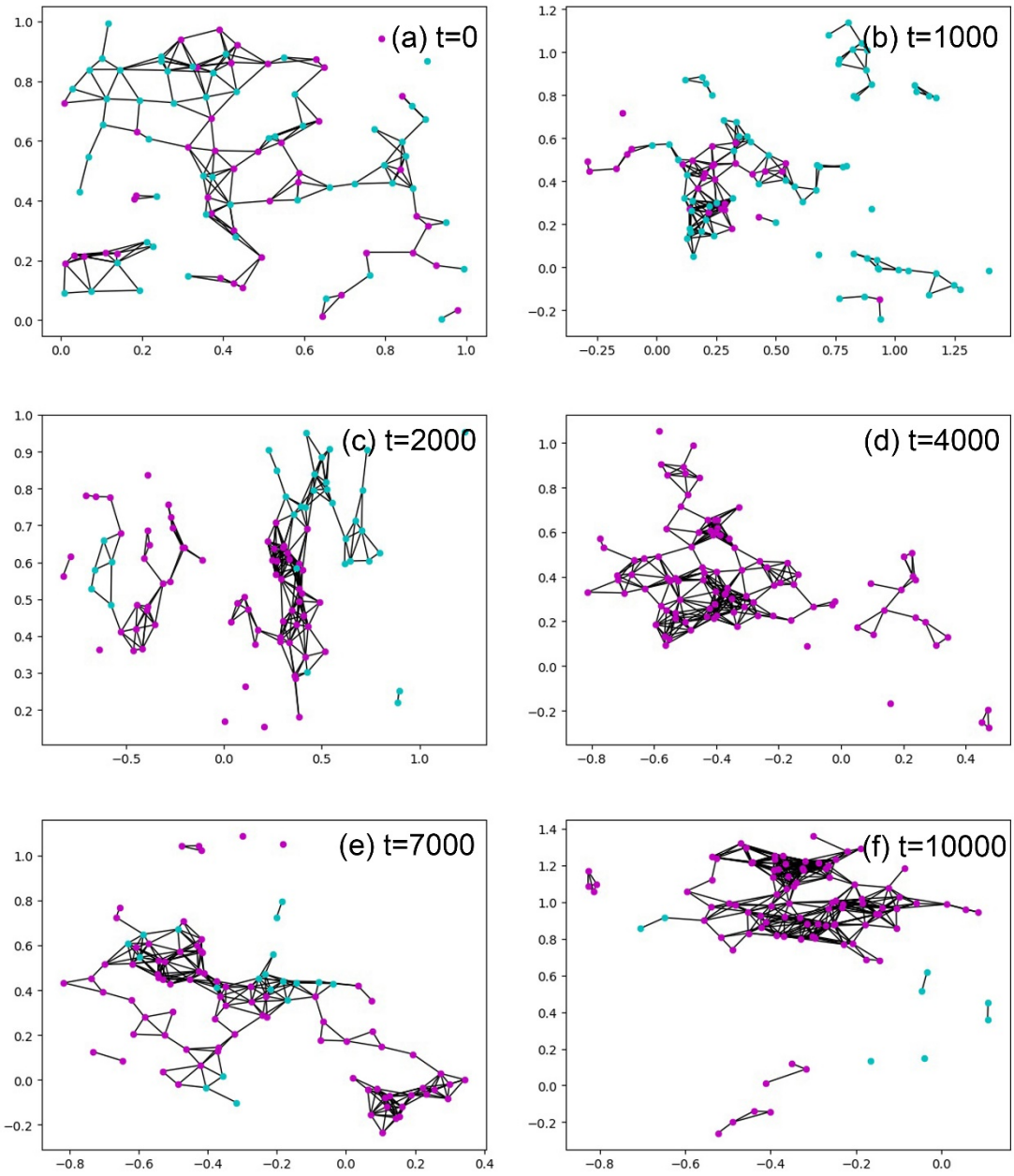
Proceedings B. 10.1098/rspb.2019.0690

Proximity inheritance explains evolution of cooperation under natural selection and mutation

