of research...

$\Longrightarrow$ Some of these operations are based on the Laplacian matrix

Motivation I. Signal denoising: 1D background



Motivation I. Signal denoising: 1D background



Motivation I. Signal denoising: 1D background

- Formally, we assume that:
  - There is an underlying signal $x(t)$
  - We have noisy observations for a discrete set of time samples:

$$y_i = x(t_i) + \epsilon_i$$

  - $\epsilon_i \sim N(0, \sigma^2)$ (white Gaussian noise)
- We seek to estimate x from the noisy observations y

Motivation I. Signal denoising: 1D background

- Consider the following estimator (for $q > 0$)

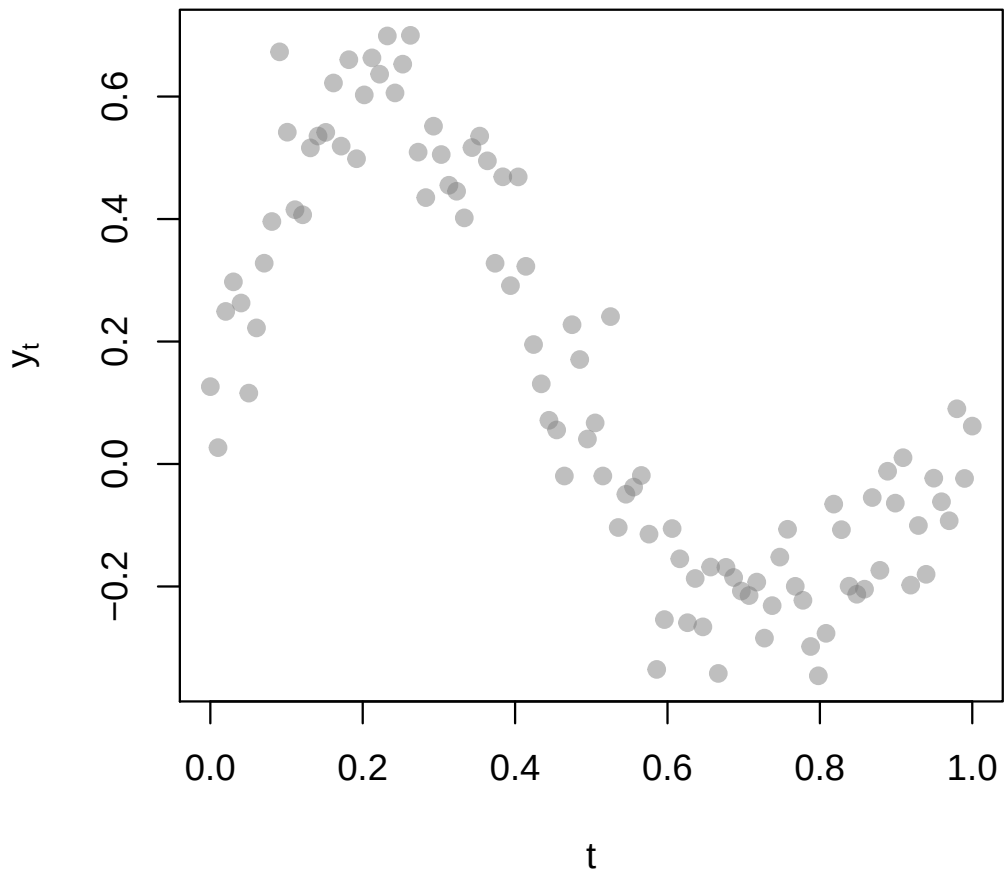
$$\hat{x} = \underset{z}{\text{argmin}} \quad q \, \frac{1}{2} \sum_{i=1}^{n} (z(t_i) - y(t_i))^2 + \frac{1}{2} \int_{\Omega} \left(\frac{d}{dt} z(t) \right)^2 dt$$

- $\sum_{i=1}^{n} (z(t_i) - y(t_i))^2$ is a *data-fidelity* term
- $\int_{\Omega} (\frac{d}{dt} z(t))^2 dt$ is a *smoothness* penalty
- q sets the trade-off between data fidelity and smoothness

Motivation I. Signal denoising: 1D background

$$\hat{x} = \underset{z}{\text{argmin}} \quad q \, \frac{1}{2} \sum_{i=1}^{n} (z(t_i) - y(t_i))^2 + \frac{1}{2} \int_{\Omega} \left(\frac{d}{dt} z(t) \right)^2 dt$$

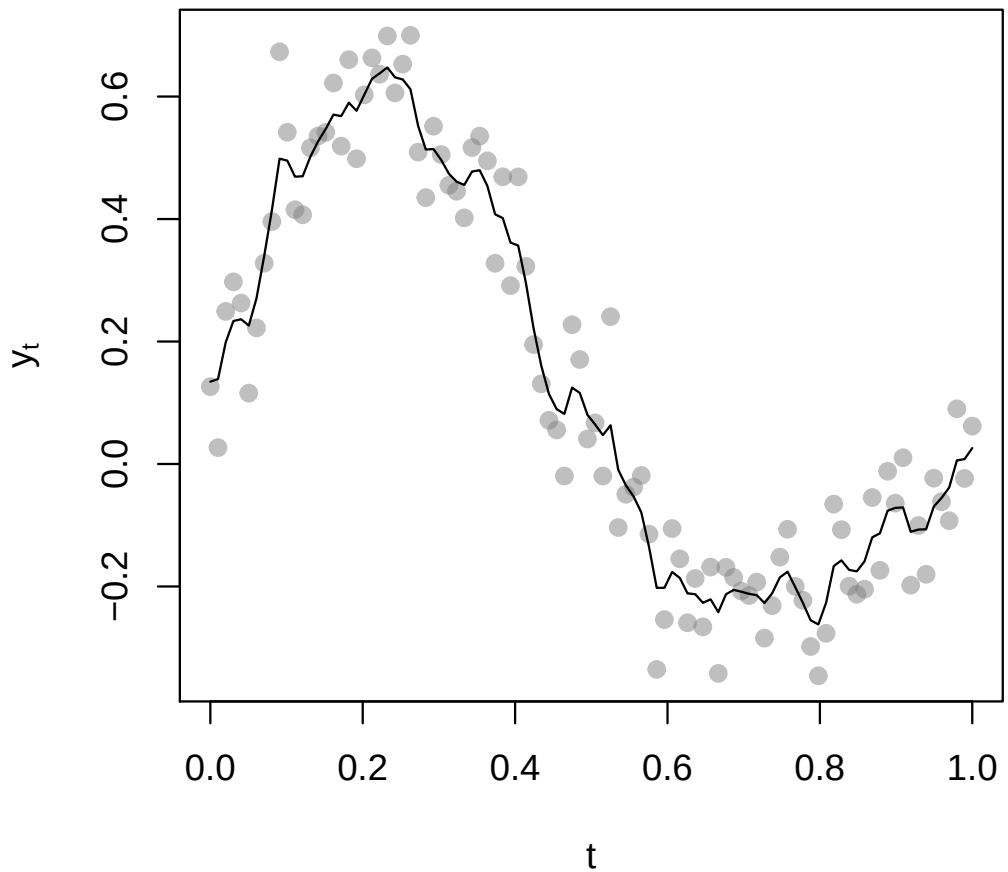
Observations y



Motivation I. Signal denoising: 1D background

$$\hat{x} = \underset{z}{\text{argmin}} \quad q \, \frac{1}{2} \sum_{i=1}^{n} (z(t_i) - y(t_i))^2 + \frac{1}{2} \int_{\Omega} \left(\frac{d}{dt} z(t) \right)^2 dt$$

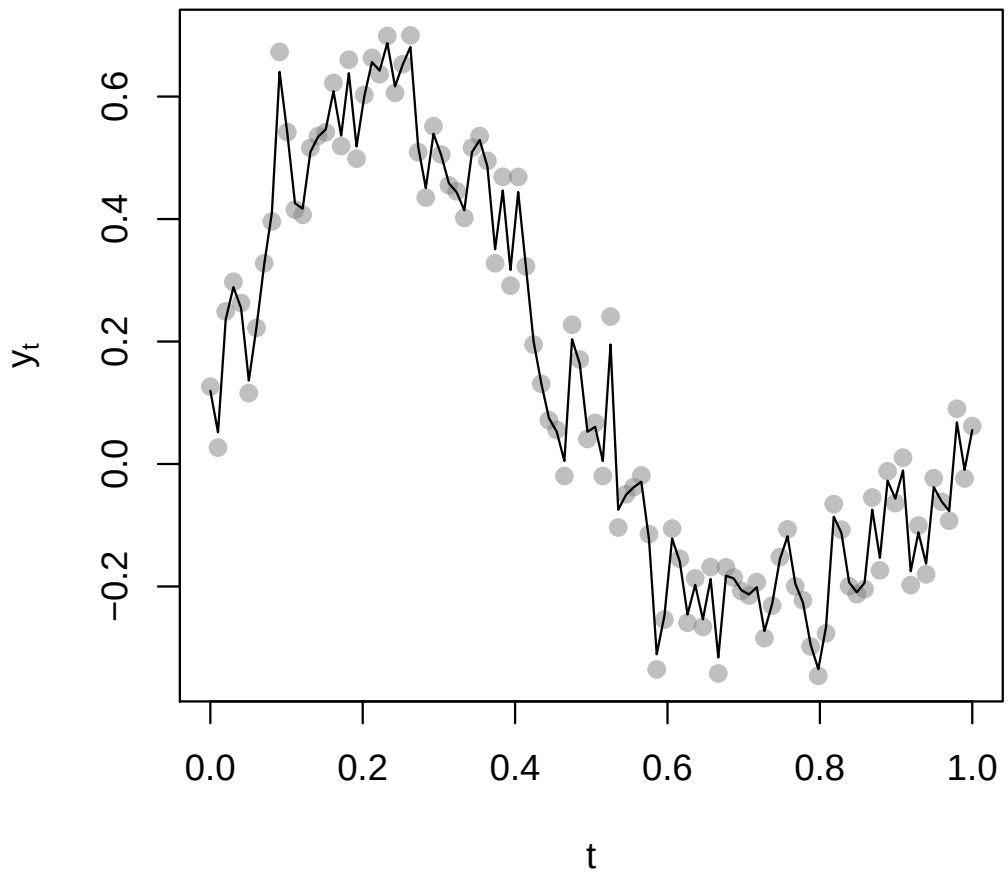
For a given $q > 0$



Motivation I. Signal denoising: 1D background

$$\hat{x} = \underset{z}{\text{argmin}} \quad q \, \frac{1}{2} \sum_{i=1}^{n} (z(t_i) - y(t_i))^2 + \frac{1}{2} \int_{\Omega} \left(\frac{d}{dt} z(t) \right)^2 dt$$

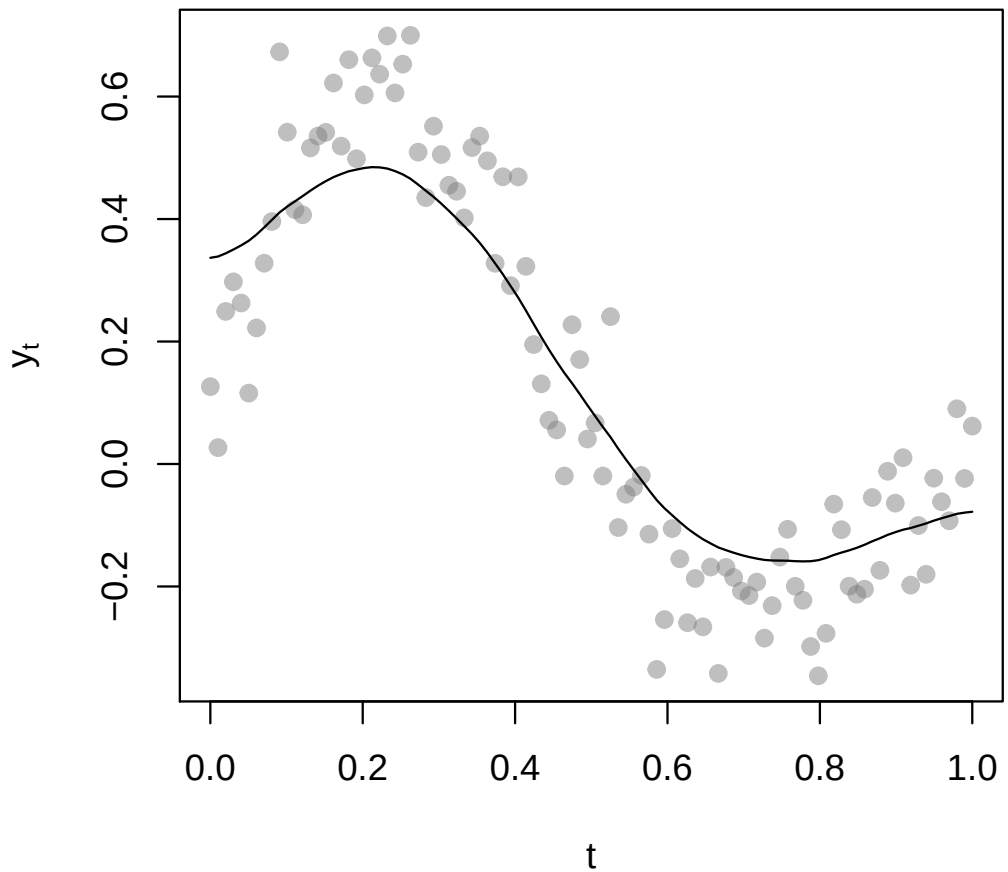
For a large $q > 0$



Motivation I. Signal denoising: 1D background

$$\hat{x} = \underset{z}{\text{argmin}} \quad q \, \frac{1}{2} \sum_{i=1}^{n} (z(t_i) - y(t_i))^2 + \frac{1}{2} \int_{\Omega} \left(\frac{d}{dt} z(t) \right)^2 dt$$

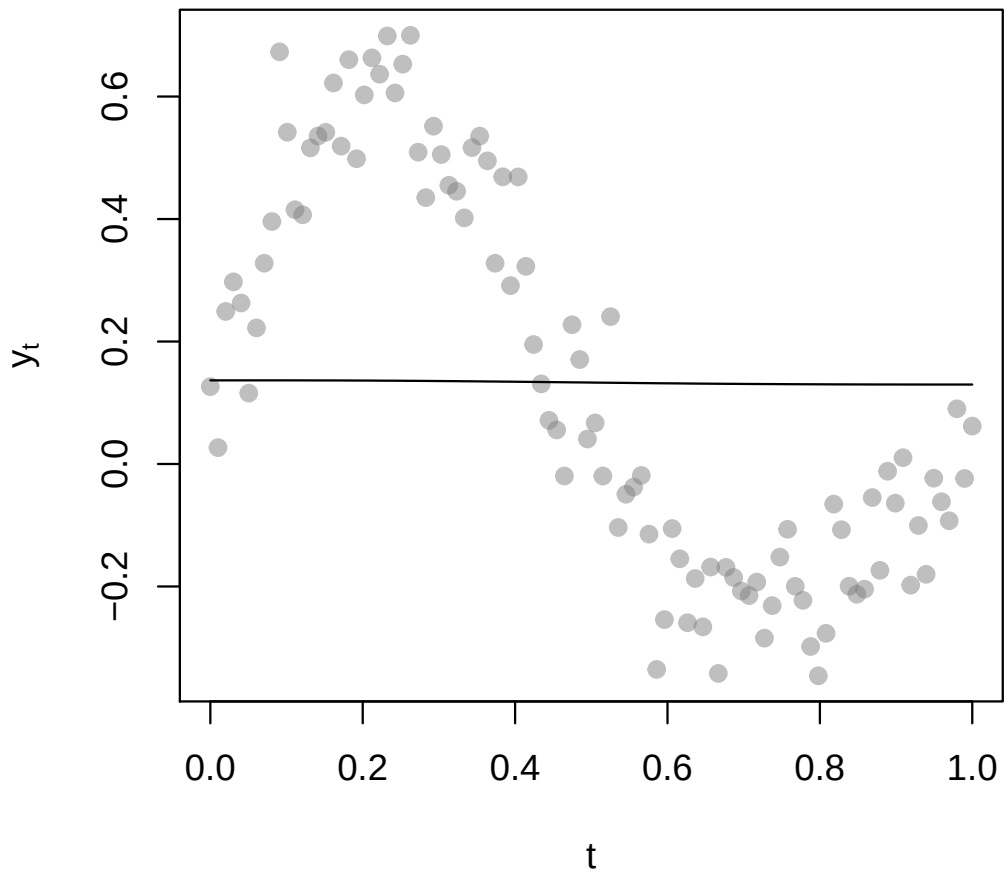
For a small $q > 0$



Motivation I. Signal denoising: 1D background

$$\hat{x} = \underset{z}{\text{argmin}} \quad q \, \frac{1}{2} \sum_{i=1}^{n} (z(t_i) - y(t_i))^2 + \frac{1}{2} \int_{\Omega} \left(\frac{d}{dt} z(t) \right)^2 dt$$

