Fostering Collective Action in Complex Societies using Community-Based Agents
(Supplementary Material)

Jonathan Skaggs*, **Michael Richards***, **Melissa Morris**,
Michael A. Goodrich and **Jacob W. Crandall**
Computer Science Department, Brigham Young University, Provo, UT, USA
{jbskaggs12, michael.richards256, mel4college}@gmail.com, {mike, crandall}@cs.byu.edu

Overview of the Supplementary Material

This document is part of the Supplementary Material (SM) for the following paper:

J. Skaggs, M. Richards, M. Morris, M. A. Goodrich, and J. W. Crandall
Fostering Collective Action in Complex Societies using Community-Based Agents.
Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-24), Jeju, South Korea, 2024

The Supplmenetary Material for this paper consists of the following elements:

1. This document
   Contains additional details about (1) the Junior High Game (JHG; Section 1), CAB agents (Section 2), the evolutionary simulations described in the paper (Section 3), Conspiring autonomous thieves (CATs; Section 4), and additional results (Sections 5 and 6).
2. Data: https://github.com/jakecrandall/IJCAI2024_SM.git
   Contains raw data logs for each of the 18 games of the user study reported in the paper. Visualizations summarizing these games are provided in Section 6 of this document.
3. Simulation code: https://github.com/jakecrandall/IJCAI2024_SM.git
   Contains C++ and Python implementations of the JHG Engine, CAB and CAT agents, and C++ code to run evolutionary simulations to train the CAB agents. Java code is also supplied to allow for a human to play the game with CAB and CAT agents through a simple GUI.
4. The JHG online platform (effective by August 2024): juniorhighgame.com
   Online platform used to play the JHG games between human participants and CAB agents in the user studies reported in the paper. This online platform can be used by the public to play JHG games between human players. Configurable CAB agents will also be available, so that users can add them into their games.

1 The Junior High Game: More General Description

The main manuscript contains a correct yet simplified description of the JHG, which was limited due to page constraints. A more general description of the game is provided in this section.

In the main manuscript, popularity (the explicit form of capital used in the JHG) is assumed to be material wealth (i.e., fortune). However, other forms of capital, such as fame, behave differently. Thus, in Section 1.1, we provide a more general specification of popularity update dynamics that allows for such forms of capital. Second, in Section 1.2, we add the concept of a poverty line, which prohibits players from perpetually benefiting from attacking players that have no (or little) popularity to steal. Both of these features were used in the user study and simulations reported in the main manuscript, with $\beta = 0.5$ (see Section 1.1) and $L = 0$ (see Section 1.2).

In Section 1.3, we discuss how parameters and features of the JHG can be varied to model many different scenarios.

1.1 Capital in the JHG: Fame vs. Fortune

The JHG models a society with a single commodity of exchange (i.e., capital), which we refer to as *popularity*. Eqs. (1-4) in the main paper model this commodity as material capital. Under this model, once a token is allocated (given, kept, or used to attack), the value of the allocation does not change over time, except that its value fades with time (newer allocations have

*Equal contribution

higher impact on popularity than older exchanges). Thus, we could say that Eqs. (1-4) model an exchange of *fortune* or material goods. However, many commodities have the property that the value of exchanges could fluctuate over time due to reflective processes. As an example, suppose individuals i and j have a childhood friendship (which consists of exchanges of tokens), but then stop interacting once they reach adulthood. If j becomes famous later in adulthood, this could also reflect positively on the fame of i, even if the two no longer interact. That is, the exchanges in early childhood between individuals i and j may initially seem insignificant to outsiders. However, once j becomes famous the interactions may seem more significant. The exchange of reputation-based commodities such as *fame* behave differently than that of *fortune*.

As an example, we may consider the year $\tau = 1939$ in Germany (during the height of Hitler's power). Suppose that an individual received a positive endorsement from Hitler in 1939. When viewed from the year $t = 1939$, this exchange perhaps seemed to give individual i great prestige (among certain classes of the population). However, when viewed from the year $t = 2000$, this same exchange would appear to have little value. Thus, we provide a set of more general equations of popularity update designed to capture such notions.

These equations are as follows:

$$\mathcal{P}_i(\tau, t) = \max\left(0, (1-\alpha)^{(\tau-1)}\mathcal{P}_i^{\text{init}} + \sum_{j \in I} \mathcal{I}_{i,j}(\tau - 1, t - 1)\right) \tag{1}$$

$$\mathcal{I}_{i,j}(\tau, t) = \begin{cases} \text{if } \tau \geq 1, & \alpha\mathcal{V}_{i,j}(\tau, t) + (1-\alpha)\mathcal{I}_{i,j}(\tau - 1, t) \\ \text{otherwise,} & 0 \end{cases} \tag{2}$$

$$\mathcal{V}_{i,j}(\tau, t) = \begin{cases} \text{if } i = j, & \mathcal{W}_i(\tau, t)(c^{\text{keep}} x^+_{i,i}(\tau) + \sum_{k \in I} c^{\text{steal}}_k(\tau, t) x^-_{k,i}(\tau)) \\ \text{otherwise,} & \mathcal{W}_j(\tau, t)(c^{\text{give}} x^+_{i,j}(\tau) - c^{\text{steal}}_i(\tau, t) x^-_{i,j}(\tau)) \end{cases} \tag{3}$$

$$\mathcal{W}_j(\tau, t) = \beta\mathcal{P}_j(\tau, \tau) + (1-\beta)\eta(\tau, t)\mathcal{P}_j(t, t) \tag{4}$$

$$\eta(\tau, t) = \frac{\sum_k \mathcal{P}_k(\tau, \tau)}{\sum_k \mathcal{P}_k(t, t)} \tag{5}$$

$$c^{\text{steal}}_k(\tau, t) = \begin{cases} \text{if } \exists j : x^-_{j,k}(\tau) \neq 0, & c^{\text{steal}} \max\left(0, 1 - \frac{x^+_{k,k}(\tau) W_k(\tau, t)}{\sum_{j \in I} x^-_{j,k}(\tau) W_j(\tau, t)}\right) \\ \text{otherwise,} & c^{\text{steal}} \end{cases} \tag{6}$$

In these equations, the popularity of player i is defined in Eq. (1) as $\mathcal{P}_i(\tau, t)$, or the popularity of player i at the beginning or round τ when viewed from some time $t \geq \tau$. Note that $\mathcal{P}_i(\tau, t_1)$ need not be equal to $\mathcal{P}_i(\tau, t_2)$ for $t_1 \neq t_2$ as the perspective on a player's popularity at some point in time τ can change with perspective.

The popularity update defined by SM-Eqs. (1-6) differs from the update defined in Eqs. (1-4) in that exchanges are weighted by $\mathcal{W}_j(\tau, t)$ instead of just player popularity. $\mathcal{W}_j(\tau, t)$, defined in SM-Eq. 4, is a convex combination, of player j's popularity at the beginning of round τ (i.e., $\mathcal{P}_j(\tau, \tau)$) and player j's popularity at the time the exchange is being viewed (i.e., $\mathcal{P}_j(t, t)$) multiplied by $\eta(\tau, t)$ (SM-Eq. 5) which corrects for inflation between rounds τ and t. The parameter $\beta \in [0, 1]$ determines whether the commodity behaves as fame or fortune. When $\beta = 1$, the weight of token allocations depends only on the popularity of player j at the time it allocated the token (and makes SM-Eqs. 1-6 identical to Eqs. 1-4), and hence the commodity acts as an exchange of material goods (i.e., *fortune*). On the other hand, when $\beta = 0$, the weight of token allocations in the past depends only on the current popularity of player j. This represents the exchange of *fame*. In such conditions, if player i receives a token from player j and then player j subsequently becomes more popular, the value of that token exchange increases. In this way, the popularities of players j and i become intertwined. Intermediate values of β form scenarios in which the commodity exchanged represents a combination of fame and fortune.

1.2 Advanced Detail: The Poverty Line

A final specification to the popularity-update functions is needed to ensure that players cannot benefit from stealing from a player that has no wealth. Typically, we might say that player i has no wealth if $\mathcal{P}_i(\tau, t) = 0$. However, we may want to specify a more general *poverty line*, which we denote as $L \geq 0$, which allows that poor players with positive popularity may still have nothing valuable to steal. A player has no stealable popularity if $\mathcal{P}_i(\tau, t) \leq L$. To ensure that players do not benefit from stealing from players that do not have any wealth to take, we specify an adjustment to the computation of $\mathcal{I}_{i,j}(\tau, t)$.

First, as in SM-Eq. (2), for all i and j, we make a preliminary computation of the influence of j on i:

$$\mathcal{I}'_{i,j}(\tau, t) = \begin{cases} \text{if } \tau \geq 1, & \alpha\mathcal{V}_{i,j}(\tau, t) + (1-\alpha)\mathcal{I}_{i,j}(\tau - 1, t) \\ \text{else,} & 0 \end{cases} \tag{7}$$

We then compute how much these preliminary computations of influence put player i below the poverty line:

$$B_i(\tau,t) = \max\left(0, L - \left[(1-\alpha)^{(\tau-1)}\mathcal{P}_i^{\text{init}} + \sum_{j\in I}\mathcal{I}'_{i,j}(\tau-1,t-1)\right]\right) \tag{8}$$

We next determine the amount of popularity that was taken from player i in the last round:

$$T_i(\tau,t) = \alpha\sum_{k\in I}\mathcal{V}^-_{i,k}(\tau,t), \tag{9}$$

where $\mathcal{V}^-_{i,k}(\tau,t) = -\min(\mathcal{V}_{i,k}(\tau,t),0)$. Thus, the proportion of damage inflicted on player i in round τ, viewed from round $t \geq \tau$, that is valid is:

$$D_i(\tau,t) = \begin{cases}\text{if } T_i(\tau,t) > 0, & \frac{B_i(\tau,t)}{T_i(\tau,t)}\\ \text{else,} & 1\end{cases} \tag{10}$$

Now, we can recompute influence as follows:

$$\mathcal{I}_{i,j}(\tau,t) = \begin{cases}\text{if } i=j, & \mathcal{I}'_{i,j}(\tau,t) - \alpha\sum_{k\in I} D_k(\tau,t)\mathcal{V}^-_{k,i}(\tau,t)\\ \text{else,} & \mathcal{I}'_{i,j}(\tau,t) + \alpha D_i(\tau,t)\mathcal{V}^-_{i,k}(\tau,t)\end{cases} \tag{11}$$

The top part of Eq. (11) subtracts out the attempted steals by player i on players that did not have anything to steal (because their popularity is below the poverty line). The bottom part restores invalid stolen popularity to player i (because its popularity was below the poverty line).

1.3 Scenario Modeling: Parameter Flexibility

The JHG provides for an interesting case of agent interactions in mixed-motive scenarios subject to reputation, power, and network dynamics, as well as hierarchical organizations. The game contains a variety of parameters that allow it to be tuned to specific scenarios. In this subsection, we briefly describe these parameters and their impacts on JHG game scenarios.

The JHG has the following kinds of parameters that allow flexibility in modeling different kinds of scenarios.

- *Augmenter coefficients*: As mentioned previously, the coefficients c^{keep}, c^{give}, and c^{steal} define weights for keeping, giving, and attacking. In the game's vanilla form, these weights are the same for all players and all interactions. In such cases, all players can interact with all other players with equal impact (subject to the popularity dynamics defined). However, more generally, coefficients can be made to be interaction (and time) specific. For example, $c^{\text{give}}_{i,j}$ (the *give* coefficient for player j giving tokens to player i) need not be the same as $c^{\text{give}}_{i,k}$ (for some $j \neq k$). This can be useful to model, for example, geographic constraints that make it potentially ineffectual (or less unproductive) for players to interact with players that are not geographically adjacent to themselves.

