Appendix A: Calculation of Pooled Competence

A voter i has neighbourhood $N(i)$. Let the cardinality of that neighbourhood be $n(i)$. Let the number of Elite types in $N(i)$ be $e(i)$ and the number of Mass types be $m(i)$. The pooled competence of the neighborhood for voter i depends on the number of Mass and Elite voters in the neighbourhood. We are interested in the likelihood of majorities that identify the correct interest of i. Let there be a set K of all possible ordered pairs $\langle v_E, v_M \rangle$ with v_E representing the number Elite types in $N(i)$ voting in the interest of the Elites and v_M the Mass types voting in the interest of the Masses under the constraints that $0 \leq v_E \leq e(i)$, $0 \leq v_M \leq m(i)$. Each tuple represents a possible outcome of Elite types and Mass types voting in a specific way and the set of all tuples represents all possible ways for the Elites and the Mass votes in the neighbourhood to go.

Case 1: Voter i is a Mass Type

In this case we are interested in a subset of K, namely all elements in which the Mass interest gains a majority. Let this subset be

$$O_M = \left\{ \text{all } \langle v_E, v_M \rangle \in K : \ e(i) - v_E + v_M > \frac{n(i)}{2} \right\}.$$

Call the event of a majority for the Masses in that neighbourhood MW. For the calculations to follow, it is useful to recall the standard binomial formula for the probability of x successes out of n draws with success probability p:

$$P(n, x, p) = \binom{n}{x} p^x (1-p)^{n-x}.$$

The probability of all possible vote combinations from the Elites and Mass types in such that the Masses win is:

$$\Pr(\mathrm{MW}) = \sum_{\text{all } \langle v_E, v_M \rangle \in O_M} P(e(i), v_E, p_E) \times P(m(i), v_M, p_M).$$

Since tied outcomes are decided by coin toss, we need to calculate the probability of making the correct choice for the Masses by coin toss. Call this event T. We are now interested in the subset O_T of K that leads to ties:

$$M_T = \left\{ \text{all } \langle v_E, v_M \rangle \in K : \ e(i) - v_E + v_M = \frac{n(i)}{2} \right\}.$$

The probability of a correct vote by coin toss after a tie is:

$$\Pr(\mathrm{T}) = \frac{1}{2} \sum_{\text{all } \langle v_E, v_M \rangle \in O_T} P(e(i), v_E, p_E) \times P(m(i), v_M, p_M).$$

The pooled competence is $\Pr(\mathrm{MW}) + \Pr(\mathrm{T})$.

Case 2: Voter i as an Elite Type

The same reasoning applied symmetrically. The subset O_E consists of all tuples of Elite and Mass votes voting according to their true interest in

which the Elites obtain a majority:

$$O_E = \left\{ \text{all } \langle v_E, v_M \rangle \in K : \; v_E + m(i) - v_M > \frac{n(i)}{2} \right\}.$$

Call the event of a majority for the Elites in that neighbourhood EW. The likelihood of this occuring is calculated as above:

$$\Pr(\text{EW}) = \sum_{\text{all } \langle v_E, v_M \rangle \in O_E} P(e(i), v_E, p_E) \times P(m(i), v_M, p_M).$$

Again, results from tie-breaking coin tosses need to be taken into account. $\Pr(\text{T})$ is determined just as above. The pooled competence is $\Pr(\text{EW}) + \Pr(\text{T})$.

