Discrete logs and other collisions

Tanja Lange

Guest lecture at National Taiwan University

13 April 2026

Diffie–Hellman key exchange

- 1976 Diffie and Hellman introduce public-key cryptography.
- To use it, standardize group G and $g \in G$.
  Everybody knows G and g as well as how to compute in G.
- Warning #1: Many G are unsafe!

Diffie–Hellman key exchange

- 1976 Diffie and Hellman introduce public-key cryptography.
- To use it, standardize group G and $g \in G$.
  Everybody knows G and g as well as how to compute in G.
- Warning #1: Many G are unsafe!
  - $G = (\mathbf{Q}, \cdot), g = 2$, $h_A = 65536$

Diffie–Hellman key exchange

- 1976 Diffie and Hellman introduce public-key cryptography.
- To use it, standardize group G and $g \in G$.
  Everybody knows G and g as well as how to compute in G.
- Warning #1: Many G are unsafe!
  - $G = (\mathbf{Q}, \cdot), g = 2$, $h_A = 65536$ means $a = 16$.
    In general, just check bitlength.
  - $G = (\mathbf{F}_p, +)$, i.e., A sends $h_A \equiv ag \bmod p$.

Diffie–Hellman key exchange

- 1976 Diffie and Hellman introduce public-key cryptography.
- To use it, standardize group G and $g \in G$.
  Everybody knows G and g as well as how to compute in G.
- Warning #1: Many G are unsafe!
  - $G = (\mathbf{Q}, \cdot), g = 2$, $h_A = 65536$ means $a = 16$.
    In general, just check bitlength.
  - $G = (\mathbf{F}_p, +)$, i.e., A sends $h_A \equiv ag \bmod p$.
    Can recover a using XGCD.
- Diffie and Hellman suggested $G = (\mathbf{F}_p^*, \cdot)$ with g a primitive element, i.e., a generator of the whole group.

Diffie–Hellman key exchange

- 1976 Diffie and Hellman introduce public-key cryptography.
- To use it, standardize group G and $g \in G$.
  Everybody knows G and g as well as how to compute in G.
- Warning #1: Many G are unsafe!
  - $G = (\mathbf{Q}, \cdot), g = 2$, $h_A = 65536$ means $a = 16$.
    In general, just check bitlength.
  - $G = (\mathbf{F}_p, +)$, i.e., A sends $h_A \equiv ag \bmod p$.
    Can recover a using XGCD.
- Diffie and Hellman suggested $G = (\mathbf{F}_p^*, \cdot)$ with g a primitive element, i.e., a generator of the whole group.
- Used in practice $G \subset (\mathbf{F}_p^*, \cdot)$ with g an element of large prime order.
- Miller and Koblitz suggested $G = E(\mathbf{F}_p, +)$, i,e., points on an elliptic curve over a finite field with addition of points.

Diffie–Hellman key exchange

- 1976 Diffie and Hellman introduce public-key cryptography.
- To use it, standardize group G and $g \in G$.
  Everybody knows G and g as well as how to compute in G.
- Warning #1: Many G are unsafe!
  - $G = (\mathbf{Q}, \cdot), g = 2$, $h_A = 65536$ means $a = 16$.
    In general, just check bitlength.
  - $G = (\mathbf{F}_p, +)$, i.e., A sends $h_A \equiv ag \bmod p$.
    Can recover a using XGCD.
- Diffie and Hellman suggested $G = (\mathbf{F}_p^*, \cdot)$ with g a primitive element, i.e., a generator of the whole group.
- Used in practice $G \subset (\mathbf{F}_p^*, \cdot)$ with g an element of large prime order.
- Miller and Koblitz suggested $G = E(\mathbf{F}_p, +)$, i,e., points on an elliptic curve over a finite field with addition of points.
- Used in practice $G \subset E(\mathbf{F}_p, +)$, i,e., prime-order subgroup of points on an elliptic curve over a finite field with addition of points.

Hardness assumptions

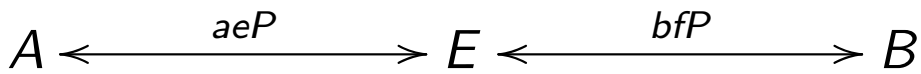
- Computational Diffie-Hellman Problem (CDHP):
  Given P, aP, bP compute abP.
- Decisional Diffie-Hellman Problem (DDHP):
  Given P, aP, bP, and cP decide whether $cP = abP$.
- Discrete Logarithm Problem (DLP):
  Given P, aP, compute a.
- If one can solve DLP, then CDHP and DDHP are easy.

Hardness assumptions

- Computational Diffie-Hellman Problem (CDHP):
  Given P, aP, bP compute abP.
- Decisional Diffie-Hellman Problem (DDHP):
  Given P, aP, bP, and cP decide whether $cP = abP$.
- Discrete Logarithm Problem (DLP):
  Given P, aP, compute a.
- If one can solve DLP, then CDHP and DDHP are easy.
- If one can solve CDHP, then DDHP is easy.
- In many groups, DLP and CDHP are equally hard
  (up to some constants).
- In some groups, DDHP is significantly easier than CDHP.

Practical problems

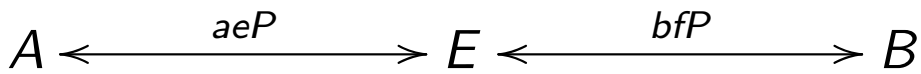
- Eve can set up a *man-in-the-middle* attack:



  - E chooses e and f, presents eP to Alice as Bob's key, and fP to Bob as Alice's key.
  - E computes both DH keys aeP and bfP.
  - E decrypts everything from A and reencrypts it to B and vice versa.

Practical problems

- Eve can set up a *man-in-the-middle* attack:



  - E chooses e and f, presents eP to Alice as Bob's key, and fP to Bob as Alice's key.
  - E computes both DH keys aeP and bfP.
  - E decrypts everything from A and reencrypts it to B and vice versa.
  - This attack requires E to be in charge of the network.
    We typically assume such strong attackers.
- This attack cannot be detected unless A and B compare their keys out of band.

Semi-static DH

- A cryptosystem combining public-key and symmetric-key crypto is called a *hybrid system*[1].
- Alice publishes long-term public key $P_A = aP$,
  keeps long-term private key a.
- Any user can encrypt to Alice using this key:
  - Pick random k and compute $R = kP$.
  - Encrypt message m using symmetric keys derived from $\mathrm{KDF}(kP_A)$, for key-derivation function $\mathrm{KDF} : G \to \{0,1\}^n$,
  - Send ciphertext c along with R.
  - Alice decrypts, by obtaining symmetric key from

$$\mathrm{KDF}(aR) = \mathrm{KDF}(akP)$$

[1] Now also used for combining pre-quantum and post-quantum systems.

Semi-static DH

- A cryptosystem combining public-key and symmetric-key crypto is called a *hybrid system*[1].
- Alice publishes long-term public key $P_A = aP$,
  keeps long-term private key a.
- Any user can encrypt to Alice using this key:
  - Pick random k and compute $R = kP$.
  - Encrypt message m using symmetric keys derived from $\mathrm{KDF}(kP_A)$,
    for key-derivation function $\mathrm{KDF} : G \to \{0,1\}^n$,
  - Send ciphertext c along with R.
  - Alice decrypts, by obtaining symmetric key from

$$\mathrm{KDF}(aR) = \mathrm{KDF}(akP)$$

- Alice's key here is static, Bob's key is ephemeral.
- Note: ephemeral does not mean one-time;
  it means that is not long term.

[1] Now also used for combining pre-quantum and post-quantum systems.

Semi-static DH

- A cryptosystem combining public-key and symmetric-key crypto is called a *hybrid system*[1].
- Alice publishes long-term public key $P_A = aP$,
  keeps long-term private key a.
- Any user can encrypt to Alice using this key:
  - Pick random k and compute $R = kP$.
  - Encrypt message m using symmetric keys derived from $\mathrm{KDF}(kP_A)$,
    for key-derivation function $\mathrm{KDF} : G \to \{0,1\}^n$,
  - Send ciphertext c along with R.
  - Alice decrypts, by obtaining symmetric key from

$$\mathrm{KDF}(aR) = \mathrm{KDF}(akP)$$

- Alice's key here is static, Bob's key is ephemeral.
- Note: ephemeral does not mean one-time;
  it means that is not long term.
- Attacker solving DLP or CDHP can *compute* shared secret.
  Attacker solving DDHP can *confirm* guess.

[1] Now also used for combining pre-quantum and post-quantum systems.

First approaches to solving DLP

Notation and target

Setting: cyclic group $G = \langle P \rangle$ with n elements;
group operation written additively on these slides.

Given $Q \in G$ as well as P and how to compute in G.

Target: Compute $a = \log_P Q$, i.e. a such that $Q = aP$.
a is called the discrete logarithm of Q to base P.
We typically take $0 \leq a < n$.

For elliptic curves, point counting over $\mathbf{F}_p$ runs in time polynomial in $\log p$.

Number of points in $[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$.

The group is isomorphic to $\mathbf{Z}/m \times \mathbf{Z}/n$, where $m \mid n$ and $m \mid (p - 1)$.

Running example

$p = 1000003$, Weierstrass curve $y^2 = x^3 - x$ over $\mathbf{F}_p$.

This curve has

Running example

$p = 1000003$, Weierstrass curve $y^2 = x^3 - x$ over $\mathbf{F}_p$.

This curve has $1000004 = 2^2 \cdot 53^2 \cdot 89$ points.
$P = (101384, 614510)$ is a point of order $2 \cdot 53^2 \cdot 89$.

Running example

$p = 1000003$, Weierstrass curve $y^2 = x^3 - x$ over $\mathbf{F}_p$.

This curve has $1000004 = 2^2 \cdot 53^2 \cdot 89$ points.
$P = (101384, 614510)$ is a point of order $2 \cdot 53^2 \cdot 89$.

Can we find an integer $a \in \{1, 2, 3, \ldots, 500001\}$ such that $aP = (670366, 740819)$?

This point was generated as a multiple of P.
Random point could also be outside cyclic group.

Could find a by brute force.

Running example

$p = 1000003$, Weierstrass curve $y^2 = x^3 - x$ over $\mathbf{F}_p$.

This curve has $1000004 = 2^2 \cdot 53^2 \cdot 89$ points.
$P = (101384, 614510)$ is a point of order $2 \cdot 53^2 \cdot 89$.

Can we find an integer $a \in \{1, 2, 3, \ldots, 500001\}$ such that
$aP = (670366, 740819)$?

This point was generated as a multiple of P.
Random point could also be outside cyclic group.

Could find a by brute force.
$1P = (101384, 614510)$, $2P = (102361, 628914)$, $3P = (77571, 87643)$,
$4P = (650289, 31313)$, $\ldots$, $500001P = (101384, 385493) = -P$.
$500002P = \infty$.

At some point we'll find a with $aP = (670366, 740819)$.

Maximum cost of computation: ≤ 500001 additions of P;

Smarter brute force attacks

This naive brute force attack takes up to n steps in general.

Smarter brute force attacks

This naive brute force attack takes up to n steps in general.

Computation has a good chance of finishing earlier.

Chance scales linearly:
$1/2$ chance of $n/2$ cost; $1/10$ chance of $n/10$ cost; etc.

"So users should choose large a."

Smarter brute force attacks

This naive brute force attack takes up to n steps in general.

Computation has a good chance of finishing earlier.

Chance scales linearly:
$1/2$ chance of $n/2$ cost; $1/10$ chance of $n/10$ cost; etc.

"So users should choose large a."

Actually, no. We can apply "random self-reduction":
choose random r, compute rP, compute $(a+r)P = aP + rP$,
compute discrete log of this point.

Smarter brute force attacks

This naive brute force attack takes up to n steps in general.

Computation has a good chance of finishing earlier.

Chance scales linearly:
$1/2$ chance of $n/2$ cost; $1/10$ chance of $n/10$ cost; etc.

"So users should choose large a."

Actually, no. We can apply "random self-reduction":
choose random r, compute rP, compute $(a+r)P = aP + rP$,
compute discrete log of this point. Obtain a from $a+r$ by subtracting r.

Example: $r = 69961$, compute $rP = (593450, 987590)$, compute
$(a+r)P = (670366, 740819) + (593450, 987590)$.

Smarter brute force attacks

This naive brute force attack takes up to n steps in general.

Computation has a good chance of finishing earlier.

Chance scales linearly:
$1/2$ chance of $n/2$ cost; $1/10$ chance of $n/10$ cost; etc.

"So users should choose large a."

Actually, no. We can apply "random self-reduction":
choose random r, compute rP, compute $(a+r)P = aP + rP$,
compute discrete log of this point. Obtain a from $a+r$ by subtracting r.

Example: $r = 69961$, compute $rP = (593450, 987590)$, compute
$(a+r)P = (670366, 740819) + (593450, 987590)$.

Choosing large, random r moves target around.

Many targets

Computation can be applied to many targets at once.

Given 100 DL targets a_1P, a_2P, ..., $a_{100}P$:
Can find *all* of $a_1, a_2, \ldots, a_{100}$ with $\leq n - 1$ ADDs.

Simplest approach:
First build a sorted table containing $a_1P, a_2P, \ldots, a_{100}P$.
Then check table for $1P$, $2P$, etc.

Many targets

Computation can be applied to many targets at once.

Given 100 DL targets a_1P, a_2P, ..., $a_{100}P$:
Can find *all* of $a_1, a_2, \ldots, a_{100}$ with $\leq n - 1$ ADDs.

Simplest approach:
First build a sorted table containing $a_1P, a_2P, \ldots, a_{100}P$.
Then check table for $1P$, $2P$, etc.

Interesting consequence #1:
Solving all 100 DL problems costs about as much as solving one.

Interesting consequence #2:
Solving *at least one* out of 100 DL problems is much faster than solving one specific DL problem.

First DL found after roughly $n/100$ ADDs.

Many targets out of one

Computation can be applied to many targets at once.

Given 100 DL targets a_1P, a_2P, ..., $a_{100}P$:
Can find *all* of $a_1, a_2, \ldots, a_{100}$ with $\leq n-1$ ADDs.

Simplest approach:
First build a sorted table containing $a_1P, a_2P, \ldots, a_{100}P$.
Then check table for $1P$, $2P$, etc.

Interesting consequence #1:
Solving all 100 DL problems costs about as much as solving one.

Interesting consequence #2:
Solving *at least one* out of 100 DL problems is much faster than solving one specific DL problem.

First DL found after roughly $n/100$ ADDs.

Can turn one DLP into 100 to benefit: compute $mP = \lfloor n/100 \rfloor P$.
New targets: $aP, (a+m)P, (a+2m)P, \ldots (a+99m)P$.

Many targets out of one

Computation can be applied to many targets at once.

Given 100 DL targets a_1P, a_2P, ..., $a_{100}P$:
Can find *all* of $a_1, a_2, \ldots, a_{100}$ with $\leq n-1$ ADDs.

Simplest approach:
First build a sorted table containing $a_1P, a_2P, \ldots, a_{100}P$.
Then check table for $1P$, $2P$, etc.

Interesting consequence #1:
Solving all 100 DL problems costs about as much as solving one.

Interesting consequence #2:
Solving *at least one* out of 100 DL problems is much faster than solving one specific DL problem.

First DL found after roughly $n/100$ ADDs.

Can turn one DLP into 100 to benefit: compute $mP = \lfloor n/100 \rfloor P$.
New targets: $aP, (a+m)P, (a+2m)P, \ldots (a+99m)P$.
Save a factor 100.

Many targets out of one

Computation can be applied to many targets at once.

Given 100 DL targets a_1P, a_2P, ..., $a_{100}P$:
Can find *all* of $a_1, a_2, \ldots, a_{100}$ with $\leq n-1$ ADDs.

Simplest approach:
First build a sorted table containing $a_1P, a_2P, \ldots, a_{100}P$.
Then check table for $1P$, $2P$, etc.

Interesting consequence #1:
Solving all 100 DL problems costs about as much as solving one.

Interesting consequence #2:
Solving *at least one* out of 100 DL problems is much faster than solving one specific DL problem.

First DL found after roughly $n/100$ ADDs.

Can turn one DLP into 100 to benefit: compute $mP = \lfloor n/100 \rfloor P$.
New targets: $aP, (a+m)P, (a+2m)P, \ldots (a+99m)P$.
Save a factor 100. Save factor 200, 300, 400, ... 1000, 1100, 1200,

Many targets out of one

Computation can be applied to many targets at once.

Given 100 DL targets a_1P, a_2P, ..., $a_{100}P$:
Can find *all* of $a_1, a_2, \ldots, a_{100}$ with $\leq n-1$ ADDs.

Simplest approach:
First build a sorted table containing $a_1P, a_2P, \ldots, a_{100}P$.
Then check table for $1P$, $2P$, etc.

Interesting consequence #1:
Solving all 100 DL problems costs about as much as solving one.

Interesting consequence #2:
Solving *at least one* out of 100 DL problems is much faster than solving one specific DL problem.

First DL found after roughly $n/100$ ADDs.

Can turn one DLP into 100 to benefit: compute $mP = \lfloor n/100 \rfloor P$.
New targets: $aP, (a+m)P, (a+2m)P, \ldots (a+99m)P$.
Save a factor 100. Save factor 200, 300, 400, ...1000, ~~1100, 1200 ...~~
Cannot save more than $\sqrt{n}$, else computing targets dominates cost.

Baby-step giant-step algorithm

Systematize and optimize this approach.
Let $0 < m < n$ then there exist $0 \leq a_0 < m$ and $0 \leq a_1 \leq n/m + 1$ with

$$a = a_0 + ma_1.$$

Compute all possibilities for a_0P. This costs m steps and m space.

We first do the small steps to get mP with just one addition.

$a = a_0 + ma_1$ means $aP = (a_0 + ma_1)P$, thus

$$a_0P = aP - a_1mP$$

Each step adds -mP (remember, $-(x, y) = (x, -y)$ on Weierstrass).

Baby-step giant-step algorithm

Systematize and optimize this approach.
Let $0 < m < n$ then there exist $0 \leq a_0 < m$ and $0 \leq a_1 \leq n/m + 1$ with

$$a = a_0 + ma_1.$$

Compute all possibilities for $a_0 P$. This costs m steps and m space.

We first do the small steps to get mP with just one addition.

$a = a_0 + ma_1$ means $aP = (a_0 + ma_1)P$, thus

$$a_0 P = aP - a_1 mP$$

Each step adds -mP (remember, $-(x, y) = (x, -y)$ on Weierstrass).
Finding this match takes at most $n/m + 1$ steps.

These targets are evenly spaced, so we are guaranteed a match.

We balance both for $m \sim \sqrt{n}$ to get to a total cost of $2\sqrt{n}+$ some small constant expenses, hence the cost of $O(\sqrt{n})$.

Note that this also costs $\sqrt{n}$ in storage.

Baby-step giant-step algorithm

Systematize and optimize this approach.
Let $0 < m < n$ then there exist $0 \leq a_0 < m$ and $0 \leq a_1 \leq n/m + 1$ with

$$a = a_0 + ma_1.$$

Compute all possibilities for $a_0 P$. This costs m steps and m space.

We first do the small steps to get mP with just one addition.

$a = a_0 + ma_1$ means $aP = (a_0 + ma_1)P$, thus

$$a_0 P = aP - a_1 mP$$

Each step adds -mP (remember, $-(x, y) = (x, -y)$ on Weierstrass).
Finding this match takes at most $n/m + 1$ steps.

These targets are evenly spaced, so we are guaranteed a match.

We balance both for $m \sim \sqrt{n}$ to get to a total cost of $2\sqrt{n}+$ some small constant expenses, hence the cost of $O(\sqrt{n})$.

Note that this also costs $\sqrt{n}$ in storage.

This is a typical divide-and-conquer speedup.
Search for these whenever doing a brute-force search.

Random walks and cycle finding

Random walks

Target $Q = aP$ in $\langle P \rangle$. Group has n elements.

Make a pseudo-random walk in the group $\langle P \rangle$,
where the next step depends on current point: $W_{i+1} = f(W_i)$.

Birthday paradox:
Randomly choosing from n elements picks one element twice after about

Random walks

Target $Q = aP$ in $\langle P \rangle$. Group has n elements.

Make a pseudo-random walk in the group $\langle P \rangle$,
where the next step depends on current point: $W_{i+1} = f(W_i)$.

Birthday paradox:
Randomly choosing from n elements picks one element twice after about $\sqrt{\pi n/2}$ draws.

Random walks

Target $Q = aP$ in $\langle P \rangle$. Group has n elements.

Make a pseudo-random walk in the group $\langle P \rangle$,
where the next step depends on current point: $W_{i+1} = f(W_i)$.

Birthday paradox:
Randomly choosing from n elements picks one element twice after about $\sqrt{\pi n/2}$ draws.

Assume that for each point we know $0 \leq a_i, b_i < n$ so that $W_i = a_i P + b_i Q$.

Then $W_i = W_j$ means that

$$a_i P + b_i Q = a_j P + b_j Q$$

Random walks

Target $Q = aP$ in $\langle P \rangle$. Group has n elements.

Make a pseudo-random walk in the group $\langle P \rangle$,
where the next step depends on current point: $W_{i+1} = f(W_i)$.

Birthday paradox:
Randomly choosing from n elements picks one element twice after about $\sqrt{\pi n/2}$ draws.

Assume that for each point we know $0 \leq a_i, b_i < n$ so that $W_i = a_i P + b_i Q$.

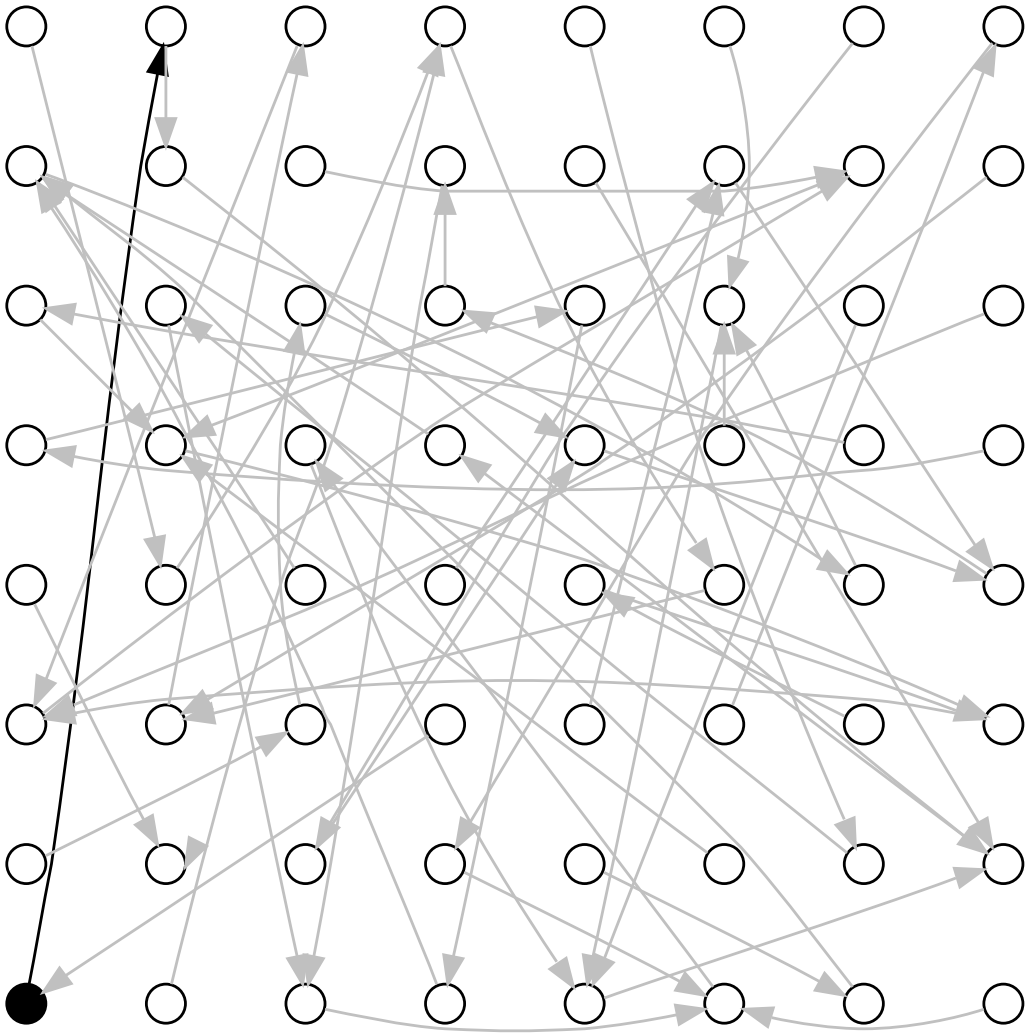
Then $W_i = W_j$ means that

$$a_i P + b_i Q = a_j P + b_j Q \Leftrightarrow (b_i - b_j)Q = (a_j - a_i)P.$$

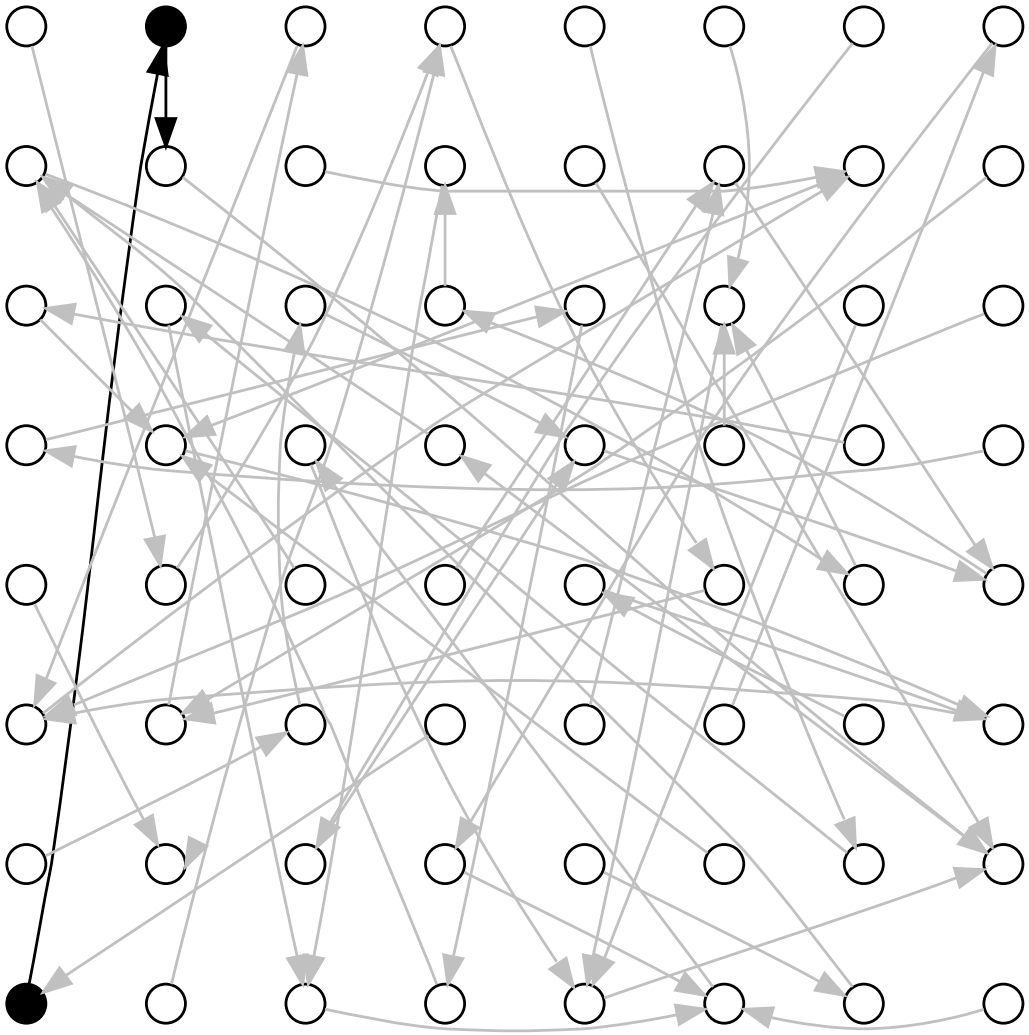
If $b_i - b_j$ invertible modulo n, the DLP is solved:

$$a \equiv (a_j - a_i)/(b_i - b_j) \bmod n$$

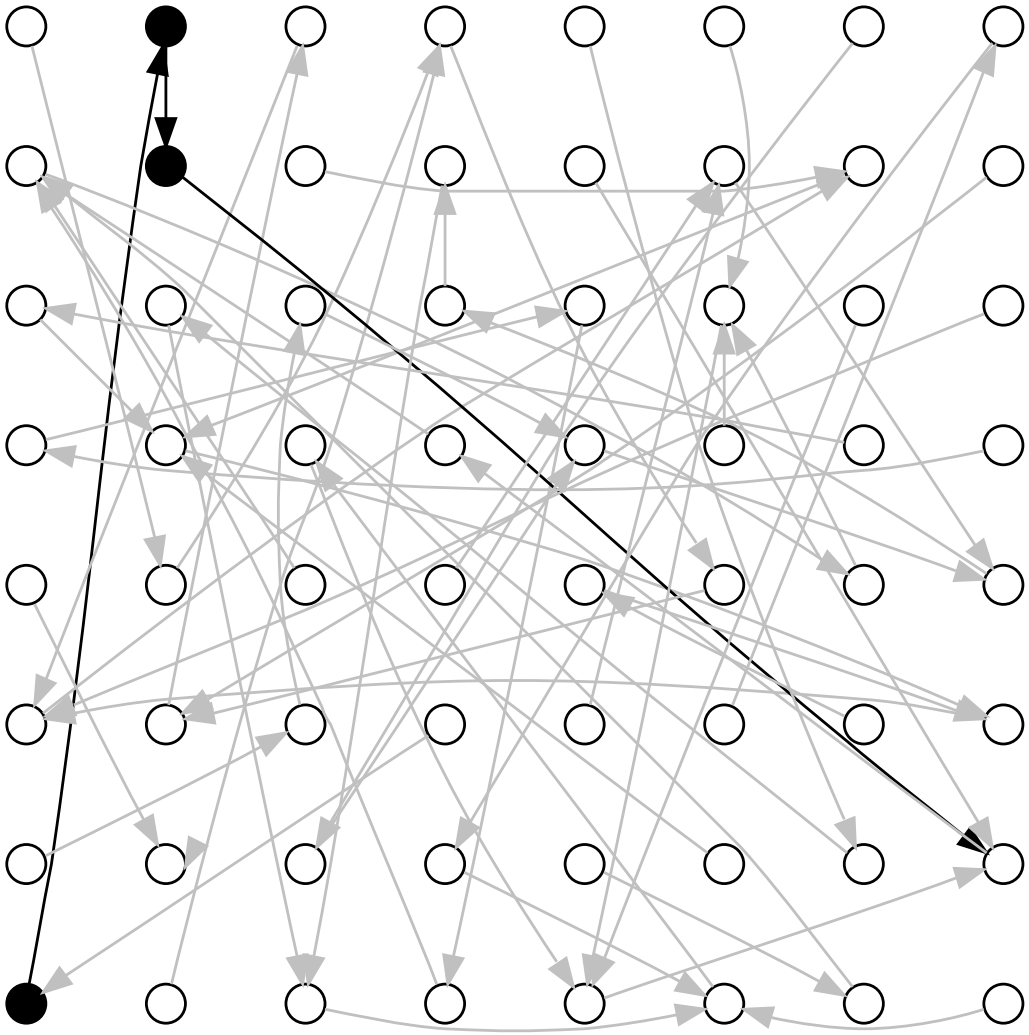
Random walks have collisions



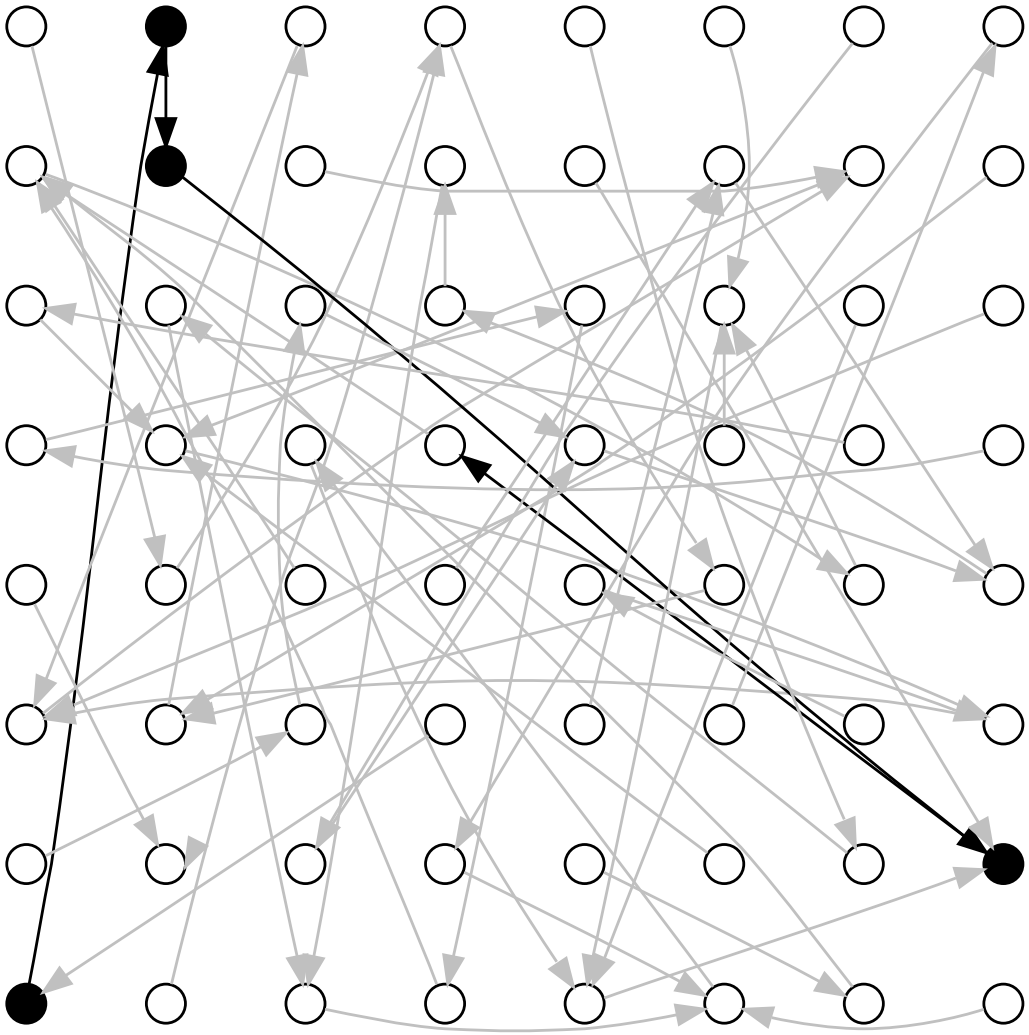
Random walks have collisions



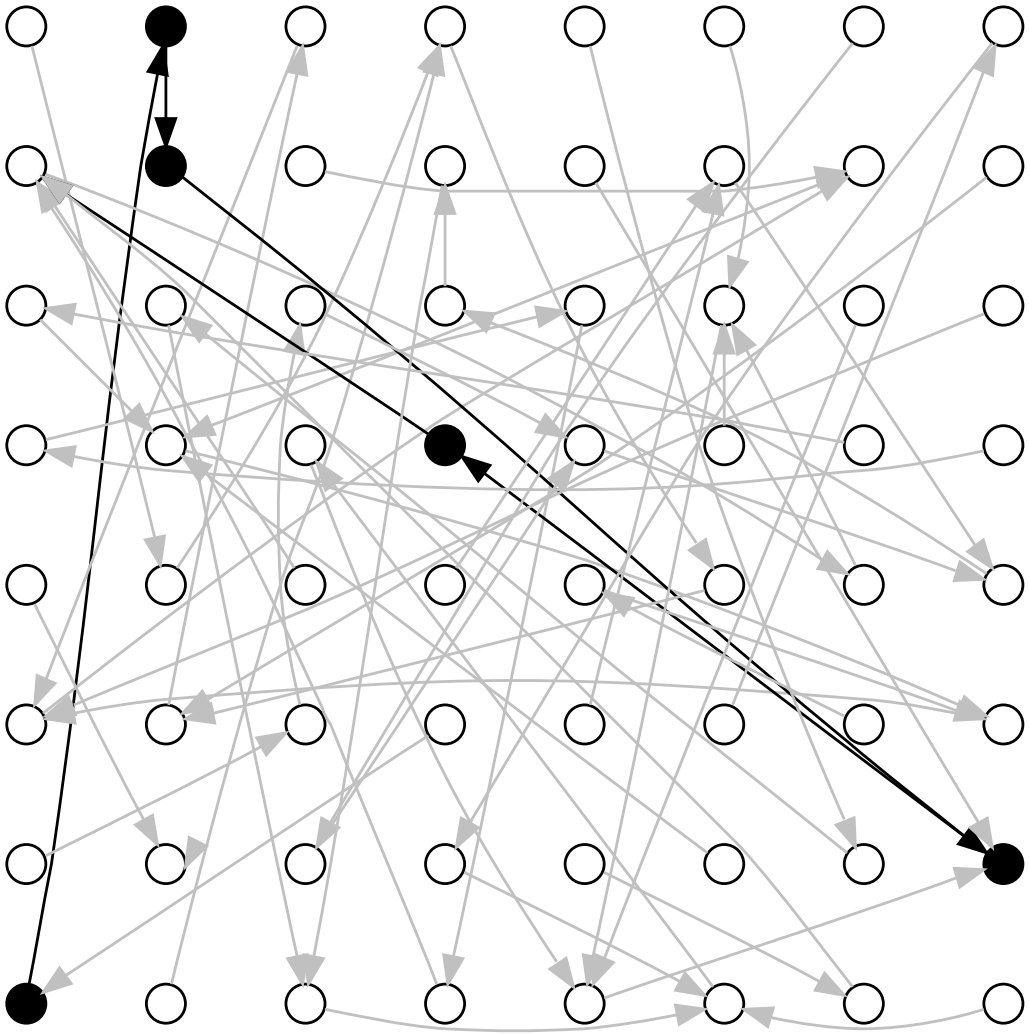
Random walks have collisions



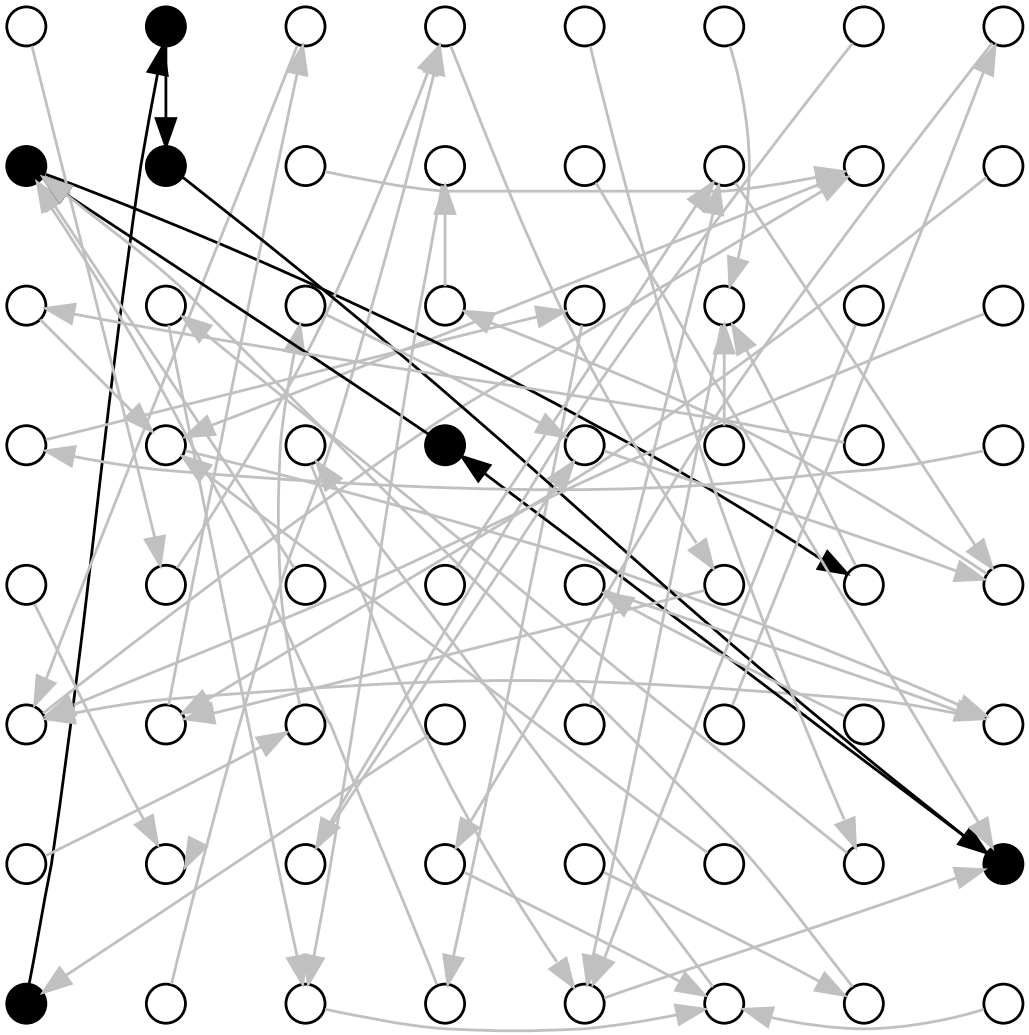
Random walks have collisions



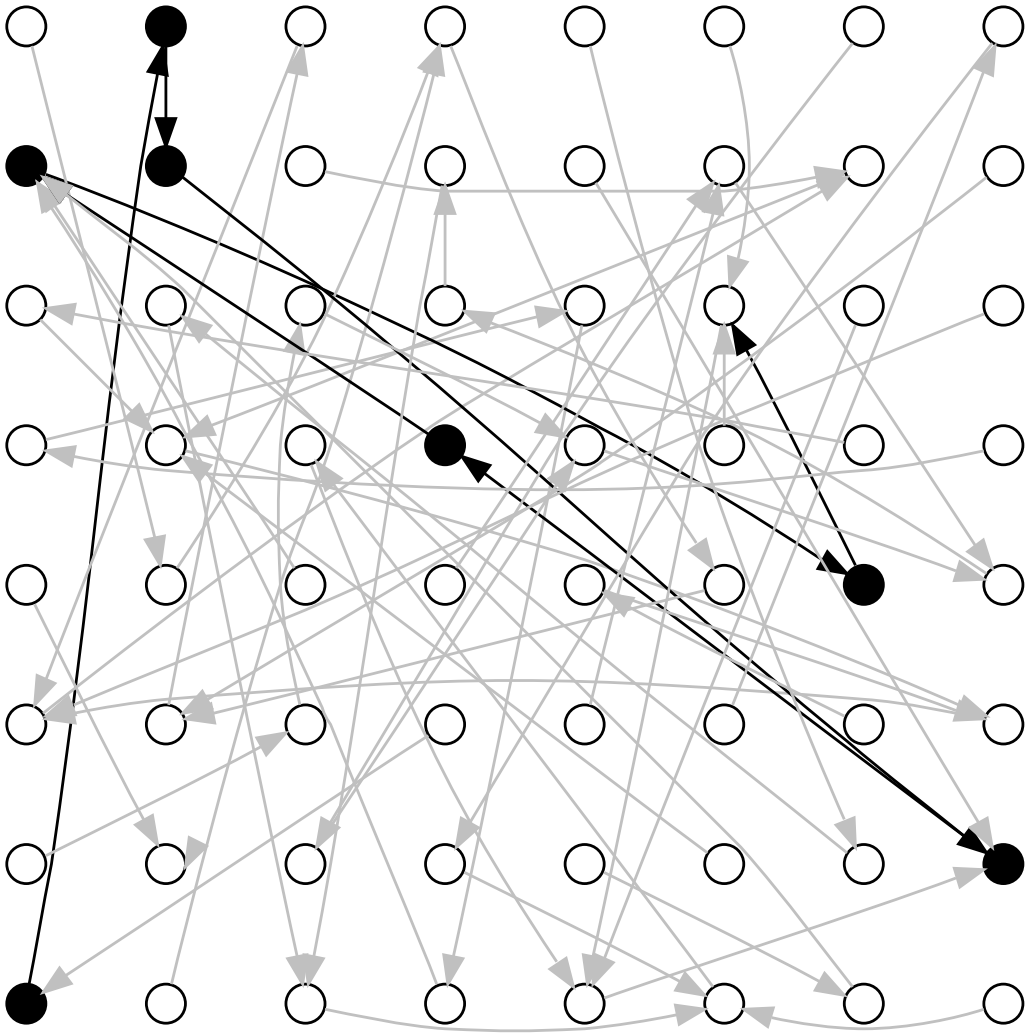
Random walks have collisions



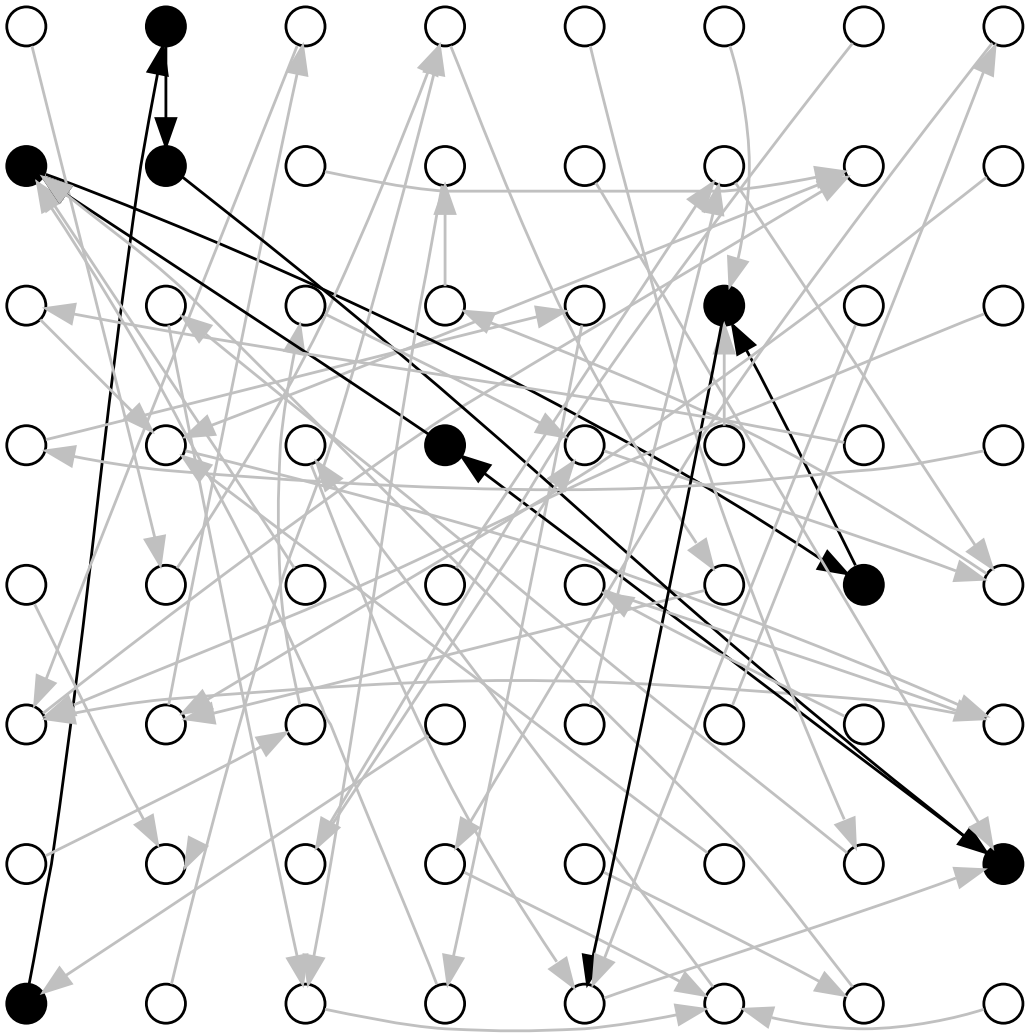
Random walks have collisions



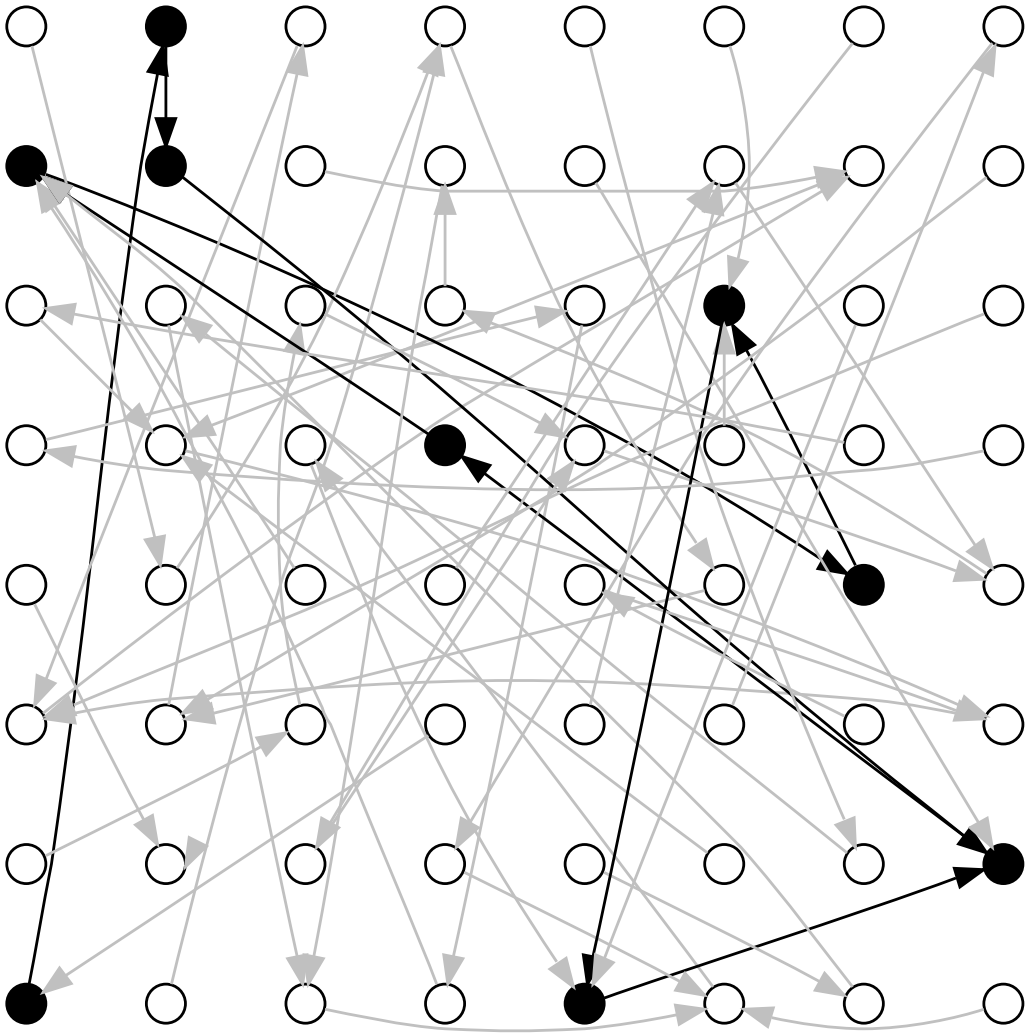
Random walks have collisions



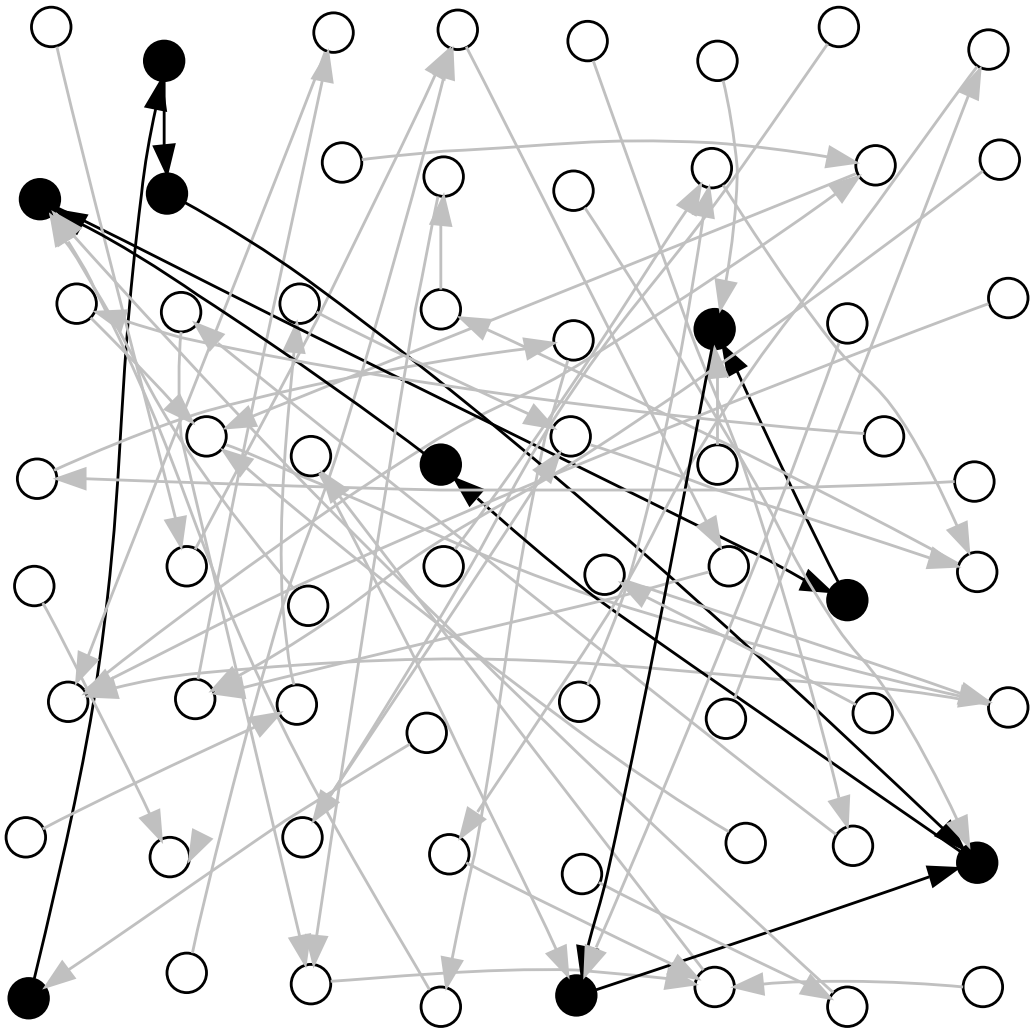
Random walks have collisions



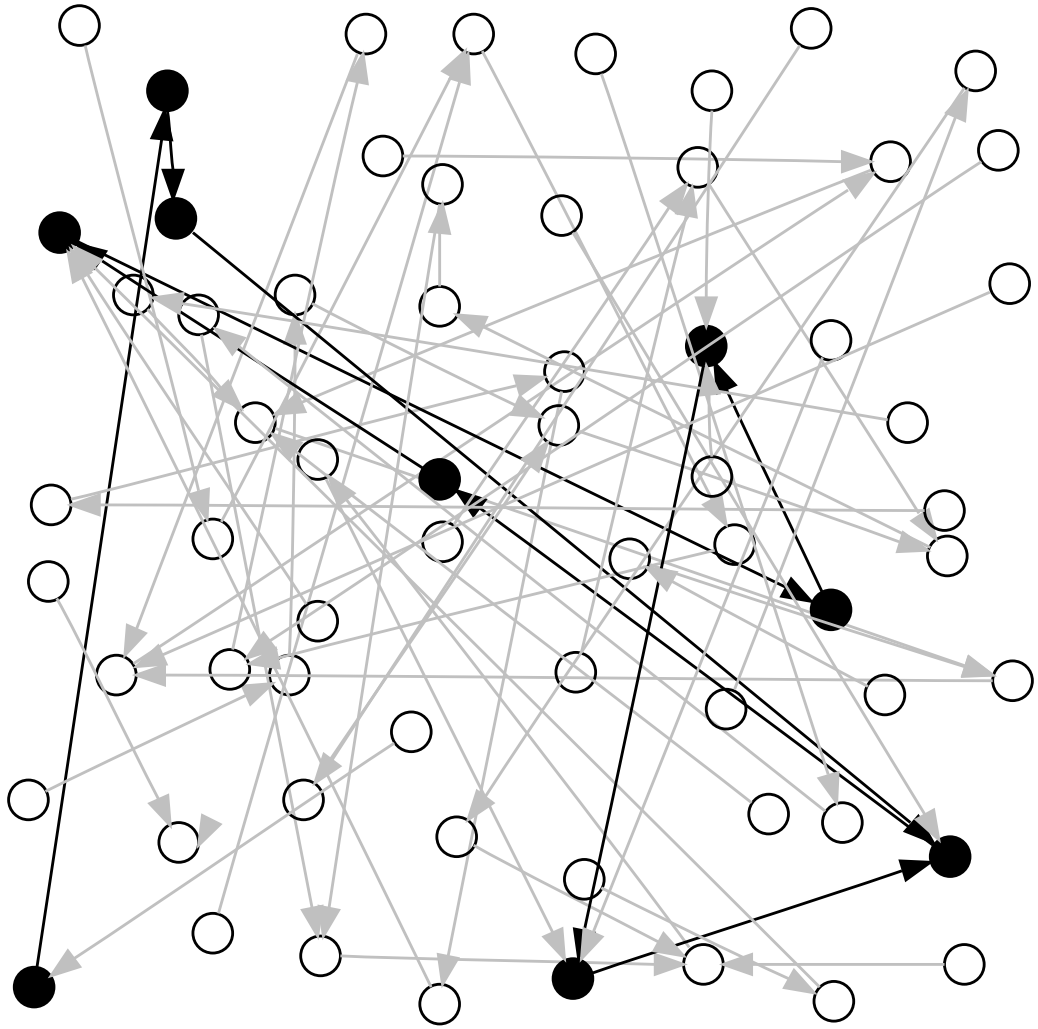
Random walks have collisions



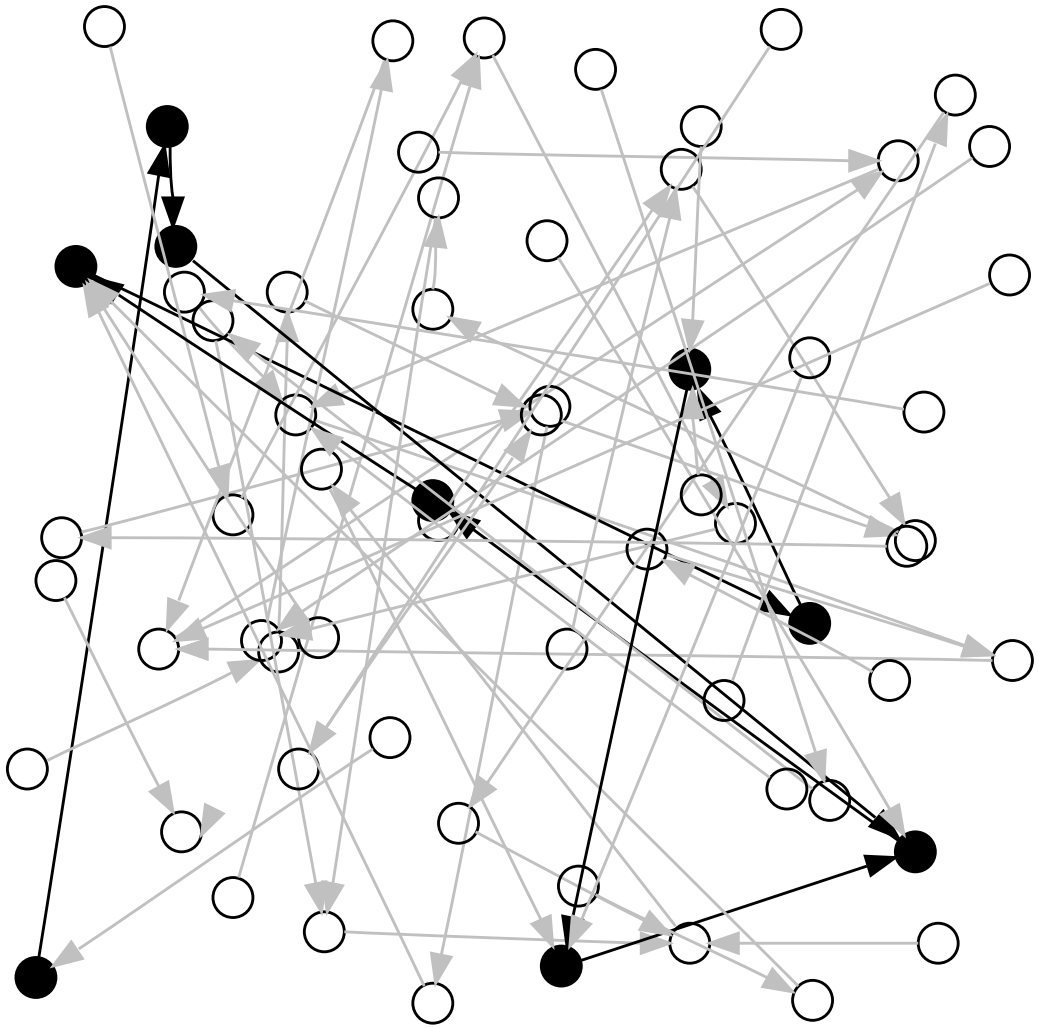
Random walks have collisions



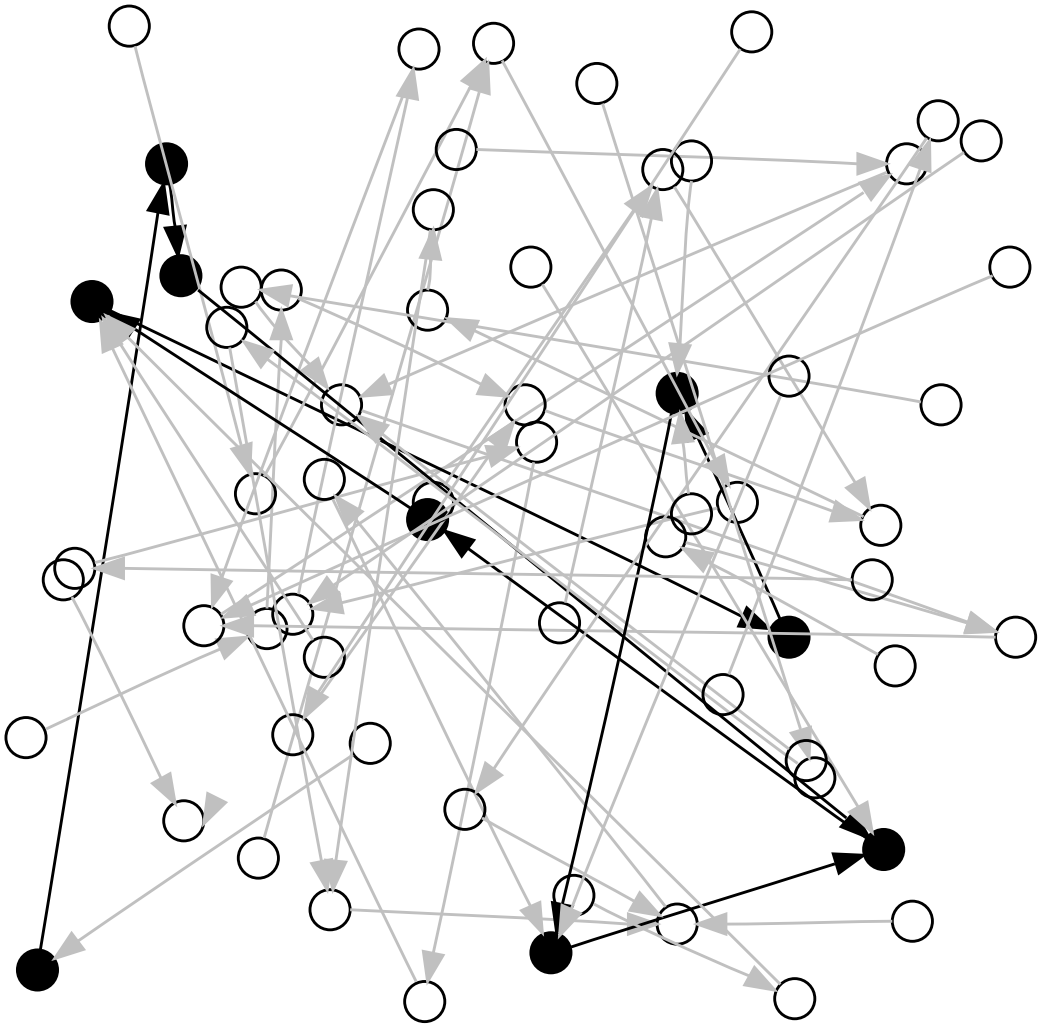
Random walks have collisions



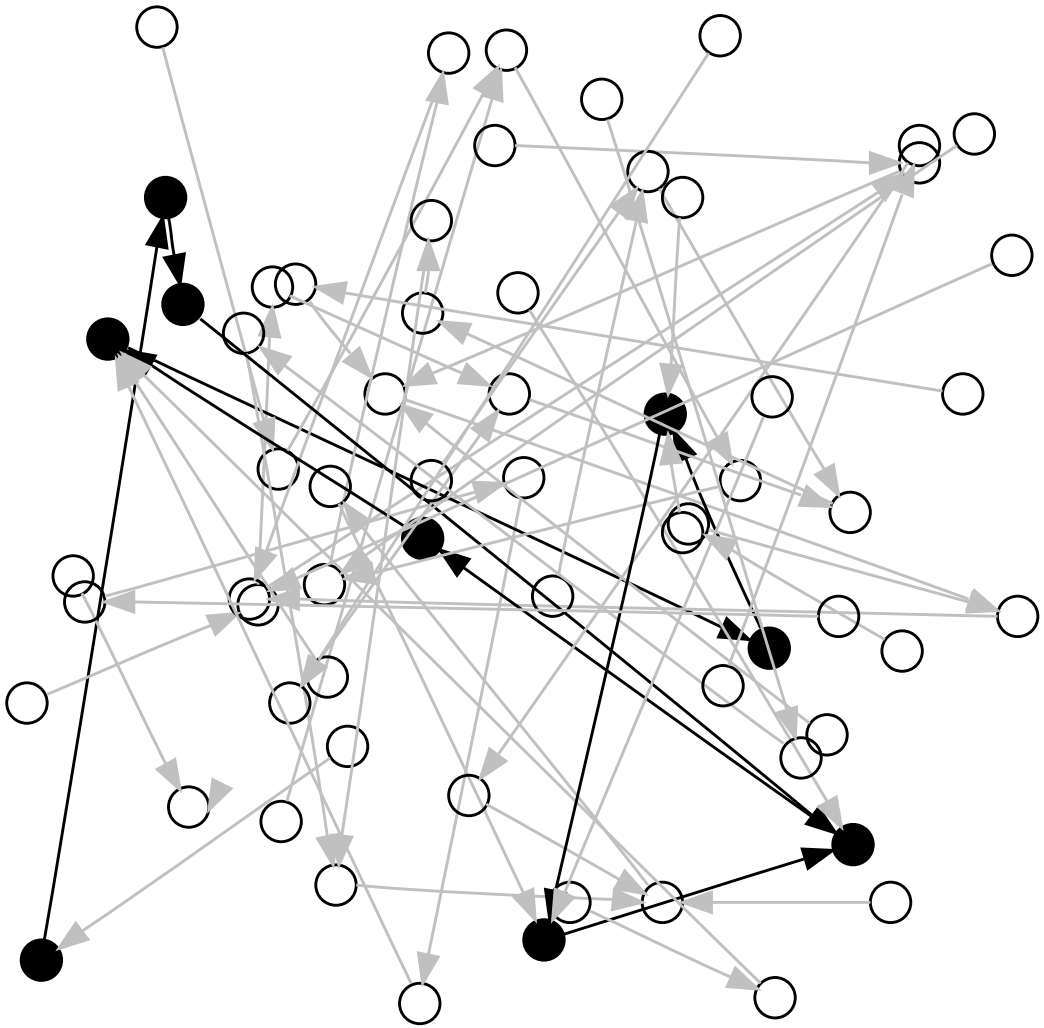
Random walks have collisions



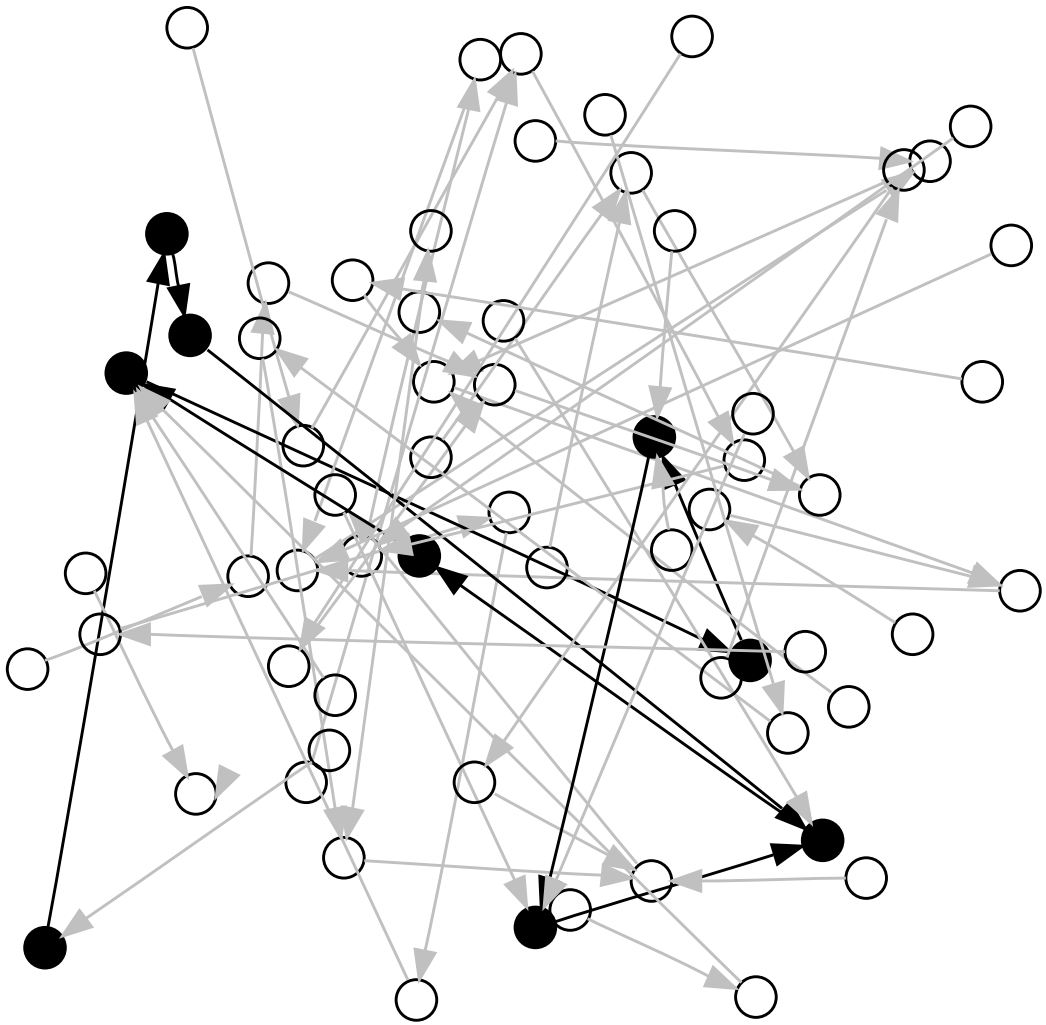
Random walks have collisions



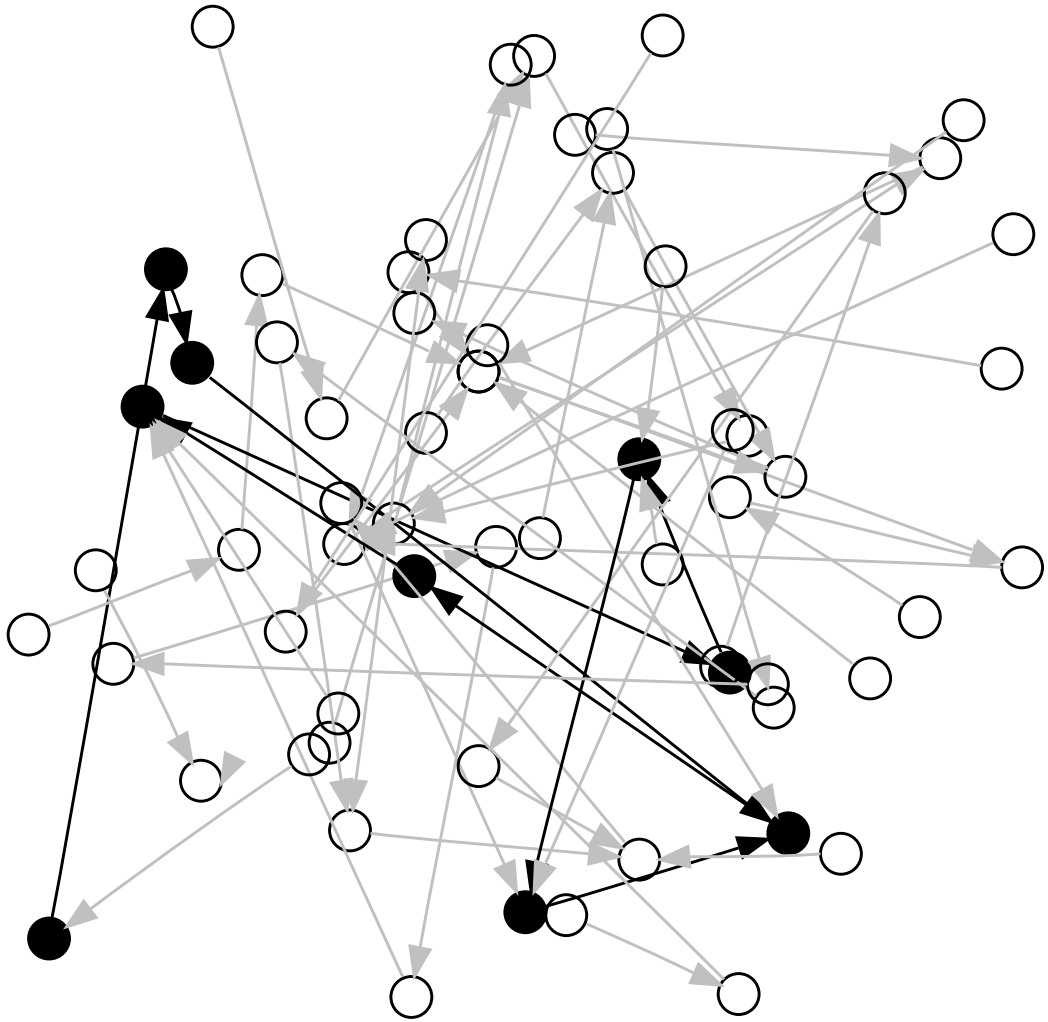
Random walks have collisions



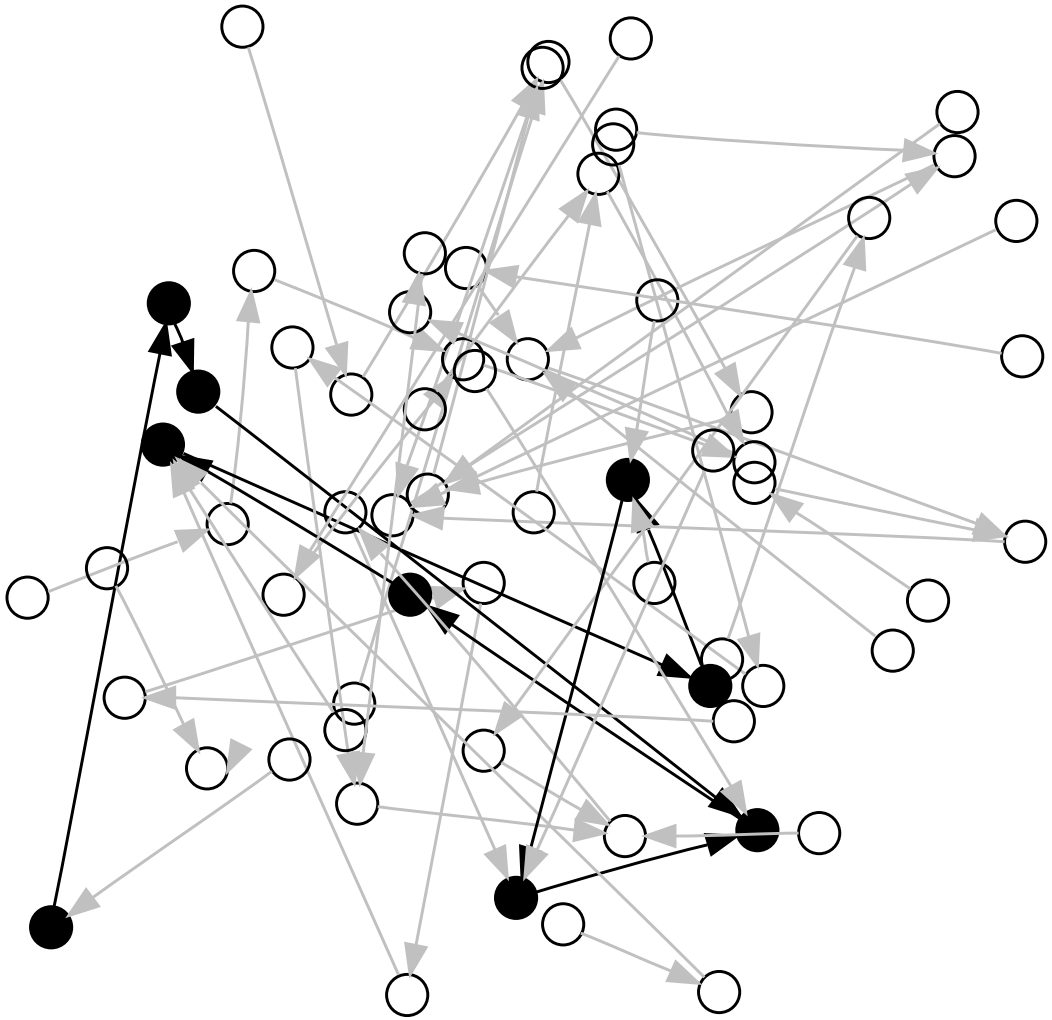
Random walks have collisions



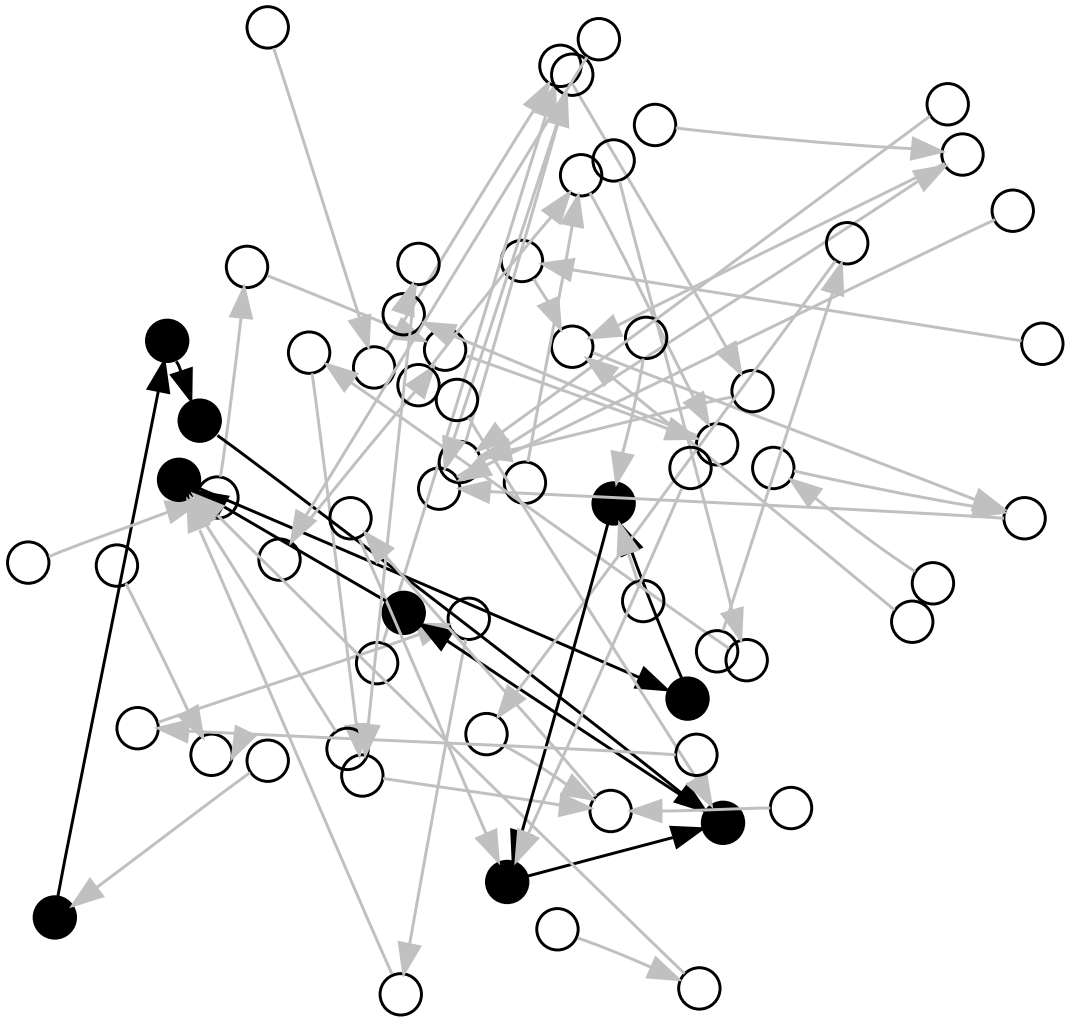
Random walks have collisions



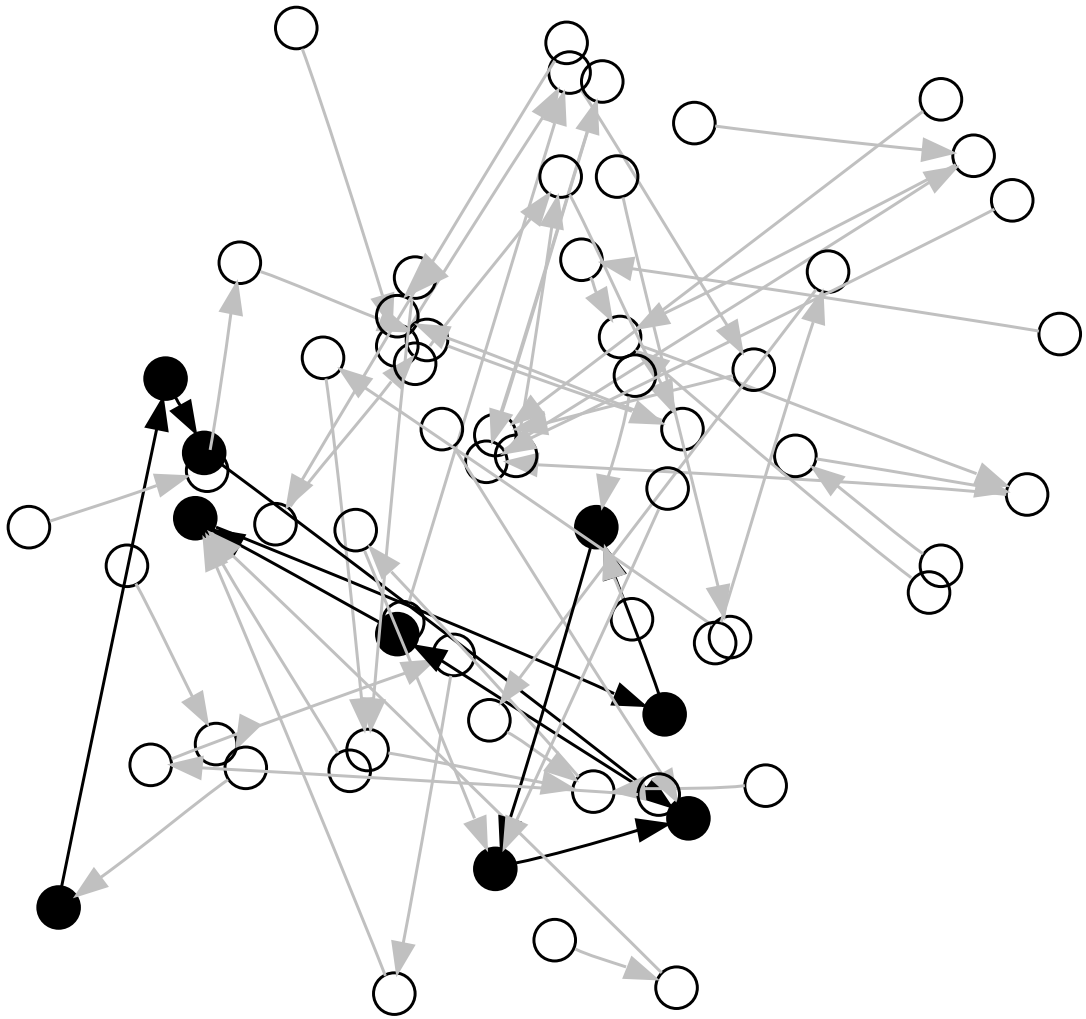
Random walks have collisions



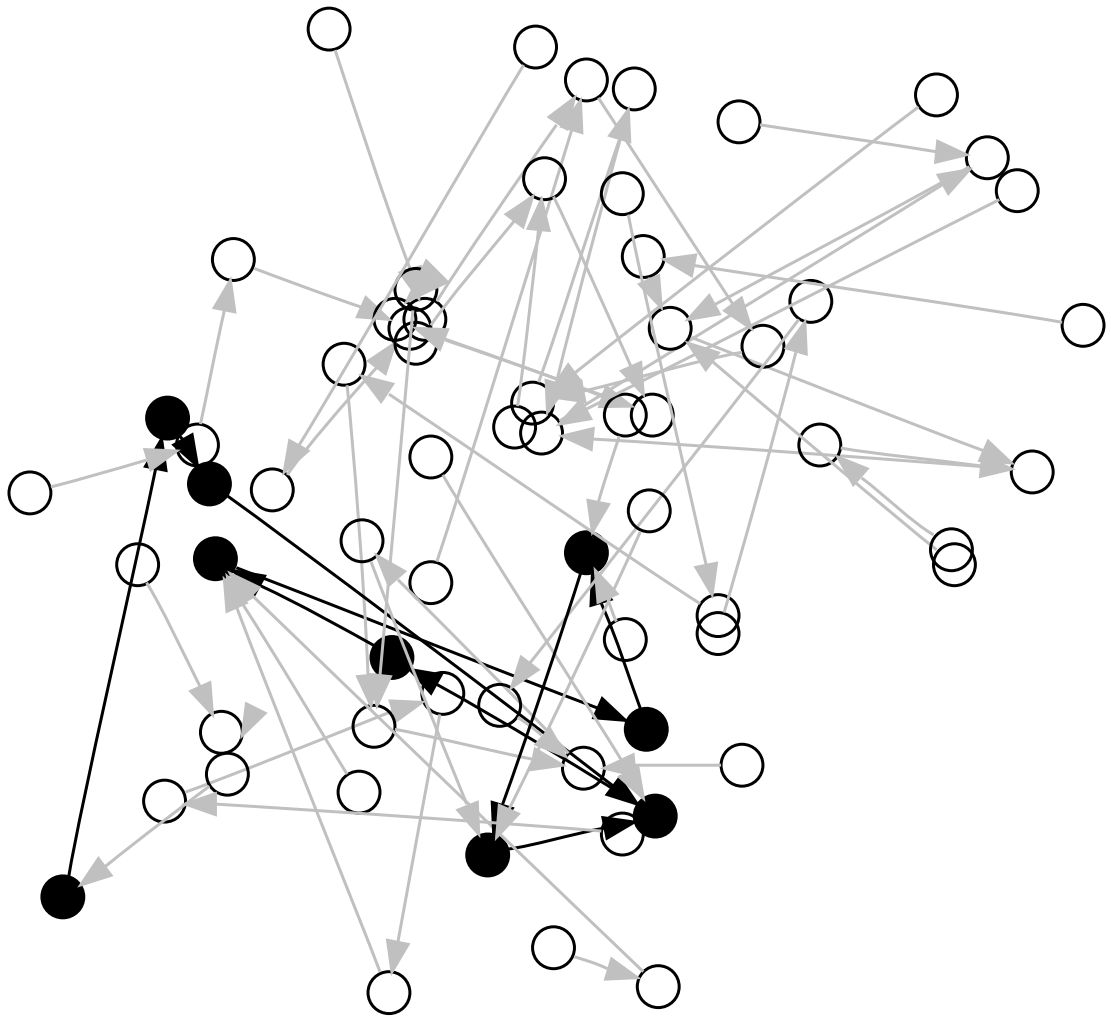
Random walks have collisions



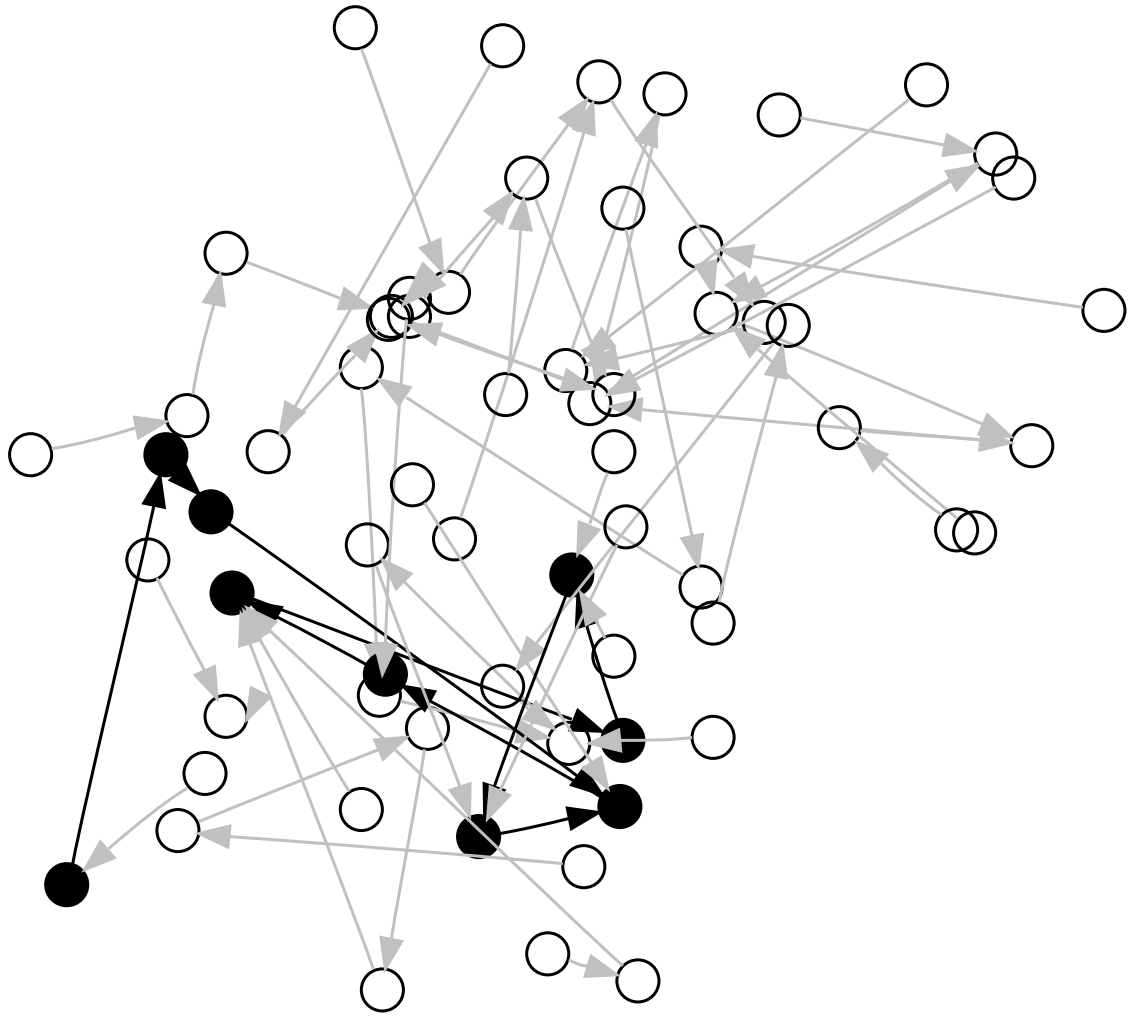
Random walks have collisions



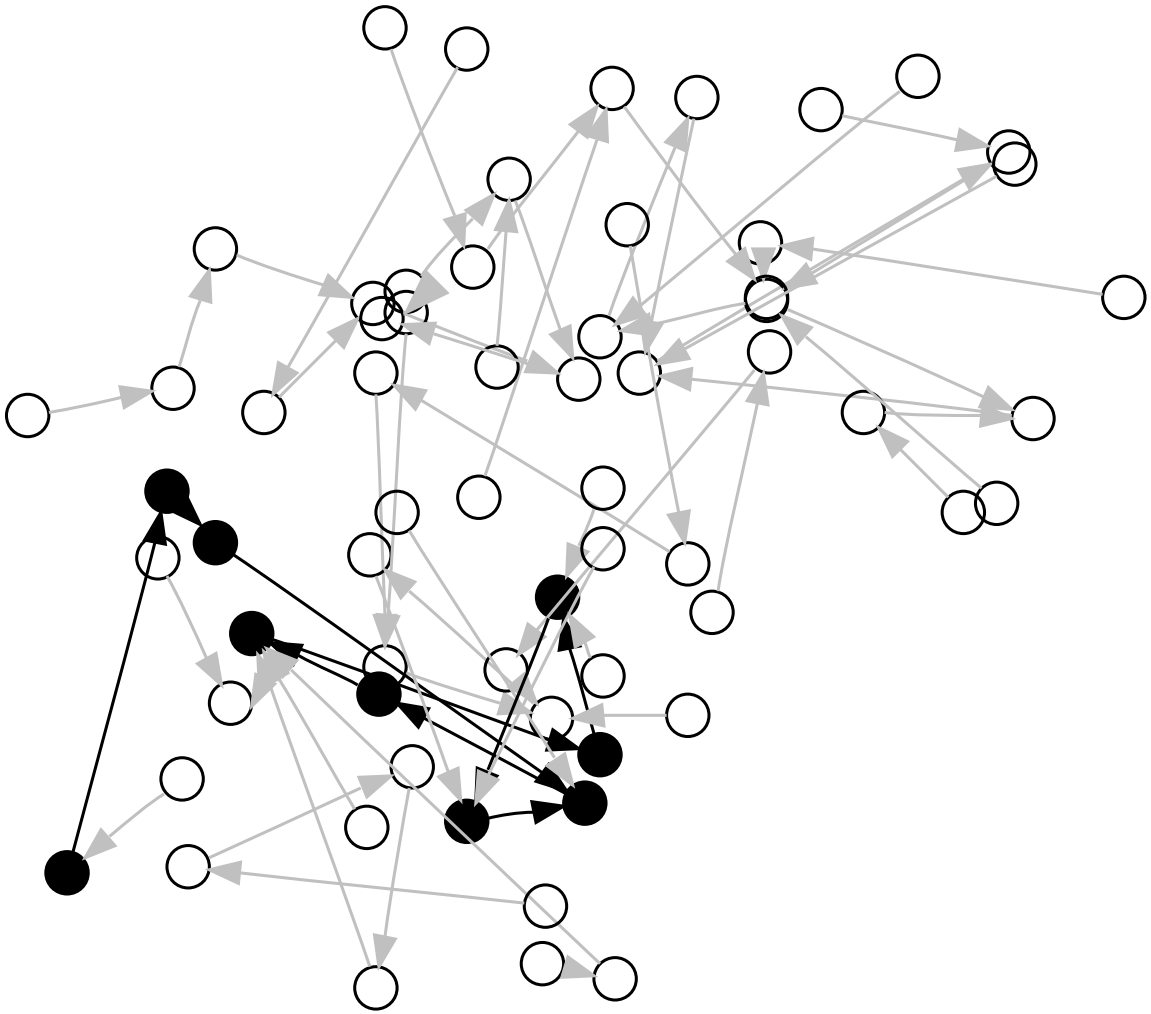
Random walks have collisions



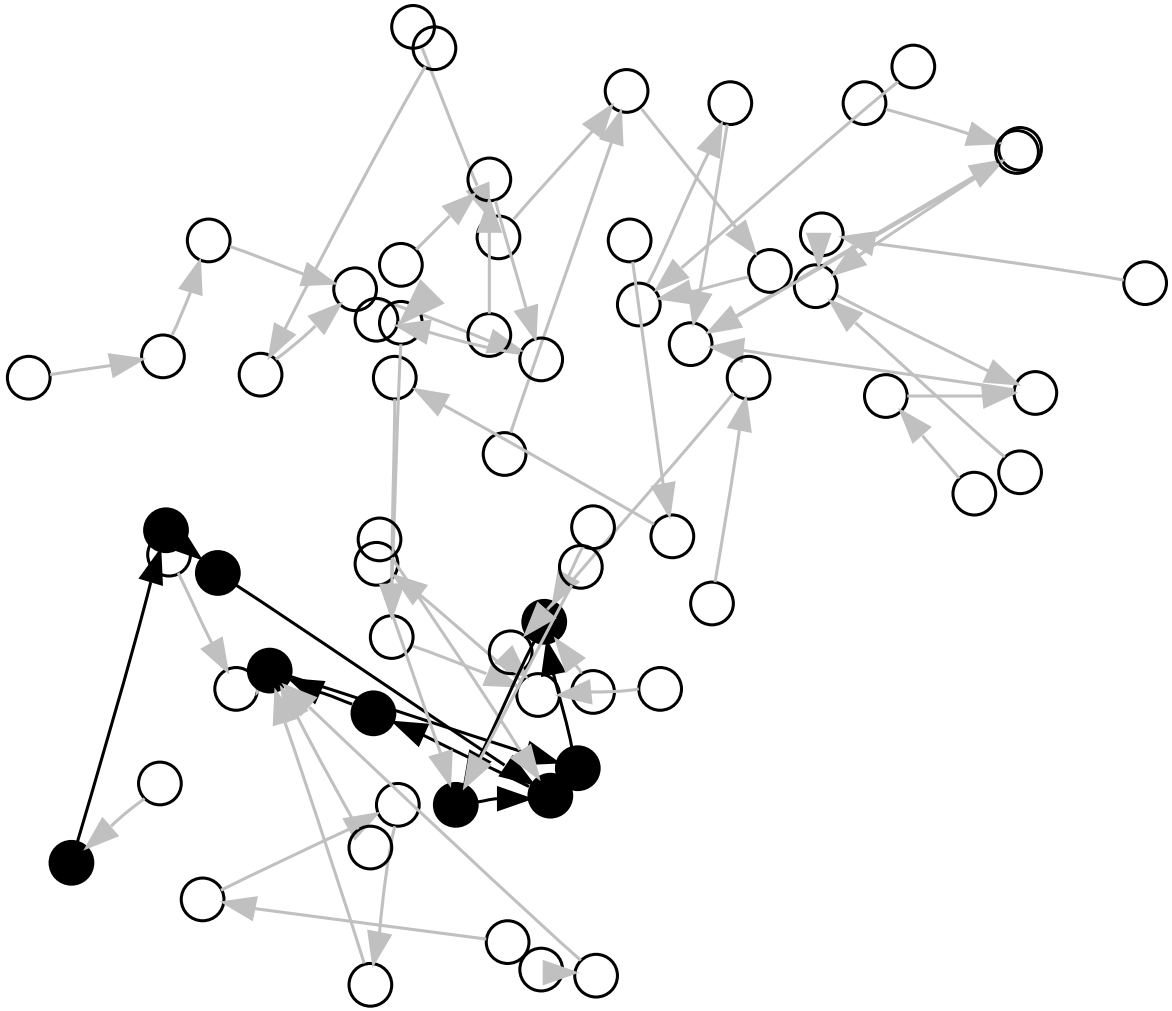
Random walks have collisions



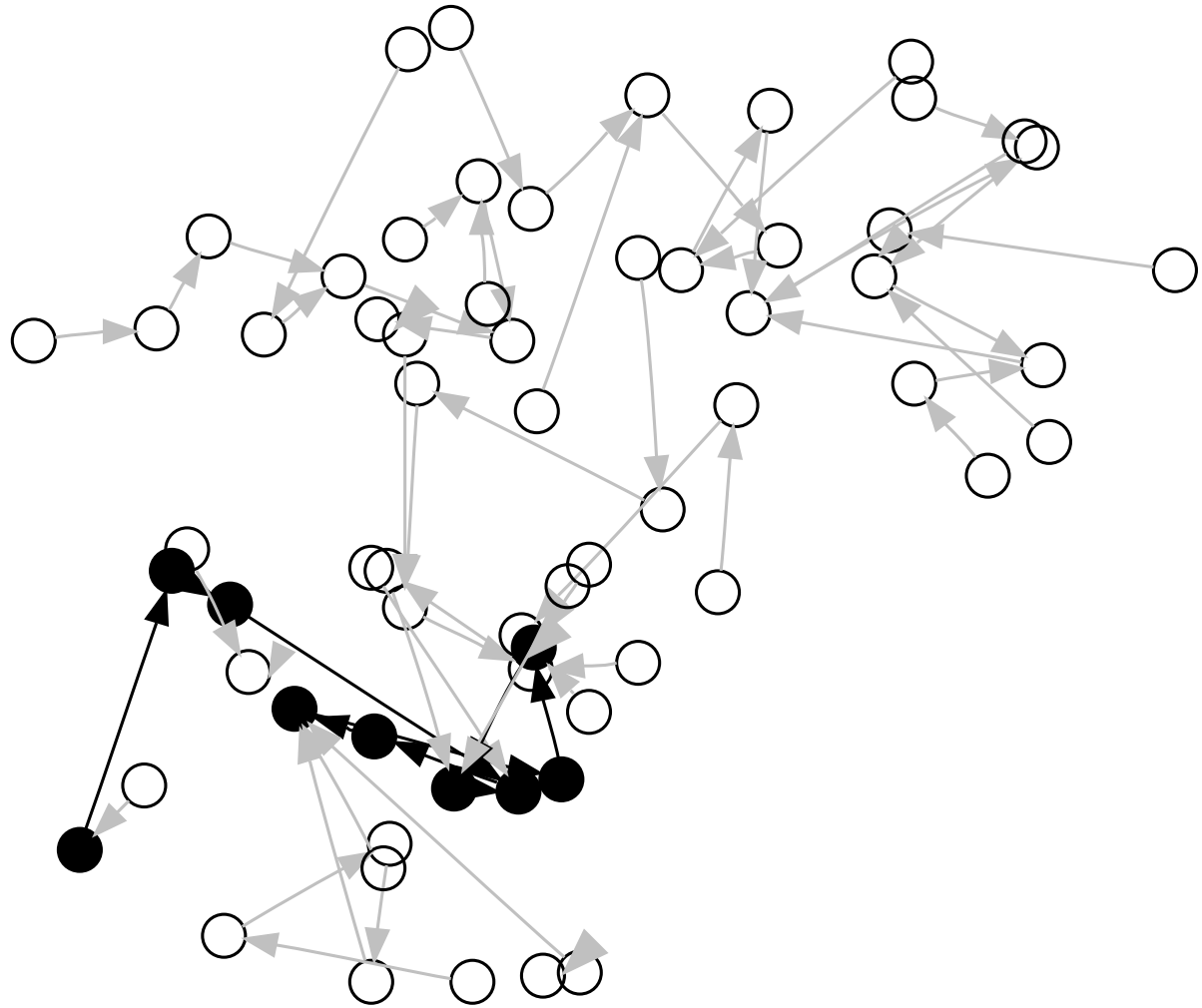
