Random generation of discrete structures

Philippe Duchon

U. Bordeaux - Inria - CNRS

Stochastic geometry - June 25, 2015

- **Topic :** algorithms to generate random (discrete) structures, according to some *prescribed* probability distribution
- Quick overview of two "classes" of methods
  - counting-based methods
  - locally-defined structures, scrambling methods
- Focus on "exact" generation methods, and "geometric" examples

Why random generation ?

- to visualize what "typical" (large) structures in a given class look like
- hints to possible limit behaviors
- to provide test cases for algorithms, when a theoretical average-case analysis is unavailable
- sometimes looking for a good random generation algorithm is a good way of "understanding" the objects under consideration

Model

- Some (finite or countable) family $\mathcal{C}$ of "objects" is defined

Model

- Some (finite or countable) family $\mathcal{C}$ of "objects" is defined
- Some "target" probability distribution μ is defined

Model

- Some (finite or countable) family $\mathcal{C}$ of "objects" is defined
- Some "target" probability distribution μ is defined
- Typically, $\mathcal{C}$ is endowed with a *size function* $|.| : \mathcal{C} \to \mathbb{N}$, with the condition that **for each integer** n**,** $\mathcal{C}_n$ **(set of** $x \in \mathcal{C}$ **with size** n**) is finite**; then $\mu = \mu_n$ can be the uniform distribution over $\mathcal{C}_n$.

Model

- Some (finite or countable) family $\mathcal{C}$ of "objects" is defined
- Some "target" probability distribution μ is defined
- Typically, $\mathcal{C}$ is endowed with a *size function* $|.| : \mathcal{C} \to \mathbb{N}$, with the condition that **for each integer** n**,** $\mathcal{C}_n$ **(set of** $x \in \mathcal{C}$ **with size** n**) is finite**; then $\mu = \mu_n$ can be the uniform distribution over $\mathcal{C}_n$.
- A μ-sampler (μ_n-sampler) is a randomized algorithm that takes no input (n as input) and outputs some random $x \in \mathcal{C}$ according to μ (μ_n).

Model

- Some (finite or countable) family $\mathcal{C}$ of "objects" is defined
- Some "target" probability distribution μ is defined
- Typically, $\mathcal{C}$ is endowed with a *size function* $|.| : \mathcal{C} \to \mathbb{N}$, with the condition that **for each integer** n**,** $\mathcal{C}_n$ **(set of** $x \in \mathcal{C}$ **with size** n**) is finite**; then $\mu = \mu_n$ can be the uniform distribution over $\mathcal{C}_n$.
- A μ-sampler (μ_n-sampler) is a randomized algorithm that takes no input (n as input) and outputs some random $x \in \mathcal{C}$ according to μ (μ_n).
- We assume we have access to some perfect source of randomness (**independent** random bits, **independent** uniform r.v. over $[0, 1]$).

Picking a distribution

- One practical way of defining μ is "proportional to some weight function" $w : \mathcal{C} \to \mathbb{R}^+$:

$$\mu(x) := \frac{w(x)}{\sum_{y \in \mathcal{C}} w(y)}$$

Picking a distribution

- One practical way of defining μ is "proportional to some weight function" $w : \mathcal{C} \to \mathbb{R}^+$:

$$\mu(x) := \frac{w(x)}{\sum_{y \in \mathcal{C}} w(y)}$$

- Requires $S_w = \sum_{y \in \mathcal{C}} w(y) < \infty$

Picking a distribution

- One practical way of defining μ is "proportional to some weight function" $w : \mathcal{C} \to \mathbb{R}^+$:

$$\mu(x) := \frac{w(x)}{\sum_{y \in \mathcal{C}} w(y)}$$

- Requires $S_w = \sum_{y \in \mathcal{C}} w(y) < \infty$
- "Uniform over $\mathcal{C}_n$" as a special case : $w(x) = [|x| = n]$

Rejection principle

A simple, but sometimes efficient idea : "try, reject or accept"

- Assume two weights $w \leq w'$, and "easy" to sample proportionally to w'

Rejection principle

A simple, but sometimes efficient idea : "try, reject or accept"

- Assume two weights $w \leq w'$, and "easy" to sample proportionally to w'
- The **rejection** algorithm :
  - Draw random x, proportionally to $w'(x)$
  - Draw U, uniform on $[0, 1]$
  - If $U > w(x)/w'(x)$ then start over, otherwise output x

Rejection principle

A simple, but sometimes efficient idea : "try, reject or accept"

- Assume two weights $w \leq w'$, and "easy" to sample proportionally to w'
- The **rejection** algorithm :
  - Draw random x, proportionally to $w'(x)$
  - Draw U, uniform on $[0, 1]$
  - If $U > w(x)/w'(x)$ then start over, otherwise output x
- On average : $S_{w'}/S_w$ calls to the w' sampler

Rejection principle

A simple, but sometimes efficient idea : "try, reject or accept"

- Assume two weights $w \leq w'$, and "easy" to sample proportionally to w'
- The **rejection** algorithm :
  - Draw random x, proportionally to $w'(x)$
  - Draw U, uniform on $[0, 1]$
  - If $U > w(x)/w'(x)$ then start over, otherwise output x
- On average : $S_{w'}/S_w$ calls to the w' sampler
- Special case : $\mathcal{A} \subset \mathcal{C}$, where $\mathcal{C}_n$ is easy to sample from and $|\mathcal{A}_n|/|\mathcal{C}_n|$ is "not too small" ; expected number of trials is $|\mathcal{C}_n|/|\mathcal{A}_n|$

Notations

- $\mathcal{C}$: the whole class
- $\mathcal{C}_n$: subclass of objects of size n
- $c_n = |\mathcal{C}_n|$

If we know c_n, it should help generate us get uniform random $x \in \mathcal{C}_n$.

Notations

- $\mathcal{C}$: the whole class
- $\mathcal{C}_n$: subclass of objects of size n
- $c_n = |\mathcal{C}_n|$

If we know c_n, it should help generate us get uniform random $x \in \mathcal{C}_n$.

