

ECDLP course

Generic attacks

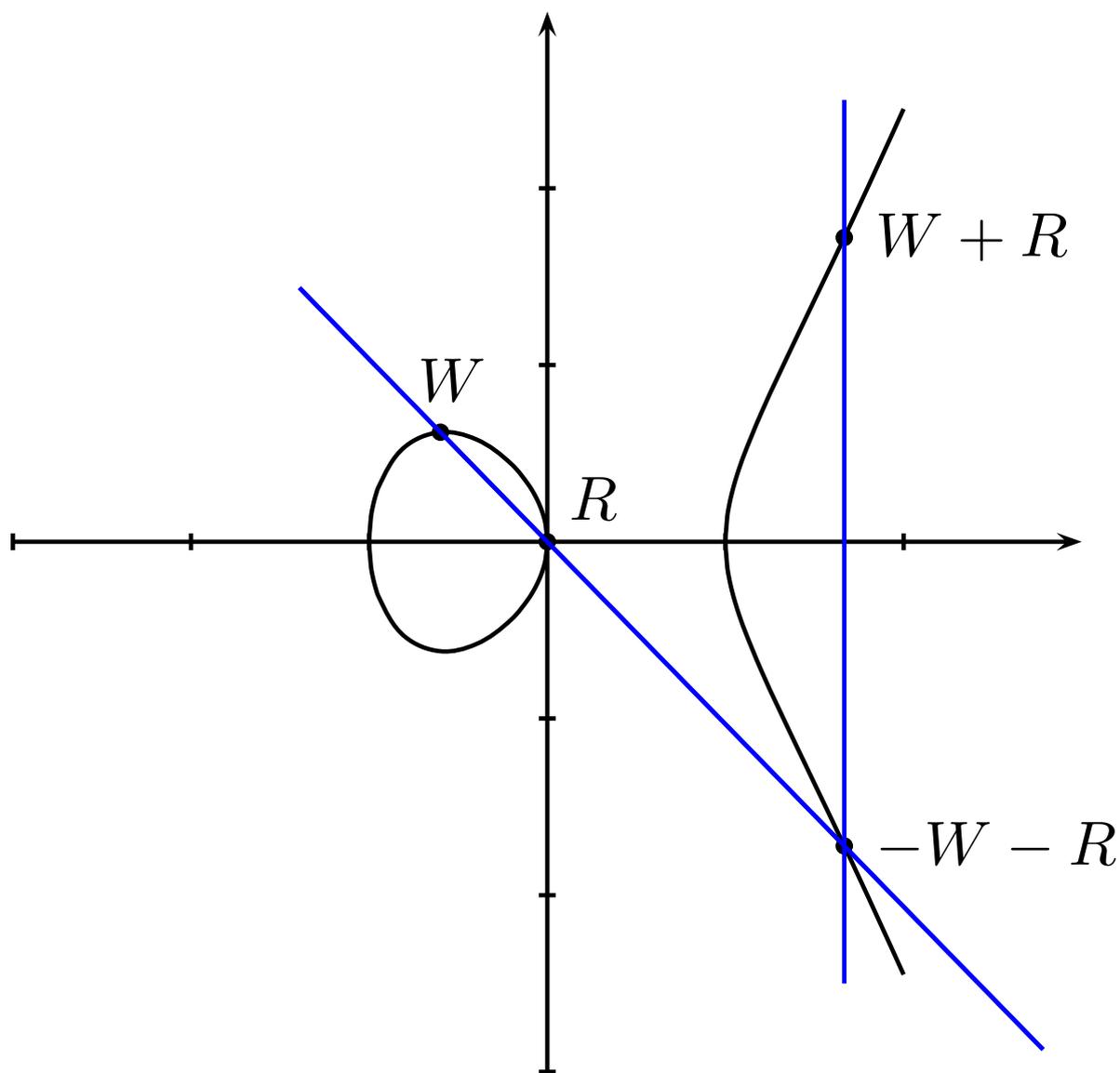
Daniel J. Bernstein

University of Illinois at Chicago

Tanja Lange

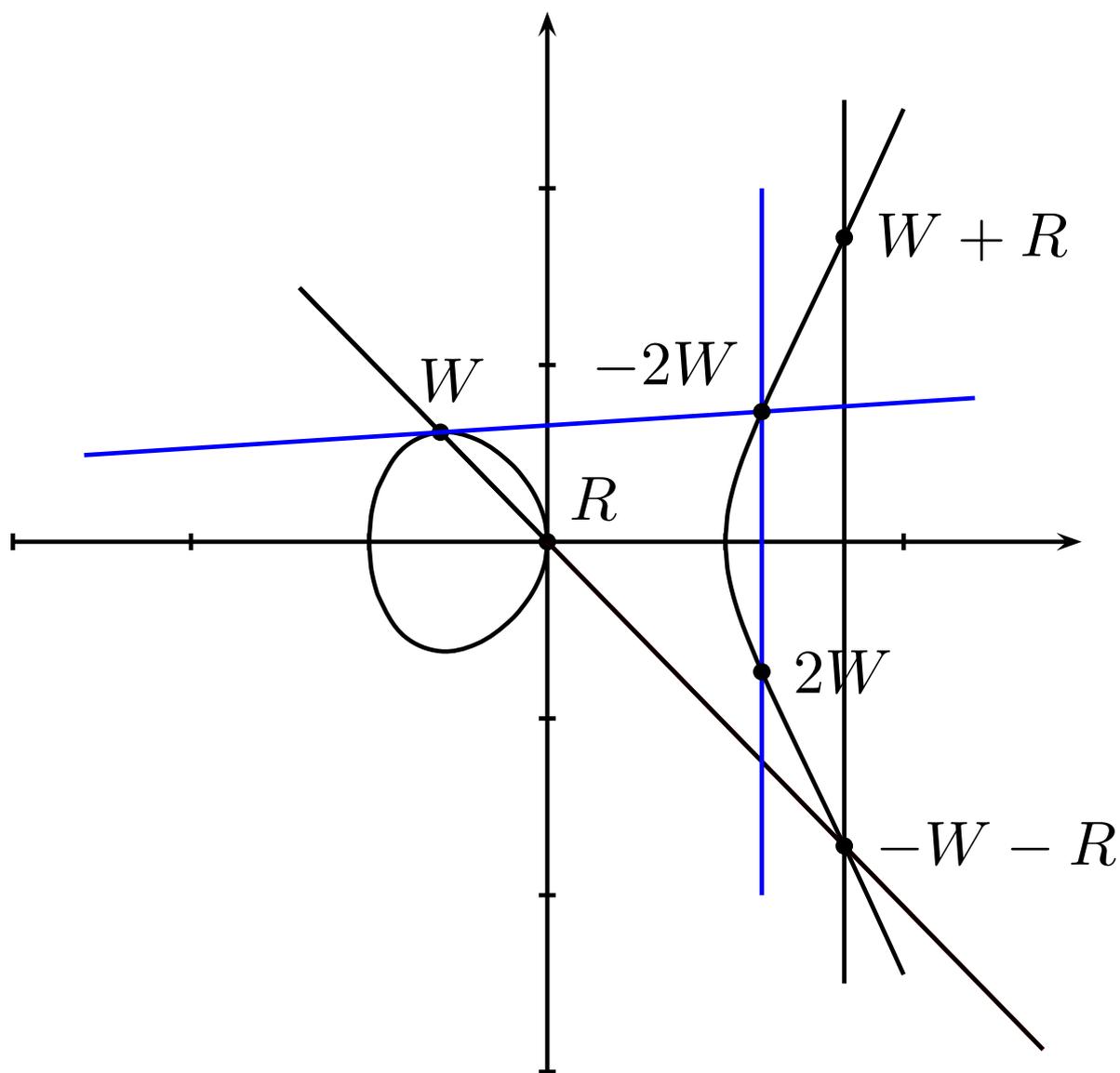
Technische Universiteit Eindhoven

Elliptic-curve groups



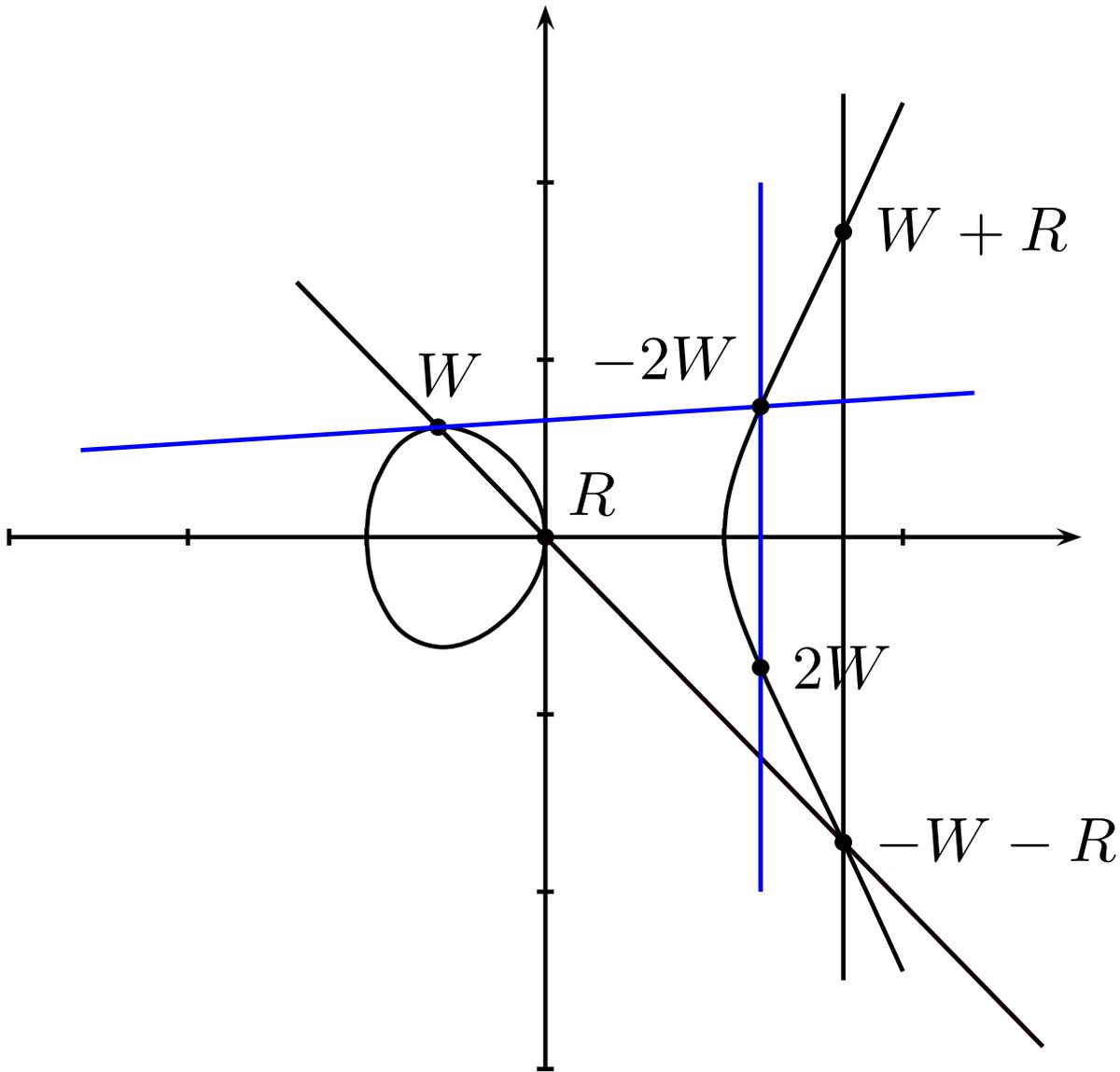
$$y^2 = x^3 + ax + b.$$

Elliptic-curve groups



$$y^2 = x^3 + ax + b.$$

Elliptic-curve groups



$$y^2 = x^3 + ax + b.$$

Also neutral element at ∞ .

$$-(x, y) = (x, -y).$$

$$\begin{aligned}
 (x_W, y_W) + (x_R, y_R) &= \\
 (x_{W+R}, y_{W+R}) &= \\
 (\lambda^2 - x_W - x_R, \lambda(x_W - x_{W+R}) - y_W).
 \end{aligned}$$

$x_W \neq x_R$, “addition”:

$$\lambda = (y_R - y_W) / (x_R - x_W).$$

Total cost **1I + 2M + 1S**.

$W = R$ and $y_W \neq 0$, “doubling”:

$$\lambda = (3x_W^2 + a) / (2y_W).$$

Total cost **1I + 2M + 2S**.

Following algorithms will need a unique representative per point.

For that Weierstrass curves are the speed leader

$$\begin{aligned}
& (x_W, y_W) + (x_R, y_R) = \\
& (x_{W+R}, y_{W+R}) = \\
& (\lambda^2 - x_W - x_R, \lambda(x_W - x_{W+R}) - y_W).
\end{aligned}$$

$x_W \neq x_R$, “addition”:

$$\lambda = (y_R - y_W) / (x_R - x_W).$$

Total cost **1I + 2M + 1S**.

$W = R$ and $y_W \neq 0$, “doubling”:

$$\lambda = (3x_W^2 + a) / (2y_W).$$

Total cost **1I + 2M + 2S**.

Following algorithms will need a unique representative per point.

For that Weierstrass curves

are the speed leader ... and I

thought turtles were defensive.

The discrete-logarithm problem

Define $p = 1000003$ and

consider the Weierstrass curve

$$y^2 = x^3 - x \text{ over } \mathbf{F}_p.$$

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
 $n \in \{1, 2, 3, \dots, 500001\}$
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

$$\text{with } nP = (670366, 740819).$$

Maximum cost of computation:

$$\leq 500001 \text{ additions of } P;$$

$$\leq 500001 \text{ nanoseconds on a CPU}$$

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 .”

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_1P , n_2P , \dots , $n_{100}P$:

Can find *all* of n_1, n_2, \dots, n_{100} with ≤ 500002 ADDs.

Simplest approach: First build a sorted table containing $n_1P, \dots, n_{100}P$.

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 ?

Typically $\approx 500002/100$ mults.

From now on, use $\ell = \text{ord}(P)$.
Can use random self-reduction
to turn a single target
into multiple targets.

Given nP :

Choose random r_1, r_2, \dots, r_{100} .

Compute r_1P ,
 r_2P , etc.

Solve these 100 DL problems.

Typically $\approx \ell/100$ mults

to find *at least one*

$r_i + n \pmod{\ell}$,

immediately revealing n .

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

Faster: Choose $r_i = ir_1$
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:

$$708P + Q = (342867, 153817),$$

$$2 \cdot 708P + nP = (430321, 994742),$$

$$3 \cdot 708P + nP = (423151, 635197),$$

$$\dots, 706 \cdot 708P + nP =$$

$$(534170, 450849).$$

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.

$$620P = (950652, 688508); \text{ find}$$

$$596 \cdot 708P + Q = (950652, 688508)$$

in the table of targets;

$$\text{so } 620 = 596 \cdot 708 + n \pmod{500002};$$

$$\text{deduce } n = 78654.$$

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 $n_1 = \log_R S \equiv n \pmod{2}$.

$R = (2 \cdot 53 \cdot 89)P$ has order 53,

and

$S = (2 \cdot 53 \cdot 89)Q$ is multiple of R .

Compute $n_2 = \log_R S \equiv n \pmod{53}$.

This is a DLP in a group of size 53.

$T = (2 \cdot 89)(Q - n_2P)$ is also a multiple of R .

Compute $n_3 = \log_R T \equiv n \pmod{53}$.

Now $n_2 + 53n_3 \equiv n \pmod{53^2}$.

$R = (2 \cdot 53^2)P$ has order 89, and $S = (2 \cdot 53^2)Q$ is multiple of R .

Compute $n_4 = \log_R S \equiv n \pmod{89}$.