Random walks have collisions



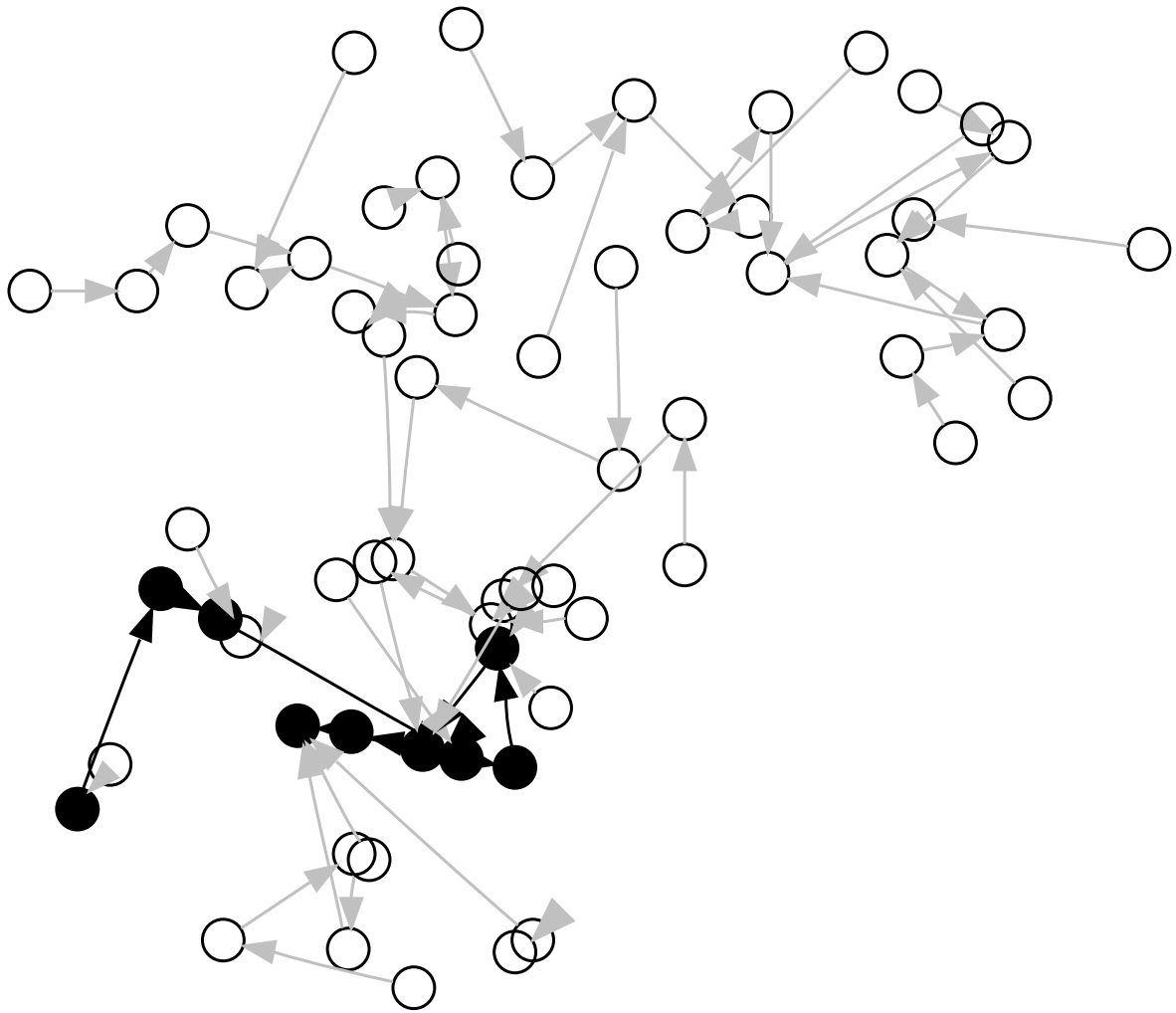
Random walks have collisions



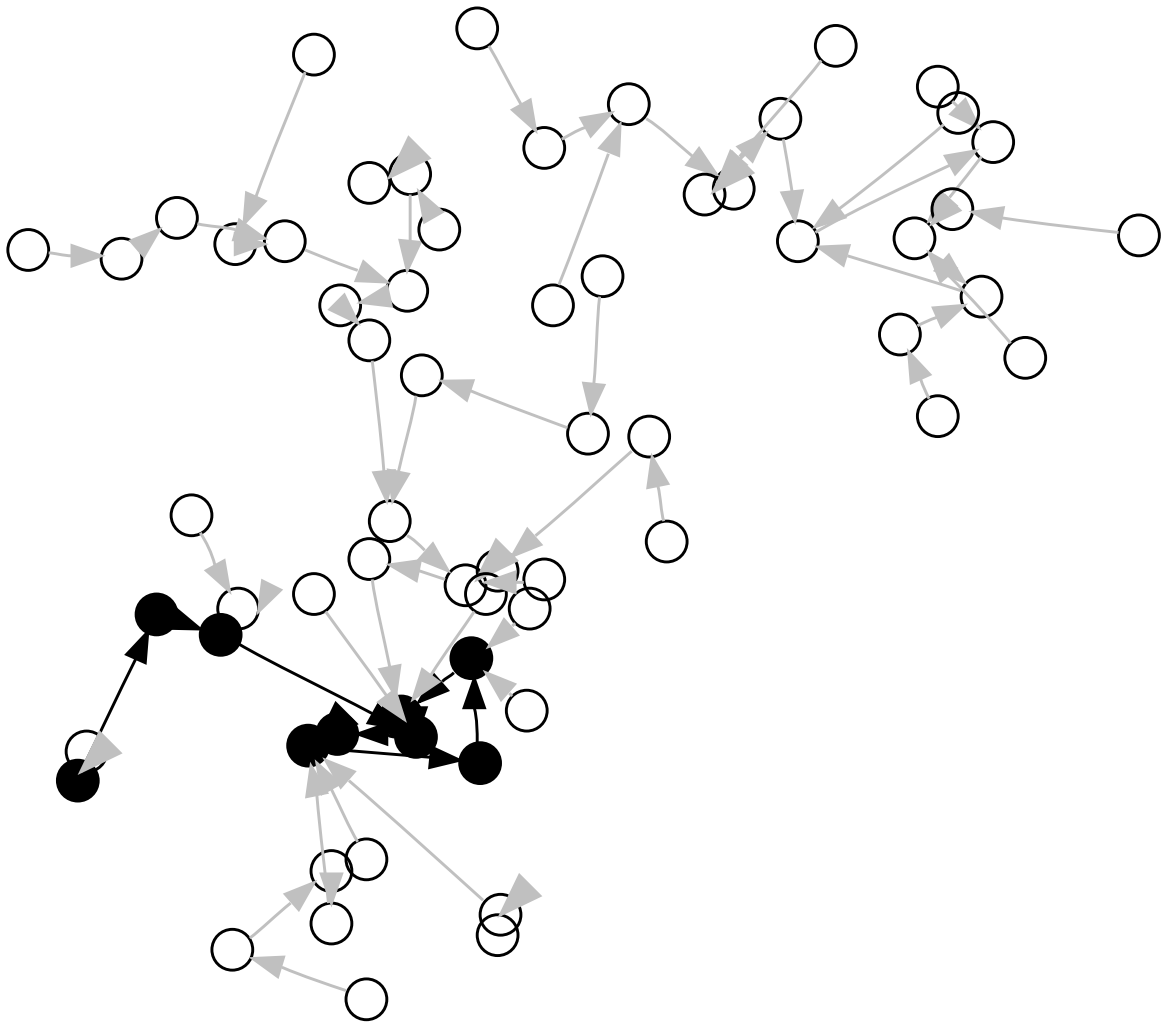
Random walks have collisions



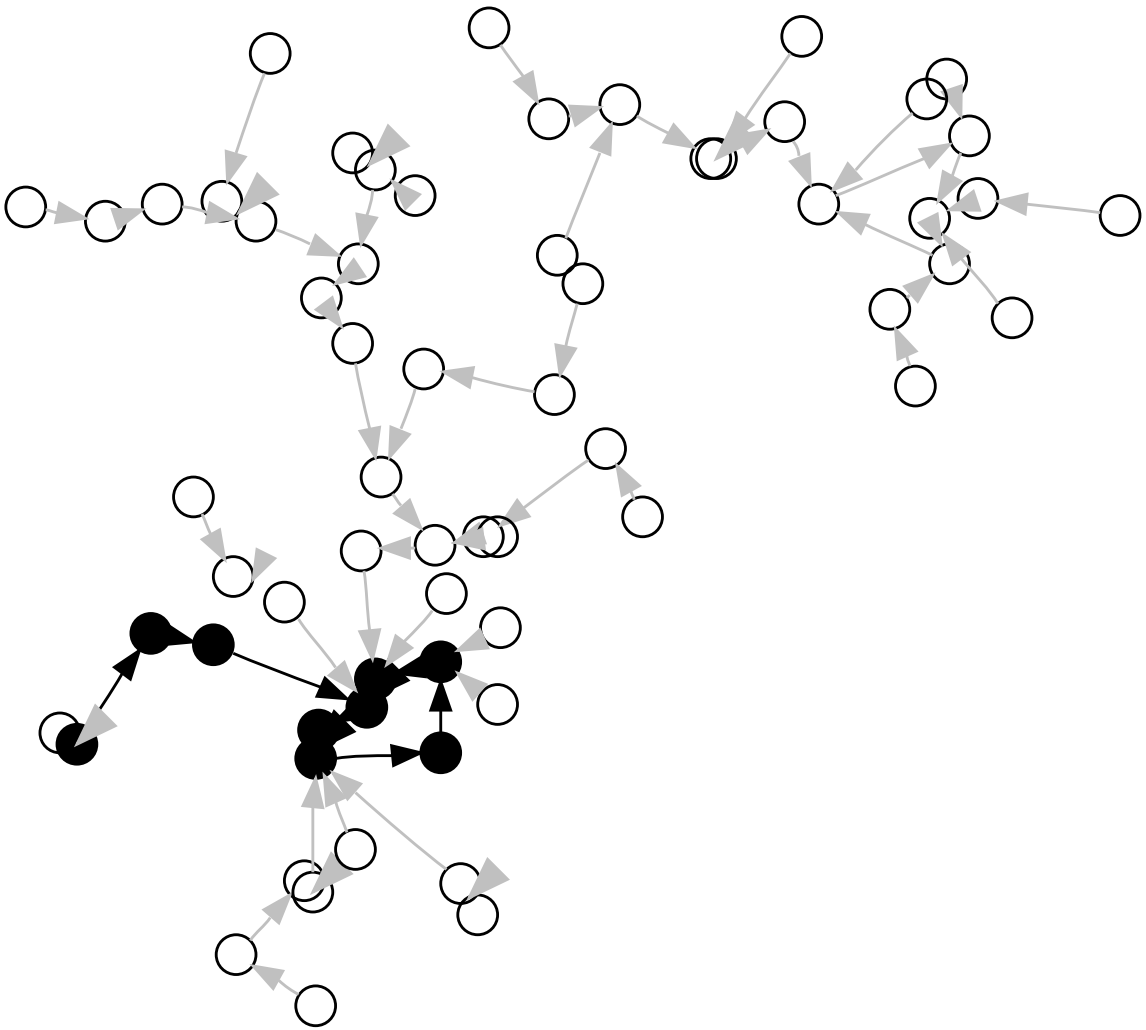
Random walks have collisions



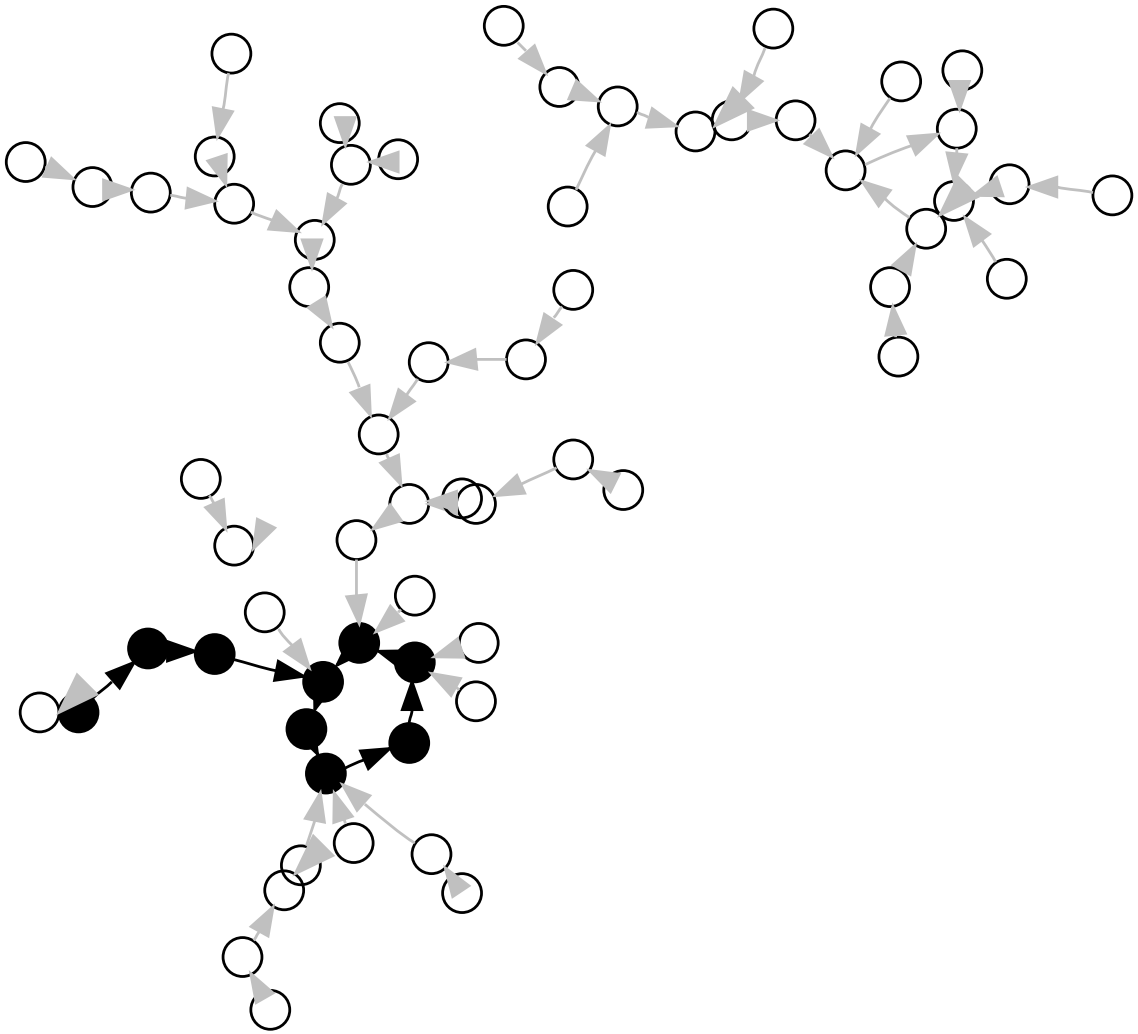
Random walks have collisions



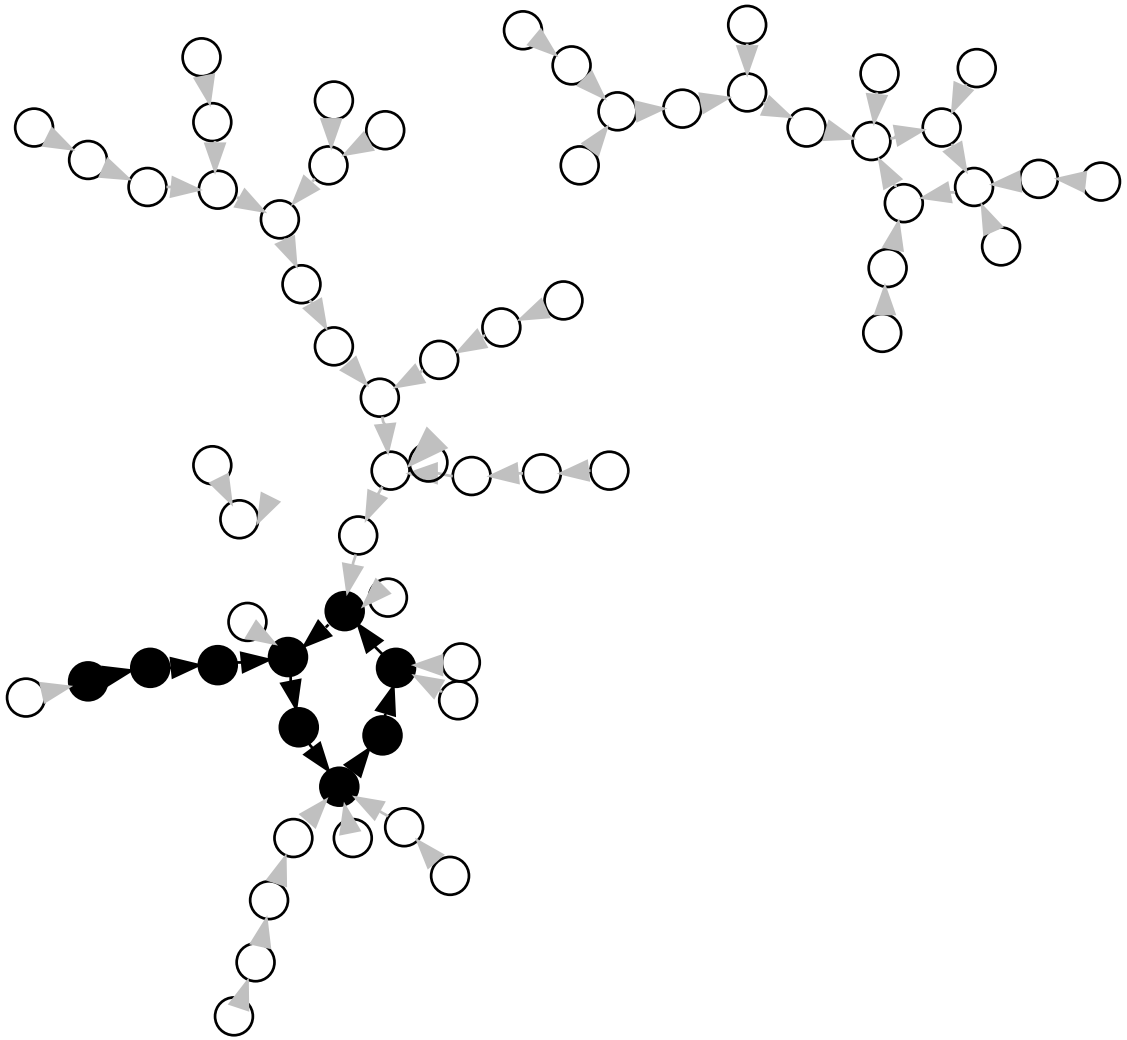
Random walks have collisions



Random walks have collisions



Random walks have collisions



For random mappings:

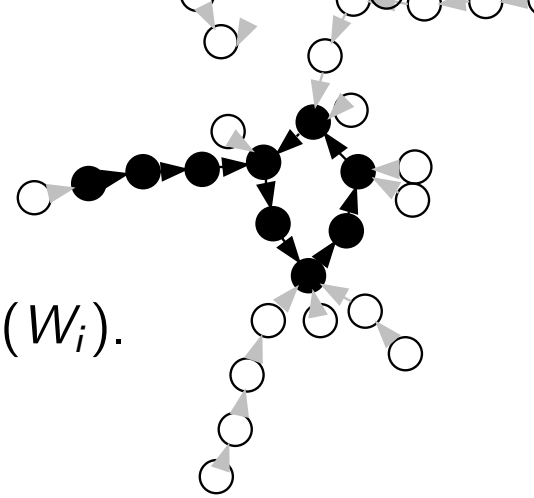
Tail length: $\sqrt{\pi n/8}$

Cycle length: $\sqrt{\pi n/8}$

See Flajolet & Odlyzko URL.

Collision after: $2\sqrt{\pi n/8} = \sqrt{\pi n/2}$ steps.

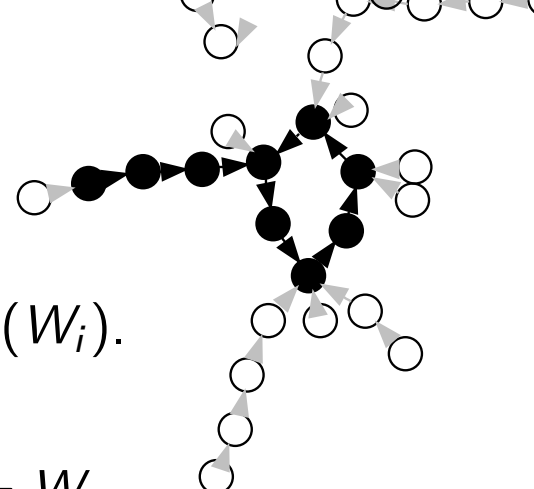
Eliminate the storage



Make a pseudo-random walk in the group $\langle P \rangle$,
where the next step depends on current point: $W_{i+1} = f(W_i)$.

After about $\sqrt{\pi n/2}$ steps we have a collision:
$W_i = W_j$.

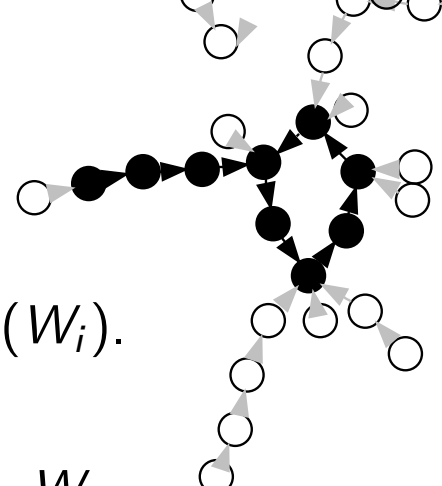
Eliminate the storage



Make a pseudo-random walk in the group $\langle P \rangle$,
where the next step depends on current point: $W_{i+1} = f(W_i)$.

After about $\sqrt{\pi n/2}$ steps we have a collision:
$W_i = W_j$.Then also $W_{i+1} = W_{j+1}, W_{i+2} = W_{j+2}, W_{i+3} = W_{j+3}, \ldots$.

Eliminate the storage

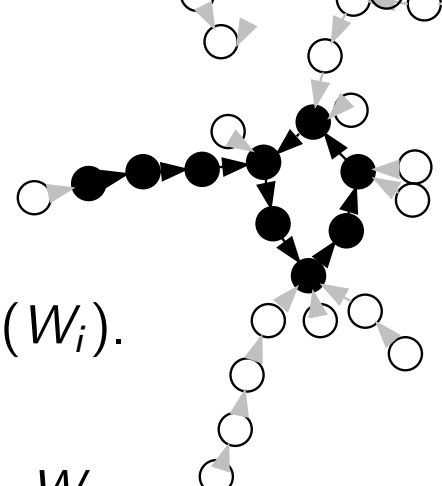


Make a pseudo-random walk in the group $\langle P \rangle$,
where the next step depends on current point: $W_{i+1} = f(W_i)$.

After about $\sqrt{\pi n/2}$ steps we have a collision:
$W_i = W_j$.Then also $W_{i+1} = W_{j+1}, W_{i+2} = W_{j+2}, W_{i+3} = W_{j+3}, \ldots$.