Supplemental Information

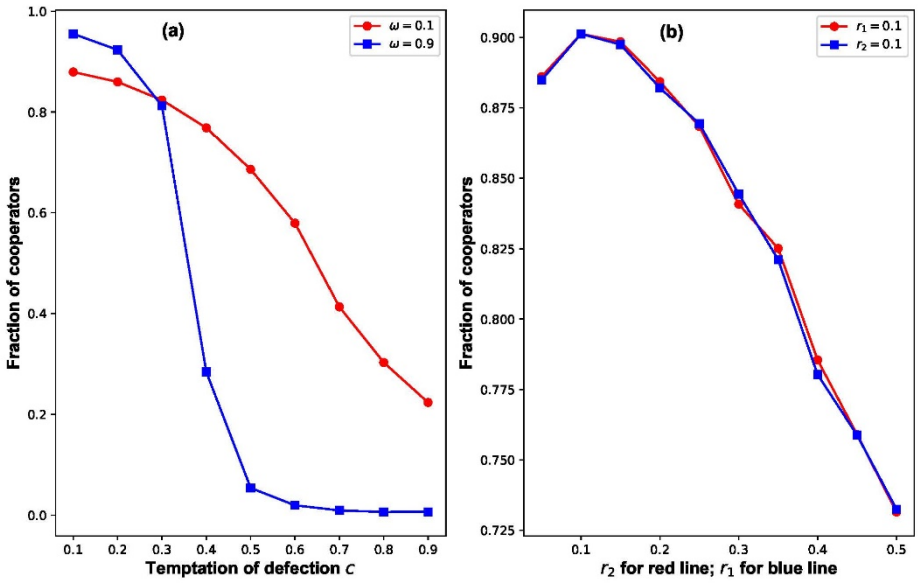
Shaolin Tan

College of Electrical and Information Engineering, Hunan University, Changsha 410082, China

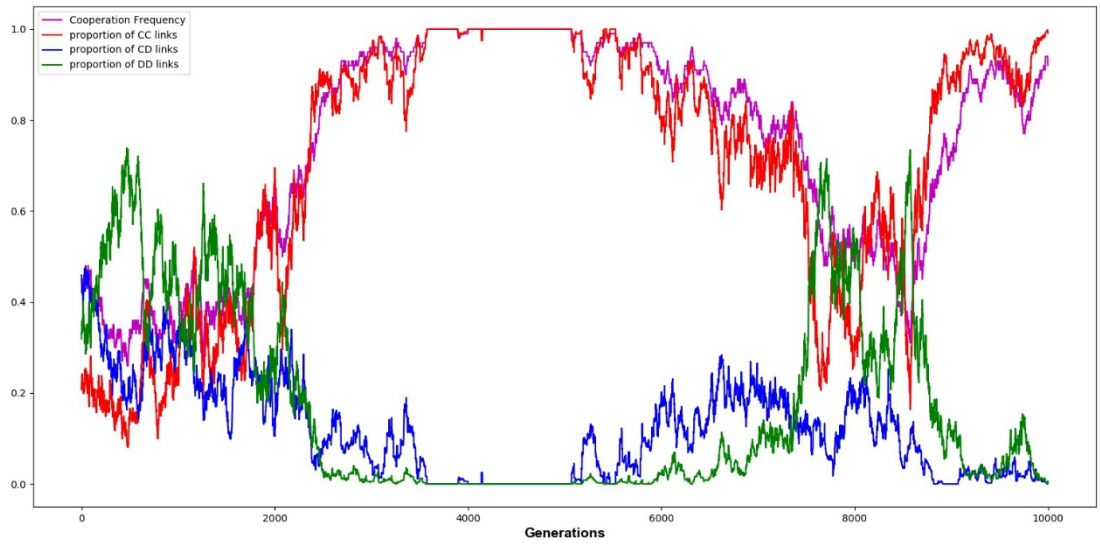


Supplementary Figure 1. Typical snapshots in a simulation of the evolutionary model. The parameters are as default. Cyan and magenta nodes denote defectors and cooperators, respectively. (a) The initial generation, cooperation frequency=0.45, the proportion of CC, CD, and DD links are 22.2%,