In many situations, we know c_n but we have no obvious (algorithmic) bijection $\Phi_n : \{1, \ldots, c_n\} \to \mathcal{C}_n$

Classical example : triangulations of a convex polygon

- $n+2$ vertices $1, \ldots, n+2$, ccw on a circle
- $\mathcal{C}_n$: set of triangulations into n triangles

Classical example : triangulations of a convex polygon

- $n+2$ vertices $1, \ldots, n+2$, ccw on a circle
- $\mathcal{C}_n$: set of triangulations into n triangles
- must have a single triangle $\{1, n+2, k\}$, for some $2 \leq k \leq n+1$

Classical example : triangulations of a convex polygon

- $n+2$ vertices $1, \ldots, n+2$, ccw on a circle
- $\mathcal{C}_n$: set of triangulations into n triangles
- must have a single triangle $\{1, n+2, k\}$, for some $2 \leq k \leq n+1$
- the rest must form a triangulation on $\{1, \ldots, k\}$ (size $k-2$) and a triangulation on $\{k, \ldots, n+2\}$ (size $n-k+1$)

Classical example : triangulations of a convex polygon

- $n+2$ vertices $1, \ldots, n+2$, ccw on a circle
- $\mathcal{C}_n$: set of triangulations into n triangles
- must have a single triangle $\{1, n+2, k\}$, for some $2 \leq k \leq n+1$
- the rest must form a triangulation on $\{1, \ldots, k\}$ (size $k-2$) and a triangulation on $\{k, \ldots, n+2\}$ (size $n-k+1$)
- **Consequence :** $c_n = \sum_{k=0}^{n-1} c_k c_{n-1-k}$, $c_0 = 1$.

Classical example : triangulations of a convex polygon

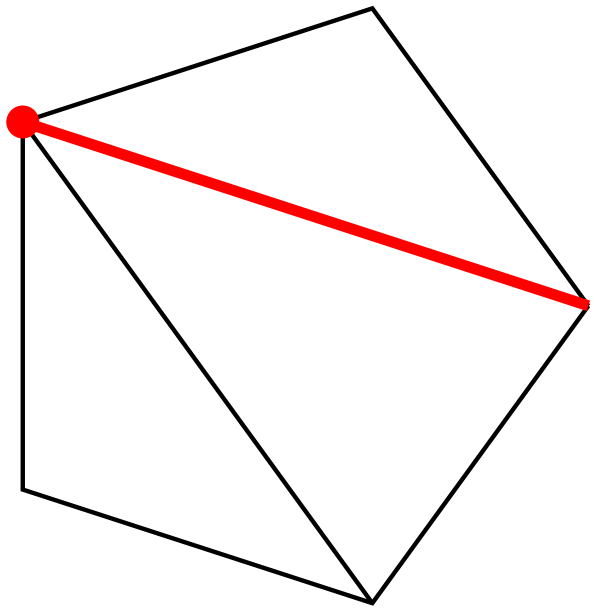
- $n+2$ vertices $1, \ldots, n+2$, ccw on a circle
- $\mathcal{C}_n$: set of triangulations into n triangles
- must have a single triangle $\{1, n+2, k\}$, for some $2 \leq k \leq n+1$
- the rest must form a triangulation on $\{1, \ldots, k\}$ (size $k-2$) and a triangulation on $\{k, \ldots, n+2\}$ (size $n-k+1$)
- **Consequence :** $c_n = \sum_{k=0}^{n-1} c_k c_{n-1-k}$, $c_0 = 1$.
- "Catalan numbers" $c_n = \frac{1}{n+1}\binom{2n}{n}$

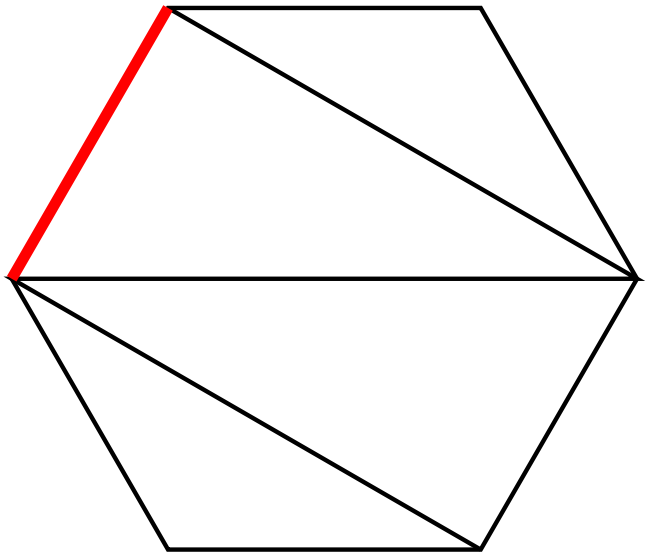
Triangulations : *ad hoc* algorithm

- The Catalan sequence satisfies a simple recursion : $(n+2)c_{n+1} = 2(2n+1)c_n$

Triangulations : *ad hoc* algorithm

- The Catalan sequence satisfies a simple recursion : $(n+2)c_{n+1} = 2(2n+1)c_n$
- Becomes an algorithm for obtaining a uniform triangulation of size $n+1$ from one of size n :
  - pick an edge at random (including border edge : $2n+1$ choices)
  - pick an endpoint at random (2 choices)
  - inflate the edge into a triangle, splitting the chosen endpoint
  - result is a larger triangulation with a marked border edge
  - (adapted from a classic algorithm **[Rémy, 1985]** for binary trees)





Triangulations (cont.)

- $c_0 = 1,\ c_n = \sum_{k=0}^{n-1} c_k c_{n-1-k}$
- Allows to compute $(c_1, \ldots, c_n)$ in $O(n^2)$ arithmetic operations

Triangulations (cont.)

- $c_0 = 1$, $c_n = \sum_{k=0}^{n-1} c_k c_{n-1-k}$
- Allows to compute $(c_1, \ldots, c_n)$ in $O(n^2)$ arithmetic operations (can do better in this case)

Triangulations (cont.)

- $c_0 = 1$, $c_n = \sum_{k=0}^{n-1} c_k c_{n-1-k}$
- Allows to compute $(c_1, \ldots, c_n)$ in $O(n^2)$ arithmetic operations (can do better in this case)
- Leads to **uniform, fixed size** sampling algorithm

GenT(n)

[Precompute $c_0, \ldots, c_n$, once]
If $n = 0$: Return()
Draw a random k, $0 \leq k \leq n-1$, w.p $p_k = c_k c_{n-1-k}/c_n$
Draw $X = \mathrm{GenT}(k)$, $Y = \mathrm{GenT}(n-1-k)$ [with indices shifted by $k-1$]
Return $(\{1, n+2, k\}, X, Y)$

The "recursive" method

[Flajolet, Zimmermann, Van Cutsem 1994] : for a wide variety of classes, information on how objects are "built" from smaller ones translates into recurrences on the sequence $(c_n)_{n\geq 0}$, from which one can

- compute the first $n+1$ terms in the sequence $c_0, \ldots, c_n$
- use the counting sequence to sample uniformly from $\mathcal{C}_n$

The method is widely applicable in a systematic way, and the complexity is $O(n \log n)$ per sample after a more costly precomputation (n numbers, typically growing exponentially).