For $q \to 0^+$



Motivation I. Signal denoising: 1D background

- There is a simple way of estimating the smoothness penalty term when the observations fall on a regular grid:

$$\int_\Omega \left(\frac{d}{dt} z(t)\right)^2 dt \simeq \sum_{t=2}^{n} (z_t - z_{t-1})^2 = \frac{1}{2} \boldsymbol{z}^t \mathsf{L} \boldsymbol{z}$$

where $\mathsf{L} \in \mathbb{R}^{n \times n}$ is the

- discrete Laplace operator.
- equivalently the Laplacian of the path graph:

Motivation I. Signal denoising: 1D background

- There is a simple way of estimating the smoothness penalty term when the observations fall on a regular grid:

$$\int_\Omega \left(\frac{d}{dt} z(t)\right)^2 dt \simeq \sum_{t=2}^{n} (z_t - z_{t-1})^2 = \frac{1}{2} \boldsymbol{z}^t \mathsf{L} \boldsymbol{z}$$

where $\mathsf{L} \in \mathbb{R}^{n \times n}$ is the

- discrete Laplace operator.
- equivalently the Laplacian of the path graph:

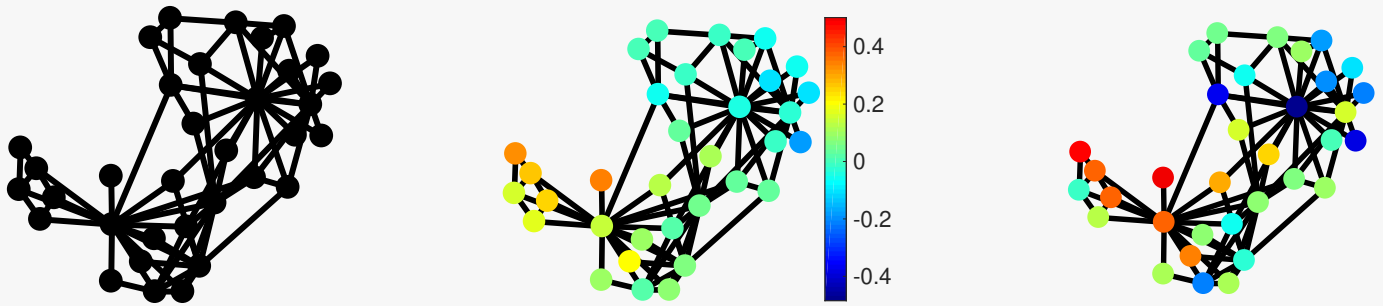
- One obtains the so-called Tikhonov regularisation problem:

$$\hat{\boldsymbol{x}} = \underset{\boldsymbol{z}}{\operatorname{argmin}} \quad q \, \frac{1}{2} \|\boldsymbol{z} - \boldsymbol{y}\|^2 + \frac{1}{2} \boldsymbol{z}^t \mathsf{L} \boldsymbol{z},$$

which exact solution reads

$$\hat{\boldsymbol{x}} = \mathsf{K} \boldsymbol{y} \text{ with } \mathsf{K} = (q\mathsf{I} + \mathsf{L})^{-1} q\mathsf{I} \in \mathbb{R}^{n \times n}.$$

Motivation I. Signal denoising: extension to arbitrary graphs



The graph Tikhonov regularization problem is similarly defined as

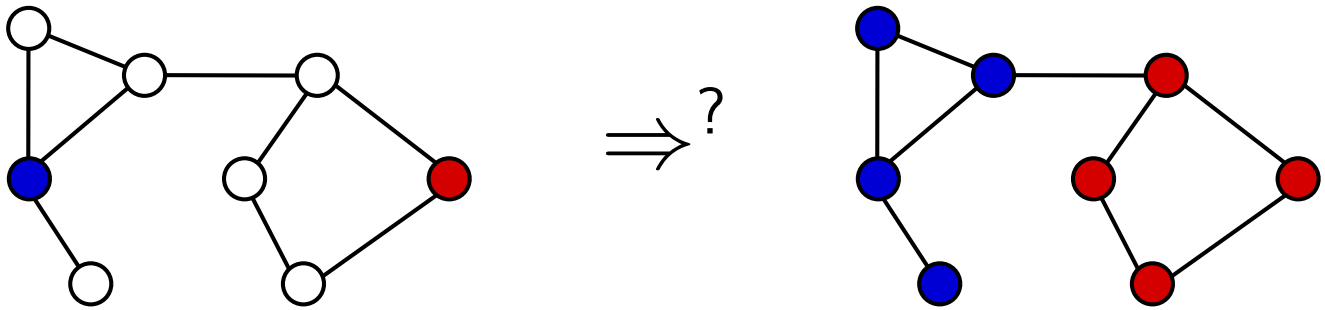
$$\hat{\boldsymbol{x}} = \underset{\boldsymbol{z}}{\operatorname{argmin}} \quad q \, \frac{1}{2} \left\| \boldsymbol{z} - \boldsymbol{y} \right\|^2 + \frac{1}{2} \boldsymbol{z}^t \mathrm{L} \boldsymbol{z}.$$

Note that the smoothness penalty term verifies $\frac{1}{2} \boldsymbol{z}^t \mathrm{L} \boldsymbol{z} = \sum_{i,j} \mathrm{A}_{ij} (z_i - z_j)^2$.

The analytic solution reads: $\hat{\boldsymbol{x}} = \mathrm{K} \boldsymbol{y}$ with $\mathrm{K} = (q\mathrm{I} + \mathrm{L})^{-1} q\mathrm{I} \in \mathbb{R}^{n \times n}$.

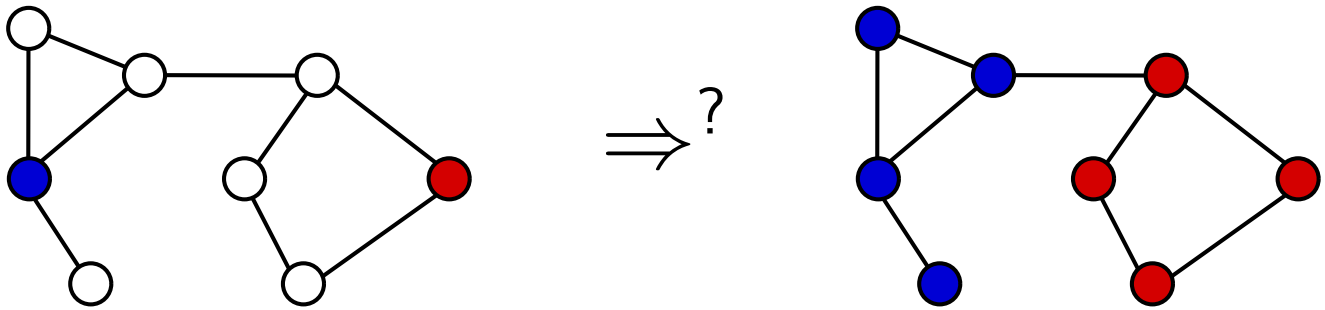
Motivation II. Semi-supervised learning on graphs

- SSL: given a few pre-labeled nodes of a graph, infer communities



Motivation II. Semi-supervised learning on graphs

- SSL: given a few pre-labeled nodes of a graph, infer communities

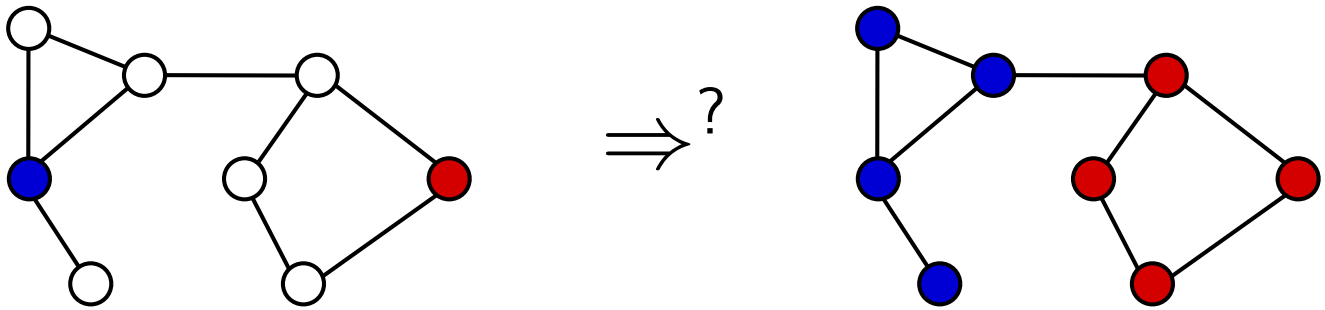


- Let $\boldsymbol{y}_\ell$ encode the information *a priori* known:

$$y_\ell(i) = \begin{cases} 1 & \text{if node } i \text{ is known to belong to class } \ell \\ 0 & \text{if not.} \end{cases}$$

Motivation II. Semi-supervised learning on graphs

- SSL: given a few pre-labeled nodes of a graph, infer communities



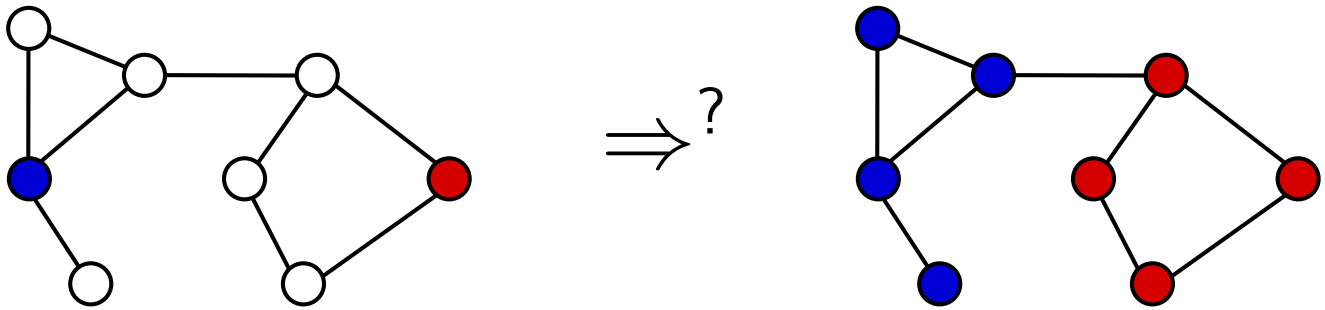
- Let $\boldsymbol{y}_\ell$ encode the information *a priori* known:

$$y_\ell(i) = \begin{cases} 1 & \text{if node } i \text{ is known to belong to class } \ell \\ 0 & \text{if not.} \end{cases}$$

- A classical SSL algo looks for a labeling function $\boldsymbol{f}_\ell$ that is both (with a trade-off parameter $\mu > 0$):
  - smooth on the graph ($\boldsymbol{f}_\ell^\top \mathrm{L} \boldsymbol{f}_\ell$ is small)
  - close to $\boldsymbol{y}_\ell$

Motivation II. Semi-supervised learning on graphs

- SSL: given a few pre-labeled nodes of a graph, infer communities



- Let $\boldsymbol{y}_\ell$ encode the information *a priori* known:

$$y_\ell(i) = \begin{cases} 1 & \text{if node } i \text{ is known to belong to class } \ell \\ 0 & \text{if not.} \end{cases}$$

- A classical SSL algo looks for a labeling function $\boldsymbol{f}_\ell$ that is both (with a trade-off parameter $\mu > 0$):
  - smooth on the graph ($\boldsymbol{f}_\ell^\top \mathrm{L} \boldsymbol{f}_\ell$ is small)
  - close to $\boldsymbol{y}_\ell$
- the larger $f_\ell(i)$, the higher the chance that i belongs to class ℓ.

Motivation II. Semi-supervised learning on graphs

- The solution may be written (in the usual SSL formalism) as

$$\boldsymbol{f}_\ell = \frac{\mu}{2+\mu}\left(\mathrm{I} - \frac{2}{2+\mu}\mathrm{D}^{-1}\mathrm{W}\right)^{-1}\boldsymbol{y}_\ell.$$

- Can be re-written as:

$$\boldsymbol{f}_\ell = \mathrm{K}\boldsymbol{y}_\ell.$$

  where $\mathrm{K} = (\mathrm{Q}+\mathrm{L})^{-1}\,\mathrm{Q}$ and $\mathrm{Q} = \frac{\mu}{2}\mathrm{D}$ is diagonal and strictly positive.
- This generalizes the previous case where we needed

$$\hat{\boldsymbol{x}} = \mathrm{K}\boldsymbol{y} \quad \text{with} \quad \mathrm{K} = (q\mathrm{I}+\mathrm{L})^{-1}q\mathrm{I}.$$

Motivation III. Degrees of freedom of the linear smoother $\mathrm{K} = (\mathrm{Q} + \mathrm{L})^{-1}\mathrm{Q}$

- $\hat{\boldsymbol{x}} = \mathrm{K}\boldsymbol{y}$ is one instance of the more general class of *linear smoothers*
- a useful quantity in this context is its so-called *effective degrees of freedom* defined as

$$\delta = \mathrm{Tr}\left(\mathrm{K}\right) = \mathrm{Tr}\left((\mathrm{Q} + \mathrm{L})^{-1}\mathrm{Q}\right)$$

- this quantity is useful for instance for optimal choice of parameter q in Tikhonov's regularization (using AIC, GCV, etc.)

Motivation IV. Effective resistances

Definition Let $\mathrm{L}^\dagger$ be the Moore-Penrose pseudo-inverse of L. The effective resistance of any edge $e = (i, j) \in \mathcal{E}$ reads

$$r_e = \mathrm{L}^\dagger_{ii} + \mathrm{L}^\dagger_{jj} - 2\mathrm{L}^\dagger_{ij} > 0.$$

Motivation IV. Effective resistances

Definition Let $\mathrm{L}^\dagger$ be the Moore-Penrose pseudo-inverse of L. The effective resistance of any edge $e = (i, j) \in \mathcal{E}$ reads

$$r_e = \mathrm{L}^\dagger_{ii} + \mathrm{L}^\dagger_{jj} - 2\mathrm{L}^\dagger_{ij} > 0.$$

Property $\forall e \in \mathcal{E}, p_e \propto \mathrm{A}_e r_e$ defines a probability distribution over the edges.

Motivation IV. Effective resistances

Definition Let $\mathrm{L}^\dagger$ be the Moore-Penrose pseudo-inverse of L. The effective resistance of any edge $e = (i, j) \in \mathcal{E}$ reads

$$r_e = \mathrm{L}^\dagger_{ii} + \mathrm{L}^\dagger_{jj} - 2\mathrm{L}^\dagger_{ij} > 0.$$

Property $\forall e \in \mathcal{E}, p_e \propto \mathrm{A}_e r_e$ defines a probability distribution over the edges.

What for? For instance: spectral sparsification.

Motivation IV. Effective resistances

Definition Let $\mathrm{L}^\dagger$ be the Moore-Penrose pseudo-inverse of L. The effective resistance of any edge $e = (i, j) \in \mathcal{E}$ reads

$$r_e = \mathrm{L}^\dagger_{ii} + \mathrm{L}^\dagger_{jj} - 2\mathrm{L}^\dagger_{ij} > 0.$$

Property $\forall e \in \mathcal{E}, p_e \propto \mathrm{A}_e r_e$ defines a probability distribution over the edges.

What for? For instance: spectral sparsification.

Definition Let $\epsilon \in]0, 1[$. Let $\mathcal{G}$ be a graph and L its Laplacian. Let $\mathcal{H}$ be a subgraph of $\mathcal{G}$, of same size but hopefully much sparser, and X its Laplacian. Then $\mathcal{H}$ is a ϵ-spectral sparsifier of $\mathcal{G}$ if

$$(1 - \epsilon)\mathrm{L} \preceq \mathrm{X} \preceq (1 + \epsilon)\mathrm{L}.$$

Motivation IV. Effective resistances

Definition Let $\mathrm{L}^\dagger$ be the Moore-Penrose pseudo-inverse of L. The effective resistance of any edge $e = (i, j) \in \mathcal{E}$ reads

$$r_e = \mathrm{L}^\dagger_{ii} + \mathrm{L}^\dagger_{jj} - 2\mathrm{L}^\dagger_{ij} > 0.$$

Property $\forall e \in \mathcal{E}, p_e \propto \mathrm{A}_e r_e$ defines a probability distribution over the edges.

What for? For instance: spectral sparsification.

