In [ ]:
```python
import numpy as np
import random
import scipy.stats
import matplotlib.pyplot as plt
```

In [ ]:
```python
# each voter is an index in the array

def construct_neighbors(truecompetencies, neighborpairs):
    """Returns direct neighbors of voters given pairs of neighbors (edges in g
raph)
    ARGS:
        truecompetencies: list of true competencies (between 0 and 1) for each
voter
        neighborpairs: list of unrepeated, undirected pairs of neighbors; not
 necessarily ordered

    OUTPUTS:
        neighbors: list of direct neighbors for each voter
    """
    n = len(truecompetencies)

    if np.sum(np.array(neighborpairs).flatten() > n - 1) != 0:
        return("Error: Pair contains nonexistent voter.")

    # initialize neighborlist
    neighbors = [[] for i in range(n)]

    # add neighbors
    for pair in neighborpairs:
        (p1, p2) = pair
        neighbors[p1].append(p2)
        neighbors[p2].append(p1)

    return neighbors
```

In [ ]:
```python
# perceived competencies are just based on public voting record.
def prior_competencies(competencies):
    """Initializes Beta(1,1) public competency distribution priors for everyon
e
    ARGS:
        competencies: list of true competencies (between 0 and 1) for each vot
er
    OUTPUTS:
        priors: list of Beta parameters for each individual
    """
    n = len(competencies)
    priors = [(1,1) for i in range(n)] # initialize everyone to Beta(1,1)=Unif
(0,1) distributions
    return priors
```

In [ ]:

```python
def allocate(truecompetencies, pubcompetencies, neighbors, mechanism="standard"):
    """Returns index of final voter whom each voter allocates their vote to
    ARGS:
        truecompetencies: list of true competencies (between 0 and 1) for each voter
        pubcompetencies: list of public competency distributions for each voter, as paramters for Beta dist
        neighbors: list of direct neighbors for each voter
    Options:
        mechanism: local delegation mechanism to use (see below)
    Mechanisms:
        "standard": each voter delegates to their neighbor with highest mean competency (if they exist).
                    in case of ties, voter delegates to the first in their list
        "truestandard": each voter delegates to their neighbor with highest true competency (if they exist).
                    in case of ties, voter delegates to the first in their list
        "probstandard": each voter delegates to their neighbor with highest mean competency  (if they exist) with the probability they are actually better.
                    in case of ties, voter delegates to the first in their list
        "weightedprob": each voter randomly delegates to a more competent neighbor (if they exist), weighted by their perceived probability of being better
        "direct": no allocation

    OUTPUTS:
        allocation: list of indices, where each value is the index of the sink
                    voter whom that voter delegates to
    """

    allocation = [i for i in range(len(truecompetencies))] # initially each voter allocates vote to self

    meanpubcompetencies = [j[0]/(j[0]+j[1]) for j in pubcompetencies] # mean of each dist

    # for every voter, consider their neighbors
    for i, ineighbors in enumerate(neighbors):
        # i is index of voter being considered, ineighbors is list of indices of i's neighbors

        # if i has no neighbors
        if len(ineighbors) == 0:
            continue

        elif mechanism=="standard":
            neighborcompetencies = [meanpubcompetencies[j] for j in ineighbors]
            bestneighbor = ineighbors[np.argmax(neighborcompetencies)] # picks neighbor with highest competency; if ties, chooses first

            if truecompetencies[i] < meanpubcompetencies[bestneighbor]:
                allocation[i] = bestneighbor
```

```
        elif mechanism=="truestandard":
            neighborcompetencies = [truecompetencies[j] for j in ineighbors]
            bestneighbor = ineighbors[np.argmax(neighborcompetencies)] # picks neighbor with highest competency; if ties, chooses first

            if truecompetencies[i] < truecompetencies[bestneighbor]:
                allocation[i] = bestneighbor

        elif mechanism=="probstandard":
            neighborcompetencies = [meanpubcompetencies[j] for j in ineighbors]

            bestneighbor = ineighbors[np.argmax(neighborcompetencies)] # picks neighbor with highest competency; if ties, chooses first

            # allocate to best neighbor with probability equal to cdf of distribtuion for neighbor from own distribution up to 1
            if random.random() < 1 - scipy.stats.beta.cdf(x=truecompetencies[i], a=pubcompetencies[bestneighbor][0], b=pubcompetencies[bestneighbor][1]):
                allocation[i] = bestneighbor

        elif mechanism=="weightedprob":
            neighborcompetencies = [meanpubcompetencies[j] for j in ineighbors]

            betterneighbors = np.array(ineighbors)[np.array(neighborcompetencies) > truecompetencies[i]] # neighbors whose man is better than i

            if len(betterneighbors) > 0: # if at least one better neighbor
                neighborweights = [1 - scipy.stats.beta.cdf(x=truecompetencies[i], a=pubcompetencies[j][0], b=pubcompetencies[j][1]) for j in betterneighbors] # probability of being better than i
                allocation[i] = betterneighbors[random.choices(range(len(betterneighbors)), weights=neighborweights)]
        elif mechanism=="direct":
            return [i for i in range(len(truecompetencies))]


    # Here we allocate to the last sink in the chain
    for node in range(len(allocation)): # check across all voters
        path = [node] # create running cycle list

        while allocation[path[-1]] not in path: # if hasn't gotten to someone who we've seen before
            # print(path)
            path.append(allocation[path[-1]]) # add them to the running list of voters in cycle

        delegate = random.choice(path[path.index(allocation[path[-1]]):]) # pick someone random in the cycle (cycle starts from point where while loop broken)

        for point_to_sink in path: # make everyone delegate to same person, completing liquid flow
            allocation[point_to_sink] = delegate

    return allocation
```

In [ ]:
```
star_truecomps = np.array([.8]+[.6 for i in range(8)]) # true competencies
star_pubcomps = prior_competencies(star_truecomps) # initialize parameters for
priors
star_nbpairs = np.array([(0, i) for i in range(1,9)]) # pairs of neighbors
print(star_truecomps, star_pubcomps, star_nbpairs)
```