Example : words without consecutives 1's

- $\mathcal{F}$: set of all words (sequences) over the alphabet $\{0, 1\}$, with the condition that *no two consecutive letters can be 1*.

Example : words without consecutives 1's

- $\mathcal{F}$: set of all words (sequences) over the alphabet $\{0, 1\}$, with the condition that *no two consecutive letters can be 1*.
- **size** of a word is its length.

Example : words without consecutives 1's

- $\mathcal{F}$: set of all words (sequences) over the alphabet $\{0, 1\}$, with the condition that *no two consecutive letters can be 1*.
- **size** of a word is its length.
- Easy recurrence : $f_n = f_{n-1} + f_{n-2}$, $f_0 = 1$, $f_1 = 2$ (shifted Fibonacci sequence).

Example : words without consecutives 1's

- $\mathcal{F}$: set of all words (sequences) over the alphabet $\{0, 1\}$, with the condition that *no two consecutive letters can be 1*.
- **size** of a word is its length.
- Easy recurrence : $f_n = f_{n-1} + f_{n-2}$, $f_0 = 1$, $f_1 = 2$ (shifted Fibonacci sequence).
- Generating function is $F(x) = \frac{1+x}{1-x-x^2}$, radius of convergence is positive root of $1 - x - x^2$ (inverse golden ratio).

Example 2 : binary (plane, rooted) trees

- A binary tree is defined recursively as :
  - either a root/leaf, with size 0
  - or a root, a left subtree t_1 (which is a binary tree), and a right subtree t_2 (also a binary tree) ; size is $|t_1| + |t_2| + 1$

Example 2 : binary (plane, rooted) trees

- A binary tree is defined recursively as :
  - either a root/leaf, with size 0
  - or a root, a left subtree t_1 (which is a binary tree), and a right subtree t_2 (also a binary tree) ; size is $|t_1| + |t_2| + 1$
- The number of binary trees of size n is the Catalan number $C_n = \frac{1}{n+1}\binom{2n}{n}$; $C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$.

Example 2 : binary (plane, rooted) trees

- A binary tree is defined recursively as :
  - either a root/leaf, with size 0
  - or a root, a left subtree t_1 (which is a binary tree), and a right subtree t_2 (also a binary tree) ; size is $|t_1| + |t_2| + 1$
- The number of binary trees of size n is the Catalan number $C_n = \frac{1}{n+1}\binom{2n}{n}$; $C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$.
- (Triangulations are binary trees in disguise)

Example 2 : binary (plane, rooted) trees

- A binary tree is defined recursively as :
  - either a root/leaf, with size 0
  - or a root, a left subtree t_1 (which is a binary tree), and a right subtree t_2 (also a binary tree) ; size is $|t_1| + |t_2| + 1$
- The number of binary trees of size n is the Catalan number $C_n = \frac{1}{n+1}\binom{2n}{n}$; $C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$.
- (Triangulations are binary trees in disguise)
- Other conditions on degrees of nodes lead to different recurrences ; the method carries over

Markov chain methods

- "Easy" to get **convergence** to the target (uniform) distribution

Markov chain methods

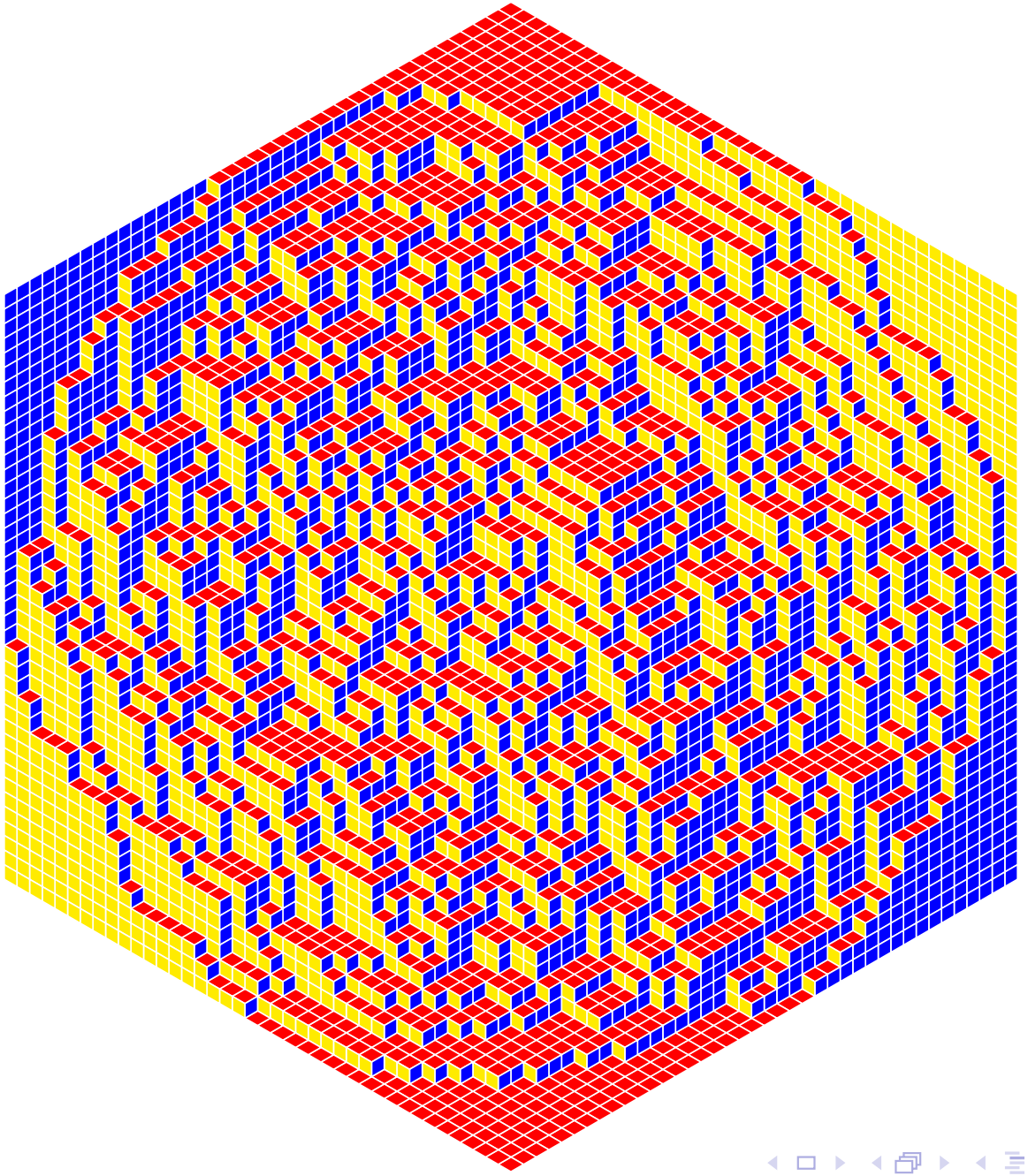
- "Easy" to get **convergence** to the target (uniform) distribution
- "Hard" to get **estimates of the speed of convergence**