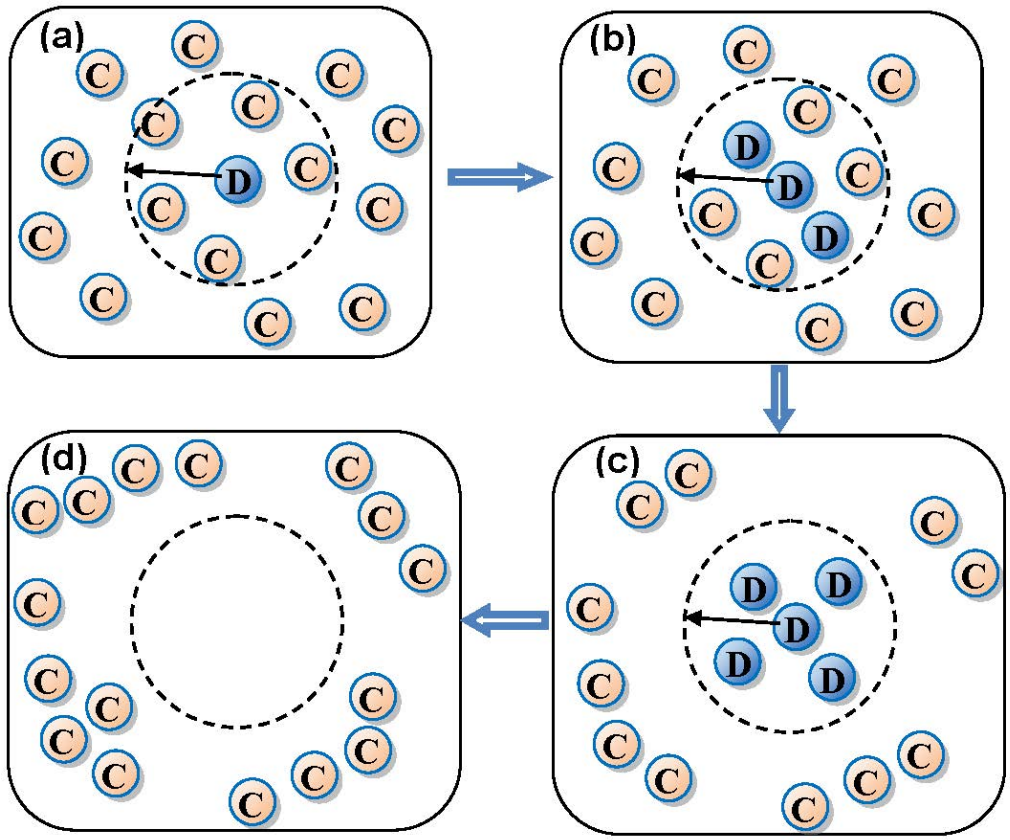
45.9%, and 31.9%, respectively. (b) The 1×10^3-th generation, cooperation frequency=0.3, the proportion of CC, CD, and DD links are 22.6%, 33.5%, and 43.9%, respectively. (c) The 2×10^3 -th generation, cooperation frequency=0.67, the proportion of CC, CD, and DD links are 69.5%, 10.6%, and 19.9%, respectively. (d) The 4×10^3 -th generation, cooperation frequency=1, the proportion of CC, CD, and DD links are 100%, 0%, and 0%, respectively. (e) The 7×10^3-th generation, cooperation frequency=0.81, the proportion of CC, CD, and DD links are 72.1%, 19.3%, and 8.6%, respectively. (f) The 10×10^3-th generation, cooperation frequency=0.92, the proportion of CC, CD, and DD links are 99.1%, 0.2%, and 0.7%, respectively.



Supplementary Figure 2. (a) The fraction of cooperators as a function of temptation of defection c under different values of selection intensity. (b) The fraction of cooperators as a function of the pervading radius r_2 with a fixed value of connection threshold (red line), and a function of the connection threshold r_1 with a fixed value of pervading radius (blue line). The population size is 200 and hence the default connection threshold and pervading radius is set as 0.1. The default mutation rate, selection intensity and temptation of defection are set as 0.01, 0.2, and 0.2, respectively. It can be observed that self-organization of clustering cooperators and its variances with other parameters are still preserved for a larger population size 200.

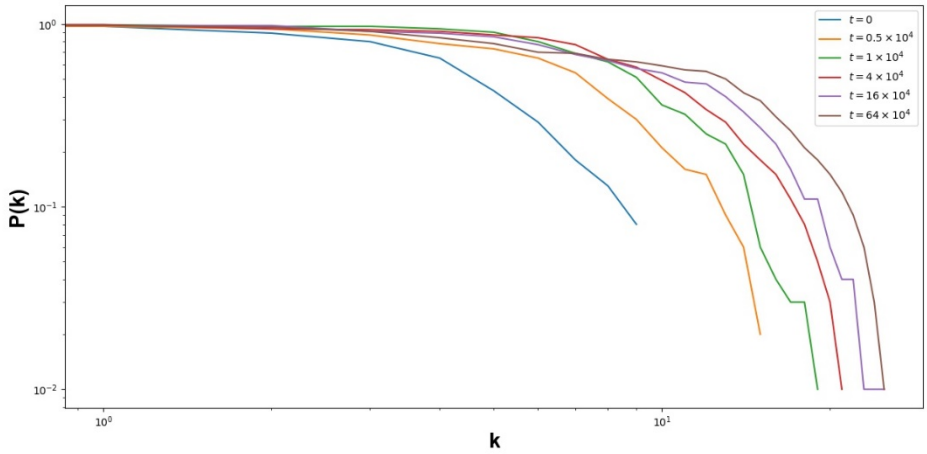


Supplementary Figure 3. Typical evolutionary traces of cooperator frequency, proportions of cooperator-cooperator, cooperator-defector, and defector-defector links in a simulation of the evolutionary model (corresponding to the Supplementary Figure 1). An impressive invasion of defectors due to mutation happens at around the 8×10^3-th step, where the cooperators are on the verge of extinction. However, they can survive and eventually take back the whole population.

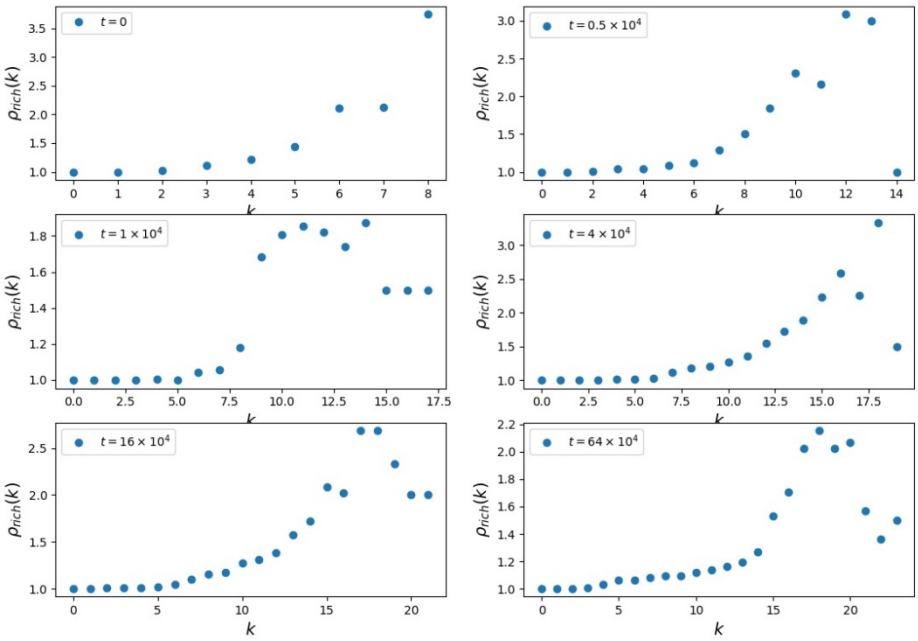


Supplementary Figure 4. Illustration of an invasion process of mutated defector. Cooperators and defectors, denoted by yellow and blue nodes respectively, are distributed spatially. In a population of cooperators, if one cooperator mutates into a defector, as shown in (a), then the defector will

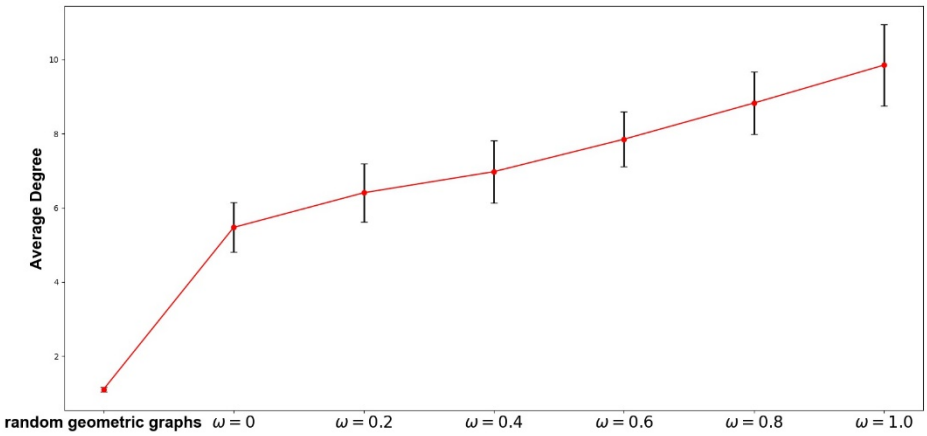
possess a high fitness by exploiting the neighboring cooperators. Hence, according to birth-death model and proximity inheritance mechanism, more defectors will be reproduced around the initial defector, as shown in (b). At the same time, due to the exploitation of defectors, the neighboring cooperators receive a low fitness, and thus are eventually eliminated, as shown in (c). Finally, without cooperators in their neighborhood, the defectors get a very low fitness, and therefore are eventually eliminated out of the population, as shown in (d).



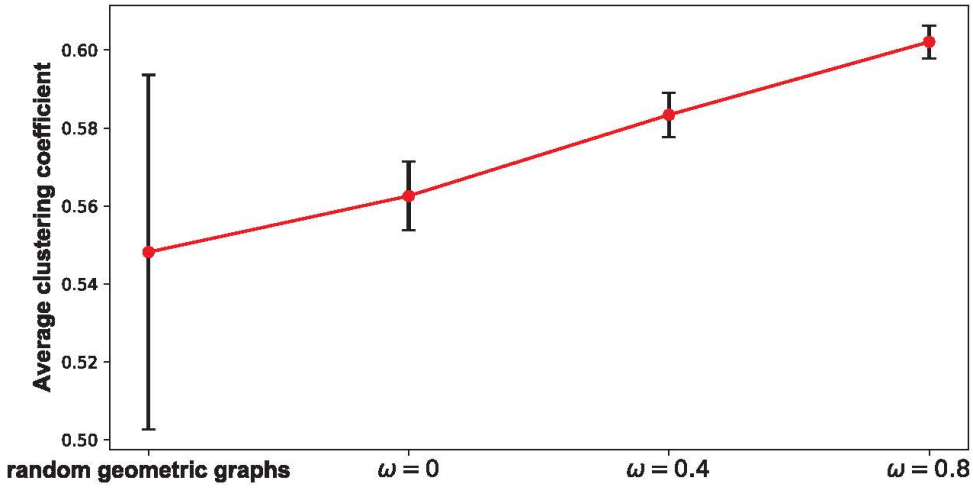
Supplementary Figure 5. Plots of the cumulative degree distributions for different snapshots of the dynamic networks in the coevolutionary model. The parameters are as default. The degree distribution of the dynamic network flattens during the coevolutionary process, indicating an emergence of the 'fat-tail' phenomenon.



Supplementary Figure 6. Plots of the normalized rich club coefficient for different snapshots of the dynamic networks in the coevolutionary model. The parameters are as default. The rich club coefficient $\phi_{\mathrm{rich}}(\mathrm{k})$ as a function of degree k becomes shaper during the coevolutionary process, indicating that nodes with large degree were more densely connected among themselves than nodes with low degree.



Supplementary Figure 7. Average degree of the dynamic networks in the coevolutionary model with selection strength $\omega = 0, 0.2, 0.4, 0.6, 0.8, 1.0$, respectively. Other parameters are as default.



Supplementary Figure 8. Average clustering coefficients of the random geometric graphs and the dynamic networks in the coevolutionary model with selection strength $\omega = 0, 0.4, 0.8$, respectively. Other parameters are as default.

Python Code

```
from math import sqrt
```

```
import math
import random as rd
from scipy.spatial import cKDTree as KDTree
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt


def euclidean(x, y):
    return sqrt(sum((a - b) ** 2 for a, b in zip(x, y)))


def _fast_edges(G, radius, p):
    pos = nx.get_node_attributes(G, 'pos')
    nodes, coords = list(zip(*pos.items()))
    kdtree = KDTree(coords)  # Cannot provide generator.
    edge_indexes = kdtree.query_pairs(radius, p)
    edges = ((nodes[u], nodes[v]) for u, v in edge_indexes)
    return edges


# N is the number of nodes, and r is the radius
def initiate_graph(N,r,p):
    g=nx.Graph()
    g.add_nodes_from(range(N))
    nodes=g.nodes()
    pos = {v: [rd.random() for i in range(2)] for v in nodes}
    nx.set_node_attributes(g, pos, 'pos')
    for d in nx.nodes(g):
        if rd.random()<p:
```

```
            g.node[d]['strategy']=1
        else:
            g.node[d]['strategy']=0
    #strategy={v: rd.randint(0,1) for v in nodes}
    #nx.set_node_attributes(g,strategy,'strategy')
    edges=_fast_edges(g, r, 2)
    g.add_edges_from(edges)
    return g


# g is the graph, pm is the game, w is the selection intensity
def calculate_fitness(g,pm,w):
    payoff={}
    fitness={}
    for d1 in nx.nodes(g):
        payoff[d1]=0
        for d2 in nx.all_neighbors(g,d1):
            mid1=np.dot([g.node[d1]['strategy'],1-g.node[d1]['strategy']],pm)
            payoff[d1]+=np.dot(mid1,[g.node[d2]['strategy'],1-g.node[d2]['strategy']])
        fitness[d1]=1-w+w*payoff[d1]
    nx.set_node_attributes(g,payoff,'payoff')
    nx.set_node_attributes(g,fitness,'fitness')
    return g


# calculate the average fitness of each individual
# g is the graph, pm is the game, w is the selection intensity
def calculate_average_fitness(g,pm,w):
    payoff={}
```

```
    fitness={}
    for d1 in nx.nodes(g):
        payoff[d1]=0
        fitness[d1]=1
        for d2 in nx.all_neighbors(g,d1):
            mid1=np.dot([g.node[d1]['strategy'],1-g.node[d1]['strategy']],pm)
            payoff[d1]+=np.dot(mid1,[g.node[d2]['strategy'],1-g.node[d2]['strategy']])
        if g.degree(d1)!=0:
            fitness[d1]=1-w+w*payoff[d1]/float(g.degree(d1))
    nx.set_node_attributes(g,payoff,'payoff')
    nx.set_node_attributes(g,fitness,'fitness')
    return g


# selecte the individual for reproducation with a probability proportional to their fitness
def selected_birth(g):
    fitness=nx.get_node_attributes(g,'fitness')
    totalfit=sum(fitness.values())
    rand=rd.random()
    end=0
    for d in nx.nodes(g):
        end=end+g.node[d]['fitness']/float(totalfit)
        if rand<end:break
    return d


def link_number(g):
```

```
    cc=0
    cd=0
    dd=0
    for d1 in nx.nodes(g):
        for d2 in nx.all_neighbors(g,d1):
            if g.node[d1]['strategy']==g.node[d2]['strategy']:
                if g.node[d1]['strategy']==1:
                    cc+=1
                else:
                    dd+=1
            else:
                cd+=1
    cc1=cc/float(cc+cd+dd)
    cd1=cd/float(cc+cd+dd)
    dd1=dd/float(cc+cd+dd)
    return [cc1,cd1,dd1]


# The evolutionary process average degree

# N,r is used to initialize graph, c is the defection tempation, w is the selection intensity,
u is the mutation rate, M is the steps

def evol_game_2(N,r_1,p,r_2,c,w,u,M):
    g=initiate_graph(N,r_1,p)
    g=calculate_average_fitness(g,[[1,0],[1+c,c]],w)
    i=1
    result=[]
    while i<M:
        cooper=nx.get_node_attributes(g,'strategy')
        aver_cooper=sum(cooper.values())/float(g.order())
```

```
        result.append(aver_cooper)

        birth=selected_birth(g)

        k=i+N-1

        pos_k=[g.node[birth]['pos'][0]+r_2*(2*rd.random()-
1),g.node[birth]['pos'][1]+r_2*(2*rd.random()-1)]

        g.add_node(k,pos=pos_k,strategy=g.node[birth]['strategy'],payoff=0,fitness=1)

        for d in nx.nodes(g):

            if sqrt((g.node[d]['pos'][0] - pos_k[0]) ** 2+(g.node[d]['pos'][1] - pos_k[1])
** 2)<r_1:

                g.add_edge(d,k)

        g.remove_edge(k,k)

        p=rd.random()

        if p<u/float(2):

            g.node[k]['strategy']=1-g.node[birth]['strategy']

        for d in nx.all_neighbors(g,k):

            mid1=np.dot([g.node[k]['strategy'],1-g.node[k]['strategy']],[[1,0],[1+c,c]])

            mid2=np.dot(mid1,[g.node[d]['strategy'],1-g.node[d]['strategy']])

            g.node[k]['payoff']+=mid2

            mid3=np.dot([g.node[d]['strategy'],1-g.node[d]['strategy']],[[1,0],[1+c,c]])

            mid4=np.dot(mid3,[g.node[k]['strategy'],1-g.node[k]['strategy']])

            g.node[d]['payoff']+=mid4

            g.node[d]['fitness']=1-w+w*g.node[d]['payoff']/float(g.degree(d))

        if g.degree(k)!=0:

            g.node[k]['fitness']=1-w+w*g.node[k]['payoff']/float(g.degree(k))

        nodes=g.nodes()

        death=rd.sample(nodes,1)[0]

        for d in nx.all_neighbors(g,death):

            mid1=np.dot([g.node[d]['strategy'],1-g.node[d]['strategy']],[[1,0],[1+c,c]])
```

```
            mid2=np.dot(mid1,[g.node[death]['strategy'],1-g.node[death]['strategy']])
            g.node[d]['payoff']-=mid2
            if g.degree(d)!=0:
                g.node[d]['fitness']=1-w+w*g.node[d]['payoff']/float(g.degree(d))
        g.remove_node(death)
        i+=1
    return result


# repeat the evolution process for K times
def rep_evol_game(N,r_1,p,r_2,c,w,u,M,K):
    result=[]
    for i in range(K):
        mid=evol_game_2(N,r_1,p,r_2,c,w,u,M)
        mid1=mid[4000:]
        mid3=sum(mid1)/float(len(mid1))
        result.append(mid3)

    return result
#


# N,r is used to initialize graph, c is the defection tempation, w is the selection intensity,
u is the mutation rate, M is the steps
def evol_game_compare1(N,r_1,p,c,w,u,M):
    g=initiate_graph(N,r_1,p)
    g=calculate_average_fitness(g,[[1,0],[1+c,c]],w)
    i=1
    result=[]
    while i<M:
```

```
        cooper=nx.get_node_attributes(g,'strategy')
        aver_cooper=sum(cooper.values())/float(g.order())
        result.append(aver_cooper)
        birth=selected_birth(g)
        k=i+N-1
        pos_k=[rd.random(),rd.random()]
        g.add_node(k,pos=pos_k,strategy=g.node[birth]['strategy'],payoff=0,fitness=1)
        for d in nx.nodes(g):
            if sqrt((g.node[d]['pos'][0] - pos_k[0]) ** 2+(g.node[d]['pos'][1] - pos_k[1])
** 2)<r_1:
                g.add_edge(d,k)
        g.remove_edge(k,k)
        p=rd.random()
        if p<u/float(2):
            g.node[k]['strategy']=1-g.node[birth]['strategy']
        for d in nx.all_neighbors(g,k):
            mid1=np.dot([g.node[k]['strategy'],1-g.node[k]['strategy']],[[1,0],[1+c,c]])
            mid2=np.dot(mid1,[g.node[d]['strategy'],1-g.node[d]['strategy']])
            g.node[k]['payoff']+=mid2
            mid3=np.dot([g.node[d]['strategy'],1-g.node[d]['strategy']],[[1,0],[1+c,c]])
            mid4=np.dot(mid3,[g.node[k]['strategy'],1-g.node[k]['strategy']])
            g.node[d]['payoff']+=mid4
            g.node[d]['fitness']=1-w+w*g.node[d]['payoff']/float(g.degree(d))
        if g.degree(k)!=0:
            g.node[k]['fitness']=1-w+w*g.node[k]['payoff']/float(g.degree(k))
        nodes=g.nodes()
        death=rd.sample(nodes,1)[0]
        for d in nx.all_neighbors(g,death):
```

```
            mid1=np.dot([g.node[d]['strategy'],1-g.node[d]['strategy']],[[1,0],[1+c,c]])
            mid2=np.dot(mid1,[g.node[death]['strategy'],1-g.node[death]['strategy']])
            g.node[d]['payoff']-=mid2
            if g.degree(d)!=0:
                g.node[d]['fitness']=1-w+w*g.node[d]['payoff']/float(g.degree(d))
        g.remove_node(death)
        i+=1
    return result
#
# N,r is used to initialize graph, c is the defection tempation, w is the selection intensity, u is the mutation rate, M is the steps
def evol_game_compare2(N,r_1,p,c,w,u,M):
    g=initiate_graph(N,r_1,p)
    g=calculate_average_fitness(g,[[1,0],[1+c,c]],w)
    i=1
    result=[]
    result2=[]
    while i<M:
        cooper=nx.get_node_attributes(g,'strategy')
        aver_cooper=sum(cooper.values())/float(g.order())
        result.append(aver_cooper)
        result2.append(nx.average_clustering(g))
        birth=selected_birth(g)
        k=i+N-1
        g.add_node(k,pos=[0,0],strategy=g.node[birth]['strategy'],payoff=0,fitness=1)
        nodes=g.nodes()
        death=rd.sample(nodes,1)[0]
        if death==k:
```

```
            g.remove_node(death)

        else:

            g.node[k]['pos']=[g.node[death]['pos'][0],g.node[death]['pos'][1]]

            for d in nx.all_neighbors(g,death):

                g.add_edge(d,k)

            p=rd.random()

            if p<u/float(2):

                g.node[k]['strategy']=1-g.node[birth]['strategy']

            for d in nx.all_neighbors(g,k):

                mid1=np.dot([g.node[k]['strategy'],1-
g.node[k]['strategy']],[[1,0],[1+c,c]])

                mid2=np.dot(mid1,[g.node[d]['strategy'],1-g.node[d]['strategy']])

                g.node[k]['payoff']+=mid2

                mid3=np.dot([g.node[d]['strategy'],1-
g.node[d]['strategy']],[[1,0],[1+c,c]])

                mid4=np.dot(mid3,[g.node[k]['strategy'],1-g.node[k]['strategy']])

                g.node[d]['payoff']+=mid4

                mid5=np.dot([g.node[d]['strategy'],1-
g.node[d]['strategy']],[[1,0],[1+c,c]])

                mid6=np.dot(mid5,[g.node[death]['strategy'],1-g.node[death]['strategy']])

                g.node[d]['payoff']-=mid6

                g.node[d]['fitness']=1-w+w*g.node[d]['payoff']/float(g.degree(d))

            if g.degree(k)!=0:

                g.node[k]['fitness']=1-w+w*g.node[k]['payoff']/float(g.degree(k))

            g.remove_node(death)

        i+=1

    return [result,result2]
```