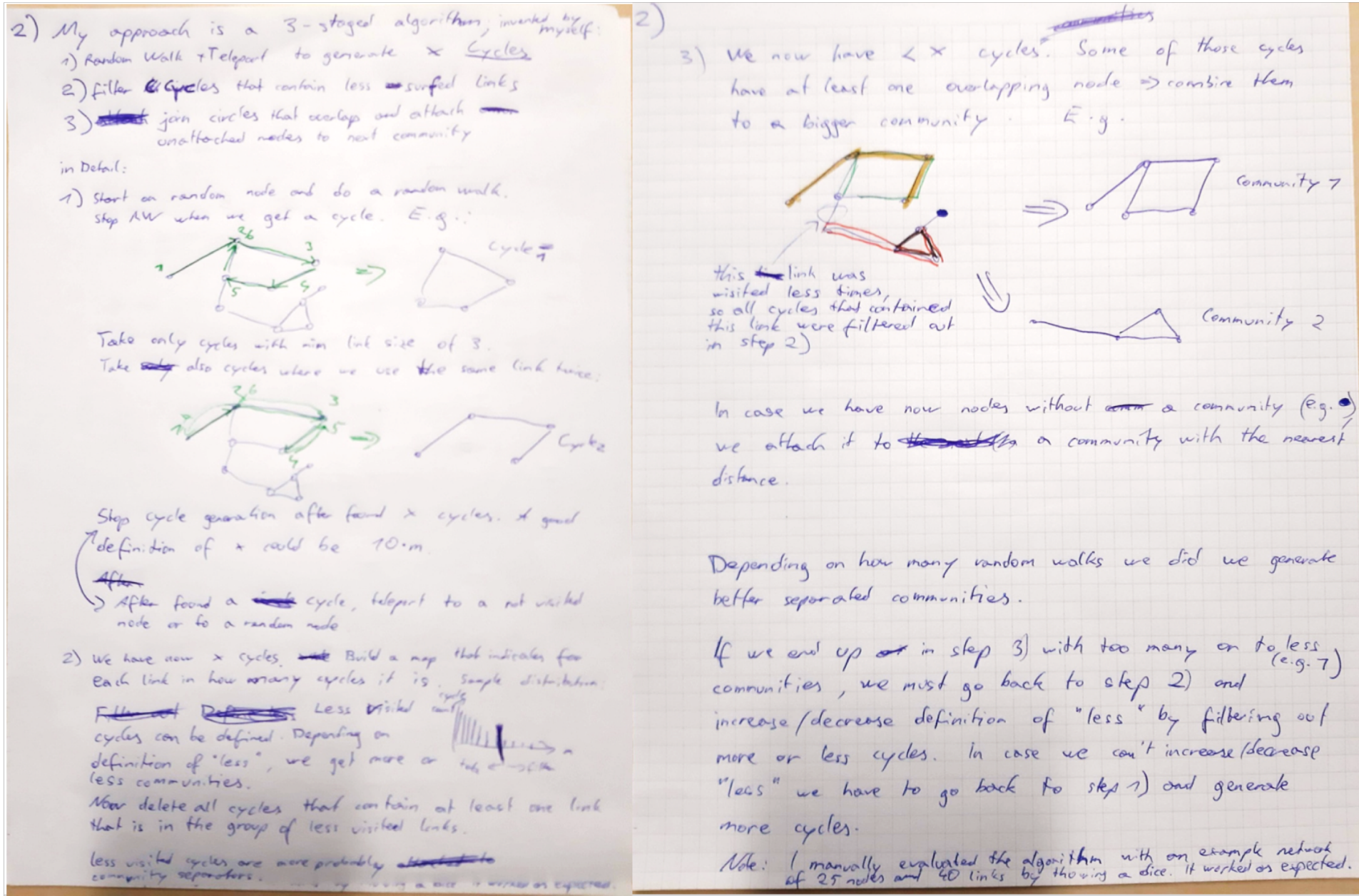
1. Motivation

NetSci HW3.2:
extract communities from networks with random walk



2) My approach is a 3-staged algorithm, invented by myself:
1) Random Walk + Teleport to generate x Cycles
2) filter Cycles that contain less surfed links
3) join circles that overlap and attach unattached nodes to next community

in Detail:

1) Start on random node and do a random walk. Stop RW when we get a cycle. E.g.:

Cycle = 7

Take only cycles with min link size of 3.
Take also cycles where we use the same link twice:

Cycle 2

Stop cycle generation after found x cycles. A good definition of x could be 10·n.