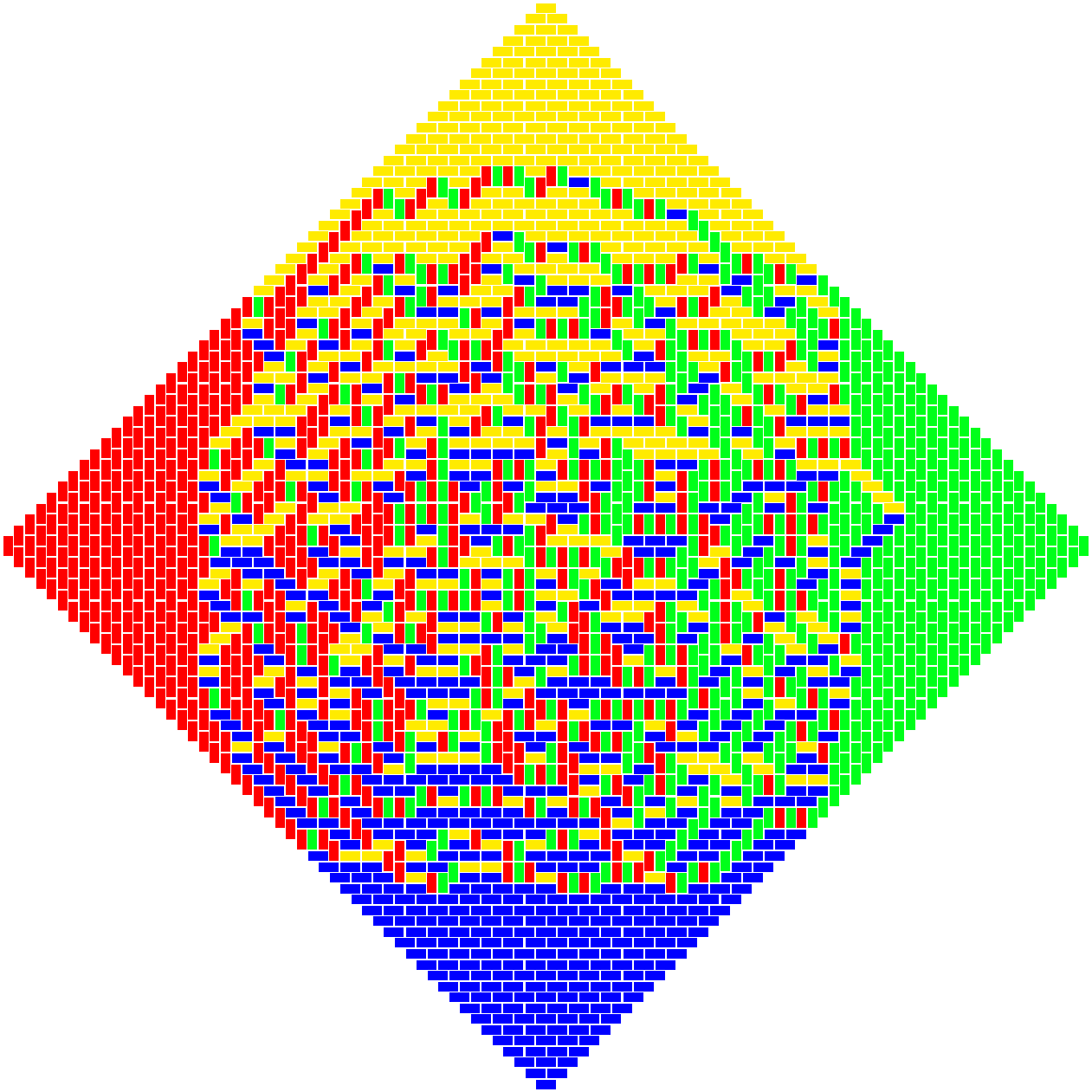
Markov chain methods

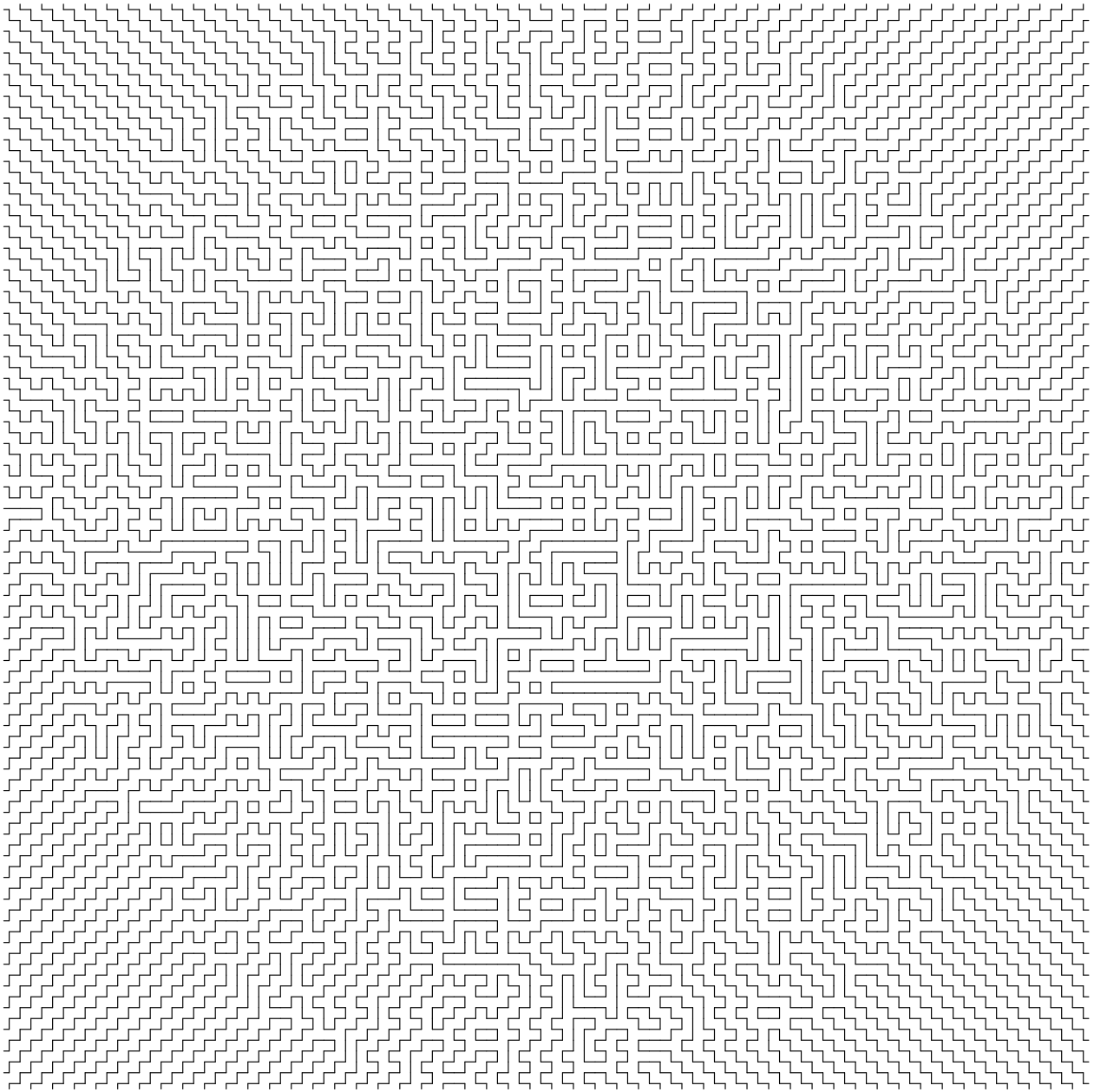
- "Easy" to get **convergence** to the target (uniform) distribution
- "Hard" to get **estimates of the speed of convergence**
- **Sometimes** the "Coupling from the past" technique can give **exact** uniform distribution

Markov chain methods

- "Easy" to get **convergence** to the target (uniform) distribution
- "Hard" to get **estimates of the speed of convergence**
- **Sometimes** the "Coupling from the past" technique can give **exact** uniform distribution
- A few pictures (uniform via CFTP)...







(Biased) random walk in a graph

- $G = (V, E)$ a graph (directed, no vertex of outdegree 0)

(Biased) random walk in a graph

- $G = (V, E)$ a graph (directed, no vertex of outdegree 0)
- for each u, set **weights** (nonnegative, summing to 1) for arcs leaving u

(Biased) random walk in a graph

- $G = (V, E)$ a graph (directed, no vertex of outdegree 0)
- for each u, set **weights** (nonnegative, summing to 1) for arcs leaving u
- **Random walk on** G : "start from some vertex X_0, then at each time $t \in \mathbb{N}$, jump from X_t to a neighbour X_{t+1} chosen at random, according to outgoing weights"

(Biased) random walk in a graph

- $G = (V, E)$ a graph (directed, no vertex of outdegree 0)
- for each u, set **weights** (nonnegative, summing to 1) for arcs leaving u
- **Random walk on** G : "start from some vertex X_0, then at each time $t \in \mathbb{N}$, jump from X_t to a neighbour X_{t+1} chosen at random, according to outgoing weights"
- Implicitly : the choice of next vertex is made **independently** of the previous trajectory ; only "remember" the current vertex

(Biased) random walk in a graph

- $G = (V, E)$ a graph (directed, no vertex of outdegree 0)
- for each u, set **weights** (nonnegative, summing to 1) for arcs leaving u
- **Random walk on** G : "start from some vertex X_0, then at each time $t \in \mathbb{N}$, jump from X_t to a neighbour X_{t+1} chosen at random, according to outgoing weights"
- Implicitly : the choice of next vertex is made **independently** of the previous trajectory ; only "remember" the current vertex
- This is **exactly** what a (homogeneous, finite state) Markov chain is.

Transition matrix

The whole Markov chain is entirely defined by

- the (probability distribution for) initial state : $(\pi_u)_{u \in V}$

Transition matrix

The whole Markov chain is entirely defined by

- the (probability distribution for) initial state : $(\pi_u)_{u \in V}$
- the *transition matrix* M with coefficients

$$p(u, v) = \mathbb{P}(X_{t+1} = v | X_t = u)$$

Transition matrix

The whole Markov chain is entirely defined by

- the (probability distribution for) initial state : $(\pi_u)_{u \in V}$
- the *transition matrix* M with coefficients

$$p(u, v) = \mathbb{P}(X_{t+1} = v | X_t = u)$$

- This is just the (weighted) adjacency matrix !

Transition matrix

The whole Markov chain is entirely defined by

- the (probability distribution for) initial state : $(\pi_u)_{u \in V}$
- the *transition matrix* M with coefficients

$$p(u, v) = \mathbb{P}(X_{t+1} = v | X_t = u)$$

- This is just the (weighted) adjacency matrix !
- The **probability distribution** for X_t (state at time t) is just

$$\pi^{(t)} = \pi . M^t$$

Possible asymptotic behaviors

Important question : $\pi^{(t)}$ for large t ; completely described in terms of the graph G :

- Convergence can only be to an 1-eigenvector for M

Possible asymptotic behaviors

Important question : $\pi^{(t)}$ for large t ; completely described in terms of the graph G :

- Convergence can only be to an 1-eigenvector for M
- Dimension of eigenspace is the number of (sink) strongly connected components (with 0-weight arcs removed) ; each component's stationary probability gives positive probability to each of its states.

Possible asymptotic behaviors

Important question : $\pi^{(t)}$ for large t ; completely described in terms of the graph G :

- Convergence can only be to an 1-eigenvector for M
- Dimension of eigenspace is the number of (sink) strongly connected components (with 0-weight arcs removed) ; each component's stationary probability gives positive probability to each of its states.
- Convergence to some limit is guaranteed (no matter what the initial distribution $\pi^{(0)}$) **if and only if** each (sink) strongly connected component is **aperiodic** (gcd of cycle lengths is 1)

Possible asymptotic behaviors

Important question : $\pi^{(t)}$ for large t ; completely described in terms of the graph G :

- Convergence can only be to an 1-eigenvector for M
- Dimension of eigenspace is the number of (sink) strongly connected components (with 0-weight arcs removed) ; each component's stationary probability gives positive probability to each of its states.
- Convergence to some limit is guaranteed (no matter what the initial distribution $\pi^{(0)}$) **if and only if** each (sink) strongly connected component is **aperiodic** (gcd of cycle lengths is 1)
- Provided the graph is strongly connected and aperiodic, the Markov chain converges to the unique probability distribution, for each possible starting state

Possible asymptotic behaviors

Important question : $\pi^{(t)}$ for large t ; completely described in terms of the graph G :

- Convergence can only be to an 1-eigenvector for M
- Dimension of eigenspace is the number of (sink) strongly connected components (with 0-weight arcs removed) ; each component's stationary probability gives positive probability to each of its states.
- Convergence to some limit is guaranteed (no matter what the initial distribution $\pi^{(0)}$) **if and only if** each (sink) strongly connected component is **aperiodic** (gcd of cycle lengths is 1)
- Provided the graph is strongly connected and aperiodic, the Markov chain converges to the unique probability distribution, for each possible starting state
- (This is all graph-dependent ; only the distribution itself depends on the weights !)