  Additionally, the popularity-update equations assume that the amount of profit and damage from stealing is equal (SM-Eq. 3). However, this constraint could easily be lifted by making the coefficients in the top and bottom cases of SM-Eq. (3) distinct from each other.
- *Popularity-update rate*: The parameter α dictates how quickly popularity changes, which allows for scenarios in which popularity changes slowly or quickly.
- *Fame vs. Fortune*: As mentioned previously, the parameter β dictates whether the commodity is modeled as *fame* $\beta = 1$, *fortune* $\beta = 0$, or something in between $\beta \in (0,1)$.
- *Initial popularity*: To start the games, each player i is assigned an initial popularity $\mathcal{P}_i^{\text{init}}$, or ascribed status/standing [1]. These initial popularities can all be the same (i.e., $\mathcal{P}_i^{\text{init}} = \mathcal{P}_j^{\text{init}}$ for all i and j), or they can be different for each player.
- *Granularity of token allocations*: In theory, players could be allowed to allocate any fraction of their tokens to any player. For simplicity in this initial investigation, we assume that each player has a fixed number of tokens (e.g., $2|I|$) to allocate in each round.

In addition to parameters that impact popularity updates, the game can also be varied in terms of the information that is provided to players. For example, the observability of popularities, token allocations, and influences among all players can be varied. In this paper, we assume that agents can observe $\mathcal{P}_j(\tau,t)$ for all $j \in I$ (for all (τ,t)). Additionally, we assume that the agents can observe the all influences $\mathcal{I}_{j,k}(\tau,t)$. However, each player only has knowledge of the token allocations that they make and the tokens given to and taken from them. That said, alternate forms of information can be provided to players and the information available need not be the same for all players.

The game can also be played with various forms of communication available to the players. Such communication abilities can and do greatly alter game dynamics. For games played using the online platform available at juniorhighgame.com, users can configure games to allow for several different forms of text-based chat.

Algorithm 1 CAB token allocation in round τ for player i.

1: **procedure** ALLOCATETOKENS($\mathcal{G}(\tau)$, Θ)
2: $\quad\mathbf{c}(\tau) \leftarrow$ detectCommunities$(\mathcal{G}(\tau), \Theta)$
3: $\quad c_i'(\tau) \leftarrow$ identifyDesiredCommunity$(\mathbf{c}(\tau), \mathcal{G}(\tau), \Theta)$
4: $\quad\hat{\kappa}_i \leftarrow$ determineDefense$(c_i'(\tau), \mathcal{G}(\tau), \Theta)$
5: $\quad\{(a_i, j), (\kappa_i, i)\} \leftarrow$ addAttack$(c_i'(\tau), \mathbf{c}(\tau), \mathcal{G}(\tau), \hat{\kappa}_i, \Theta)$
6: $\quad$**return** addGives$(c_i'(\tau), \mathcal{G}(\tau), \{(a_i, j), (\kappa_i, i)\}, \Theta)$
7: **end procedure**

2 CAB: Community-Based Agent Behavior

The JHG requires players to choose among economic actions (using tokens to give, keep, and attack) in mixed-motive scenarios impacted by community, power, and reputation dynamics. Successful players in the JHG must determine when and how to use these economic actions to cooperate and collude with each other. At the center of such dynamics is the ability to identify and choose which community or communities the player would like to belong to, and to then take actions to form and strengthen that community (as well as establish one's place in it).

Thus, in this paper, we propose an algorithm for playing the JHG that is centered around community detection and formation. We call this algorithm CAB (*C*ommunity-*A*ware *B*ehavior). CAB is summarized in Algorithm 1. In each round, CAB first computes which communities the players belong to based on observations of economic actions from prior rounds. Based on these communities and its own preferences, the agent then determines the community it would like to belong to. Finally, the agent allocates its tokens to form the desired community, to make its community successful, and to establish its position within it.

Details related to how the agent selects its desired community and chooses economic actions to help form it and strengthen it is determined by an extensive set of parameters Θ. These parameters, which are summarized in Table 1, can be varied to produce a wide range of distinct behaviors. All parameters take on integer values in the range $[0, 100]$.

In this section, we first describe the agent[1] We then describe how parameter values can be learned via evolutionary simulation to form a potentially successful agent for playing the JHG. Finally, we describe variations for parameterizing and training this agent.

2.1 Algorithm Description

The way the CAB agent allocates tokens in each round τ is overviewed in Algorithm 1. The agent takes as input its set of parameters (Θ) and the observations that the agent has observed up to time τ. In this paper, we assume this set is given by the set $\mathcal{G}_i(\tau)$, which contains the influence matrices $\mathcal{I}(t)$ for all $t \in [1, \tau)$, the popularity functions $\mathcal{P}(t)$ for all $t \in [1, \tau]$, and token allocations to and from agent i (that is, $x_{i,j}(t)$ and $x_{j,i}(t)$, for all $j \in |I|$ and $t \in [1, \tau)$). The algorithm then runs the five distinct functions specified in lines 2-6 of the algorithm to determine how to allocate agent i's tokens in the round. Note that this summary of the algorithm is more verbose than the algorithm specified in the main manuscript, as it breaks `computeAllocations` into three separate functions: `determineDefense`, `addAttack`, and `addGives`.

We describe each function in this section.

Detect Communities

In line 2 of Algorithm 1, the agent calls the function `detectCommunities` to detect the communities (alliances) that have formed in prior rounds of the JHG. This community detection is based on the influence matrix $\mathcal{I}(\tau - 1)$ (see the description of the JHG in the prior section), which is derived from all transactions that have occurred up to round τ. While a variety of different community-detection and clustering algorithms are available, each with different strengths and weaknesses, CAB detects communities using one pass (including both phase 1 and 2) of the Louvain Method [2]. Recall that phase 1 of the Louvain Method initially allocates the players into groups, while the second phase determines how (and if) pairs of groups would like to join together. The function returns the community allocation vector $\mathbf{c}(\tau) = \big(c_1(\tau), \cdots, c_{|I|}(\tau)\big)$, where $c_j(\tau)$ denotes the community of player j at the beginning of round τ. It also returns the initial community allocation vector $\hat{\mathbf{c}}(\tau)$ that was created as a result of the first phase of the Louvain Method.

The Louvain Method assigns agents into communities based on the metric of modularity, which, loosely speaking, measures the extent to which groups have separated from each other. However, computing modularity requires a non-directed graph in which all edges in the graph have non-negative weights. The influence matrix $\mathcal{I}(\tau)$ can violate both of these assumptions, as $\mathcal{I}_{i,j}$ need not equal $\mathcal{I}_{j,i}$, and it is possible for $\mathcal{I}_{i,j}$ to be negative. Thus, we use a modified computation of modularity.

[1] Python and C++ (code) implementations of the CAB algorithm are available for download at https://github.com/jakecrandall/IJCAI2024_SM.git.

Table 1: The set of parameters Θ used in the CAB algorithm. Each parameter can take on an integer value in the range $[0, 100]$. Adjusting the parameters can produce vastly different behavior. *Handcoded values* refer to the parameter values used for the *Handcoded* settings describes in Section 5 of the main paper.

Symbol	Usage	Handcoded value
θ_{α}	Specifies the weight between positive and negative influence when computing MGM (SM-Eq. 14).	20
θ_{Strength}	Specifies the desired collective strength of the agent's community as a percentage of the whole.	70
θ_{modu}	Weight of modularity in scoring potential communities	100
θ_{target}	Weight of target collective strength in scoring potential communities	80
$\theta_{\text{prominent}}$	Weight of prominence in scoring potential communities	50
θ_{familiar}	Weight of familiarity in scoring potential communities	50
$\theta_{\text{prosocial}}$	Weight of prosocial in scoring potential communities	70
θ_{initialD}	Specifies the percentage of its tokens the agent keeps in round $\tau = 1$	20
θ_{DUpdate}	Specifies how quickly the agent adapts its keeping based on new attacks on itself	50
$\theta_{\text{DPropensity}}$	Specifies degree the degree to which the agent keeps tokens based on past attacks against it	50
θ_{fearD}	Specifies degree the degree to which the agent keeps tokens based on attacks on its friends	0
θ_{minD}	Specifies the minimum number of tokens the agent keeps in a round	20
θ_{pFury}	Percentage of tokens to use in a pillage attack	0
θ_{pDelay}	Number of rounds to wait at the start of the game before considering pillage	10
$\theta_{\text{pPriority}}$	The priority of pillage attacks	0
θ_{pMargin}	Determines the gain margin required before a particular pillage attack is considered	0
$\theta_{\text{pCompanion}}$	The amount of help the agent initially believes it will get in a pillage attack over the first 5 rounds	50
θ_{pFriends}	Determines whether an agent will considering pillaging a friend.	0
θ_{vMult}	Specifies how much vengeance to reciprocate	100
θ_{vMax}	Specifies the maximum number of tokens that can be used in a vengeance attack	100
$\theta_{\text{vPriority}}$	The priority of vengeance attacks	100
θ_{dfMult}	Determines how much to reciprocate in defend-friend attacks	100
θ_{dfMax}	Specifies the maximum number of tokens that can be used in a defend-friend attack	100
$\theta_{\text{dfPriority}}$	The priority of defend-friend attacks	90
$\theta_{\text{attackGGuys}}$	Determines whether the agent will consider defend-friend attacks against players that were not the first to attack and did not over-attack	0
$\theta_{\text{groupAware}}$	Determines whether the agent will consider defend-friend attacks against players that belong to communities that have more collective strength than its own.	0
θ_{Safety}	Specifies with the agent will defend or attack if it does not have enough tokens to do both	0
$\theta_{\text{debtLimits}}$	Defines how much the agent will give to another player without reciprocation	25
$\theta_{\text{limitGive}}$	Sets limits how much the agent can give to a poor player	100
$\theta_{\text{fixedUsage}}$	Determines the percentage of tokens the agent gives uniformly to its group (as opposed to distributing based on past friendship).	50

Let $\mathcal{I}^{+}_{k,j}(\tau) = \max(0, \mathcal{I}_{k,j}(\tau))$ and $\mathcal{I}^{-}_{k,j}(\tau) = |\min(0, \mathcal{I}_{k,j}(\tau))|$. Then, let

$$\hat{\mathcal{I}}^{+}_{k,j}(\tau) = \frac{\left(\max(\mathcal{I}^{+}_{k,j}(\tau), \mathcal{I}^{+}_{j,k}(\tau)) + \left(\mathcal{I}^{+}_{k,j}(\tau) + \mathcal{I}^{+}_{j,k}(\tau)\right)/2\right)}{2} \tag{12}$$

and

$$\hat{\mathcal{I}}^{-}_{k,j}(\tau) = \max\left(\mathcal{I}^{-}_{k,j}(\tau), \mathcal{I}^{-}_{j,k}(\tau)\right). \tag{13}$$

respectively. In words, $\hat{\mathcal{I}}^{+}_{k,j}(\tau)$ is the average of the max and average of $\mathcal{I}^{+}_{k,j}(\tau)$ and $\mathcal{I}^{+}_{j,k}(\tau)$, while $\hat{\mathcal{I}}^{-}_{k,j}(\tau)$ is the max of $\mathcal{I}^{-}_{k,j}(\tau)$ and $\mathcal{I}^{-}_{j,k}(\tau)$. We chose to encode positive and negative edges differently since a single negative directed edge typically causes agents to not want to be in the same community, whereas one-sided positive connections may not make both players want to be in the same community.

After computing these quantities, both matrices are undirected, weighted graphs. As such, the Louvain Method can compute modularity for any proposed community allocation $\mathbf{c}$ based on positive and negative influence. The modularity of some community allocation $\mathbf{c}$, which we call multi-graph modularity (MGM), is computed as follows:

$$\text{MGM}(\mathbf{c}, \mathcal{I}(\tau)) = \frac{\theta_\alpha}{100}\text{modularity}(\mathbf{c}, \hat{\mathcal{I}}^+(\tau)) - (1 - \frac{\theta_\alpha}{100})\text{modularity}(\mathbf{c}, \hat{\mathcal{I}}^-(\tau)). \tag{14}$$

Here, $\theta_\alpha \in [0, 100]$ is a weighting factor contained in CAB's parameters Θ. Intuitively, positive connections within a community increase MGM and negative connections within a community decrease MGM. On the other hand, positive connections among members of different communities decrease MGM, while negative connections among members of different communities increase MGM.

Determine Desired Community

While `detectCommunities` specifies a community for agent i (and all other agents), it may not be the community the agent wants to belong to. The agent may want to switch communities, add or remove players from its community, or join communities together in order to strengthen its position in the society. Thus, after identifying communities among the players of the game using the Louvain Method, the agent next computes the community it would like to be part of (function `identifyDesiredCommunity`). This function returns the set $C_i'(\tau) \subseteq I$ (line 3, Algorithm 1).

Let $\mathcal{C}$ be a set of possible communities that player i would like to belong to. Then to determine $C_i'(\tau)$, agent i scores (giving a value in the range $[0, 1]$) each possible set $\bar{C} \in \mathcal{C}$ based on the following five attributes:

1. *Modularity*: This attribute favors communities that are modular. To score communities with respect to this attribute, we compute a community allocation $\bar{\mathbf{c}}$ given the set $\bar{C}$. $\bar{\mathbf{c}}$ is formed by first removing all members $k \in \bar{C}$ from their allocation in $\mathbf{c}(\tau)$ into a new (separate) community. The remaining players are re-allocated to communities based on a single pass (phases 1 and 2) of the Louvain Method. The modularity score for the set $\bar{C}$ is then given by:

$$C_{\text{Modularity}} = \frac{\text{MGM}(\bar{c}, \hat{\mathcal{I}}(\tau))}{\max_{B \in \mathcal{C}} \text{MGM}(\bar{b}, \hat{\mathcal{I}}(\tau))}, \tag{15}$$

   where $\bar{b}$ is formed from B as $\bar{c}$ was formed from $\bar{C}$.

2. *Target Group Strength*: This attribute favors communities whose relative collective power (sum of the popularities of the members of the group relative to the sum of popularities of the entire population) are closest to the players ideal (target) relative power. To score communities with respect to this attribute, we determine how close the collective strength of the members of C (i.e., $\sum_{k \in C} \mathcal{P}_k(\tau)$) are relative to player i's desired collective strength, which is specified by the parameter θ_{Strength} in the parameter set Θ. C_{Target} is computed by the function `getCollectiveStrength` in the supplied C++ code (in GeneAgent.h) and `get_collective_strength` in the supplied Python code (in geneagent3.py). We refer readers to the code to view the nuances of this computation.