In [ ]:
```
# test once
star_nbs = construct_neighbors(star_truecomps, star_nbpairs)
print(allocate(star_truecomps, star_pubcomps, star_nbs, mechanism="standard"))
# allocate using mean perceived competency
print(allocate(star_truecomps, star_pubcomps, star_nbs, mechanism="truestandar
d")) # allocate using true competency
```

In [ ]:
```
def update_competencies(initialcompetencies, votes):
    """Returns updated competencies
    ARGS:
        initialcompetencies: list of initial public competency distributions f
or each voter, as parameters for Beta dist
        votes: list of voting correctness for each voter

    OUTPUTS:
        finalcompetencies: list of initial public competency distributions for
each voter, as parameters for Beta dist
    """
    finalcompetencies = initialcompetencies.copy()
    for i in range(len(votes)):
        finalcompetencies[i] = (initialcompetencies[i][0] + 1, initialcompeten
cies[i][1]) if votes[i]==1 else (initialcompetencies[i][0], initialcompetencie
s[i][1] + 1)

    return finalcompetencies
```

In [ ]:
```
star_pubcomps
testvotes = [1 for i in range(5)] + [0 for i in range(4)]
star_pubcomps2 = update_competencies(star_pubcomps, testvotes)
star_pubcomps2
```

In [ ]:
```python
def simulate_liquid(truecompetencies, neighborpairs, pubcompetencies=None, rounds=100, mechanism="standard"):
    """Returns final public competencies, arrays of numbers = np.random.binomial(n, p, 1000)s = np.random.binomial(n, p, 1000) of voters who allocate, nubmer of voters who vote correctly
    ARGS:
        truecompetencies: list of true competencies (between 0 and 1) for each voter
        neighborpairs: list of unrepeated, undirected pairs of neighbors; not necessarily ordered
    Options:
        pubcompetencies: list of public competency distributions for each voter, as parameters for Beta dist
        rounds: # of iterative rounds to vote
        mechanism: local delegation mechanism to use (see allocate function)

    OUTPUTS:
        pubcompetencies: list of final public competency distributions for each voter, as parameters for Beta dist
        numallocated: number of voters who allocate their votes in each round
        numcorrect: number of voters who vote correctly in each round
        numdirect: under direct democracy
    """

    n = len(truecompetencies)
    neighbors = construct_neighbors(truecompetencies, neighborpairs)

    if pubcompetencies==None:
        pubcompetencies = prior_competencies(truecompetencies) # initialize to Beta(1,1)

    numallocated = []
    numcorrect = []
    numdirect = [] # can ignore: testing, for comparison
    for t in range(rounds):
        # allocate
        allocation = allocate(truecompetencies, pubcompetencies, neighbors, mechanism=mechanism)
        # vote
        initialvotes = np.random.binomial(1, truecompetencies)
        allocatedvotes = initialvotes[allocation]

        allocated = np.count_nonzero(np.array(allocation)-np.array([i for i in range(len(truecompetencies))])) # number of voters who allocated their vote to someone else

        numallocated.append(allocated) # add number of voters who give up their vote
        numcorrect.append(np.sum(allocatedvotes)) # add number correct to correct
        numdirect.append(np.sum(initialvotes)) # add initial number correct under directdemocracy to numdirect

        # update perceived competencies
        pubcompetencies = update_competencies(pubcompetencies, initialvotes) # NOTE: judging everyone based on their broadcasted vote (possibly not counted)
```

```
        # pubcompetencies = update_competencies(pubcompetencies, allocatedvote
s) #NOTE: judging everyone by their final (possibly allocated) vote!!!!

    propcorrect = np.sum(numcorrect>np.ceil((n+1)/2.))/rounds # proportion ove
r time of votes that are "correct" (>1/2 of voters vote correctly)
    propdirect = np.sum(numdirect>np.ceil((n+1)/2.))/rounds
    print("Proportion of rounds where liquid electorate is right:", propcorrec
t, "\nProportion of rounds where direct democracy is right: ", propdirect)
    return propcorrect, propdirect, pubcompetencies, np.array(numallocated), n
p.array(numcorrect), np.array(numdirect)
```

In [ ]:

```
# Star structure 1
starprops_liquid = []
starprops_direct = []
competencies = [0.0, 0.2, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
for center_competency in competencies:
    star_truecomps = np.array([center_competency]+[.6 for i in range(8)]) # tr
ue competencies
    star_nbpairs = np.array([(0, i) for i in range(1,9)]) # pairs of neighbors
    print("Center competency:", center_competency)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(star_truecomps, star
_nbpairs, rounds=1000, mechanism="standard")

    starprops_liquid.append(propcorrect)
    starprops_direct.append(propdirect)
    # simulate_liquid(star_truecomps, star_nbpairs, rounds=100, mechanism="tru
estandard")
#print(np.mean(starprops_liquid), np.mean(starprops_direct),starprops_liquid,
 starprops_direct)
plt.plot(competencies, starprops_liquid, label="Liquid")
plt.plot(competencies, starprops_direct)
plt.xlabel("Competency"); plt.ylabel("Majority Voted Correctly Proportion")
plt.title("Voting Majority by Center Competency (1000 rounds)")
plt.legend()
```

Random graphs

In [ ]:

```
# Random graph simulation

n = 100 # number of people
rand_truecomps = np.random.random_sample(size = n)
rand_nbpairs = construct_neighborpairs(n)
simulate_liquid(rand_truecomps, rand_nbpairs, rounds=50, mechanism="standard")
```