Use Chinese Remainder Theorem

$$n \equiv n_1 \pmod{2},$$

$$n \equiv n_2 + 53n_3 \pmod{53^2},$$

$$n \equiv n_4 \pmod{89},$$

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

$$n \equiv 0 \pmod{2},$$

$$n \equiv 2 \pmod{53^2},$$

$$n \equiv 67 \pmod{89},$$

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

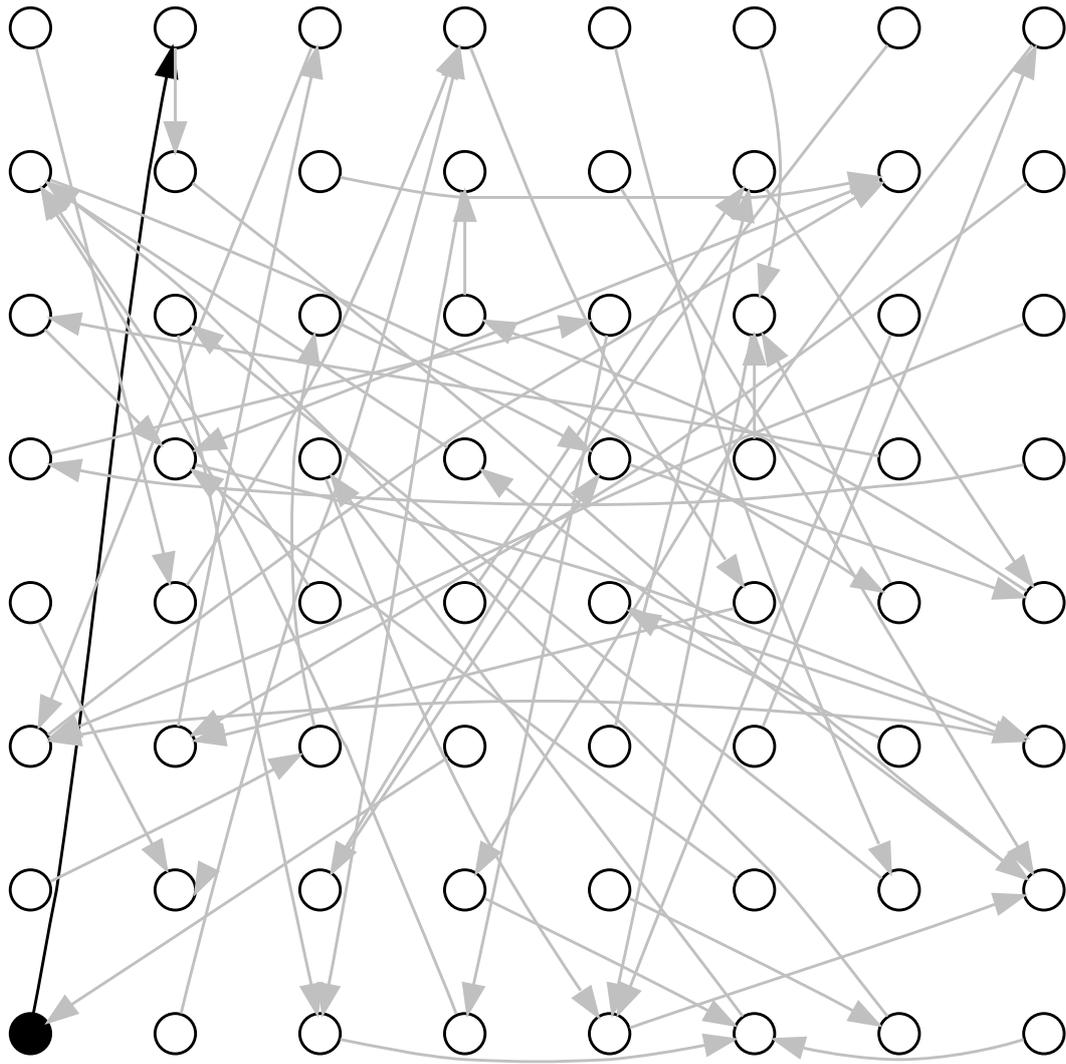
Birthday paradox:

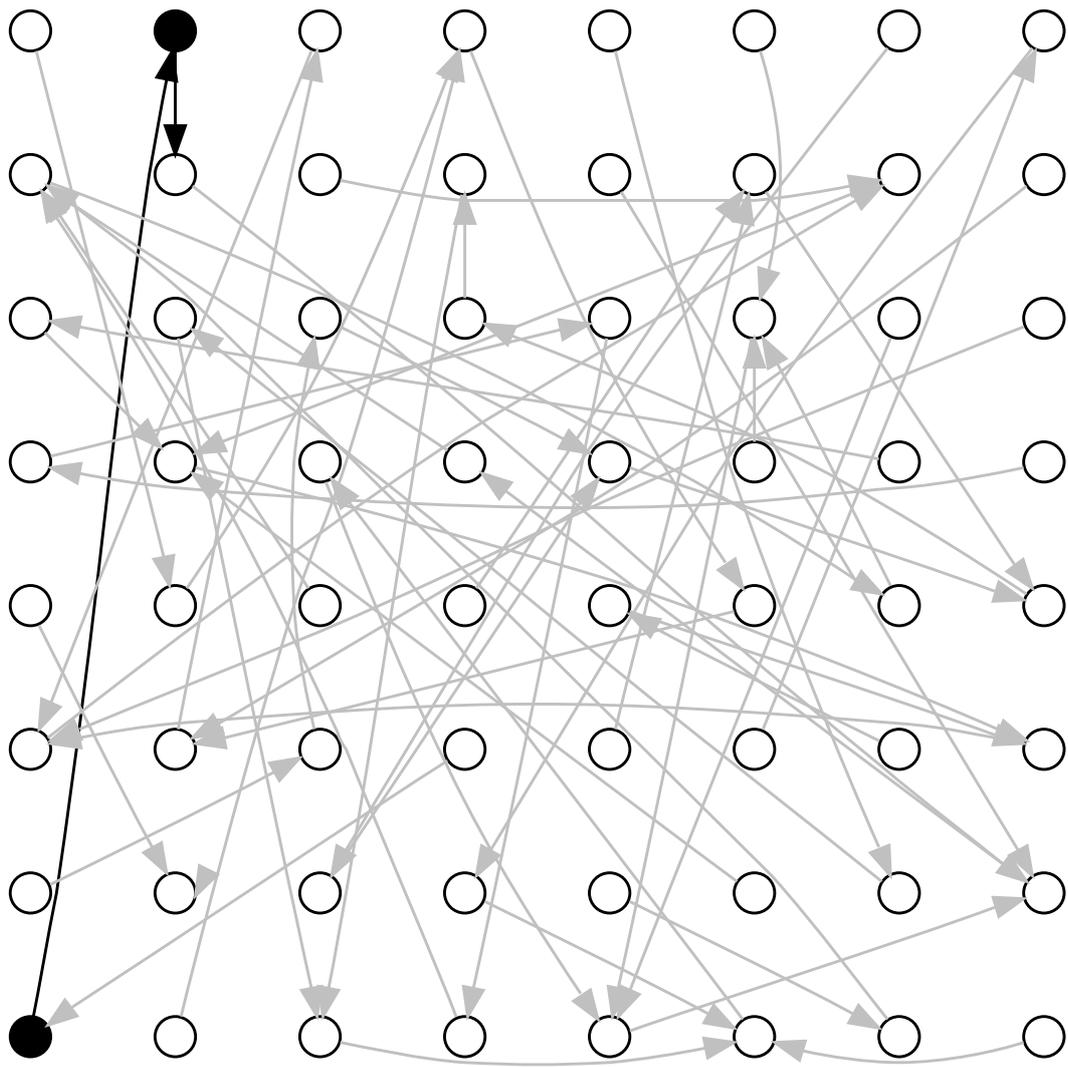
Randomly choosing from ℓ
elements picks one element twice
after about $\sqrt{\pi\ell/2}$ draws.

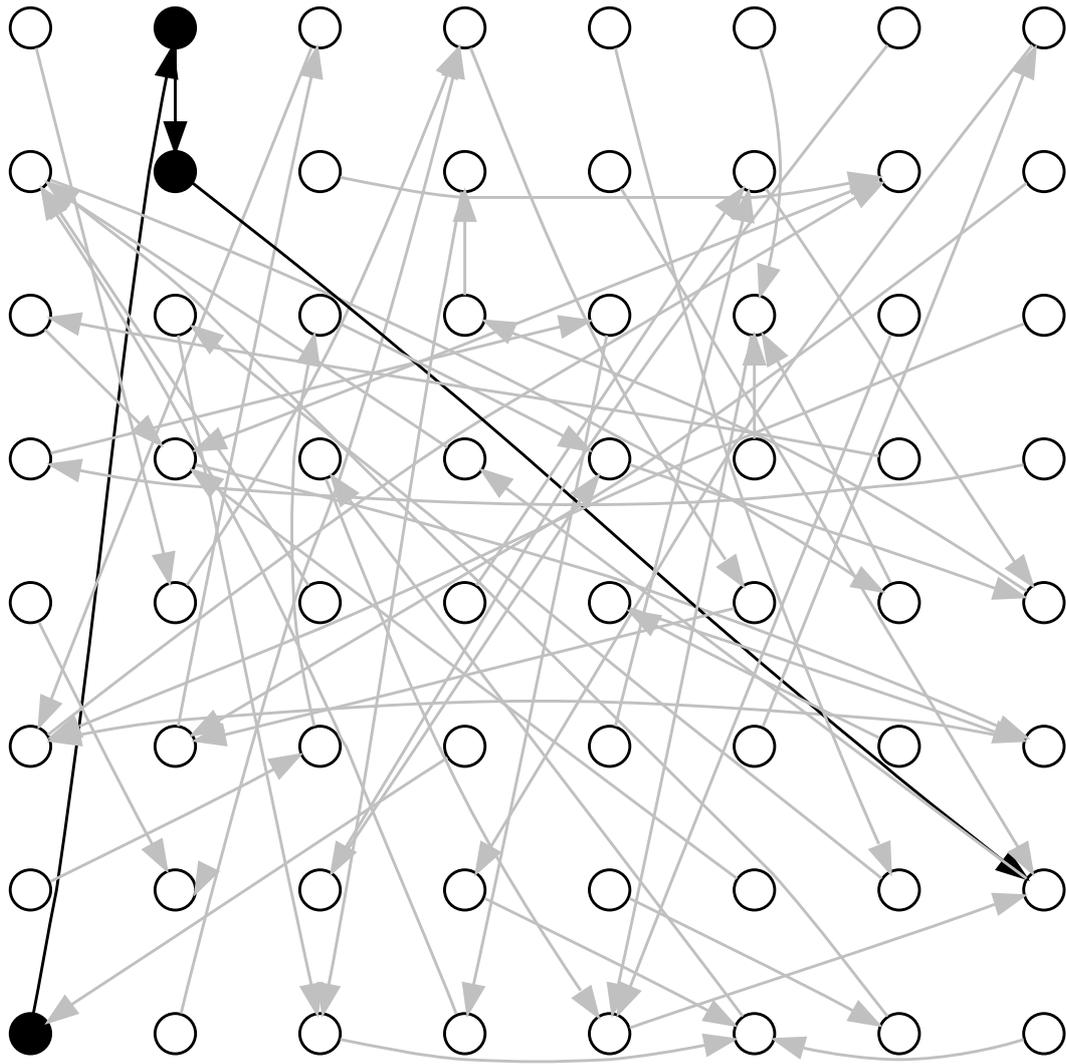
The walk now enters a cycle.

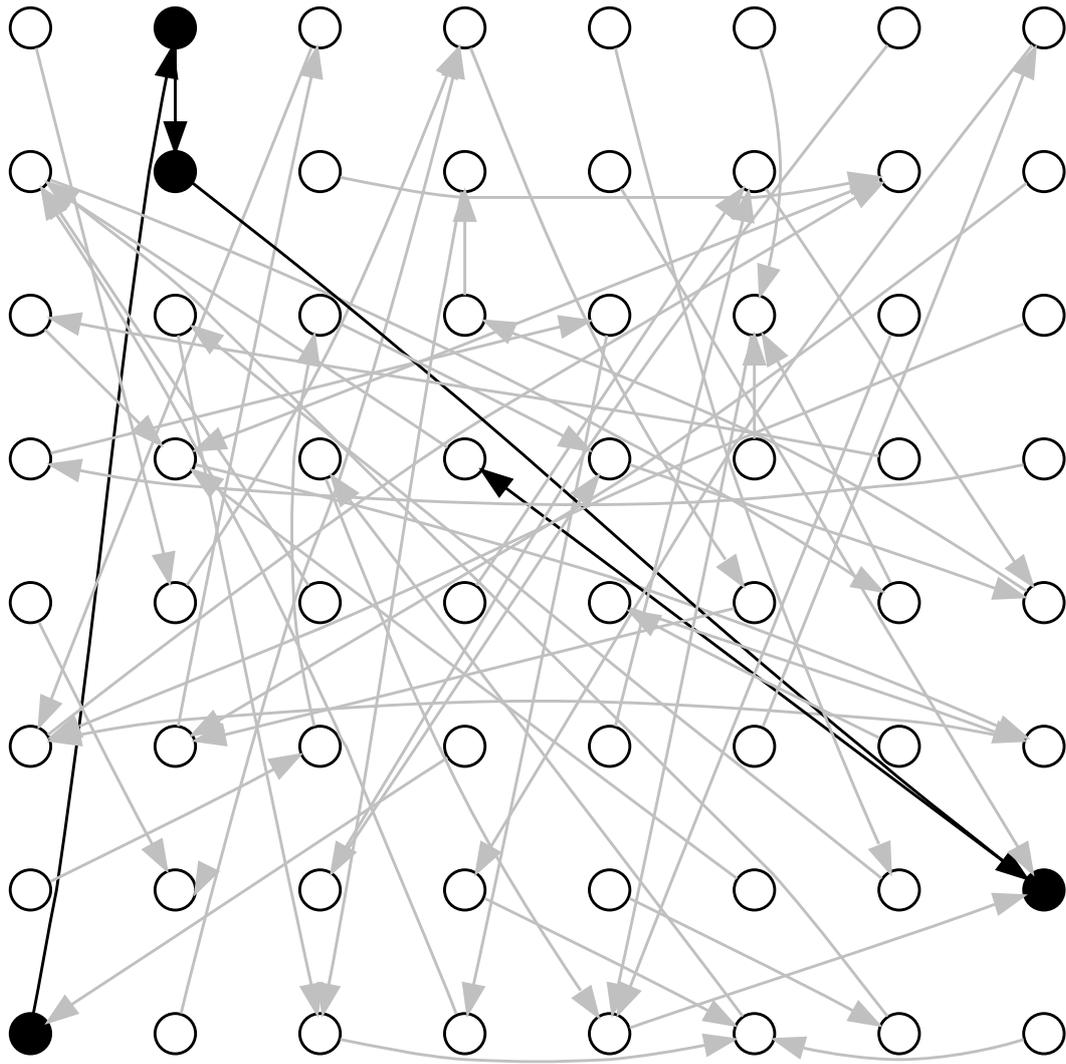
Cycle-finding algorithm

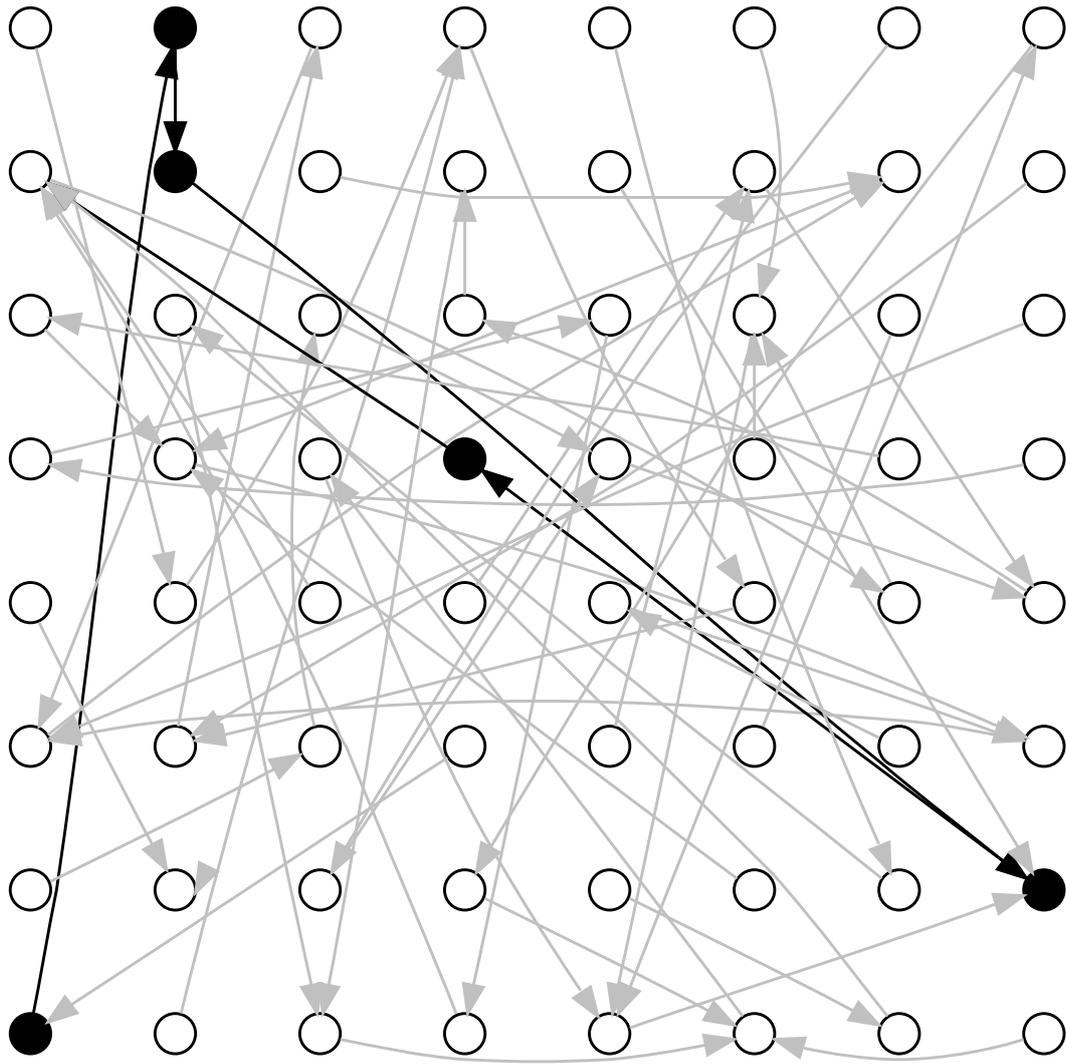
(e.g., Floyd) quickly detects this.

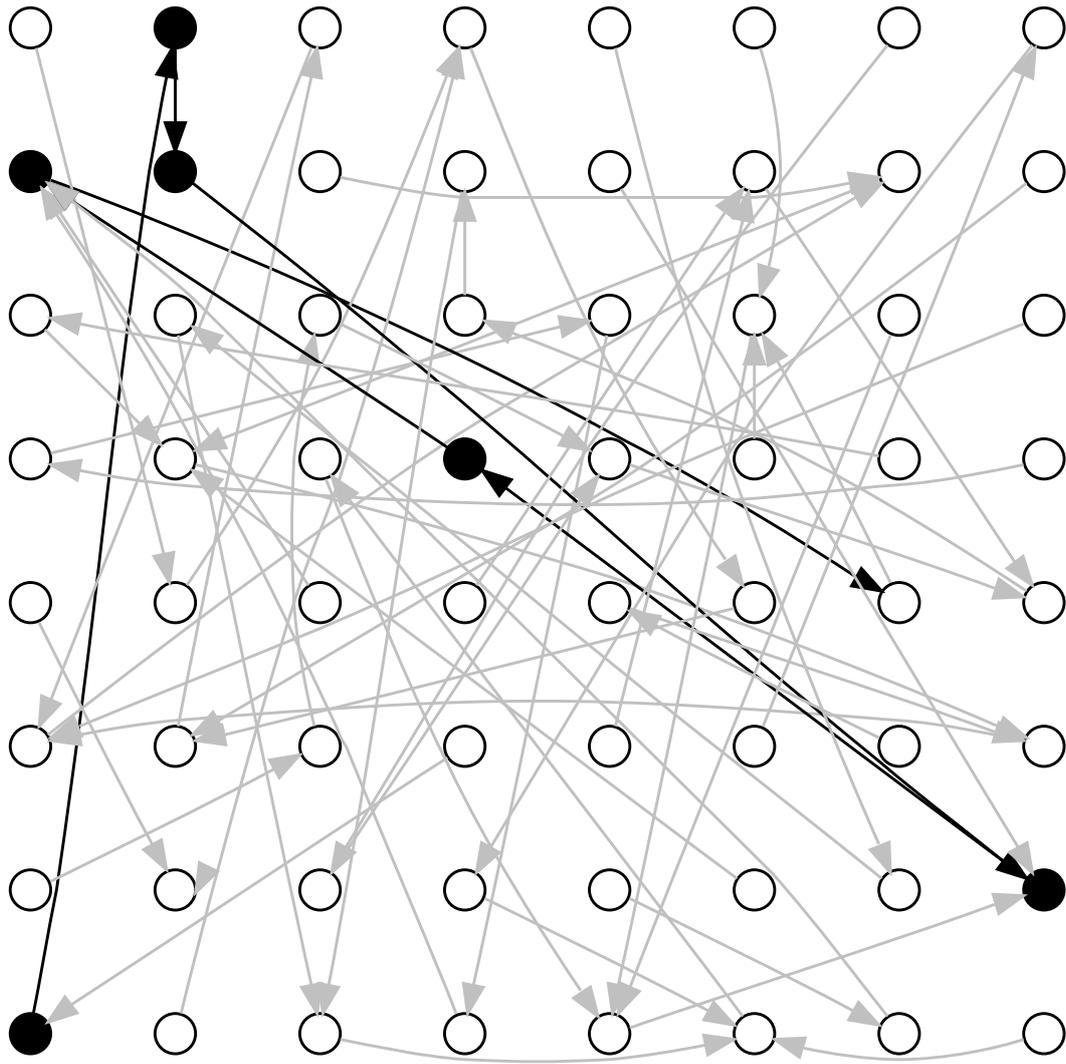


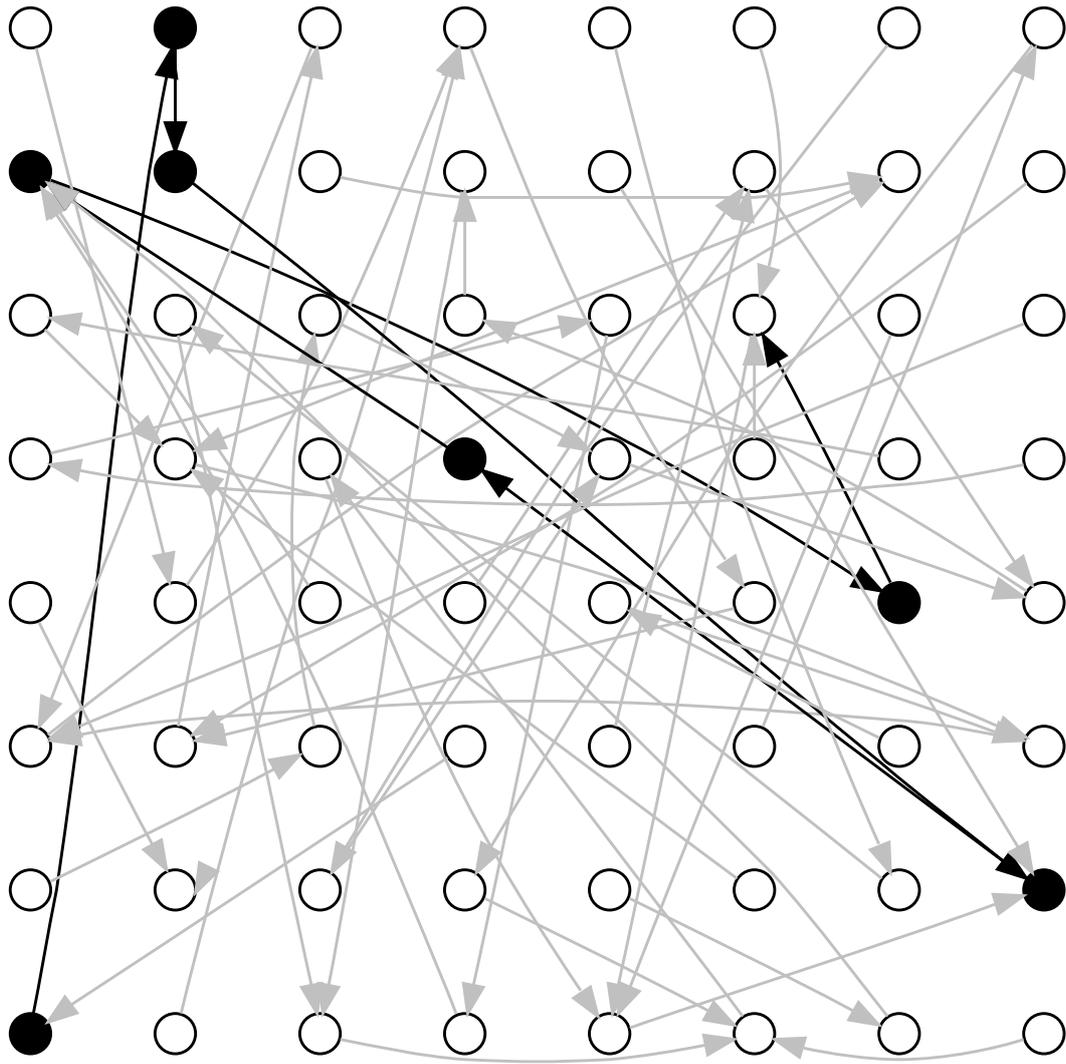


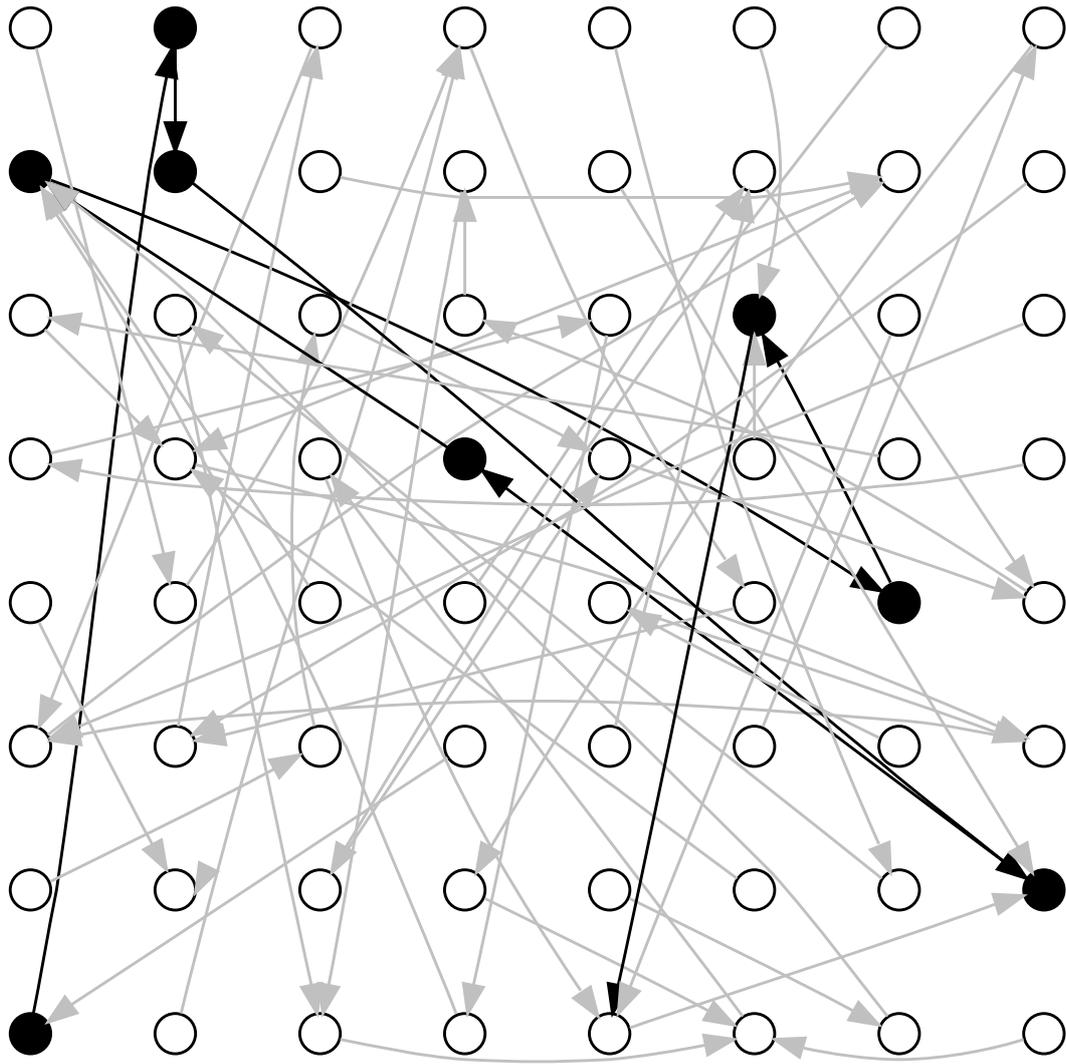


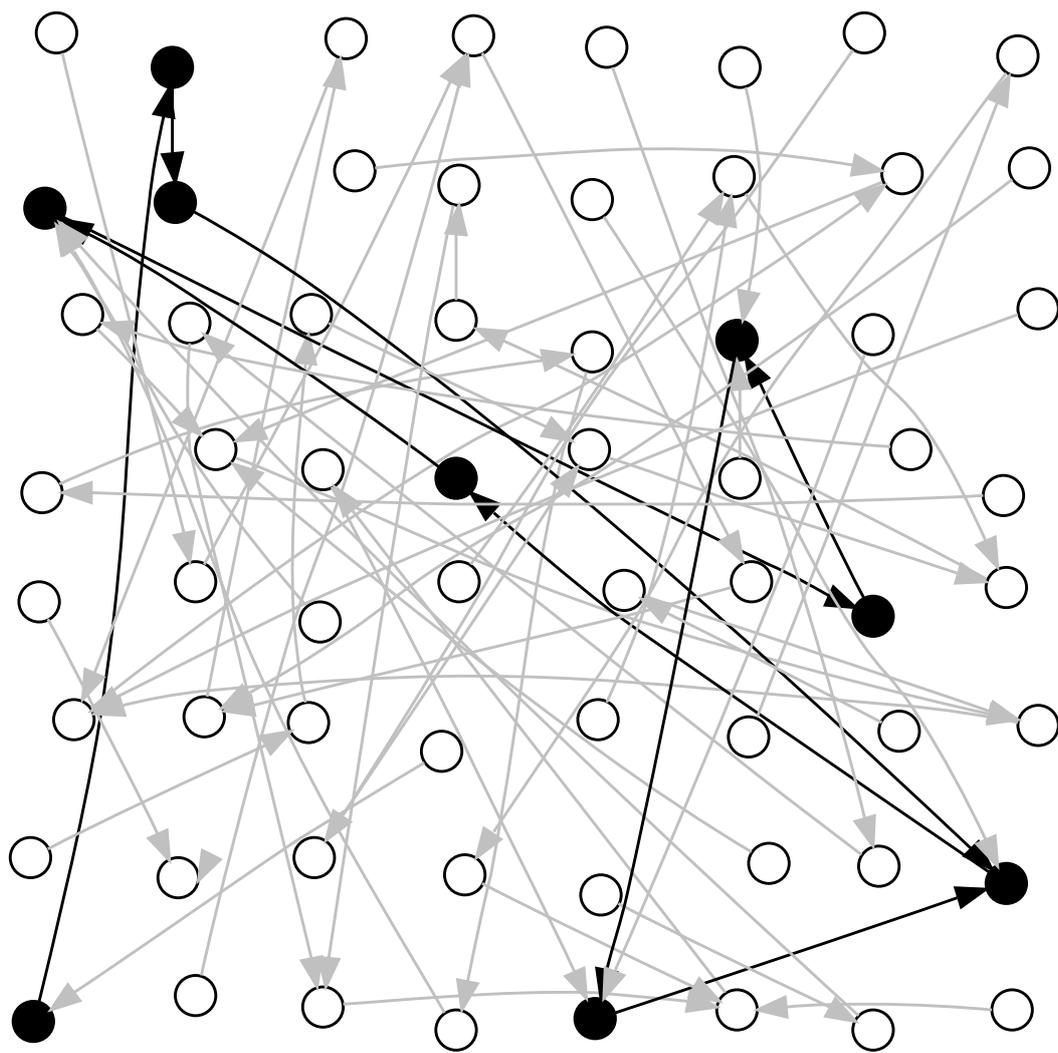


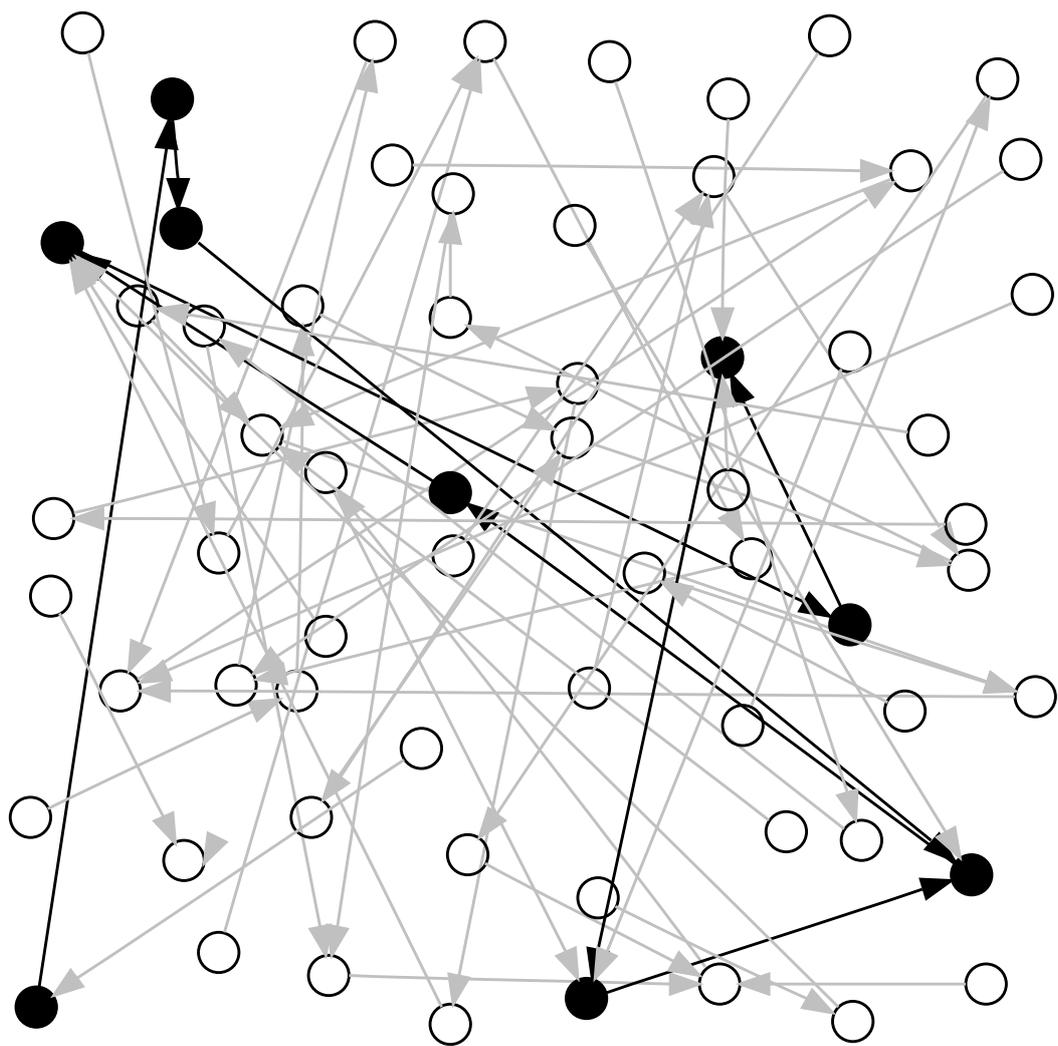


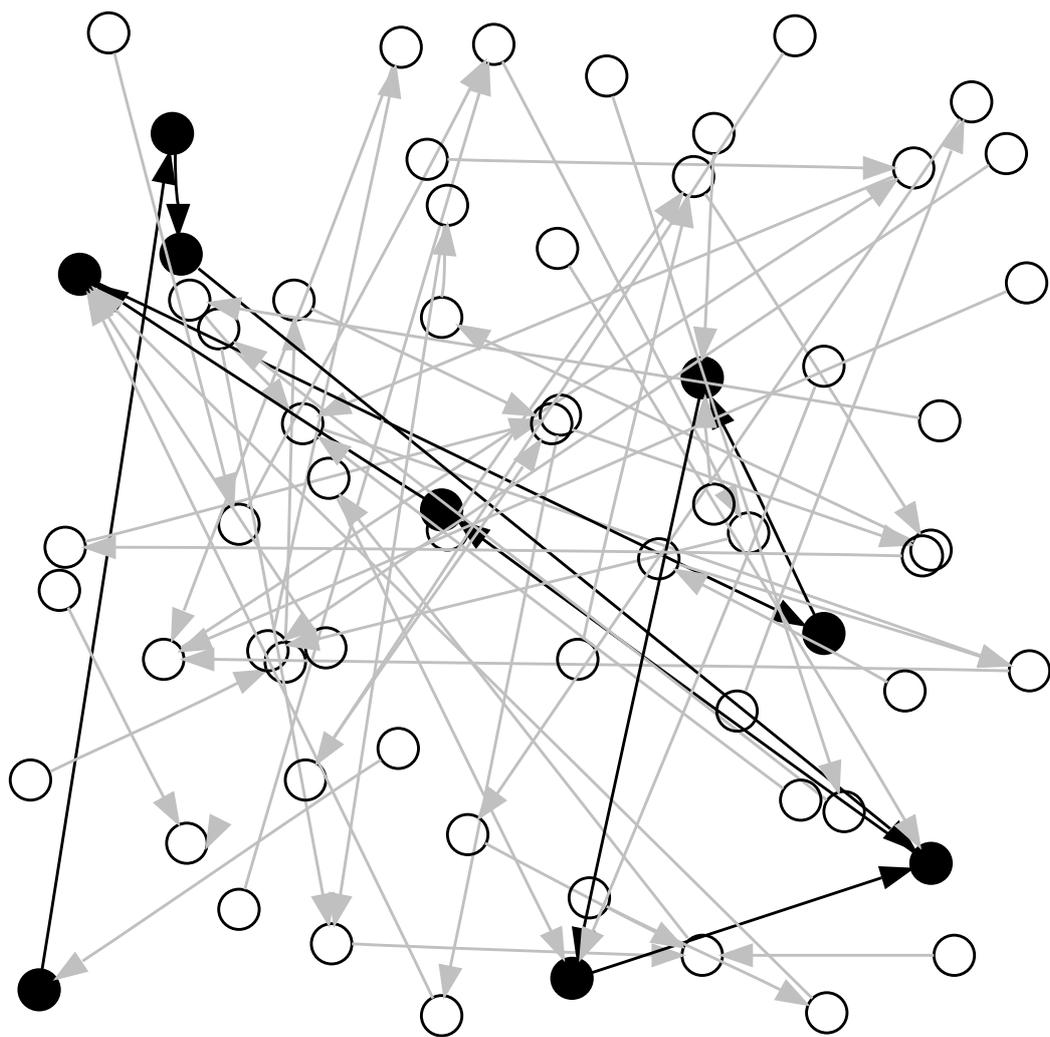


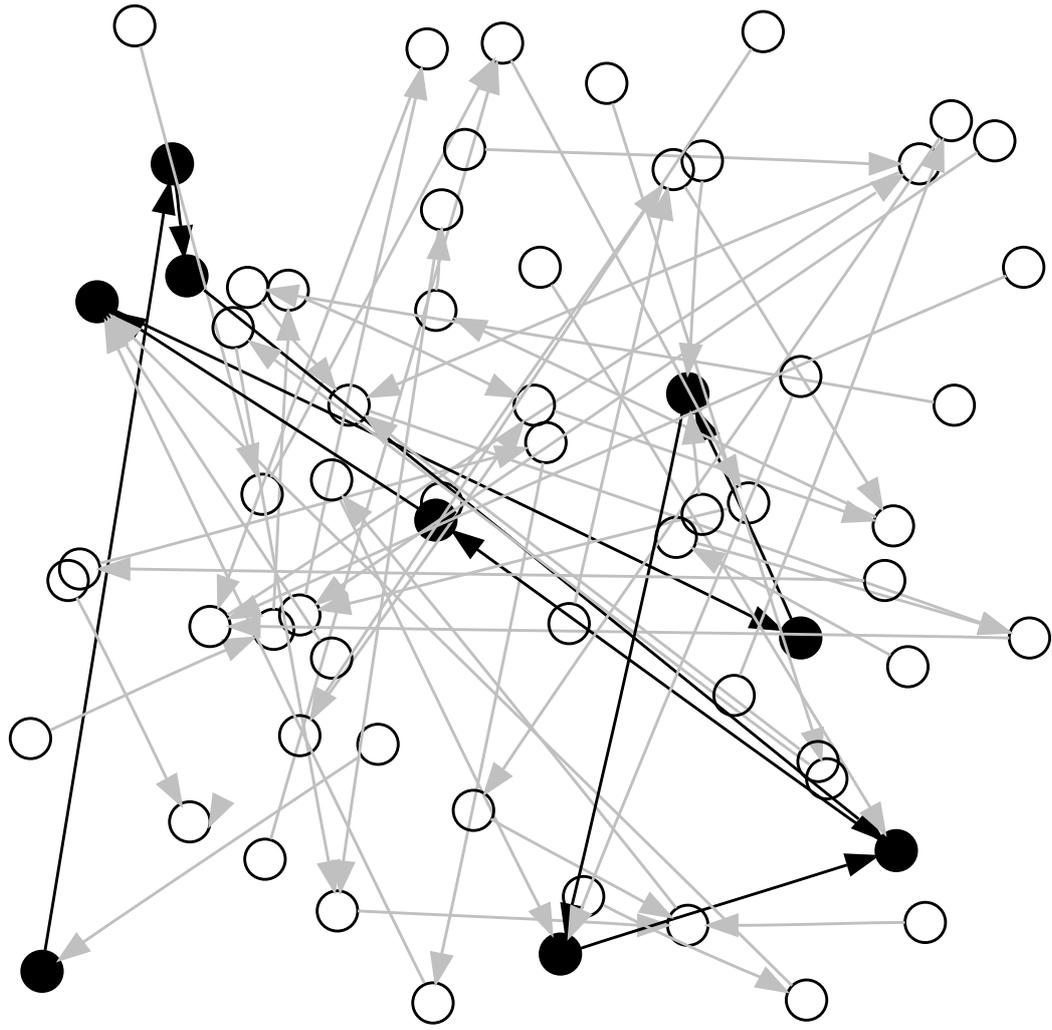


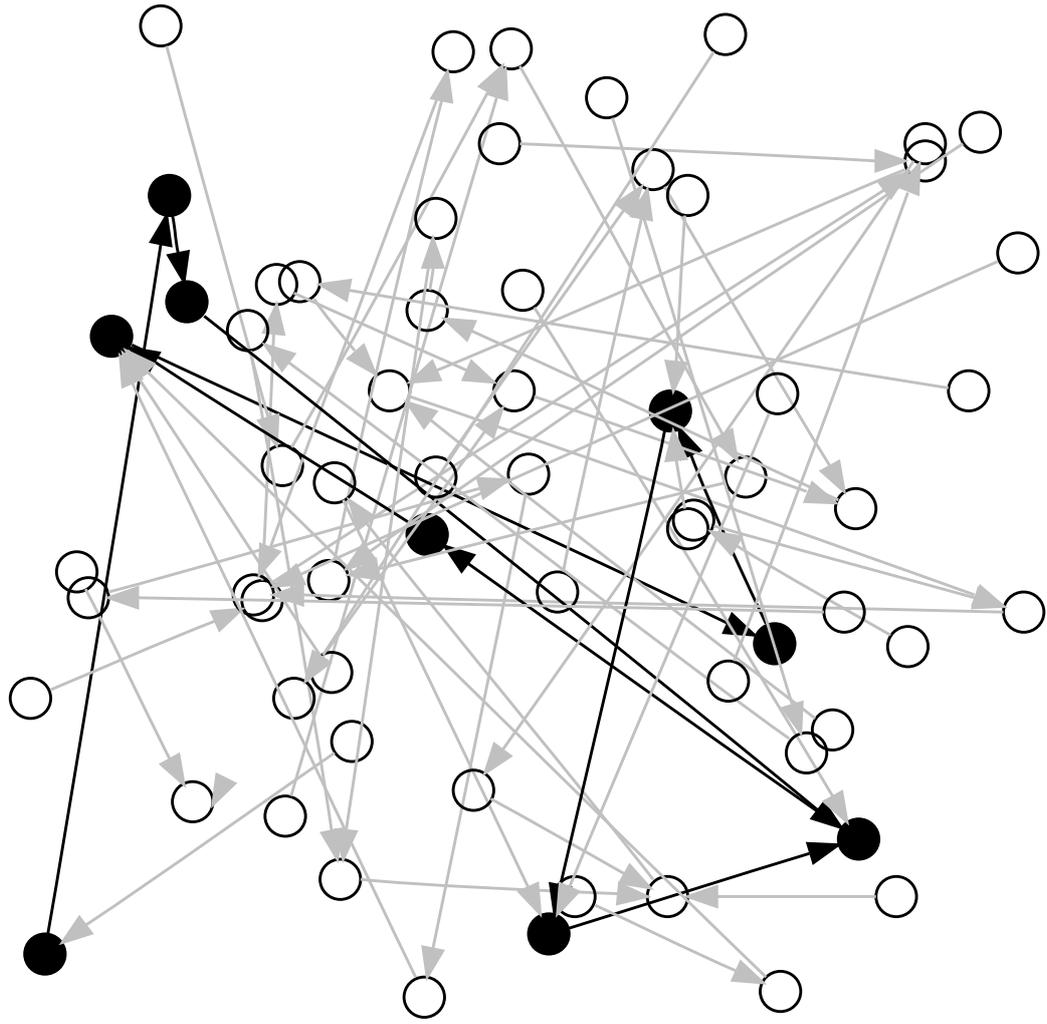


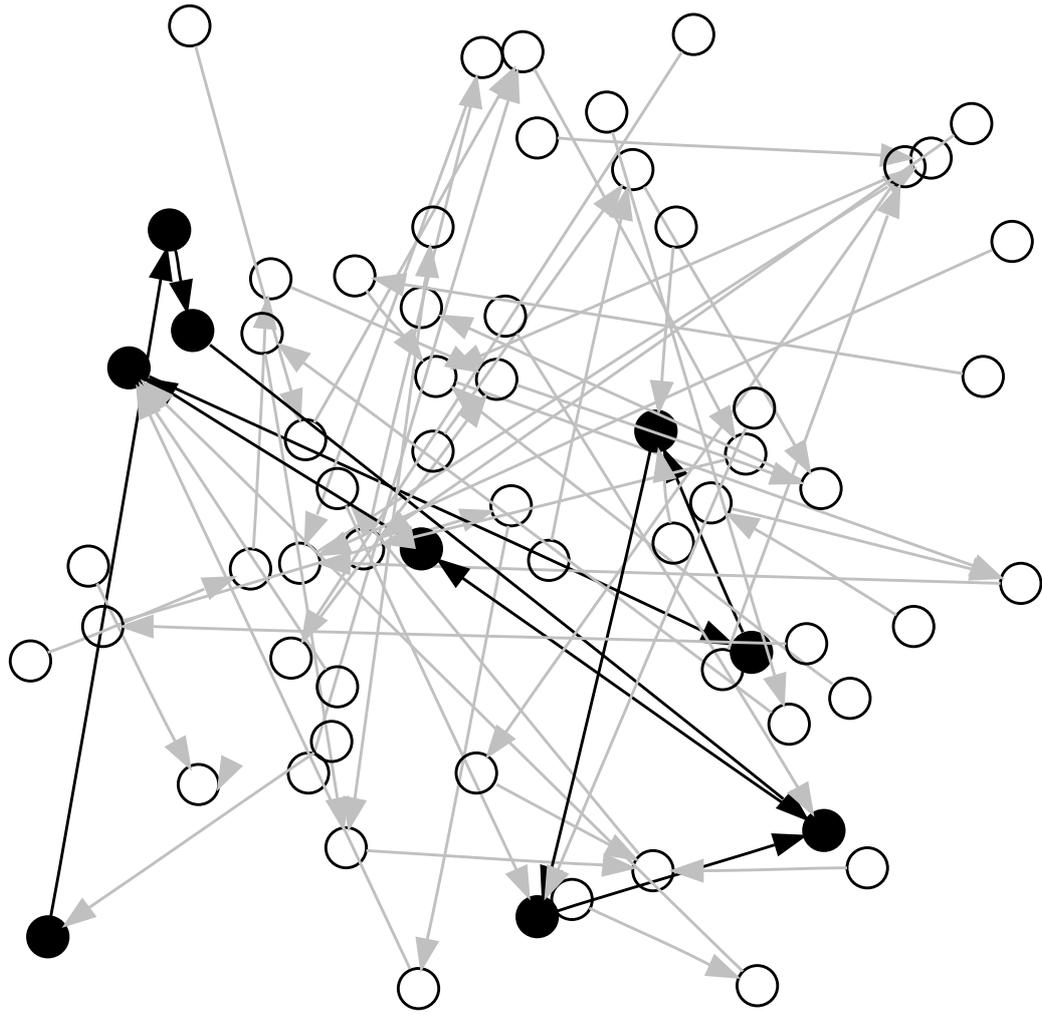


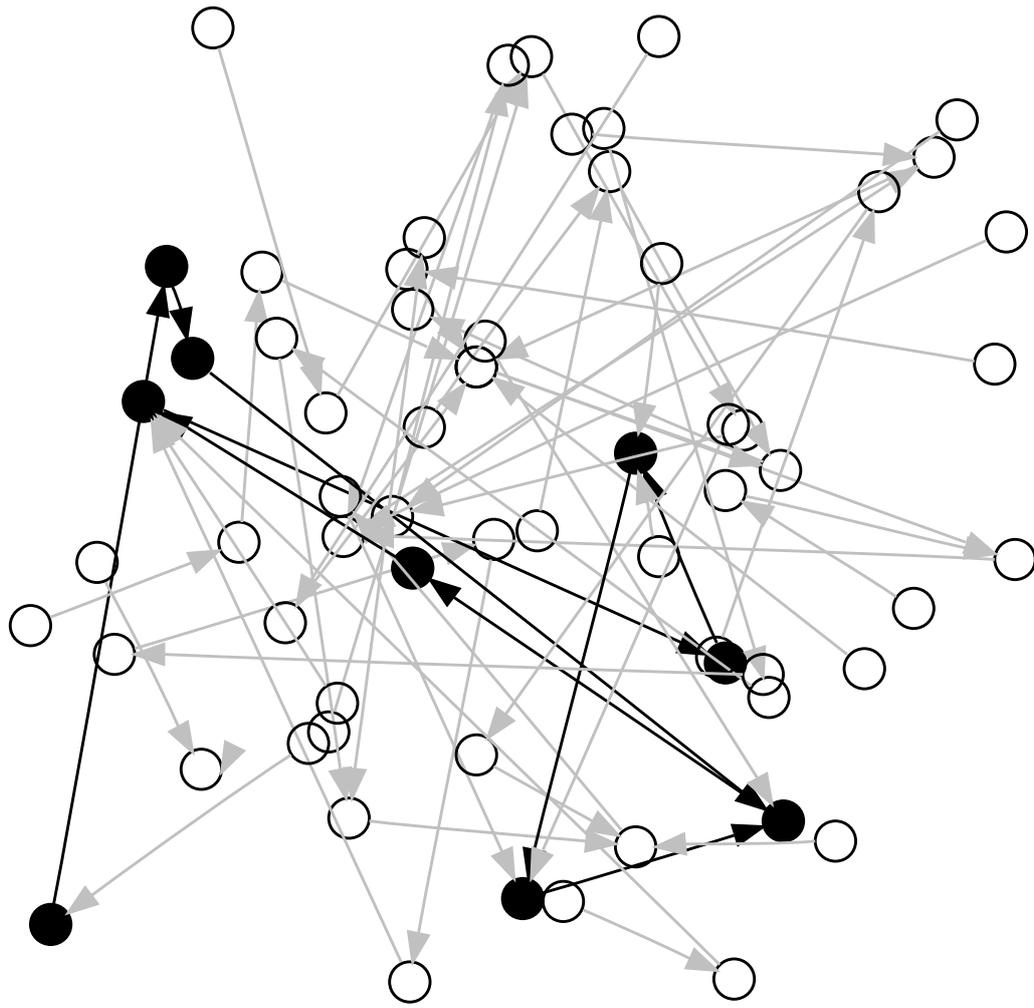


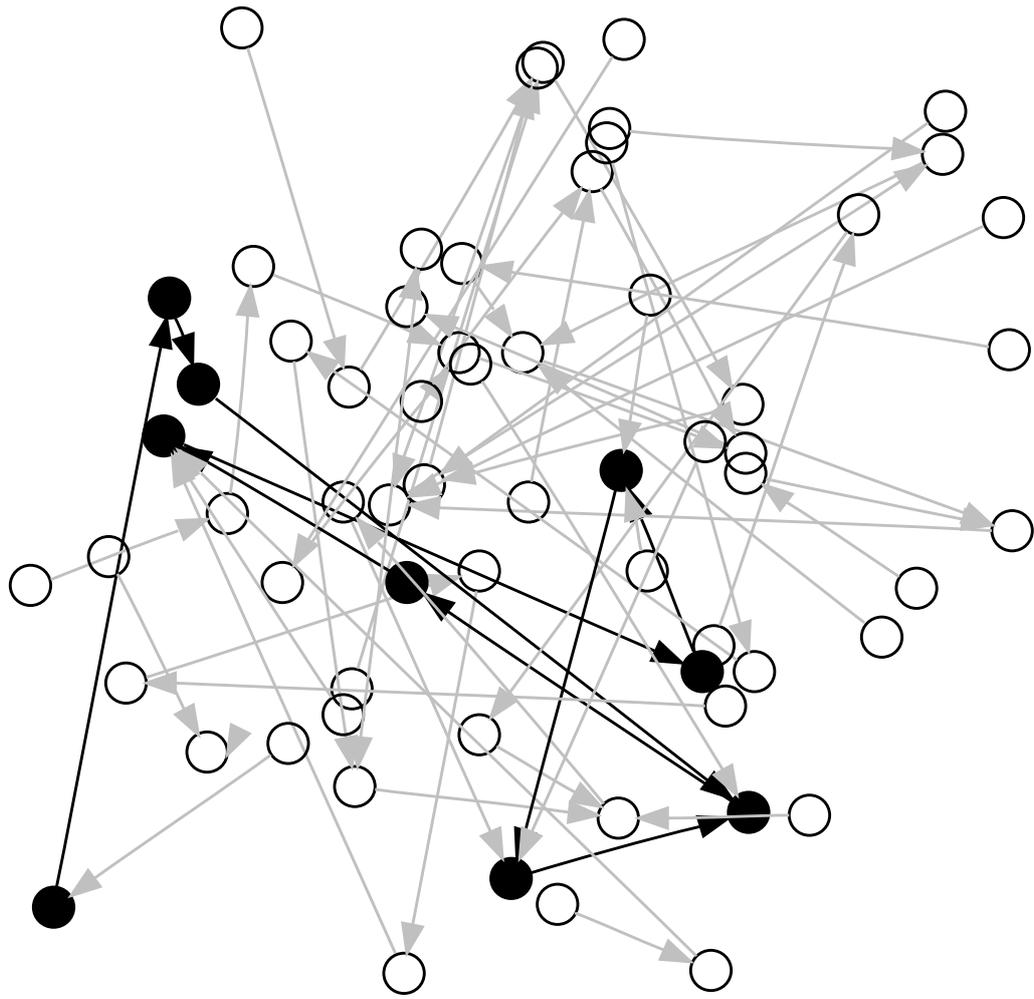


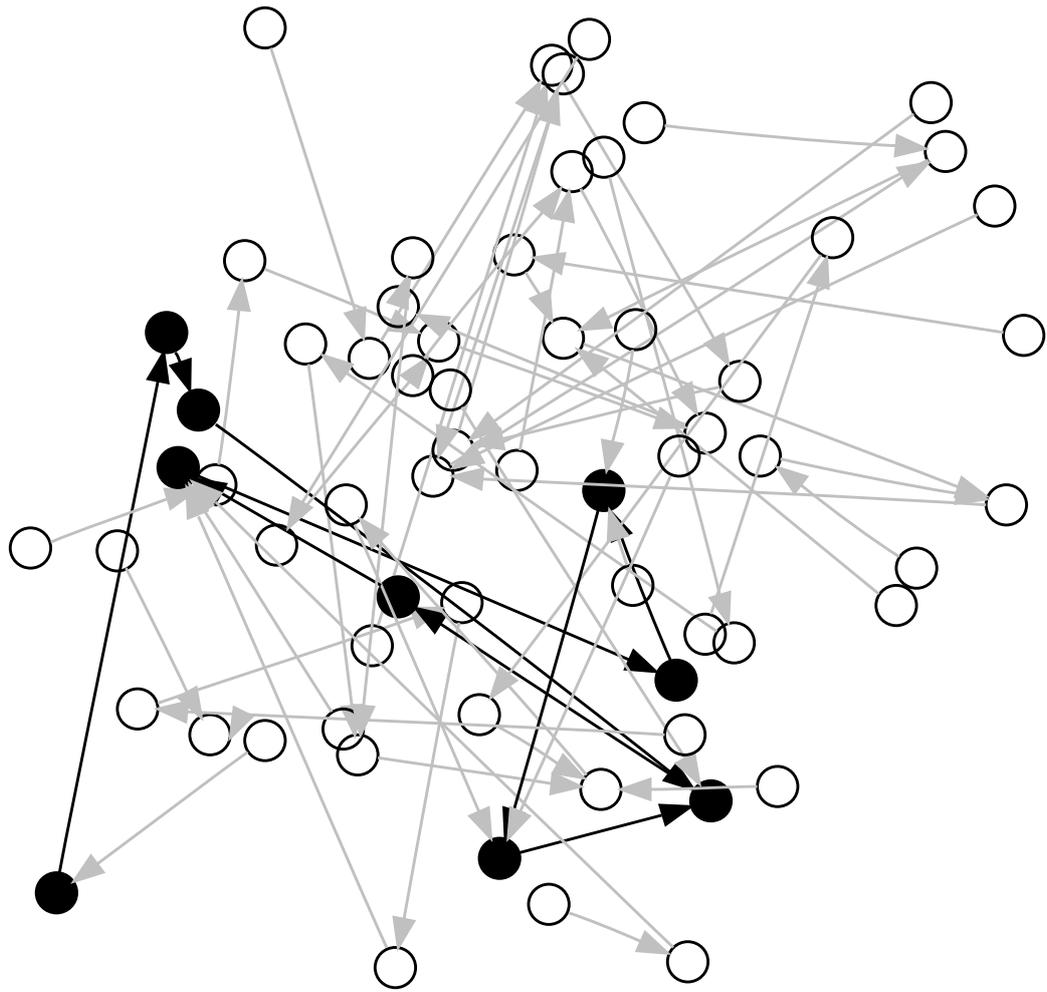