Definition Let $\epsilon \in]0, 1[$. Let $\mathcal{G}$ be a graph and L its Laplacian. Let $\mathcal{H}$ be a subgraph of $\mathcal{G}$, of same size but hopefully much sparser, and X its Laplacian. Then $\mathcal{H}$ is a ϵ-spectral sparsifier of $\mathcal{G}$ if

$$(1 - \epsilon)\mathrm{L} \preceq \mathrm{X} \preceq (1 + \epsilon)\mathrm{L}.$$

Theorem Let $\epsilon \in]0, 1[$. Sample m edges iid with replacement according to $\boldsymbol{p}$. The obtained subgraph (with properly re-weighted edges) is a spectral sparsifier with high probability provided

$$m \geq \mathcal{O}\left(\epsilon^{-2} n \log n\right).$$

A (partial) list of Laplacian-based computation problems

Problem 1 Compute the regularized inverse

$$\hat{\boldsymbol{x}} = \mathrm{K}\boldsymbol{y} \text{ with } \begin{cases} \mathrm{K} = (\mathrm{Q} + \mathrm{L})^{-1}\mathrm{Q} \\ \mathrm{Q} \text{ diagonal and positive} \end{cases}$$

Problem 2 Compute the degrees of freedom of the linear smoother K:

$$\delta = \mathrm{Tr}\left((\mathrm{Q} + \mathrm{L})^{-1}\mathrm{Q}\right)$$

Problem 3 Compute the effective resistance $r_e = \mathrm{L}^\dagger_{ii} + \mathrm{L}^\dagger_{jj} - 2\mathrm{L}^\dagger_{ij}$ of all edges

Many other L-based problems partial spectral information for spectral clustering, graph cuts, etc.

In this talk, we will discuss:

Problem Compute the degrees of freedom of the linear smoother K:

$$\delta = \mathrm{Tr}\left((\mathrm{Q}+\mathrm{L})^{-1}\mathrm{Q}\right)$$

- to simplify, in this talk, we only consider the case where L is the combinatorial Laplacian of a connected graph and $\mathrm{Q} = q\mathrm{I}$ $(q > 0)$
- however, keep in mind that the techniques developed can be extended to:
  - any positive diagonal matrix Q
  - any choice of Laplacian, *e.g.*, the degree-normalized Laplacian matrix $\mathcal{L}$:

$$\hat{\boldsymbol{x}} = (\mathrm{Q}+\mathcal{L})^{-1}\mathrm{Q}\boldsymbol{y} = \left(\mathrm{Q}+\mathrm{D}^{-1/2}\mathrm{L}\mathrm{D}^{-1/2}\right)^{-1}\mathrm{Q}\boldsymbol{y}$$
$$= \mathrm{D}^{1/2}\left(\mathrm{Q}'+\mathrm{L}\right)^{-1}\mathrm{Q}'\mathrm{D}^{-1/2}\boldsymbol{y}$$

  - in fact, L may even be any diagonally dominant matrix, via a standard trick involving two graphs
- It so happens that Yigit Pilavci will discuss some other problems (graph Tikhonov regularisation) this afternoon at the MLSP seminar

Computational challenges

Goal

Compute $\delta = \mathrm{Tr}\left((q\mathrm{I} + \mathrm{L})^{-1} q\right)$ for large graphs (say $n > 10^6$)

Difficulty

Need to compute an inverse, with (naive) cost $\mathcal{O}(n^3)$

State-of-the-art

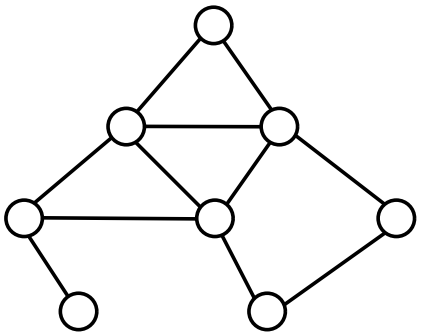
For sparse graphs, a combination of Hutchinson's estimator and iterative methods such as Preconditioned Conjugate Gradients (effective to approximate efficiently $(q\mathrm{I} + \mathrm{L})^{-1} q\boldsymbol{y}$ given a vector $\boldsymbol{y}$) is state-of-the-art

In this work

We formulate (very different) Monte Carlo estimators based on random spanning forests

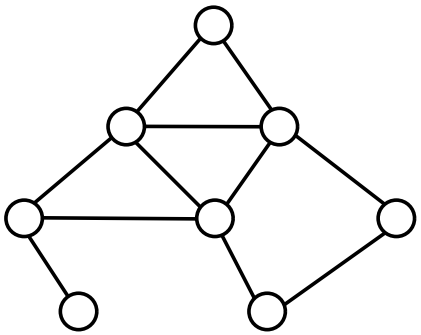
Contents

Forests?

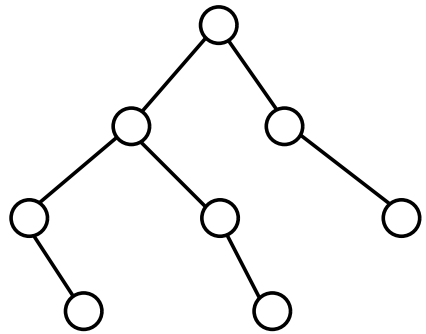


A graph

Forests?

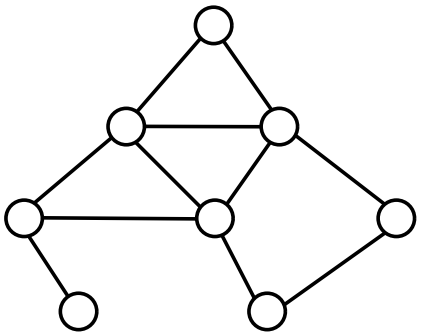


A graph

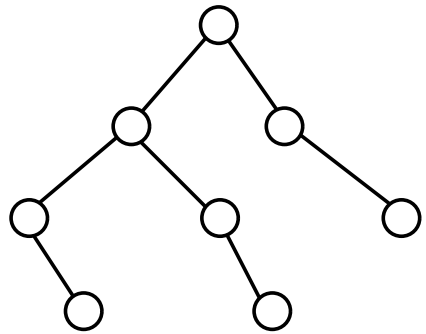


A spanning tree

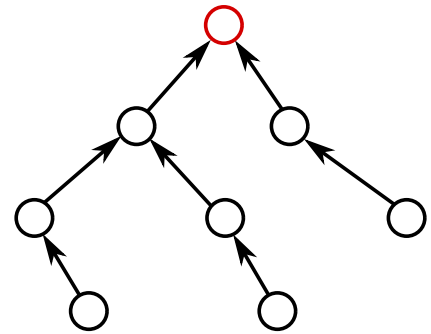
Forests?



A graph

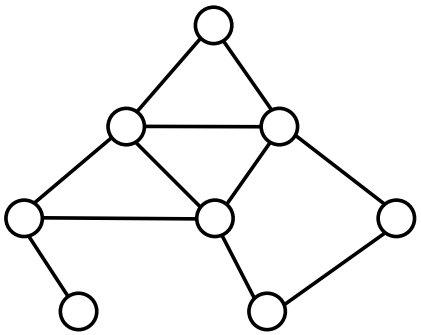


A spanning tree

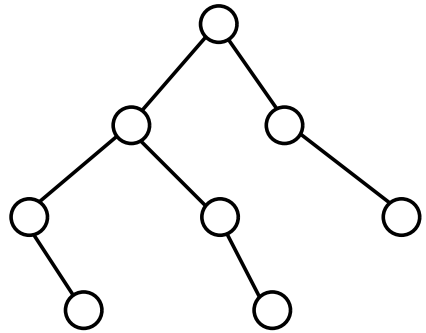


A rooted spanning tree

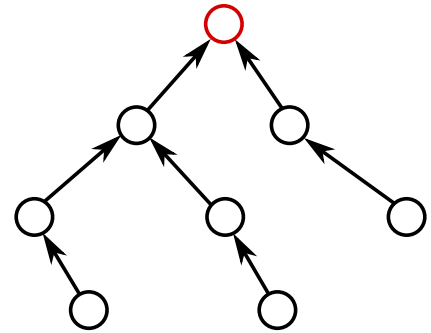
Forests?



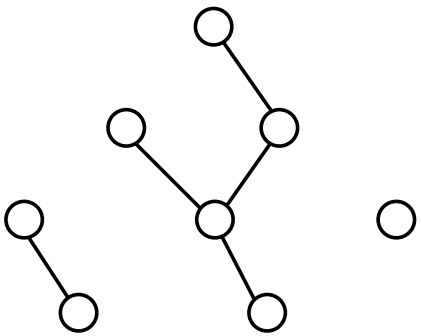
A graph



A spanning tree

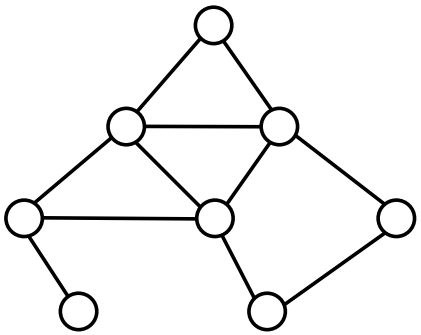


A rooted spanning tree

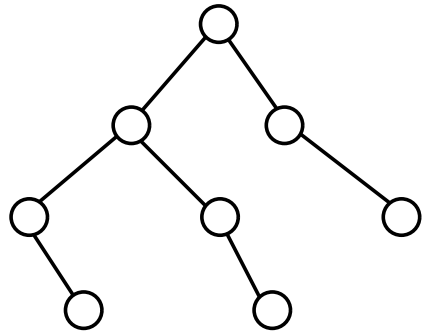


A spanning forest

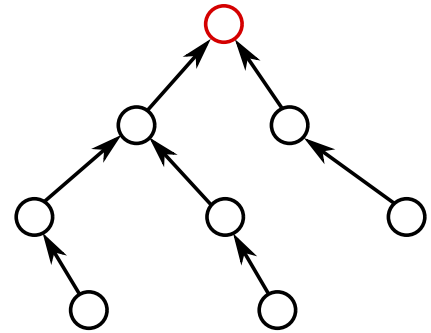
Forests?



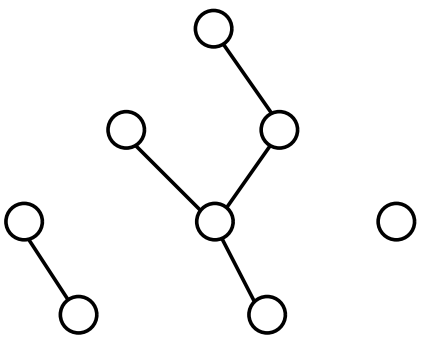
A graph



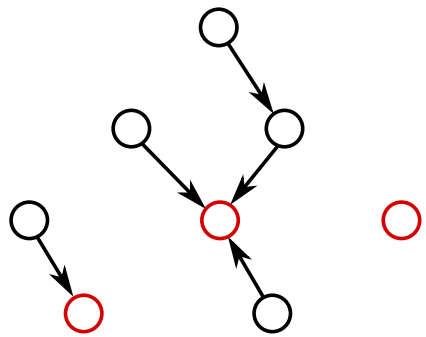
A spanning tree



A rooted spanning tree

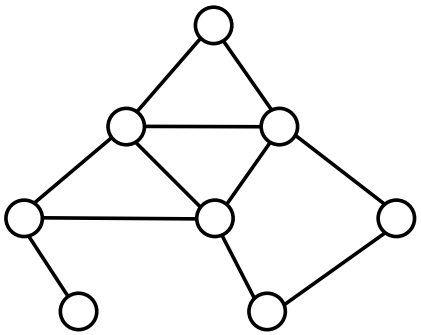


A spanning forest

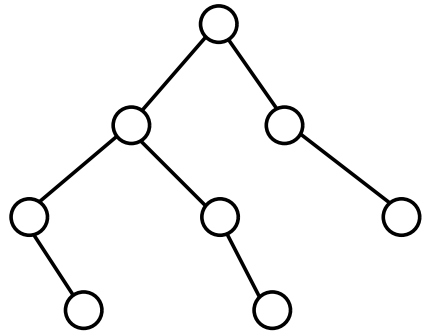


A rooted spanning forest

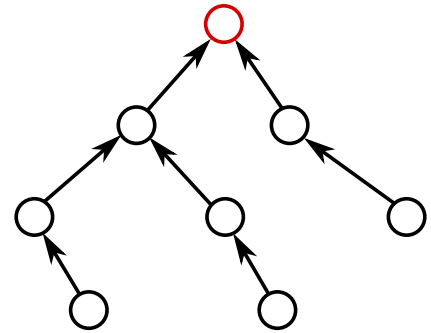
Forests?



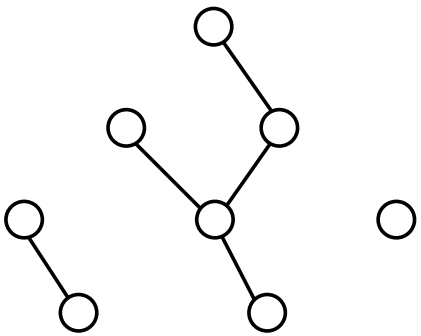
A graph



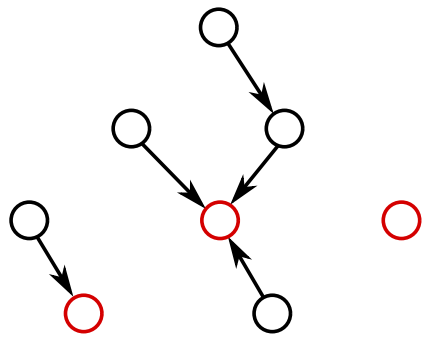
A spanning tree



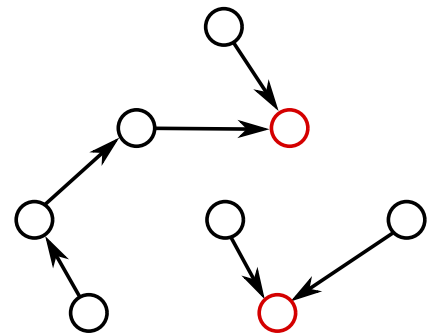
A rooted spanning tree



A spanning forest

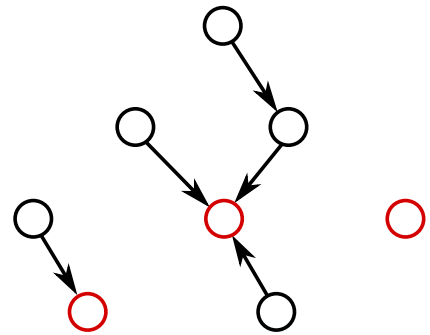


A rooted spanning forest



Another rooted spanning forest

A few notations



Given a rooted spanning forest ϕ, we define:

- $\rho(\phi)$ its set of roots (the red nodes)
- the root function: $r_\phi(i)$ is the root of the tree containing node i

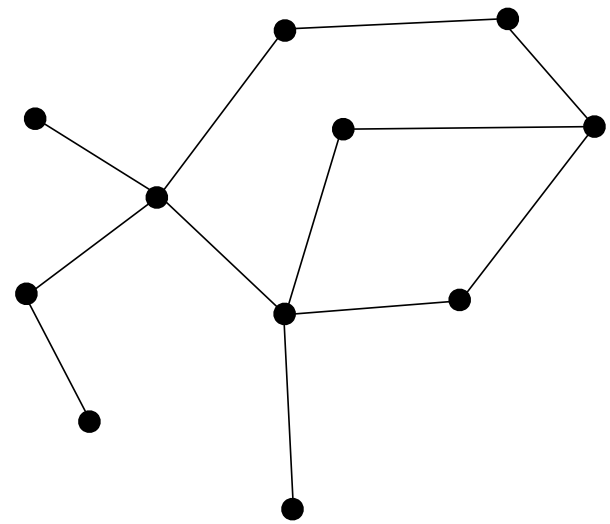
Random forests?

- Consider the following distribution on forests ($q > 0$):

$$\mathbb{P}(\Phi_q = \phi) \propto q^{|\rho(\phi)|} \prod_{(ij)\in\phi} \mathrm{A}_{ij}$$

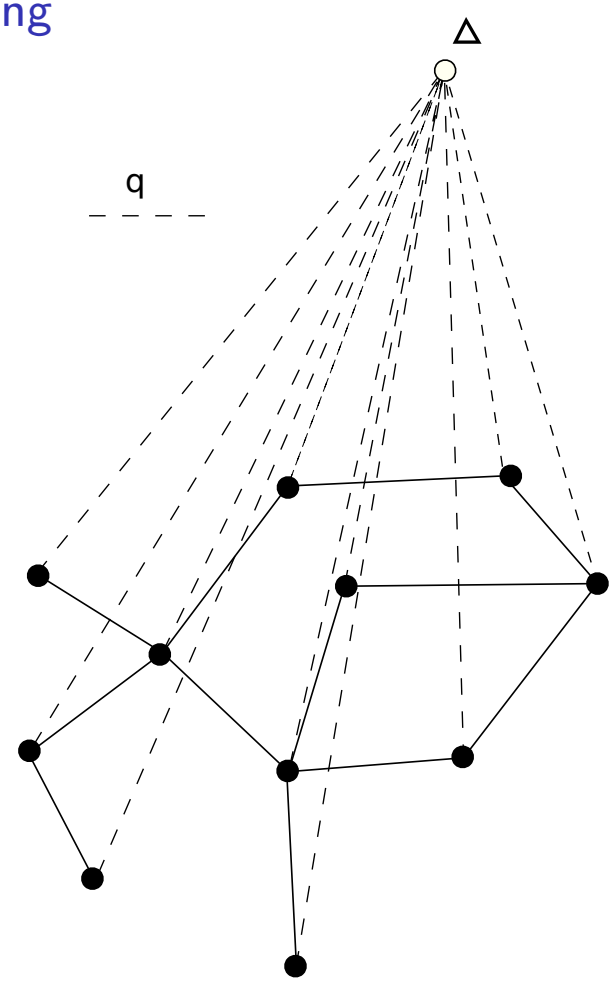
  where A_{ij} is the weight of the link between nodes i and j.
- Importantly, sampling from this distribution is efficient (in time roughly $\mathcal{O}(|E|/q)$) via a variant of Wilson's algorithm.
- The forest obtained, Φ_q, is called a Random Spanning Forest (RSF)

Wilson's algorithm for RSF sampling



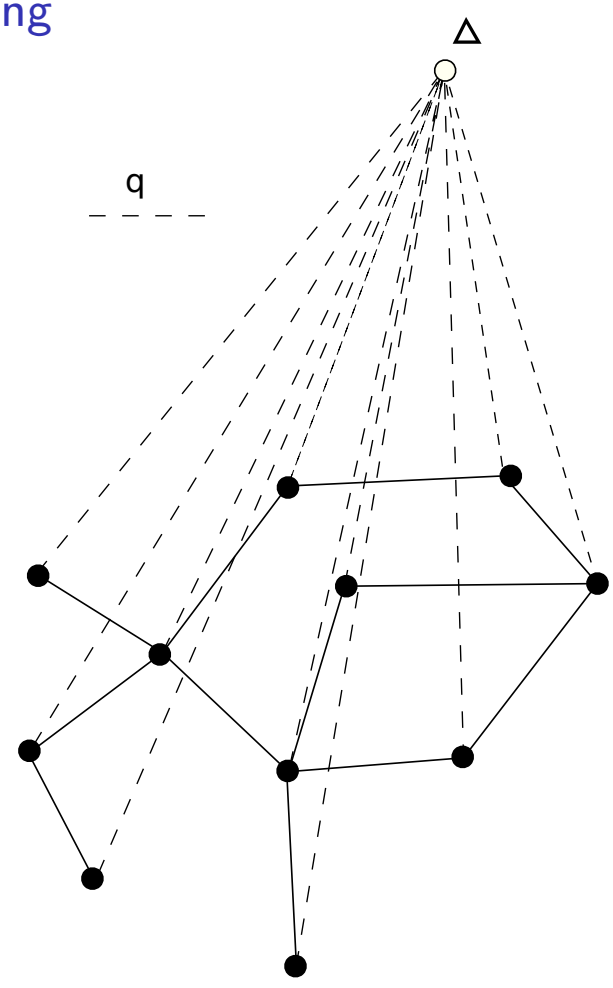
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q



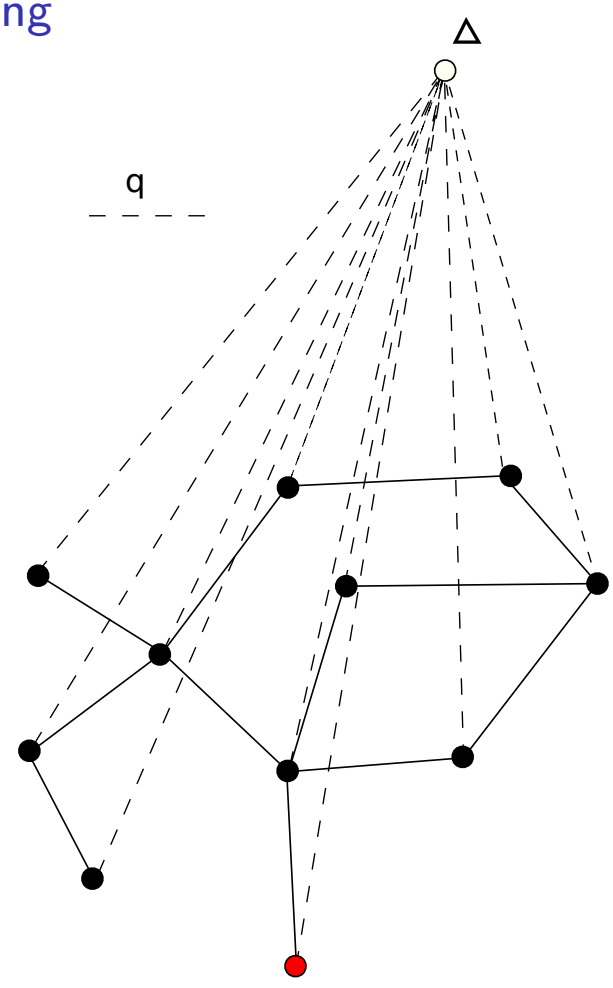
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ



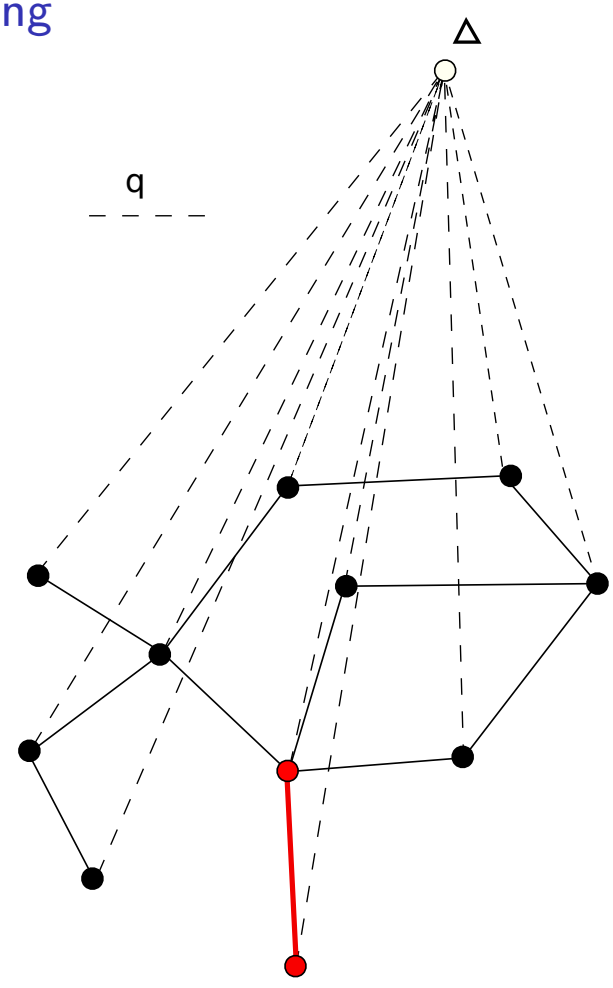
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ



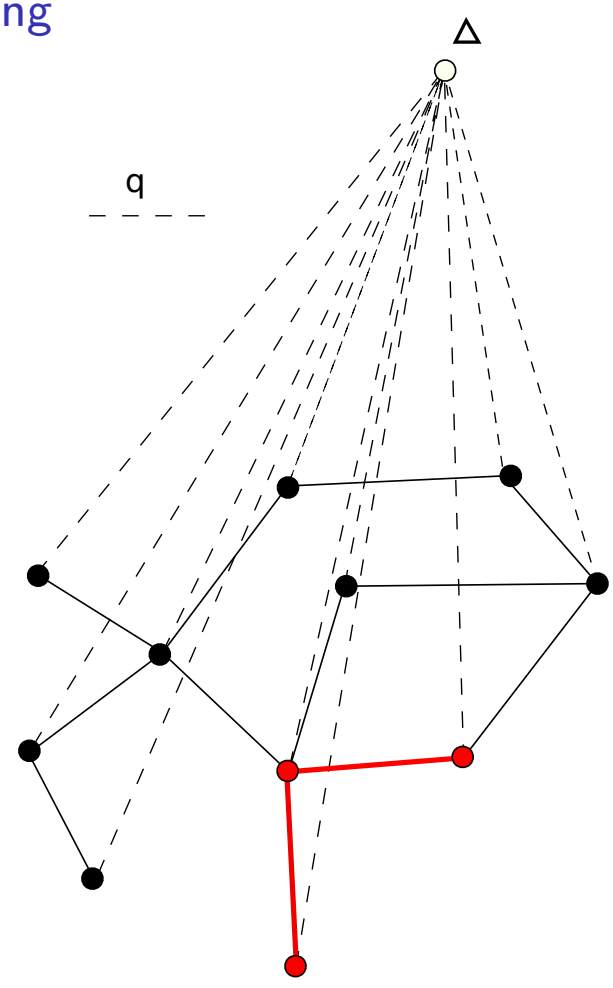
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ



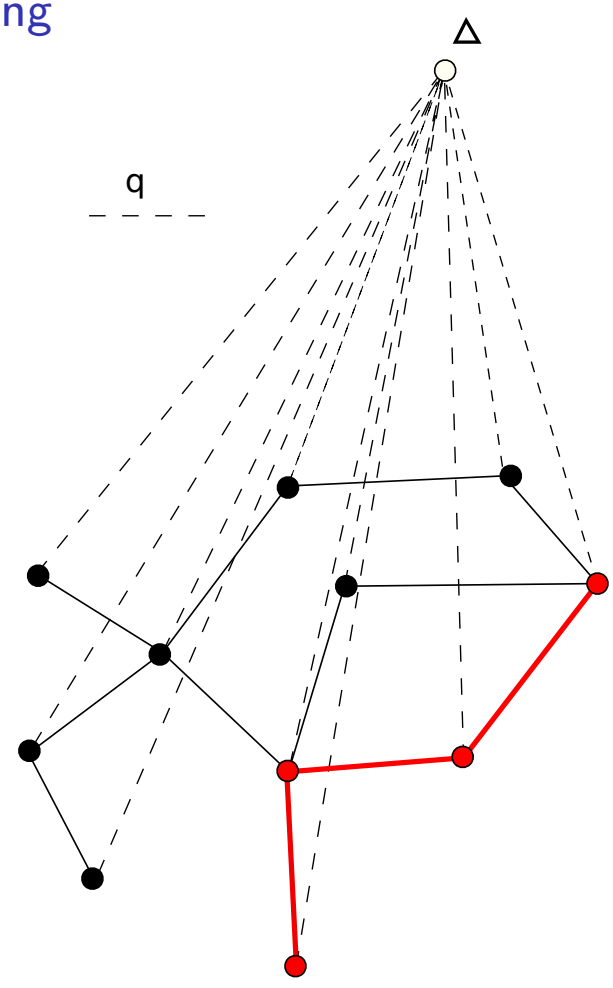
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ



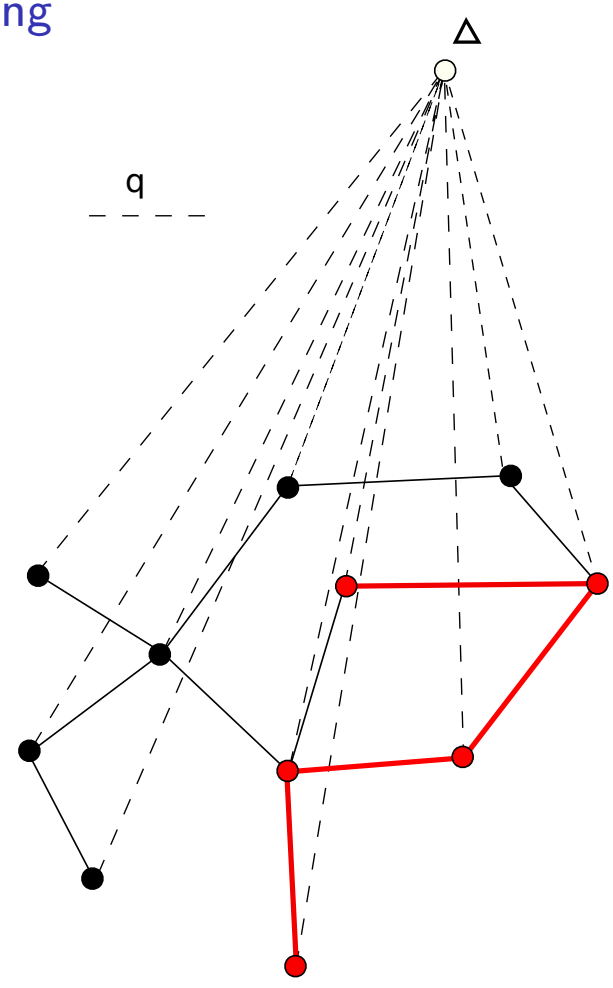
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ



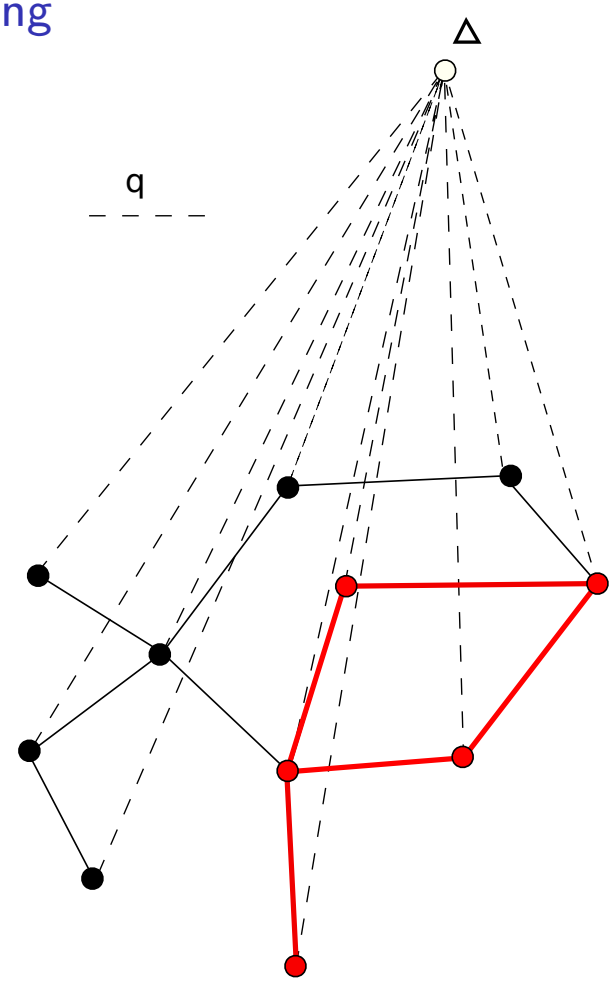
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ



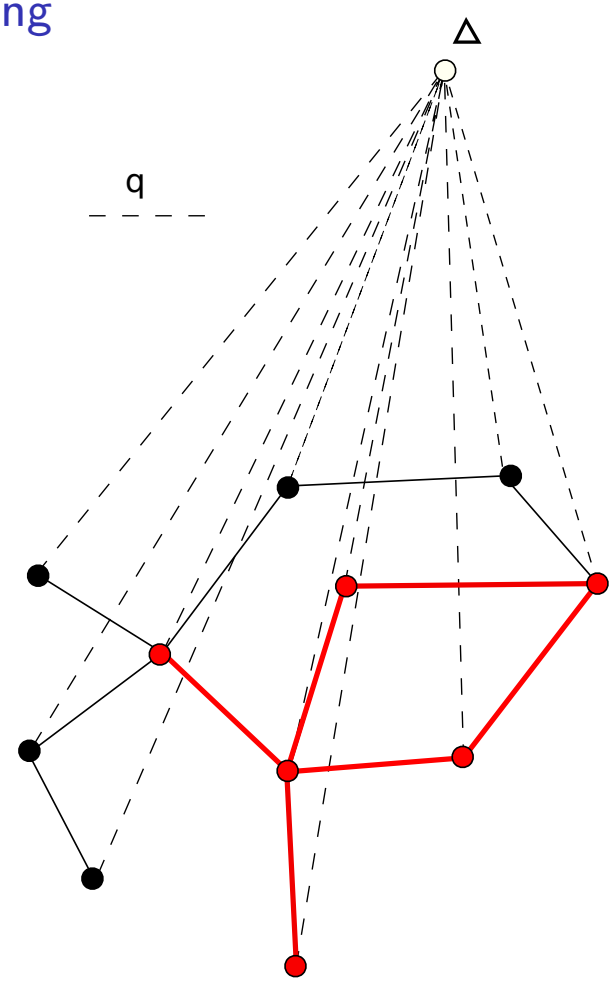
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ



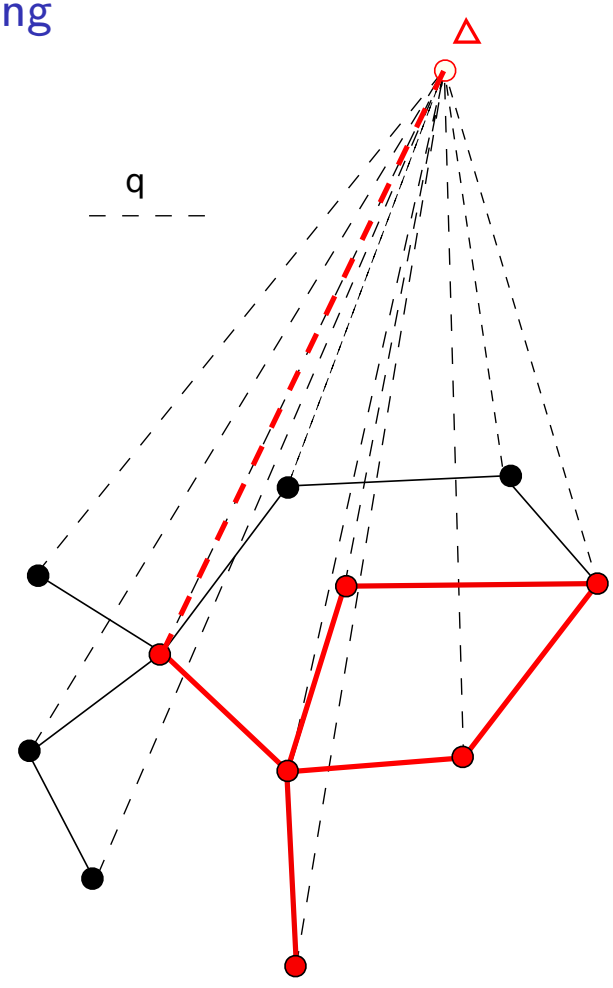
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ



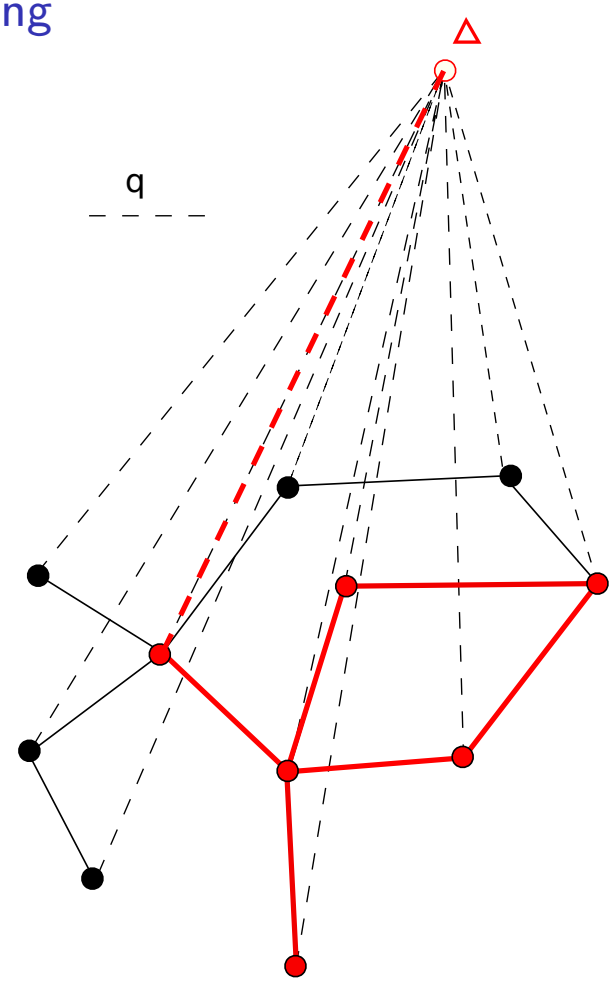
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ



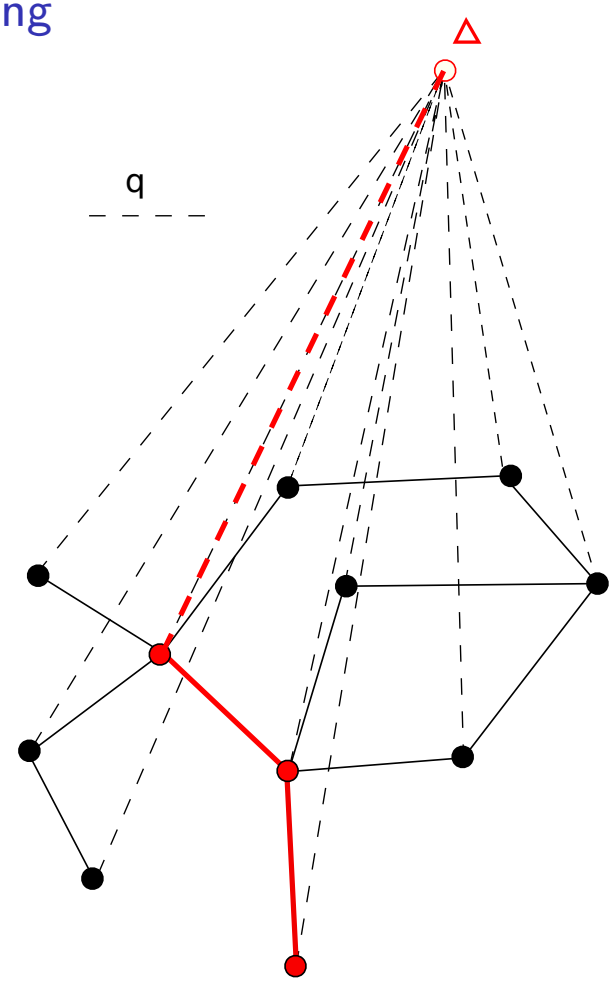
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops



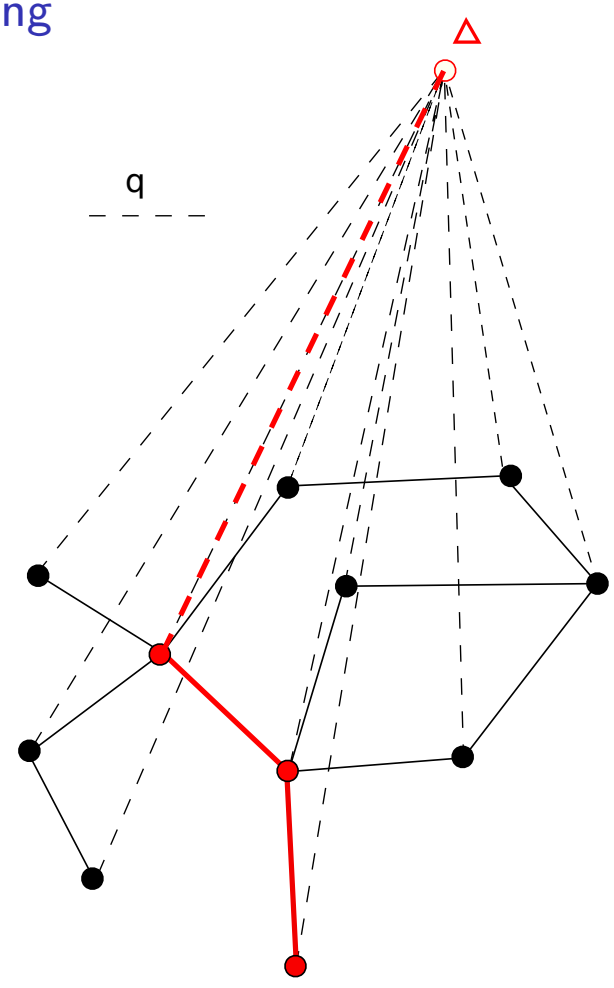
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops



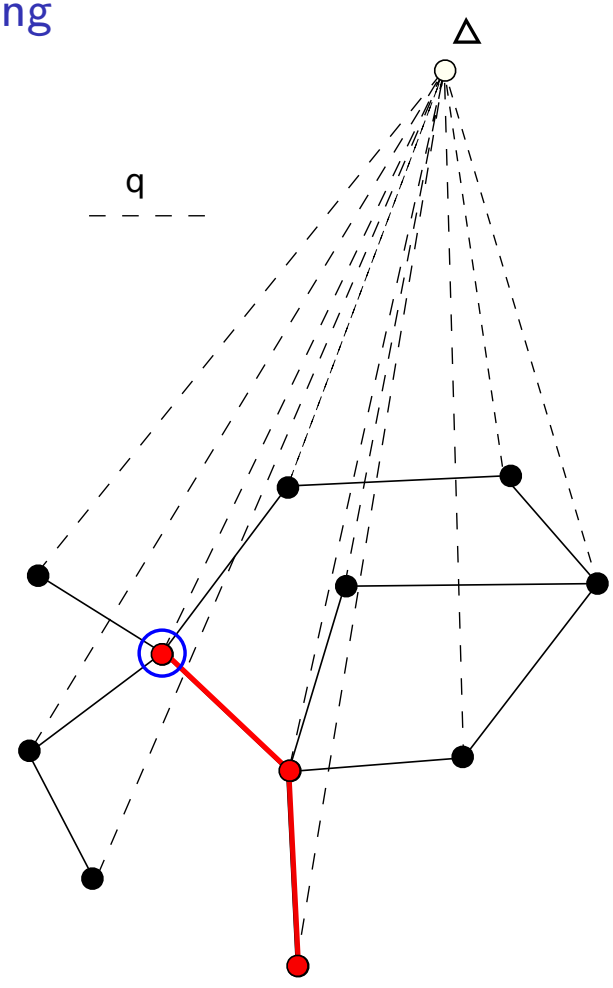
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots



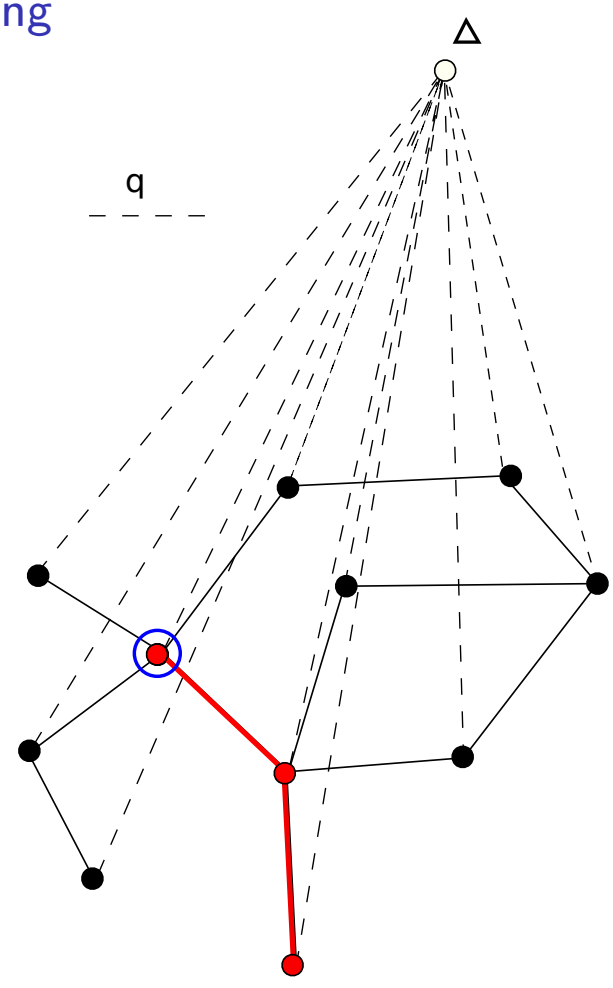
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots



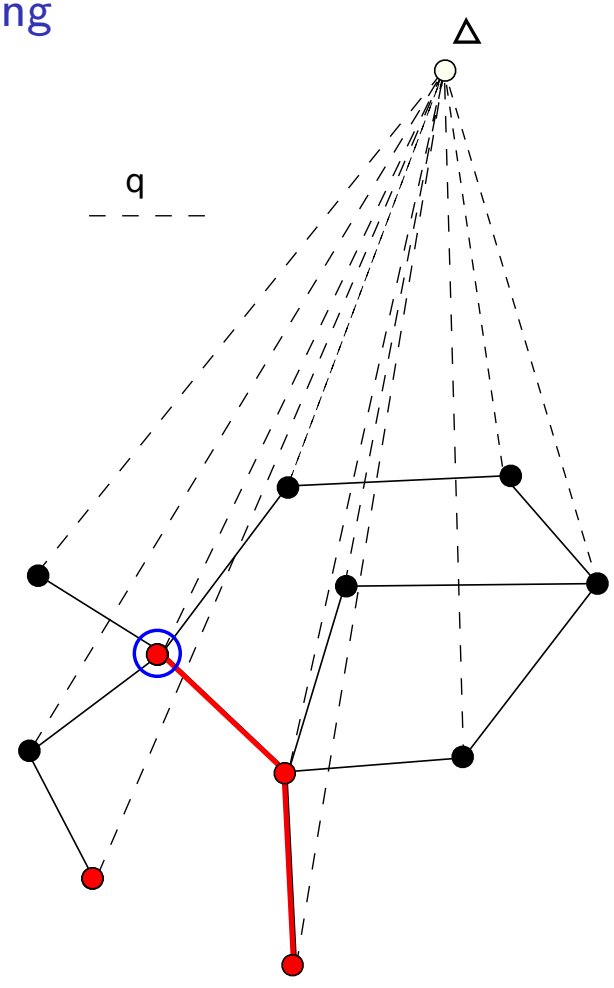
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node



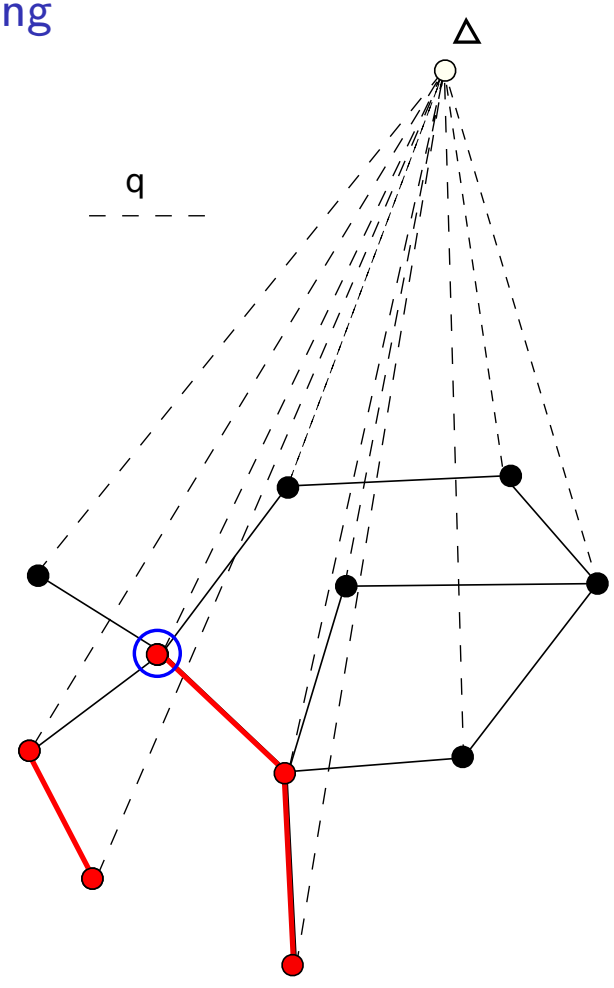
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node



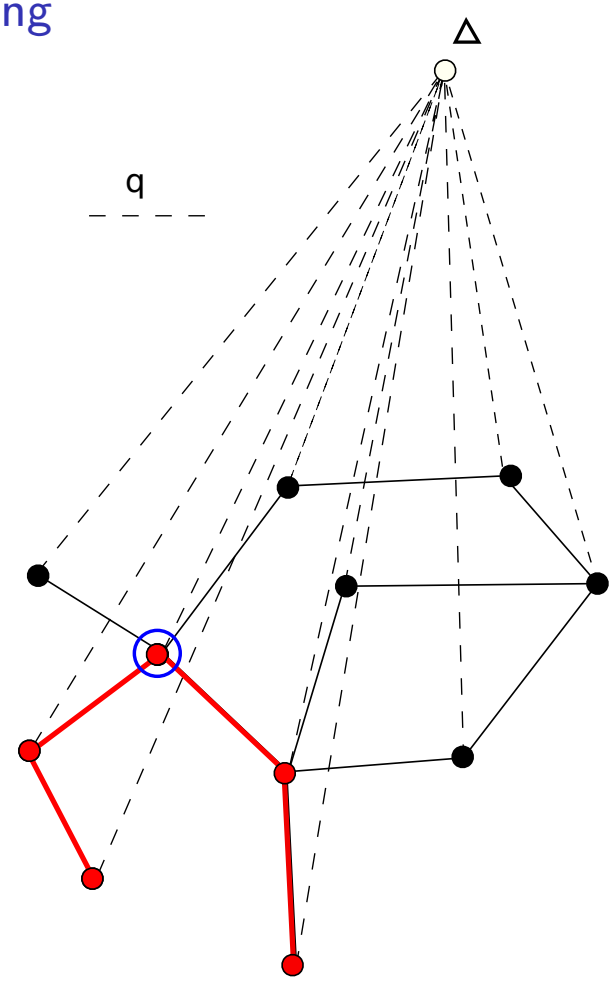
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node



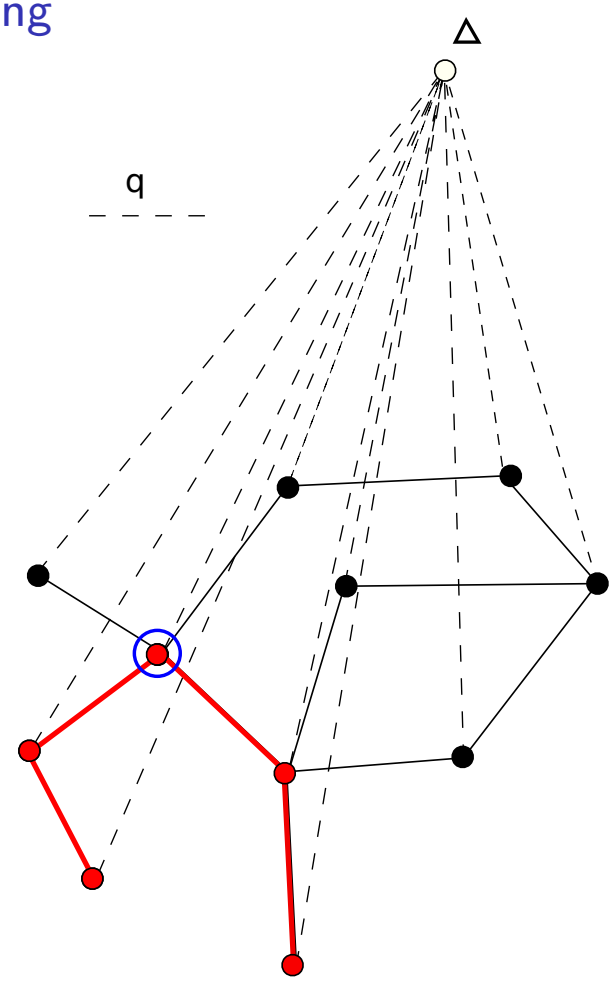
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node



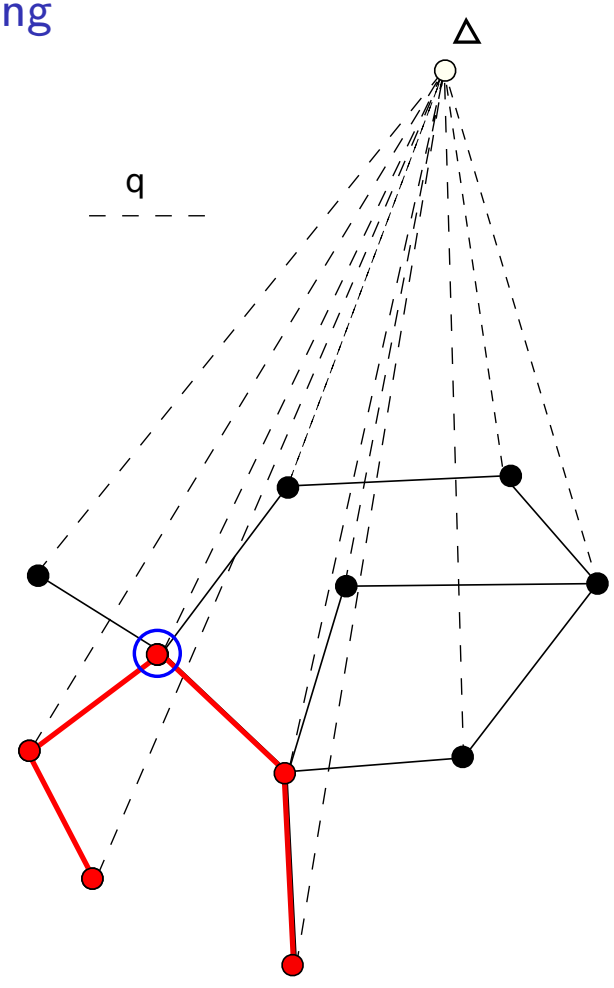
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges



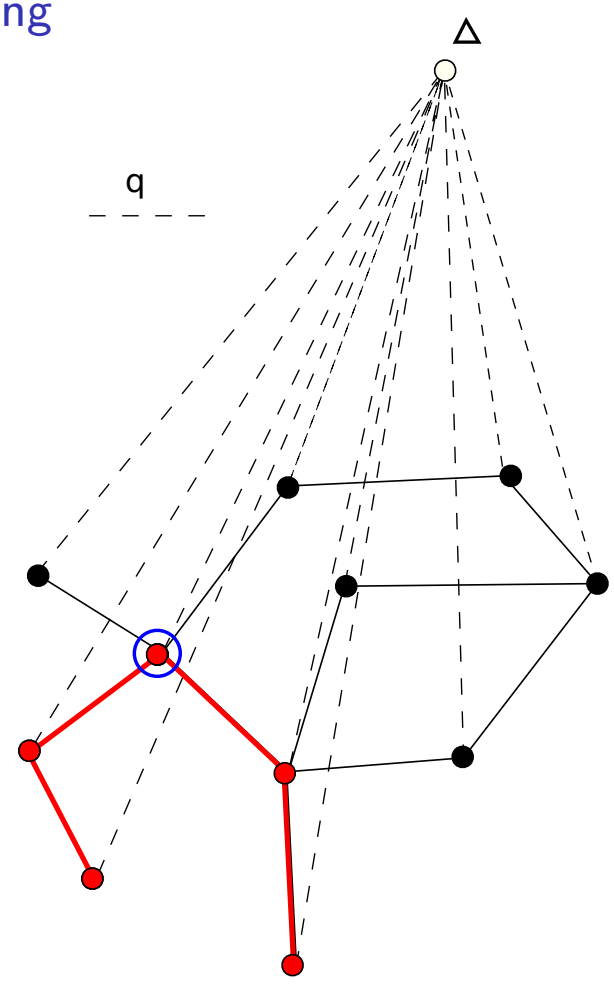
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.



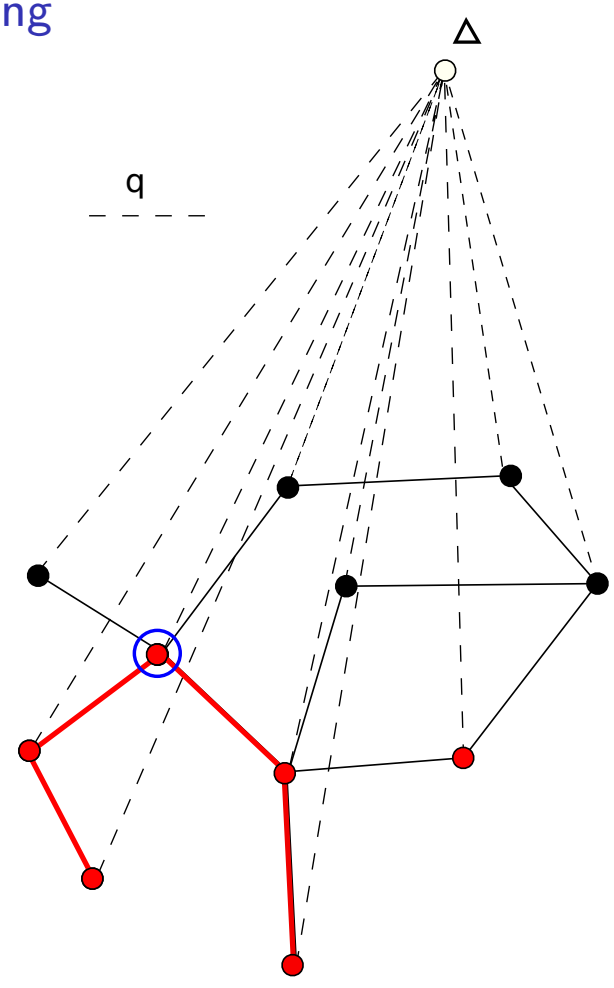
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



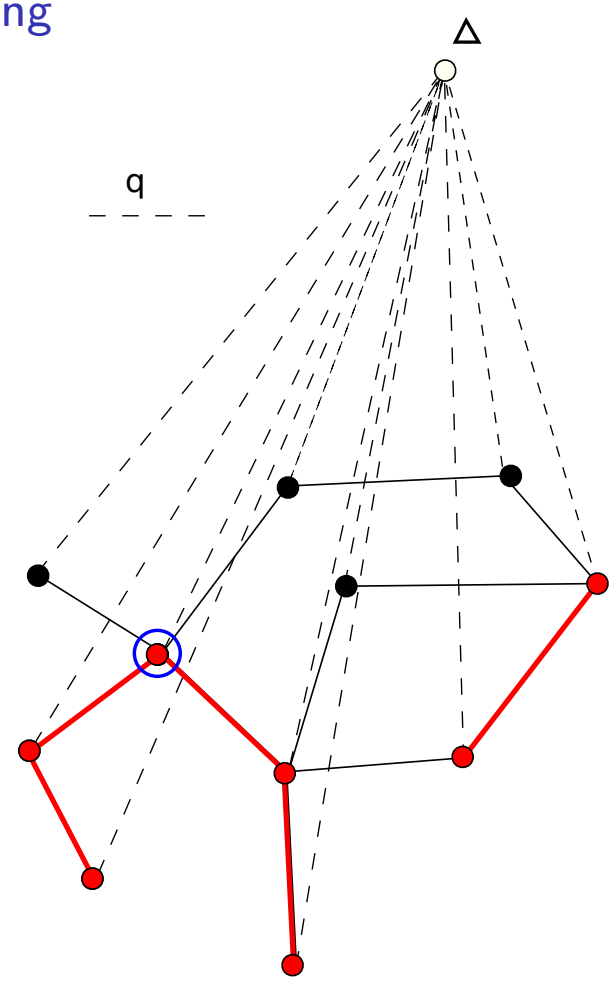
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



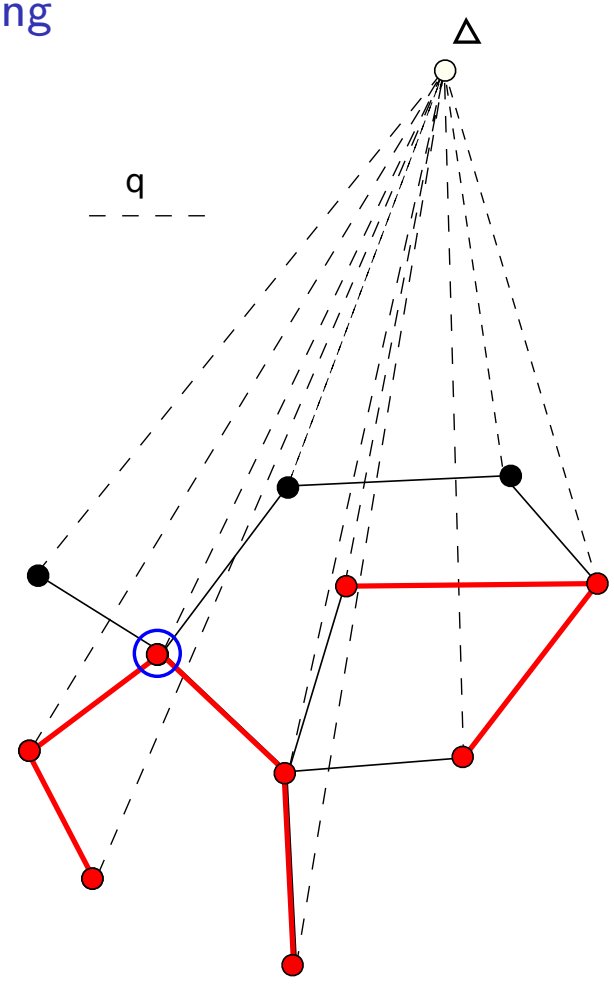
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



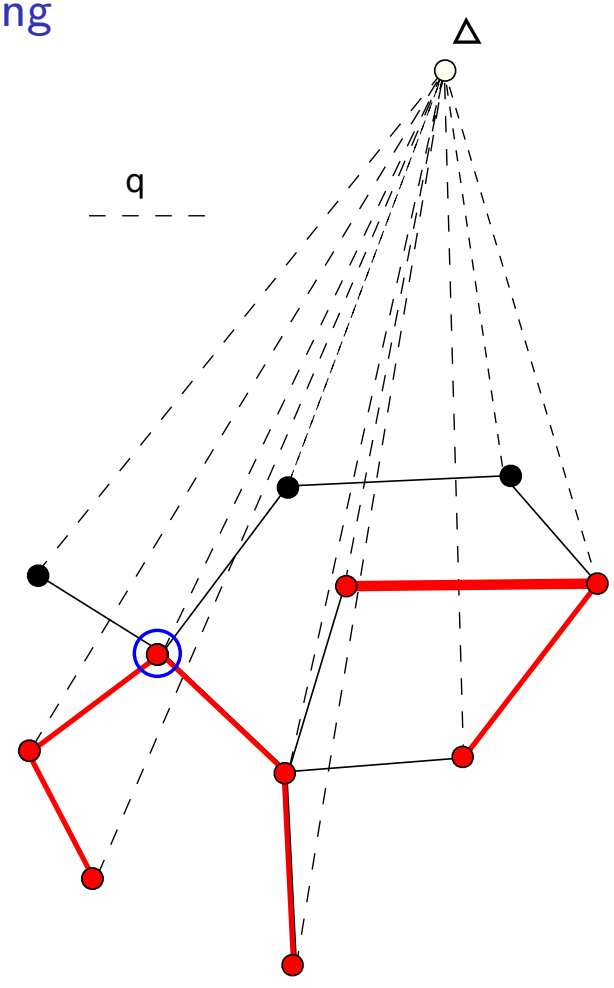
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



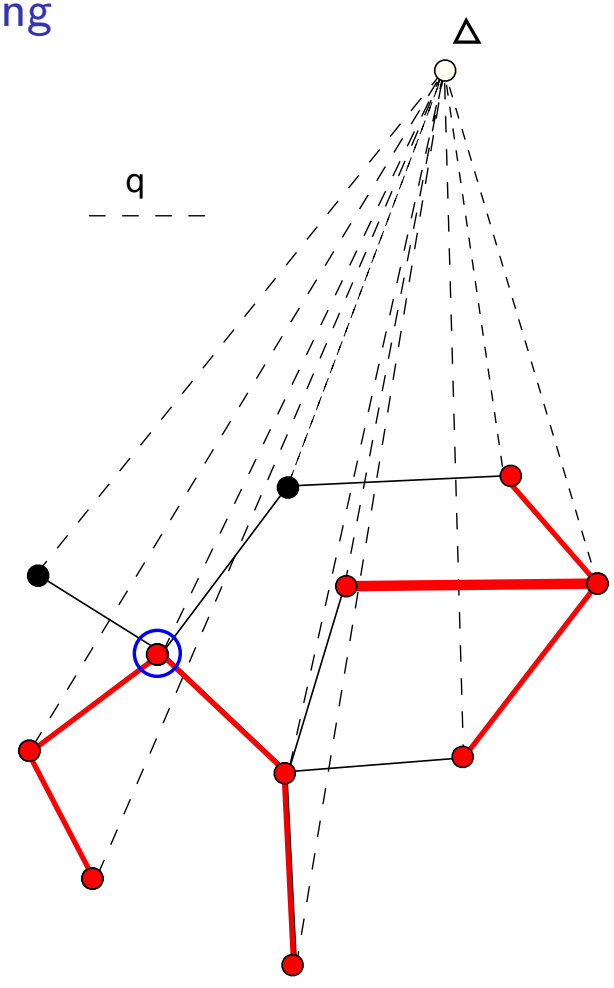
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