In [ ]:
```
# Simulate a bunch, 50 rounds

randprops_liquid, randprops_direct = [], []
for i in range(100):
    n = 100 # number of people
    rand_truecomps = np.random.random_sample(size = n)
    rand_nbpairs = construct_neighborpairs(n)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(rand_truecomps, rand
_nbpairs, rounds=50, mechanism="standard")

    randprops_liquid.append(propcorrect)
    randprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!---------------------------------------
----------------")

print(np.mean(randprops_liquid), np.mean(randprops_direct), randprops_liquid,
randprops_direct)
```

In [ ]:
```
# Simulate a bunch, 5 rounds

for i in range(100):
    n = 10 # number of people
    rand_truecomps = np.random.random_sample(size = n)
    rand_nbpairs = construct_neighborpairs(n)
    simulate_liquid(rand_truecomps, rand_nbpairs, rounds=1, mechanism="standar
d")

    print(i+1, "th iteration complete!!---------------------------------------
----------------")
```

Zipf Random

In [ ]:
```
import scipy.stats
scipy.stats.pareto.pdf(1,1) # = 1. normalized correctly
scipy.stats.pareto.pdf(5,1) # = .04
100*scipy.stats.pareto.pdf(1,1/100) # = 1. normalized correctly
100*scipy.stats.pareto.pdf(5,1/100) # = .2

# apply zipf?
```

In [ ]:

```
# Generate popularities according to Pareto dist

def generate_popularities(n, repeat=1):
    """Returns voters' popularities according to Zipf dist
    ARGS:
        n: number of voters
        repeat: number of voters with 1/i popularity

    OUTPUTS:
        popularities: list of popularity for each voter, following an approxim
ate Zipf distribution
    """

    popularities = [1/np.ceil(i/repeat) for i in range(1,n+1)] # repeated Zipf
function
    return popularities

    # print("Popularities:", popularities)


def generate_zipfgraph(n, repeat=1):
    """Returns voters' popularities according to Zipf dist
    ARGS:
        n: number of voters
        repeat: number of voters with 1/i popularity

    OUTPUTS:
        popularities: list of popularity for each voter, following an approxim
ate Zipf distribution
        nbpairs: generated pairs of neighbors, based on popularity
    """

    popularities = generate_popularities(n, repeat)

    nbpairs = []
    for i in range(n):
        for j in range(i+1,n):
            if random.random() < popularities[i]*popularities[j]:
                nbpairs.append((i,j))
    # print(len(nbpairs), nbpairs)

    return popularities, nbpairs

generate_zipfgraph(100, repeat=3)
```

In [ ]:
```
zipfprops_liquid, zipfprops_direct = [], []
for i in range(10):
    n = 100 # number of people
    zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=3)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(zipf_truecomps, zipf
_nbpairs, rounds=100, mechanism="standard")

    zipfprops_liquid.append(propcorrect)
    zipfprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!---------------------------------------
----------------")

print(np.mean(zipfprops_liquid), np.mean(zipfprops_direct),zipfprops_liquid, z
ipfprops_direct)
```

In [ ]:
```
# 100 instances of 10 rounds
zipfprops_liquid, zipfprops_direct = [], []
for i in range(100):
    n = 100 # number of people
    zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=3)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(zipf_truecomps, zipf
_nbpairs, rounds=10, mechanism="standard")

    zipfprops_liquid.append(propcorrect)
    zipfprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!---------------------------------------
----------------")

print(np.mean(zipfprops_liquid), np.mean(zipfprops_direct),zipfprops_liquid, z
ipfprops_direct)
```

In [ ]:
```
# 100 instances of 5 rounds
zipfprops_liquid, zipfprops_direct = [], []
for i in range(100):
    n = 100 # number of people
    zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=3)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(zipf_truecomps, zipf
_nbpairs, rounds=5, mechanism="standard")

    zipfprops_liquid.append(propcorrect)
    zipfprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!---------------------------------------
----------------")

print(np.mean(zipfprops_liquid), np.mean(zipfprops_direct),zipfprops_liquid, z
ipfprops_direct)
```

Probabilistic delegation mechanisms

Probstandard

In [ ]:
```
# Star structure 1
starprops_liquid = []
starprops_direct = []
for center_competency in [0.0, 0.2, 0.4, 0.6, 0.7, 0.8, 0.9, 1.0]:
    star_truecomps = np.array([center_competency]+[.6 for i in range(8)]) # tr
ue competencies
    star_nbpairs = np.array([(0, i) for i in range(1,9)]) # pairs of neighbors
    print("Center competency:", center_competency)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(star_truecomps, star
_nbpairs, rounds=1000, mechanism="probstandard")

    starprops_liquid.append(propcorrect)
    starprops_direct.append(propdirect)
    # simulate_liquid(star_truecomps, star_nbpairs, rounds=100, mechanism="tru
estandard")
print(np.mean(starprops_liquid), np.mean(starprops_direct),starprops_liquid, s
tarprops_direct)
```

In [ ]:
```
# Simulate a bunch, 50 rounds

randprops_liquid, randprops_direct = [], []
for i in range(100):
    n = 100 # number of people
    rand_truecomps = np.random.random_sample(size = n)
    rand_nbpairs = construct_neighborpairs(n)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(rand_truecomps, rand
_nbpairs, rounds=50, mechanism="probstandard")

    randprops_liquid.append(propcorrect)
    randprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!----------------------------------------
----------------")

print(np.mean(randprops_liquid), np.mean(randprops_direct), randprops_liquid,
randprops_direct)
```

In [ ]:
```
zipfprops_liquid, zipfprops_direct = [], []
for i in range(10):
    n = 100 # number of people
    zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=3)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(zipf_truecomps, zipf
_nbpairs, rounds=100, mechanism="probstandard")

    zipfprops_liquid.append(propcorrect)
    zipfprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!----------------------------------------
----------------")

print(np.mean(zipfprops_liquid), np.mean(zipfprops_direct),zipfprops_liquid, z
ipfprops_direct)
```