After found a cycle, teleport to a not visited node or to a random node

2) We have now x cycles. Build a map that indicates for each link in how many cycles it is. Sample distribution:

Less visited cycles can be defined. Depending on definition of "less", we get more or less communities.
Now delete all cycles that contain at least one link that is in the group of less visited links.

less visited cycles are more probably community separators.

2)

3) We now have < x cycles. Some of those cycles have at least one overlapping node ⇒ combine them to a bigger community. E.g.

Community 7

this link was visited less times, so all cycles that contained this link were filtered out in step 2)

Community 2

In case we have now nodes without a community (e.g.) we attach it to a community with the nearest distance.

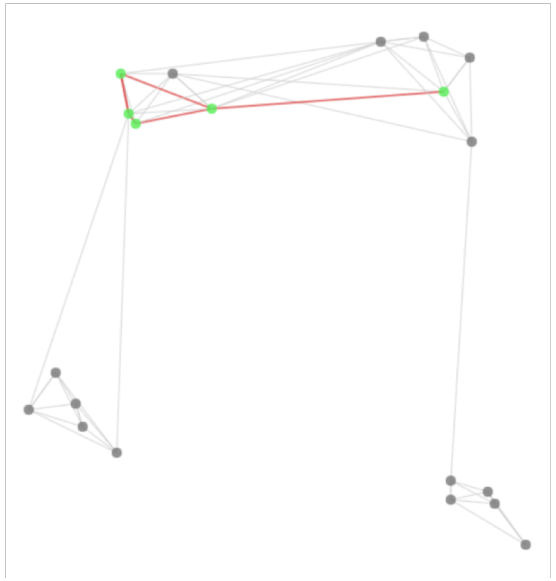
Depending on how many random walks we did we generate better separated communities.

If we end up in step 3) with too many or to less communities, we must go back to step 2) and increase/decrease definition of "less" by filtering out more or less cycles. In case we can't increase/decrease "less" we have to go back to step 1) and generate more cycles. (e.g. 7)

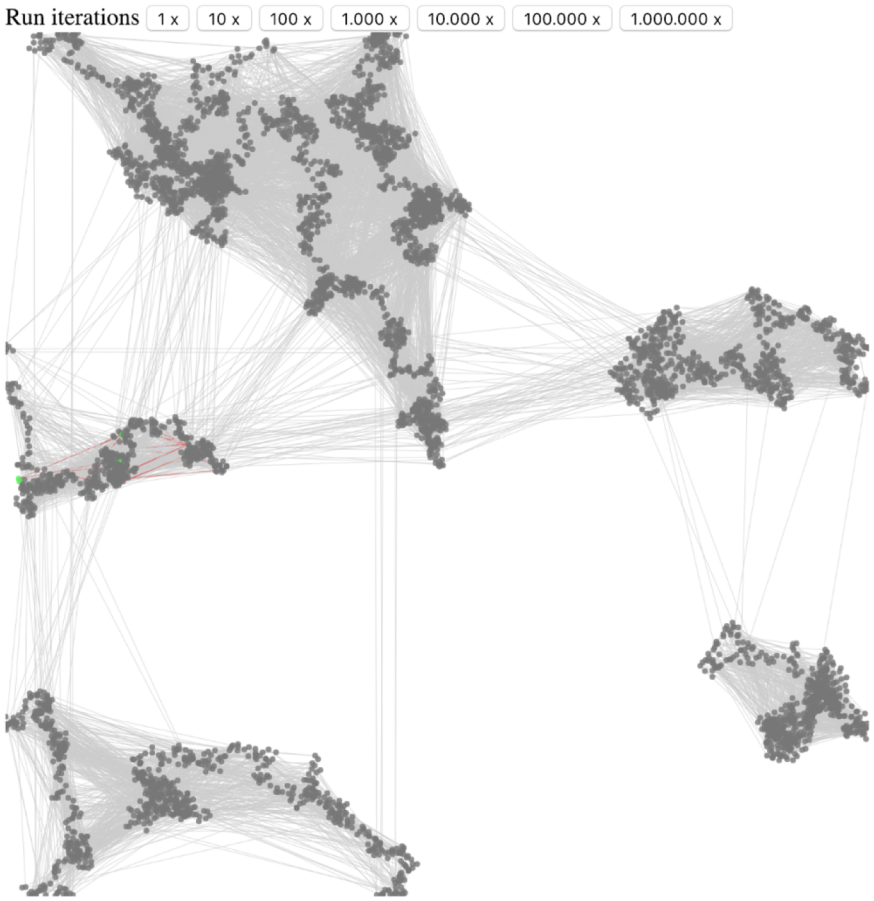
Note: I manually evaluated the algorithm with an example network of 25 nodes and 40 links by throwing a dice. It worked as expected.

