Supplementary Information:

Indirect reciprocity provides a narrow margin of efficiency for costly punishment

Hisashi Ohtsuki\textsuperscript{1}, Yoh Iwasa\textsuperscript{2} & Martin A. Nowak\textsuperscript{3}

\textsuperscript{1}Department of Value and Decision Science, Tokyo Institute of Technology, Tokyo 152-8552, Japan
\textsuperscript{2}Department of Biology, Faculty of Sciences, Kyushu University, Fukuoka 812-8581, Japan
\textsuperscript{3}Program for Evolutionary Dynamics, Department of Organismic and Evolutionary Biology, Department of Mathematics, Harvard University, Cambridge MA 02138, USA

Supplementary Methods

1 The basic model

Consider an infinitely large population. At each small time interval, $\Delta t$, a fraction $2\Delta t$ of players is randomly sampled from the population to form pairs in order to participate in an evolutionary game. In each pair, one player acts as a donor and the other player as a recipient. The donor has three behavioral choices: cooperation ($C$), defection ($D$), and punishment ($P$). Cooperation involves a cost, $c$, for the donor and a benefit, $b$, for the recipient. Defection has no cost and yields no benefit. Punishment has cost, $\alpha$, for the donor and cost, $\beta$, for the recipient.

Each individual is endowed with a binary reputation, which is either good ($G$) or bad ($B$). In the analytical model, everyone agrees on the reputation of an individual. No private opinions are allowed, but errors in assigning reputation are possible. The donor can base his decision on the recipient’s reputation. After each interaction, the reputation of the donor is updated according to the ‘social norm’ of the population, while the reputation of the recipient remains the same. Then each participant (donor and recipient) goes back to the population with probability $\omega$ ($0 < \omega < 1$), or leaves the population with probability ($1 - \omega$) never to return. The parameter $\omega$ plays a role of a discounting factor of the future. In exchange for each player who leaves the population, a new individual enters with either a good or bad reputation according to the proportion of good and bad players in the current population. Hence, the total population size remains constant.
1.1 Action rules

A donor can base his action on the recipient’s reputation. Each player has an ‘action rule’, $s$, which depends on the recipient’s reputation. A player with an action rule, $s$, takes the action $s(G)$ toward a good recipient, and the action $s(B)$ toward a bad one. Each of $s(G)$ and $s(B)$ can be either $C$, $D$, or $P$. There are $3^2 = 9$ possible action rules: $s(G)s(B) = CC, CD, CP, DC, DD, DP, PC, PD, and PP$.

1.2 Social norms

A social norm, $n$, is used for updating the reputations of players. We assume that all players in the population share the same norm. A donor who has taken the action $X$ ($X = C, D, P$) toward a recipient whose reputation is $J$ ($J = G, B$), is assigned the new reputation $n(J, X)$ (= $G$ or $B$) by the social norm $n$. There are $2^3 \times 2 = 64$ possible social norms. Social norms of this type are based on ‘second-order assessment’\textsuperscript{12,13}: they depend on (i) the action of the donor and (ii) the reputation of the recipient.

1.3 Social resolution

After every interaction, the donor’s reputation is updated by all members of the population. We assume that this process is susceptible to errors. With probability $\mu$, where $0 < \mu < 1/2$, an incorrect reputation is assigned. With probability $1 - \mu$ the correct reputation is assigned. All individuals come to the same conclusion; there are no private lists of reputation.

The social resolution is given by $q = 1 - 2\mu$. This parameter quantifies the ability to distinguish between good and bad. Denoted by $x_G$ and $x_B$ are the fraction of people who would have a good and bad reputation in the absence of errors. Denote by $x_g$ and $x_b$ are the perceived fraction of good and bad people in the presence of errors. Clearly, $x_G + x_B = 1$ and $x_g + x_b = 1$. We have

$$x_g = (1 - \mu)x_G + \mu x_B \quad \text{and} \quad x_b = \mu x_G + (1 - \mu)x_B.$$  

(1)

From eq.(1) we obtain $x_g - x_b = q(x_G - x_B)$. Therefore, the perceived difference between good and bad, $x_g - x_b$, is given by the actual difference, $x_G - x_B$, times $q$. Thus, $q$ can be interpreted as the
social resolution between good and bad. If \( q = 1 \) the social resolution is perfect. If \( q = 0 \) there is no distinction between good and bad.

2 A search for cooperative ESS

We want to find action rules that are evolutionarily stable strategies (ESS) under a given social norm. Coexistence of action rules\(^{10} \) is not the scope of our analysis. An action rule is an ESS if and only if it can resist invasion by any of the other 8 action rules (see Figure S1). More formally speaking, we will search for the combination of an action rule and a social norm, \((s, n)\), that satisfies the following two criteria.

**Cooperativity (CO)** : more than a fraction \( 1 - \mu \) of all game interactions are cooperative.

**Evolutionary stability (ES)** : under the social norm \( n \), the action rule \( s \) is evolutionarily stable against any other action rule \( s' \neq s \).

2.1 Cooperativity

Consider a monomorphic population that adopts \((s, n)\). For notational convenience, let us define \( \delta_G \) and \( \delta_B \) as

\[
\delta_G = \begin{cases} 
1 & \text{if } n(G, s(G)) = G \\
0 & \text{if } n(G, s(G)) = B 
\end{cases} \quad \text{and} \quad \delta_B = \begin{cases} 
1 & \text{if } n(B, s(B)) = G \\
0 & \text{if } n(B, s(B)) = B 
\end{cases}.
\]

Namely, \( \delta_G(\delta_B) \) equals one if a player using action rule \( s \) gains a good reputation after interacting with a good(bad) recipient. Otherwise it is zero.

Let us first derive a differential equation describing the fraction of good players in the population. We denote the fraction of good players by \( g \). Taking errors into account, we obtain

\[
\frac{dg}{dt} = \omega \left[ g \left( (1 - \mu)\delta_G + \mu(1 - \delta_G) \right) + (1 - g) \left( (1 - \mu)\delta_B + \mu(1 - \delta_B) \right) - g \right]
= \omega \left[ (1 - 2\mu) \left( g\delta_G + (1 - g)\delta_B \right) + \mu - g \right].
\]

We assume that the fraction, \( g \), is at the equilibrium. From eq.(3), the abundance of good players
Figure S1: Action rules and social norms. The action rule specifies for the donor to cooperate, $C$, defect, $D$, or punish, $P$ against a recipient, whose reputation is either good, $G$, or bad $B$. The social norm is used to update the donor’s reputation taking into account (i) the donor’s action and (ii) the recipient’s reputation. Each social norm specifies which reputation to assign for all 6 possible scenarios. This leads to $2^6 = 64$ social norms. For each of the 64 social norms we search for evolutionarily stable action rules that are cooperative. The $DD$ action rule is always evolutionarily stable. For some norms, either the $CD$ or the $CP$ action rule is also stable. Here we show two social norms. One of them allows the evolutionary stability of $CD$, the other one of $CP$. 
at the equilibrium, $g^*$, is calculated as

$$
g^* = \begin{cases} 
1 - \mu & \text{if } (\delta_G, \delta_B) = (1, 1) \\
1/2 & \text{if } (\delta_G, \delta_B) = (1, 0), (0, 1) \\
\mu & \text{if } (\delta_G, \delta_B) = (0, 0)
\end{cases}
$$

We show if $g^* = 1/2$ then the CO and ES criteria cannot be satisfied at the same time. For that purpose, imagine $g^* = 1/2$ holds. A player meets a good recipient with probability one-half and meets a bad one with probability one-half. To achieve the CO criterion, the action rule, $s$, must prescribe cooperation with both types of recipients, i.e. $s(G) = s(B) = C$. However this implies that the action rule $s$ is $CC$ (= always cooperate), which is obviously susceptible to the invasion by $DD$ action rule (=always defect). Thus the ES criterion is not satisfied.

Since the CO criterion must be satisfied, two possibilities remain: (i) ‘$(\delta_G, \delta_B) = (1, 1)$ and $s(G) = C$’, or (ii) ‘$(\delta_G, \delta_B) = (0, 0)$ and $s(B) = C$’. The former scenario corresponds to the situation where a majority of players are good and they cooperate with good recipients. In the latter scenario, a majority of players are bad and they cooperate with bad recipients. There exists a complete symmetry between two labels, ‘good’ and ‘bad’. We can swap them without changing anything. Therefore, to break the symmetry we adopt the former scenario. Note that the argument here is theoretically equivalent to restricting our attention to action rules that cooperate with good players, $s(G) = C$, in the main text.

In summary, we have obtained $(\delta_G, \delta_B) = (1, 1)$ and $s(G) = C$. These equations can be rewritten as $s(G) = C$, $n(G, C) = G$, and $n(B, s(B)) = G$.

### 2.2 Evolutionary stability

Let us now search for the combination $(s, n)$ that satisfies the ES criterion. Remember that if $s$ is the unique best response to itself (meaning $s$ is a strict Nash equilibrium), then the action rule $s$ is evolutionarily stable. This fact enables us to search for evolutionarily stable strategies by using dynamic optimization. Below we will derive the best response, that is, the action rule which maximizes player’s payoff, under the assumption that the other players adopt $s$ and that the social norm of the population is $n$.