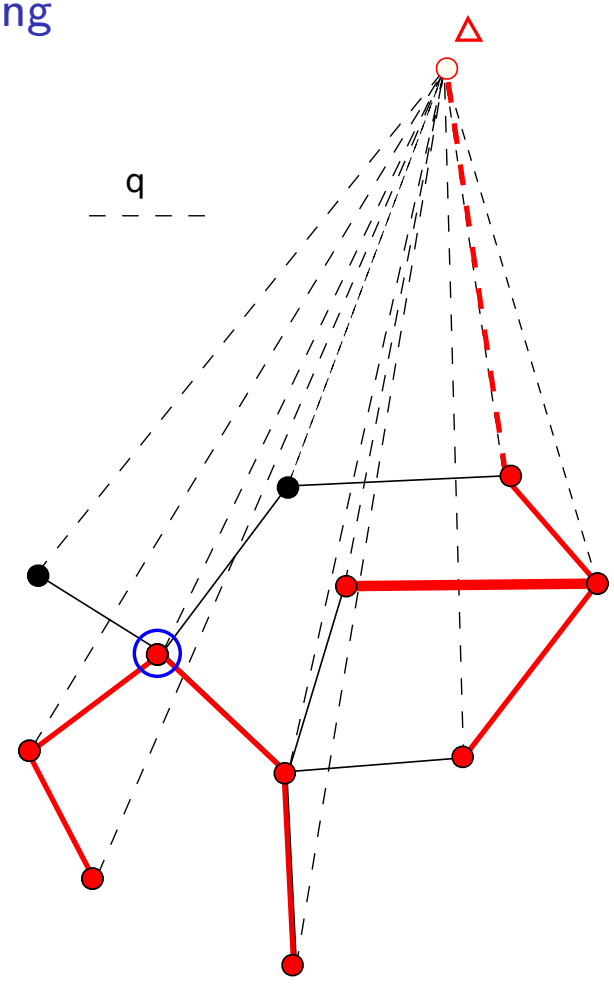
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



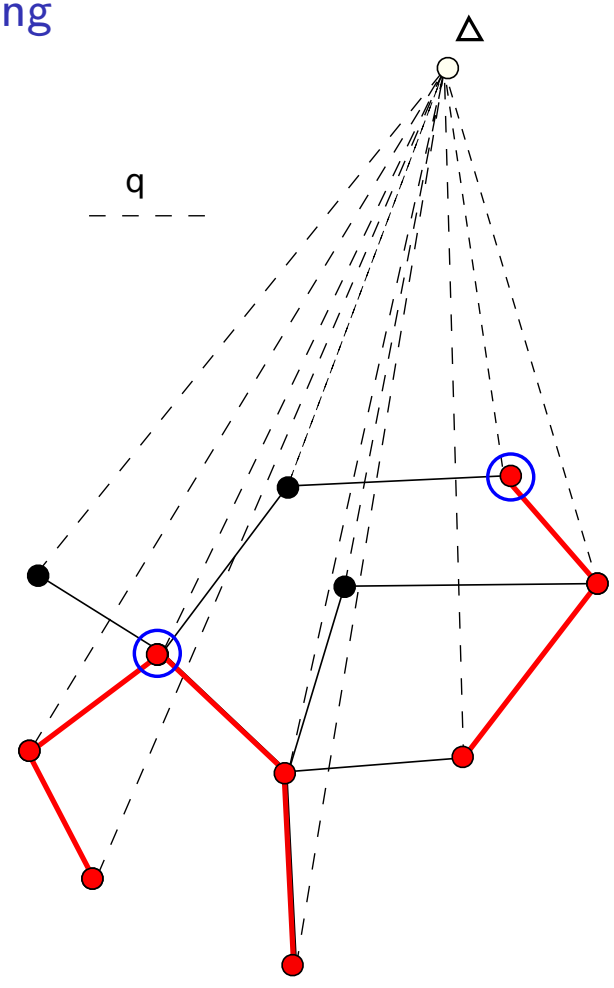
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



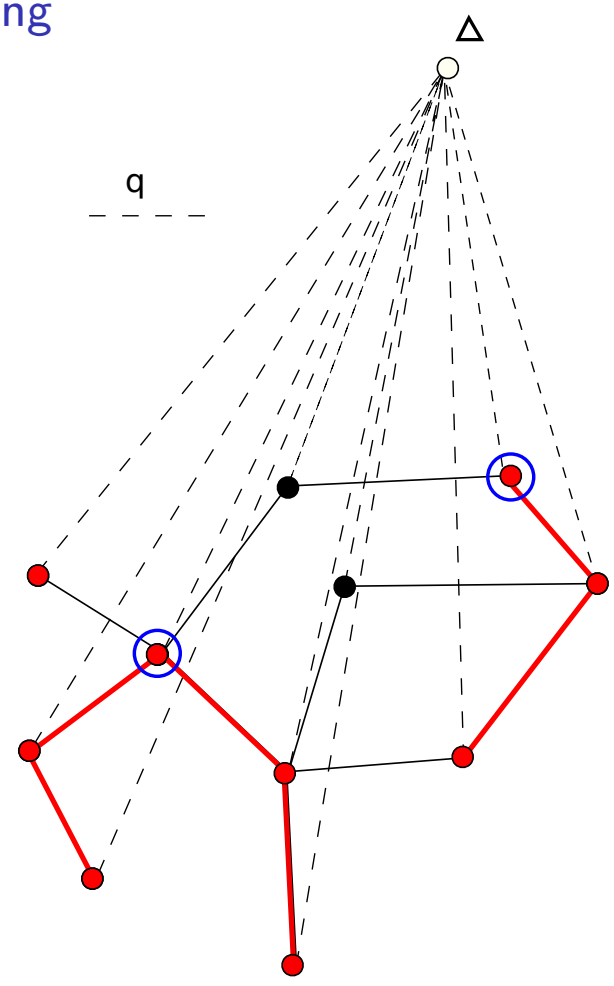
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



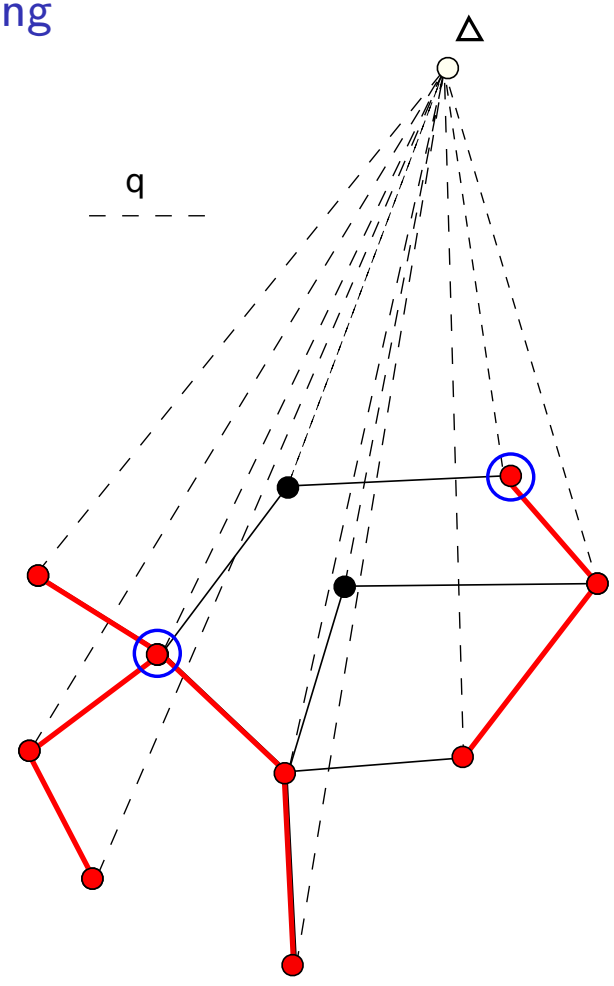
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



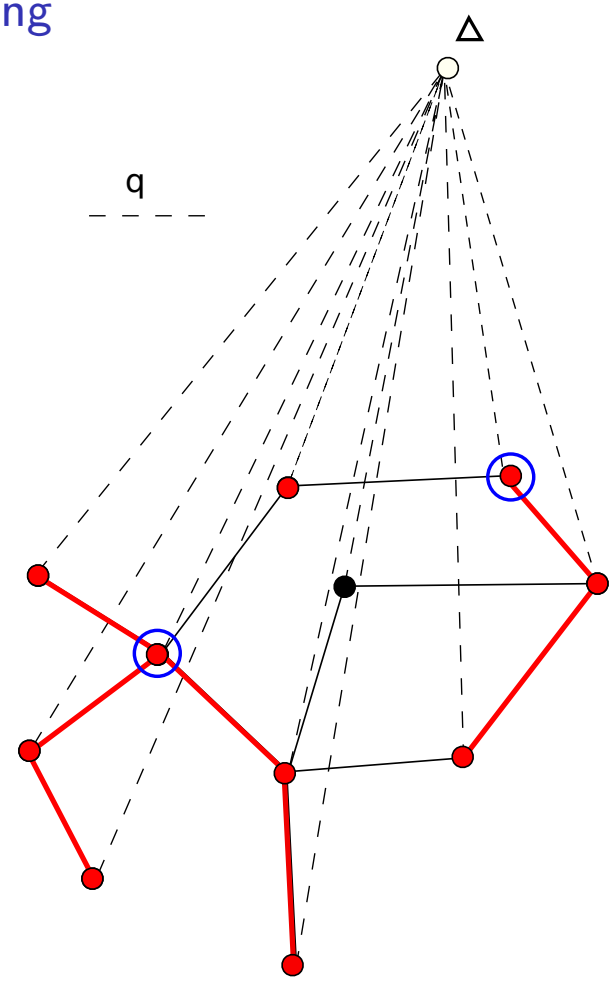
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



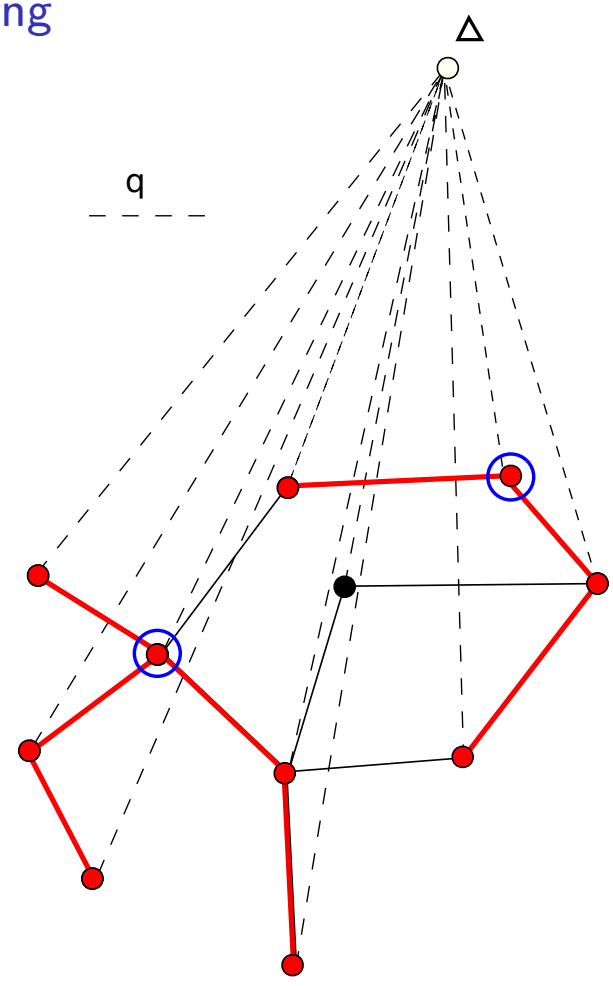
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



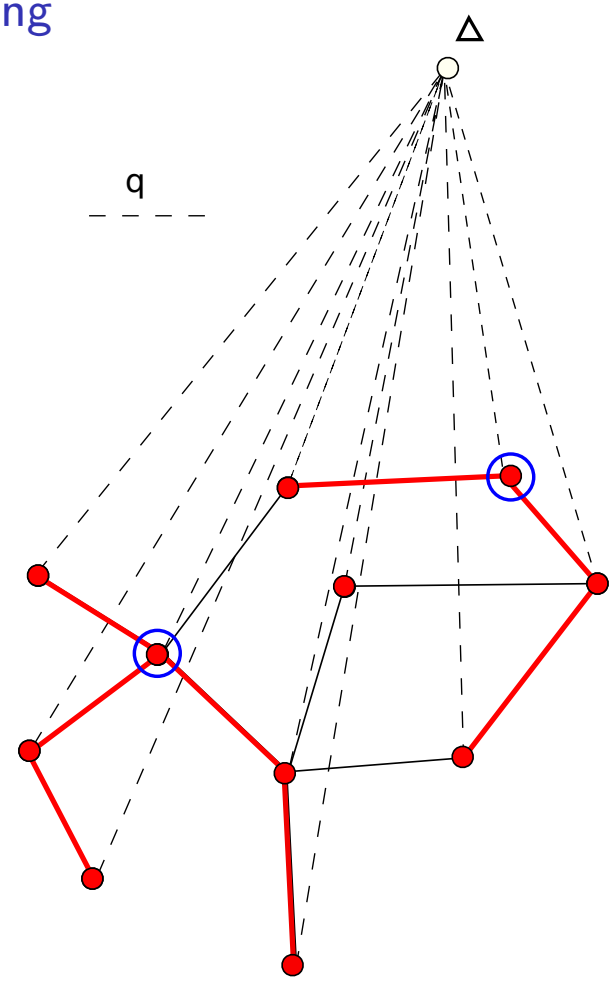
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



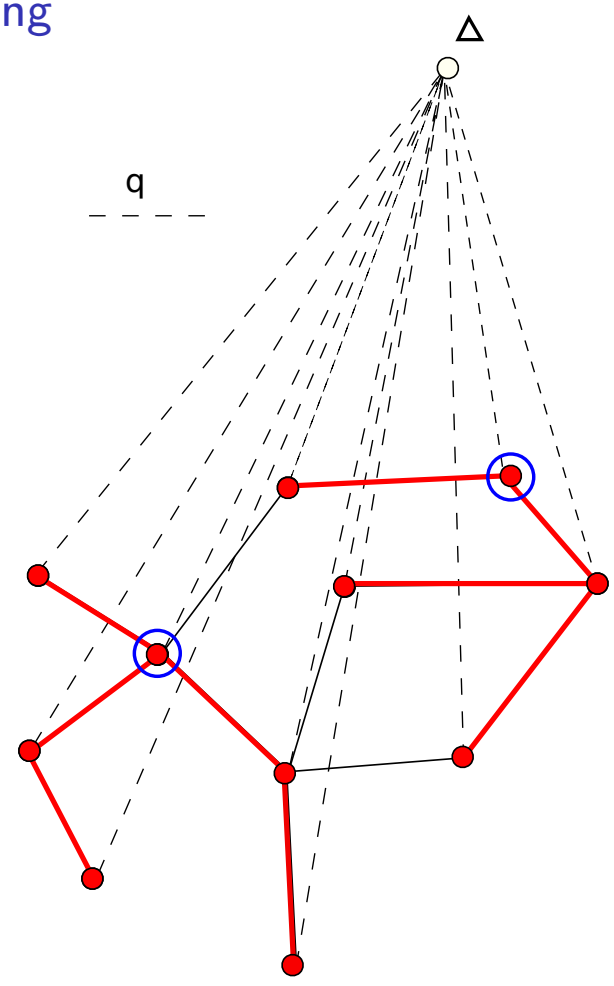
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



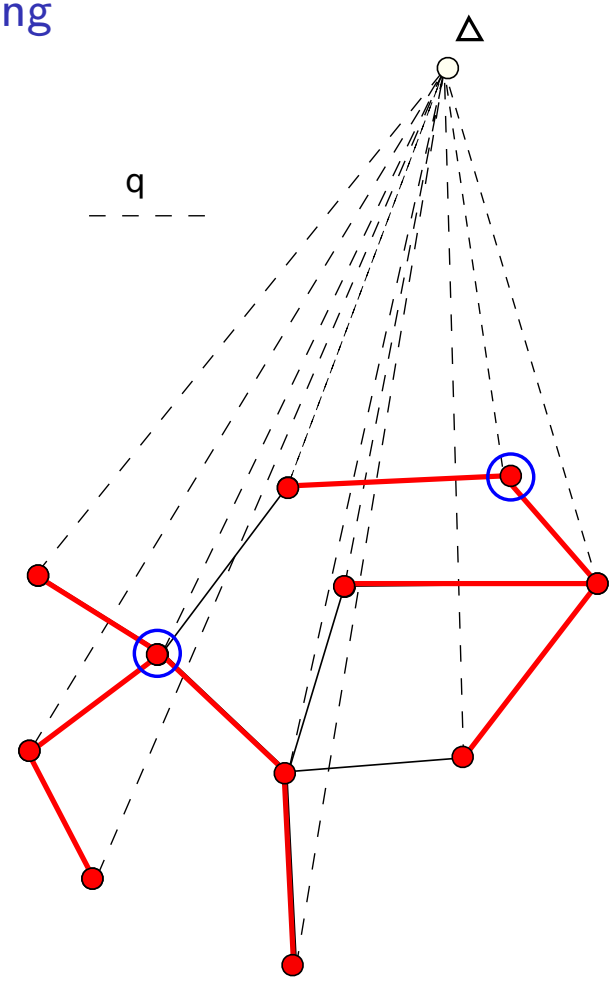
Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



Wilson's algorithm for RSF sampling

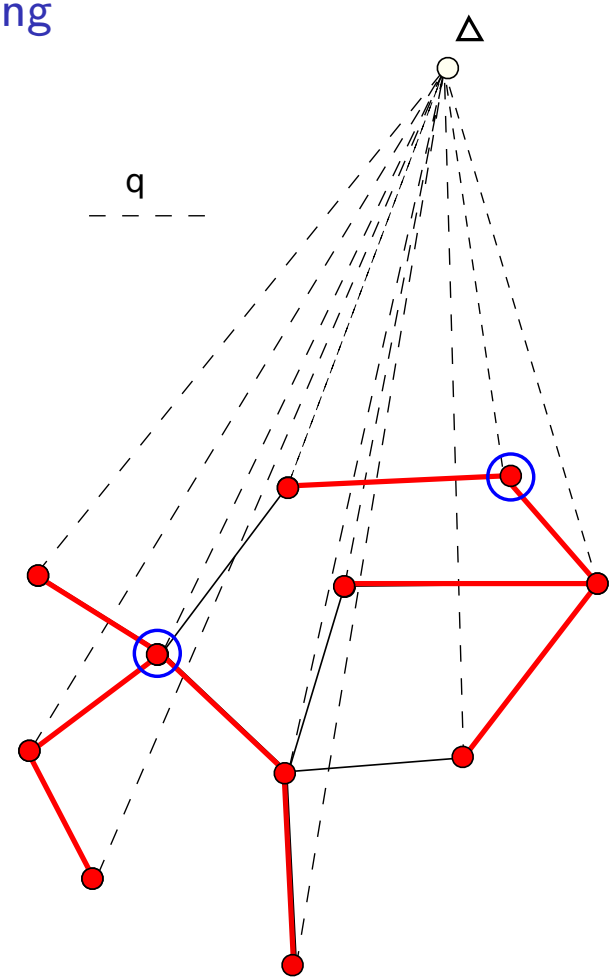
1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



Output: a random spanning forest Φ_q

Wilson's algorithm for RSF sampling

1. add Δ and connect it with all nodes with weight q
2. Start from any node and propagate a random walk until it reaches Δ
3. Erase its eventual loops
4. Keep the selected edges and add the last node before Δ to the set of roots
5. Start a new random walk from any node not yet visited and stop when:
   - it reaches Δ
   - it reaches any already visited node
6. Erase its eventual loops and keep the remaining edges
7. If it "died" in Δ, then add the last visited node before Δ to the set of roots.
8. Repeat 5-7 until all nodes have been visited



Output: a random spanning forest Φ_q

$$\mathbb{E}\,(\text{number of steps}) = \mathrm{Tr}\left[(\mathsf{L}+q\mathsf{I})^{-1}(\mathsf{D}+q\mathsf{I})\right]$$
$$\leq n + 2\frac{|\mathcal{E}|}{q}$$

Two key properties of RSFs

The matrix $\mathrm{K} = (\mathrm{L} + q\mathrm{I})^{-1} q\mathrm{I}$ plays a central role in the theory of random forests. Among other things:

1. the root process $\rho(\Phi_q)$ is a determinantal point process (DPP) with marginal kernel K, implying for instance:
   - K_{ii} is the probability that node i is a root
   - $\mathrm{Tr}\,(\mathrm{K})$ is the expected number of roots
2. the probability that node i is rooted at root j in a random forest Φ_q equals

$$\mathbb{P}\left(r_{\Phi_q}(i) = j\right) = \mathrm{K}_{ij}$$

Contents

From random forests to the inverse trace

- Recall our objective: compute $\mathrm{Tr}\left(\mathrm{K}\right) = \mathrm{Tr}\left[(q\mathrm{I} + \mathrm{L})^{-1} q\mathrm{I}\right]$
- Recall that $\mathrm{Tr}\left(\mathrm{K}\right)$ is the expected number of roots of a RSF Φ_q
- A trivial RSF-based Monte-Carlo estimator of $\mathrm{Tr}\left(\mathrm{K}\right)$ is thus:
  - draw N independent RSFs
  - compute the average of their number of roots
  - it is unbiased and its per-sample variance is $\sum_{i=1}^{n} \frac{q\lambda_i}{(q+\lambda_i)^2}$

From random forests to the inverse trace

- Recall our objective: compute $\mathrm{Tr}\left(\mathrm{K}\right) = \mathrm{Tr}\left[(q\mathrm{I} + \mathrm{L})^{-1} q\mathrm{I}\right]$
- Recall that $\mathrm{Tr}\left(\mathrm{K}\right)$ is the expected number of roots of a RSF Φ_q
- A trivial RSF-based Monte-Carlo estimator of $\mathrm{Tr}\left(\mathrm{K}\right)$ is thus:
  - draw N independent RSFs
  - compute the average of their number of roots
  - it is unbiased and its per-sample variance is $\sum_{i=1}^{n} \frac{q\lambda_i}{(q+\lambda_i)^2}$
- State-of-the-art: Hutchinson's (aka Girard's) estimator:
  - draw N independent random vectors $\boldsymbol{x}$ verifying $\mathbb{E}\left(\boldsymbol{x}\boldsymbol{x}^\top\right) = \mathrm{I}_n$
  - compute the average $\frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}^\top \mathrm{K}\boldsymbol{x}$ (via sparse Cholesky, PCG or poly approx to compute $\mathrm{K}\boldsymbol{x}$)
  - unbiased: $\mathbb{E}\left[\frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}^\top \mathrm{K}\boldsymbol{x}\right] = \mathbb{E}\left(\boldsymbol{x}^\top \mathrm{K}\boldsymbol{x}\right) = \mathbb{E}\left(\mathrm{Tr}\left(\mathrm{K}\boldsymbol{x}\boldsymbol{x}^\top\right)\right) = \mathrm{Tr}\left(\mathrm{K}\right)$
  - per-sample variance in case of Gaussian entries is $\sum_{i=1}^{n} \frac{2q^2}{(q+\lambda_i)^2}$

From random forests to the inverse trace

- Recall our objective: compute $\mathrm{Tr}\left(\mathsf{K}\right) = \mathrm{Tr}\left[(q\mathsf{I} + \mathsf{L})^{-1} q\mathsf{I}\right]$
- Recall that $\mathrm{Tr}\left(\mathsf{K}\right)$ is the expected number of roots of a RSF Φ_q
- A trivial RSF-based Monte-Carlo estimator of $\mathrm{Tr}\left(\mathsf{K}\right)$ is thus:
  - draw N independent RSFs
  - compute the average of their number of roots
  - it is unbiased and its per-sample variance is $\sum_{i=1}^{n} \frac{q\lambda_i}{(q+\lambda_i)^2}$
- State-of-the-art: Hutchinson's (aka Girard's) estimator:
  - draw N independent random vectors $\boldsymbol{x}$ verifying $\mathbb{E}\left(\boldsymbol{x}\boldsymbol{x}^\top\right) = \mathsf{I}_n$
  - compute the average $\frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}^\top \mathsf{K}\boldsymbol{x}$ (via sparse Cholesky, PCG or poly approx to compute $\mathsf{K}\boldsymbol{x}$)
  - unbiased: $\mathbb{E}\left[\frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}^\top \mathsf{K}\boldsymbol{x}\right] = \mathbb{E}\left(\boldsymbol{x}^\top \mathsf{K}\boldsymbol{x}\right) = \mathbb{E}\left(\mathrm{Tr}\left(\mathsf{K}\boldsymbol{x}\boldsymbol{x}^\top\right)\right) = \mathrm{Tr}\left(\mathsf{K}\right)$
  - per-sample variance in case of Gaussian entries is $\sum_{i=1}^{n} \frac{2q^2}{(q+\lambda_i)^2}$
- for large/small enough q, RSF's per-sample variance is better. The precise interval is spectrum-dependent
- both SOTA and RSF-based estimator run in time $\mathcal{O}(|\mathcal{E}|)$

Inverse trace estimation: RSF-based vs. Hutchinson/Girard, pros and cons

Notable advantages of RSF-estimator:

- very easy to implement ($\sim$ 20 lines in Julia)
- minimal memory footprint
- no preprocessing of the graph needed (in fact: no centralized knowledge of L is needed): only need the ability of running a random walk

Notable disadvantages of RSF-estimator:

- only for diagonally-dominant matrices L and $q > 0$
- even if theoretical complexities are comparable, the RSF-estimator cannot take advantage of over-optimized matrix-vector implementations.

Inverse trace estimation: comparison on a few graphs

Five methods to compare...

- *rf* RSF-based estimator
- *direct* uses Julia's backslash operator (calls CHOLMOD)
- *amg* Algebraic Multigrid with Ruge-Stuben coarsening
- *cg* Conjugate Gradient with diagonal preconditioning
- *cg-amg* Conjugate Gradient with AMG preconditioning

Inverse trace estimation: comparison on a few graphs

Five methods to compare...

- *rf* RSF-based estimator
- *direct* uses Julia's backslash operator (calls CHOLMOD)
- *amg* Algebraic Multigrid with Ruge-Stuben coarsening
- *cg* Conjugate Gradient with diagonal preconditioning
- *cg-amg* Conjugate Gradient with AMG preconditioning

... on five different graphs: all have around 10^4 nodes and less than 10^5 edges

Inverse trace estimation: comparison on a few graphs

Five methods to compare...

- *rf* RSF-based estimator
- *direct* uses Julia's backslash operator (calls CHOLMOD)
- *amg* Algebraic Multigrid with Ruge-Stuben coarsening
- *cg* Conjugate Gradient with diagonal preconditioning
- *cg-amg* Conjugate Gradient with AMG preconditioning

... on five different graphs: all have around 10^4 nodes and less than 10^5 edges

For fair comparison, we plot the effective runtime = average time needed per iteration $\times$ the number of iterations needed to reach a relative error of 2%

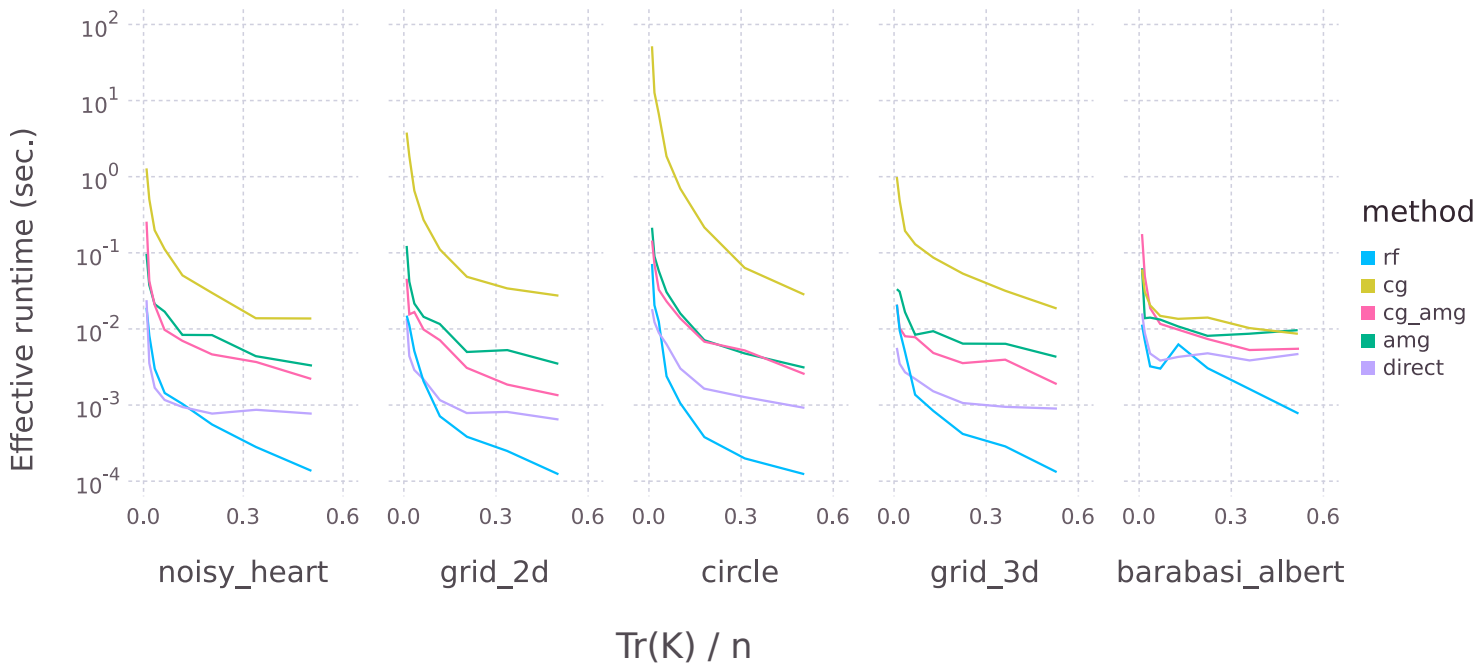
Inverse trace estimation: comparison on a few graphs

Five methods to compare...

- *rf* RSF-based estimator
- *direct* uses Julia's backslash operator (calls CHOLMOD)
- *amg* Algebraic Multigrid with Ruge-Stuben coarsening
- *cg* Conjugate Gradient with diagonal preconditioning
- *cg-amg* Conjugate Gradient with AMG preconditioning

... on five different graphs: all have around 10^4 nodes and less than 10^5 edges

For fair comparison, we plot the effective runtime = average time needed per iteration $\times$ the number of iterations needed to reach a relative error of 2%



Contents

Extensions

- Variance reduction techniques (control variates, stratified sampling, etc.) can be used to further improve the RSF-based estimator
- more importantly, Wilson's algorithm can be modified to sample not only one forest at a given q for a cost of order

$$n + \frac{m}{q}$$

but the whole trajectory of (coupled) forests between $q_{\min}$ and $q_{\max}$; for a cost of order

$$n + \frac{m}{q_{\min}} + n \log \frac{d_{\max}}{q_{\min}}$$

This is very powerful if one wishes to estimate the function $q \to \mathrm{Tr}\left((q\mathrm{I} + \mathrm{L})^{-1} q\right)$ on a given range $[q_{\min}, q_{\max}]$. A precise comparison with state-of-the-art will be the object of a future publication.

Conclusion

- State-of-the-art performance for inverse trace estimation via RSFs
- Computational cost: $\mathcal{O}\left(\frac{|E|}{q}\right)$ will probably require multiscale approximations in small q to become competitive in practice in this range
- Very simple and "natural" algorithm on graphs, VERY low memory footprint, no need of centralized knowledge of the graph, no preprocessing
- Similar works (some published, some in progress in ANR JCJC GRANOLA):
  - i/ RSFs for graph Tikhonov regularization, for effective resistance estimation, for partial spectrum estimation, etc.
    Main research question: what is the extent of the information one can retrieve from a few RSFs?
  - ii/ other similar DPPs over edges and/or nodes of graphs (hopefully "Wilson-able") for similar objectives?

More details?

- A journal paper is here (IEEE TSIPN 2021):

  https://arxiv.org/pdf/2011.10450

- Conference papers are here:
  - (GRETSI 2019):

    https://arxiv.org/pdf/1905.02086
  - (ICASSP 2020):

    https://arxiv.org/pdf/1910.07963
  - (EUSIPCO 2022):

    https://arxiv.org/pdf/2110.07894
  - (GRETSI 2022):

    not online yet
- Julia code publicly available
- These slides can be found on my website:

  gipsa-lab.fr/∼nicolas.tremblay