In [ ]:
```
# 100 instances of 10 rounds
zipfprops_liquid, zipfprops_direct = [], []
for i in range(100):
    n = 100 # number of people
    zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=3)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(zipf_truecomps, zipf
_nbpairs, rounds=10, mechanism="probstandard")

    zipfprops_liquid.append(propcorrect)
    zipfprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!---------------------------------------
----------------")

print(np.mean(zipfprops_liquid), np.mean(zipfprops_direct),zipfprops_liquid, z
ipfprops_direct)
```

In [ ]:
```
# 100 instances of 5 rounds
zipfprops_liquid, zipfprops_direct = [], []
for i in range(100):
    n = 100 # number of people
    zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=3)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(zipf_truecomps, zipf
_nbpairs, rounds=5, mechanism="probstandard")

    zipfprops_liquid.append(propcorrect)
    zipfprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!---------------------------------------
----------------")

print(np.mean(zipfprops_liquid), np.mean(zipfprops_direct),zipfprops_liquid, z
ipfprops_direct)
```

weightedprob

In [ ]:
```
# Star structure 1
starprops_liquid = []
starprops_direct = []
for center_competency in [0.0, 0.2, 0.4, 0.6, 0.7, 0.8, 0.9, 1.0]:
    star_truecomps = np.array([center_competency]+[.6 for i in range(8)]) # tr
ue competencies
    star_nbpairs = np.array([(0, i) for i in range(1,9)]) # pairs of neighbors
    print("Center competency:", center_competency)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(star_truecomps, star
_nbpairs, rounds=1000, mechanism="weightedprob")

    starprops_liquid.append(propcorrect)
    starprops_direct.append(propdirect)
    # simulate_liquid(star_truecomps, star_nbpairs, rounds=100, mechanism="tru
estandard")
print(np.mean(starprops_liquid), np.mean(starprops_direct),starprops_liquid, s
tarprops_direct)
```

In [ ]:
```
# Simulate a bunch, 50 rounds

randprops_liquid, randprops_direct = [], []
for i in range(100):
    n = 100 # number of people
    rand_truecomps = np.random.random_sample(size = n)
    rand_nbpairs = construct_neighborpairs(n)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(rand_truecomps, rand
_nbpairs, rounds=50, mechanism="weightedprob")

    randprops_liquid.append(propcorrect)
    randprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!----------------------------------------
----------------")

print(np.mean(randprops_liquid), np.mean(randprops_direct), randprops_liquid,
randprops_direct)
```

In [ ]:
```
zipfprops_liquid, zipfprops_direct = [], []
for i in range(10):
    n = 100 # number of people
    zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=3)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(zipf_truecomps, zipf
_nbpairs, rounds=100, mechanism="weightedprob")

    zipfprops_liquid.append(propcorrect)
    zipfprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!----------------------------------------
----------------")

print(np.mean(zipfprops_liquid), np.mean(zipfprops_direct),zipfprops_liquid, z
ipfprops_direct)
```

In [ ]:
```
# 100 instances of 10 rounds
zipfprops_liquid, zipfprops_direct = [], []
for i in range(100):
    n = 100 # number of people
    zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=3)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(zipf_truecomps, zipf
_nbpairs, rounds=10, mechanism="weightedprob")

    zipfprops_liquid.append(propcorrect)
    zipfprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!----------------------------------------
----------------")

print(np.mean(zipfprops_liquid), np.mean(zipfprops_direct),zipfprops_liquid, z
ipfprops_direct)
```

In [ ]:

```
# 100 instances of 5 rounds
zipfprops_liquid, zipfprops_direct = [], []
for i in range(100):
    n = 100 # number of people
    zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=3)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(zipf_truecomps, zipf
_nbpairs, rounds=5, mechanism="weightedprob")

    zipfprops_liquid.append(propcorrect)
    zipfprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!---------------------------------------
----------------")

print(np.mean(zipfprops_liquid), np.mean(zipfprops_direct),zipfprops_liquid, z
ipfprops_direct)
```

Comparisons & Plots

In [ ]:

```
# Simulate liquid, compare stuff

def simulate_liquid_compare(truecompetencies, neighborpairs, pubcompetencies=N
one, rounds=100, mechanism="standard", mode="allocation"):
    """Returns final public competencies, arrays of numbers = np.random.binomi
al(n, p, 1000)s = np.random.binomial(n, p, 1000) of voters who allocate, nubme
r of voters who vote correctly
    ARGS:
        truecompetencies: list of true competencies (between 0 and 1) for each
voter
        neighborpairs: list of unrepeated, undirected pairs of neighbors; not
 necessarily ordered
    Options:
        pubcompetencies: list of public competency distributions for each vote
r, as parameters for Beta dist
        rounds: # of iterative rounds to vote
        mechanism1: local delegation mechanism to use (see allocate function)
        mechanism2: local delegation mechanism to compare to (see allocate fun
ction)

    OUTPUTS:
        pubcompetencies: list of final public competency distributions for eac
h voter, as parameters for Beta dist
        numallocated: number of voters who allocate their votes in each round
        numcorrect: number of voters who vote correctly in each round
        numdirect: under direct democracy
    """

    n = len(truecompetencies)
    neighbors = construct_neighbors(truecompetencies, neighborpairs)

    if pubcompetencies==None:
        pubcompetencies = prior_competencies(truecompetencies) # initialize to
Beta(1,1)

    numallocated = []
    numcorrect = []
    numdirect = [] # can ignore: testing, for comparison
    prop_correct_allocations = []
    better_than_direct = []


    for t in range(rounds):
        # allocate
        allocation = allocate(truecompetencies, pubcompetencies, neighbors, me
chanism=mechanism)
        true_allocation = allocate(truecompetencies, pubcompetencies, neighbor
s, mechanism="truestandard")
        direct_allocation = [i for i in range(n)]

        correct_allocation_count = 0

        for i in range(n):
            if allocation[i] == true_allocation[i]:
                correct_allocation_count += 1
```

```
        prop_correct_allocations.append(correct_allocation_count / n)

        # vote
        initialvotes = np.random.binomial(1, truecompetencies)
        allocatedvotes = initialvotes[allocation]

        allocated = np.count_nonzero(np.array(allocation)-np.array([i for i in
range(len(truecompetencies))])) # number of voters who allocated their vote to
someone else

        numallocated.append(allocated) # add number of voters who give up thei
r vote
        numcorrect.append(np.sum(allocatedvotes)) # add number correct to corr
ect
        numdirect.append(np.sum(initialvotes)) # add initial number correct un
der directdemocracy to numdirect
        better_than_direct.append(numcorrect[t] >= numdirect[t])

        # update perceived competencies
        pubcompetencies = update_competencies(pubcompetencies, initialvotes) #
NOTE: judging everyone based on their broadcasted vote (possibly not counted)
        # pubcompetencies = update_competencies(pubcompetencies, allocatedvote
s) #NOTE: judging everyone by their final (possibly allocated) vote!!!!

    propcorrect = np.sum(numcorrect>np.ceil((n+1)/2.))/rounds # proportion ove
r time of votes that are "correct" (>1/2 of voters vote correctly)
    propdirect = np.sum(numdirect>np.ceil((n+1)/2.))/rounds

    # Return allocation performance of standard (vs. truestandard) per round
    if mode=="allocation":
#         print("Proportion of correct allocations per round:\n", prop_correct
_allocations, '\n')
        return prop_correct_allocations

    # Return proportion of correct votes per round
    elif mode=="votes":
#         print("Proportion of correct votes per round:\n", np.array(numcorrec
t) / n, '\n')
        return np.array(numcorrect) / n
```