The walk now enters a cycle.
Cycle-finding algorithms (e.g., Floyd) quickly detects this without requiring storage.

Eliminate the storage



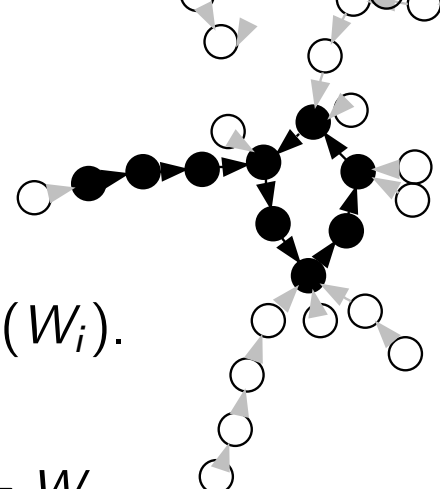
Make a pseudo-random walk in the group $\langle P \rangle$,
where the next step depends on current point: $W_{i+1} = f(W_i)$.

After about $\sqrt{\pi n/2}$ steps we have a collision:
$W_i = W_j$.Then also $W_{i+1} = W_{j+1}, W_{i+2} = W_{j+2}, W_{i+3} = W_{j+3}, \ldots$.

The walk now enters a cycle.
Cycle-finding algorithms (e.g., Floyd) quickly detects this without requiring storage.

Floyd runs two walks, a fast walk F_i and a slow walk S_i.
Only compares $F_i \stackrel{?}{=} S_i$, so does not need any older values.

Eliminate the storage



Make a pseudo-random walk in the group $\langle P \rangle$,
where the next step depends on current point: $W_{i+1} = f(W_i)$.

After about $\sqrt{\pi n/2}$ steps we have a collision:
$W_i = W_j$.Then also $W_{i+1} = W_{j+1}, W_{i+2} = W_{j+2}, W_{i+3} = W_{j+3}, \ldots$.

The walk now enters a cycle.
Cycle-finding algorithms (e.g., Floyd) quickly detects this without requiring storage.

Floyd runs two walks, a fast walk F_i and a slow walk S_i.
Only compares $F_i \stackrel{?}{=} S_i$, so does not need any older values.

Start at $S_0 = F_0 = W_0$.
Each step does $S_{i+1} = f(S_i)$ and $F_{i+1} = f(f(F_i))$.
Once both walks enter the cycle, they will collide.

Cycle length: roughly $\sqrt{\pi n/8}$.
Fast walk loops at most once after slow walk enters cycle to catch it.

Pollard's rho method

Pollard's rho method

The motivation is to remove the storage costs by turning solving the DLP into a random walk on elements of G.

This is a common strategy which we will also encounter for factorization and collision attacks for hash functions (cryptographic functions mapping to short outputs for which it should be hard to find collisions).

By the birthday paradox, after $\sqrt{\pi n/2}$ random draws from a set of n elements, one element will be drawn twice with 50% probability.

Two problems need to be solved to use the rho method successfully:

1. Design step function so that it "randomly" samples elements (so that the birthday paradox applies) while being deterministic (so we can use Floyd's cycle finding method to remove storage).
2. Design step function so that collision gives meaningful result.

Pollard's rho method

The motivation is to remove the storage costs by turning solving the DLP into a random walk on elements of G.

This is a common strategy which we will also encounter for factorization and collision attacks for hash functions (cryptographic functions mapping to short outputs for which it should be hard to find collisions).

By the birthday paradox, after $\sqrt{\pi n/2}$ random draws from a set of n elements, one element will be drawn twice with 50% probability.

Two problems need to be solved to use the rho method successfully:

1. Design step function so that it "randomly" samples elements (so that the birthday paradox applies) while being deterministic (so we can use Floyd's cycle finding method to remove storage).
2. Design step function so that collision gives meaningful result.