Appendix B: Python Code of Main Routines

In this appendix I reproduce the main part of the Python 3 code to calculate the results. This code is based on one simulation with memory. I have omitted auxilliary code to count votes, record and plot results.

```
import networkx as nx
import random as rd
from collections import Counter
from collections import deque

def modes(values):
    """function to return list of all modal values"""
    count= Counter(values)
    best = max(count.values())
    return [ k for k,v in count.items() if v == best]

def indiv_opin(c):
    """return opinion according to competence c """
    return int(round(rd.random() + c - 0.5))

def group_opin():
    """ add opinion attribute for all nodes"""
    for n in network.nodes():
        network.node[n]['opinion'] = indiv_opin(comp_vector[n])

def vote_winner(votes):
    """for any given opinions or votes, determine winner, break ties by
    random choice"""
    return rd.choice(modes(votes))

def votes():
    """determine nbh including ego, collect all opinions, vote for winner,
    break tie randomly, set vote attribute for all nodes"""
    for n in network.nodes():
        nb = list(network[n]) + [n]
```

```
        nb_v = [network.node[i]['opinion'] for i in nb]
        # determine winner and break ties by random choice
        w = rd.choice(modes(nb_v))
        network.node[n]['vote'] = w

def mem_update():
    """ go through all edges and register the opinions of all neighbors in the
    memory attribute dictionary of network. Keep memory as a deque of length
    memlen so that old opinions are forgotten"""
    for e in network.edges():
        #check if there is a dict entry in memory of first node for 2nd node
        if e[1] in network.node[e[0]]['memory']:
            #if yes, then append opinion of 2nd node to memory of 1st node
            network.node[e[0]]['memory'][e[1]].append(network.node[e[1]]['opinion'])
        #otherwise create deque with that opinion, set deque max length
        else:
            network.node[e[0]]['memory'][e[1]] = deque(
                    {network.node[e[1]]['opinion']},
                    maxlen=network.node[e[0]]['memlen'])
        if e[0] in network.node[e[1]]['memory']:
            network.node[e[1]]['memory'][e[0]].append(network.node[e[0]]['opinion'])
        else:
            network.node[e[1]]['memory'][e[0]] = deque(
                    {network.node[e[0]]['opinion']},
                    maxlen=network.node[e[1]]['memlen'])

# initialize
rounds = 1000
network = nx.Graph()
elite = 30
mass = 70
nodes = elite + mass
edges = 300
elite_comp = 0.7
mass_comp = 0.6
# create vector of probabolity to vote for Elite option
comp_vector = [elite_comp] * elite + [1 - mass_comp] * mass
#define how many previous opinions from continuously connected nb
# the different types of agents remember
elite_memory = 5
mass_memory = 1
# make network; 1 stands for elite, 0 for mass
for n in range(elite):
    network.add_node(n, type=1, competence=elite_comp,
                     memory = dict(), memlen = elite_memory)
for n in range(elite,mass+elite):
    network.add_node(n, type=0, competence=1-mass_comp,
                     memory = dict(), memlen = mass_memory)
# add edges from an undirected random graph
network.add_edges_from( nx.gnm_random_graph(nodes,edges).edges() )

# main routine
for i in range(rounds):
    #do the opinion formation and pooled voting
    # note that, without loss of generality, it is assumed that Elites always
    #have correct answer 1 and Masses correct answer 0
    group_opin()
    votes()
    mem_update()
    # find 10% of nodes
    run_nodes = rd.sample(network.nodes(), int(round(nodes / 10)))
```

```
#run through these node
for n in run_nodes:
    max_disagree_list = []
    #only start deleting if a neighbor is wrong at least 50% of the time
    max_d = 1/2.0
    # run through all entries in memory
    for k, m in network.node[n]['memory'].items():
        #calculate neighbour rate of disagreement by comparing with type of
        #node deletting edge. Note that agents know correct answer with
        #hindsight so they can identify who was not voting correctly from
        #their perspective
        disagreement = (len(m) - m.count(network.node[n]['type'])) / float(len(m))
        #if this rate higher than all found previously, set nb as
        #nb with new highest disagreement
        if disagreement > max_d:
            max_d = disagreement
            max_disagree_list = [k]
            # if it is equal to what has been found previously, add nb to list
        elif disagreement == max_d and disagreement > 0:
            max_disagree_list.append(k)
    if len(max_disagree_list) > 0:
        delete_target = rd.choice(max_disagree_list)
        network.remove_edge(n, delete_target)
        # when edge is deleted, delete memory of nodes about each other
        del network.node[n]['memory'][delete_target]
        del network.node[delete_target]['memory'][n]
        partner_list = []
        #ensure to look for an initiator such that an unconnected partner
        #exists (as no edge can be added to completely connected node)
        while not partner_list:
            # create new random edge
            # find random initiator
            initiator = rd.choice(list(network.nodes()))
            #find candidate partners not connected to initiator
            nb_init = list(network[initiator]) + [initiator]
            partner_list = [z for z in network.nodes() if z not in nb_init]
        partner = rd.choice(partner_list)
        network.add_edge(initiator, partner)
```