In [ ]:
```
import matplotlib.pyplot as plt
```

In [ ]:
```
def avg_allocation_performance(truecompetencies, neighborpairs, pubcompetencie
s=None, rounds=100, iter=10, mechanism="standard"):

    return simulate_avg(simulate_liquid_compare, {"truecompetencies": truecomp
etencies, "neighborpairs": neighborpairs,
                                                  "pubcompetencies": pubcompet
encies,
                                                  "rounds": rounds, "mechanis
m": mechanism, "mode": "allocation"}, iter)
```

In [ ]:
```
def avg_vote_performance(truecompetencies, neighborpairs, pubcompetencies=None
, rounds=100, iter=10):
    standard_performance = simulate_avg(simulate_liquid_compare, {"truecompete
ncies": truecompetencies, "neighborpairs": neighborpairs,
                                              "pubcompetencies": pubcompetenci
es,
                                              "rounds": rounds, "mechanism":
"standard", "mode": "votes"}, iter)
    truestandard_performance = simulate_avg(simulate_liquid_compare, {"truecom
petencies": truecompetencies, "neighborpairs": neighborpairs,
                                              "pubcompetencies": pubcompetenci
es,
                                              "rounds": rounds, "mechanism":
"truestandard", "mode": "votes"}, iter)
    direct_performance = simulate_avg(simulate_liquid_compare, {"truecompetenc
ies": truecompetencies, "neighborpairs": neighborpairs,
                                              "pubcompetencies": pubcompetenci
es,
                                              "rounds": rounds, "mechanism":
"direct", "mode": "votes"}, iter)
    return standard_performance, truestandard_performance, direct_performance
```

In [ ]:
```
def simulate_avg(func, pm, iter):
    lst = []
    for i in range(iter):
        lst.append(func(**pm))

    return np.mean(np.array(lst), axis=0)
```

In [ ]:
```
def run_zipf_graphs(func, pm, n, iter, correlation_parameter, repeat=3):
    lst = []
    for i in range(iter):
        pm["truecompetencies"], pm["neighborpairs"] = generate_zipfgraph(n, re
peat=repeat)
        if correlation_parameter == "anticorrelated":
            pm["truecompetencies"] = [1-c for c in pm["truecompetencies"]]
        elif correlation_parameter == "uncorrelated":
            pm["truecompetencies"] = [np.random.uniform() for c in range(len(p
m["truecompetencies"]))]
        lst.append(simulate_avg(func, pm, iter))

    return np.mean(np.array(lst), axis=0)
```

In [ ]:

```
# Generate plot for Zipf graphs with n vertices

def plot_allocation_performance_zipfgraphs(n, correlation_parameter, repeat=3,
tests=100, rounds=50):

#     allocation_performances = []

#     # Generate `tests` different graphs
#     for i in range(tests):
#         zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=repeat)
#         allocation_performances.append(avg_allocation_performance(zipf_truec
omps, zipf_nbpairs, iter=2, rounds=rounds))

#     allocation_performances = np.mean(np.array(allocation_performances), axi
s=0)

#     zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=repeat)
    allocation_performances = run_zipf_graphs(avg_allocation_performance, {"it
er":2, "rounds": rounds}, n, 2,
                                              correlation_parameter
                                              )

    plt.figure()
    plt.plot(range(1, rounds + 1), allocation_performances)
    plt.xlabel("Round")
    plt.ylabel("Proportion of Correct Allocations")
    plt.title("Proportion of Correct Allocations Per Round")
    plt.show()
```