Imagine a monomorphic population of $(s, n)$ that satisfies $s(G) = C$, $n(G, C) = G$, and $n(B, s(B)) =$
Let $W_{I,J}$ denote the maximum payoff that a player, currently having reputation $I$ (\(= G \) or $B$) and being matched with a player with reputation $J$ (\(= G \) or $B$), can gain from this interaction to future. Also let us define the cost function, $\xi$, and the benefit function, $\eta$, of each action, $X$, as follows:

| Action | $\xi[X]$ | $\eta[X]$ |
|--------|----------|----------|
| $C$    | $c$      | $b$      |
| $D$    | $0$      | $0$      |
| $P$    | $\alpha$| $-\beta$|

The Bellman equation \(^{31}\) (also called dynamic programming equation\(^{32}\)) of $W_{I,J}$ is given by

$$W_{I,J} = \max_{X=C,D,P} \left\{ \frac{1}{2} \left\{ -\xi[X] + \omega W_{(1-\mu)n(J,X)+\mu n(J,X),(1-\mu)G+\mu B} \right\} + \frac{1}{2} \left\{ \eta[s(I)] + \omega W_{I,(1-\mu)G+\mu B} \right\} \right\}. $$ \hspace{1cm} (5)

Here we use the notation, $\overline{G} = B$, and $\overline{B} = G$. We have also introduced the convenient notation, $W_{y_1G+y_2B,z_1G+z_2B} \equiv y_1z_1W_{G,G} + y_1z_2W_{G,B} + y_2z_1W_{B,G} + y_2z_2W_{B,B}$. Inside the first curly bracket on the r.h.s. of eq.(5) is the sum of the cost that a focal player pays as a donor and the benefit he gains in the future. By taking the action $X$, the donor’s reputation changes to $n(J,X)$ with probability $1 - \mu$, and to $n(J,X)$ with probability $\mu$. This is represented as $(1-\mu)n(J,X)+\mu n(J,X)$ in eq.(5). His opponent in the next round is randomly sampled from the population. Since the fraction of good players in the population is $g^* = 1 - \mu$ (eq.(4)), his next opponent is good with probability $1 - \mu$ and bad with probability $\mu$, leading to $(1 - \mu)G + \mu B$ in eq.(5). Inside the second curly bracket is the sum of the benefit he gains as a recipient and his future benefit. Because the reputation of a recipient remains unchanged after a game interaction, and because his next opponent is randomly sampled from the population, his future benefit is calculated as $W_{I,(1-\mu)G+\mu B}$, discounted by $\omega$. The factors $1/2$ represent the fact that one acts either as a donor or as a recipient with equal probability. It is easy to see that the solution of the maximization problem in eq.(5) is independent of $I$. Thus the best response to $s$, denoted by $s^*$, is

$$ s^*(J) = \arg \max_{X=C,D,P} \left\{ \frac{1}{2} \left\{ -\xi[X] + \omega W_{(1-\mu)n(J,X)+\mu n(J,X),(1-\mu)G+\mu B} \right\} \right\} \hspace{1cm} (6) $$

for $J = G, B$.

Solving eq.(6) is straightforward. First, let us define the ‘advantage of being a good player’, denoted by $v$, as

$$ v \equiv W_{G,(1-\mu)G+\mu B} - W_{B,(1-\mu)G+\mu B}. $$ \hspace{1cm} (7)
This is easily calculated from eq. (5) as
\[
v = W_{G,(1-\mu)G+\mu B} - W_{B,(1-\mu)G+\mu B}
= \frac{1}{2} \{ \eta[s(G)] - \eta[s(B)] + \omega (W_{G,(1-\mu)G+\mu B} - W_{B,(1-\mu)G+\mu B}) \} 
= \frac{1}{2} \{ \eta[s(G)] - \eta[s(B)] + \omega v \} .
\] (8)

Thus we obtain
\[
v = \frac{\eta[s(G)] - \eta[s(B)]}{2 - \omega} .
\] (9)

Observe that eq. (6) is rewritten as
\[
s^*(J) = \begin{cases} 
\arg \max_{X=C,D,P} \left[ \frac{1}{2} \left\{ -\xi[X] + \omega W_{(1-\mu)G+\mu B,(1-\mu)G+B} \right\} \right] & \text{(if } n(J,X) = G) \\
\arg \max_{X=C,D,P} \left[ \frac{1}{2} \left\{ -\xi[X] + \omega W_{(1-\mu)B+\mu G,(1-\mu)B+G} \right\} \right] & \text{(if } n(J,X) = B) \\
\end{cases}
\]
\[
= \arg \max_{X=C,D,P} \left[ \frac{1}{2} \left\{ -\xi[X] + \omega W_{(1-\mu)B+\mu G,(1-\mu)B+G} \right\} \right]
\]
\[
= \arg \max_{X=C,D,P} \left[ \frac{1}{2} \left\{ -\xi[X] + \omega q \cdot v \chi_{G}[n(J,X)] \right\} + \frac{\omega}{2} W_{(1-\mu)B+\mu G,(1-\mu)B+G} \right]
\]
\[
= \arg \max_{X=C,D,P} \left[ \frac{1}{2} \left\{ -\xi[X] + \omega q \cdot v \chi_{G}[n(J,X)] \right\} \right] 
\] (10)

where \( \chi_{G}[G] = 1 \) and \( \chi_{G}[B] = 0 \). Thus, the maximization problem is deduced to finding the action \( X \) (\( X = C, D, P \)) that maximizes \( -\xi[X] + \omega q \cdot v \chi_{G}[n(J,X)] \). The first term, \( -\xi[X] \), represents the immediate cost of action \( X \). The second term, \( \omega q \cdot v \chi_{G}[n(J,X)] \), represents the future benefit through becoming a good player via action \( X \), which is \( v \chi_{G}[n(J,X)] \), multiplied by the discounting factor \( \omega \) and the social resolution, \( q = 1 - 2\mu \). Hence we are able to derive \( s^*(G) \) and \( s^*(B) \).

For each combination of \( (s,n) \) that satisfies the CO criterion, we derive the best response \( s^* \) in the way described above. We do not consider the ungeneric case where the best response is not unique. The E\( S \) criterion is satisfied, if and only if \( s^* = s \) holds.


3 Result of the search

In order to describe the combination of \( (s, n) \) that satisfies both the CO and ES criteria we use the following notation:

| norm | \( G \) | \( B \) |
|------|--------|--------|
| \( C \) | \( n(G, C) \) | \( n(B, C) \) |
| \( D \) | \( n(G, D) \) | \( n(B, D) \) |
| \( P \) | \( n(G, P) \) | \( n(B, P) \) |

| action rule | \( s(G) \) | \( s(B) \) |

The average total payoff in the population is given by \( W_{1-\mu G+\mu B,(1-\mu)G+\mu B} \). All the results in the main text are obtained by taking the limit \( \omega \uparrow 1 \). Note that the asterisks in the tables below represent wild-cards: they can be either \( G \) or \( B \).

3.1 Cost of punishment is smaller than cost of cooperation

If \( \alpha < c \) then the following \( (s, n) \) pairs satisfy both the CO and the ES criteria:

| norm | \( G \) | \( B \) | ESS condition: \( \omega qb > (2 - \omega)c \) | Average payoff: \( \frac{1}{2} \left[ (1 - \mu) \frac{b - c}{1 - \omega} \right] \) |
|------|--------|--------|------------------|------------------|
| \( C \) | \( G \) | * | \( \omega qb > (2 - \omega)c \) | \( \frac{1}{2} \left[ (1 - \mu) \frac{b - c}{1 - \omega} \right] \) |
| \( D \) | \( B \) | \( G \) | \( \omega qb > (2 - \omega)c \) | \( \frac{1}{2} \left[ (1 - \mu) \frac{b - c}{1 - \omega} \right] \) |
| \( P \) | \( B \) | * | \( \omega qb > (2 - \omega)c \) | \( \frac{1}{2} \left[ (1 - \mu) \frac{b - c}{1 - \omega} \right] \) |

| action rule | \( C \) | \( D \) |

The average payoff is lower if the action rule uses costly punishment, \( P \), but the ESS condition is less restrictive.

3.2 Cost of punishment is larger than cost of cooperation

If \( \alpha > c \) then the following \( (s, n) \) pairs satisfy both the CO and the ES criteria:

| norm | \( G \) | \( B \) | ESS condition: \( \omega qb > (2 - \omega)c \) | Average payoff: \( \frac{1}{2} \left[ (1 - \mu) \frac{b - c}{1 - \omega} - \mu \frac{\alpha + \beta}{1 - \omega} \right] \) |
|------|--------|--------|------------------|------------------|
| \( C \) | \( G \) | * | \( \omega qb > (2 - \omega)c \) | \( \frac{1}{2} \left[ (1 - \mu) \frac{b - c}{1 - \omega} - \mu \frac{\alpha + \beta}{1 - \omega} \right] \) |
| \( D \) | \( B \) | \( G \) | \( \omega qb > (2 - \omega)c \) | \( \frac{1}{2} \left[ (1 - \mu) \frac{b - c}{1 - \omega} - \mu \frac{\alpha + \beta}{1 - \omega} \right] \) |
| \( P \) | * | * | \( \omega qb > (2 - \omega)c \) | \( \frac{1}{2} \left[ (1 - \mu) \frac{b - c}{1 - \omega} - \mu \frac{\alpha + \beta}{1 - \omega} \right] \) |

action rule |\( C \) | \( D \) |
ESS condition: $\omega q (b + \beta) > (2 - \omega) \alpha$

Average payoff: $\frac{1}{2} \left[ (1 - \mu) \frac{b - c}{1 - \omega} - \mu \frac{\alpha + \beta}{1 - \omega} \right]$

Again, the average payoff is lower if the action rule uses costly punishment, $P$.

### 3.3 Classification of social norms

Brandt & Sigmund\textsuperscript{12} have classified social norms according to the amount of information that is used when updating a player’s reputation. Norms with ‘first-order assessment’ use only the action of the donor. For example, it might be good to cooperate and bad to defect. Norms with ‘second-order assessment’ use the action of the donor and the reputation of the recipient. For example, it might be good to cooperate with good recipients, but bad to cooperate with bad recipients. Norms with ‘third-order assessment’ use (i) the action of the donor, (ii) the reputation of the recipient and (iii) the reputation of the donor. Table S1 summarizes these three classes. Our present study considers norms with first and second order assessment. We find that second order assessment is necessary for evolutionary stability of cooperation in our analysis. Any social norm that stabilizes cooperation must take into account the donor’s action and the recipient’s reputation.

#### Table S1: Information used by different classes of social norms

| social norm         | donor’s action ($X$) | recipient’s reputation ($J$) | donor’s reputation ($I$) |
|---------------------|----------------------|----------------------------|--------------------------|
| 1st order assessment| yes                  | no                         | no                       |
| 2nd order assessment| yes                  | yes                        | no                       |
| 3rd order assessment| yes                  | yes                        | yes                      |

Most studies of indirect reciprocity so far have focused on games without punishment (but see reference 4). In this case, players choose either cooperation ($C$) or defection ($D$). A well-known social norm of first-order assessment is scoring (Table S2), which always regards cooperation as a good action and defection as a bad action, irrespective of the recipient’s and the donor’s reputations.

Table S2 shows three examples of second-order social norms. Under ‘simple-standing’, only defection against a good recipient leads to a bad reputation, while every other behavior leads to
a good reputation. Under ‘stern-judging’ (also called ‘Kandori’), a good reputation is achieved by cooperation with a good recipient and by defection against a bad recipient, while the reverse behavior leads to bad reputation. Under ‘shunning’, only cooperation with a good recipient leads to a good reputation, while every other behavior leads to a bad reputation Table S2 also shows two examples of third-order social norms. ‘Standing’ differs from ‘simple-standing’ and ‘judging’ differs from ‘stern-judging’, in that a bad player defecting against a bad player remains bad. For references see references 10,13,17, and 33.

Table S2: Examples of social norms. $X$ represents the donor’s action. $I$ and $J$ denote, respectively, the donor’s and the recipient’s reputation.

| order | social norm | $X$ | $IJ$ | GG | GB | BG | BB |
|-------|-------------|-----|------|-----|-----|-----|-----|
| 1st   | scoring     | $C$ | $G$  | $G$ | $G$ | $G$ |
|       |             | $D$ | $B$  | $B$ | $B$ | $B$ |
| 2nd   | simple-standing | $C$ | $G$  | $G$ | $G$ | $G$ |
|       |             | $D$ | $B$  | $B$ | $G$ | $G$ |
|       | stern-judging | $C$ | $G$  | $B$ | $G$ | $B$ |
|       |             | $D$ | $B$  | $B$ | $G$ | $G$ |
|       | shunning    | $C$ | $G$  | $B$ | $G$ | $B$ |
|       |             | $D$ | $B$  | $B$ | $B$ | $B$ |
| 3rd   | standing    | $C$ | $G$  | $G$ | $G$ | $G$ |
|       |             | $D$ | $B$  | $G$ | $B$ | $B$ |
|       | judging     | $C$ | $G$  | $B$ | $G$ | $B$ |
|       |             | $D$ | $B$  | $G$ | $B$ | $B$ |

In the present paper we observe that social norms that stabilize the $CD$ action rule are based on ‘simple-standing’ or ‘stern-judging’. Social norms that stabilize the $CP$ action rule are based on ‘scoring’ or ‘shunning’.

4 Computer simulations

4.1 Evolutionary stability and average payoffs

To confirm our analytic predictions, we have run individual-based computer simulations. We study a population of fixed size, $N = 100$. Each new player receives an initial reputation, which is either
good or bad with equal probability. Each player adopts one of 9 possible action rules. All players share the same social norm that is fixed in the population. In any one elementary step of updating, each individual has exactly 10 interactions with other randomly chosen individuals. Individuals play donor and recipient on average 5 times each. After each interaction the reputation of the donor is updated according to the social norm, but with probability $\mu = 0.02$ a wrong reputation is assigned. In this case, every player agrees on the wrong reputation of this particular player. No private lists of reputation are considered. (Later we will relax this assumption.) After all interactions have taken place, an individual is chosen for reproduction with a probability proportional to $P_i - P_{\text{min}}$, where $P_i$ is the total payoff of the individual and $P_{\text{min}}$ is the minimum payoff in the population. The offspring inherits the action rule of the parent and replaces another random player. Mutation occurs with probability $\epsilon = 0.01$. In this case the action rule of the offspring is randomly chosen out of all 9 possibilities. After reproduction, the payoffs of all players are reset to zero. Thus, older players do not accumulate their payoffs. Each generation consists of $N = 100$ elementary steps of updating. We have run simulations for 1000 generations.

Figure S2 shows the frequencies of action rules and the average payoff per round when $b > c$. On the left, we study the social norm which is based on ‘stern-judging’. The simulation result is consistent with our analytical prediction that the $CD$ action rule is an ESS. On the right, we study the social norm which is based on ‘shunning’. The simulation result is consistent with our analytical prediction that the $CP$ action rule is an ESS. Comparing these two simulations, we find that the $CD$ action rule achieves a higher average payoff than the $CP$ action rule. Again this observation is in agreement with the analytical prediction.

In Figure S3, we study the case of $b < c$. On the left, the initial dominance of the $CD$ action rule is not maintained. On the right, the $CP$ action rule remains dominant. The $CP$ rule is evolutionarily stable because of the large effect of punishment ($\beta = 4$). However, the existence of $CP$ action rule leads to a negative average payoff. Therefore, always defect, $DD$, would be a more profitable action rule for the group. All of those observations are in agreement with our analytical theory.
Figure S2: Simulation results for \( b > c \). The top panels show the social norms that are being used by all players of the population. The middle panels show the frequencies of all 9 action rules. The bottom panels show the average payoff per round. On the left, we study a social norm based on ‘stern-judging’. The initial frequency of the \( CD \) action rule is 0.8. The initial frequencies of the other 8 action rules are randomly chosen. We find that the \( CD \) action rule is stable against invasion attempts by \( CC \) (dark blue) and \( CP \) (orange). The average payoff per round fluctuates with its maximum being 2.94, which agrees with our analytic prediction, \((1 - \mu)(b - c)/2 = 2.94\). Fluctuations are due to stochasticity in the simulation. On the right, we study a social norm based on ‘shunning’. The initial frequency of \( CP \) is 0.8. \( CP \) is stable against other action rules. The average payoff per round fluctuates with its maximum being 2.89, which also agrees with our analytic prediction, \(\{(1 - \mu)(b - c) - \mu(\alpha + \beta)\}/2 = 2.89\). Parameter values: \( b = 9, c = 3, \alpha = 1, \beta = 4, \mu = 0.02, \) and \( \epsilon = 0.01 \).
Figure S3: Simulation results for $b < c$. Everything is the same as in Fig.S2 except that $b = 2$ is used here. On the left, the CD action rule, whose initial frequency is 0.8, is immediately invaded by DD (light blue). This observation agrees with our theoretical prediction that the CD rule is not evolutionarily stable for this parameter region. The average payoff is around 0. On the right, the CP action rule is robust against invasion by the other 8 action rules, although cooperation is non-productive because $b < c$. The average payoff is negative and is about -0.54. Hence the group does not benefit from the cooperation that is enforced by costly punishment.
4.2 Various types of errors

In this section, we study how different kinds of errors affect our simulation results. In Figure S4, we explore errors in executing the action. In this simulation, a player fails to perform his intended action with probability $\epsilon_{\text{act}} = 0.02$. If such an error occurs, then the player takes one of the other two unintended actions at random. There are no errors in assigning reputation; $\mu = 0$. We have found that the robustness of both ESS in Fig.S2 remains unchanged, although the average payoff is slightly smaller than that of Fig.S2. The average payoff realized by the $CD$ action rule is greater than that realized by the $CP$ rule.

In Figure S5, we study errors that occur in the recalling of reputation. As in the previous setting, all players agree on the actual reputation of each player, but in a particular interaction a donor temporarily fails to recall the recipient’s reputation. This mistake happens with probability $\epsilon_{\text{rep}} = 0.02$. Thus, the donor who commits this error might behave differently toward the recipient. This one-time error does not modify the actual reputation of the recipient. Figure S5 shows that our theoretical predictions are robust under errors in recalling reputation.
Figure S4: Simulation results when errors in executing action ($e_{act} = 0.02$) are possible. There are no errors in assigning reputation ($\mu = 0$). All other parameters and settings are the same as in Fig.S2.
Figure S5: Simulation results when errors in recalling reputation ($e_{rep} = 0.02$) are possible. There are no errors in assigning reputation ($\mu = 0$). All other parameters and settings are the same as in Fig.S2.
4.3 Private reputation

Our analytical theory assumes that everyone has the same opinion concerning the reputation of each individual in the population. Here we relax this assumption of public reputation and study private reputation. In the new computer simulation, each player has his own list of the reputation of others. It is possible that players have different opinions on the same player. In assigning a new reputation to the donor of an interaction, each player independently commits an error with probability $\mu$.

Figure S6 shows the result of computer simulations. Even with a small amount of errors ($\mu = 0.02$), the stability of both $CD$ and $CP$ is lost. The reason is that a privately committed error triggers other errors in reputation assignment in future rounds. This accumulation of errors causes the opinions of players to be completely mixed, which leads to the collapse of the reputation system. Therefore, when there are no mechanisms to maintain coherence in opinion among individuals, cooperation is destroyed.

One way to overcome this problem of error accumulation is to introduce ‘communication rounds’ into the model. Between game interactions, players communicate with each other and adjust their opinions. We assume that communication occurs after every $N/2 = 50$ game interactions. For each communication event, we choose three players, $i, j, k$, at random. Player $i$ asks player $j$ about player $k$’s reputation. Player $i$ adopts $j$’s opinion on $k$. We allow many rounds of communication. Each player has on average $T$ chances of asking in communication rounds.

Figure S7 shows a typical result for $T = 50$. The $CP$ action rule can be stably maintained, but the $CD$ action rule cannot. It is because under the norm on the left (based on stern-judging), a player with a $DD$ action rule sometimes gains a good reputation, whereas the norm on the right (based on shunning) always gives a player with a $DD$ action rule a bad reputation. The result is qualitatively consistent with our finding in section 3.1 that the $CP$ action rule has a less restrictive condition for evolutionary stability than the $CD$ action rule when $\alpha < c$ (note that $\alpha = 1$ and $c = 3$ are used in Figures S6-8).

Once the number of communication rounds is increased to $T = 100$, we observe that both $CD$ and $CP$ are stably maintained under the corresponding social norms. See Figure S8.
Figure S6: Results of computer simulations with private reputation. The error rate for reputation assignment is $\mu = 0.02$, but these errors occur privately. The other parameters and settings are the same as in Fig.S2. Neither $CD$ nor $CP$ is stable.
Figure S7: Simulation results when private lists of reputation are allowed but players can adjust their opinions through communication. After every $N/2 = 50$ game interactions, each player has on average $T = 50$ chances of asking in communication rounds. Parameters are the same as in Fig.S6.
Figure S8: Parameters and settings are the same as in Fig.S7 but $T = 100$. 
5 Comparison of different ESS rules

We have found two different types of cooperative ESS rules. One is the CD action rule, which prescribes cooperation with good recipients and defection with bad recipients. It does not use costly punishment. The other ESS is achieved by the CP action rule, which prescribes cooperation with good recipients and costly punishment of bad recipients. Furthermore, the DD action rule always achieves an uncooperative ESS for any social norm. Here we compare these three types of ESS. The following result is obtained by taking the $\omega \uparrow 1$ limit in Section 3.

There are five parameters in our model: $b, c, \alpha, \beta$, and $q$. For each parameter region we ask which types of ESS are possible. If only one type of ESS is possible, then this must be DD. If multiple types are possible, we ask which one leads to the highest average payoff for the population. This is a concept of Pareto efficiency. We obtain the following result.

(i) If $q > c/b$, then CD rule is the ESS with the highest average payoff.

(ii) If $c/b > q > c/(b + \beta)$, CD rule is not an ESS. DD is always an ESS. CP is an ESS for $q > \alpha b + \beta$.

If eq.(11) holds, then the average payoff of CP is greater than that of DD if

$$q > \frac{\alpha}{\beta + \beta}.$$  \hspace{1cm} (11)

If eq.(11) holds, then the average payoff of CP is greater than that of DD if

$$q > \frac{\alpha + \beta - (b - c)}{(\alpha + \beta) + (b - c)}.$$  \hspace{1cm} (12)

Therefore, CP is the ESS with the highest average payoff if and only if eqs.(11, 12) hold.

For $b > c$, it is possible to find a social resolution, $q$, that satisfies eqs.(11, 12) if and only if the following two conditions hold:

$$1 + \frac{\beta}{b} > \frac{\alpha}{c} \quad \text{and} \quad b + c > \alpha + \beta.$$  \hspace{1cm} (13)

(iii) If $c/(b + \beta) > q$, then only DD is the ESS.

In Figure 3 of the main paper we illustrate the various parameter regions. We find that there is only a small parameter region where the CP action rule is stable and allows the highest average payoff. In this sense, costly punishment has only a narrow margin of efficiency under indirect reciprocity. For higher values of $q$ this margin becomes even smaller and eventually disappears.
The efficiency argument alone, however, does not imply that costly punishment is unimportant for indirect reciprocity. A population could be stuck with an inefficient (sub-optimum) ESS for a long time. In particular, it seems to be complicated to move from a social norm that stabilizes a \( CP \) action rule to another social norm that stabilizes a \( CD \) action rule.

One way to select for an efficient ESS of indirect reciprocity is offered by a ‘contingent movement’ model (for another example of selection of norms, see references 15 and 17). In order to test this idea, we have designed a computer simulation, where players can preferentially move between two groups that have two different social norms.

In this new simulation, each group size can change over time, but the total population size is fixed at \( N = 200 \). Initially, each group has \( N/2 = 100 \) players. The social norm of group 1 is based on stern-judging, which makes the \( CD \) action rule an ESS. The social norm of group 2 is based on shunning, which makes the \( CP \) action rule an ESS. Initially there are 80 \( CD \) players in group 1 and 80 \( CP \) players in group 2. The remaining 40 players (20 in each group) have randomly assigned action rules at the beginning of the simulation.

The simulation proceeds as follows. Within each group, each individual has 10 pairwise interactions; on average 5 times as a donor and 5 times as a recipient. There are no group-size effects on the number of games one plays. After these interactions, one player (=learner) is randomly chosen from the WHOLE population. The learner randomly samples a ‘teacher’ from HIS OWN GROUP with probability 0.8. With the remaining probability, 0.2, he randomly samples a teacher from the WHOLE population. The learner compares his payoff with that of the teacher. Whenever the teacher’s payoff is greater than that of the learner’s, the learner (i) imitates the teacher’s action rule and (ii) enters the teacher’s group and adopts the corresponding social norm. In case the learner moves to the other group, his reputation is initialized with either good or bad with equal probability. Reputation is public: no private lists of reputation are allowed.

Figure S9 shows the result of the computer simulation. We use \( b = 9 \), \( c = 3 \), \( \alpha = 1 \), \( \beta = 4 \), and \( \mu = 0.02 \). For these parameter values, both the \( CD \) and the \( CP \) action rules are ESS under their respective social norms. Our theory predicts that the average payoffs in groups 1 and 2 are 2.94 and 2.89, respectively. We find that the group with the higher average payoff (group 1) is preferentially
selected by players. Eventually all players have adopted the social norm of group 1, which allows the evolutionary stability of the $CD$ action rule and does not use costly punishment.
Figure S9

Figure S9: Simulation results of our 'contingent movement' model. Left and right panels are for group 1 and 2, respectively. Top panels show the social norm in each group. In the bottom panels, black lines show the size of each group. Green and orange lines show, respectively, the number of players with the $CD$ action rule in group 1 and the number of players with the $CP$ action rule in group 2. Group 1 is increasingly popular, while group 2 goes to extinction around $t = 550$. The mutation rate in action rules is set to $\epsilon = 0.01$. 
Supplementary Notes

31. Bellman, R. E. *Dynamic Programming* (Princeton University Press, Princeton, 1957).
32. Mangel, M., Clark, C. W. *Dynamic modeling in behavioral ecology* (Princeton University Press, Princeton, 1988).
33. Ohtsuki, H., Iwasa, Y. Global analyses of evolutionary dynamics and exhaustive search for social norms that maintain cooperation by reputation. *J. Theor. Biol.* **244**, 518-531 (2007).
34. Hamilton, I. M., Taborsky, M. Contingent movement and cooperation evolve under generalized reciprocity. *Proc. R. Soc. B* **272**, 2259-2267 (2005).
Simulation code for Figures S2-8

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>
const int N=100; // population size (must be even)
const int GENERATION=1000; // total number of generations
const int ROUND=10; // total number of rounds per generation
const int UPDATE_MODE=2; // updating rule: 1=Wright-Fisher, 2=Moran
const int MODE=1; // simulation mode (see below)
  // 1: shared norm, public reputation
  // 2: shared norm, private reputation
  // 3: private norm, private reputation

  // action  a: 0=cooperation, 1=defection, 2=punishment
  // reputation  j: 0=good, 1=bad
  
  // public norm (for modes 1 & 2) : 0=good, 1=bad
const int PUBLIC_NORM_CG=0;
const int PUBLIC_NORM_CB=1;
const int PUBLIC_NORM_DG=1;
const int PUBLIC_NORM_DB=0;
const int PUBLIC_NORM_PG=1;
const int PUBLIC_NORM_PB=1;

  // major strategy at the beginning: 0=cooperation, 1=defection, 2=punishment
const int MAJOR_STRATEGY_NUMBER=80;
const int MAJOR_STRATEGY_G=0;
const int MAJOR_STRATEGY_B=1;

  // initial fraction of good players
const double INITIAL_GOOD=0.5;
const int INITIAL_CORRELATION=0; // (for modes 2 & 3) correlation in initial private reputation: 0=no, 1=yes

  // communication round (for mode 2)
const int COMMUNICATION=100; // average number of communication rounds per individual
const int COM_MODE=1; // mode of communication : 1=believe all, 2=believe the good

  // payoff parameters
const double B=9.0;
const double C=3.0;
const double ALPHA=1.0;
const double BETA=4.0;

  // accumulating errors
const double MU=0.02; // error rate in assigning reputation (accumulating; public in mode 1 and private in modes 2 and 3)

  // temporal errors
const double E_ACT=0.00; // error rate in executing intended action
const double E_REP=0.00; // error rate in recalling recipient's reputation (temporal, not accumulating)

  // mutation rates
const double MUTATION_RATE=0.01; // mutation rate in action rules (and in social norms for mode 3)

  // functions

  // displaying strategies
void display_strategy(int** strategy)
{
```
int n, j;

printf("player # : G Bn");
for(n=0; n<N; n++)
{
    printf("player %d : ", n);
    for(j=0; j<2; j++)
    {
        switch(strategy[n][j])
        {
            case 0: printf("C "); break;
            case 1: printf("D "); break;
            case 2: printf("P "); break;
        }
    }
    printf("n");
}

// displaying public_norm
void display_public_norm(int** public_norm)
{
    int a, j;

    printf("-------n");
    printf(" G B <-- PUBLIC NORN");
    for(a=0; a<3; a++)
    {
        switch(a)
        {
            case 0: printf("C :"); break;
            case 1: printf("D :"); break;
            case 2: printf("P :"); break;
        }
        for(j=0; j<2; j++)
        {
            switch(public_norm[a][j])
            {
                case 0: printf("G "); break;
                case 1: printf("B "); break;
            }
        }
        printf("n");
    }
    printf("-------n");
}

// displaying norm
void display_norm(int*** public_norm)
{
    int n, a, j;

    printf("-------n");
    for(n=0; n<N; n++)
    {
        printf(" G B <-- player %d's norm", n);
        for(a=0; a<3; a++)
        {
            switch(a)
            {
                case 0: printf("C :"); break;
                case 1: printf("D :"); break;
                case 2: printf("P :"); break;
            }
            for(j=0; j<2; j++)
        }
    }
}
{ switch(public_norm[n][a][j])
{ case 0: printf("G "); break;
 case 1: printf("B "); break;
}
 printf("n");
}
 printf("------n");
}

// displaying public_reputation
void display_public_reputation(int* public_reputation)
{
 int n;
 for(n=0; n<N; n++)
 { printf("PUBLIC thinks player \%d is ", n);
 switch(public_reputation[n])
 { case 0: printf("Goodn"); break;
 case 1: printf("Badn"); break;
 }
 }
}

// displaying reputation
void display_reputation(int** reputation)
{
 int m, n;
 for(m=0; m<N; m++)
 { for(n=0; n<N; n++)
 { printf("player \%d thinks player \%d is ", m, n);
 switch(reputation[m][n])
 { case 0: printf("Goodn"); break;
 case 1: printf("Badn"); break;
 }
 }
 }
}

// calculating frequency of a given strategy
double frequency(int** strategy, int a1, int a2)
{
 int n;
 int total=0;
 for(n=0; n<N; n++)
 { if(strategy[n][0]==a1&&strategy[n][1]==a2) total++;
 }
 return((double)total/N);
}

// displaying payoff
void display_payoff(double* payoff)
{
int n;
for(n=0; n<N; n++) printf("player %d's payoff = %fn", n, payoff[n]);
}

// calculating average payoff per round
double average_per_round(double* payoff)
{
    int n;
    double total=0.0;
    for(n=0; n<N; n++) total+=payoff[n];
    total = total/(double)N;
    total = total/(double)ROUND;
    return(total);
}

// shifting the minimum payoff to zero
void minimum_to_zero(double* payoff)
{
    int n;
    double min;
    min=payoff[0];
    for(n=1; n<N; n++)
    {
        if(payoff[n]<min) min=payoff[n];
    }
    for(n=0; n<N; n++) payoff[n]=min;
}

// generating a cumulative distribution function (cdf)
void payoff_to_cdf(double* payoff)
{
    int n;
    double total=0.0;
    for(n=0; n<N; n++) total+=payoff[n];
    if(total>0.0)
    {
        for(n=0; n<N; n++) payoff[n]=payoff[n]/total;
        for(n=1; n<N; n++) payoff[n]=payoff[n-1]+payoff[n];
    }
    else
    {
        for(n=0; n<N; n++) payoff[n]=(double)(n+1)/N;
    }
    payoff[N-1]=1.0;
}

int main(void)
{
    int steps, total_steps, round;
    int l, m, n, j, a, a1, a2, reproduce, pair;
    int stack_length;
    int donor_action;
    int times_of_cooperation;
    double average_payoff;
    int rd_int; // random variable (int)
    double rd_double; // random variable (double)
    double *payoff;  // payoff[n] = player n's payoff
    int **strategy;   // strategy[n][j] = player n's action towards j
    int **public_norm; // public_norm[a][j] = evaluation of action a towards j

int ***norm; // norm[n][a][j] = player n's evaluation of action a towards j
int *public_reputation; // public_reputation[n] = player n's public reputation
int **reputation; // reputation[m][n] = player n's reputation in the eyes of m

// for donor-recipient matching
int *stack;
int *donor; // donor[pair]: the donor in #pair
int *recipient; // recipient[pair]: the recipient in #pair
int *parent;

int **old_strategy;
int ***old_norm;

FILE *fp;

// file open
fp = fopen("data.dat", "w");
if(NULL == fp) printf("error!
");

// seed for a random variable
srand((unsigned)time(NULL));

// // dynamic memory allocation
//

payoff = new double [N];
strategy = new int* [N]; old_strategy = new int* [N];
for(n=0; n<N; n++)
{
    strategy[n] = new int [2]; old_strategy[n] = new int [2];
}
if(MODE==1||MODE==2)
{
    public_norm = new int* [3];
    for(a=0; a<3; a++) public_norm[a] = new int[2];
}
if(MODE==3)
{
    norm = new int** [N]; old_norm = new int** [N];
    for(n=0; n<N; n++)
    {
        norm[n] = new int* [3]; old_norm[n] = new int* [3];
        for(a=0; a<3; a++)
        {
            norm[n][a] = new int [2]; old_norm[n][a] = new int [2];
        }
    }
}
if(MODE==1) public_reputation = new int[N];
if(MODE==2||MODE==3)
{
    reputation = new int*[N];
    for(m=0; m<N; m++) reputation[n] = new int[N];
}
stack = new int[N];
donor = new int[N/2];
recipient = new int[N/2];
parent = new int[N];

// ******** initialization of variables (for a whole simulation) ********
//
for(n=0; n<N; n++)
{
  for(j=0; j<2; j++)
  {
    // initialization of strategy[n][j]: 0=cooperation, 1=defection; 2=punishment
    if(n<MAJOR_STRATEGY_NUMBER) // for major strategy
    {
      if(j==0) strategy[n][j]=MAJOR_STRATEGY_G;
      if(j==1) strategy[n][j]=MAJOR_STRATEGY_B;
    }
    else strategy[n][j]=rand() %3; // strategy is assigned randomly to the others
  }
}

if(MODE==1||MODE==2)
{
  for(a=0; a<3; a++)
  {
    for(j=0; j<2; j++)
    {
      // initialization of public_norm[a][j]: 0=good, 1=bad
      if(a==0&&j==0) public_norm[a][j]=PUBLIC_NORM_CG;
      if(a==0&&j==1) public_norm[a][j]=PUBLIC_NORM_CB;
      if(a==1&&j==0) public_norm[a][j]=PUBLIC_NORM_DG;
      if(a==1&&j==1) public_norm[a][j]=PUBLIC_NORM_DB;
      if(a==2&&j==0) public_norm[a][j]=PUBLIC_NORM_PG;
      if(a==2&&j==1) public_norm[a][j]=PUBLIC_NORM_PB;
    }
  }
}

if(MODE==3)
{
  for(n=0; n<N; n++)
  {
    for(a=0; a<3; a++)
    {
      for(j=0; j<2; j++)
      {
        // initialization of norm[n][a][j]: 0=good, 1=bad
        norm[n][a][j]=rand() % 2; // good or bad is randomly assigned
      }
    }
  }
}

// ******** initialization of reputation ********
//
if(MODE==1)
{
  for(n=0; n<N; n++)
  {
    // initialization of public_reputation[n]: 0=good, 1=bad
    if((double)rand()/RAND_MAX<=INITIAL_GOOD) public_reputation[n]=0;
    else public_reputation[n]=1;
  }
}
if (MODE==2 | MODE==3)
{
    if INITIAL_CORRELATION==0 // without initial correlation
    {
        for(m=0; m<N; m++)
        {
            for(n=0; n<N; n++)
            {
                // initialization of reputation[m][n] (how m thinks n): 0=good, 1=bad
                if ((double)rand()/RAND_MAX<INITIAL_GOOD) reputation[m][n]=0;
                else reputation[m][n]=1;
            }
        }
    }
    if INITIAL_CORRELATION==1 // with initial correlation
    {
        for(n=0; n<N; n++)
        {
            // initialization of reputation[m][n] (how m thinks n): 0=good, 1=bad
            if ((double)rand()/RAND_MAX<INITIAL_GOOD)
            {
                for(m=0; m<N; m++) reputation[m][n]=0;
            }
            else
            {
                for(m=0; m<N; m++) reputation[m][n]=1;
            }
        }
    }
}

// calculating total steps required
if UPDATE_MODE==1 total_steps=GENERATION;
if UPDATE_MODE==2 total_steps=GENERATION*N;

// ********** start of simulation ************
//
for(steps=0; steps<total_steps; steps++) // start of one step
{
    // ********** displaying generation & frequencies of strategies at the beginning of generation **********
    //
    if ((UPDATE_MODE==1 & steps % 10 ==0) | (UPDATE_MODE==2 & steps % (10*N) ==0))
    {
        printf(" | CC CD CP DC DD DP PC PD PP | n\n");
    }
    if UPDATE_MODE==1
    {
        printf("gen = %4d |", steps);
        fprintf(fp, "%d", steps);
        for(a1=0; a1<3; a1++)
        {
            for(a2=0; a2<3; a2++)
            {
                printf(" %2.2f", frequency(strategy,a1,a2));
                fprintf(fp, " %f", frequency(strategy,a1,a2));
            }
        }
    }
if(UPDATE_MODE==2)
{
    if(steps % N==0)
    {
        
        printf("gen = %4d |", steps/N);
        fprintf(fp, "%d", steps/N);
        for(a1=0; a1<3; a1++)
        {
            for(a2=0; a2<3; a2++)
            {
                printf(" %2.2f", frequency(strategy,a1,a2));
                fprintf(fp, " %f", frequency(strategy,a1,a2));
            }
        }
    }
}

// ******** initialization of variables (for each generation) ********
//
if((UPDATE_MODE==1)||(UPDATE_MODE==2&&steps % N==0))
{
    times_of_cooperation=0;
    average_payoff=0.0;
}

// ******** initialization of variables (for each step) ********
//
for(n=0; n<N; n++) payoff[n]=0.0;
for(round=0; round<ROUND; round++) // start of one round
{
    
    // ******** donor-recipient pair matching ********
    //
    for(n=0; n<N; n++) stack[n]=n;
    stack_length=N;
    for(pair=0; pair<N/2; pair++)
    {
        rd_int = rand() % stack_length;
        donor[pair]=stack[rd_int];
        stack[rd_int]=stack[stack_length-1];
        stack_length--;
        rd_int = rand() % stack_length;
        recipient[pair]=stack[rd_int];
        stack[rd_int]=stack[stack_length-1];
        stack_length--;
    }
    
    // ******** a game in each pair ********
    //
    for(pair=0; pair<N/2; pair++)
}
// ******** a giving game and payoff calculation ********
//
// error in recalling recipient's reputation
if((double)rand()/RAND_MAX<E_REP) // if error occurs
{
    if(MODE==1) donor_action=strategy[donor[pair]][1-public_reputation[recipient[pair]]];
    if(MODE==2||MODE==3) donor_action=strategy[donor[pair]][1-reputation[donor[pair]][recipient[pair]]];
}
else // if error does not occur
{
    if(MODE==1) donor_action=strategy[donor[pair]][public_reputation[recipient[pair]]];
    if(MODE==2||MODE==3) donor_action=strategy[donor[pair]][reputation[donor[pair]][recipient[pair]]];
}

// error in executing intended action
if((double)rand()/RAND_MAX<E_ACT) // if error occurs
{
    // one of the other two actions is taken randomly
    if((double)rand()/RAND_MAX<0.5) donor_action = (donor_action + 1) % 3;
    else donor_action = (donor_action + 2) % 3;
}
if(donor_action==0) times_of_cooperation++;

switch(donor_action)
{
    case 0: payoff[donor[pair]]-=C; payoff[recipient[pair]]+=B; break;
    case 1: break;
    case 2: payoff[donor[pair]]-=ALPHA; payoff[recipient[pair]]-=BETA; break;
}

// ******** start of updating reputation********
//
if(MODE==1)
{
    public_reputation[donor[pair]]=public_norm[donor_action][public_reputation[recipient[pair]]];
    // error in assignment
    if((double)rand()/RAND_MAX<MU)
    {
        public_reputation[donor[pair]]=1-public_reputation[donor[pair]];
    }
}
if(MODE==2)
{
    for(m=0; m<N; m++)
    {
        reputation[m][donor[pair]]=public_norm[donor_action][reputation[m][recipient[pair]]];
        // error in assignment
        if((double)rand()/RAND_MAX<MU)
        {
            reputation[m][donor[pair]]=1-reputation[m][donor[pair]];
        }
    }
}
if(MODE==3)
{
    for(m=0; m<N; m++)
}
{ 
  reputation[m][donor[pair]]=norm[m][donor_action][reputation[m][recipient[pair]]];

  // error in assignment
  if((double)rand()/RAND_MAX<MU)
  {
    reputation[m][donor[pair]]=1-reputation[m][donor[pair]];
  }
}
} // end of game for each pair

// ****** communication round ******
//
if(MODE==2)
{
  for(j=0; j<N*COMMUNICATION; j++)
  {
    l = rand() % N;
    m= rand() % N;
    n= rand() % N;

    if(COM_MODE==1)
    {
      reputation[l][n] = reputation[m][n]; // l immitates m's opinion on n
    }
    if(COM_MODE==2)
    {
      if(reputation[l][m] == 0) reputation[l][n] = reputation [m][n];
      // l immitates m's opinion on n, only when l thinks m is good
    }
  }
}
} //end of one step

average_payoff+=average_per_round(payoff);

//
// displaying C% and average payoff per generation
//
if(UPDATE_MODE==1)
{
  printf(" | C%% = %5.3f", (double)times_of_cooperation/(ROUND*N/2));
  fprintf(fp, " %f", (double)times_of_cooperation/(ROUND*N/2));

  printf(" | payoff = %6.2fn", average_payoff);
  fprintf(fp, " %fn", average_payoff);
}
if(UPDATE_MODE==2&&steps % N==N-1)
{
  printf(" | C%% = %5.3f", (double)times_of_cooperation/(ROUND*N*N/2));
  fprintf(fp, " %f", (double)times_of_cooperation/(ROUND*N*N/2));

  printf(" | payoff = %6.2fn", (double)average_payoff/N);
  fprintf(fp, " %fn", (double)average_payoff/N);
}

// // reproduction
//
minimum_to_zero(payoff);
payoff_to_cdf(payoff);

if(UPDATE_MODE==1) // Wright-Fisher process
{
    // choosing a parent proportionally to payoff
    // for(n=0; n<N; n++)
    {
        rd_double=(double)rand()/RAND_MAX;
        parent[n]=0;
        while(rd_double>payoff[parent[n]])
        {
            parent[n]++;
        }
    }
    // input to old_strategy and old_norm
    // for(n=0; n<N; n++)
    {
        for(j=0; j<2; j++)
        {
            old_strategy[n][j]=strategy[n][j];
        }
    }
    if(MODE==3)
    {
        for(n=0; n<N; n++)
        {
            for(a=0; a<3; a++)
            {
                for(j=0; j<2; j++)
                {
                    old_norm[n][a][j]=norm[n][a][j];
                }
            }
        }
    }
    // ********** inheritance of strategy & norm **********
    // for(n=0; n<N; n++)
    {
        for(j=0; j<2; j++)
        {
            strategy[n][j]=old_strategy[parent[n]][j];
        }
    }
    if(MODE==3)
    {
        for(a=0; a<3; a++)
        {
            for(j=0; j<2; j++)
            {
                norm[n][a][j]=old_norm[parent[n]][a][j];
            }
        }
    }
    // mutation
    if((double)rand()/RAND_MAX<MUTATION_RATE)
for(j=0; j<2; j++) strategy[n][j]=rand() % 3;
if(MODE==3)
{
    for(a=0; a<3; a++)
    {
        for(j=0; j<2; j++) norm[n][a][j]=rand() % 2;
    }
}

// assigning a new reputation to offspring
if(MODE==1)
{
    for(n=0; n<N; n++)
    {
        // initialization of public_reputation[n]: 0=good, 1=bad
        if((double)rand()/RAND_MAX<=INITIAL_GOOD) public_reputation[n]=0;
        else public_reputation[n]=1;
    }
}

if(MODE==2||MODE==3)
{
    if(INITIAL_CORRELATION==0) // without initial correlation
    {
        for(m=0; m<N; m++)
        {
            for(n=0; n<N; n++)
            {
                // initialization of reputation[m][n] (how m thinks n): 0=good, 1=bad
                if((double)rand()/RAND_MAX<=INITIAL_GOOD) reputation[m][n]=0;
                else reputation[m][n]=1;
            }
        }
    }
    if(INITIAL_CORRELATION==1) // with initial correlation
    {
        for(n=0; n<N; n++)
        {
            if((double)rand()/RAND_MAX<=INITIAL_GOOD)
            {
                for(m=0; m<N; m++) reputation[m][n]=0;
            }
            else
            {
                for(m=0; m<N; m++) reputation[m][n]=1;
            }
        }
    }
}

if(UPDATE_MODE==2) // Moran process
{
    // choosing a dying individual (n)
    n=rand() % N;
    // choosing a parent proportionally to payoff (reproduce)
    rd_double=(double)rand()/RAND_MAX;
    reproduce=0;
    while(rd_double>payoff[reproduce]) reproduce++;
    //
// ********** inheritance of strategy & norm **********
//
for(j=0; j<2; j++)
{
    strategy[n][j]=strategy[reproduce][j];
}
if(MODE==3)
{
    for(a=0; a<3; a++)
    {
        for(j=0; j<2; j++)
        {
            norm[n][a][j]=norm[reproduce][a][j];
        }
    }
}
// mutation
if((double)rand()/RAND_MAX<MUTATION_RATE)
{
    for(j=0; j<2; j++) strategy[n][j]=rand() % 3;
    if(MODE==3)
    {
        for(a=0; a<3; a++)
        {
            for(j=0; j<2; j++) norm[n][a][j]=rand() % 2;
        }
    }
}
// assigning a new reputation to the new-born
if(MODE==1)
{
    // initialization of public_reputation[n]: 0=good, 1=bad
    if((double)rand()/RAND_MAX<=INITIAL_GOOD) public_reputation[n]=0;
    else public_reputation[n]=1;
}
if(MODE==2|MODE==3)
{
    if(INITIAL_CORRELATION==0) // without initial correlation
    {
        for(n=0; n<N; n++)
        {
            // initialization of reputation[m][n] (how m thinks n): 0=good, 1=bad
            if((double)rand()/RAND_MAX<=INITIAL_GOOD) reputation[m][n]=0;
            else reputation[m][n]=1;
        }
    }
    if(INITIAL_CORRELATION==1) // with initial correlation
    {
        // initialization of reputation[m][n] (how m thinks n): 0=good, 1=bad
        if((double)rand()/RAND_MAX<=INITIAL_GOOD) reputation[m][n]=0;
        else reputation[m][n]=1;
    }
}
} // end of simulation
if(MODE==1||MODE==2) display_public_norm(public_norm);

// closing the file
fclose(fp);

// // deleting dynamic memory
//
delete[] payoff;

for(n=0; n<N; n++)
{
    delete[] strategy[n]; delete[] old_strategy[n];
}
delete[] strategy; delete[] old_strategy;

if(MODE==1||MODE==2)
{
    for(a=0; a<3; a++) delete[] public_norm[a];
delete[] public_norm;
}

if(MODE==3)
{
    for(n=0; n<N; n++)
    {
        for(a=0; a<3; a++)
        {
            delete[] norm[n][a]; delete[] old_norm[n][a];
        }
        delete[] norm[n]; delete[] old_norm[n];
    }
delete[] norm; delete[] old_norm;
}

if(MODE==1) delete[] public_reputation;

if(MODE==2||MODE==3)
{
    for(m=0; m<N; m++) delete[] reputation[m];
delete[] reputation;
}

delete[] stack;
delete[] donor;
delete[] recipient;
delete[] parent;
Simulation code for Figure S9

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>

const int N=200; // total population size (must be even)
const int GENERATION=1000; // total number of generations
const int ROUND=10; // total number of rounds per generation (must be even)
const int UPDATE_MODE=2; // updating rule: 2=Moran
const int MODE=1; // simulation mode: 1=shared norm, public reputation

//
// action a: 0=cooperation, 1=defection, 2=punishment
// reputation j: 0=good, 1=bad
//
// public norm of subpopulation_0 : 0=good, 1=bad
const int PUBLIC_NORM_0_CG=0;
const int PUBLIC_NORM_0_CB=1;
const int PUBLIC_NORM_0_DG=1;
const int PUBLIC_NORM_0_DB=0;
const int PUBLIC_NORM_0_PG=1;
const int PUBLIC_NORM_0_PB=1;

// public norm of subpopulation_1 : 0=good, 1=bad
const int PUBLIC_NORM_1_CG=0;
const int PUBLIC_NORM_1_CB=1;
const int PUBLIC_NORM_1_DG=1;
const int PUBLIC_NORM_1_DB=1;
const int PUBLIC_NORM_1_PG=1;
const int PUBLIC_NORM_1_PB=0;

// major strategy at the beginning in subpopulation_0: 0=cooperation, 1=defection, 2=punishment
const int MAJOR_STRATEGY_0_NUMBER=80;
const int MAJOR_STRATEGY_0_G=0;
const int MAJOR_STRATEGY_0_B=1;

// major strategy at the beginning in subpopulation_1: 0=cooperation, 1=defection, 2=punishment
const int MAJOR_STRATEGY_1_NUMBER=80;
const int MAJOR_STRATEGY_1_G=0;
const int MAJOR_STRATEGY_1_B=2;

// initial fraction of good players
const double INITIAL_GOOD=0.5;

// payoff parameters
const double B=9.0;
const double C=3.0;
const double ALPHA=1.0;
const double BETA=4.0;

// probability of in group imitation
const double IN_GROUP=0.8;

// accumulating errors
const double MU=0.02; // error rate in assigning reputation (accumulating; public in mode 1 and private in modes 2 and 3)

// temporal errors
const double E_ACT=0.00; // error rate in executing intended action
const double E_REP=0.00; // error rate in recalling recipient’s reputation (temporal, not accumulating)

// mutation rates
const double MUTATION_RATE=0.01; // mutation rate in action rules
```

int main(void)
{
    int round, steps, total_steps;
    int g, n, a, j, pair, player_donor, player_recipient, learn_g, learn_n, teach_g, teach_n;
    int stack_length;
    int donor_action;
    int rd_int; // random variable (int)
    int group_size[2]; // group_size[g] = the size of subpopulation g
    int count_major_0, count_major_1;

double **payoff; // payoff[g][n] = in group g, player n's payoff
int ***strategy; // strategy[g][n][j] = in group g, player n's action towards j
int ***public_norm; // public_norm[g][a][j] = in group g, evaluation of action a towards j
int **public_reputation; // public_reputation[g][n] = in group g, player n's public reputation

    // for donor-recipient matching
int *stack;
int *donor; // donor[pair]: the donor in #pair
int *recipient; // recipient[pair]: the recipient in #pair

    FILE *fp;

    // file open
fp= fopen("data.dat","w");
if(NULL==fp) printf("error!n");

    // seed for a random variable
srand((unsigned)time(NULL));

    // ******** initialization of variables (for a whole simulation) ********

    group_size[0]=N/2;
group_size[1]=N/2;
for(g=0; g<2; g++)
{
    group_size[g] = N/2;
    for(n=0; n<N; n++)
    {
        group_size[g][n] = new int[2];
    }
}

    stack = new int[N];
donor = new int[N/2];
recipient = new int[N/2];

    // *********** initialization of variables (for a whole simulation) ***********

}
for(n=0; n<group_size[g]; n++)
{
    for(j=0; j<2; j++)
    {
        // initialization of strategy[g][n][j]: 0=cooperation, 1=defection, 2=punishment
        if((g==0) && (n<MAJOR_STRATEGY_0_NUMBER)) // for major strategy of subpopulation_0
        {
            if(j==0) strategy[g][n][j]=MAJOR_STRATEGY_0_G;
            if(j==1) strategy[g][n][j]=MAJOR_STRATEGY_0_B;
        }
        else if((g==1) && (n<MAJOR_STRATEGY_1_NUMBER)) // for major strategy of subpopulation_1
        {
            if(j==0) strategy[g][n][j]=MAJOR_STRATEGY_1_G;
            if(j==1) strategy[g][n][j]=MAJOR_STRATEGY_1_B;
        }
        else strategy[g][n][j]=rand() %3; // strategy is assigned randomly to the others
    }
}

for(g=0; g<2; g++)
{
    for(a=0; a<3; a++)
    {
        for(j=0; j<2; j++)
        {
            // initialization of public_norm[g][a][j]: 0=good, 1=bad
            if(g==0)
            {
                if(a==0 && j==0) public_norm[g][a][j]=PUBLIC_NORM_0_CG;
                if(a==0 && j==1) public_norm[g][a][j]=PUBLIC_NORM_0_CB;
                if(a==1 && j==0) public_norm[g][a][j]=PUBLIC_NORM_0_DG;
                if(a==1 && j==1) public_norm[g][a][j]=PUBLIC_NORM_0_DB;
                if(a==2 && j==0) public_norm[g][a][j]=PUBLIC_NORM_0_PG;
                if(a==2 && j==1) public_norm[g][a][j]=PUBLIC_NORM_0_PB;
            }
            if(g==1)
            {
                if(a==0 && j==0) public_norm[g][a][j]=PUBLIC_NORM_1_CG;
                if(a==0 && j==1) public_norm[g][a][j]=PUBLIC_NORM_1_CB;
                if(a==1 && j==0) public_norm[g][a][j]=PUBLIC_NORM_1_DG;
                if(a==1 && j==1) public_norm[g][a][j]=PUBLIC_NORM_1_DB;
                if(a==2 && j==0) public_norm[g][a][j]=PUBLIC_NORM_1_PG;
                if(a==2 && j==1) public_norm[g][a][j]=PUBLIC_NORM_1_PB;
            }
        }
    }
}

// ******** initialization of reputation ********
//
for(g=0; g<2; g++)
{
    for(n=0; n<group_size[g]; n++)
    {
        // initialization of public_reputation[g][n]: 0=good, 1=bad
        if((double)rand()/RAND_MAX<=INITIAL_GOOD) public_reputation[g][n]=0;
        else public_reputation[g][n]=1;
    }
}

// calculating total steps required

42
if(UPDATE_MODE==2) total_steps=GENERATION*N;

// ********** start of simulation ************
//
for(steps=0; steps<total_steps; steps++) // start of one step
{
    // displaying subpopulation sizes (and the number of major strategies)
    //
    if(steps % N ==0)
    {
        // counting the number of the initially major strategy in each subpopulation
        count_major_0=0; count_major_1=0;
        for(n=0; n<group_size[0]; n++)
        {
            if(strategy[0][n][0]==MAJOR_STRATEGY_0_G&&strategy[0][n][1]==MAJOR_STRATEGY_0_B) count_major_0++;
        }
        for(n=0; n<group_size[1]; n++)
        {
            if(strategy[1][n][0]==MAJOR_STRATEGY_1_G&&strategy[1][n][1]==MAJOR_STRATEGY_1_B) count_major_1++;
        }
        printf("gen = %4d \ [group 0] %3d (%3d) : \ [group 1] %3d (%3d) n", steps/N, group_size[0], count_major_0, group_size[1], count_major_1);
        fprintf(fp, "%d %d %d %d %d", steps/N, group_size[0], count_major_0, group_size[1], count_major_1);
    }
    // ******** initialization of variables (for each step) ********
    //
    for(g=0; g<2; g++)
    {
        for(n=0; n<group_size[g]; n++) payoff[g][n]=0.0;
    }
    for(g=0; g<2; g++) // start of game interactions in group g
    {
        for(round=0; round<ROUND; round++) // start of one round
        {
            // ******** donor-recipient pair matching ********
            //
            for(n=0; n<group_size[g]; n++) stack[n]=n;
            stack_length=group_size[g];
            for(pair=0; pair<group_size[g]/2; pair++)
            {
                rd_int = rand() % stack_length;
                donor[pair]=stack[rd_int];
                stack[rd_int]=stack[stack_length-1];
                stack_length--;
                rd_int = rand() % stack_length;
                recipient[pair]=stack[rd_int];
                stack[rd_int]=stack[stack_length-1];
            }
        }
    }
}

43
stack_length--;
}

// ******** a game in each pair ********

for(pair=0; pair<group_size[g]/2; pair++) {
    // ******** a giving game and payoff calculation ********
    
    // error in recalling recipient's reputation
    if((double)rand()/RAND_MAX<E_REP) // if error occurs
    {
        donor_action=strategy[g][donor[pair]][1-public_reputation[g][recipient[pair]]];
    }
    else // if error does not occur
    {
        donor_action=strategy[g][donor[pair]][public_reputation[g][recipient[pair]]];
    }
    
    // error in executing intended action
    if((double)rand()/RAND_MAX<E_ACT) // if error occurs
    {
        // one of the other two actions is taken randomly
        if((double)rand()/RAND_MAX<0.5) donor_action = (donor_action + 1) % 3;
        else donor_action = (donor_action + 2) % 3;
    }
    
    switch(donor_action)
    {
    case 0: payoff[g][donor[pair]]-=C; payoff[g][recipient[pair]]+=B; break;
    case 1: break;
    case 2: payoff[g][donor[pair]]-=ALPHA; payoff[g][recipient[pair]]-=BETA; break;
    }
    
    // ******** start of updating reputation********
    
    public_reputation[g][donor[pair]]=public_norm[g][donor_action][public_reputation[g][recipient[pair]]];
    
    // error in assignment
    if((double)rand()/RAND_MAX<MU)
    {
        public_reputation[g][donor[pair]]=1-public_reputation[g][donor[pair]];
    }
}

} // end of game for each pair

} // end of game interactions in group g

// when group_size[g] is odd, we need ROUND/2 more game interactions in group g,
// in order for each player to play on average 10 rounds of the game
if(group_size[g]>1 && group_size[g]%2 == 1) {
    for(round=0; round<ROUND/2; round++) // here, round means one game interaction
    {
        // donor-recipient matching
        player_donor = rand() % group_size[g];
        
        // donor-recipient matching
        donor_action = strategy[g][donor[pair]][public_reputation[g][recipient[pair]]];
        
        // error in recalling recipient's reputation
        if((double)rand()/RAND_MAX<E_REP) // if error occurs
        {
            donor_action=strategy[g][donor[pair]][1-public_reputation[g][recipient[pair]]];
        }
        else // if error does not occur
        {
            donor_action=strategy[g][donor[pair]][public_reputation[g][recipient[pair]]];
        }
        
        // error in executing intended action
        if((double)rand()/RAND_MAX<E_ACT) // if error occurs
        {
            // one of the other two actions is taken randomly
            if((double)rand()/RAND_MAX<0.5) donor_action = (donor_action + 1) % 3;
            else donor_action = (donor_action + 2) % 3;
        }
        
        switch(donor_action)
        {
        case 0: payoff[g][donor[pair]]-=C; payoff[g][recipient[pair]]+=B; break;
        case 1: break;
        case 2: payoff[g][donor[pair]]-=ALPHA; payoff[g][recipient[pair]]-=BETA; break;
        }
        
        // ******** start of updating reputation********
        
        public_reputation[g][donor[pair]]=public_norm[g][donor_action][public_reputation[g][recipient[pair]]];
        
        // error in assignment
        if((double)rand()/RAND_MAX<MU)
        {
            public_reputation[g][donor[pair]]=1-public_reputation[g][donor[pair]];
        }
    }
}
player_recipient = player_donor;
while(player_recipient==player_donor)
{
  player_recipient = rand() % group_size[g];
}

// ******** a game between the donor and the recipient ********
//
// error in recalling recipient's reputation
if((double)rand() / RAND_MAX < E_REP) // if error occurs
{
  donor_action = strategy[g][player_donor][1-public_reputation[g][player_recipient]];  
}
else // if error does not occur
{
  donor_action = strategy[g][player_donor][public_reputation[g][player_recipient]];  
}

// error in executing intended action
if((double)rand() / RAND_MAX < E_ACT) // if error occurs
{
  // one of the other two actions is taken randomly
  if((double)rand() / RAND_MAX < 0.5) donor_action = (donor_action + 1) % 3;
  else donor_action = (donor_action + 2) % 3;
}

switch(donor_action)
{
  case 0: payoff[g][player_donor] -= C; payoff[g][player_recipient] += B; break;
  case 1: break;
  case 2: payoff[g][player_donor] -= ALPHA; payoff[g][player_recipient] -= BETA; break;
}

// ******** start of updating reputation********
//
public_reputation[g][player_donor] = public_norm[g][donor_action][public_reputation[g][player_recipient]];

// error in assignment
if((double)rand() / RAND_MAX < M)  
{
  public_reputation[g][player_donor] = 1-public_norm[g][donor_action][public_reputation[g][player_recipient]];
}

}  // end of one matching

} // end of if

} // end of g loop

// reproduction

// (1) a random player (=learner) is chosen from the population
// (2) with probability IN_GROUP, he samples a random player (=teacher) from his subpopulation
// with the remaining probability, he samples a random player (=teacher) from the whole population
// (3) he compares his payoff with that of a chosen player
// (4) if the chosen player has the higher payoff, he imitates the action rule and the group of the chosen player
// (5) in case he moves to the other subpopulation, his reputation there is initialized randomly

// (1) choosing who is a learner
n = rand() % N;
if(n<group_size[0])
{
    learn_g=0;
    learn_n=n;
}
else
{
    learn_g=1;
    learn_n=n-group_size[0];
}
// in group learn_g, player learn_n is a learner

// (2) sampling a teacher
if(((double)rand()/RAND_MAX<IN_GROUP)
{
    teach_g = learn_g;
    teach_n = rand() % group_size[learn_g];
}
else
{
    n = rand() % N;
    if(n<group_size[0])
    {
        teach_g=0;
        teach_n=n;
    }
    else
    {
        teach_g=1;
        teach_n=n-group_size[0];
    }
}
// in group teach_g, player teach_n is teacher

// (3) payoff comparison
if(payoff[teach_g][teach_n]>payoff[learn_g][learn_n])
{

    // (4) mimicking strategy and group
    if(teach_g == learn_g) // if a learner mimics a teacher in his subpopulation
    {
        for(j=0; j<2; j++) strategy[learn_g][learn_n][j] = strategy[teach_g][teach_n][j];
        // mutation
        if(((double)rand()/RAND_MAX<MUTATION_RATE)
        {
            for(j=0; j<2; j++) strategy[learn_g][learn_n][j]=rand() % 3;
        }
    }
    else // if a learner mimics a teacher in the other subpopulation
    {
        // in teacher's group, a new player is created
        for(j=0; j<2; j++) strategy[teach_g][group_size[teach_g]][j] = strategy[teach_g][teach_n][j];
        // mutation
        if(((double)rand()/RAND_MAX<MUTATION_RATE)
        {
            for(j=0; j<2; j++) strategy[teach_g][group_size[teach_g]][j]=rand() % 3;
        }
        if(((double)rand()/RAND_MAX<INITIAL_GOOD) public_reputation[teach_g][group_size[teach_g]] = 0;
        else public_reputation[teach_g][group_size[teach_g]] = 1;
        group_size[teach_g]++;
    }
    // in learner's group, the learner is replaced with player group_size[learn_g]-1

}
if (group_size[learn_g] >1)
{
    if (learn_n != group_size[learn_g]-1)
    {
        for (j=0; j<2; j++)
        {
            strategy[learn_g][learn_n][j] = strategy[learn_g][group_size[learn_g]-1][j];
            public_reputation[learn_g][learn_n] = public_reputation[learn_g][group_size[learn_g]-1];
        }
    }
    else
    {
        for (j=0; j<2; j++)
        {
            strategy[learn_g][learn_n][j] = strategy[learn_g][group_size[learn_g]-2][j];
            public_reputation[learn_g][learn_n] = public_reputation[learn_g][group_size[learn_g]-2];
        }
    }
    group_size[learn_g]--;
}

} // end of one step

// closing the file
fclose(fp);

// deleting dynamic memory

for (g=0; g<2; g++)
{
    delete[] payoff[g];
    delete[] payoff;
    for (g=0; g<2; g++)
    {
        for (n=0; n<N; n++)
        {
            delete[] strategy[g][n];
        }
        delete[] strategy[g];
    }
    delete[] strategy;
    for (g=0; g<2; g++)
    {
        for (a=0; a<3; a++)
        {
            delete[] public_norm[g][a];
        }
        delete[] public_norm[g];
    }
    delete[] public_norm;
    for (g=0; g<2; g++)
    {
        delete[] public_reputation[g];
        delete[] public_reputation;
        delete[] stack;
        delete[] donor;
        delete[] recipient;
    }
}