Identifying the limit

(Strongly connected case) unique vector (with sum 1) satisfying, for each u, the "balance condition"

$$\pi_u = \sum_{vu \in E} p(v,u)\pi_v.$$

Identifying the limit

(Strongly connected case) unique vector (with sum 1) satisfying, for each u, the "balance condition"

$$\pi_u = \sum_{vu \in E} p(v,u)\pi_v.$$

Special case : "detailed balance" condition,

$$\pi_u p(u,v) = \pi_v p(v,u)$$

(requires the directed graph to be symmetric)

Identifying the limit

(Strongly connected case) unique vector (with sum 1) satisfying, for each u, the "balance condition"

$$\pi_u = \sum_{vu \in E} p(v,u)\pi_v.$$

Special case : "detailed balance" condition,

$$\pi_u p(u,v) = \pi_v p(v,u)$$

(requires the directed graph to be symmetric)
Special special case : unbiased walk in undirected graph, $p(u,v) = 1/\deg(u)$: π_u is **proportional to the degree of** u. (If the graph is bipartite, the walk is periodic)

What about random generation ?

To use a Markov chain to generate π-random elements from a (finite) class $\mathcal{C}$, you need to

- devise a (strongly connected) graph on vertex set $\mathcal{C}$

What about random generation ?

To use a Markov chain to generate π-random elements from a (finite) class $\mathcal{C}$, you need to

- devise a (strongly connected) graph on vertex set $\mathcal{C}$
- pick weights for arcs that ensure π is stationary

What about random generation ?

To use a Markov chain to generate π-random elements from a (finite) class $\mathcal{C}$, you need to

- devise a (strongly connected) graph on vertex set $\mathcal{C}$
- pick weights for arcs that ensure π is stationary
- ensure aperiodicity : *e.g.* add loops on every state with weight $1/2$ (dividing all other weights by 2)

What about random generation ?

To use a Markov chain to generate π-random elements from a (finite) class $\mathcal{C}$, you need to

- devise a (strongly connected) graph on vertex set $\mathcal{C}$
- pick weights for arcs that ensure π is stationary
- ensure aperiodicity : *e.g.* add loops on every state with weight $1/2$ (dividing all other weights by 2)
- run the chain for a "large" number t of rounds

What about random generation ?

To use a Markov chain to generate π-random elements from a (finite) class $\mathcal{C}$, you need to

- devise a (strongly connected) graph on vertex set $\mathcal{C}$
- pick weights for arcs that ensure π is stationary
- ensure aperiodicity : *e.g.* add loops on every state with weight $1/2$ (dividing all other weights by 2)
- run the chain for a "large" number t of rounds
- output X_t : "close" to π distribution.

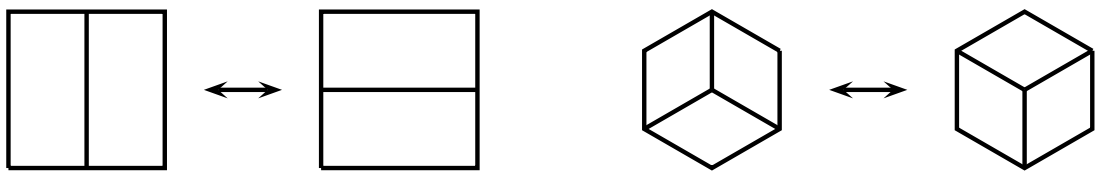
Choosing the graph : adjacences

Typically, choose a symmetric graph where two states (objects) are adjacent if they differ by some "small, local change".

Choosing the graph : adjacences

Typically, choose a symmetric graph where two states (objects) are adjacent if they differ by some "small, local change".
You need a property of the form : any object can be reached from any other by a sequence of such moves.

Sufficient moves for tilings (strongly connected regions)



Choosing transition probabilities

- A good solution is to look for the detailed balance condition : pick $p(u, v)$ and $p(v, u)$ together, with the condition

$$\frac{p(u, v)}{p(v, u)} = \frac{\pi(v)}{\pi(u)}.$$

Choosing transition probabilities

- A good solution is to look for the detailed balance condition : pick $p(u, v)$ and $p(v, u)$ together, with the condition

$$\frac{p(u, v)}{p(v, u)} = \frac{\pi(v)}{\pi(u)}.$$

- If π is uniform over $\mathcal{C}$: just pick $p(u, v) = p(v, u)$.

Running the Markov chain

To simulate the Markov chain for an arbitrary time, you must be able to :

- Pick a starting state (can you construct **one** object from your class ?)

Running the Markov chain

To simulate the Markov chain for an arbitrary time, you must be able to :

- Pick a starting state (can you construct **one** object from your class ?)
- Algorithmically simulate one step : given any state u,
  - compute the list of its neighbours $v_1, \ldots, v_k$
  - compute transition probabilities $p(u, v_i)$
  - pick next state v_i with probability $p(u, v_i)$
  - (or alternatively, pick v_i with probability $p(u, v_i)$ without actually computing the whole list)

How long is long enough ?

- Usually the most difficult question : we want to output X_t, and must choose t such that $\pi^{(t)}$ is close to π.

How long is long enough ?

- Usually the most difficult question : we want to output X_t, and must choose t such that $\pi^{(t)}$ is close to π.
- This is the problem of **mixing time evaluation** :

$$\tau(\epsilon) = \min \left\{ t : d(\pi^{(t)}, \pi) \leq \epsilon \right\}.$$

How long is long enough ?

- Usually the most difficult question : we want to output X_t, and must choose t such that $\pi^{(t)}$ is close to π.
- This is the problem of **mixing time evaluation** :

$$\tau(\epsilon) = \min\left\{t : d(\pi^{(t)}, \pi) \leq \epsilon\right\}.$$

- The **diameter** of the graph is an obvious lower bound.

How long is long enough ?

- Usually the most difficult question : we want to output X_t, and must choose t such that $\pi^{(t)}$ is close to π.
- This is the problem of **mixing time evaluation** :

$$\tau(\epsilon) = \min\left\{t : d(\pi^{(t)}, \pi) \leq \epsilon\right\}.$$

- The **diameter** of the graph is an obvious lower bound.
- Any inequality bounding the **second largest eigenvalue** away from 1 is useful.

How long is long enough ?

- Usually the most difficult question : we want to output X_t, and must choose t such that $\pi^{(t)}$ is close to π.
- This is the problem of **mixing time evaluation** :

$$\tau(\epsilon) = \min \left\{ t : d(\pi^{(t)}, \pi) \leq \epsilon \right\}.$$

- The **diameter** of the graph is an obvious lower bound.
- Any inequality bounding the **second largest eigenvalue** away from 1 is useful.
- For an overview of bounding techniques : **Jerrum** in **Probabilistic Methods for Algorithmic Discrete Mathematics (Springer, 1998)**

Coupling from the past

CFTP [Propp-Wilson, 1996] : a technique to sample from the **exact** distribution π, with a Markov chain that converges to π.

Coupling from the past

CFTP [Propp-Wilson, 1996] : a technique to sample from the **exact** distribution π, with a Markov chain that converges to π. No need to estimate the mixing time : the algorithm stops by itself, and when it does, outputs a π-distributed object.

Generalized coupling

View the simulation of the Markov chain as a two step algorithm :

- Draw a *random update function* $F : V \to V$ from some appropriate distribution
- Apply the function : if current state is x, next state is $F(x)$.

Generalized coupling

View the simulation of the Markov chain as a two step algorithm :

- Draw a *random update function* $F : V \to V$ from some appropriate distribution
- Apply the function : if current state is x, next state is $F(x)$.

The distribution for F must satisfy :

$$\forall (x, y) \in V^2, \Pr(F(x) = y) = p(x, y).$$

Generalized coupling

View the simulation of the Markov chain as a two step algorithm :

- Draw a *random update function* $F : V \to V$ from some appropriate distribution
- Apply the function : if current state is x, next state is $F(x)$.

The distribution for F must satisfy :

$$\forall (x, y) \in V^2, \Pr(F(x) = y) = p(x, y).$$

As a byproduct, this defines a "generalized coupling" of the Markov chain : one copy $(X_t^{(u)})_{t \geq 0}$ starting from each state u, with the "sticky" property

$$X_t^{(u)} = X_t^{(v)} \Rightarrow \forall t' > t, X_{t'}^{(u)} = X_{t'}^{(v)}.$$

Note on update functions

For a given transition matrix, one can design many different distributions for transition functions.

- Images can be chosen independently (extremely costly !)

Note on update functions

For a given transition matrix, one can design many different distributions for transition functions.

- Images can be chosen independently (extremely costly !)
- A "good" design will try to make it more likely that chains starting from different states will reach the same state.

Exact, but useless, simulation algorithm

For any integer t, here is an exact simulation algorithm for π :

- Draw t independent update functions $F_1, \ldots, F_n$;
- Compute $G = F_n \circ \cdots \circ F_1$;
- Draw a random initial state u from distribution π ;
- Output $G(u)$.

Exact, but useless, simulation algorithm

For any integer t, here is an exact simulation algorithm for π :

- Draw t independent update functions $F_1, \ldots, F_n$;
- Compute $G = F_n \circ \cdots \circ F_1$;
- Draw a random initial state u from distribution π ;
- Output $G(u)$.

(Useless : if we know how to choose u, we don't need a more complex algorithm)

But. . .

If we make the right choice for the distribution of F, it is very likely that, for large t, the composite function G is a **constant function** over V; then the result **does not depend on choice of** u.

But. . .

If we make the right choice for the distribution of F, it is very likely that, for large t, the composite function G is a **constant function** over V ; then the result **does not depend on choice of** u.
Warning : there is a trap

Forward coupling (to the future)

(Run the coupling until coalescence)

Forward coupling (to the future)

(Run the coupling until coalescence)

- $G \leftarrow I$, $u \leftarrow u_0$
- While G is not constant, $F \leftarrow \text{RandomF}()$; $G \leftarrow F \circ G$; $u \leftarrow F(u)$
- Return u

Forward coupling (to the future)

(Run the coupling until coalescence)

- $G \leftarrow I$, $u \leftarrow u_0$
- While G is not constant, $F \leftarrow \text{RandomF}()$; $G \leftarrow F \circ G$; $u \leftarrow F(u)$
- Return u

This is a forward coupling : after t steps, $G = G_t = F_t \circ \cdots \circ F_1$; $G_t(u) = \text{RandomF}()(G_{t-1}(u))$.

Backward coupling (from the future)

- $G \leftarrow I$
- While G is not constant, $G \leftarrow G \circ \text{RandomF}()$
- Return $G(u_0)$

Backward coupling (from the future)

- $G \leftarrow I$
- While G is not constant, $G \leftarrow G \circ \text{RandomF}()$
- Return $G(u_0)$

This is backward coupling : $G_t = F_1 \circ \cdots \circ F_t$; renaming F_i as F_{-i}, $G_t = F_{-1} \circ F_{-2} \circ \cdots \circ F_{-t}$.

Backward coupling (from the future)

- $G \leftarrow I$
- While G is not constant, $G \leftarrow G \circ \text{RandomF}()$
- Return $G(u_0)$

This is backward coupling : $G_t = F_1 \circ \cdots \circ F_t$; renaming F_i as F_{-i}, $G_t = F_{-1} \circ F_{-2} \circ \cdots \circ F_{-t}$.
$G_t(u) = G_{t-1}(\text{RandomF}(u))$: to compute an image, compositions happen in the wrong order !

Backward coupling (from the future)

- $G \leftarrow I$
- While G is not constant, $G \leftarrow G \circ \text{RandomF}()$
- Return $G(u_0)$

This is backward coupling : $G_t = F_1 \circ \cdots \circ F_t$; renaming F_i as F_{-i}, $G_t = F_{-1} \circ F_{-2} \circ \cdots \circ F_{-t}$.
$G_t(u) = G_{t-1}(\text{RandomF}(u))$: to compute an image, compositions happen in the wrong order !
View as : Take a coupling that has already run for an infinite time, it must have become coalescent at time 0 ; we are simply looking into its recent past to discover its state at time 0.