In [ ]:
```
# Generate plot for Zipf graphs with n vertices

def plot_vote_performance_zipfgraphs(n, correlation_parameter, repeat=3, tests
=100, rounds=50):

    standard_vote_performances, truestandard_vote_performances, direct_vote_pe
rformances = [], [], []

    # Generate `tests` different graphs
    for i in range(tests):
        zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=repeat)
        if correlation_parameter == "anticorrelated":
            zipf_truecomps = [1-c for c in zipf_truecomps]
        elif correlation_parameter == "uncorrelated":
            zipf_truecomps = [np.random.uniform() for c in range(len(zipf_true
comps))]
        standard, truestandard, direct = avg_vote_performance(zipf_truecomps,
zipf_nbpairs, iter=2, rounds=rounds)
        standard_vote_performances.append(standard)
        truestandard_vote_performances.append(truestandard)
        direct_vote_performances.append(direct)

    standard_vote_performances = np.mean(np.array(standard_vote_performances),
axis=0)
    truestandard_vote_performances = np.mean(np.array(truestandard_vote_perfor
mances), axis=0)
    direct_vote_performances = np.mean(np.array(direct_vote_performances), axi
s=0)

    plt.figure()
    standard, = plt.plot(range(1, rounds + 1), standard_vote_performances, col
or="g", label="Standard")
    truestandard, = plt.plot(range(1, rounds + 1), truestandard_vote_performan
ces, color="b", label="True Standard")
    direct, = plt.plot(range(1, rounds + 1), direct_vote_performances, color=
"r", label="Direct")
    plt.legend(handles=[standard, truestandard, direct])

    plt.xlabel("Round")
    plt.ylabel("Proportion of Correct Votes")
    plt.title("Proportion of Correct Votes Per Round")
    plt.show()
```

In [ ]:
```
# Random graph simulation

n = 1000 # number of people
rand_truecomps = np.random.random_sample(size = n)
rand_nbpairs = construct_neighborpairs(n)
plot_allocation_performance(rand_truecomps, rand_nbpairs, rounds=50)
plot_vote_performance(rand_truecomps, rand_nbpairs, rounds=100)
```

In [ ]:
```
plot_vote_performance_zipfgraphs(100, "correlated", tests=2)
plot_vote_performance_zipfgraphs(100, "anticorrelated", tests=2)
plot_vote_performance_zipfgraphs(100, "uncorrelated", tests=2)
```

In [ ]:

```
plot_allocation_performance_zipfgraphs(100, "correlated", tests=2)
plot_allocation_performance_zipfgraphs(100, "anticorrelated", tests=2)
plot_allocation_performance_zipfgraphs(100, "uncorrelated", tests=2)
```

Grid Structure

In [ ]:
```
def make_grid(s, competency="random"):
    """ Creates a s by s square grid
    ARGS:
        s: number of voters along each axis, total of s*s voters

    Options:
        competency: method of initializing competencies
    Competencies:
        random: assigns random competencies between 0 and 1
        [prop]: assigns all voters a fixed competency equal to the input
        periodic: other periodic tilings interesting, esp. if s is not divisib
le by period
        gradient1d: gradient of competencies, each with (i+1)/s (with 1/s on t
he left and 1 on the right (each row))
        gradient2d: gradient of comeptencies, each with (i+1+j+1)/(s*s) (with
 2/(s*s) on top left and 1 in bottom right)
        rings: ring of evenly spaced competencies, with center (1 voter if od
d, 4 voters if even) having competency 1 and outermost ring having competency
 0

    OUTPUTS:
        truecomps: list of true competencies
        nbpairs: list of pairs of neighbors
    """

    nbpairs = []
    for i in range(s-1):
        for j in range(s):
            nbpairs.append((i+j*s, i+j*s+1))
    for i in range(s):
        for j in range(s-1):
            nbpairs.append((i+j*s, i+j*s+s))

    try:
        if competency>=0 and competency<=1:
            truecomps = np.repeat(competency, s*s)
    except:
        if competency=="gradient1d":
            truecomps = np.tile(1/s*np.arange(1, s+1).flatten(),s)
        elif competency=="gradient2d":
            truecomps = 1/(s*s)*np.array([int(i/s)+1+i%s+1 for i in range(s*
s)])
        elif competency=="rings":
            maxring = np.floor((s+1)/2)
            if s%2==1: # odd
                truecomps = np.eye(s, s)
                for i in range(s):
                    for j in range(s):
                        ring = max(np.abs(i-(s-1)/2), np.abs(j-(s-1)/2)) # rin
g from inside
                        truecomps[i][j] = (maxring-1-ring)/(maxring-1)
            elif s%2==0: # even
                truecomps = np.eye(s, s)
                for i in range(s):
                    for j in range(s):
                        ring = max(np.abs(i-(s-1)/2)-0.5, np.abs(j-(s-1)/2)-0.
```

```
5) # ring from inside
                        truecomps[i][j] = (maxring-1-ring)/(maxring-1)
            # print(np.reshape(truecomps,(s,s)))
            truecomps = truecomps.flatten()

        else: #if competency=="random":
            truecomps = np.random.random_sample(size = s*s)

    return truecomps, nbpairs
```

In [ ]:

```
# Random
randgridprops_liquid, randgridprops_direct = [], []
for i in range(100):
    randgrid_comps, randgrid_nbpairs = make_grid(s=5, competency="random")
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(randgrid_comps, rand
grid_nbpairs, rounds=100, mechanism="standard")

    randgridprops_liquid.append(propcorrect)
    randgridprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!---------------------------------------
----------------")
print(np.mean(randgridprops_liquid), np.mean(randgridprops_direct),randgridpro
ps_liquid, randgridprops_direct)
```

In [ ]:

```
# All fixed same. Note: interesting to consider various shared competencies,
 e.g. improvement at low competencies but deprovement at high competencies
fixedgridprops_liquid, fixedgridprops_direct = [], []
for i in range(100):
    fixedgrid_comps, fixedgrid_nbpairs = make_grid(s=5, competency=0.2)
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(fixedgrid_comps, fix
edgrid_nbpairs, rounds=100, mechanism="standard")

    fixedgridprops_liquid.append(propcorrect)
    fixedgridprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!---------------------------------------
----------------")
print(np.mean(fixedgridprops_liquid), np.mean(fixedgridprops_direct),fixedgrid
props_liquid, fixedgridprops_direct)
```

In [ ]:

```
# 1d gradient
gradient1dgridprops_liquid, gradient1dgridprops_direct = [], []
for i in range(100):
    gradient1dgrid_comps, gradient1dgrid_nbpairs = make_grid(s=5, competency=
"gradient1d")
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(gradient1dgrid_comps
, gradient1dgrid_nbpairs, rounds=100, mechanism="standard")

    gradient1dgridprops_liquid.append(propcorrect)
    gradient1dgridprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!---------------------------------------
----------------")
print(np.mean(gradient1dgridprops_liquid), np.mean(gradient1dgridprops_direct
),gradient1dgridprops_liquid, gradient1dgridprops_direct)
```