2. Methodology: JavaScript Algorithm Implementation

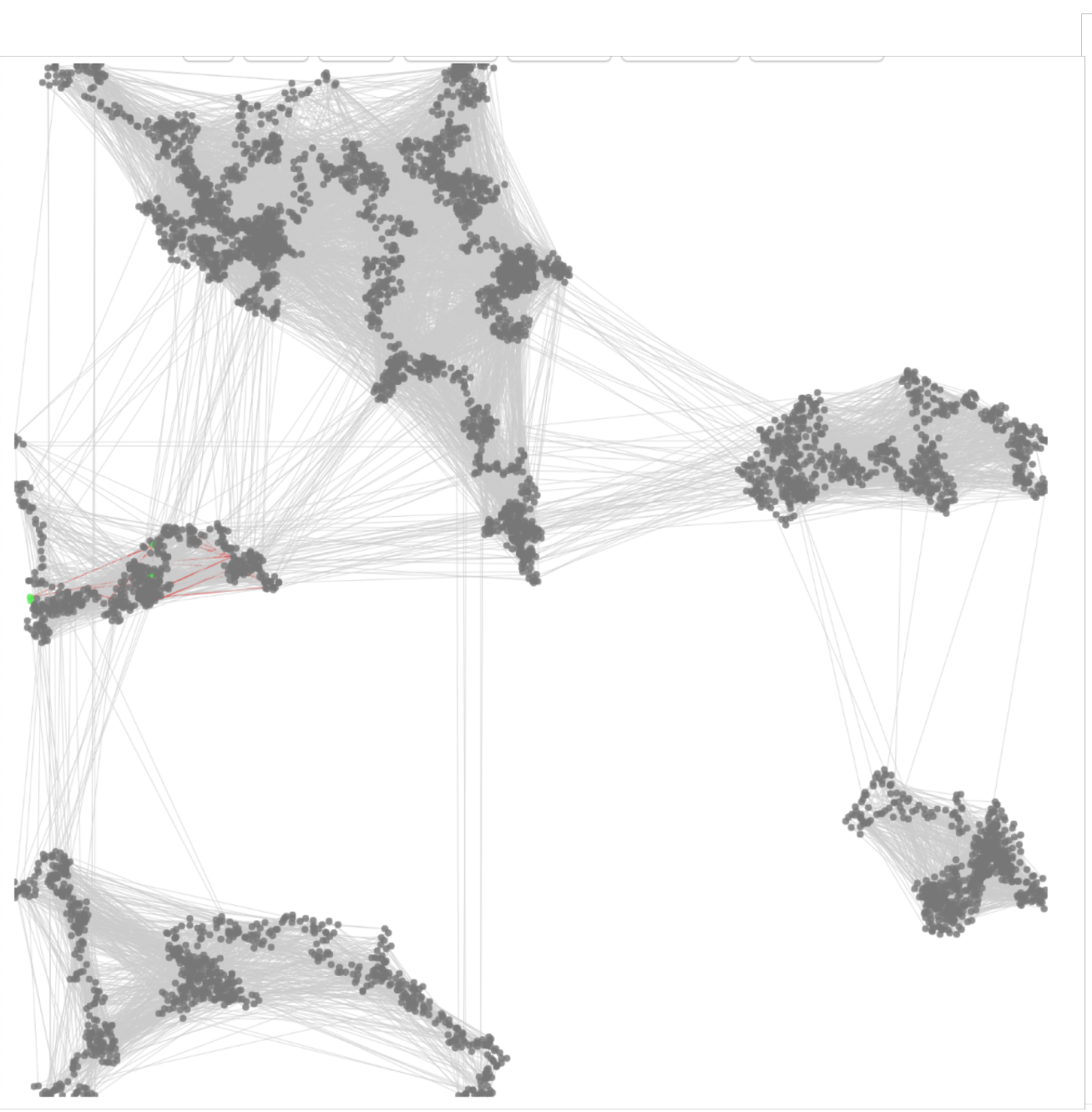
- Generate Network with 50 – 5.000 nodes and connections
  Coordinates and distance only for generation & visualization
- Algorithm: Detect communities
  1) Find circles
  2) Measure connection usage in circles
  3) Remove less used connections
  → Communities
- Visualize and validate algorithm workflow via canvas
- Measure runtime speed