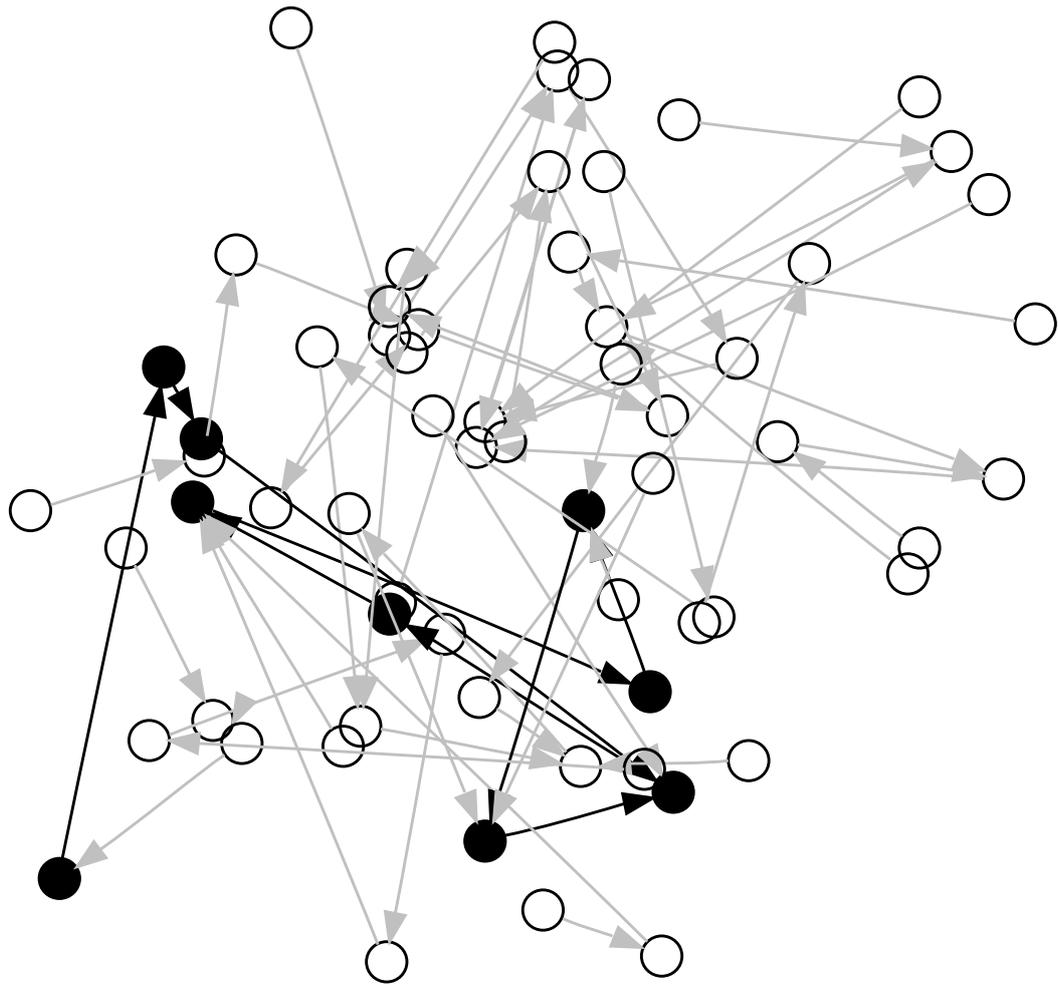


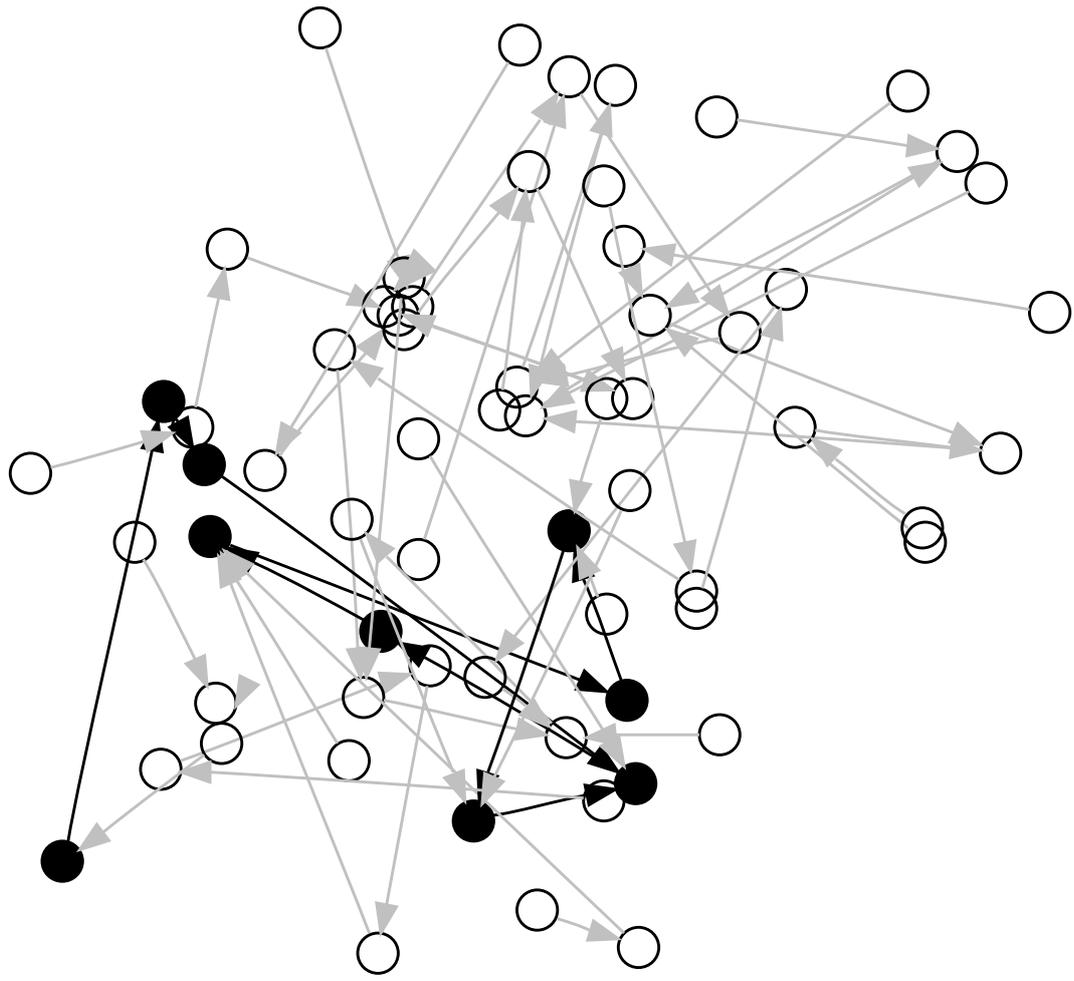


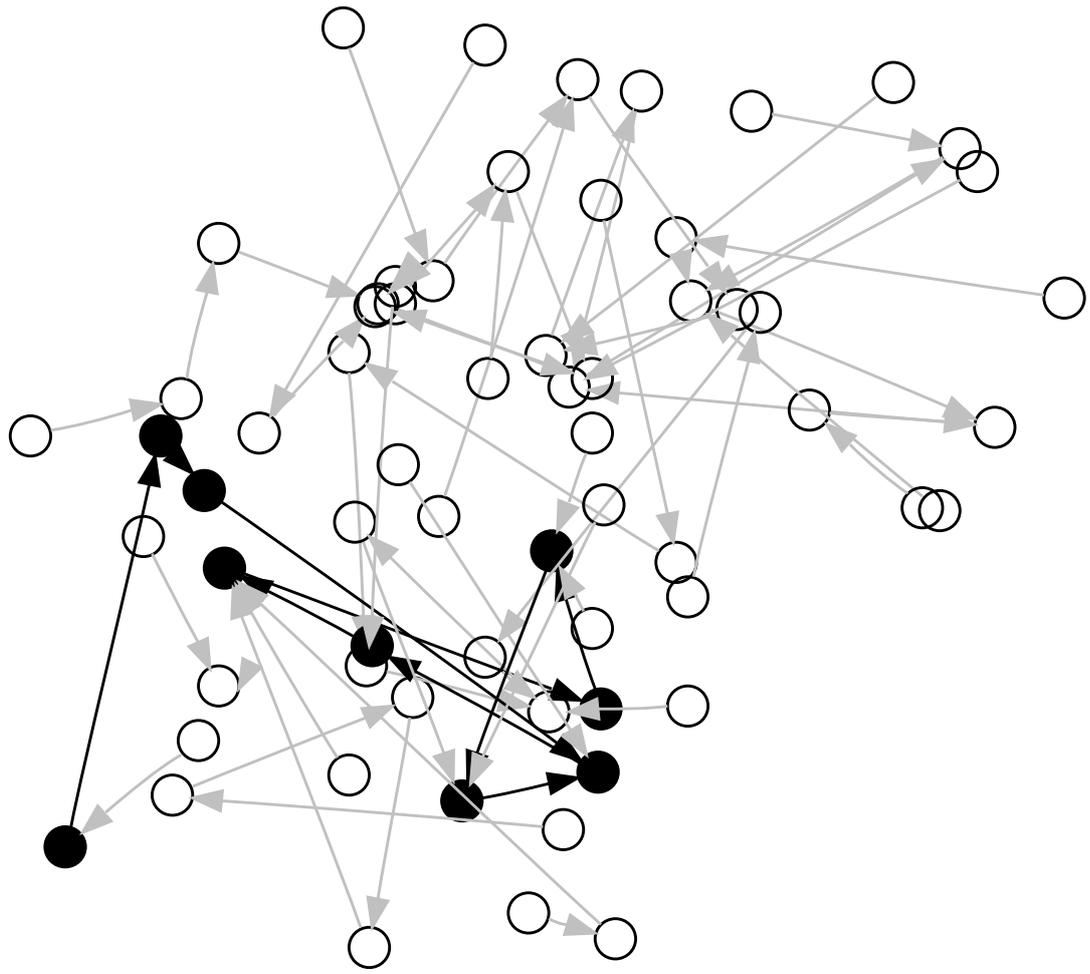


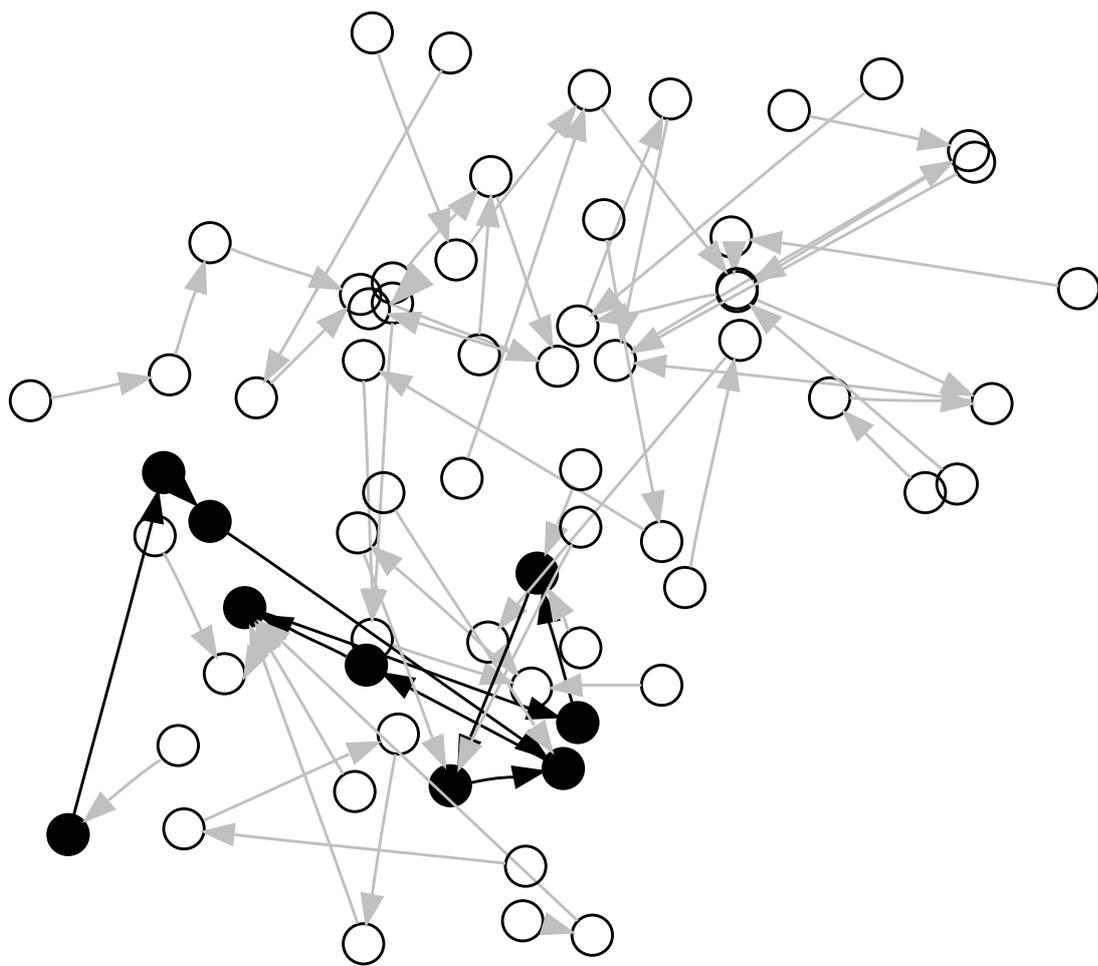


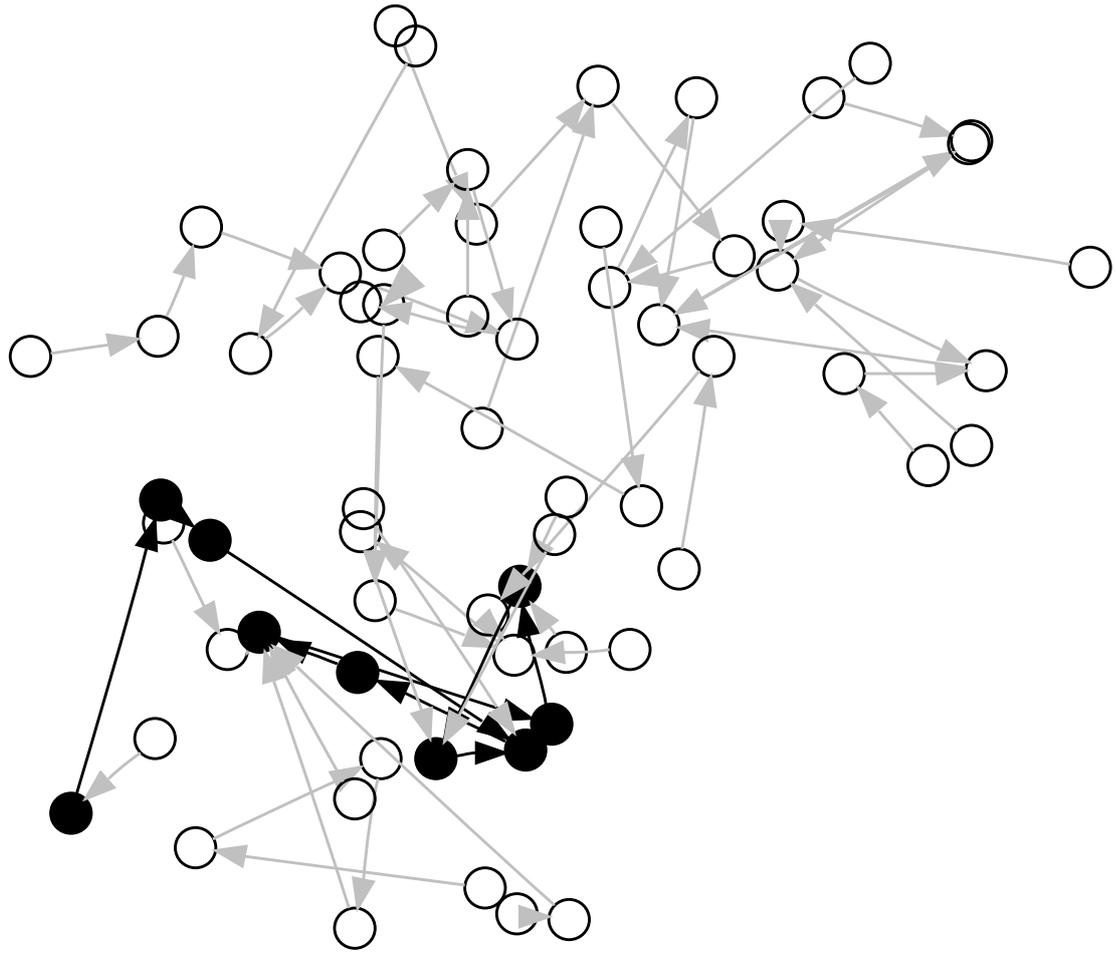


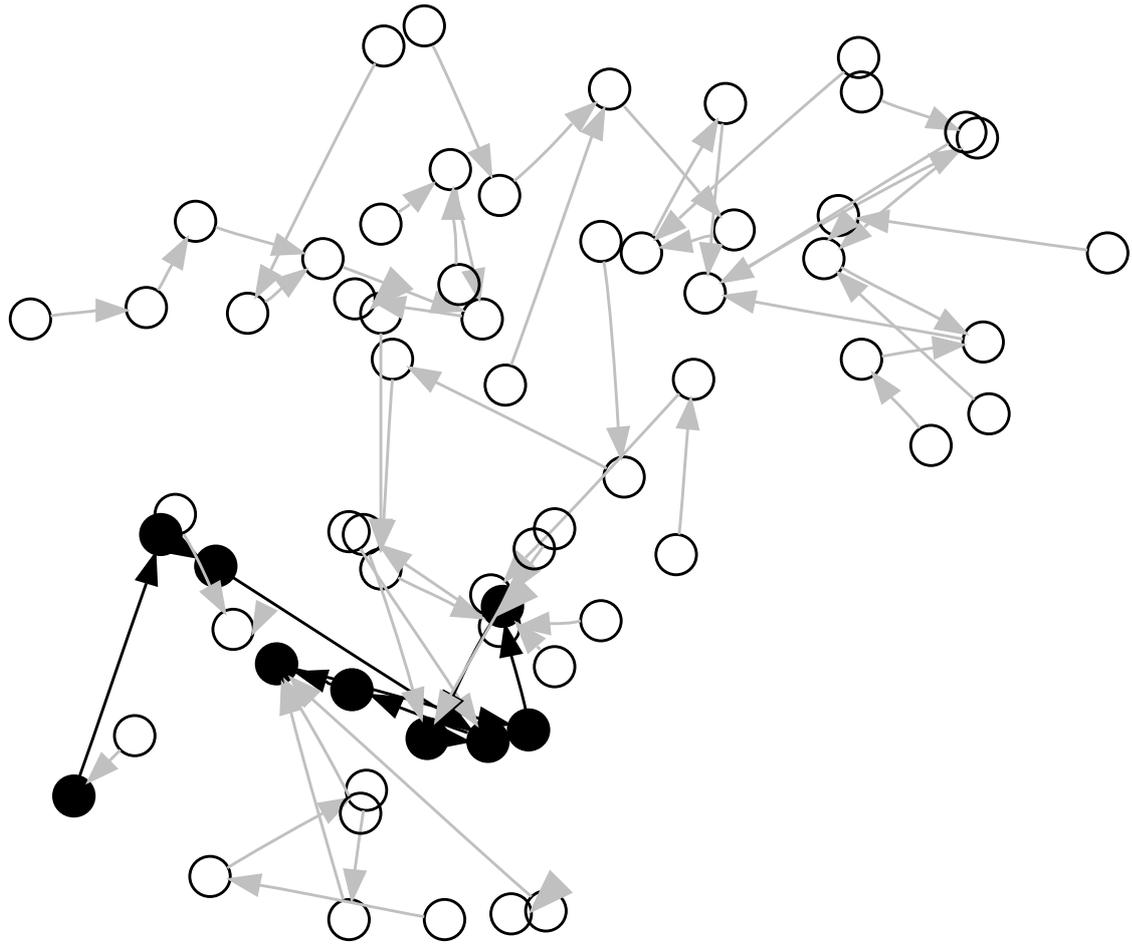


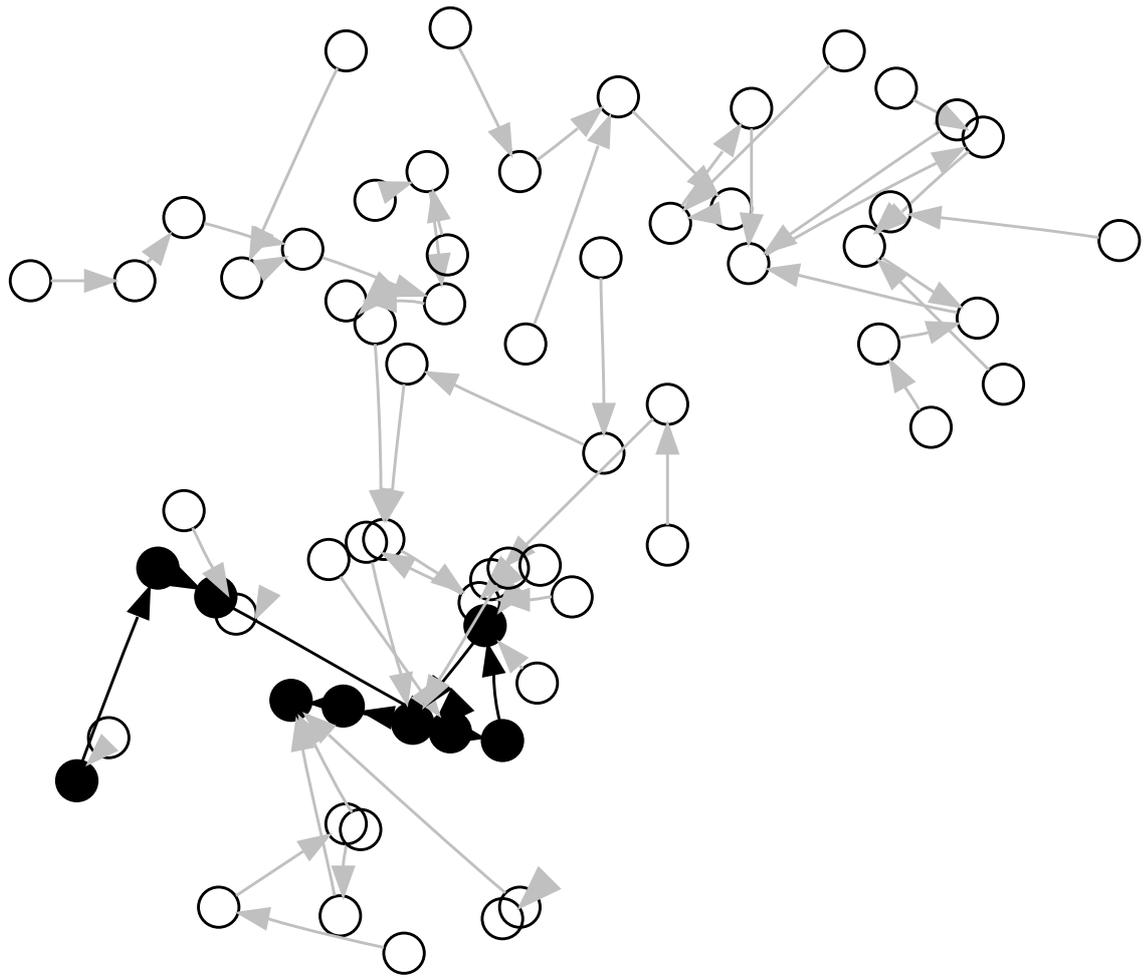


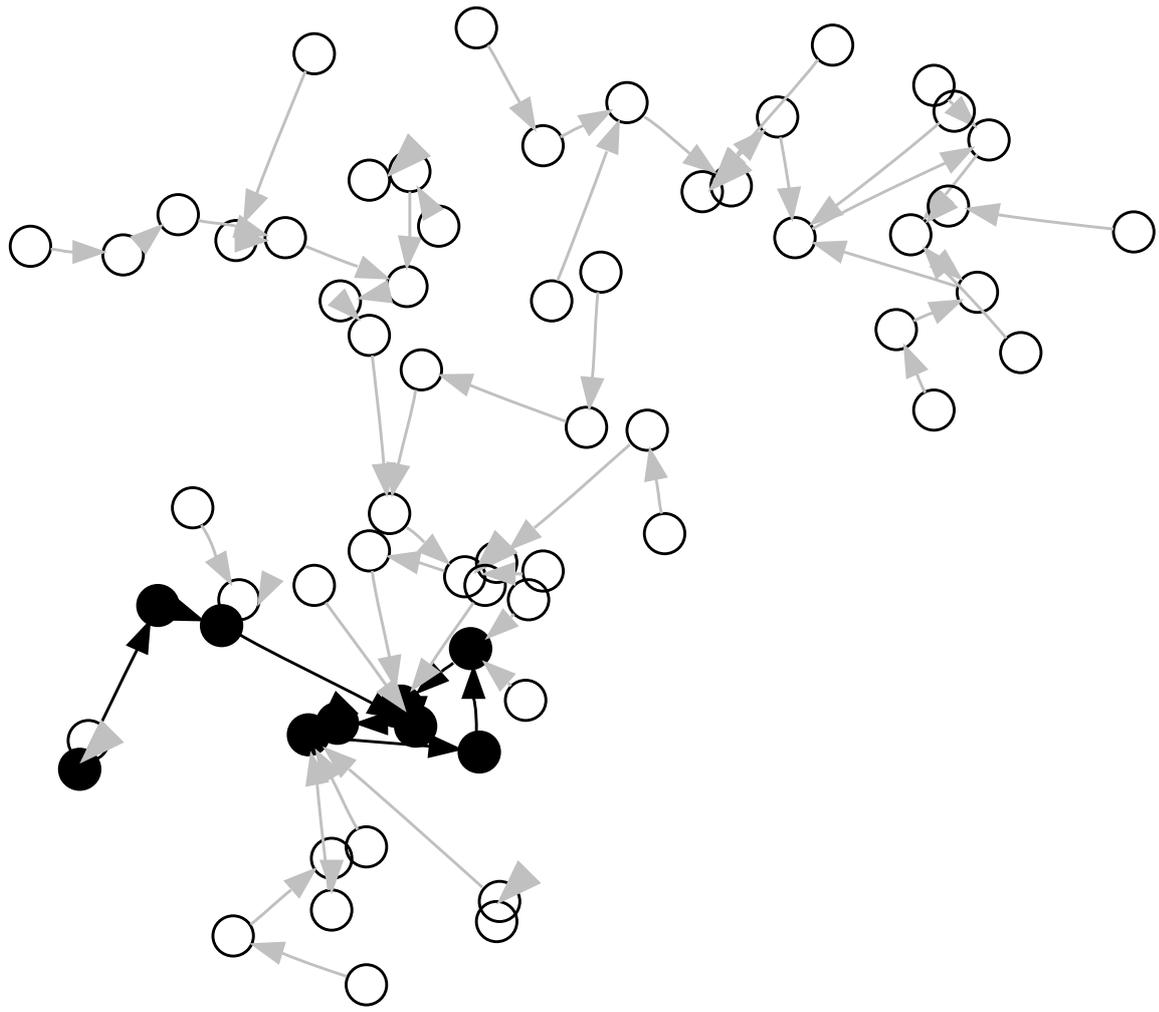


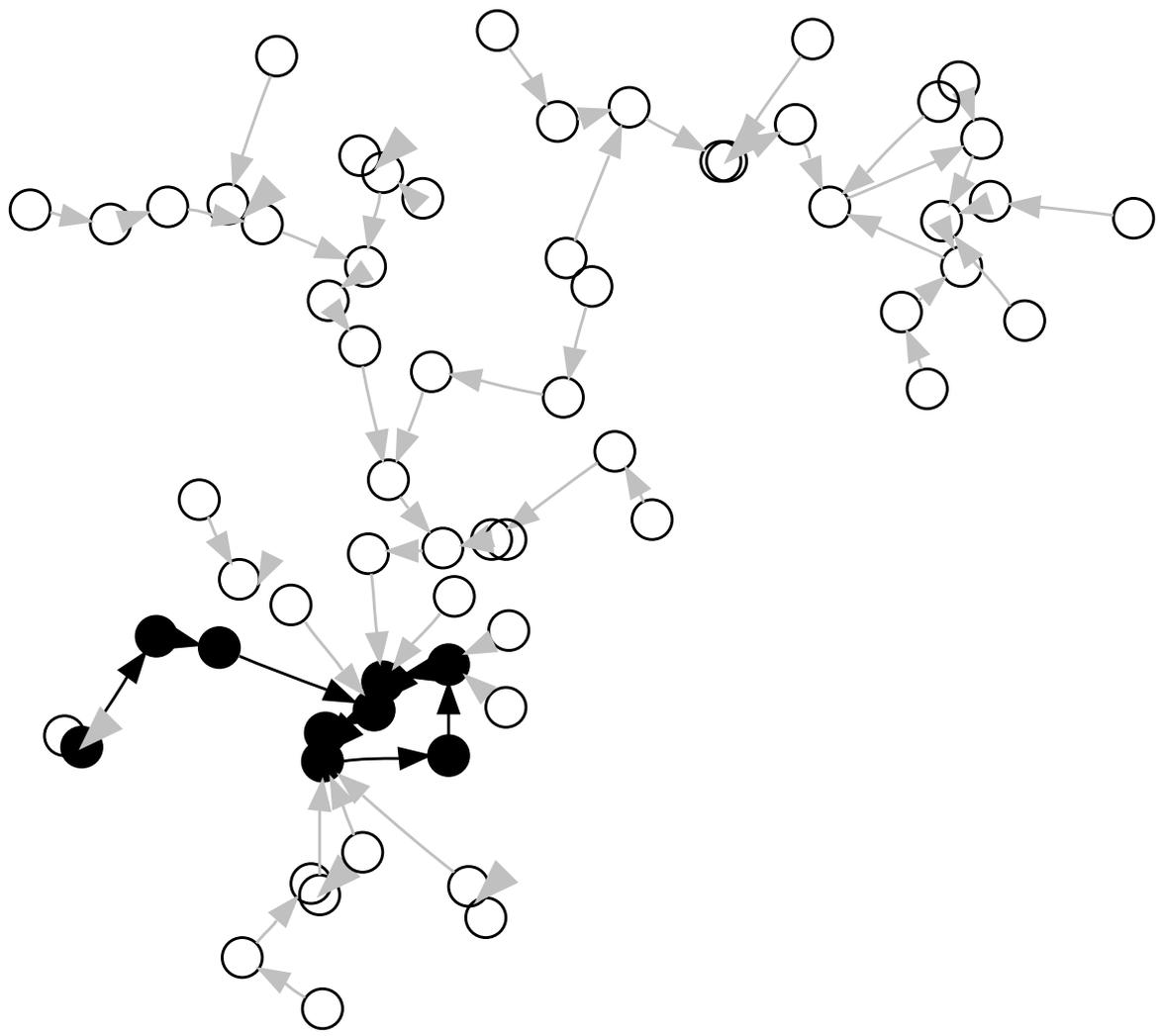


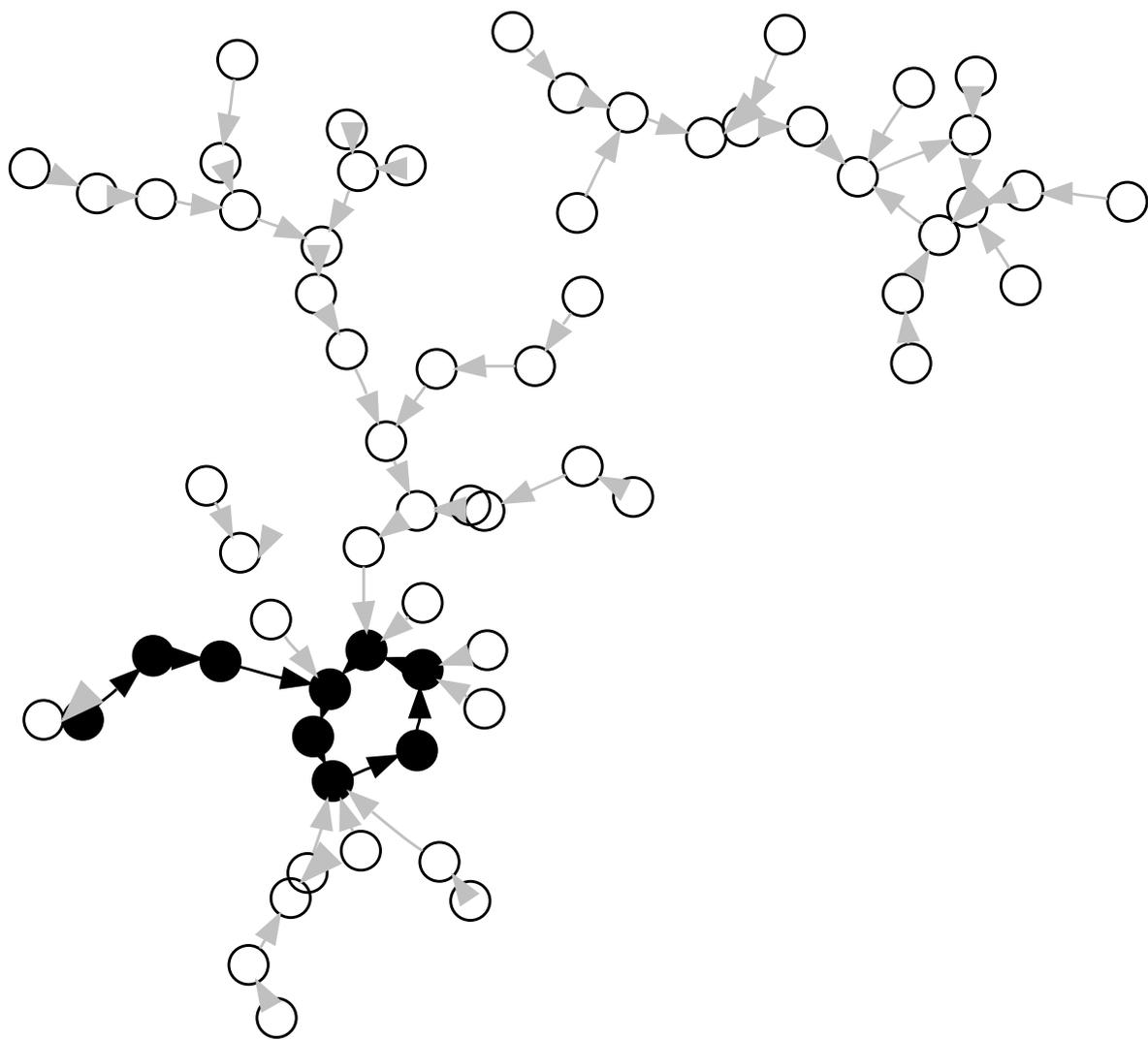


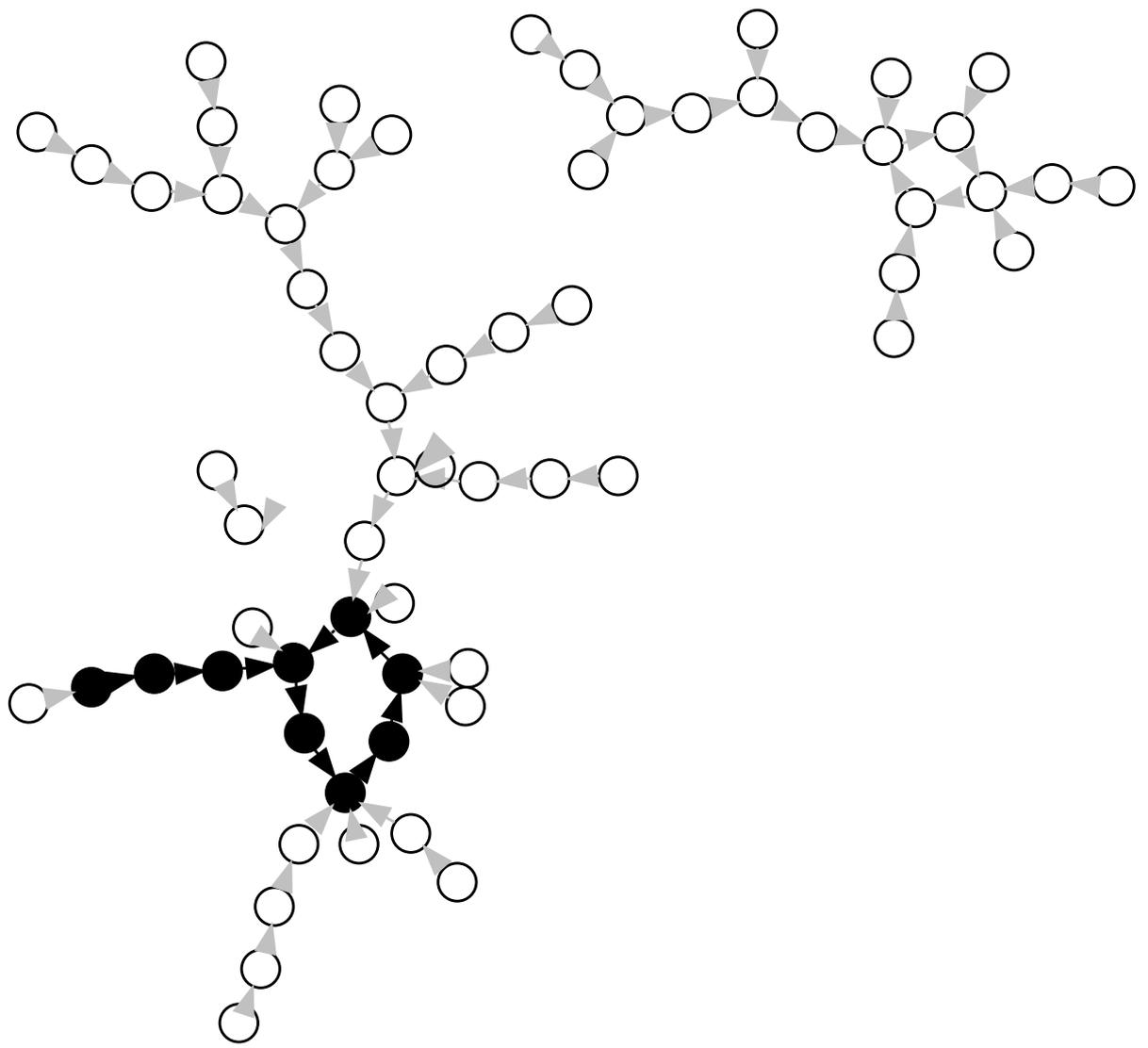












Assume that for each point
we know $a_i, b_i \in \mathbf{Z}/\ell\mathbf{Z}$
so that $W_i = a_i P + b_i Q$.

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. More on that later.

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.

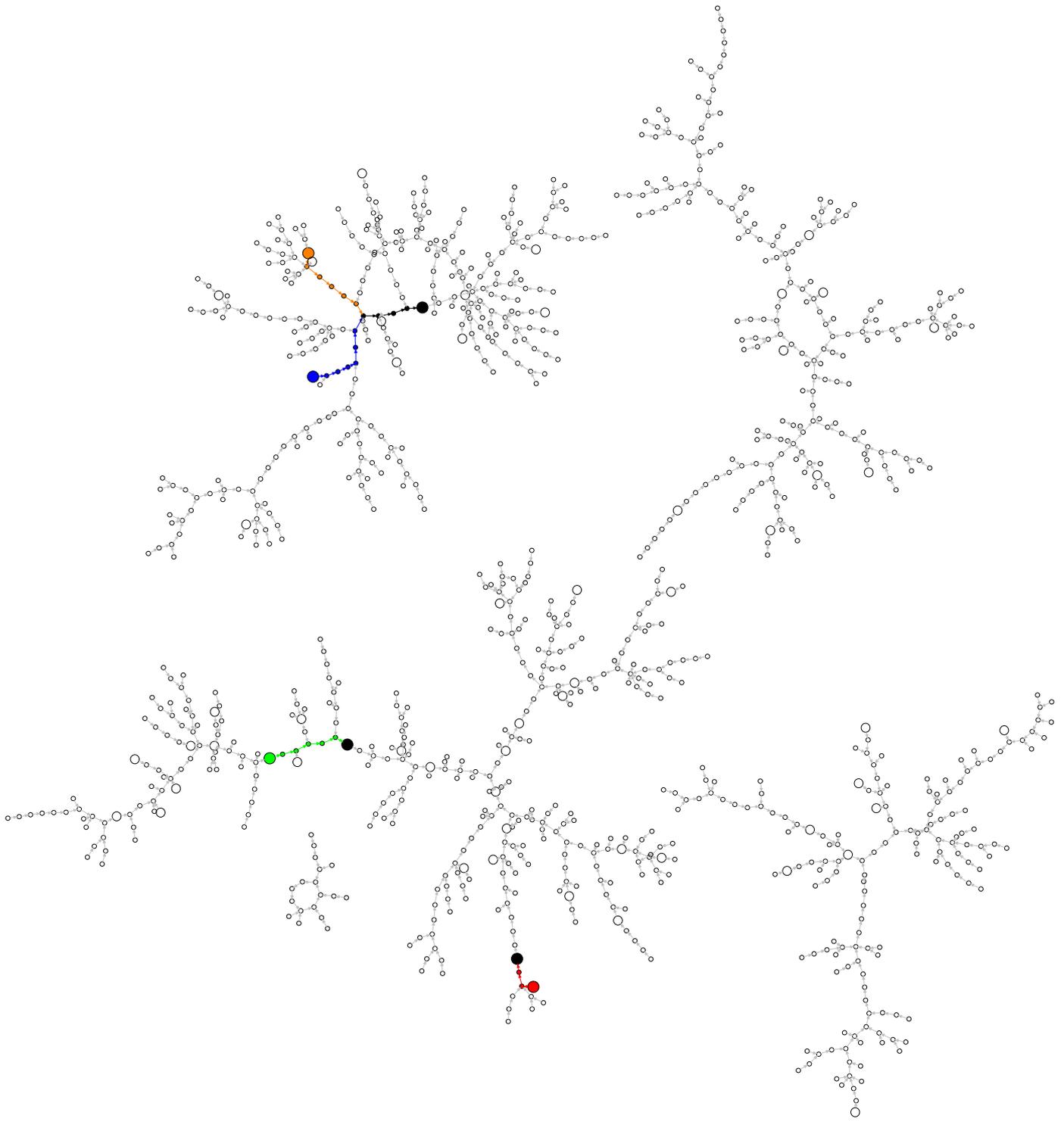
Tradeoffs are possible:

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.



Additive walks

Generic rho method requires two scalar multiplications for each iteration.

Could replace by double-scalar multiplication (see yesterday); could further merge the 2-scalar multiplications across several parallel iterations.

Additive walks

Generic rho method requires two scalar multiplications for each iteration.

Could replace by double-scalar multiplication (see yesterday); could further merge the 2-scalar multiplications across several parallel iterations.

More efficient: use *additive walk*:

Start with $W_0 = a_0 P$ and put

$$f(W_i) = W_i + c_j P + d_j Q$$

where $j = h(W_i)$.

Pollard's initial proposal:

Use $x(W_i) \bmod 3$ as h

and update:

$$W_{i+1} = \begin{cases} W_i + P & \text{for } x(W_i) \bmod 3 = 0 \\ 2W_i & \text{for } x(W_i) \bmod 3 = 1 \\ W_i + Q & \text{for } x(W_i) \bmod 3 = 2 \end{cases}$$

Easy to update a_i and b_i .

$$(a_{i+1}, b_{i+1}) = \begin{cases} (a_i + 1, b_i) & \text{for } x(W_i) \bmod 3 = 0 \\ (2a_i, 2b_i) & \text{for } x(W_i) \bmod 3 = 1 \\ (a_i, b_i + 1) & \text{for } x(W_i) \bmod 3 = 2 \end{cases}$$

Additive walk requires only one addition per iteration.

h maps from $\langle P \rangle$ to $\{0, 1, \dots, r - 1\}$, and $R_j = c_j P + d_j Q$ are precomputed for each $j \in \{0, 1, \dots, r - 1\}$.

Easy coefficient update:

$$W_i = a_i P + b_i Q,$$

where a_i and b_i are defined recursively as follows:

$$a_{i+1} = a_i + c_{h(W_i)} \text{ and}$$

$$b_{i+1} = b_i + d_{h(W_i)}.$$

Additive walks have disadvantages:

The walks are noticeably nonrandom; this means they need more iterations than the generic rho method to find a collision.

This effect disappears as r grows, but but then the precomputed table R_0, \dots, R_{r-1} does not fit into fast memory. This depends on the platform, e.g. trouble for GPUs.

More trouble with adding walks later.

Randomness of adding walks

Let $h(W) = i$ with probability p_i .

Fix a point T , and let W and W' be two independent uniform random points.

Let $W \neq W'$ both map to T .

This event occurs if there are $i \neq j$ such that simultaneously:

$$T = W + R_i = W' + R_j;$$

$$h(W) = i; h(W') = j.$$

These conditions have probability $1/\ell^2$, p_i , and p_j respectively.

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

$$\left(\sum_{i \neq j} p_i p_j \right) / \ell^2 =$$

$$\left(\sum_{i, j} p_i p_j - \sum_i p_i^2 \right) / \ell^2 =$$

$$\left(1 - \sum_i p_i^2 \right) / \ell^2.$$

This means that the probability of an immediate collision from W and W' is $(1 - \sum_i p_i^2) / \ell$, where we added over the ℓ choices of T . In the simple case that all the p_i are $1/r$, the difference from the optimal $\sqrt{\pi \ell / 2}$ iterations is a factor of

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

Various heuristics leading to standard $\sqrt{1 - 1/r}$ formula in different ways:

1981 Brent–Pollard;

2001 Teske;

2009 ECC2K-130 paper,
eprint 2009/541.

Various heuristics leading to standard $\sqrt{1 - 1/r}$ formula in different ways:

1981 Brent–Pollard;

2001 Teske;

2009 ECC2K-130 paper,
eprint 2009/541.

2010 Bernstein–Lange:

Standard formula is wrong!

There is a further slowdown

from higher-order anti-collisions:

e.g. $W + R_i + R_k \neq W' + R_j + R_l$

if $R_i + R_k = R_j + R_l$.

$\approx 1\%$ slowdown for ECC2K-130.

Eliminating storage

Usual description: each walk keeps track of a_i and b_i with $W_i = a_i P + b_i Q$.

This requires each client to implement arithmetic modulo ℓ or at least keep track of how often each R_j is used.

For distinguished points these values are transmitted to server (bandwidth) which stores them as e.g. (W_i, a_i, b_i) (space).

2009 ECC2K-130 paper:

Remember where you started.

If $W_i = W_j$ is the collision of distinguished points,

can recompute these walks

with $a_i, b_i, a_j,$ and b_j ;

walk is deterministic!

Server stores 2^{45} distinguished points; only needs to know coefficients for 2 of them.

Our setup: Each walk remembers seed; server stores distinguished point and seed.

Saves time, bandwidth, space.

Negation and rho

$W = (x, y)$ and $-W = (x, -y)$

have same x -coordinate.

Search for x -coordinate collision.

Search space for collisions is

only $\lceil \ell/2 \rceil$; this gives factor $\sqrt{2}$

speedup ... if $f(W_i) = f(-W_i)$.

To ensure $f(W_i) = f(-W_i)$:

Define $j = h(|W_i|)$ and

$f(W_i) = |W_i| + c_j P + d_j Q$.

Define $|W_i|$ as, e.g., lexicographic minimum of $W_i, -W_i$.

This negation speedup

is textbook material.

Problem: this walk can run into fruitless cycles!

Example: If $|W_{i+1}| = -W_{i+1}$ and $h(|W_{i+1}|) = j = h(|W_i|)$

then $W_{i+2} = f(W_{i+1}) = -W_{i+1} + c_j P + d_j Q = -(|W_i| + c_j P + d_j Q) + c_j P + d_j Q = -|W_i|$ so $|W_{i+2}| = |W_i|$

so $W_{i+3} = W_{i+1}$

so $W_{i+4} = W_{i+2}$ etc.

If h maps to r different values then expect this example to occur with probability $1/(2r)$ at each step.

Known issue, not quite textbook.

1999 Gallant–Lambert–Vanstone
“Improving the parallelized Pollard
lambda search on anomalous
binary curves” :

“For example, the cycle could be
traversed, the lexicographically
least label identified, and a
modified iteration taking us out
of the cycle could be applied at
the point or equivalence class
corresponding to this identified
label.”

1999 Duursma–Gaudry–Morain
“Speeding up the discrete log
computation on curves with
automorphisms” :

“If the cycle is $R_1 \mapsto R_2 \mapsto \dots \mapsto R_t$, we want to get out of it in a *symmetric* way ... Our version is to sort the points R_i to obtain S_1, S_2, \dots, S_t and start again, say, from $R = \bigoplus_{i=1}^t [i^i + 1]S_i$. Anything that breaks linearity would be convenient.”

e.g. Sort 2-cycle,
obtaining $S_1 \leq S_2$.

Duursma–Gaudry–Morain “start
again, say, from” $2S_1 + 5S_2$.

Gallant–Lambert–Vanstone
keep only S_1 and
apply a “modified iteration”
but are vague about
the choice of modified iteration.
Maybe $2S_1$?

2009 Bos–Kaihara–Kleinjung–
Lenstra–Montgomery use $2S_1$.

Current ECDL record:

2009.07 Bos–Kaihara–
Kleinjung–Lenstra–Montgomery
“PlayStation 3 computing
breaks 2^{60} barrier:
112-bit prime ECDLP solved” .

Standard curve over \mathbf{F}_p

where $p = (2^{128} - 3)/(11 \cdot 6949)$.

Current ECDL record:

2009.07 Bos–Kaihara–
Kleinjung–Lenstra–Montgomery
“PlayStation 3 computing
breaks 2^{60} barrier:
112-bit prime ECDLP solved” .

Standard curve over \mathbf{F}_p

where $p = (2^{128} - 3)/(11 \cdot 6949)$.

“We did not use
the common negation map
since it requires branching
and results in code that runs
slower in a SIMD environment.”

All modern CPUs are SIMD.

2009.07 Bos–Kaihara–Kleinjung–
Lenstra–Montgomery “On the
security of 1024-bit RSA and 160-
bit elliptic curve cryptography” :

Group order $q \approx p$;

“expected number of iterations”

is “ $\sqrt{\frac{\pi \cdot q}{2}} \approx 8.4 \cdot 10^{16}$ ”; “we

do not use the negation map”;

“456 clock cycles per iteration

per SPU”; “24-bit distinguishing

property” \Rightarrow “260 gigabytes” .

“The overall calculation

can be expected to take

approximately **60 PS3 years.**”

2009.09 Bos–Kaihara–
Montgomery “Pollard rho
on the PlayStation 3” :

“Our software implementation is optimized for the SPE . . . the computational overhead for [the negation map], **due to the conditional branches required to check for fruitless cycles [13]**, results (in our implementation on this architecture) in an overall performance degradation.”

“[13]” is 2000 Gallant–Lambert–
Vanstone.

2010.07 Bos–Kleinjung–Lenstra

“On the use of the negation map in the Pollard rho method” :

“If the Pollard rho method is parallelized in SIMD fashion, it is a challenge to achieve any speedup at all. . . . Dealing with cycles entails administrative overhead and branching, which cause a non-negligible slowdown when running multiple walks in SIMD-parallel fashion. . . .

[This] is a major obstacle to the negation map in SIMD environments.”

Bernstein, Lange, Schwabe
(PKC 2011):

Our software solves
random ECDL on the same curve
(with no precomputation)
in 35.6 PS3 years on average.

For comparison:

Bos–Kaihara–Kleinjung–Lenstra–
Montgomery software
uses 65 PS3 years on average.

Bernstein, Lange, Schwabe
(PKC 2011):

Our software solves
random ECDL on the same curve
(with no precomputation)
in 35.6 PS3 years on average.

For comparison:

Bos–Kaihara–Kleinjung–Lenstra–
Montgomery software
uses 65 PS3 years on average.

First big speedup:

We use the negation map.

Second speedup: Fast arithmetic.

Bos–Kleinjung–Lenstra say that “on average more elliptic curve group operations are required per step of each walk. This is unavoidable” etc.

Specifically: If the precomputed additive-walk table has r points, need 1 extra doubling to escape a cycle after $\approx 2r$ additions.

And more: “cycle reduction” etc.

Bos–Kleinjung–Lenstra say that the benefit of large r is “wiped out by cache inefficiencies.”

Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed.

See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.

Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed.

See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.

So what to do?

Choose a big r , e.g. $r = 2048$.

$1/(2r) = 1/4096$ small;

cycles infrequent.

Define $|(x, y)|$ to mean

(x, y) for $y \in \{0, 2, 4, \dots, p-1\}$

or

$(x, -y)$ for $y \in \{1, 3, 5, \dots, p-2\}$.

Precompute points

R_0, R_1, \dots, R_{r-1} as known

random multiples of P .

Define $|(x, y)|$ to mean
 (x, y) for $y \in \{0, 2, 4, \dots, p-1\}$
or
 $(x, -y)$ for $y \in \{1, 3, 5, \dots, p-2\}$.

Precompute points

R_0, R_1, \dots, R_{r-1} as known

random multiples of P . Here you
can do full scalar multiplication in
inversion-free coordinates!

Start each walk at a point

$W_0 = |b_0 Q|$, where b_0 is chosen
randomly.

Compute W_1, W_2, \dots as $W_{i+1} =$
 $|W_i + R_{h(W_i)}|$.

Occasionally, every w iterations, check for fruitless cycles of length 2.

For those cases change the definition of W_i as follows:

Compute W_{i-1} and check whether $W_{i-1} = W_{i-3}$.

If $W_{i-1} \neq W_{i-3}$, put $W_i = W_{i-1}$.

If $W_{i-1} = W_{i-3}$, put

$$W_i = |2 \min\{W_{i-1}, W_{i-2}\}|,$$

where min means

lexicographic minimum.

Doubling the point

makes it escape the cycle.

Cycles of length 4, 6, or 12 occur far less frequently.

Cycles of length 4, or 6 are detected when checking for cycles of length 12; so skip individual ones.

Same way of escape:

define $W_i =$

$$|2\min\{W_{i-1}, W_{i-2}, W_{i-3}, W_{i-4}, \\ W_{i-5}, W_{i-6}, W_{i-7}, W_{i-8}, \\ W_{i-9}, W_{i-10}, W_{i-11}, W_{i-12}\}|$$

if trapped

and $W_i = W_{i-1}$ otherwise.

Do not store all these points!

When checking for cycle,
store only potential entry point
 W_{i-13} (one coordinate, for
comparison) and the
smallest point encountered since
(to escape).

For large DLP

look for larger cycles;

in general, look for

fruitless cycles of even lengths

up to $\approx (\log \ell) / (\log r)$.

How to choose w ?

Fruitless cycles of length 2 appear with probability $\approx 1/(2r)$.

These cycles persist until detected.

After w iterations, probability of cycle $\approx w/(2r)$, wastes $\approx w/2$ iterations (on average) if it does appear.

Do not choose w as small as possible!

If a cycle has *not* appeared then the check wastes an iteration.

The overall loss is approximately $1 + w^2/(4r)$ iterations out of w .

To minimize the quotient

$1/w + w/(4r)$ we take $w \approx 2\sqrt{r}$.

Cycles of length $2c$ appear with probability $\approx 1/r^c$,

optimal checking frequency is $\approx 1/r^{c/2}$.

Loss rapidly disappears as c increases.

Can use lcm of cycle lengths to check.

Concrete example: 112-bit DLP

Use $r = 2048$. Check for 2-cycles every 48 iterations.

Check for larger cycles much less frequently.

Unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations.

Choice of r has big impact!

$r = 512$ calls for checking for 2-cycles every 24 iterations.

In general, negation overhead \approx doubles when table size is reduced by factor of 4.

Why are we confident this works?

We only have one PlayStation 3, not the 200 that Lausanne has, and we want to wait for 36 years to show that we actually compute the right thing.

Why are we confident this works?

We only have one PlayStation 3, not the 200 that Lausanne has, and we want to wait for 36 years to show that we actually compute the right thing.

Can produce scaled versions:

Use *same* prime field

(so that we can compare the field arithmetic)

and same curve shape

$$y^2 = x^3 - 3x + b$$

but vary b to get curves with small subgroups.

This produces other curves, and many of those have smaller order subgroups.

Specify DLP in subgroup of size 2^{50} , or 2^{55} , or 2^{60} and show that the actual running time matches the expectation.

And that DLP is correct.

We used same property for a point to be distinguished as in big attack; probability is 2^{-20} .

Need to watch out that walks do not run into rho-type cycles (artefact of small group order).

We aborted overlong walks.