In [ ]:
```
#2d gradient
gradient2dgridprops_liquid, gradient2dgridprops_direct = [], []
for i in range(100):
    gradient2dgrid_comps, gradient2dgrid_nbpairs = make_grid(s=5, competency=
"gradient2d")
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(gradient2dgrid_comps
, gradient2dgrid_nbpairs, rounds=100, mechanism="standard")

    gradient2dgridprops_liquid.append(propcorrect)
    gradient2dgridprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!---------------------------------------
----------------")
print(np.mean(gradient2dgridprops_liquid), np.mean(gradient2dgridprops_direct
),gradient2dgridprops_liquid, gradient2dgridprops_direct)
```

In [ ]:
```
#rings
ringgridprops_liquid, ringgridprops_direct = [], []
for i in range(100):
    ringgrid_comps, ringgrid_nbpairs = make_grid(s=5, competency="rings")
    propcorrect, propdirect, _, _, _, _ = simulate_liquid(ringgrid_comps, ring
grid_nbpairs, rounds=100, mechanism="standard")

    ringgridprops_liquid.append(propcorrect)
    ringgridprops_direct.append(propdirect)
    print(i+1, "th iteration complete!!---------------------------------------
----------------")
print(np.mean(ringgridprops_liquid), np.mean(ringgridprops_direct),ringgridpro
ps_liquid, ringgridprops_direct)
```

Do No Harm Principle

In [ ]:
```
def simulate_liquid_no_harm(truecompetencies, neighborpairs, pubcompetencies=None, rounds=100, mechanism="standard"):
    """
    Every 10 rounds, we see if we did worse than during the previous 10. If so,
    we revert to the allocation of 10 rounds ago and save this for every iteration.
    Essentially, we freeze ourselves.
    """
    n = len(truecompetencies)
    neighbors = construct_neighbors(truecompetencies, neighborpairs)

    if pubcompetencies==None:
        pubcompetencies = prior_competencies(truecompetencies) # initialize to Beta(1,1)

    numallocated = []
    numcorrect, numcorrect_liquid = [], []
    numdirect = [] # can ignore: testing, for comparison
    last_allocation = []
    last_prop_right = 0.0
    liquid_100 = []
    correct_per_100 = []
    fixed_at_allocation = False
    for t in range(1, rounds):
        if t % 10 == 0:
            correct_per_100.append(sum(liquid_100))
            if not fixed_at_allocation and sum(liquid_100) < last_prop_right * 10.0:
                print("Fixing a certain allocation", t, sum(liquid_100), last_prop_right)
                fixed_at_allocation = True
            last_prop_right = sum(liquid_100)/10.0
            liquid_100 = []

        # allocate
        if fixed_at_allocation:
            allocation = last_allocation
        else:
            allocation = allocate(truecompetencies, pubcompetencies, neighbors, mechanism=mechanism)
        if t % 100 == 1:
            last_allocation = allocation
        liquid_allocation = allocate(truecompetencies, pubcompetencies, neighbors, mechanism=mechanism)
        # vote
        initialvotes = np.random.binomial(1, truecompetencies)
        liquid_allocatedvotes, allocatedvotes = initialvotes[liquid_allocation], initialvotes[allocation]

        numcorrect.append(np.sum(allocatedvotes)) # add number correct to correct
        numcorrect_liquid.append(np.sum(liquid_allocatedvotes))
        liquid_100.append(np.sum(allocatedvotes) > np.ceil((n+1)/2.))

        numdirect.append(np.sum(initialvotes)) # add initial number correct un
```

```
der directdemocracy to numdirect

        # update perceived competencies
        pubcompetencies = update_competencies(pubcompetencies, initialvotes) #
NOTE: judging everyone based on their broadcasted vote (possibly not counted)
        # pubcompetencies = update_competencies(pubcompetencies, allocatedvote
s) #NOTE: judging everyone by their final (possibly allocated) vote!!!!

    propcorrect = np.sum(numcorrect>np.ceil((n+1)/2.))/rounds # proportion ove
r time of votes that are "correct" (>1/2 of voters vote correctly)
    propcorrect_liquid = np.sum(numcorrect_liquid>np.ceil((n+1)/2.))/rounds
    propdirect = np.sum(numdirect>np.ceil((n+1)/2.))/rounds
    print("Proportion of rounds where smart liquid electorate is right:", prop
correct,
          "\nProportion of rounds where direct democracy is right: ", propdire
ct,
         "\nProportion of rounds where naive liquid is right: ", propcorrect_l
iquid)
    return propcorrect, propdirect, propcorrect_liquid, correct_per_100
```

In [ ]:
```
starprops_liquid = []
starprops_direct = []
star_truecomps = np.array([0.8]+[.6 for i in range(100)]) # true competencies
star_nbpairs = np.array([(0, i) for i in range(1,100)]) # pairs of neighbors
propcorrect, propdirect, propcorrect_liquid, per_100 = simulate_liquid_no_harm
(star_truecomps, star_nbpairs, rounds=1001, mechanism="standard")
plt.plot(range(0, 1000, 10), per_100)
plt.title("Accuracy per 10 Rounds (Smart Liquid)")
plt.xlabel("Round"); plt.ylabel("# of Correct")
```

Grid Plots

In [ ]:
```
def run_grid_graphs(func, pm, s, iter, competency):
    lst = []
    for i in range(iter):
        pm["truecompetencies"], pm["neighborpairs"] = make_grid(s, competency=
competency)
        lst.append(simulate_avg(func, pm, iter))

    return np.mean(np.array(lst), axis=0)
```