For DLP want $W_i = a_i P + b_i Q$ so that $W_i = W_j$ means that

$$a_i P + b_i Q = a_j P + b_j Q \Leftrightarrow (b_i - b_j)Q = (a_j - a_i)P \Leftrightarrow a \equiv (a_j - a_i)/(b_i - b_j) \bmod n.$$

This means that collisions are meaningful if $\gcd(n, b_i - b_j) = 1$.

Pollard rho for DLP: Attempt 1

For DLP want $W_i = a_i P + b_i Q$ so that $W_i = W_j$ means that

$$a_i P + b_i Q = a_j P + b_j Q \Leftrightarrow (b_i - b_j)Q = (a_j - a_i)P \Leftrightarrow a \equiv (a_j - a_i)/(b_i - b_j) \bmod n.$$

Simplest approach: let $g, h : G \to [0, n-1]$ and

$$f(W) = g(W)P + h(W)Q.$$

Then $a_i = g(W_{i-1}), b_i = h(W_{i-1})$.

If g and h are sufficiently different and random,
f provides a random walk on G.

Pollard rho for DLP: Attempt 1

For DLP want $W_i = a_i P + b_i Q$ so that $W_i = W_j$ means that

$$a_i P + b_i Q = a_j P + b_j Q \Leftrightarrow (b_i - b_j)Q = (a_j - a_i)P \Leftrightarrow a \equiv (a_j - a_i)/(b_i - b_j) \bmod n.$$

Simplest approach: let $g, h : G \to [0, n-1]$ and

$$f(W) = g(W)P + h(W)Q.$$

Then $a_i = g(W_{i-1}), b_i = h(W_{i-1})$.

If g and h are sufficiently different and random,
f provides a random walk on G.

Pick a random starting point

$$W_0 = a_0 P + b_0 Q.$$

Each step costs one double-scalar multiplication.
Total cost: $\sqrt{\pi n/2}$ double-scalar multiplications.

Pollard rho for DLP: Attempt 1

For DLP want $W_i = a_i P + b_i Q$ so that $W_i = W_j$ means that

$$a_i P + b_i Q = a_j P + b_j Q \Leftrightarrow (b_i - b_j)Q = (a_j - a_i)P \Leftrightarrow a \equiv (a_j - a_i)/(b_i - b_j) \bmod n.$$

Simplest approach: let $g, h : G \to [0, n-1]$ and

$$f(W) = g(W)P + h(W)Q.$$

Then $a_i = g(W_{i-1}), b_i = h(W_{i-1})$.

If g and h are sufficiently different and random,
f provides a random walk on G.

Pick a random starting point

$$W_0 = a_0 P + b_0 Q.$$

Each step costs one double-scalar multiplication.
Total cost: $\sqrt{\pi n/2}$ double-scalar multiplications.

A double-scalar multiplication is cheaper than two separate scalar multiplications. (Share doublings, add P, Q, or $P + Q$.)

Pollard rho for DLP: Attempt 2 – additive walks

[Note: We have a solution, we're just haggling about the price.]

Idea: we do not need n^2 directions to look random.
Having some fixed set of step directions suffices.

Additive walks make each step cheaper, try to keep features.

Pollard rho for DLP: Attempt 2 – additive walks

[Note: We have a solution, we're just haggling about the price.]

Idea: we do not need n^2 directions to look random.
Having some fixed set of step directions suffices.

Additive walks make each step cheaper, try to keep features.

Pick small number, e.g. $k = 32$, of random $(c_j, d_j) \in [0, n-1]^2$.
Compute and store k steps $R_j = c_j P + d_j Q, 0 \leq j < k$.

Fixed (small) number of double-scalar multiplications,
fixed (small) amount of storage.

Define the step function, costing 1 ADD per step, as

$$f(W) = W + R_{s(W)}, \text{ with } s : G \to [0, k-1].$$

s assigns one of the precomputed k steps to each group element

Pollard rho for DLP: Attempt 2 – additive walks

[Note: We have a solution, we're just haggling about the price.]

Idea: we do not need n^2 directions to look random.
Having some fixed set of step directions suffices.

Additive walks make each step cheaper, try to keep features.

Pick small number, e.g. $k = 32$, of random $(c_j, d_j) \in [0, n-1]^2$.
Compute and store k steps $R_j = c_j P + d_j Q, 0 \leq j < k$.

Fixed (small) number of double-scalar multiplications,
fixed (small) amount of storage.

Define the step function, costing 1 ADD per step, as

$$f(W) = W + R_{s(W)}, \text{ with } s : G \to [0, k-1].$$

s assigns one of the precomputed k steps to each group element

Typical choice: take $s(W) \equiv x(W) \bmod k$, where $x(W)$ is the x-coordinate of W and $\mathbf{F}_p$ is represented as integers in $[0, p-1]$.
This is why we use affine not projective coordinates. Want unique representation.

Rho for DLP with additive walks

$$f(W) = W + R_{s(W)}, \text{ with } s : G \to [0, k-1].$$

Problem 1: we don't know anymore how to write $W_i = a_i P + b_i Q$.

Rho for DLP with additive walks

$$f(W) = W + R_{s(W)}, \text{ with } s : G \to [0, k-1].$$

Problem 1: we don't know anymore how to write $W_i = a_i P + b_i Q$.

Solution: start from *known* starting point and keep track of the scalars.

Each step updates

$$W_i \leftarrow W_i + R_{s(W_i)}, \quad a_i \leftarrow a_i + c_{s(W_i)}, \quad b_i \leftarrow b_i + d_{s(W_i)}.$$

starting from $W_0 = a_0 P + b_0 Q$, with known random a_0, b_0.

Rho for DLP with additive walks

$$f(W) = W + R_{s(W)}, \text{ with } s : G \to [0, k-1].$$

Problem 1: we don't know anymore how to write $W_i = a_i P + b_i Q$.

Solution: start from *known* starting point and keep track of the scalars.

Each step updates

$$W_i \leftarrow W_i + R_{s(W_i)}, \quad a_i \leftarrow a_i + c_{s(W_i)}, \quad b_i \leftarrow b_i + d_{s(W_i)}.$$

starting from $W_0 = a_0 P + b_0 Q$, with known random a_0, b_0.

Problem 2: How big does k need to be for f to look random?

Rho for DLP with additive walks

$$f(W) = W + R_{s(W)}, \text{ with } s : G \to [0, k-1].$$

Problem 1: we don't know anymore how to write $W_i = a_i P + b_i Q$.

Solution: start from *known* starting point and keep track of the scalars.

Each step updates

$$W_i \leftarrow W_i + R_{s(W_i)}, \quad a_i \leftarrow a_i + c_{s(W_i)}, \quad b_i \leftarrow b_i + d_{s(W_i)}.$$

starting from $W_0 = a_0 P + b_0 Q$, with known random a_0, b_0.

Problem 2: How big does k need to be for f to look random?

Additive walks induce anti-collisions (see next page), this delays collisions.
If there are k steps then the runtime increases by a factor of

$$1/\sqrt{1 - 1/k} \approx 1 + 1/(2k).$$

Anti-collisions in additive walks

Fix point T. Let W and W' be two independent uniform random points.

Anti-collisions in additive walks

Fix point T. Let W and W' be two independent uniform random points.

$W \neq W'$ both map to T if simultaneously for $i \neq j$:

$$T = W + R_i = W' + R_j, \quad s(W) = i, \quad s(W') = j.$$

These conditions have probability $1/n^2$, $1/k$, and $1/k$ respectively.

Anti-collisions in additive walks

Fix point T. Let W and W' be two independent uniform random points.

$W \neq W'$ both map to T if simultaneously for $i \neq j$:

$$T = W + R_i = W' + R_j, \quad s(W) = i, \quad s(W') = j.$$

These conditions have probability $1/n^2$, $1/k$, and $1/k$ respectively.

Summing over all (i, j) gives the overall probability

$$\sum_{i \neq j} 1/(kn)^2 = k(k-1)/(kn)^2 = (1 - 1/k)/n^2.$$

Anti-collisions in additive walks

Fix point T. Let W and W' be two independent uniform random points.

$W \neq W'$ both map to T if simultaneously for $i \neq j$:

$$T = W + R_i = W' + R_j, \quad s(W) = i, \quad s(W') = j.$$

These conditions have probability $1/n^2$, $1/k$, and $1/k$ respectively.

Summing over all (i, j) gives the overall probability

$$\sum_{i \neq j} 1/(kn)^2 = k(k-1)/(kn)^2 = (1 - 1/k)/n^2.$$

Collisions can occur at any T, so add over the n choices of T.
The probability of immediate collision from W and W' is

$$(1 - 1/k)/n$$

instead of $1/n$. Thus $\sqrt{\pi n/2}$ changes to $\sqrt{\pi n/(2(1 - 1/k))}$.

Anti-collisions in additive walks

Fix point T. Let W and W' be two independent uniform random points.

$W \neq W'$ both map to T if simultaneously for $i \neq j$:

$$T = W + R_i = W' + R_j, \quad s(W) = i, \quad s(W') = j.$$

These conditions have probability $1/n^2$, $1/k$, and $1/k$ respectively.

Summing over all (i, j) gives the overall probability

$$\sum_{i \neq j} 1/(kn)^2 = k(k-1)/(kn)^2 = (1 - 1/k)/n^2.$$

Collisions can occur at any T, so add over the n choices of T.
The probability of immediate collision from W and W' is

$$(1 - 1/k)/n$$

instead of $1/n$. Thus $\sqrt{\pi n/2}$ changes to $\sqrt{\pi n/(2(1 - 1/k))}$.

Additive walks need more steps by a factor of

$$1/\sqrt{1 - 1/k} \approx 1 + 1/(2k).$$

Schoolbook method for Pollard rho

The schoolbook method uses a step function with only 3 types of steps.

This would be very bad for randomness with the $1 + 1/(2k)$ formula!

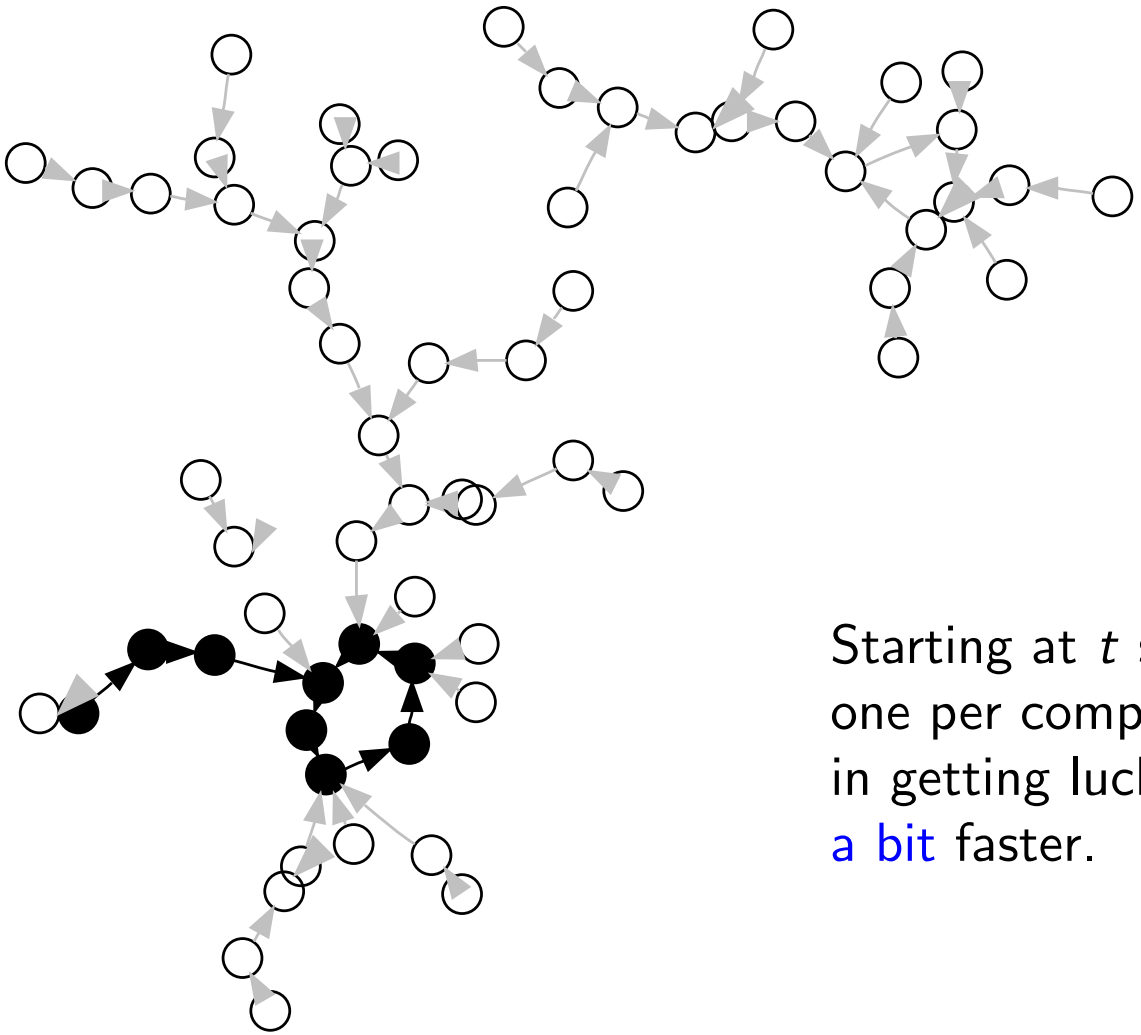
The method escapes that by using doubling as one of the steps.

$$W \leftarrow \begin{cases} W+P \\ W+Q \\ 2W \end{cases}, a \leftarrow \begin{cases} a+1 \\ a \\ 2a \end{cases}, b \leftarrow \begin{cases} b \\ b+1 \\ 2b \end{cases}, \text{ for } s(W) = \begin{cases} 0 \\ 1 \\ 2 \end{cases}.$$

Typically $s(W)$ takes an integer representation of $x(W)$ and outputs the remainder of division by 3.

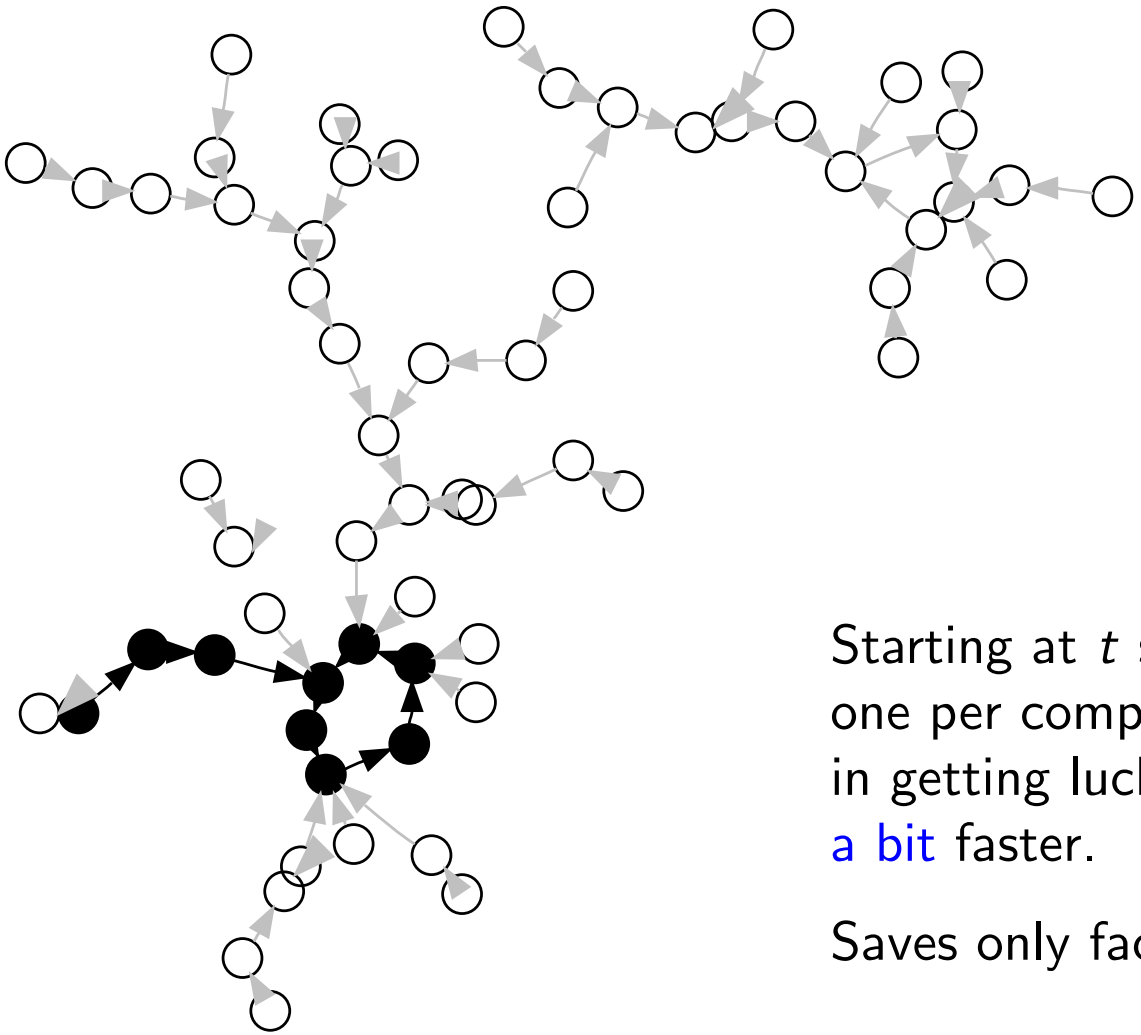
Paralel collision search

How to use more than one computer efficiently?



Starting at t starting points,
one per computer, helps
in getting lucky
a bit faster.

How to use more than one computer efficiently?



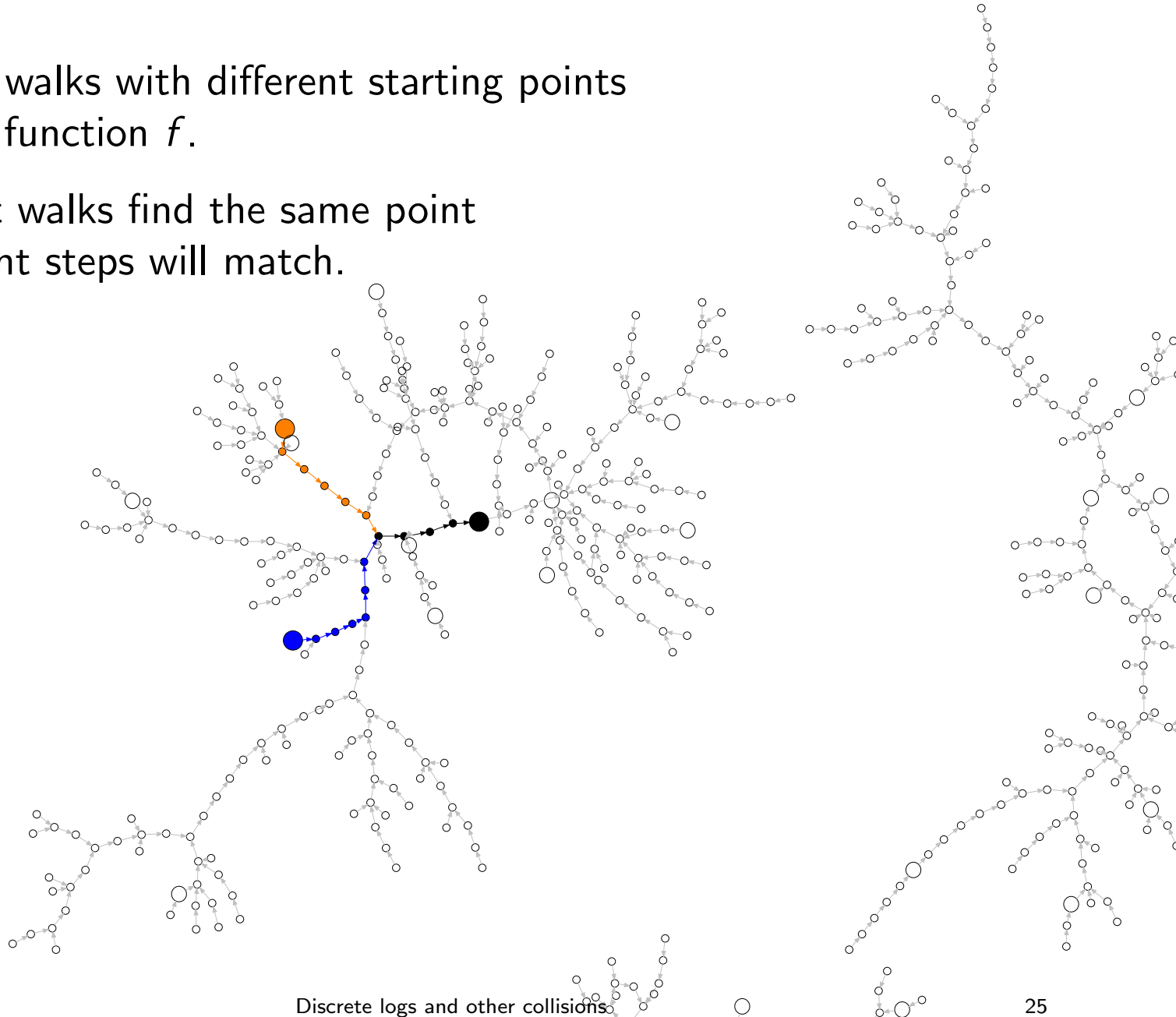
Starting at t starting points, one per computer, helps in getting lucky a bit faster.

Saves only factor of $\sqrt{t}$.

Van Oorschot and Wiener parallel collision search

Perform many walks with different starting points but same step function f.

If two different walks find the same point their subsequent steps will match.

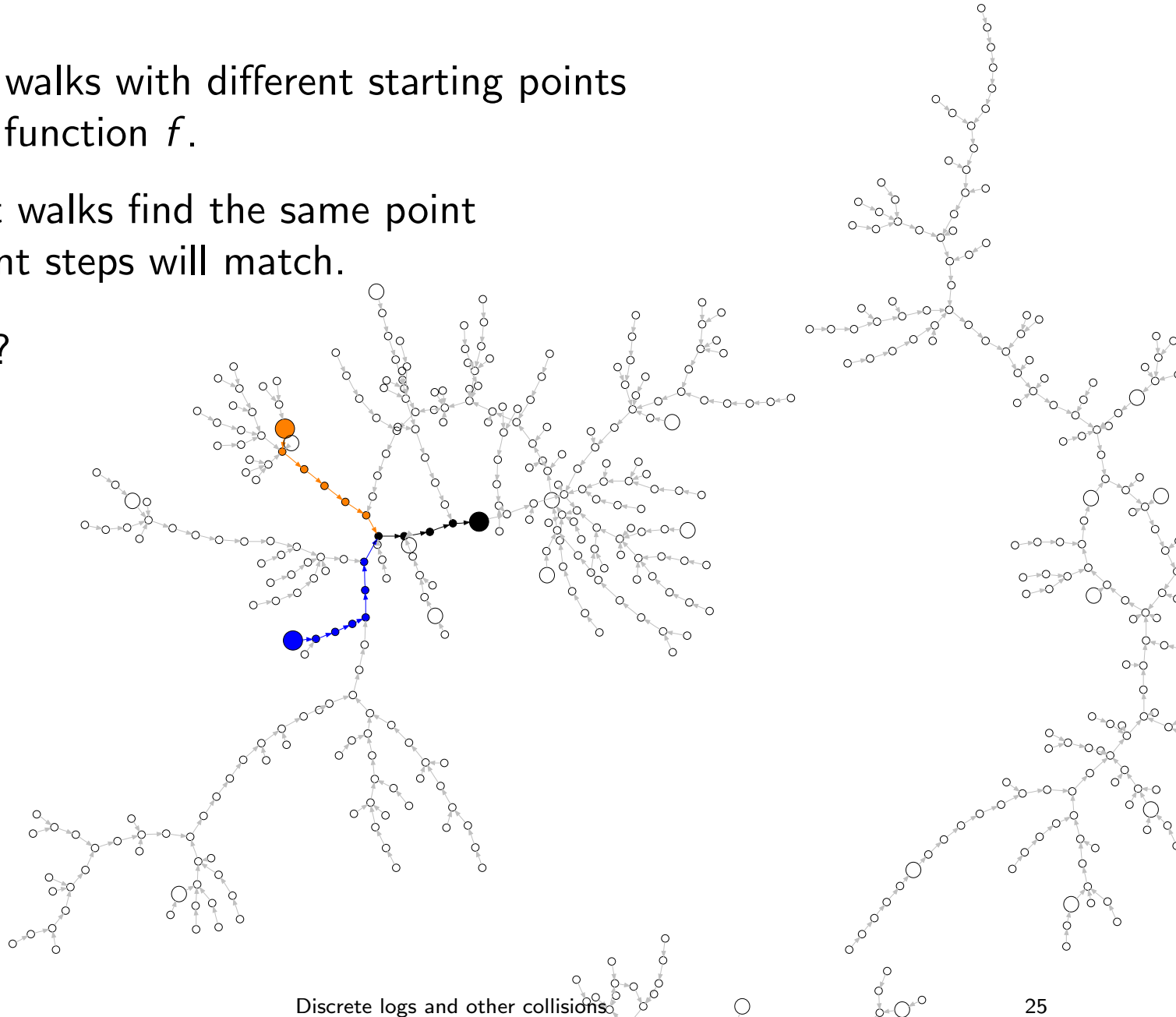


Van Oorschot and Wiener parallel collision search

Perform many walks with different starting points but same step function f.

If two different walks find the same point their subsequent steps will match.

How to notice?

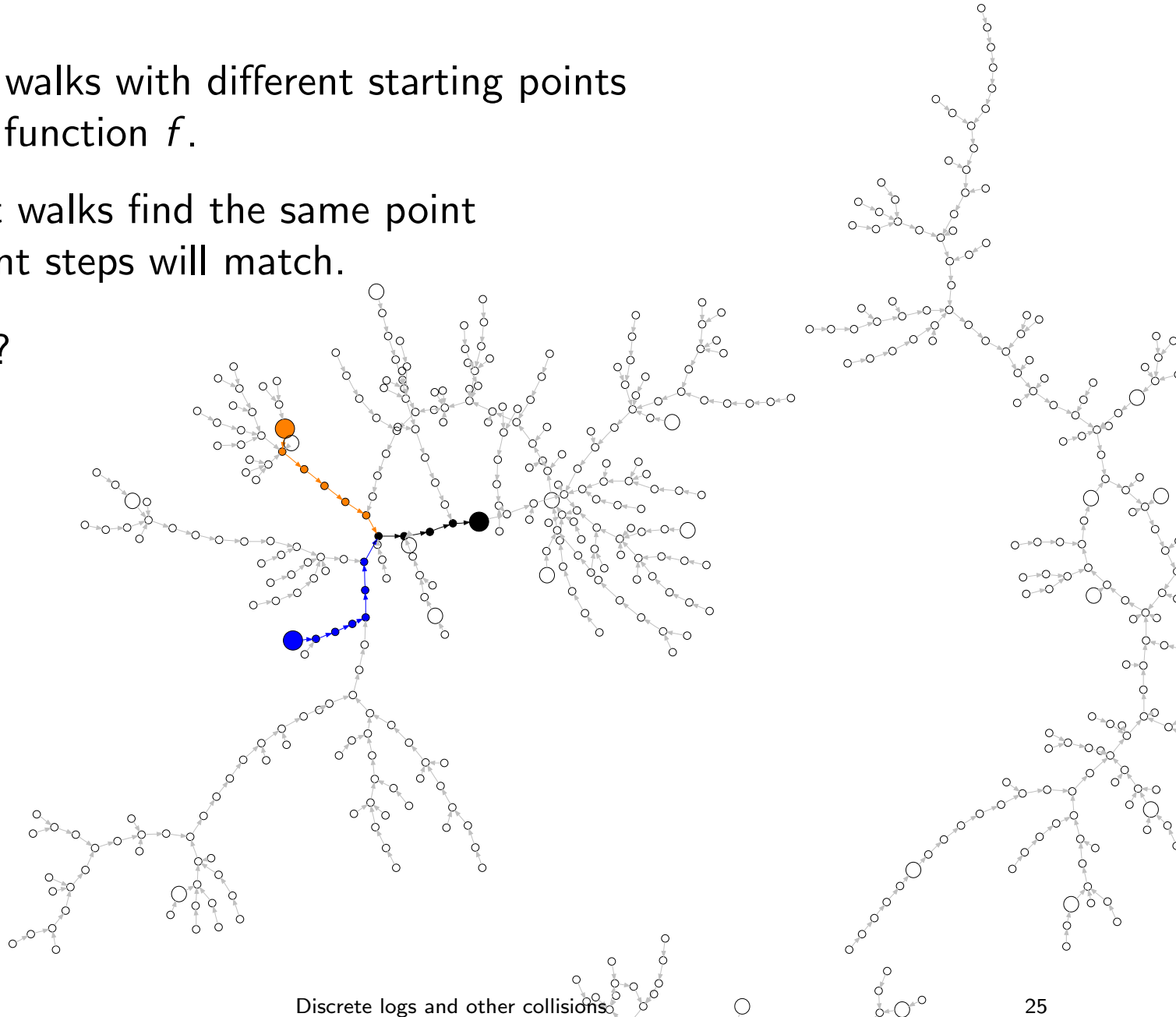


Van Oorschot and Wiener parallel collision search

Perform many walks with different starting points but same step function f.

If two different walks find the same point their subsequent steps will match.

How to notice?
when to stop?



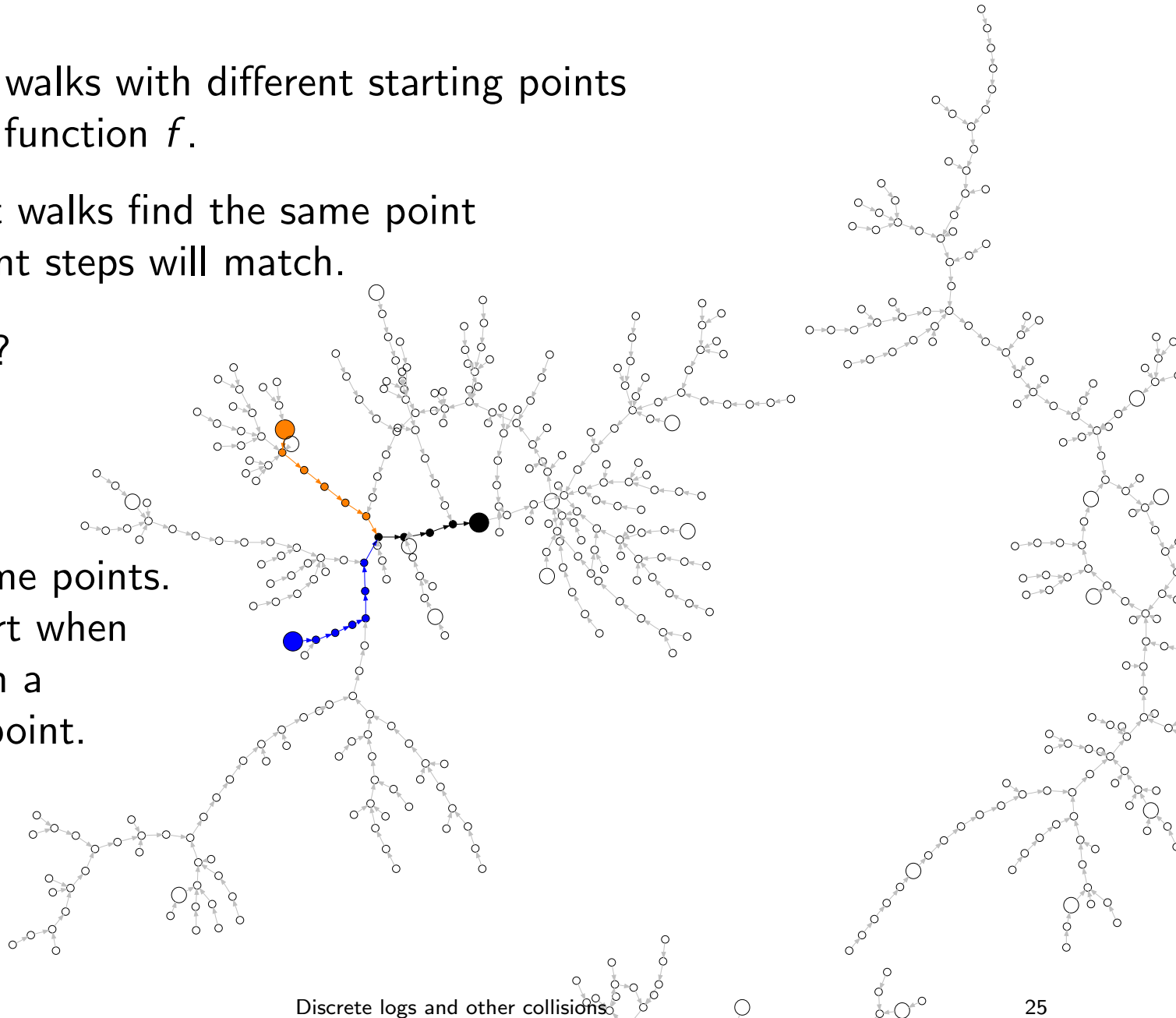
Van Oorschot and Wiener parallel collision search

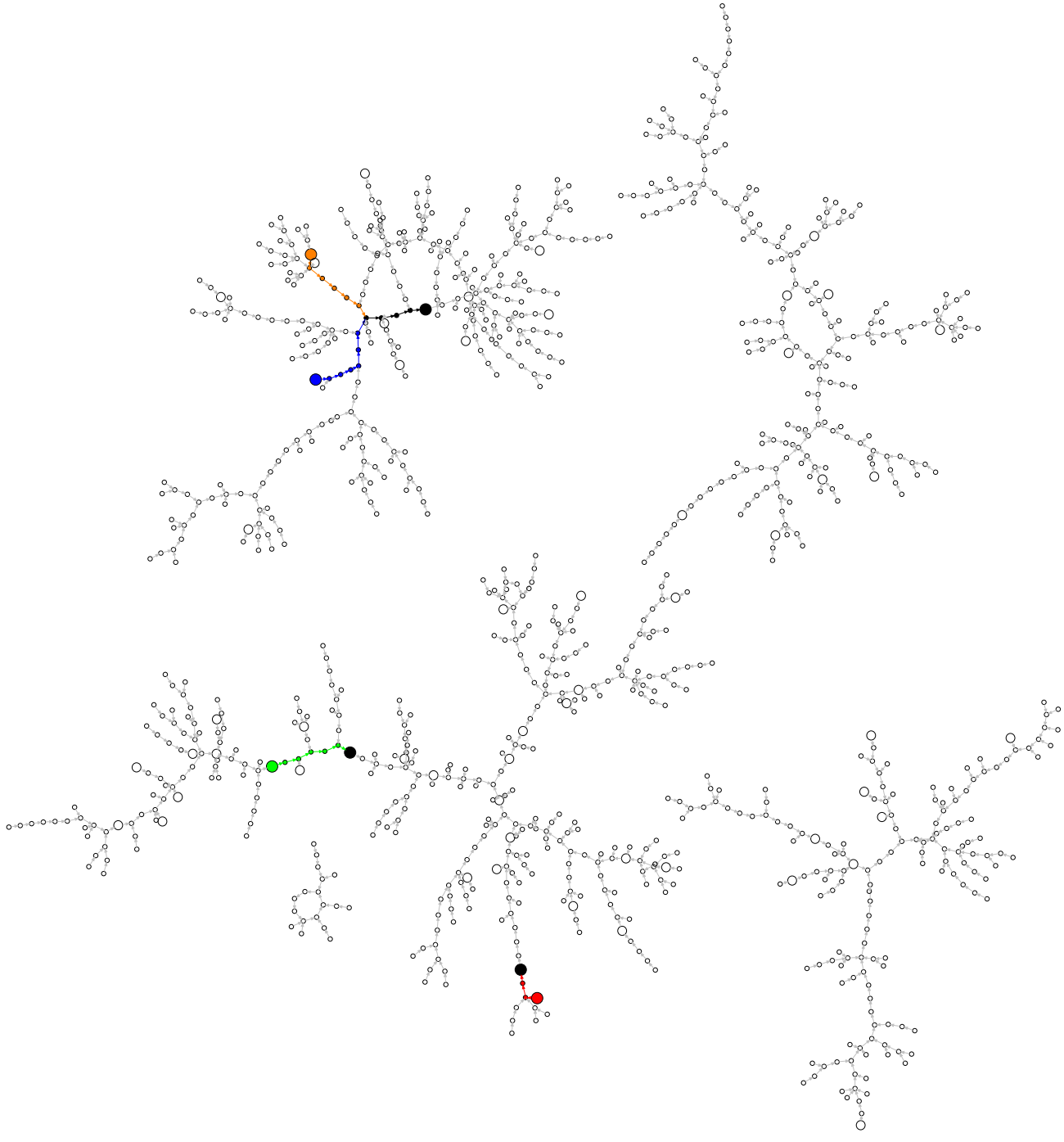
Perform many walks with different starting points but same step function f.

If two different walks find the same point their subsequent steps will match.

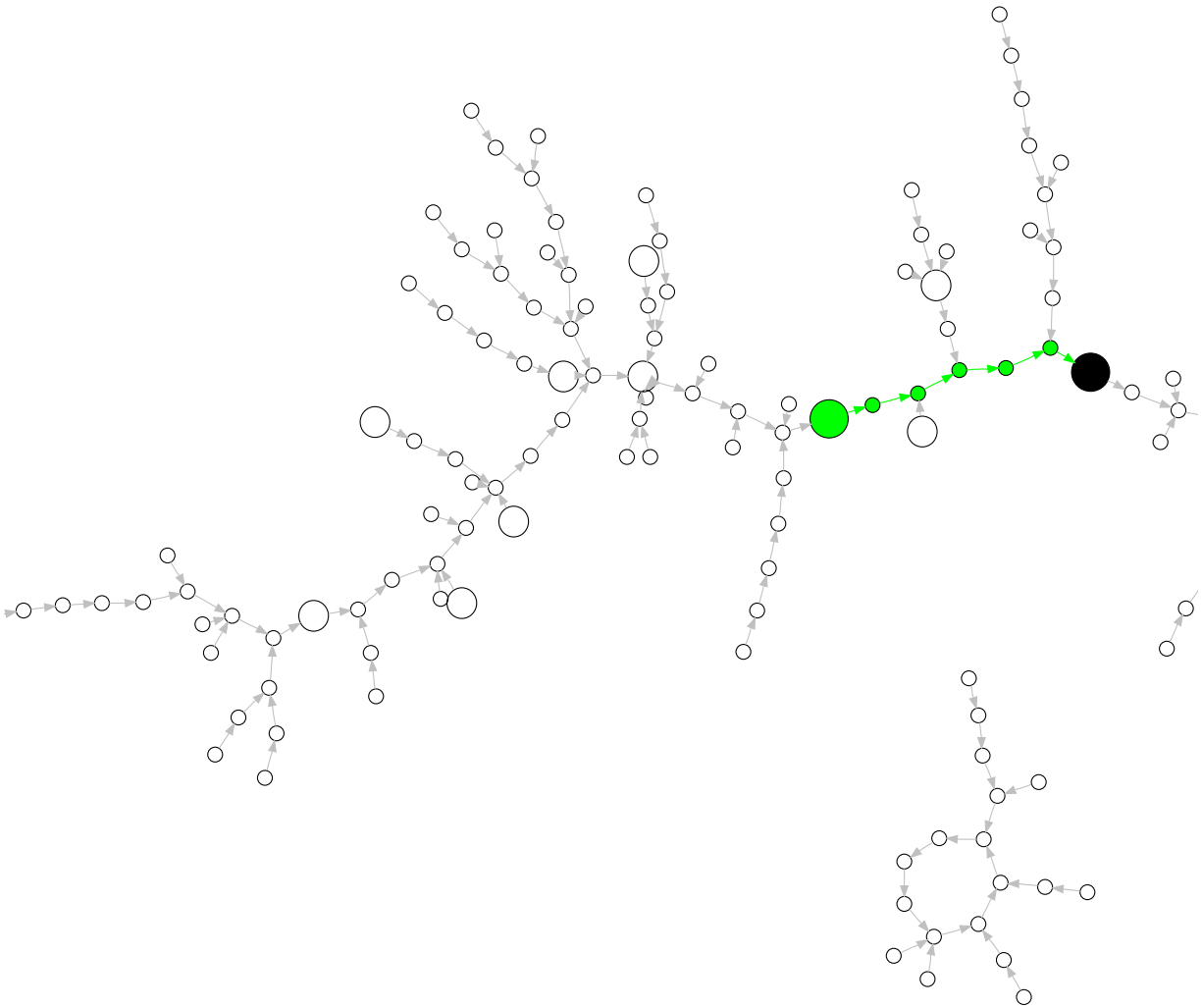
How to notice?
when to stop?

Idea: mark some points. Stop and report when getting to such a distinguished point.

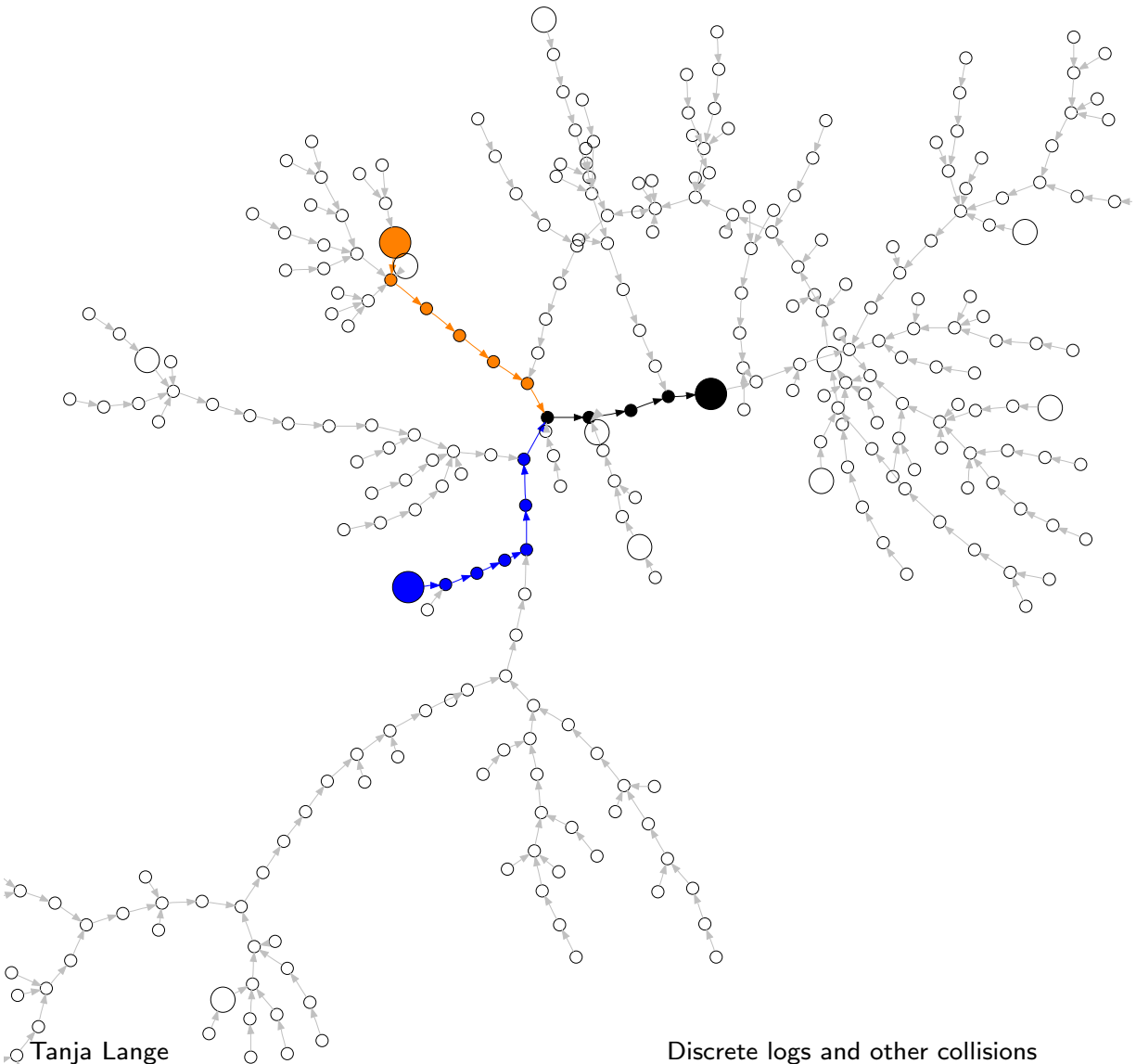




Walk to distinguished point, report to some server



Lucky case: two walks end in same distinguished point



Parallel Pollard rho

- Perform many walks with different starting points.
- Terminate each walk once it hits a distinguished point.
- Report the distinguished point and its a_i and b_i to server.
- Server receives, stores, and sorts all distinguished points.
- Two walks reaching same distinguished point give collision.
- As before, this collision solves the DLP.

Parallel Pollard rho

- Perform many walks with different starting points.
- Terminate each walk once it hits a distinguished point.
- Report the distinguished point and its a_i and b_i to server.
- Server receives, stores, and sorts all distinguished points.
- Two walks reaching same distinguished point give collision.
- As before, this collision solves the DLP.

How to identify distinguished points? Want something efficient to test.
Typical: top r bits of $x(W)$ are 0, takes $\approx 2^r$ steps to reach.

Parallel Pollard rho

- Perform many walks with different starting points.
- Terminate each walk once it hits a distinguished point.
- Report the distinguished point and its a_i and b_i to server.
- Server receives, stores, and sorts all distinguished points.
- Two walks reaching same distinguished point give collision.
- As before, this collision solves the DLP.

How to identify distinguished points? Want something efficient to test.
Typical: top r bits of $x(W)$ are 0, takes $\approx 2^r$ steps to reach.

How frequent should they be?
Infrequent: small number of very long walks, little storage on server, long delay before a collision is recognized. Must not hit cycle!

More frequent: shorter walks, more storage, more communication.
Too frequent: very short, degrades to random search.

Parallel Pollard rho

- Perform many walks with different starting points.
- Terminate each walk once it hits a distinguished point.
- Report the distinguished point and its a_i and b_i to server.
- Server receives, stores, and sorts all distinguished points.
- Two walks reaching same distinguished point give collision.
- As before, this collision solves the DLP.

How to identify distinguished points? Want something efficient to test.
Typical: top r bits of $x(W)$ are 0, takes $\approx 2^r$ steps to reach.

How frequent should they be?
Infrequent: small number of very long walks, little storage on server, long delay before a collision is recognized. Must not hit cycle!

More frequent: shorter walks, more storage, more communication.
Too frequent: very short, degrades to random search.

Any choice (unless walk enters cycle without DP) needs $\sqrt{n}$ steps.

Parallel Pollard rho

- Perform many walks with different starting points.
- Terminate each walk once it hits a distinguished point.
- Report the distinguished point and its a_i and b_i to server.
- Server receives, stores, and sorts all distinguished points.
- Two walks reaching same distinguished point give collision.
- As before, this collision solves the DLP.

How to identify distinguished points? Want something efficient to test.
Typical: top r bits of $x(W)$ are 0, takes $\approx 2^r$ steps to reach.

How frequent should they be?
Infrequent: small number of very long walks, little storage on server, long delay before a collision is recognized. Must not hit cycle!

More frequent: shorter walks, more storage, more communication.
Too frequent: very short, degrades to random search.

Any choice (unless walk enters cycle without DP) needs $\sqrt{n}$ steps.

Several speedups, e.g. identifying W and $-W$, some pitfalls.

Summary of DL systems

All attacks in this unit are *generic* attacks, i.e., they work in any group.

We did not cover the Pohlig-Hellman attack, another generic attack. See lectures 6 & 7 of playlist Discrete Logarithms on my YouTube Channel Cryptology for this.

In summary, Pohlig-Hellman reduces security of DLP to security of largest prime order subgroup. Many groups are much weaker than their size n predicts!

Summary of DL systems

All attacks in this unit are *generic* attacks, i.e., they work in any group.

We did not cover the Pohlig-Hellman attack, another generic attack. See lectures 6 & 7 of playlist Discrete Logarithms on my YouTube Channel Cryptology for this.

In summary, Pohlig-Hellman reduces security of DLP to security of largest prime order subgroup. Many groups are much weaker than their size n predicts!

Let $n = \prod p_i^{e_i}$, $\ell = \max\{p_i\}$.

Breaking DLP costs $O(\sqrt{\ell})(\log n)^{O(1)}$ bit operations.

O ignores all constants and lower order terms. $(\log n)^{O(1)}$ covers scalar multiplications and cost of group operations as well as operations in Pohlig-Hellman.

Hash functions overview and Pollard rho for collision finding

Goals of cryptographic hash functions

What do we want from a hash function $H : \{0,1\}^* \to \{0,1\}^n$?

For any string x, think of $H(x)$ as an n-bit fingerprint of x.

Goals:

- $H(x)$ looks totally random;
- nobody can find two different strings x, x' with $H(x) = H(x')$;
- any tiny change from x to x' makes a totally new $H(x')$;
- nobody can compute $H(x)$ without knowing all of x;
- nobody can compute a secret x given only $H(x)$;
- ...

Warning: Some hash goals are difficult to mathematically define.

Generic hardness of preimage resistance

Goal: Given $y \in H(\{0,1\}^*)$,
finding $x \in \{0,1\}^*$ with $H(x) = y$ is hard.

Here y is given, and is known to be the image of some $x \in \{0,1\}^*$.
Typically there are many such x,
but it should be hard to find any.

Generic hardness of preimage resistance

Goal: Given $y \in H(\{0,1\}^*)$,
finding $x \in \{0,1\}^*$ with $H(x) = y$ is hard.

Here y is given, and is known to be the image of some $x \in \{0,1\}^*$.
Typically there are many such x,
but it should be hard to find any.

Generic attack: Try $\approx 2^n$ random choices of x.
If the output of H is distributed uniformly then
each x has a $1/2^n$ chance of $H(x) = y$.

e.g. $\approx 2^{128}$ tries if $n = 128$: very expensive.

Generic hardness of second-preimage resistance

Goal: Given $x \in \{0,1\}^*$, finding $x' \in \{0,1\}^*$
with $x \neq x'$ and $H(x') = H(x)$ is hard.

Here x is given, determining $y = H(x)$.
Typically there are many other $x' \neq x$ with the same image,
but it should be computationally hard to find any.

Generic hardness of second-preimage resistance

Goal: Given $x \in \{0,1\}^*$, finding $x' \in \{0,1\}^*$
with $x \neq x'$ and $H(x') = H(x)$ is hard.

Here x is given, determining $y = H(x)$.
Typically there are many other $x' \neq x$ with the same image,
but it should be computationally hard to find any.

Generic attack: Try $\approx 2^n$ random choices of $x' \neq x$.
Same speed as for first preimages.

Generic hardness of collision resistance

Goal: Finding $x, x' \in \{0,1\}^*$
with $x \neq x'$ and $H(x') = H(x)$ is hard.

Attacker has full flexibility to choose any output y.
It should still be hard
to find two different strings x, x' with the same output.

Generic hardness of collision resistance

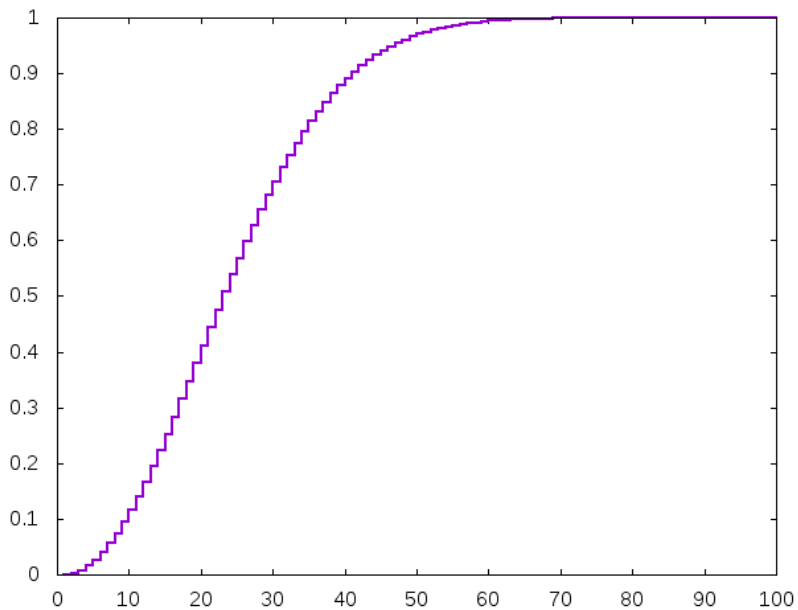
Goal: Finding $x, x' \in \{0,1\}^*$
with $x \neq x'$ and $H(x') = H(x)$ is hard.

Attacker has full flexibility to choose any output y.
It should still be hard
to find two different strings x, x' with the same output.

Generic attack: Try $\approx 2^{n/2}$ random choices of x. This number is much lower than 2^n because there is no restriction on the target.

The "birthday paradox": if one draws
$\approx 1.17\sqrt{m}$ elements at random
from a set of m elements,
then with $\approx 50\%$ probability
one has picked one element twice.

Note: this needs lots of storage,
if implemented as sketched here.



Pollard's rho method for collision finding

Avoid storage by performing random walks and checking for collisions.
How to turn collision search into random walk?

Pollard's rho method for collision finding

Avoid storage by performing random walks and checking for collisions.
How to turn collision search into random walk?

Just pick random starting point $W_0 \in \{0,1\}^\ell$ and iterate

$$W_{i+1} = h(W_i).$$

Pollard's rho method for collision finding

Avoid storage by performing random walks and checking for collisions.
How to turn collision search into random walk?

Just pick random starting point $W_0 \in \{0,1\}^\ell$ and iterate

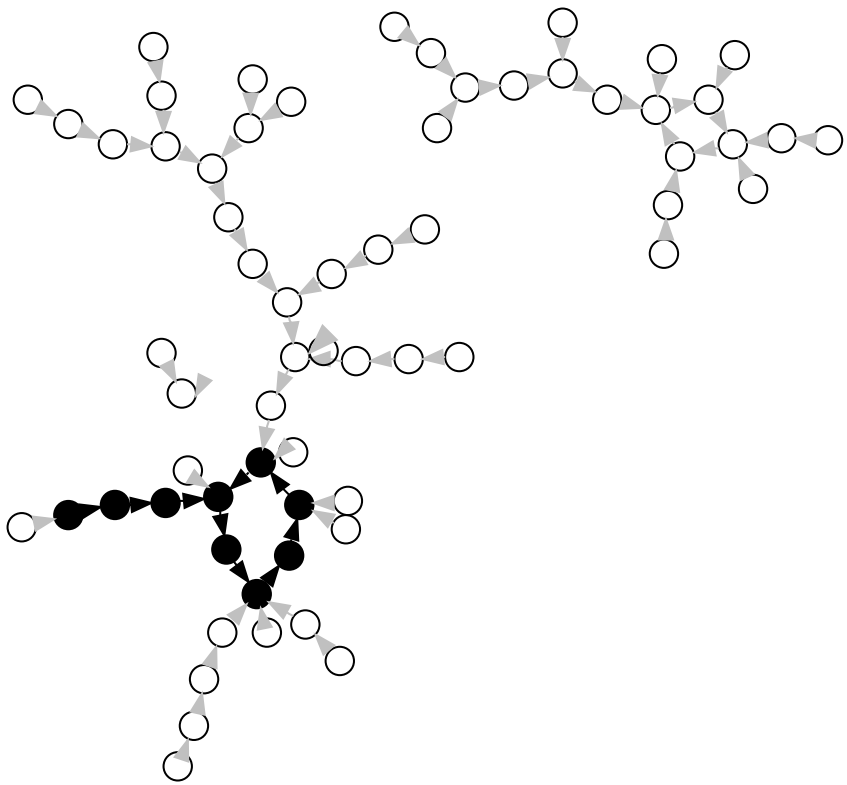
$$W_{i+1} = h(W_i).$$

Find collision using Floyd.
If $W_{2i} = W_i$ we have met on the cycle, but we need the initial collision and h is not invertible.

Requires to remember the number i of small steps taken.
Cycle length is divisor of i.

Compare $W_j \stackrel{?}{=} W_{i+j}$ for $j=0,1,2,\ldots$ to find collision at end of tail.

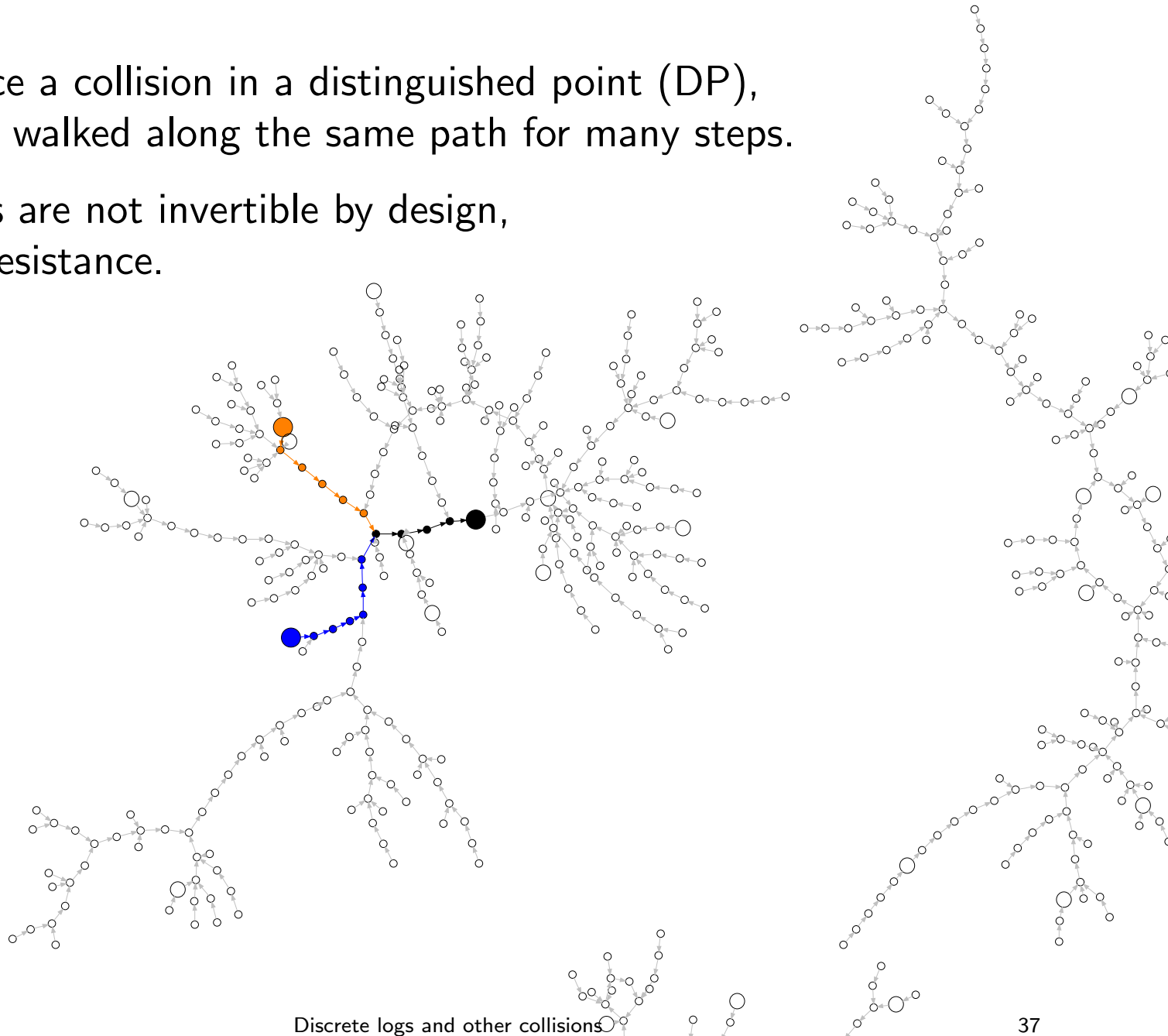
Several improvements designed to avoid full recomputation.



How to handle parallel collision search?

When we notice a collision in a distinguished point (DP), we might have walked along the same path for many steps.

Hash functions are not invertible by design, see preimage resistance.



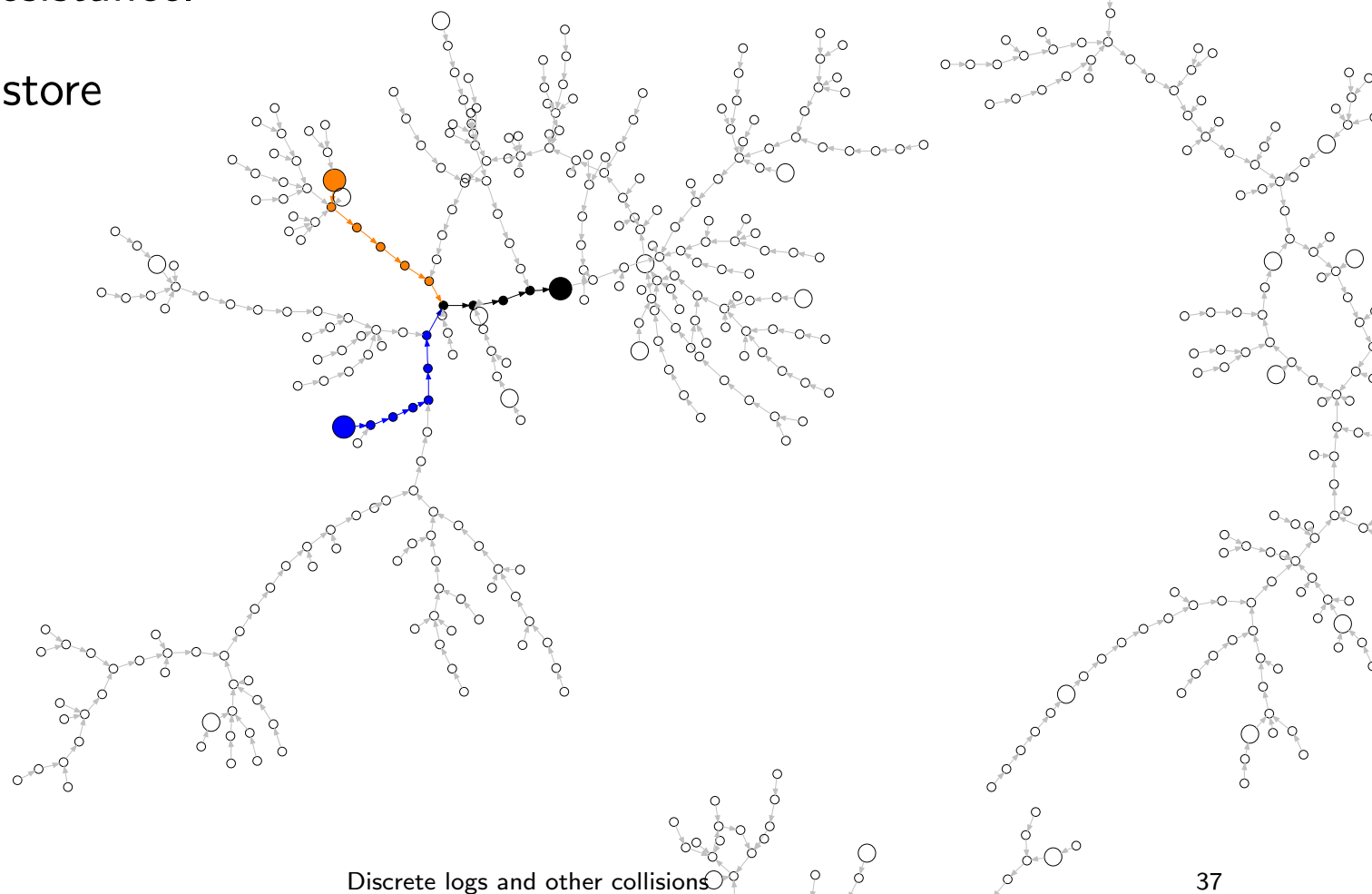
How to handle parallel collision search?

When we notice a collision in a distinguished point (DP), we might have walked along the same path for many steps.

Hash functions are not invertible by design, see preimage resistance.

For each walk store

- start
- end (DP)
- # steps



How to handle parallel collision search?

When we notice a collision in a distinguished point (DP), we might have walked along the same path for many steps.

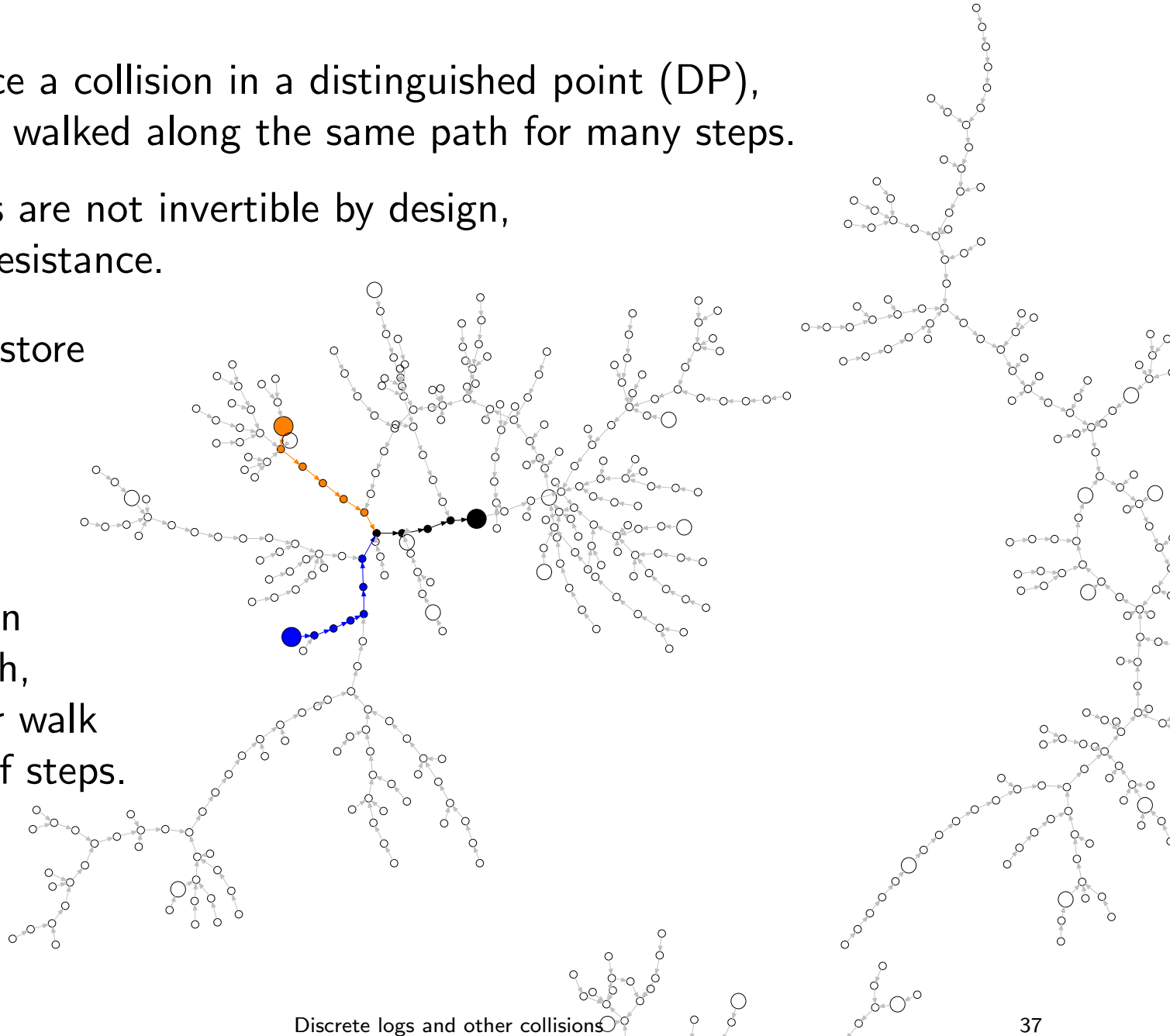
Hash functions are not invertible by design, see preimage resistance.

For each walk store

- start
- end (DP)
- # steps

To find collision after DP match, advance longer walk by difference of steps.

Then advance both, checking each time.



Factorization overview and Pollard rho for factorization

How to factor RSA numbers?

Trial division.

How to factor RSA numbers?

Trial division.

The prime-number theorem says that there are about

$$n/\ln(n)$$

primes up to n.

How to factor RSA numbers?

Trial division.

The prime-number theorem says that there are about

$$n/\ln(n)$$

primes up to n.

That means roughly

$$(2^{2048}/\ln(2^{2048})) - (2^{2047}/\ln(2^{2047})) = 1.1377 \cdot 10^{613}$$

primes with 2048 bits.

No chance to find p or q by trial factorization.

How to factor RSA numbers?

Trial division.

The prime-number theorem says that there are about

$$n/\ln(n)$$

primes up to n.

That means roughly

$$(2^{2048}/\ln(2^{2048})) - (2^{2047}/\ln(2^{2047})) = 1.1377 \cdot 10^{613}$$

primes with 2048 bits.

No chance to find p or q by trial factorization.

But: trial factorization is a useful step when factoring normal numbers.

Short summary of factorization methods

- For small factors: trial factorization.

Short summary of factorization methods

- For small factors: trial factorization.
- For medium factors:
  - Pollard's rho method.
  - $p-1$ method, $p+1$ method, ECM (elliptic curve method).

Short summary of factorization methods

- For small factors: trial factorization.
- For medium factors:
  - Pollard's rho method.
  - $p-1$ method, $p+1$ method, ECM (elliptic curve method).
- For RSA numbers: Number field sieve
  - Works by turning hard factorization of one number into many easier factorizations.
  - Uses sieving (think of Eratosthenes) to find small factors.
  - Uses the above to find medium size factors.
  - Also needs a stage of linear algebra at the end.
- The number field sieve has subexponential complexity, so we need to more than double the bit length to make the attack twice as hard.

Will use n for RSA numbers (hard to factor) and m for normal numbers. Typically, m is odd without very small prime divisors.

Pollard's rho method for factorization

Define $\rho_0 = 0$, $\rho_{k+1} = \rho_k^2 + 11$.

Every prime $\leq 2^{20}$ divides
$S = (\rho_1 - \rho_2)(\rho_2 - \rho_4)(\rho_3 - \rho_6)\cdots(\rho_{3575} - \rho_{7150})$.
Also many larger primes do.

If such p divides m, it divides $\gcd(S, m)$.
Computing S takes $\approx 2^{14}$ multiplications mod m, very little memory.

Compare to $\approx 2^{16}$ divisions for trial division up to 2^{20}.

Using Pollard rho to factor m means computing $\rho_{k+1} = \rho_k^2 + 11 \bmod m$.

Pollard's rho method for factorization

Define $\rho_0 = 0$, $\rho_{k+1} = \rho_k^2 + 11$.

Every prime $\leq 2^{20}$ divides
$S = (\rho_1 - \rho_2)(\rho_2 - \rho_4)(\rho_3 - \rho_6)\cdots(\rho_{3575} - \rho_{7150})$.
Also many larger primes do.

If such p divides m, it divides $\gcd(S, m)$.
Computing S takes $\approx 2^{14}$ multiplications mod m, very little memory.

Compare to $\approx 2^{16}$ divisions for trial division up to 2^{20}.

Using Pollard rho to factor m means computing $\rho_{k+1} = \rho_k^2 + 11 \bmod m$.

More generally: Choose z.
Compute $\gcd(S, m)$ where $S = (\rho_1 - \rho_2)(\rho_2 - \rho_4)\cdots(\rho_z - \rho_{2z})$.

Analysis of Pollard's rho method for factorization

More generally: Choose z.
Compute $\gcd(S, m)$ where $S = (\rho_1 - \rho_2)(\rho_2 - \rho_4)\cdots(\rho_z - \rho_{2z})$.

How big does z have to be for all primes $\leq y$ to divide S?

Analysis of Pollard's rho method for factorization

More generally: Choose z.
Compute $\gcd(S, m)$ where $S = (\rho_1 - \rho_2)(\rho_2 - \rho_4)\cdots(\rho_z - \rho_{2z})$.

How big does z have to be for all primes $\leq y$ to divide S?

Consider walk ρ_i modulo p.
There are p elements modulo p, so expect collision $\rho_i \equiv \rho_j \bmod p$ after $\sqrt{\pi p/2}$ steps.

Analysis of Pollard's rho method for factorization

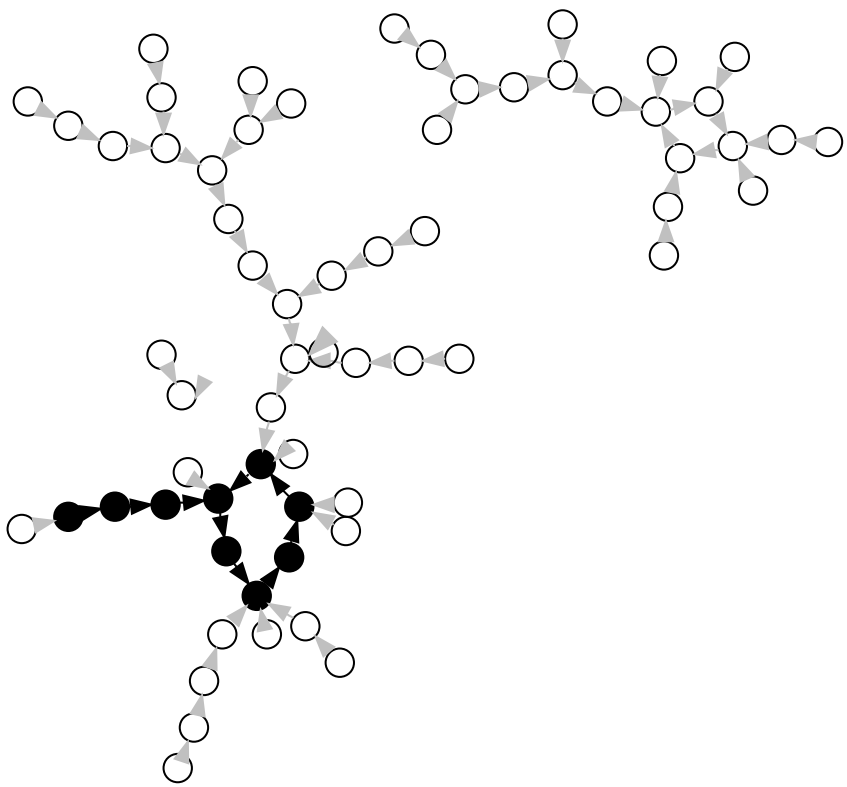
More generally: Choose z.
Compute $\gcd(S, m)$ where $S = (\rho_1 - \rho_2)(\rho_2 - \rho_4)\cdots(\rho_z - \rho_{2z})$.

How big does z have to be for all primes $\leq y$ to divide S?

Consider walk ρ_i modulo p.
There are p elements modulo p, so expect collision $\rho_i \equiv \rho_j \bmod p$ after $\sqrt{\pi p/2}$ steps.

Problem: We don't see collision as we work modulo m, not p.
But p divides $\gcd(\rho_i - \rho_j, m)$.

S implicitly uses Floyd,
product reduces number of gcd steps:
$\rho_i \equiv \rho_j \bmod p \Rightarrow \rho_k \equiv \rho_{2k} \bmod p$
for $k \in (j-i)\mathbf{Z} \cap [i, \infty] \cap [j, \infty]$.

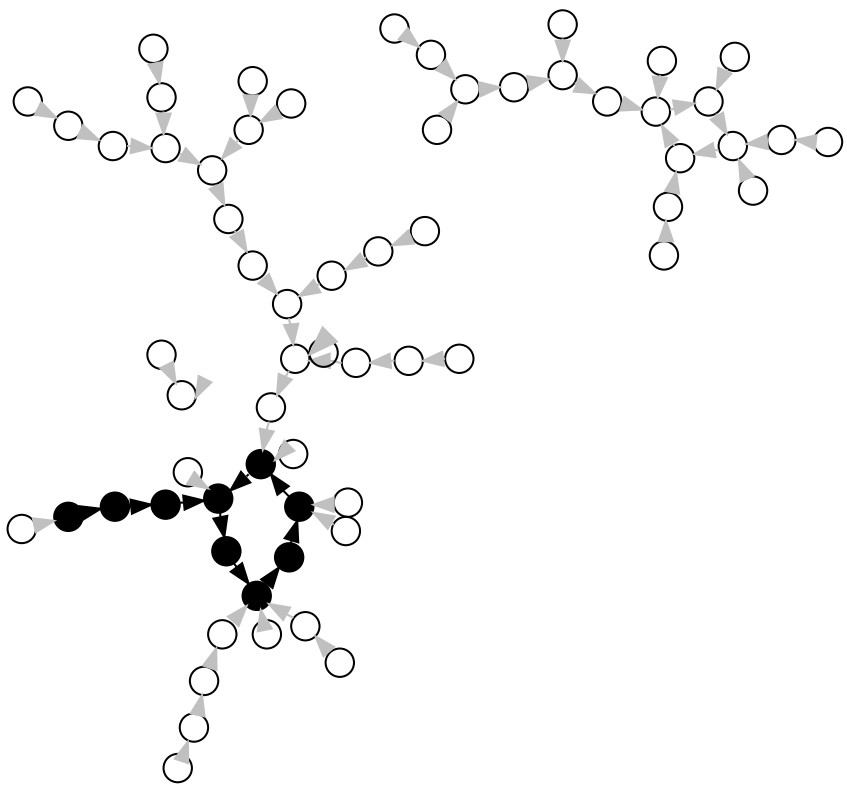


Analysis of Pollard's rho method for factorization

More generally: Choose z.
Compute $\gcd(S, m)$ where $S = (\rho_1 - \rho_2)(\rho_2 - \rho_4)\cdots(\rho_z - \rho_{2z})$.

How big does z have to be for all primes $\leq y$ to divide S?

Consider walk ρ_i modulo p.
There are p elements modulo p, so expect collision $\rho_i \equiv \rho_j \bmod p$ after $\sqrt{\pi p/2}$ steps.



Problem: We don't see collision as we work modulo m, not p.
But p divides $\gcd(\rho_i - \rho_j, m)$.

S implicitly uses Floyd,
product reduces number of gcd steps:
$\rho_i \equiv \rho_j \bmod p \Rightarrow \rho_k \equiv \rho_{2k} \bmod p$
for $k \in (j-i)\mathbf{Z} \cap [i, \infty] \cap [j, \infty]$.

Plausible conjecture: $y^{1/2+o(1)}$; so $y^{1/2+o(1)}$ mults mod m.