Here is the trap

- Forward coupling does **not**, in general, simulate distribution π ;

Here is the trap

- Forward coupling does **not**, in general, simulate distribution π ;
- Backward coupling does simulate distribution π, **provided** it has **positive probability** to terminate (this implies probability 1).

Example : walk on a line

$V = \{1, \ldots, k\}$, $p(i, i+1) = p(i, i-1) = 1/2$,
$p(0, 0) = p(k, k) = 1/2$

Example : walk on a line

$V = \{1, \ldots, k\}$, $p(i, i+1) = p(i, i-1) = 1/2$,
$p(0,0) = p(k,k) = 1/2$
$\pi(i) = 1/k$, $i = 1 \ldots k$ (uniform)

Example : walk on a line

$V = \{1, \ldots, k\}$, $p(i, i+1) = p(i, i-1) = 1/2$,
$p(0,0) = p(k,k) = 1/2$
$\pi(i) = 1/k$, $i = 1 \ldots k$ (uniform)
Realize coupling with 2 update functions : $F^+(i) = \min(k, i+1)$;
$F^-(i) = \max(1, i-1)$

Example : walk on a line

$V = \{1, \ldots, k\}$, $p(i, i+1) = p(i, i-1) = 1/2$,
$p(0,0) = p(k,k) = 1/2$
$\pi(i) = 1/k$, $i = 1 \ldots k$ (uniform)
Realize coupling with 2 update functions : $F^+(i) = \min(k, i+1)$;
$F^-(i) = \max(1, i-1)$
Forward coupling will always stop with a constant function 1 or k, so will never output any other value !

Why CFTP is correct

Consider a doubly infinite sequence of independent random update functions $(F_n)_{n\in\mathbb{Z}}$, and set $(n < m)$

$$G_{n,m} = F_{m-1} \circ F_{m-2} \circ \cdots \circ F_n$$

Why CFTP is correct

Consider a doubly infinite sequence of independent random update functions $(F_n)_{n\in\mathbb{Z}}$, and set $(n < m)$

$$G_{n,m} = F_{m-1} \circ F_{m-2} \circ \cdots \circ F_n$$

- As a random function, $G_{n,m}$ leaves distribution π invariant ;

Why CFTP is correct

Consider a doubly infinite sequence of independent random update functions $(F_n)_{n\in\mathbb{Z}}$, and set $(n < m)$

$$G_{n,m} = F_{m-1} \circ F_{m-2} \circ \cdots \circ F_n$$

- As a random function, $G_{n,m}$ leaves distribution π invariant ;
- With probability 1, there exists some $n < 0$ s.t. $G_{n,0}$ is a constant function ;

Why CFTP is correct

Consider a doubly infinite sequence of independent random update functions $(F_n)_{n\in\mathbb{Z}}$, and set $(n < m)$

$$G_{n,m} = F_{m-1} \circ F_{m-2} \circ \cdots \circ F_n$$

- As a random function, $G_{n,m}$ leaves distribution π invariant ;
- With probability 1, there exists some $n < 0$ s.t. $G_{n,0}$ is a constant function ;
- if $G_{n,0}$ is constant, $G_{n',0} = G_{n,0}$ for all $n' < n$;

Why CFTP is correct

Consider a doubly infinite sequence of independent random update functions $(F_n)_{n\in\mathbb{Z}}$, and set $(n < m)$

$$G_{n,m} = F_{m-1} \circ F_{m-2} \circ \cdots \circ F_n$$

- As a random function, $G_{n,m}$ leaves distribution π invariant ;
- With probability 1, there exists some $n < 0$ s.t. $G_{n,0}$ is a constant function ;
- if $G_{n,0}$ is constant, $G_{n',0} = G_{n,0}$ for all $n' < n$;
- thus, for all u,

$$\lim_{n\to+\infty} \mathbb{P}(G_{-n,0} = u) = \pi_u$$

Why CFTP is correct

Consider a doubly infinite sequence of independent random update functions $(F_n)_{n\in\mathbb{Z}}$, and set $(n < m)$

$$G_{n,m} = F_{m-1} \circ F_{m-2} \circ \cdots \circ F_n$$

- As a random function, $G_{n,m}$ leaves distribution π invariant ;
- With probability 1, there exists some $n < 0$ s.t. $G_{n,0}$ is a constant function ;
- if $G_{n,0}$ is constant, $G_{n',0} = G_{n,0}$ for all $n' < n$;
- thus, for all u,

$$\lim_{n\to+\infty} \mathbb{P}(G_{-n,0} = u) = \pi_u$$

(This is a monotone convergence argument ; where forward coupling fails is that we do not have $G_{0,n'} = G_{0,n}$ as soon as $G_{0,n}$ is constant and $n' > n$)

Practical versions of CFTP

- We do not need to compute $G_{n,0}$ completely, only to detect (possibly with some delay) that $G_{n,0}$ constant ;

Practical versions of CFTP

- We do not need to compute $G_{n,0}$ completely, only to detect (possibly with some delay) that $G_{n,0}$ constant ;
- **Monotone CFTP :** whenever V is a partially ordered set with some minimum and maximum elements, and update functions F are monotone increasing, coalescence is equivalent to $G(u) = G(v)$ for all extremal elements (only compute their images)

Practical versions of CFTP

- We do not need to compute $G_{n,0}$ completely, only to detect (possibly with some delay) that $G_{n,0}$ constant ;
- **Monotone CFTP :** whenever V is a partially ordered set with some minimum and maximum elements, and update functions F are monotone increasing, coalescence is equivalent to $G(u) = G(v)$ for all extremal elements (only compute their images)
- In particular, if V has a unique minimum and maximum (*e.g.*, a finite distributive lattice), only need to compute $G_{n,0}(\max)$ and $G_{n,0}(\min)$; (most easy cases are of this type)

Practical versions of CFTP

- We do not need to compute $G_{n,0}$ completely, only to detect (possibly with some delay) that $G_{n,0}$ constant ;
- **Monotone CFTP :** whenever V is a partially ordered set with some minimum and maximum elements, and update functions F are monotone increasing, coalescence is equivalent to $G(u) = G(v)$ for all extremal elements (only compute their images)
- In particular, if V has a unique minimum and maximum (*e.g.*, a finite distributive lattice), only need to compute $G_{n,0}(\max)$ and $G_{n,0}(\min)$; (most easy cases are of this type)
- **binary-backoff CFTP :** compute $G_{-2^k,0}$ for $k = 1, 2, \ldots,$ storing all functions F_n so as to be able to reuse them ; this way, composition always happen in the natural order.