In [ ]:
```
# Generate plot for Grid graphs with n vertices

def plot_allocation_performance_gridgraphs(s, competency, tests=100, rounds=50
):

#     allocation_performances = []

#     # Generate `tests` different graphs
#     for i in range(tests):
#         zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=repeat)
#         allocation_performances.append(avg_allocation_performance(zipf_truec
omps, zipf_nbpairs, iter=2, rounds=rounds))

#     allocation_performances = np.mean(np.array(allocation_performances), axi
s=0)

#     zipf_truecomps, zipf_nbpairs = generate_zipfgraph(n, repeat=repeat)
    allocation_performances = run_grid_graphs(avg_allocation_performance, {"it
er":5, "rounds": rounds}, s, tests,
                                              competency
                                              )

    plt.figure()
    plt.plot(range(1, rounds + 1), allocation_performances)
    plt.xlabel("Round")
    plt.ylabel("Proportion of Correct Allocations")
    try:
        plt.title("Correct Allocations Per Round (" + competency + "), s = " +
str(s))
        plt.savefig("grid_correctallocations_"+competency+"_"+str(s))
    except:
        plt.title("Correct Allocations Per Round (fixed=" + str(competency) +
"), s = " + str(s))
        plt.savefig("grid_correctallocations_"+"fixed_"+str(s))
    plt.show()
```

In [ ]:
```
plot_allocation_performance_gridgraphs(10, "random", tests=2)
plot_allocation_performance_gridgraphs(10, 0.5, tests=2)
plot_allocation_performance_gridgraphs(10, "gradient1d", tests=2)
plot_allocation_performance_gridgraphs(10, "gradient2d", tests=2)
plot_allocation_performance_gridgraphs(10, "rings", tests=2)
```

In [ ]:
```
# Generate plot for Zipf graphs with n vertices

def plot_vote_performance_gridgraphs(s, competency, tests=100, rounds=50):

    standard_vote_performances, truestandard_vote_performances, direct_vote_pe
rformances = [], [], []

    # Generate `tests` different graphs
    for i in range(tests):
        grid_truecomps, grid_nbpairs = make_grid(s, competency)
        standard, truestandard, direct = avg_vote_performance(grid_truecomps,
grid_nbpairs, iter=5, rounds=rounds)
        standard_vote_performances.append(standard)
        truestandard_vote_performances.append(truestandard)
        direct_vote_performances.append(direct)

    standard_vote_performances = np.mean(np.array(standard_vote_performances),
axis=0)
    truestandard_vote_performances = np.mean(np.array(truestandard_vote_perfor
mances), axis=0)
    direct_vote_performances = np.mean(np.array(direct_vote_performances), axi
s=0)

    plt.figure()
    standard, = plt.plot(range(1, rounds + 1), standard_vote_performances, col
or="g", label="Standard")
    truestandard, = plt.plot(range(1, rounds + 1), truestandard_vote_performan
ces, color="b", label="True Standard")
    direct, = plt.plot(range(1, rounds + 1), direct_vote_performances, color=
"r", label="Direct")
    plt.legend(handles=[standard, truestandard, direct])

    plt.xlabel("Round")
    plt.ylabel("Proportion of Correct Votes")
    try:
        plt.title("Correct Votes Per Round (" + competency + "), s = " + str(s
))
        plt.savefig("grid_correct_votes_"+competency+"_"+str(s))
    except:
        plt.title("Correct Votes Per Round (fixed=" + str(competency) + "), s
 = " + str(s))
        plt.savefig("grid_correct_votes_"+"fixed_"+str(s))
    plt.show()
```

In [ ]:
```
plot_vote_performance_gridgraphs(10, "random", tests=2)
plot_vote_performance_gridgraphs(10, 0.5, tests=2)
plot_vote_performance_gridgraphs(10, "gradient1d", tests=2)
plot_vote_performance_gridgraphs(10, "gradient2d", tests=2)
plot_vote_performance_gridgraphs(10, "rings", tests=2)
```

In [ ]:
```
# ALL PLOTS
t = 25 # fix
for s in [3, 5, 10]:
    plot_allocation_performance_gridgraphs(s, "random", tests=t)
    plot_allocation_performance_gridgraphs(s, 0.5, tests=t)
    plot_allocation_performance_gridgraphs(s, "gradient1d", tests=t)
    plot_allocation_performance_gridgraphs(s, "gradient2d", tests=t)
    plot_allocation_performance_gridgraphs(s, "rings", tests=t)

    print("s=", s, " allocations complete-------------------------------------
-----------------------")

    plot_vote_performance_gridgraphs(s, "random", tests=t)
    plot_vote_performance_gridgraphs(s, 0.5, tests=t)
    plot_vote_performance_gridgraphs(s, "gradient1d", tests=t)
    plot_vote_performance_gridgraphs(s, "gradient2d", tests=t)
    plot_vote_performance_gridgraphs(s, "rings", tests=t)

    print("s=", s, " DONE!!!--------------------------------------------------
----------")
```

In [ ]:
```
# ALL PLOTS
t = 25 # fix
for s in [15]:
    plot_allocation_performance_gridgraphs(s, "random", tests=t)
    #plot_allocation_performance_gridgraphs(s, 0.5, tests=t)
    plot_allocation_performance_gridgraphs(s, "gradient1d", tests=t)
    #plot_allocation_performance_gridgraphs(s, "gradient2d", tests=t)
    plot_allocation_performance_gridgraphs(s, "rings", tests=t)

    print("s=", s, " allocations complete-------------------------------------
-----------------------")

    plot_vote_performance_gridgraphs(s, "random", tests=t)
    #plot_vote_performance_gridgraphs(s, 0.5, tests=t)
    plot_vote_performance_gridgraphs(s, "gradient1d", tests=t)
    #plot_vote_performance_gridgraphs(s, "gradient2d", tests=t)
    plot_vote_performance_gridgraphs(s, "rings", tests=t)

    print("s=", s, " DONE!!!--------------------------------------------------
----------")
```

In [ ]: