Cryptanalysis Course
Part II – DLP

Tanja Lange
Technische Universiteit Eindhoven

29 Nov 2016

with some slides by
Daniel J. Bernstein

More elliptic curves

Edwards curves are elliptic.
Easiest way to understand
elliptic curves is Edwards.

Geometrically, all elliptic curves
are Edwards curves.

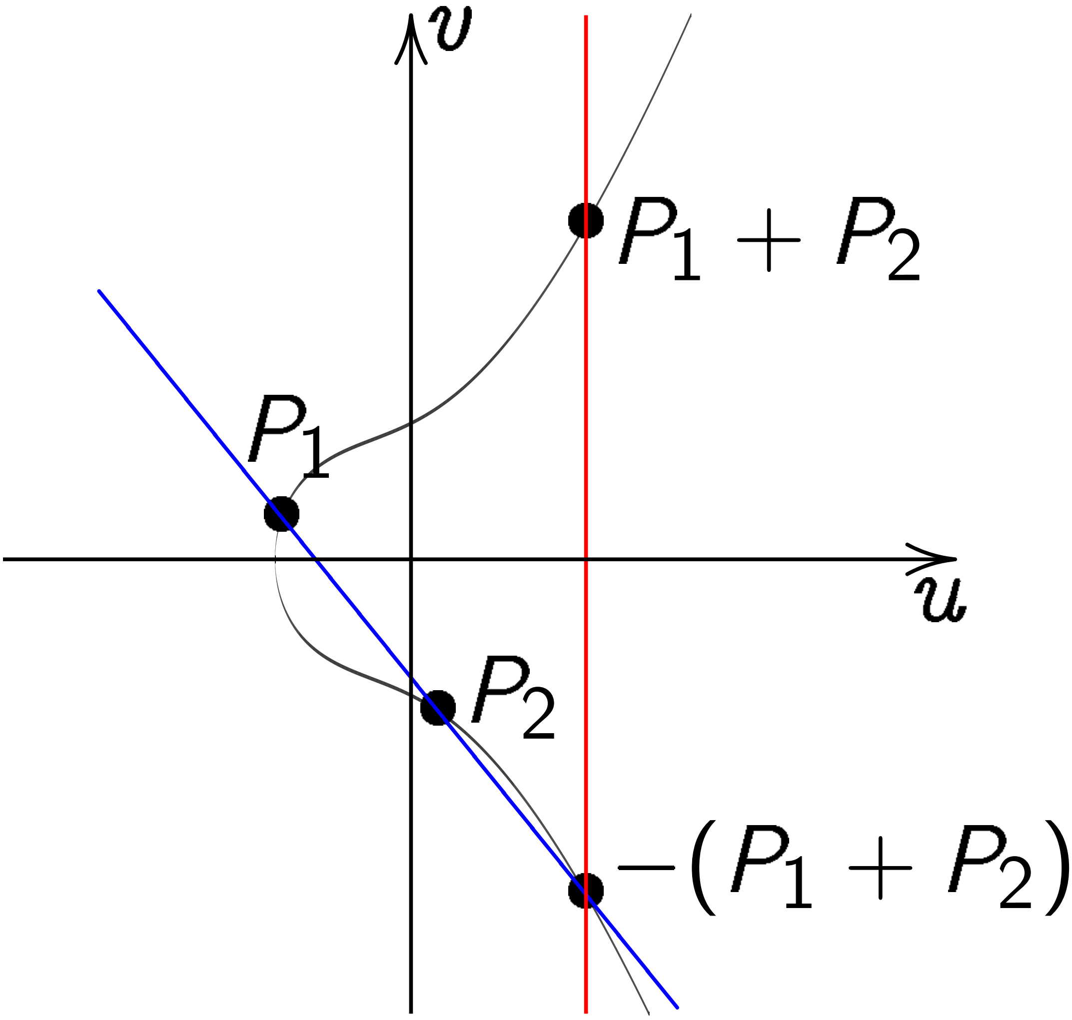
Algebraically,
more elliptic curves exist
(not always point of order 4).

Every odd-char curve can be
expressed as Weierstrass curve
$v^2 = u^3 + a_2u^2 + a_4u + a_6$.

Warning: "Weierstrass" has
different meaning in char 2.

Addition on Weierstrass curve

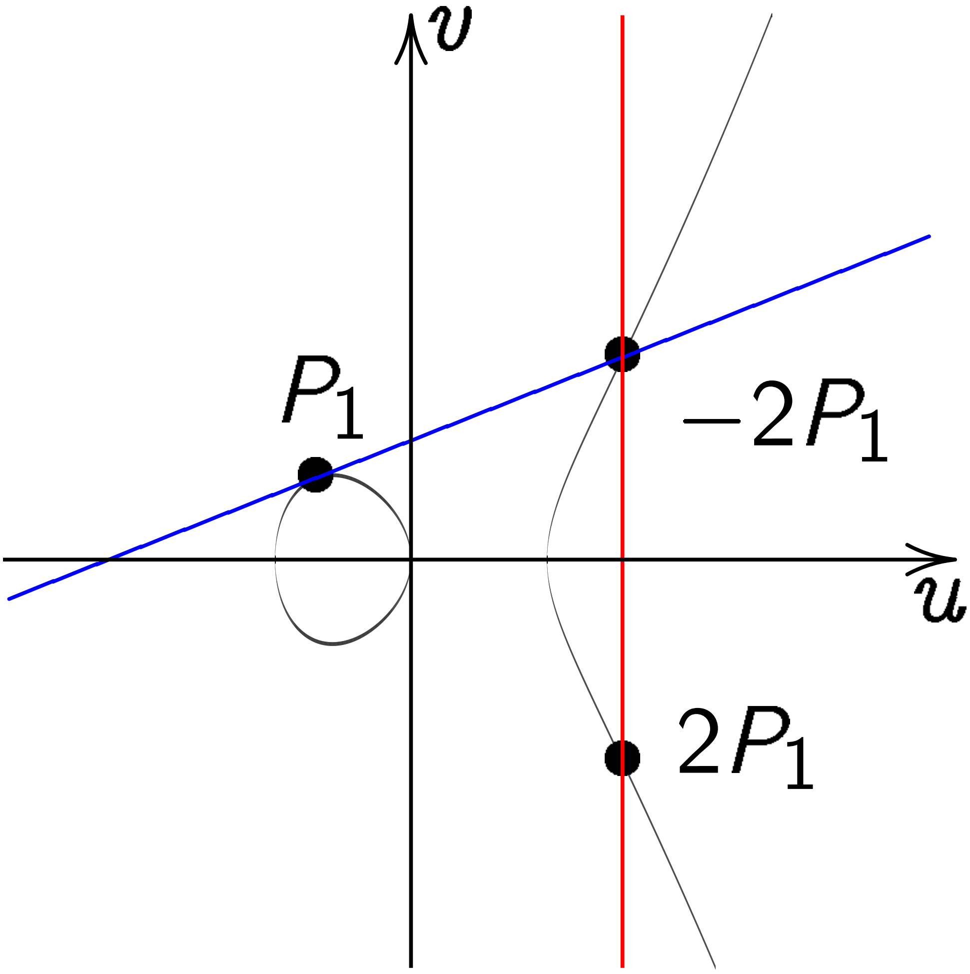
$v^2 = u^3 + u^2 + u + 1$



Slope $\lambda = (v_2 - v_1)/(u_2 - u_1)$.
Note that $u_1 \neq u_2$. Some points missing, in particular ∞.

Doubling on Weierstrass curve

$v^2 = u^3 - u$



Slope $\lambda = (3u_1^2 - 1)/(2v_1)$.

In most cases $(u_1, v_1) + (u_2, v_2) = (u_3, v_3)$ where $(u_3, v_3) = (\lambda^2 - u_1 - u_2, \lambda(u_1 - u_3) - v_1)$.

$u_1 \neq u_2$, "addition" (alert!): $\lambda = (v_2 - v_1)/(u_2 - u_1)$. Total cost $1\mathbf{I} + 2\mathbf{M} + 1\mathbf{S}$.

$(u_1, v_1) = (u_2, v_2)$ and $v_1 \neq 0$, "doubling" (alert!): $\lambda = (3u_1^2 + 2a_2u_1 + a_4)/(2v_1)$. Total cost $1\mathbf{I} + 2\mathbf{M} + 2\mathbf{S}$.

Also handle some exceptions: $(u_1, v_1) = (u_2, -v_2)$; ∞ as input. Messy to implement and test.

Birational equivalence

Starting from point (x, y)
on $x^2 + y^2 = 1 + dx^2y^2$:

Define $A = 2(1 + d)/(1 - d)$,
$B = 4/(1 - d)$;
$u = (1 + y)/(B(1 - y))$,
$v = u/x = (1 + y)/(Bx(1 - y))$.
(Skip a few exceptional points.)

Then (u, v) is a point on
a Weierstrass curve:
$v^2 = u^3 + (A/B)u^2 + (1/B^2)u$.

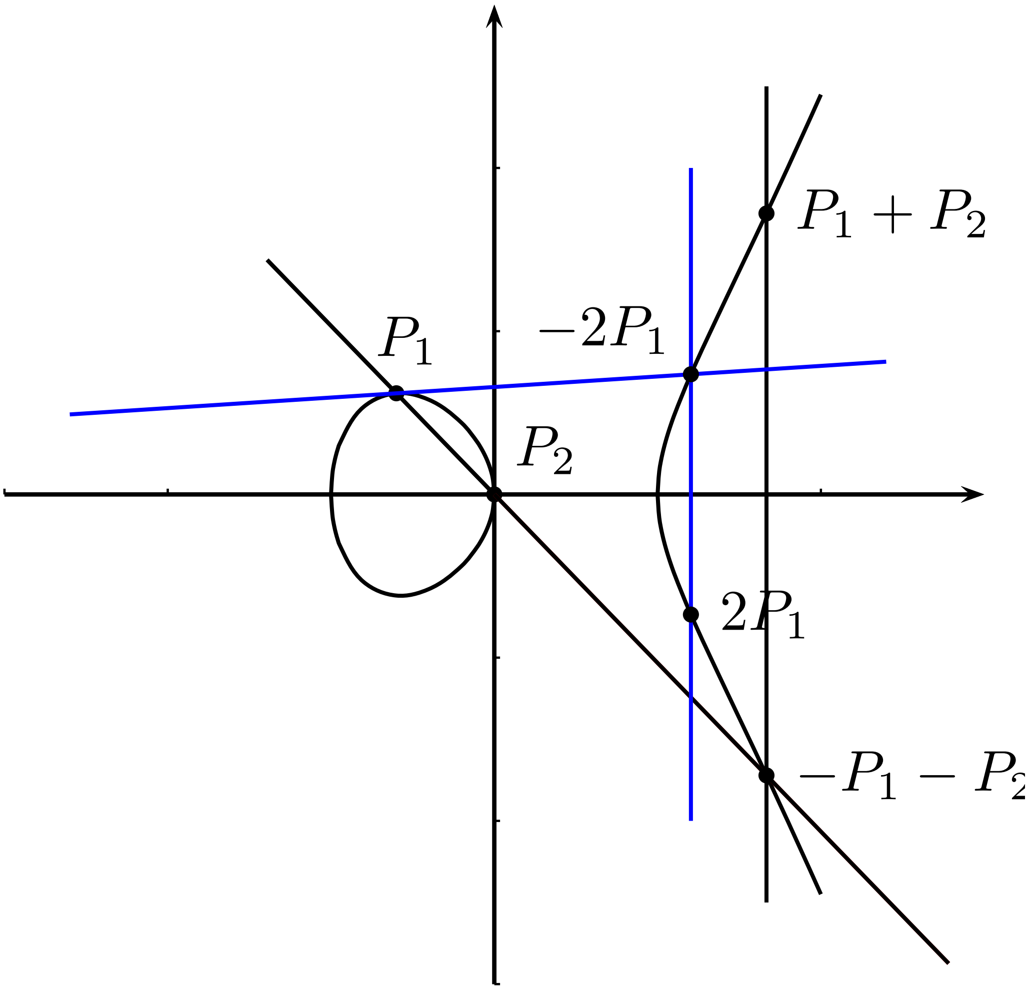
Easily invert this map:
$x = u/v$, $y = (Bu - 1)/(Bu + 1)$.

Attacker can transform Edwards curve to Weierstrass curve and vice versa; $n(x, y) \mapsto n(u, v)$.
$\Rightarrow$ Same discrete-log security!
Can choose curve representation so that implementation of attack is faster/easier.

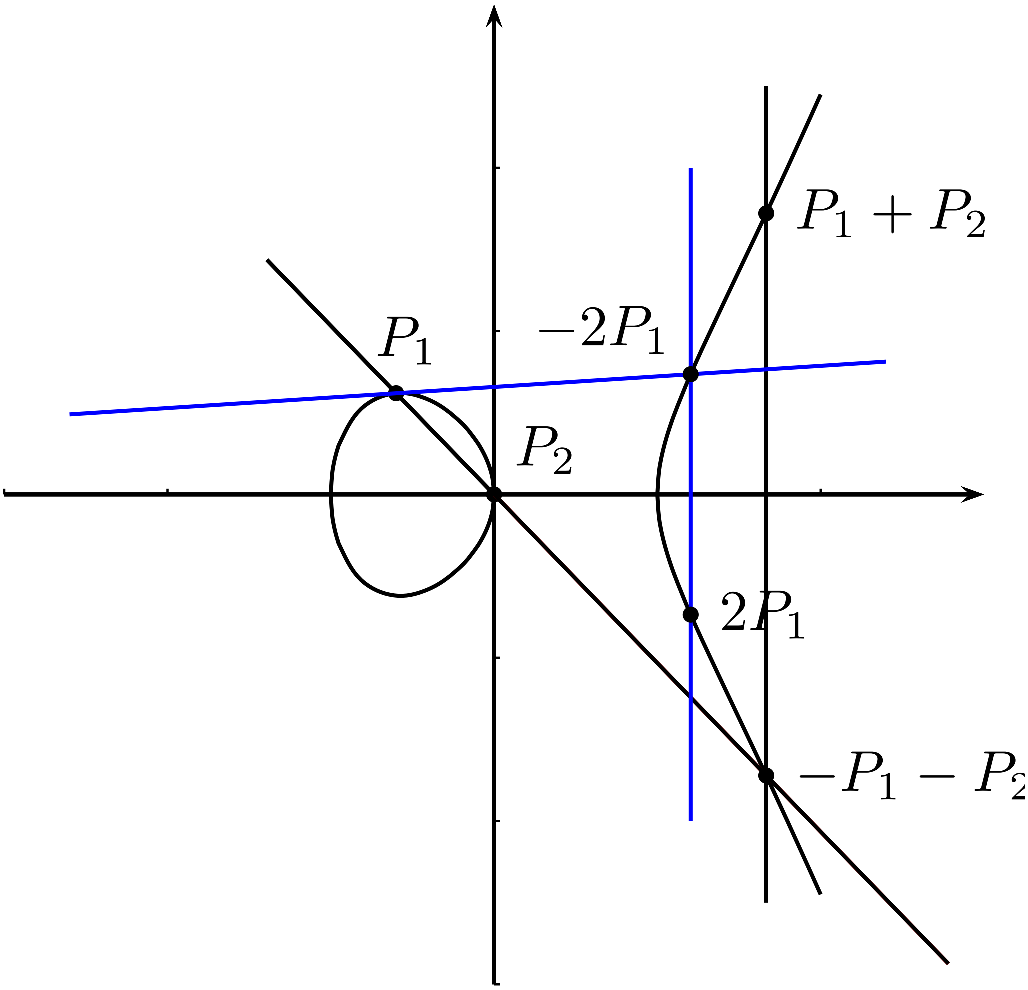
System designer can choose curve representation so that protocol runs fastest; no need to worry about security degradation.

Optimization targets are different.

Elliptic-curve groups



Elliptic-curve groups



Following algorithms will need a unique representative per point. For that Weierstrass curves are the speed leader.

The discrete-logarithm problem

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

This curve has

The discrete-logarithm problem

Define $p = 1000003$ and consider the Weierstrass curve $y^2 = x^3 - x$ over $\mathbf{F}_p$.
This curve has
$1000004 = 2^2 \cdot 53^2 \cdot 89$
points and $P = (101384, 614510)$ is a point of order $2 \cdot 53^2 \cdot 89$.

The discrete-logarithm problem

Define $p = 1000003$ and
consider the Weierstrass curve
$y^2 = x^3 - x$ over $\mathbf{F}_p$.
This curve has
$1000004 = 2^2 \cdot 53^2 \cdot 89$
points and $P = (101384, 614510)$
is a point of order $2 \cdot 53^2 \cdot 89$.
In general, 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
$\mathbf{Z}/n \times \mathbf{Z}/m$, where $n|m$ and
$n|(p-1)$.

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

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

Could find n by brute force.
Is there a faster way?

Understanding brute force

Can compute successively
$1P = (101384, 614510)$,
$2P = (102361, 628914)$,
$3P = (77571, 87643)$,
$4P = (650289, 31313)$,
$500001P = -P$.
$500002P = \infty$.

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

Maximum cost of computation:
≤ 500001 additions of P;
≤ 500001 nanoseconds on a CPU
that does 1 ADD/nanosecond.

This is negligible work
for $p \approx 2^{20}$.

But users can
standardize a larger p,
making the attack slower.

Attack cost scales linearly:
$\approx 2^{50}$ ADDs for $p \approx 2^{50}$,
$\approx 2^{100}$ ADDs for $p \approx 2^{100}$, etc.

(Not exactly linearly:
cost of ADDs grows with p.
But this is a minor effect.)

Computation has a good chance of finishing earlier.
Chance scales linearly:
1/2 chance of 1/2 cost;
1/10 chance of 1/10 cost; etc.

"So users should choose large n."

Computation has a good chance of finishing earlier.
Chance scales linearly:
1/2 chance of 1/2 cost;
1/10 chance of 1/10 cost; etc.

"So users should choose large n."

That's pointless. We can apply "random self-reduction":
choose random r, say 69961;
compute $rP = (593450, 987590)$;
compute $(r + n)P$ as
$(593450, 987590) + (670366, 740819)$;
compute discrete log;
subtract r mod 500002; obtain n.

Computation can be parallelized.

One low-cost chip can run
many parallel searches.
Example, 2^6 €: one chip,
2^{10} cores on the chip,
each 2^{30} ADDs/second?
Maybe; see SHARCS workshops
for detailed cost analyses.

Attacker can run
many parallel chips.
Example, 2^{30} €: 2^{24} chips,
so 2^{34} cores,
so 2^{64} ADDs/second,
so 2^{89} ADDs/year.

Multiple targets and giant steps

Computation can be applied to many targets at once.

Given 100 DL targets $n_1 P$, $n_2 P$, $\ldots$, $n_{100} P$:
Can find *all* of $n_1, n_2, \ldots, n_{100}$ with ≤ 500002 ADDs.

Simplest approach: First build a sorted table containing $n_1 P$, $\ldots$, $n_{100} P$.
Then check table for $1P$, $2P$, etc.

Interesting consequence #1:
Solving all 100 DL problems
isn't much harder than
solving one DL problem.

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

When did this computation
find its *first* n_i?

Interesting consequence #1:
Solving all 100 DL problems
isn't much harder than
solving one DL problem.

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

When did this computation
find its *first* n_i?
Typically $\approx 500002/100$ mults.

Can use random self-reduction
to turn a single target
into multiple targets.
Let ℓ be the order of P.

Given nP:
Choose random $r_1, r_2, \ldots, r_{100}$.
Compute $r_1P + nP$,
$r_2P + nP$, etc.

Solve these 100 DL problems.
Typically $\approx \ell/100$ mults
to find *at least one*
$r_i + n$ mod ℓ,
immediately revealing n.

Also spent some ADDs
to compute each $r_i P$:
$\approx \lg p$ ADDs for each i.

Faster: Choose $r_i = i r_1$
with $r_1 \approx \ell/100$.
Compute $r_1 P$;
$r_1 P + nP$;
$2r_1 P + nP$;
$3r_1 P + nP$; etc.
Just 1 ADD for each new i.

$\approx 100 + \lg \ell + \ell/100$ ADDs
to find n given nP.

Faster: Increase 100 to $\approx \sqrt{\ell}$.
Only $\approx 2\sqrt{\ell}$ ADDs
to solve one DL problem!
"Shanks baby-step-giant-step
discrete-logarithm algorithm."

Example: $p = 1000003, \ell = 500002$, $P = (101384, 614510)$,
$Q = nP = (670366, 740819)$.
Compute $708P = (393230, 421116)$.
Then compute 707 targets:
$708P + Q = (342867, 153817)$,
$2 \cdot 708P + nP = (430321, 994742)$,
$3 \cdot 708P + nP = (423151, 635197)$,
$\ldots, 706 \cdot 708P + nP = (534170, 450849)$.

Build a sorted table of targets:

$600 \cdot 708P + Q = (799978, 929249)$,

$219 \cdot 708P + Q = (425475, 793466)$,

$679 \cdot 708P + Q = (996985, 191440)$,

$242 \cdot 708P + Q = (262804, 347755)$,

$27 \cdot 708P + Q = (785344, 831127)$,

...

$317 \cdot 708P + Q = (599785, 189116)$.

Look up P, $2P$, $3P$, etc. in table.

$620P = (950652, 688508)$; find $596 \cdot 708P + Q = (950652, 688508)$ in the table of targets;

so $620 = 596 \cdot 708 + n \bmod 500002$;

deduce $n = 78654$.

Factors of the group order

P has order $2 \cdot 53^2 \cdot 89$.

Given $Q = nP$, find $n = \log_P Q$:

$R = (53^2 \cdot 89)P$ has order 2, and
$S = (53^2 \cdot 89)Q$ is multiple of R.
Compute $n_1 = \log_R S \equiv n \bmod 2$.

$R = (2 \cdot 53 \cdot 89)P$ has order 53,
and
$S = (2 \cdot 53 \cdot 89)Q$ is multiple of R.
Compute
$n_2 = \log_R S \equiv n \bmod 53$.
This is a DLP in a group
of size 53.

$T = (2 \cdot 89)(Q - n_2 P)$ is also
a multiple of R, i.e., has order 53.
Compute
$n_3 = \log_R T \equiv n \bmod 53$.
Now $n_2 + 53n_3 \equiv n \bmod 53^2$.

$R = (2 \cdot 53^2)P$ has order 89, and
$S = (2 \cdot 53^2)Q$ is multiple of R.
Compute
$n_4 = \log_R S \equiv n \bmod 89$.

Use Chinese Remainder Theorem
$n \equiv n_1 \bmod 2$,
$n \equiv n_2 + 53n_3 \bmod 53^2$,
$n \equiv n_4 \bmod 89$,
to determine n modulo $2 \cdot 53^2 \cdot 89$.

This "Pohlig-Hellman method" converts an order-ab DL into an order-a DL, an order-b DL, and a few scalar multiplications.

Here $(53^2 \cdot 89)P = (1, 0)$ and $(53^2 \cdot 89)Q = \infty$, thus $n_1 = 0$.

$(2 \cdot 53 \cdot 89)P = (539296, 488875)$,
$(2 \cdot 53 \cdot 89)Q = (782288, 572333)$.
A search quickly finds $n_2 = 2$.
$(2 \cdot 89)(Q - 2P) = \infty$, thus $n_3 = 0$ and $n_2 + 53n_3 = 2$.

$(2 \cdot 53^2)P = (877560, 947848)$ and $(2 \cdot 53^2)Q = (822491, 118220)$.
Compute $n_4 = 67$,
e.g. using BSGS.

Use Chinese Remainder Theorem
$n \equiv 0 \bmod 2$,
$n \equiv 2 \bmod 53^2$,
$n \equiv 67 \bmod 89$,
to determine $n = 78654$.

Pohlig-Hellman method reduces security of discrete logarithm problem in group generated by P to security of largest prime order subgroup.

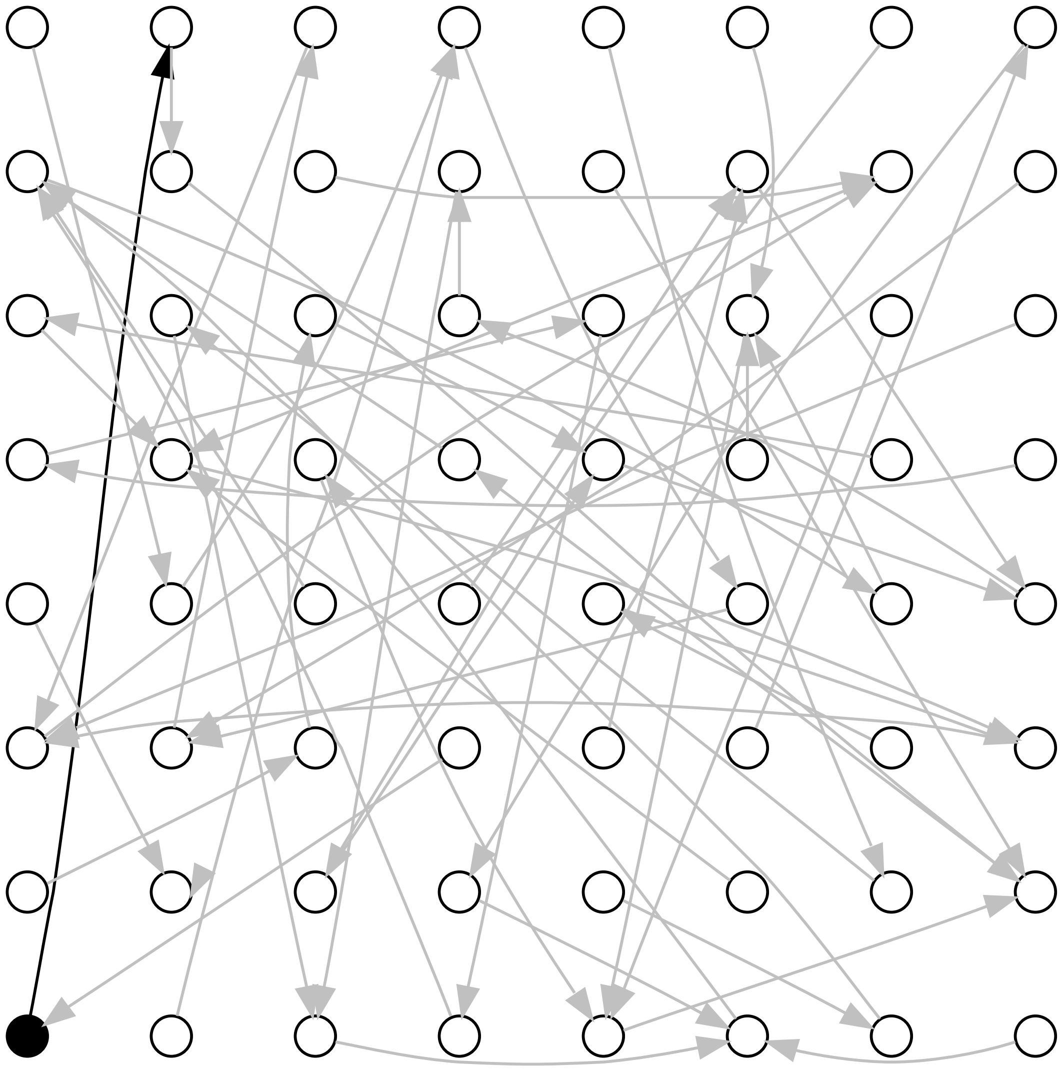
The rho method

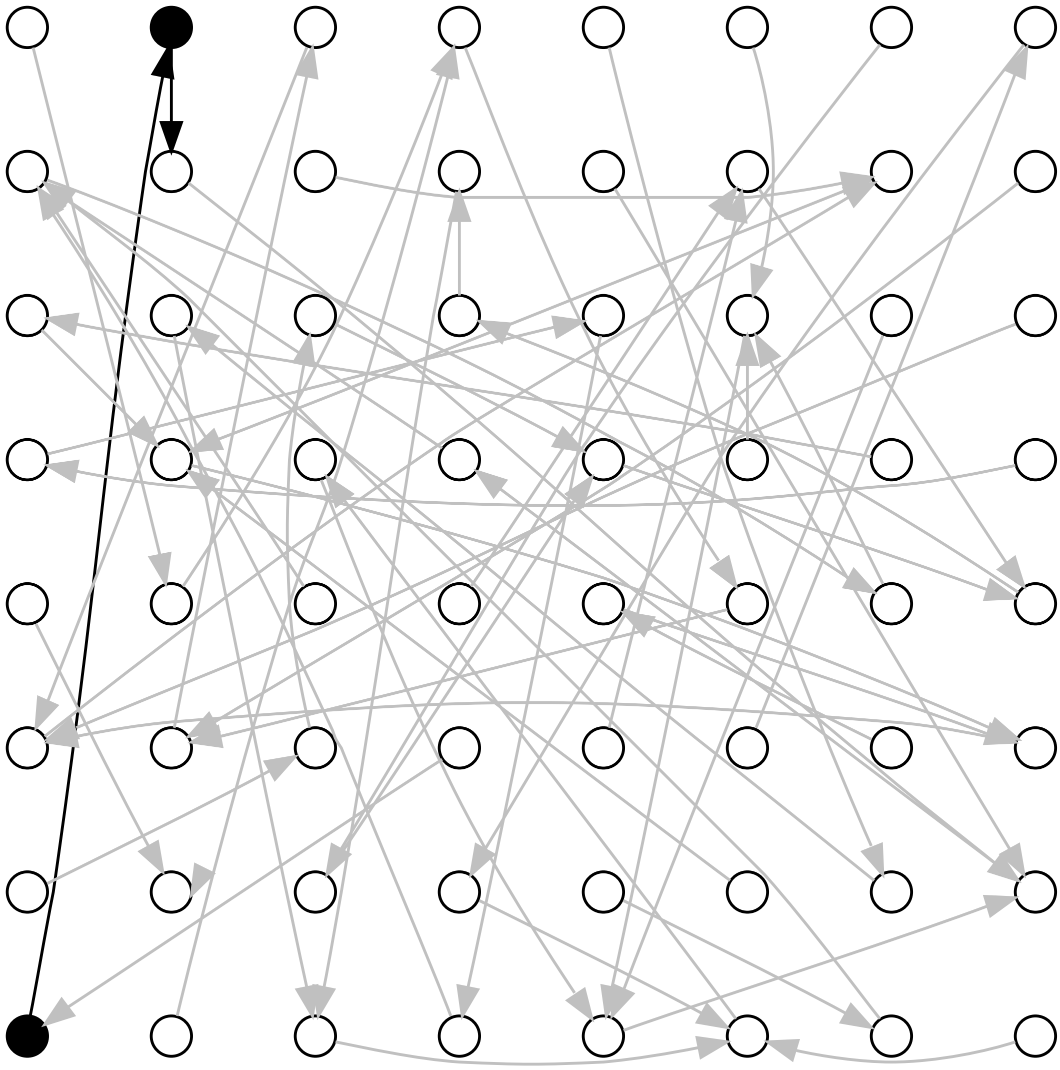
Simplified, non-parallel rho:

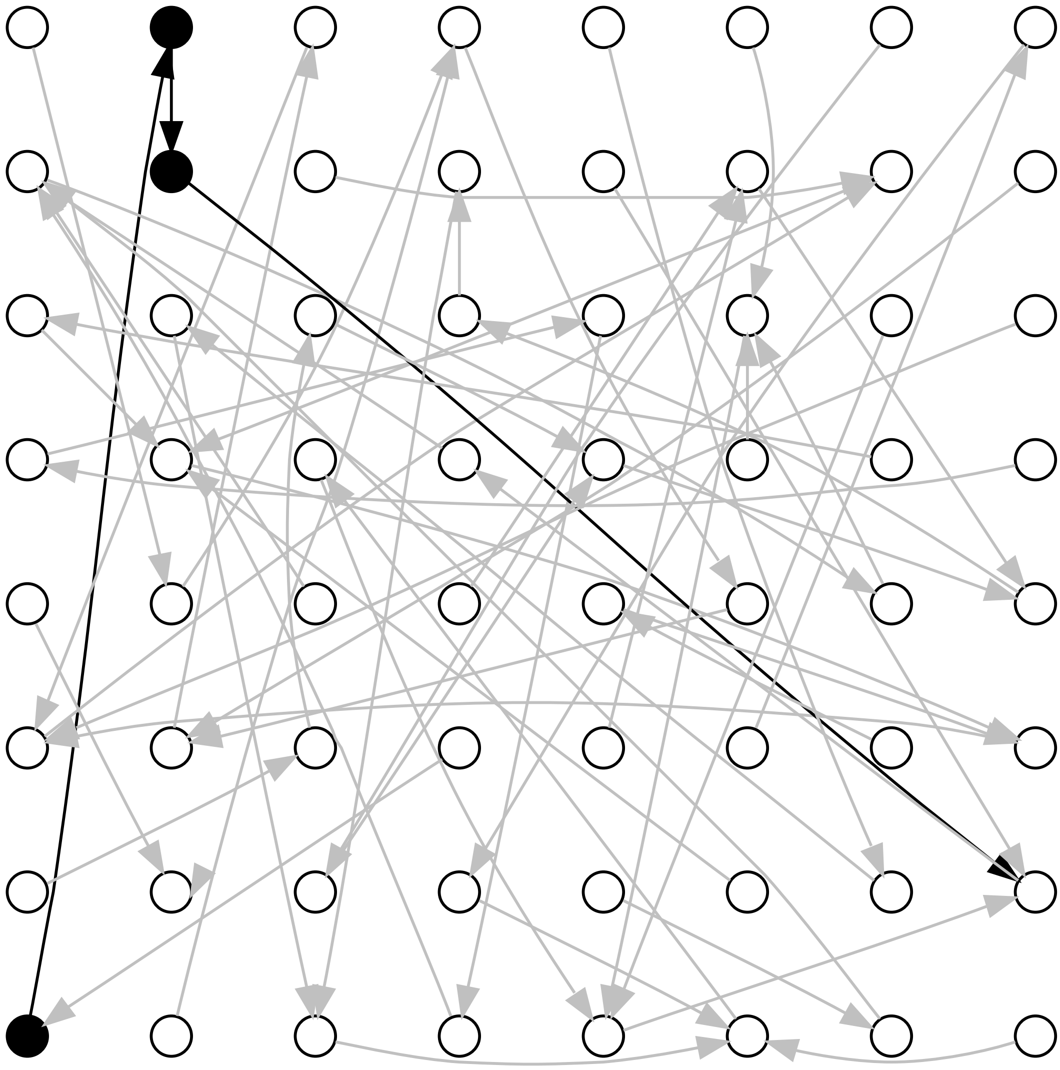
Make a pseudo-random walk
in the group $\langle P \rangle$,
where the next step depends
on current point: $W_{i+1} = f(W_i)$.

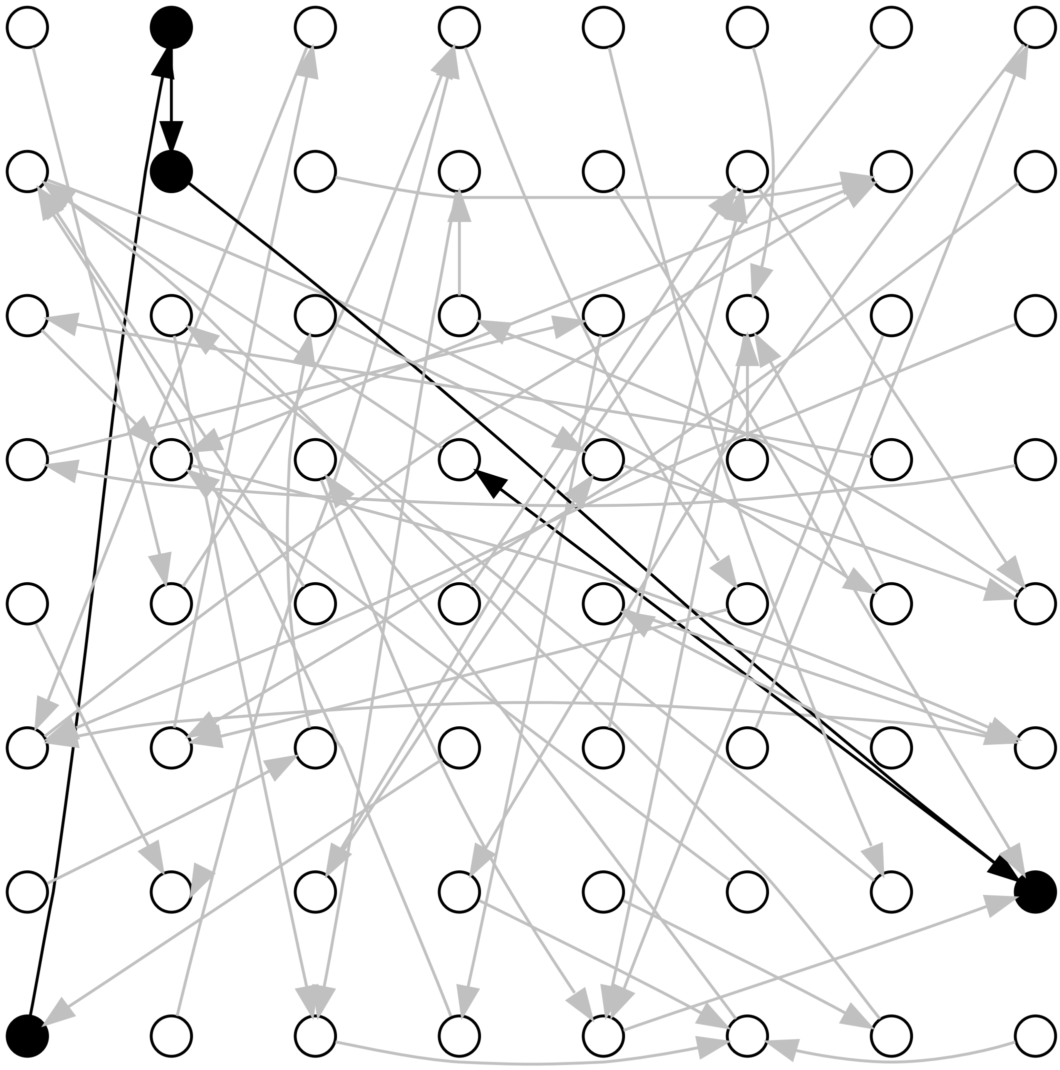
Birthday paradox:
Randomly choosing from ℓ
elements picks one element twice
after about $\sqrt{\pi\ell/2}$ draws.

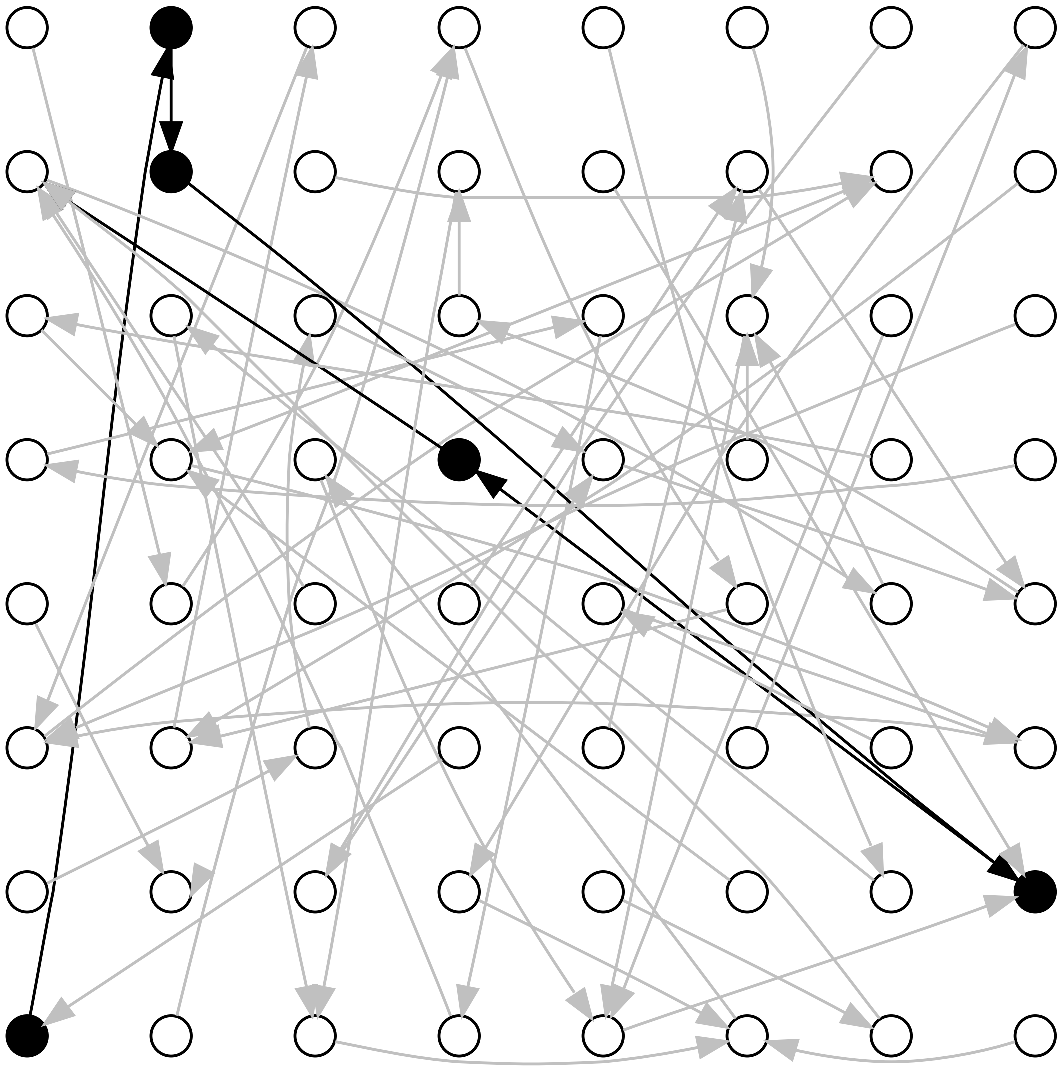
The walk now enters a cycle.
Cycle-finding algorithm
(e.g., Floyd) quickly detects this.

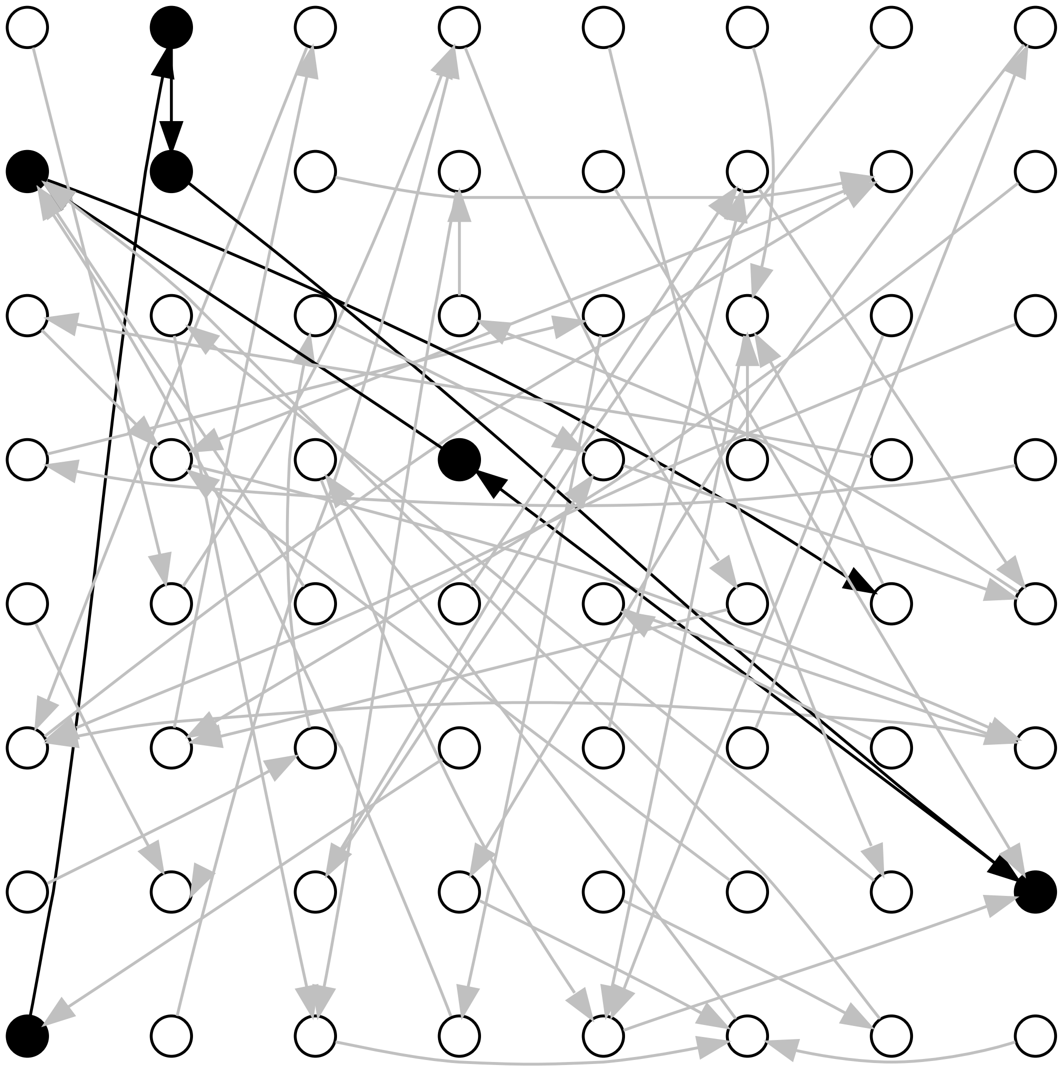


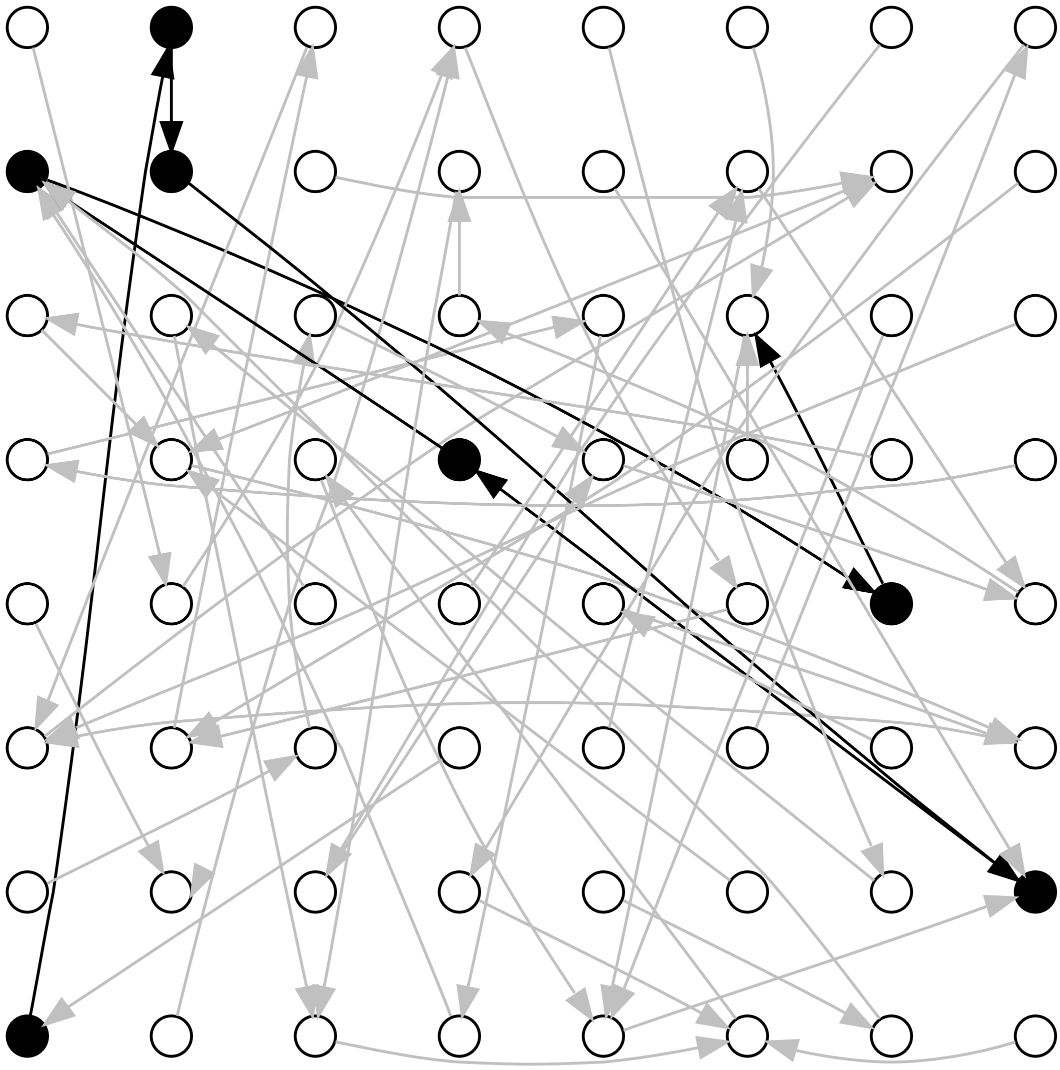


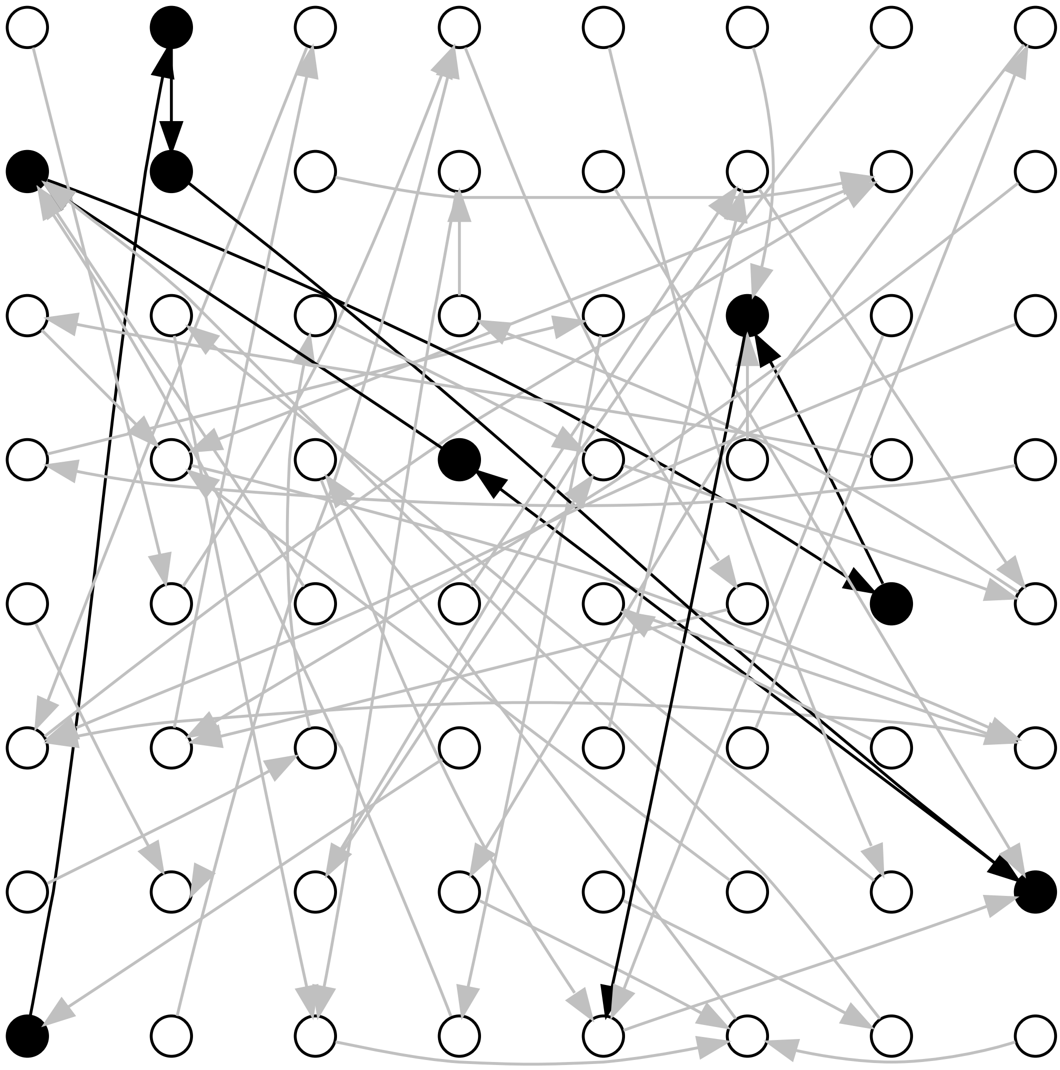


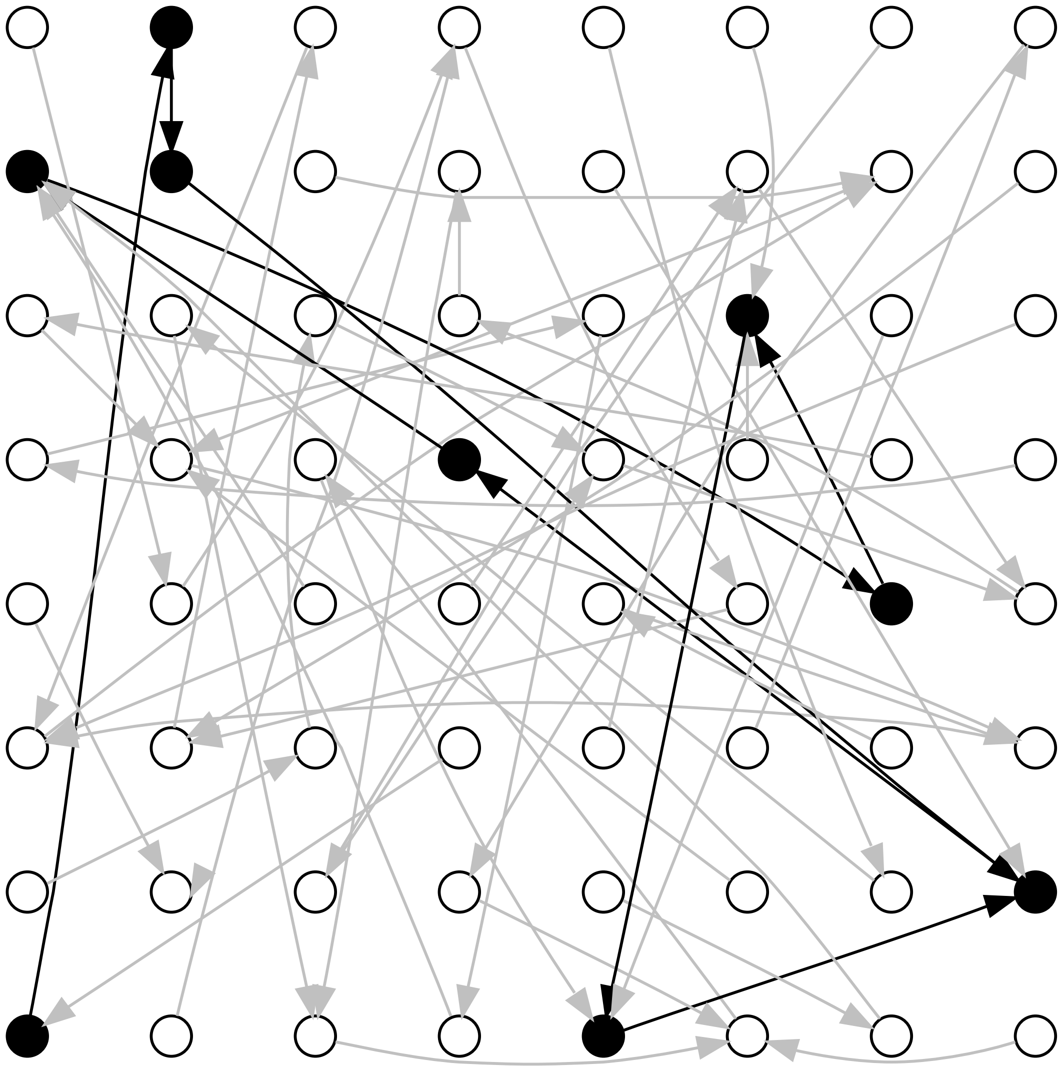


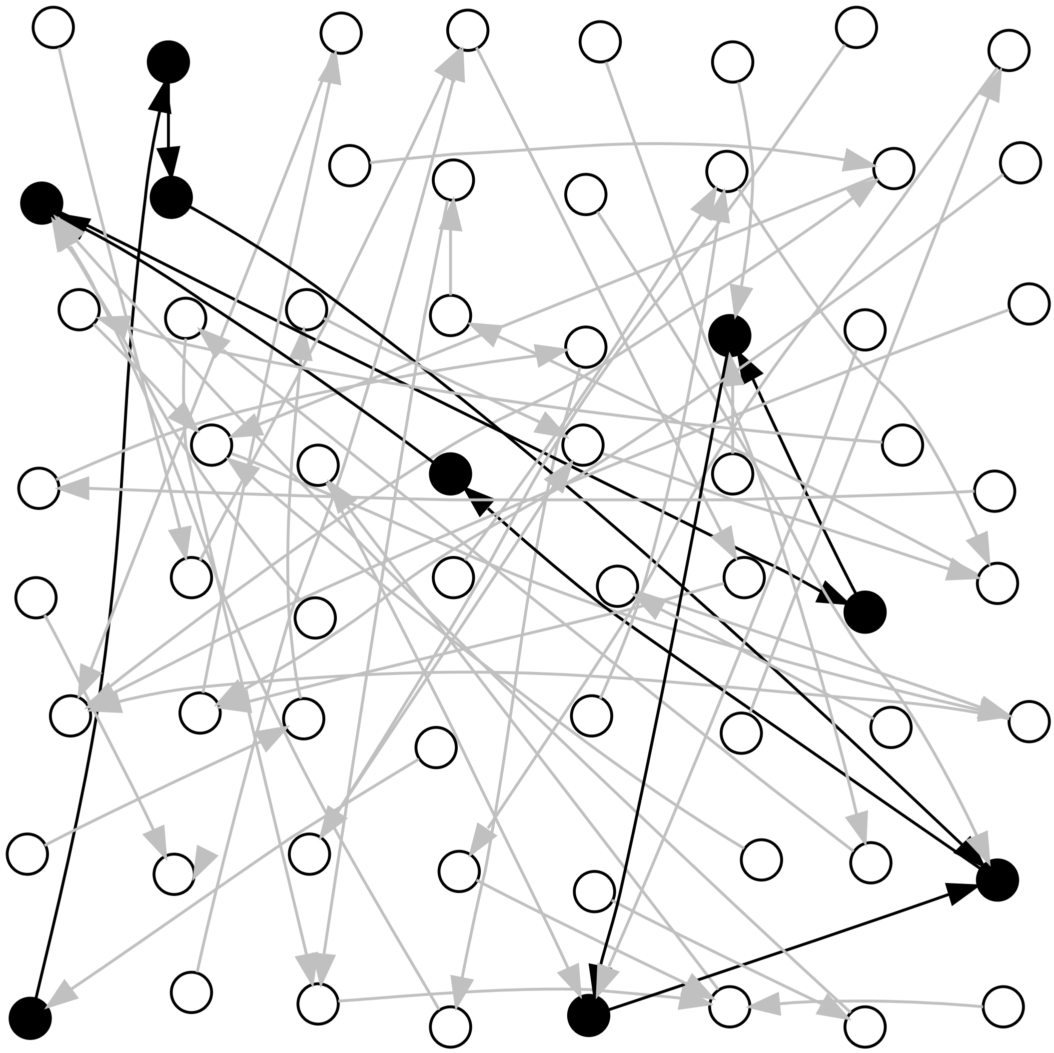


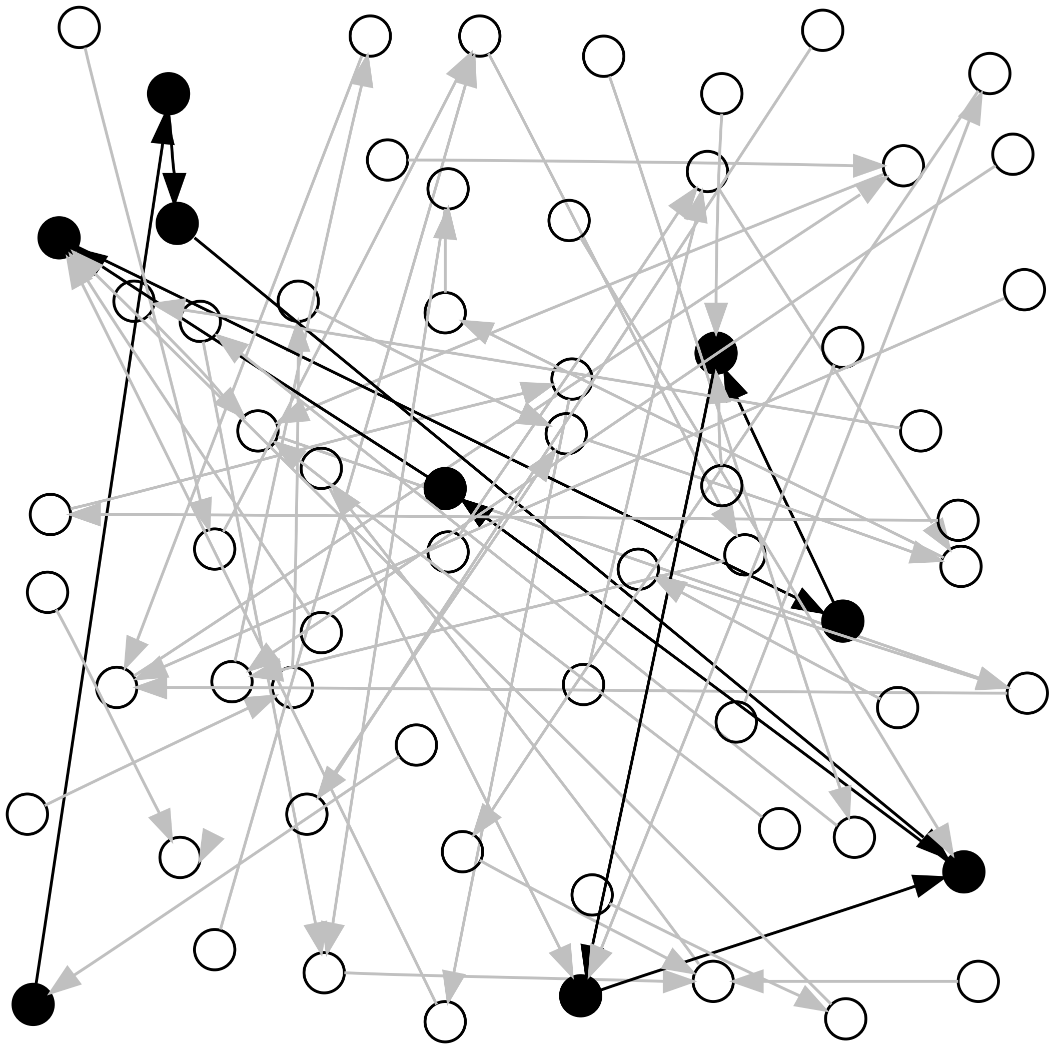


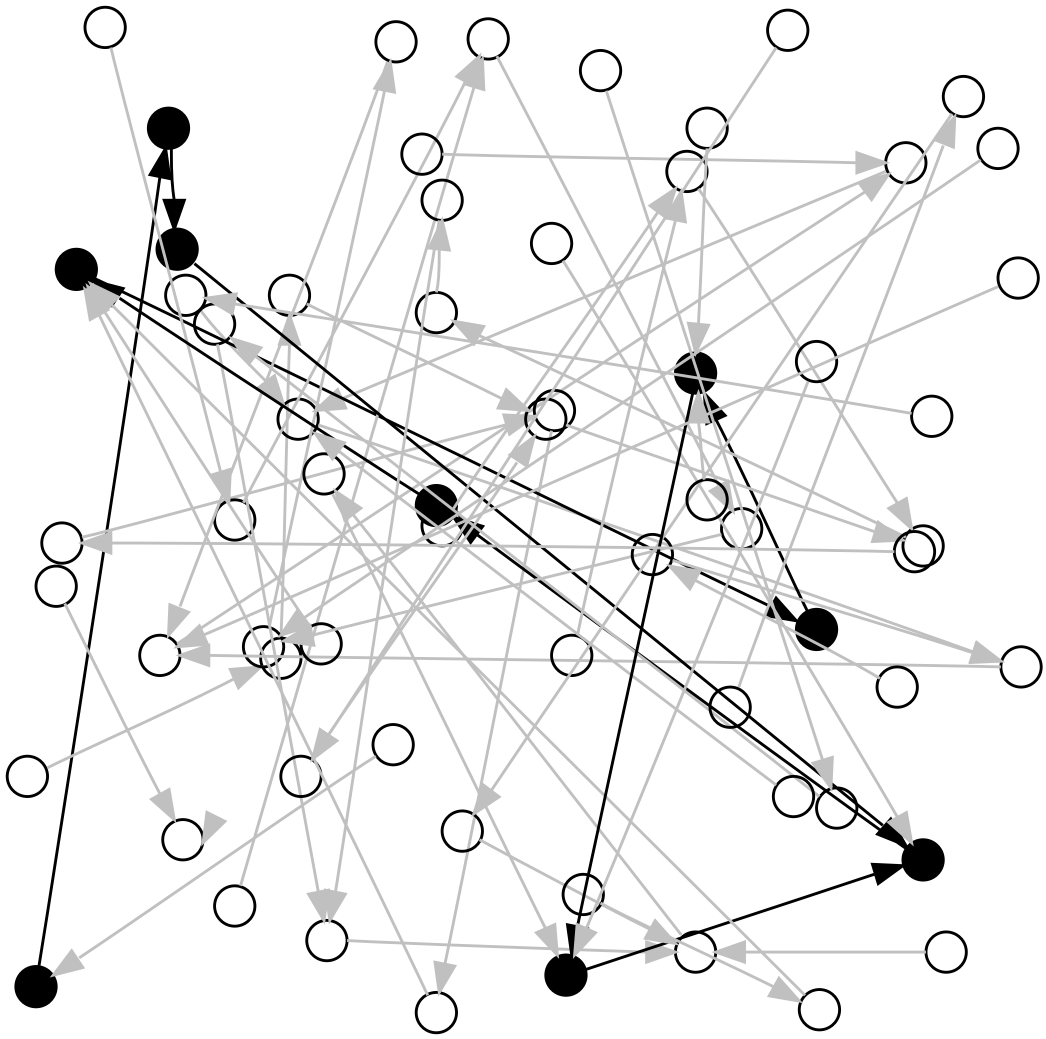


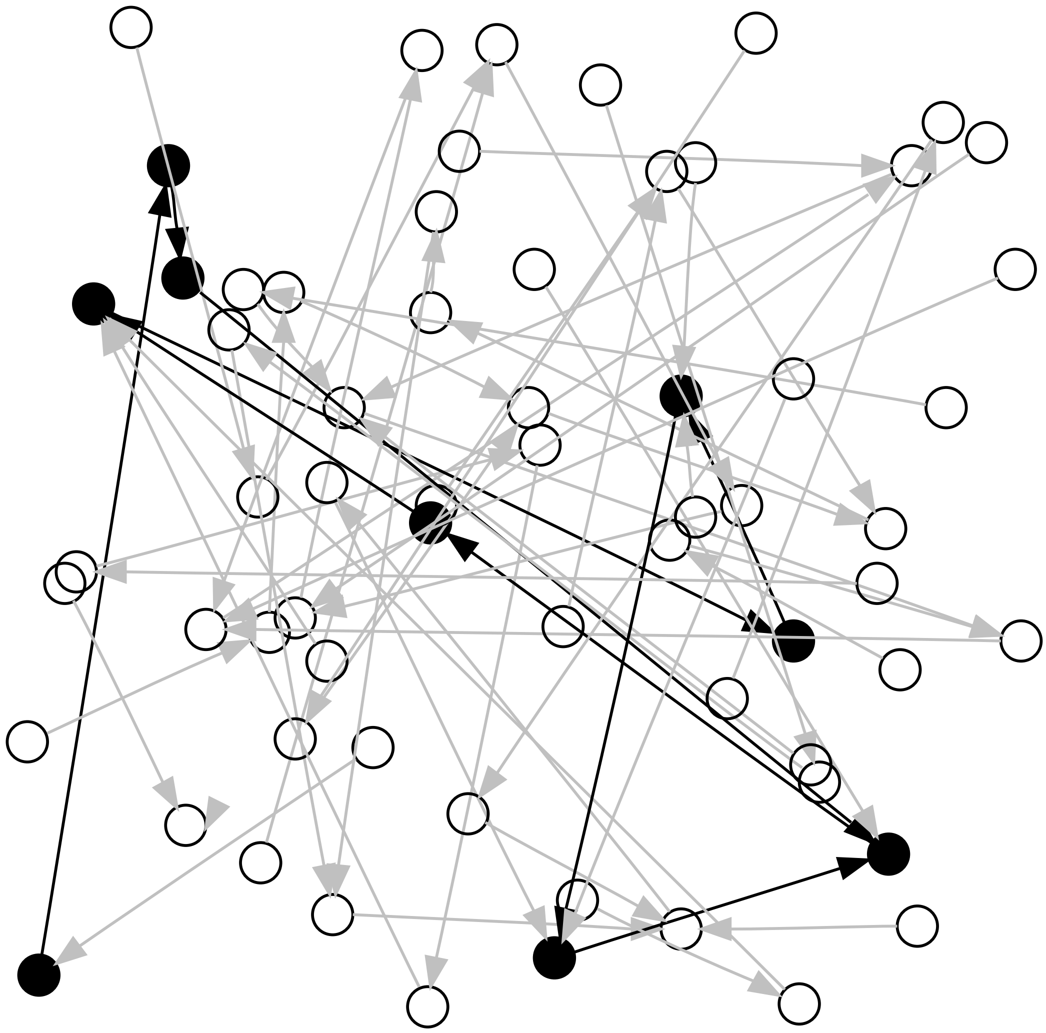


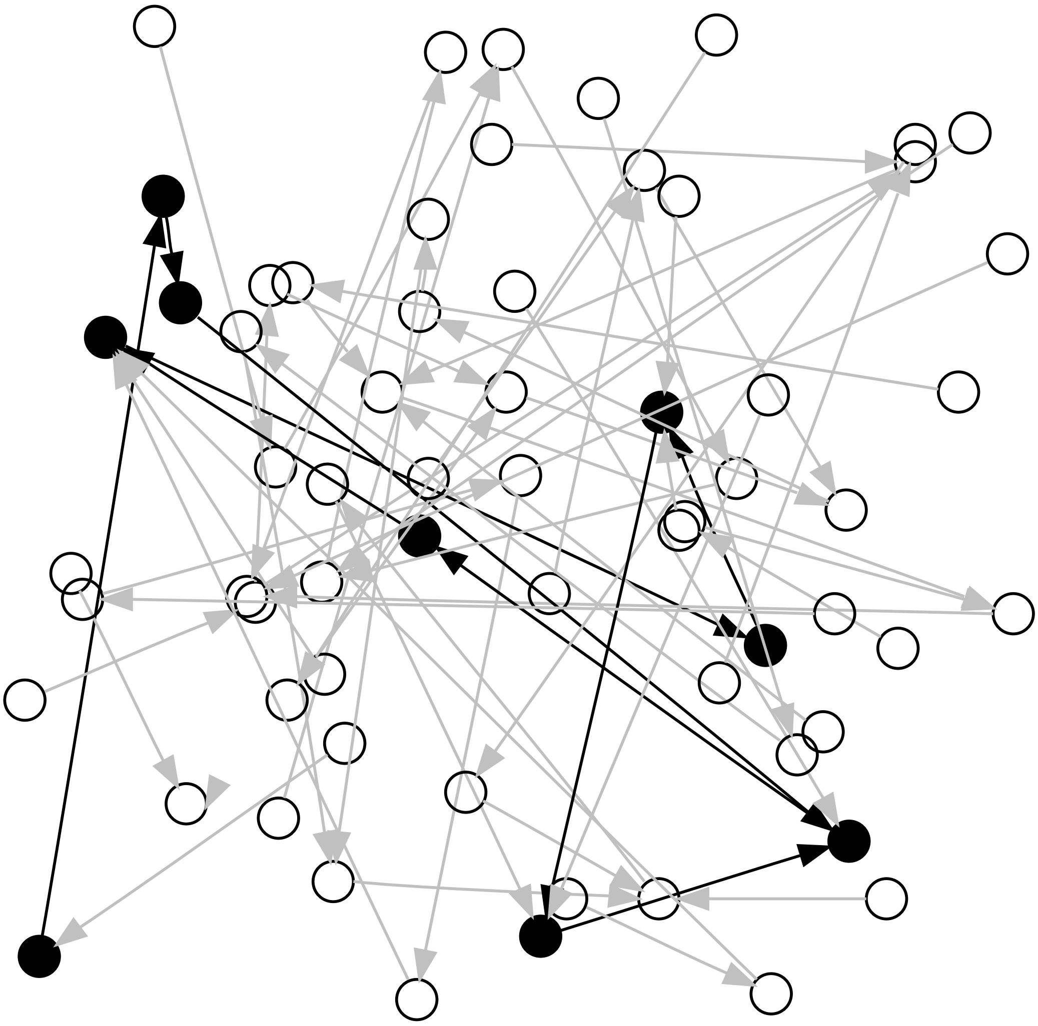


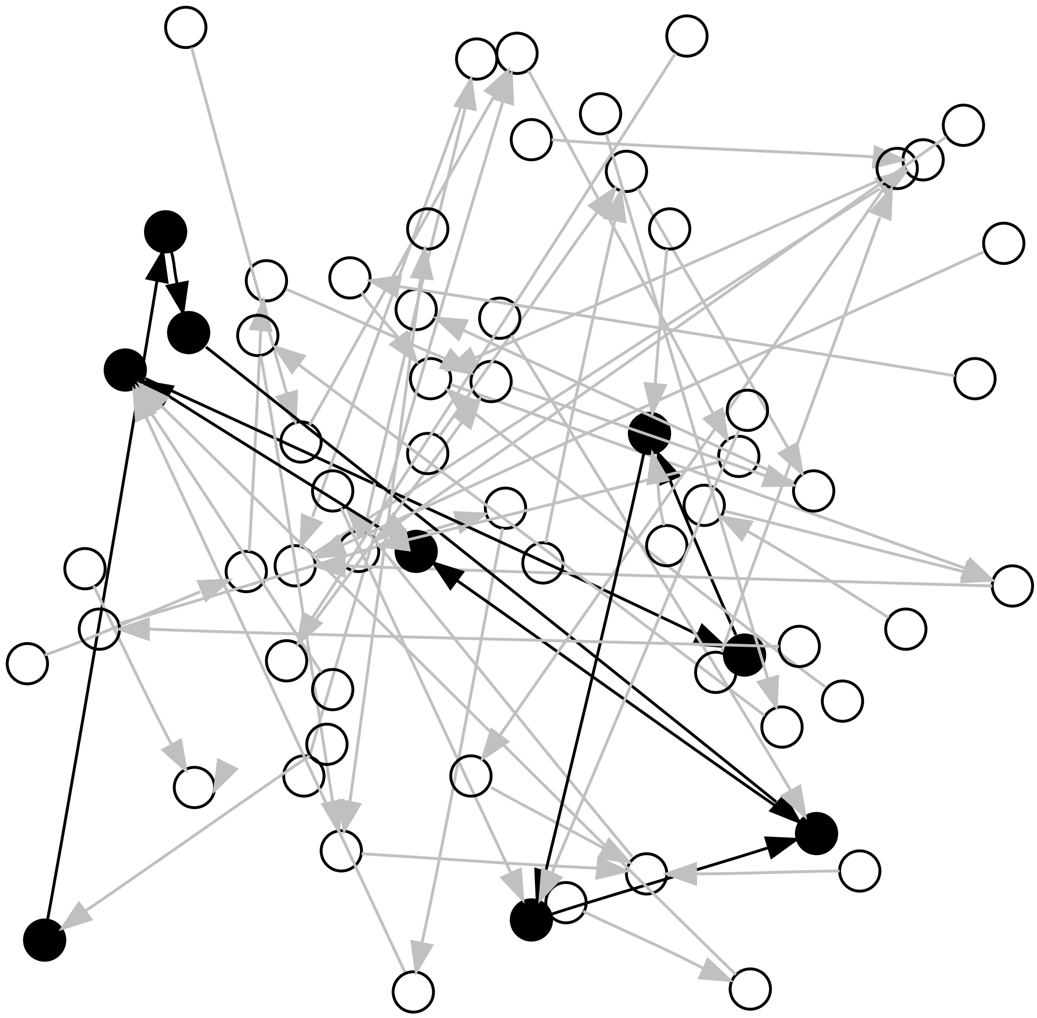


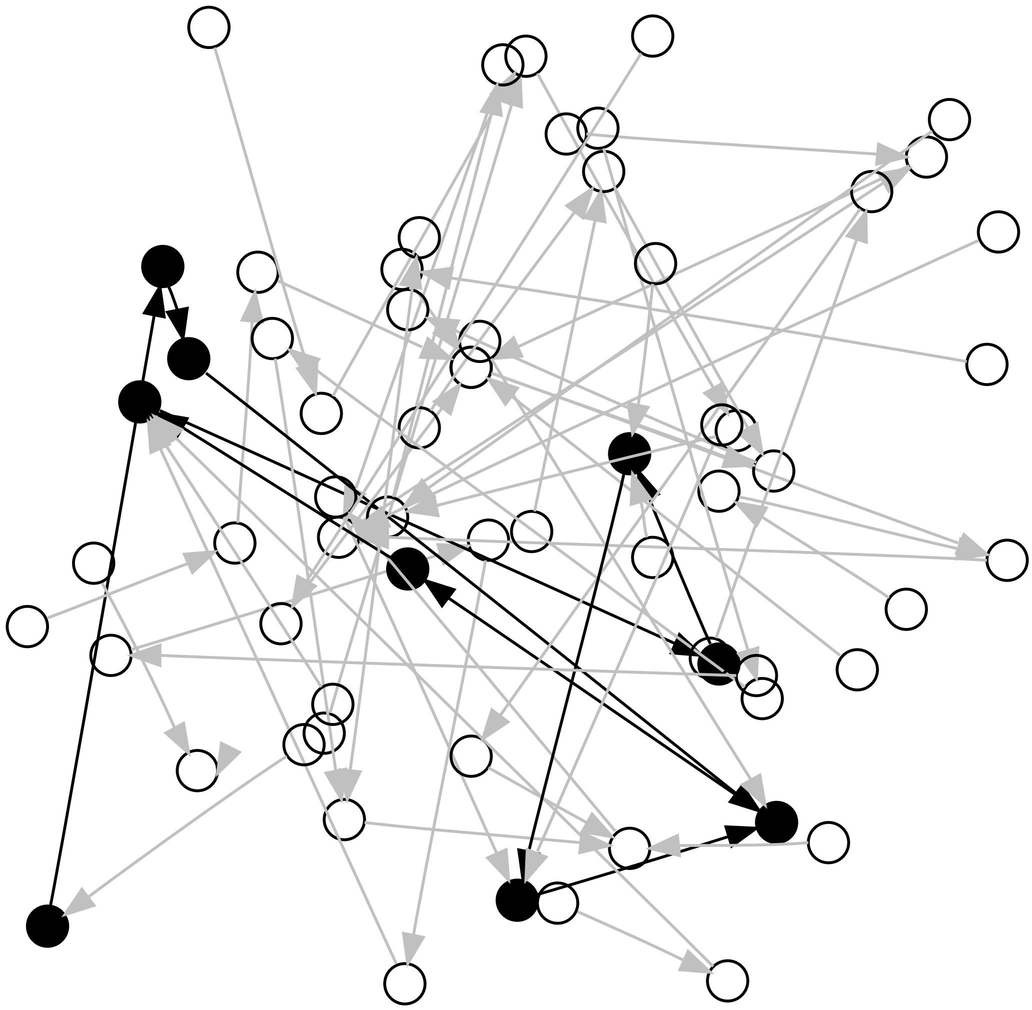


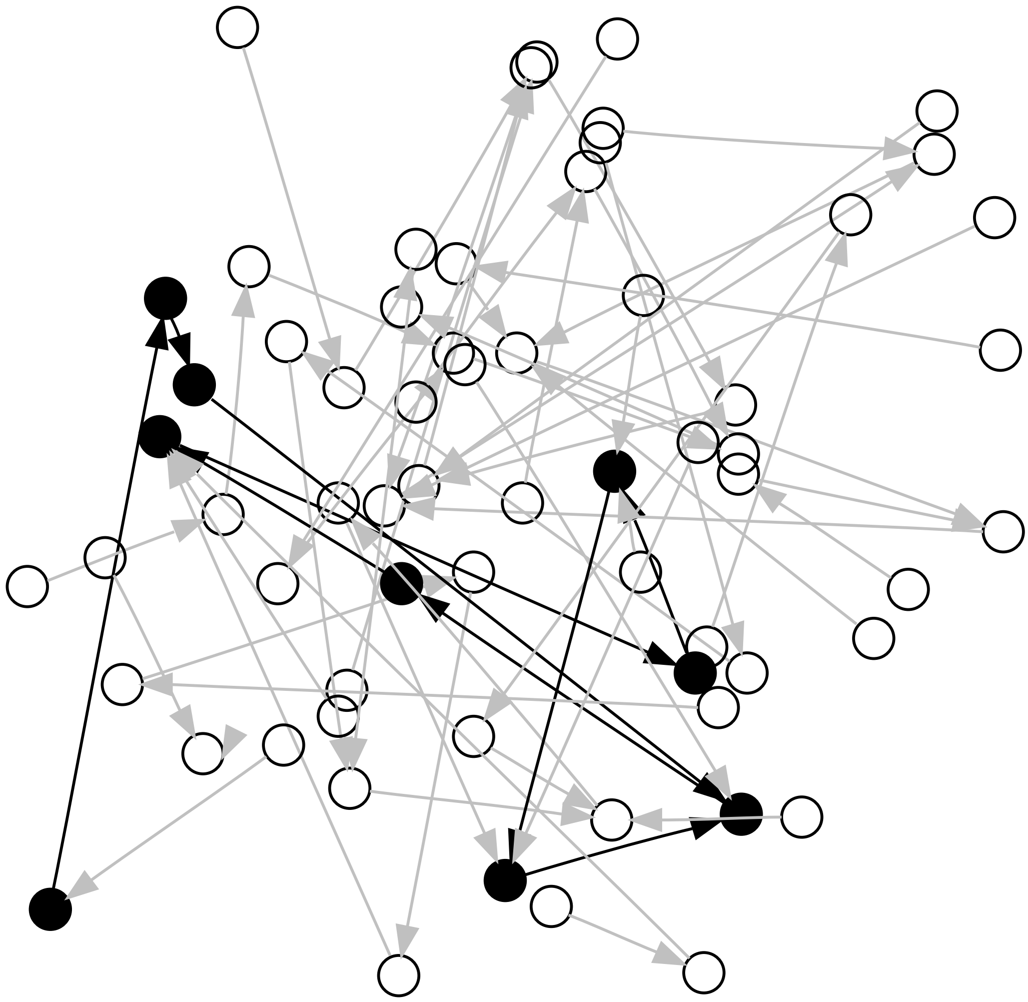


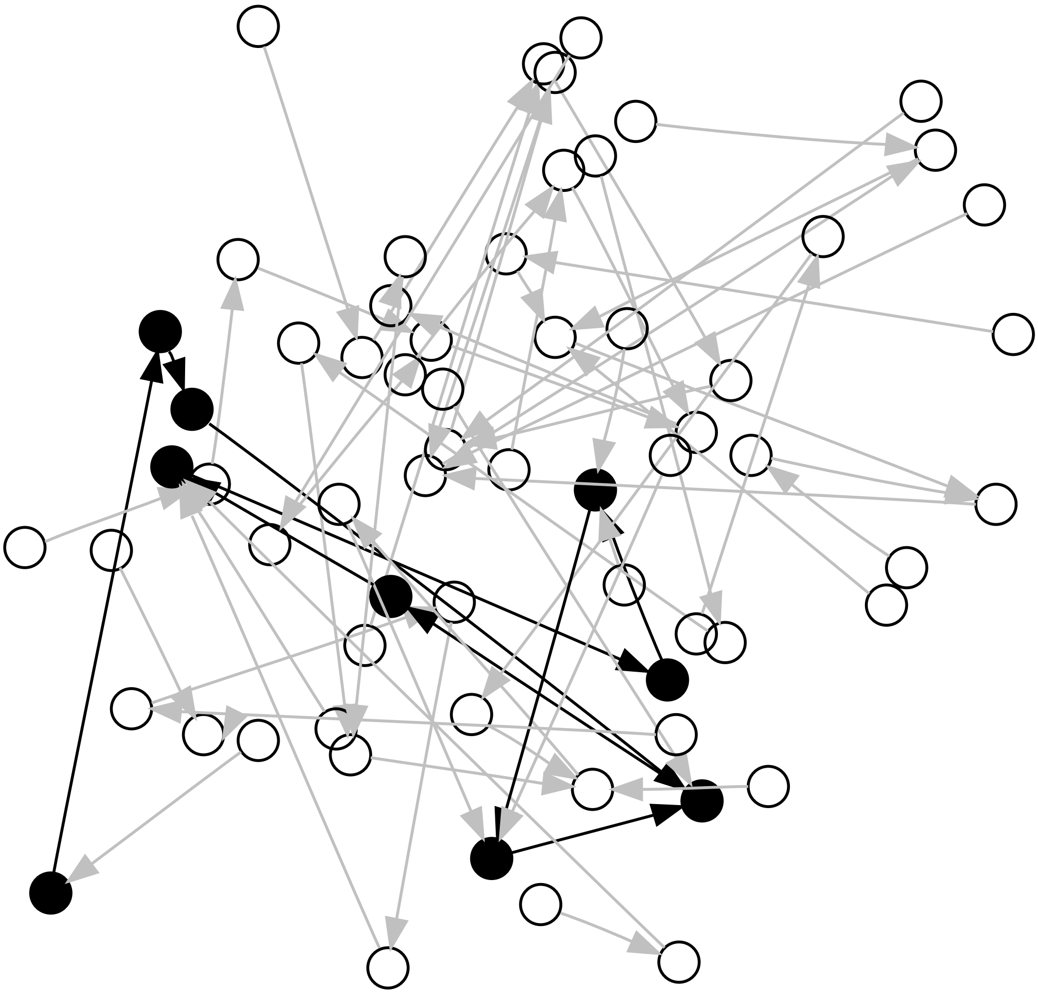


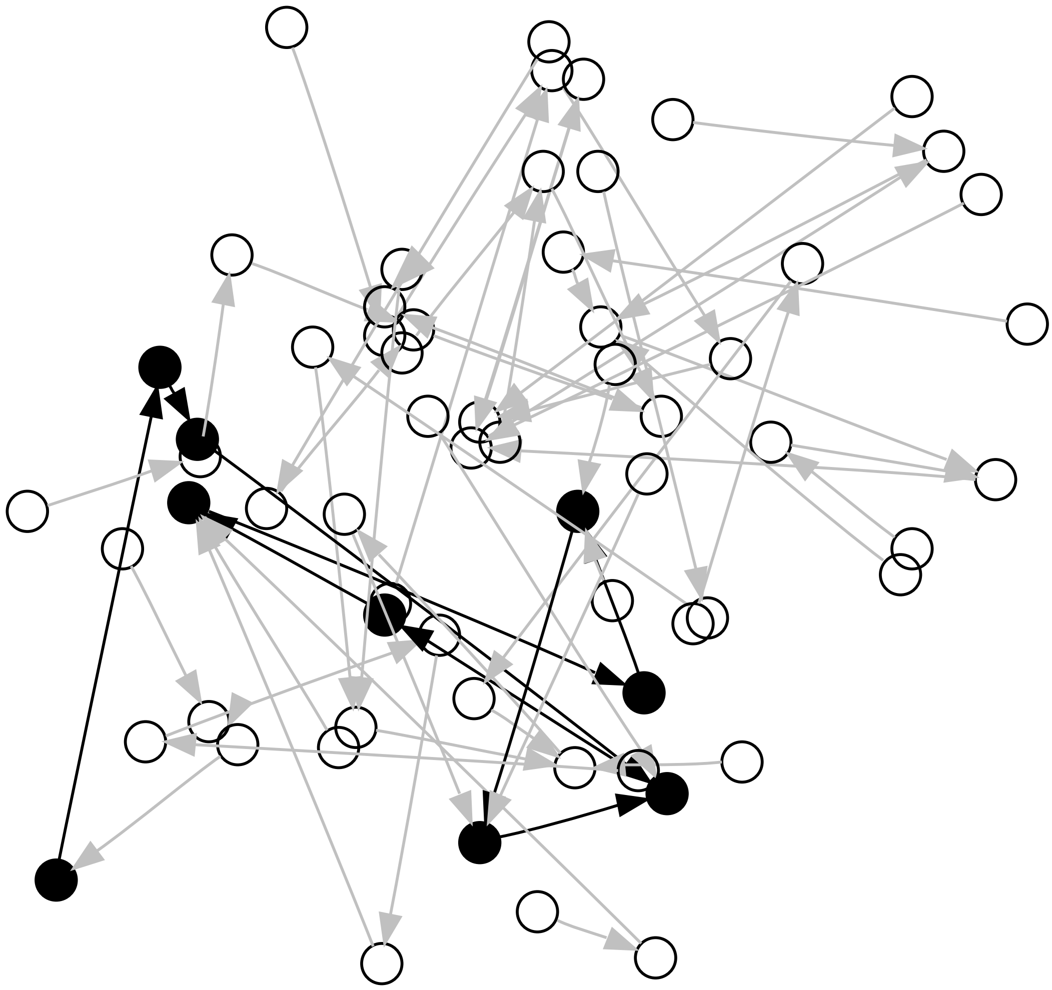


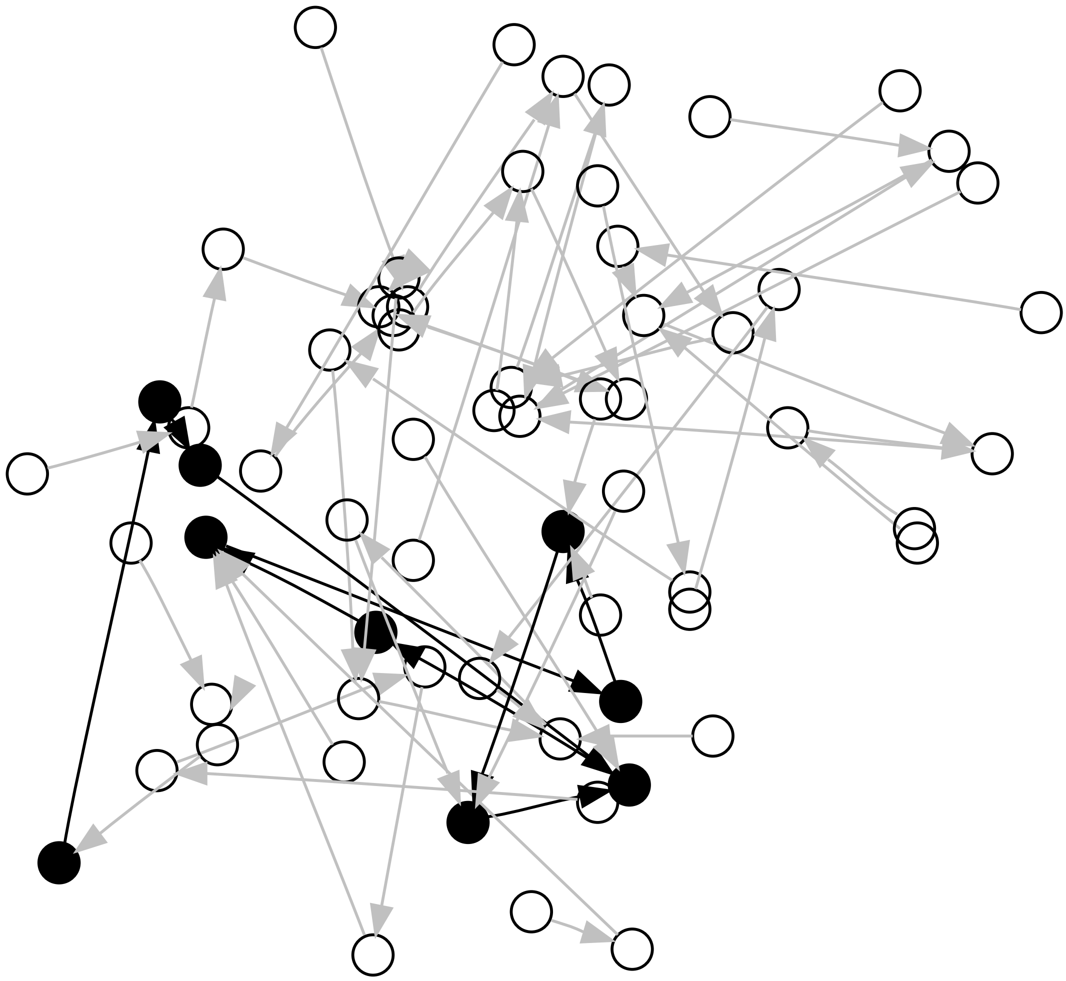


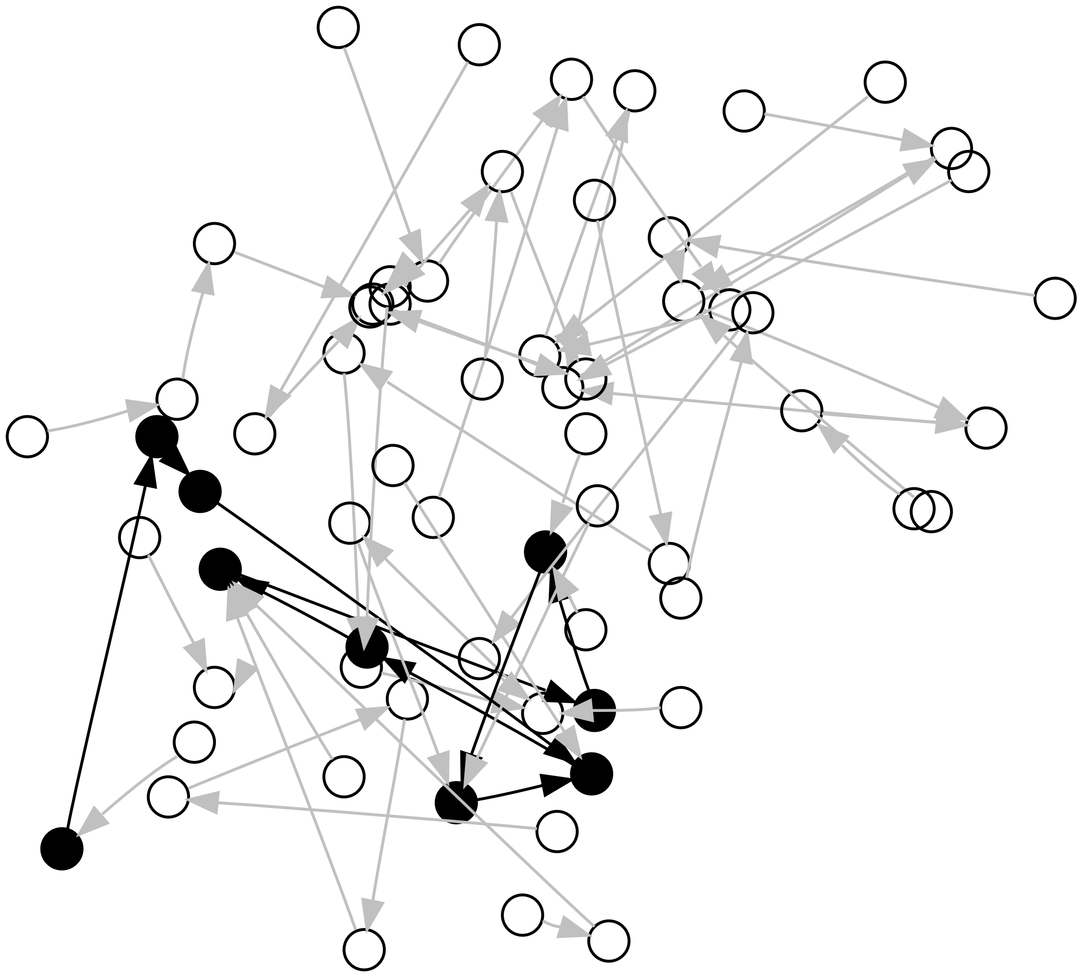


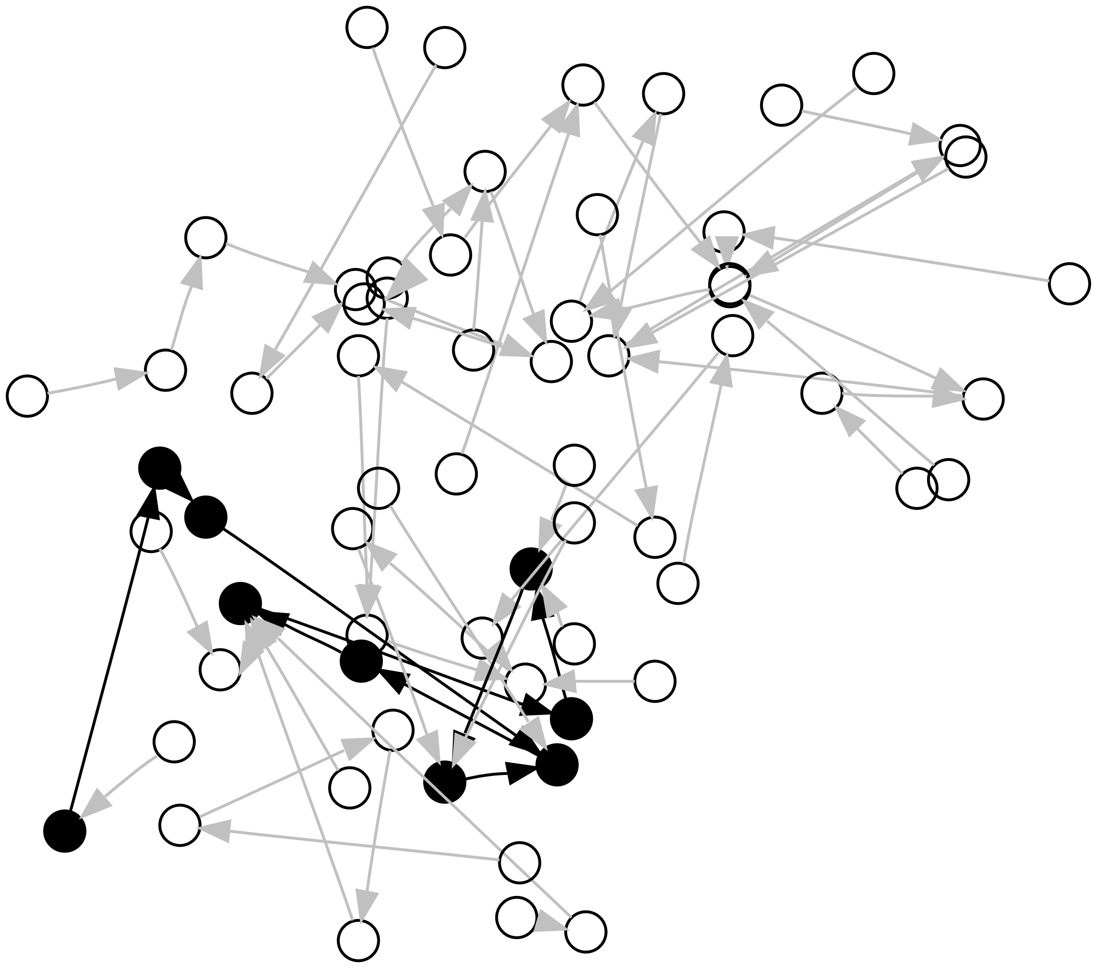


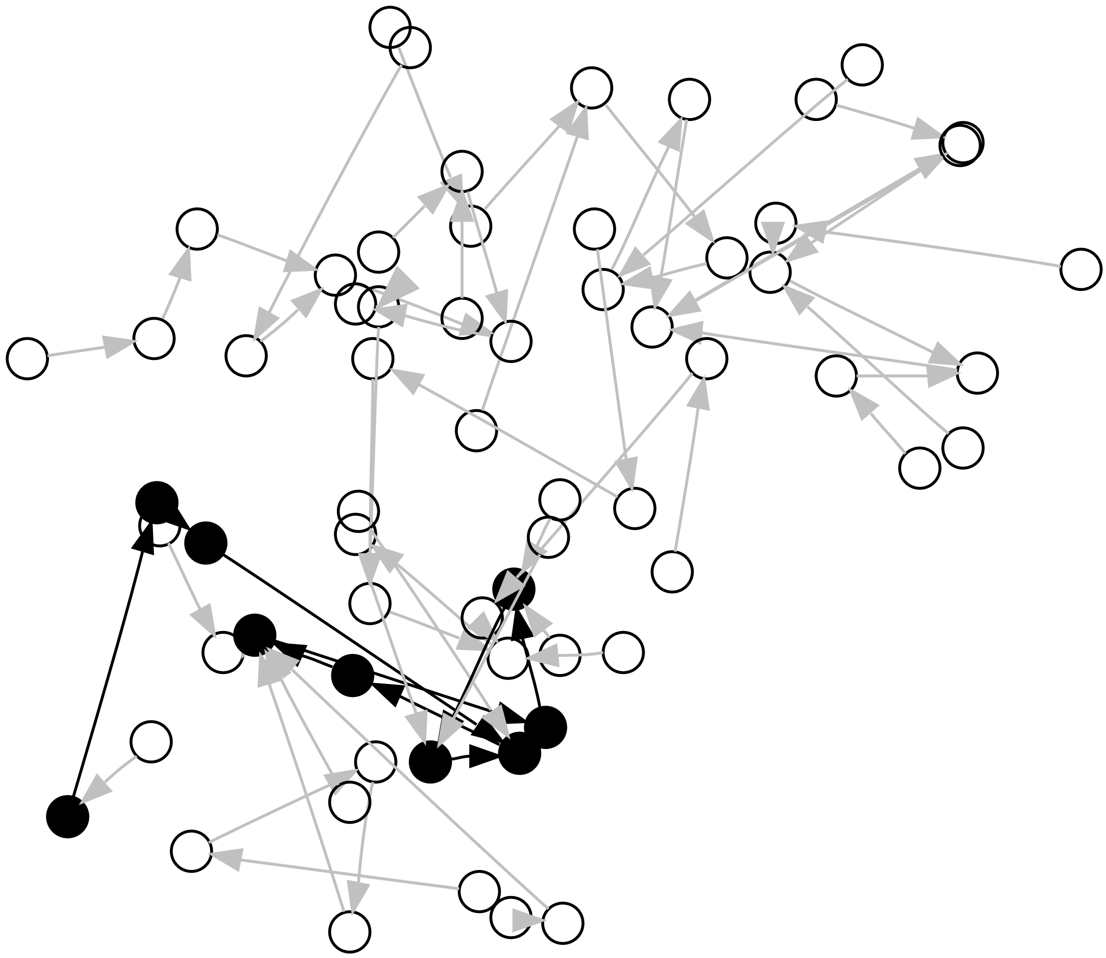


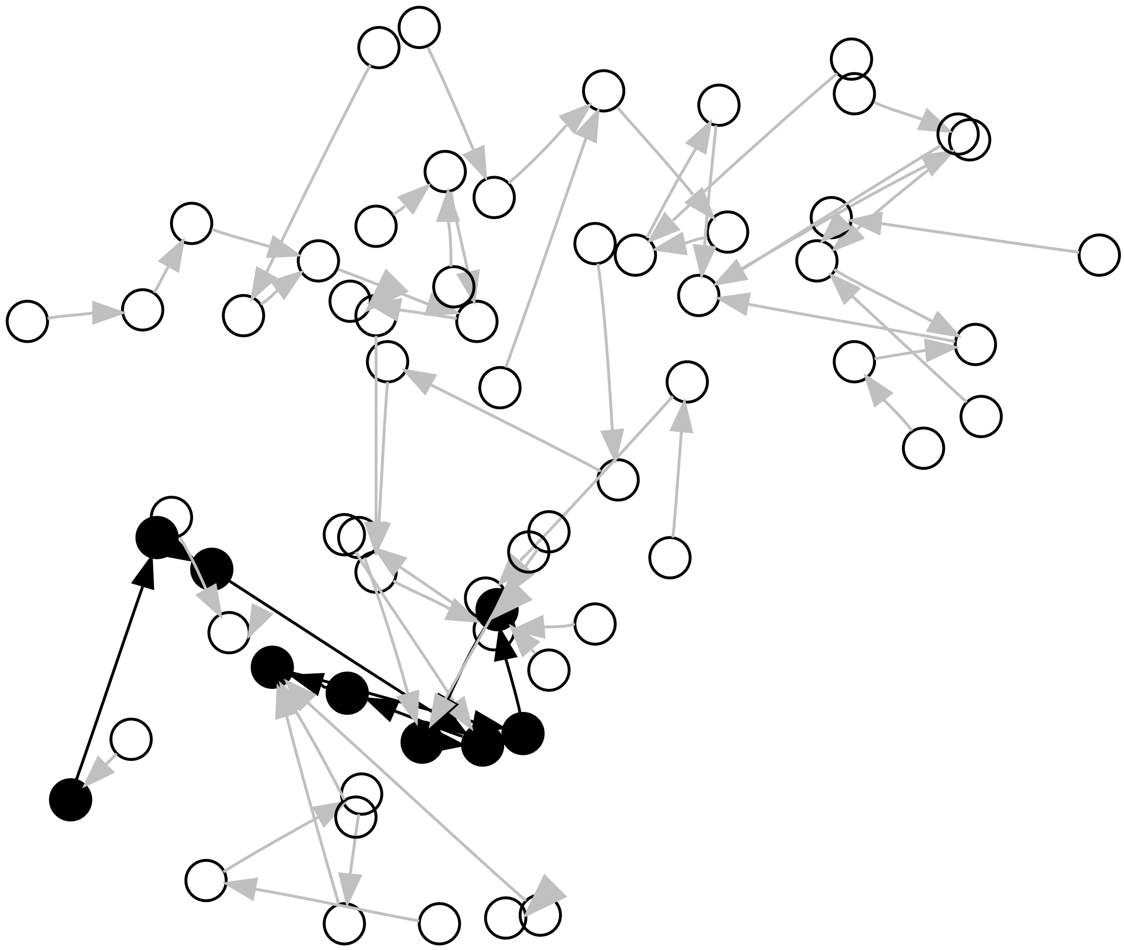


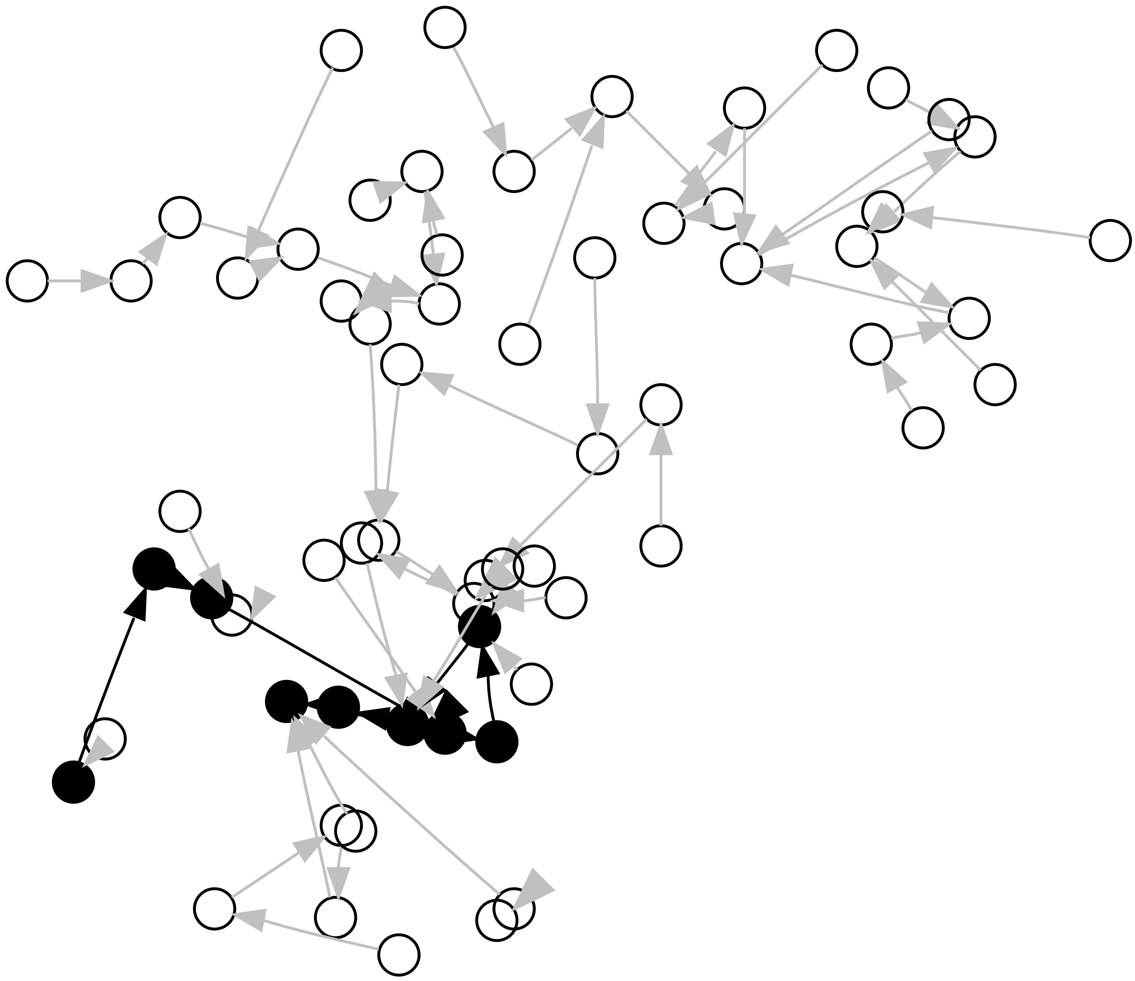


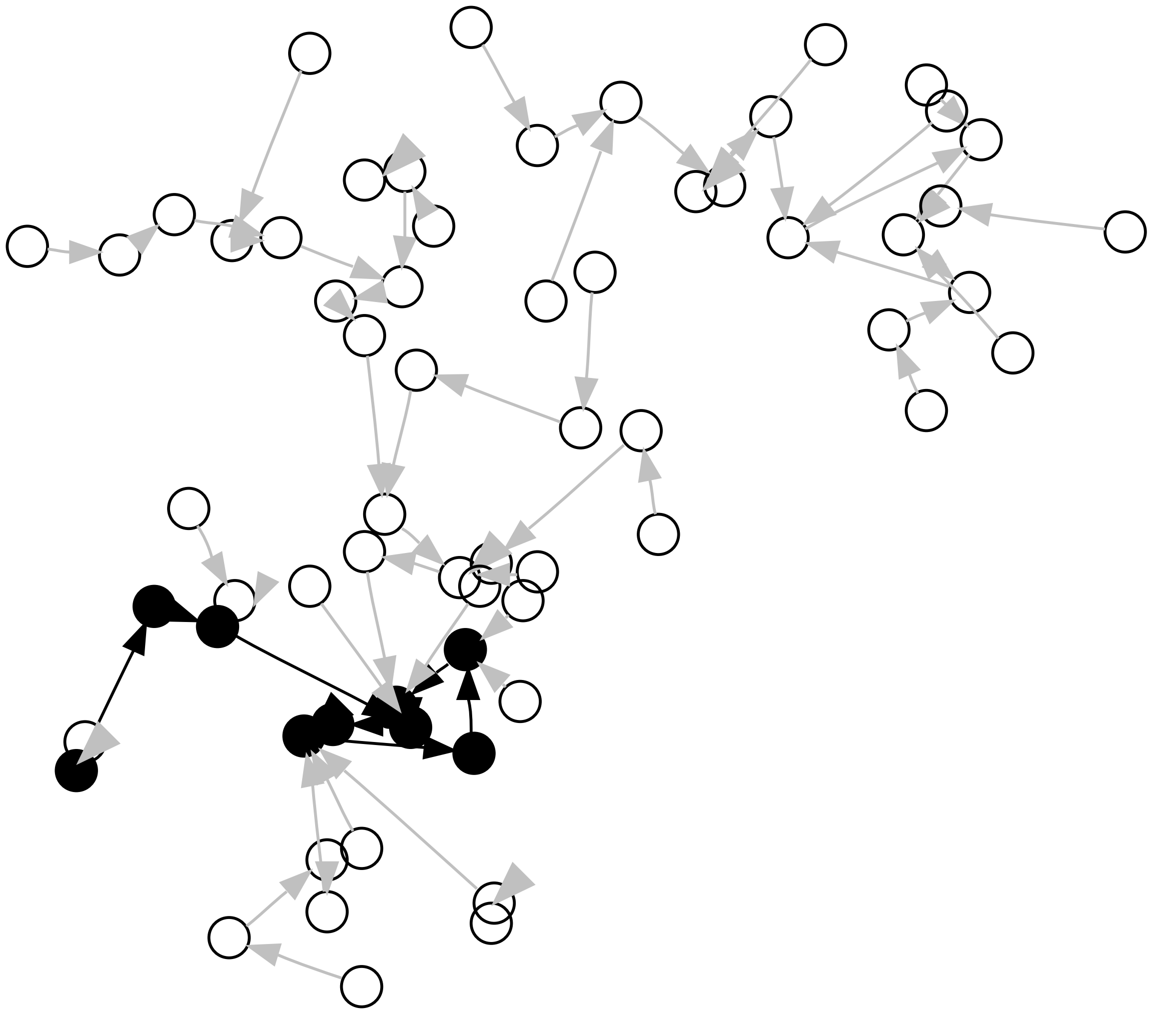


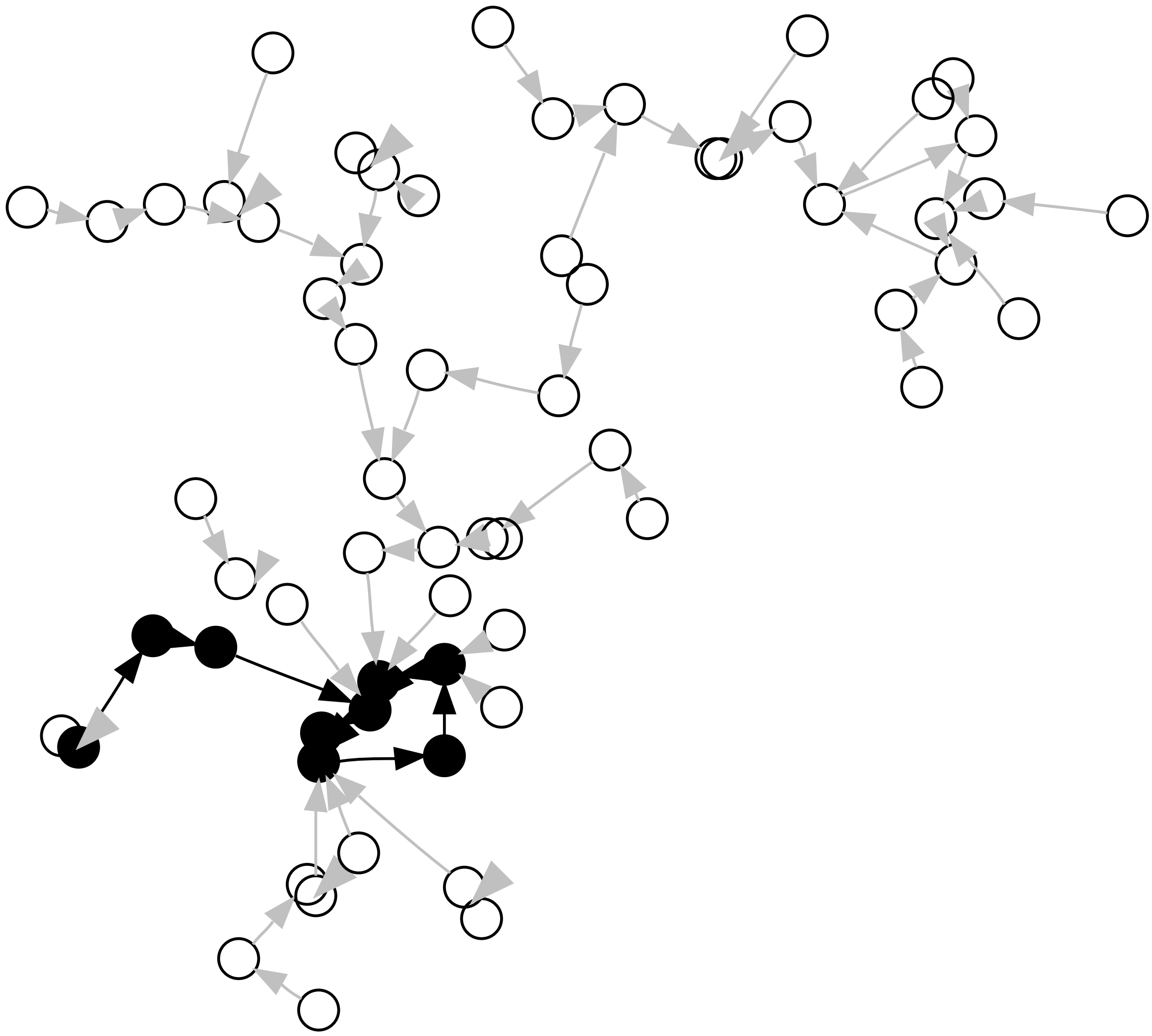


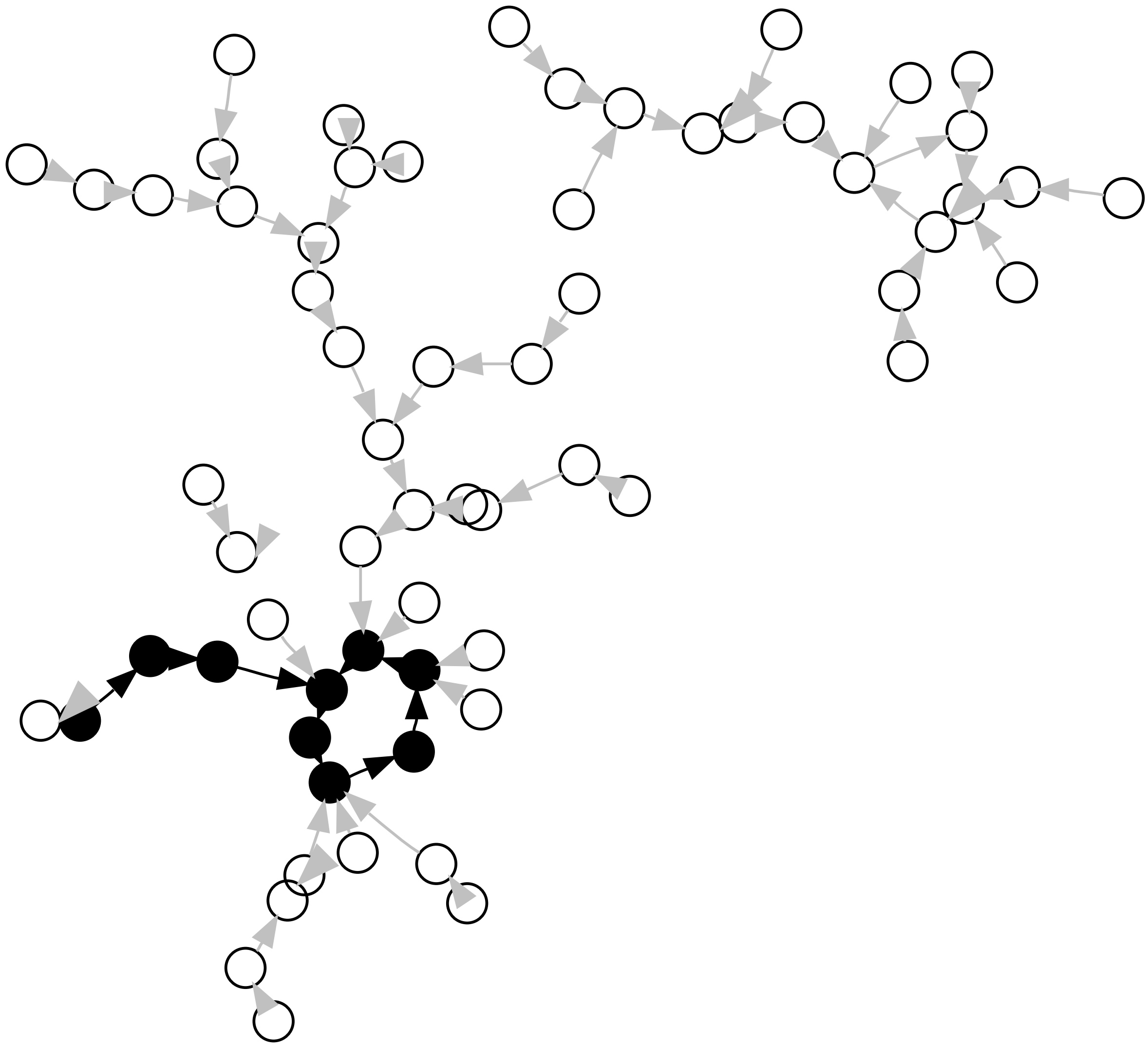


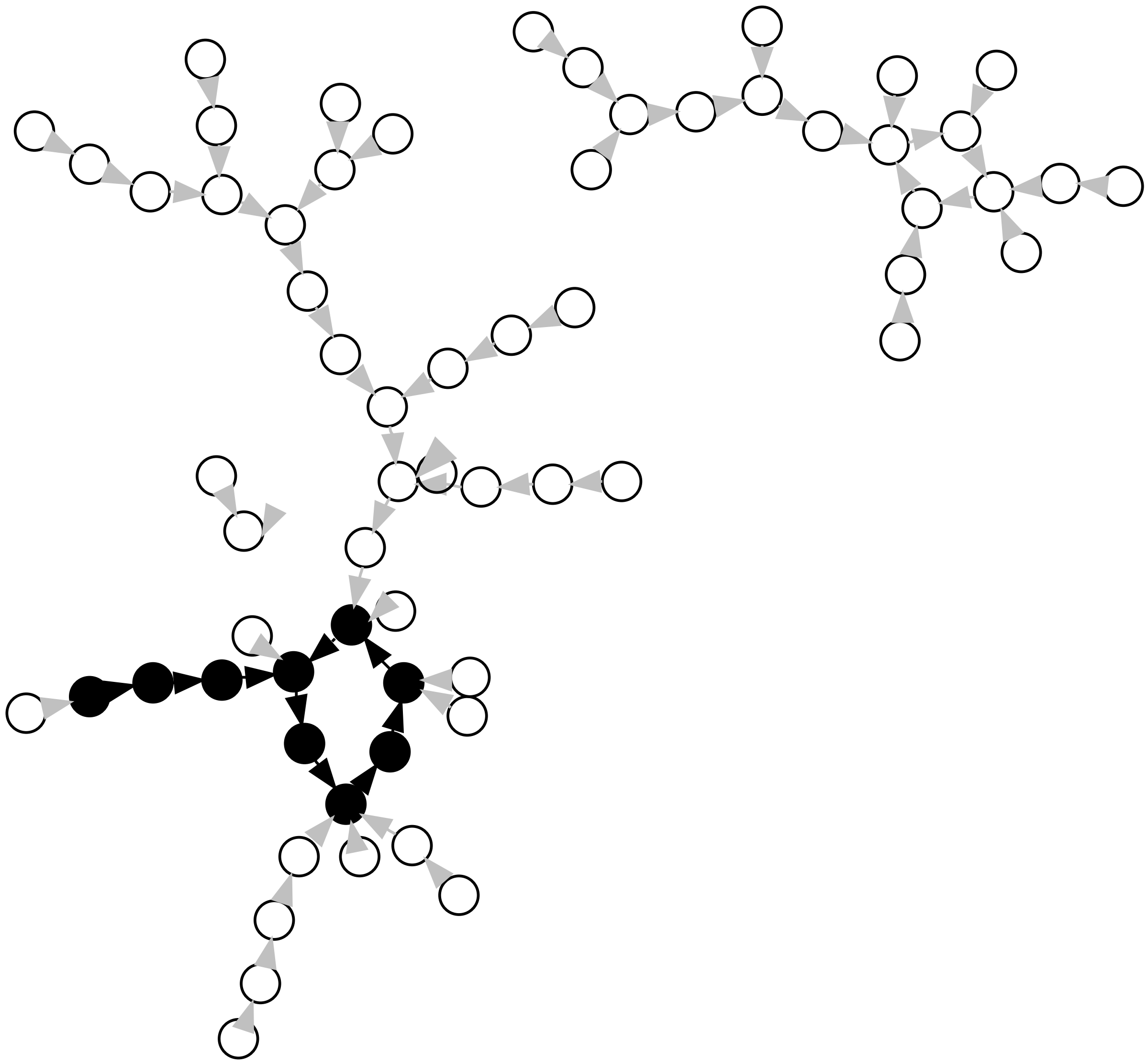












Assume that for each point
we know $a_i, b_i \in \mathbf{Z}/\ell\mathbf{Z}$