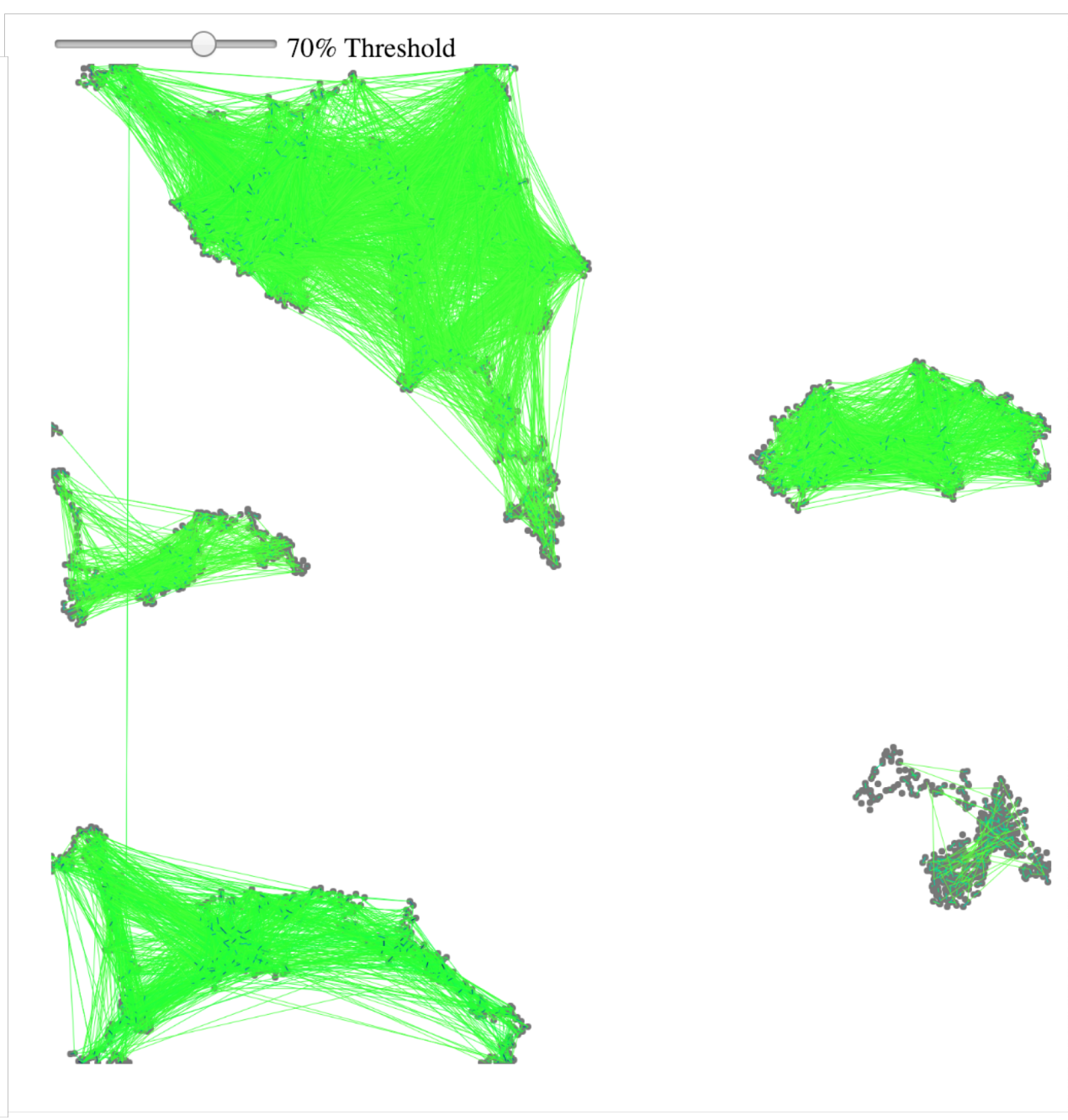


3. Experimental setup




```
5000 nodes, 25000 connections
10 iterations took 16ms
100000 iterations took 38122ms
100000 iterations took 37849ms
100000 iterations took 37929ms
10000 iterations took 4073ms
10000 iterations took 3823ms
10000 iterations took 3811ms
```




70% Threshold

4. Results

- Community detection is as expected, depends on generated network
- Runtime Speed:

  On average better than O(nodeCount) and O(connectionCount)

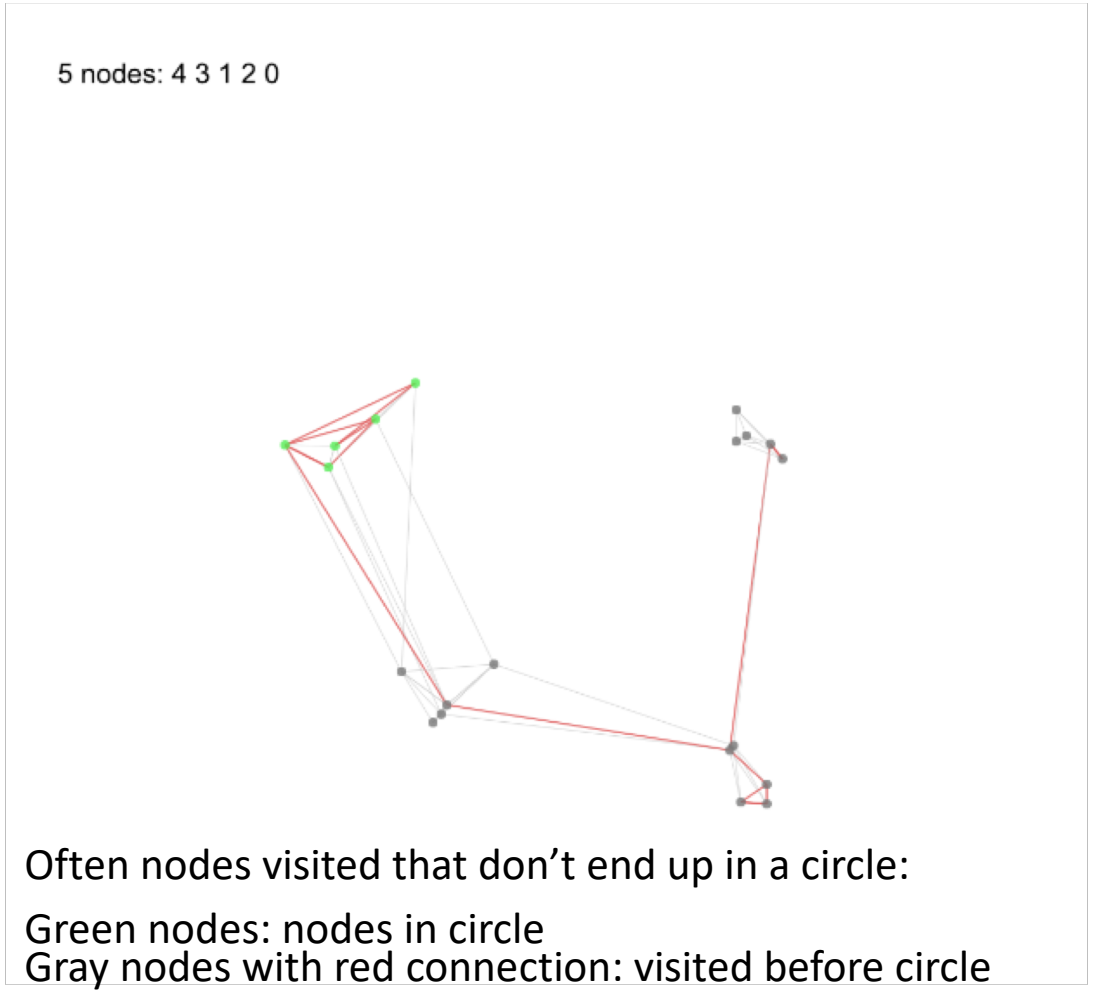
  Worst case: Infinity (random walk)

nodes	connections	ms per 100.000 runs	us per node	us per connection
50	327	1400	28	4,281345566
50	308	1500	30	4,87012987
50	303	1400	28	4,620462046
100	747	1400	14	1,87416332
100	896	2000	20	2,232142857
100	681	1500	15	2,202643172
200	1719	2250	11,25	1,308900524
200	1753	2900	14,5	1,654306902
200	1551	2200	11	1,418439716
500	4047	3900	7,8	0,963676798
500	3287	6100	12,2	1,855795558
500	5856	3400	6,8	0,580601093
1000	7673	19300	19,3	2,515313437
1000	5040	11000	11	2,182539683
1000	11024	7000	7	0,634978229
2000	6049	11000	5,5	1,818482394
2000	6390	13500	6,75	2,112676056
2000	8669	14300	7,15	1,649555889
2000	13098	16000	8	1,221560544
2000	16983	17300	8,65	1,018665725
2000	22044	20700	10,35	0,939031029
2000	27625	20000	10	0,7239819
5000	25000	38000	7,6	1,52

5. Discussion

- “Interesting” network generation was more complex than community detection
- Simulation with “real” networks needed
- Validation if “correct” communities are found
- Postprocessing needed
- Data structures and algorithm not optimized
- Algorithm is very friendly to multithreading and can be executed distributed

- Do you want to see the algorithm work?
  → Live Demo



Often nodes visited that don’t end up in a circle:

Green nodes: nodes in circle
Gray nodes with red connection: visited before circle