so that $W_i = a_i P + b_i Q$.

Then $W_i = W_j$ means that
$a_i P + b_i Q = a_j P + b_j Q$
so $(b_i - b_j)Q = (a_j - a_i)P$.
If $b_i \neq b_j$ the DLP is solved:
$n = (a_j - a_i)/(b_i - b_j)$.

Assume that for each point
we know $a_i, b_i \in \mathbf{Z}/\ell\mathbf{Z}$
so that $W_i = a_i P + b_i Q$.

Then $W_i = W_j$ means that
$a_i P + b_i Q = a_j P + b_j Q$
so $(b_i - b_j)Q = (a_j - a_i)P$.
If $b_i \neq b_j$ the DLP is solved:
$n = (a_j - a_i)/(b_i - b_j)$.

e.g. $f(W_i) = a(W_i)P + b(W_i)Q$,
starting from some initial
combination $W_0 = a_0 P + b_0 Q$.
If any W_i and W_j collide then
$W_{i+1} = W_{j+1}$, $W_{i+2} = W_{j+2}$,
etc.

If functions $a(W)$ and $b(W)$ are random modulo ℓ, iterations perform a random walk in $\langle P \rangle$.
If a and b are chosen such that $f(W_i) = f(-W_i)$ then the walk is defined on *equivalence classes* under $\pm$.
There are only $\lceil \ell/2 \rceil$ different classes. This reduces the average number of iterations by a factor of almost exactly $\sqrt{2}$.

In general, Pollard's rho method can be combined with any easily computed group automorphism of small order.

Parallel collision search

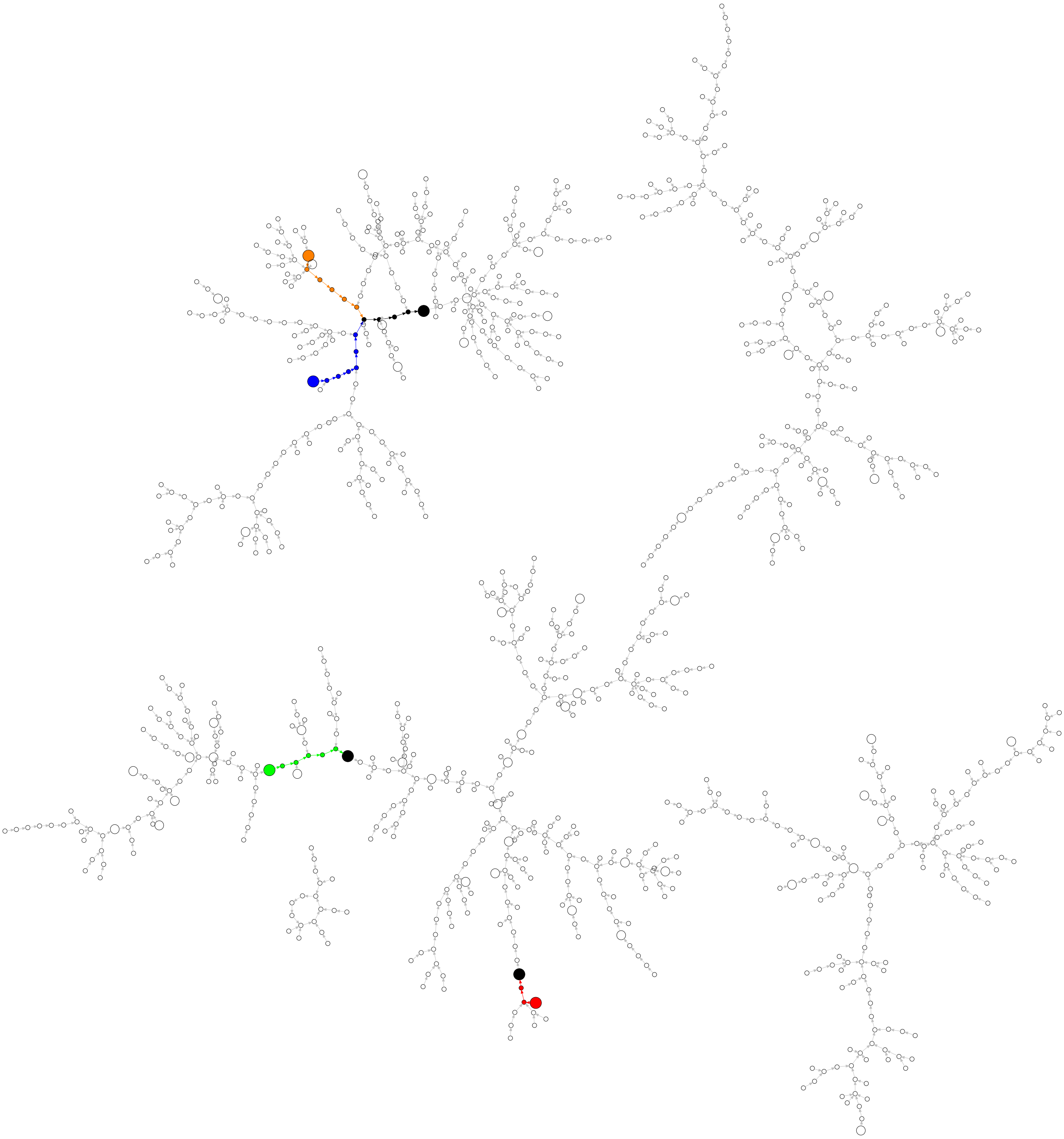
Running Pollard's rho method on N computers gives speedup of $\approx \sqrt{N}$ from increased likelihood of finding collision.

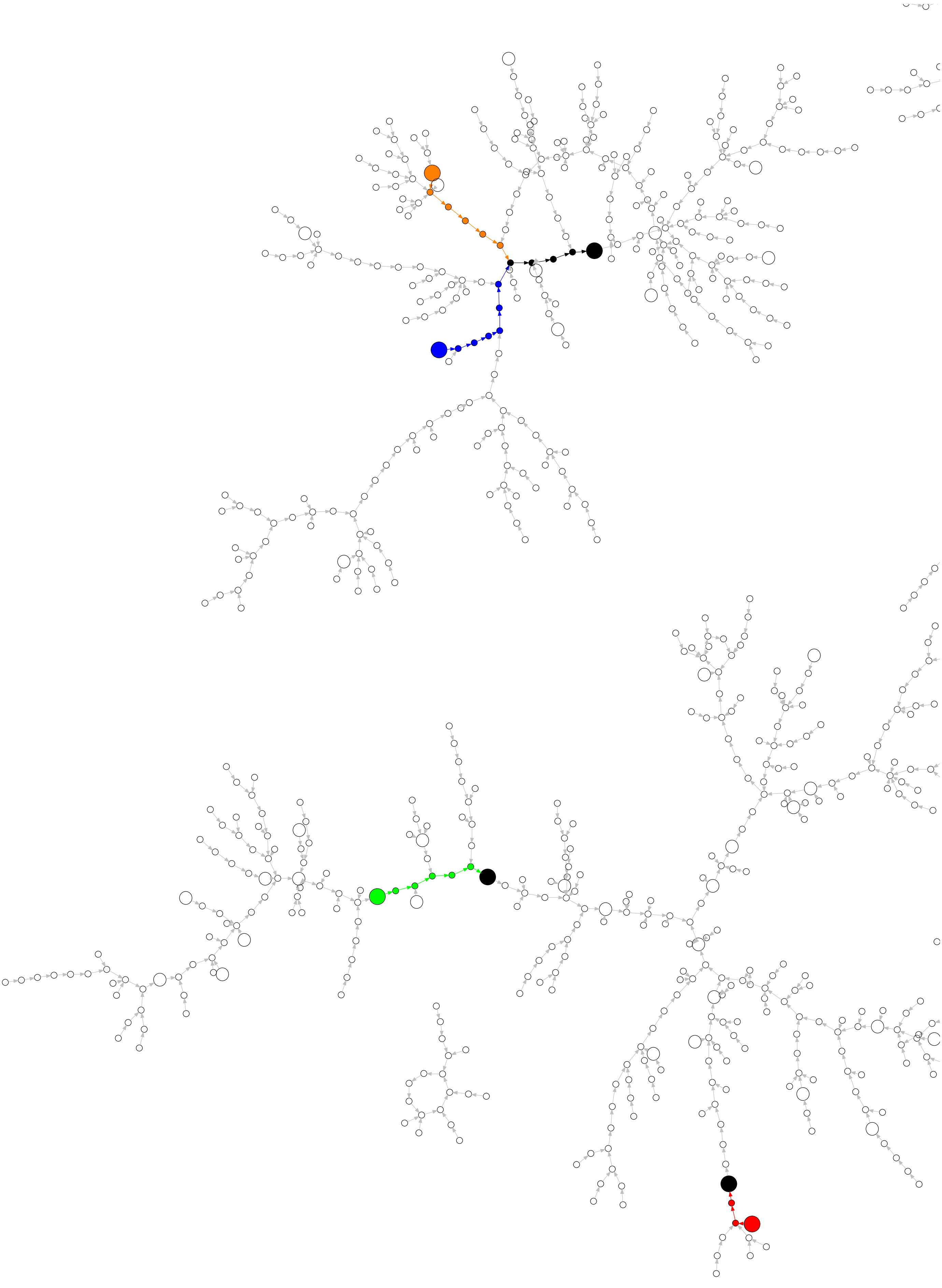
Want better way to spread computation across clients. Want to find collisions between walks on *different* machines, without frequent synchronization!

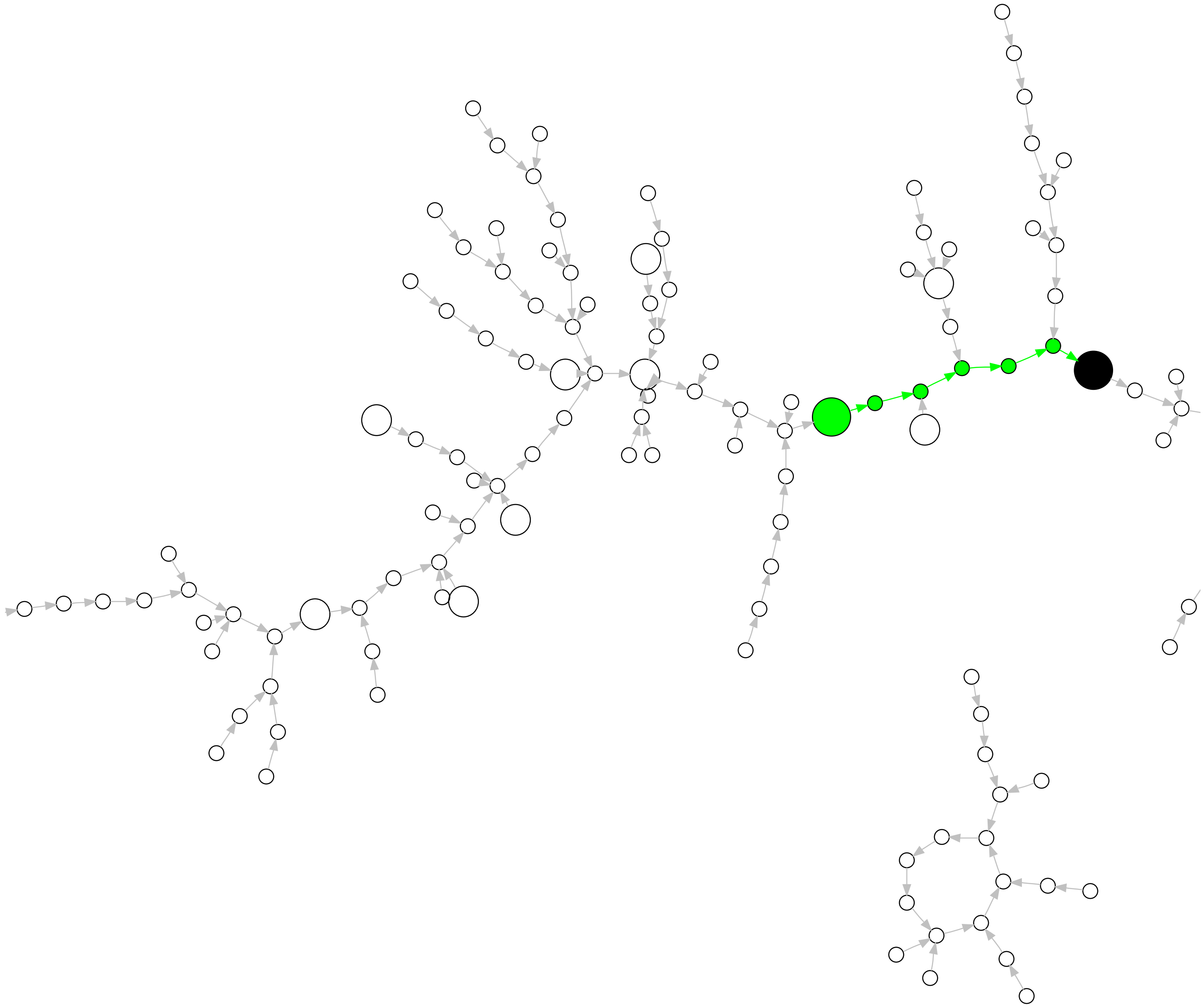
Better method due to van Oorschot and Wiener (1999). Declare some subset of $\langle P \rangle$ to be *distinguished points*.

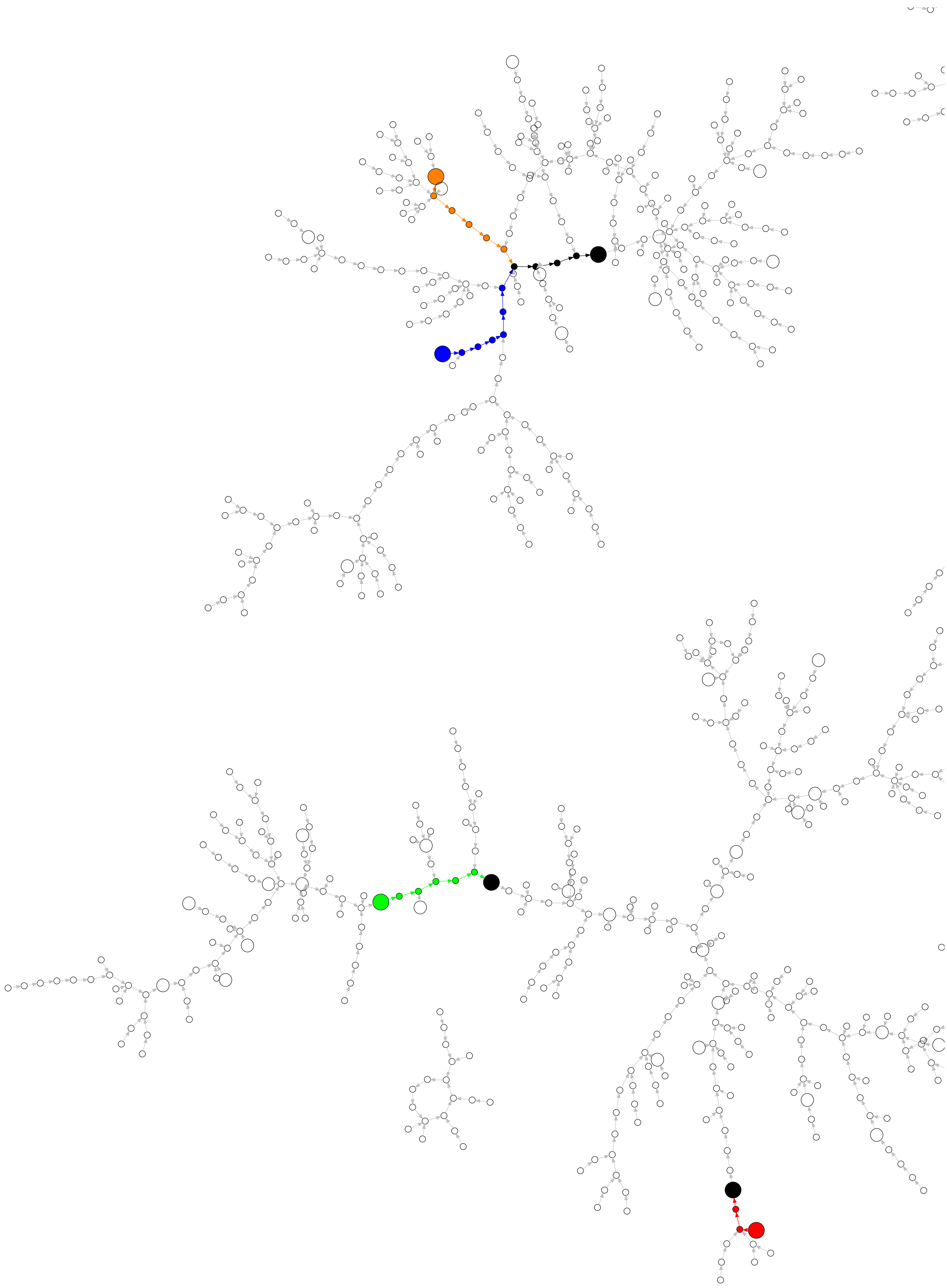
Parallel rho: Perform many walks with different starting points but same update function f.
If two different walks find the same point then their subsequent steps will match.

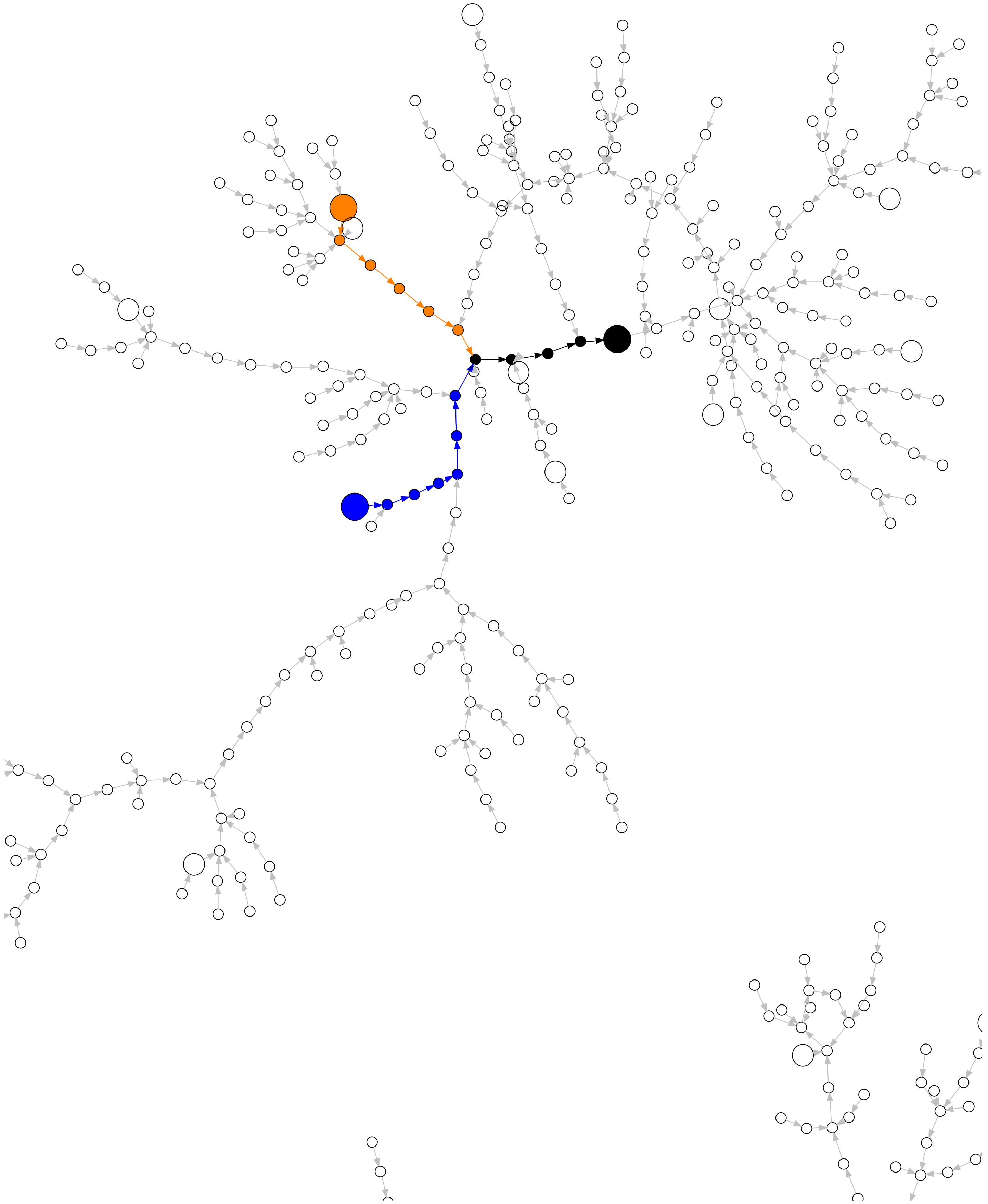
Terminate each walk once it hits a distinguished point and report the point along with a_i and b_i to server.
Server receives, stores, and sorts all distinguished points.
Two walks reaching same distinguished point give collision.
This collision solves the DLP.











Attacker chooses frequency and definition of distinguished points.
Tradeoffs are possible:
If distinguished points are rare, a small number of very long walks will be performed. This reduces the number of distinguished points sent to the server but increases the delay before a collision is recognized.
If distinguished points are frequent, many shorter walks will be performed.
In any case do not wait for cycle.
Total # of iterations unchanged.