3. *Prominence*: This attribute favors communities in which player i has high popularity compared to other members of the community. We compute prominence based on an equal weighting of three factors: (1) the proportion of individuals in the community with lower popularity than player i, (2) the ratio of player i's popularity to the most popular individual in the community, and (3) ratio of player i's popularity relative to the average popularity of the players in the community. Note that popularity, as defined in the JHG, is a similar notion as Katz centrality, but with a weighted, signed, directed and dynamic graph. Specific details of how $C_{\text{Prominent}}$ is computed is given in the function `getCentrality` in the supplied C++ code and `get_centrality` in the supplied Python code.

4. *Familiarity*: This attribute favors communities in which player i is more connected to individuals in the community than individuals outside of the community. While this is related to modularity (the first attribute), this attribute allows the agent to put emphasis on its own connections rather than the connections of individuals at large. $C_{\text{Familiarity}}$ is computed by the function `getFamiliarity` in the supplied C++ code and `get_familiarity` in the supplied Python code.

5. *Prosocial Behavior*: This attribute favors communities in which the members of the community are being "good citizens." If an individual (1) receives more than they are giving, (2) keeps a lot of tokens, (3) is not reciprocating with player i, (4) is attacking other people in the community, or (5) is attacking players outside the community without cause, then player i tends to not want that in individual in its community (and thus community allocations that place that individual in the desired community are rated lower). Details about how $C_{\text{Prosocial}}$ is computed are given in the function `getIngroupAntisocial` in the supplied C++ code and `get_ingroup_antisocial` in the supplied Pytho3 code.

The overall score of potential community C (denoted C_{Score}) is computed using weights supplied as part of the agent's set of parameters Θ. These weights are θ_{modu}, θ_{target}, $\theta_{\text{prominence}}$, θ_{familiar}, and $\theta_{\text{prosocial}}$. This score is computed by the `computeScore` function in the supplied C++ code and `compute_score` in the supplied Python code.

Because the number of possible communities is very large, the agent considers only a subset of possible communities in each round. Let $C_i(\tau)$ be the community agent i belongs to as specified by the community allocation $\mathbf{c}(\tau)$ (i.e., $C_i(\tau) = \{k \in I :$

$c_k(\tau) = c_i(\tau)\}$). Furthermore, let $\hat{C}_i(\tau)$ be agent i's community based on the first phase of the Louvain method (derived from $\hat{\mathbf{c}}(\tau)$). Then, $\mathcal{C}$ consists of the following sets: (1) Communities that result from adding a new member to either $C_i(\tau)$ or $\hat{C}_i(\tau)$; (2) Communities that result from subtracting a single member from $C_i(\tau)$ or $\hat{C}_i(\tau)$ (not including i); (3) Communities formed by agent i moving to a new community in the community allocations $\mathbf{c}(\tau)$ and $\hat{\mathbf{c}}(\tau)$; (4) Communities formed by joining other communities in $\mathbf{c}(\tau)$ ($\hat{\mathbf{c}}(\tau)$) with $C_i(\tau)$ ($\hat{C}_i(\tau)$) .

Then, we have that:

$$C_i'(\tau) = \max_{C \in \mathcal{C}} C_{\text{Score}} \tag{16}$$

Determine number of tokens to keep

Once a CAB agent has determined which community it would like to be part of, it then determines how to use economic actions (keeping, attacking, and giving tokens) to form and build the community $C_i'(\tau)$, as well as to establish its own position within that community. The first economic action the agent considers is to determine how many tokens it should keep to defend itself from attacks, which is initially determined by the function `determineDefense` (line 4, Algorithm 1). This function return $\hat{\kappa}_i$, which specifies the proposed number of tokens to keep in the round. The subsequent two functions, `addAttack` and `addGives`, adjust this quantity based on the availability of tokens.

In the first round ($\tau = 1$), $\hat{\kappa}_i$ is determined by θ_{initialD}, which specifies the percentage of its tokens the agent should keep in the first round. In subsequent rounds, the agent keeps tokens based on two different factors: (1) attacks on itself and (2) attacks on its friends. In each round, the agent observes how much itself and its friends are attacked to predict how much it could be attacked in the next round. It then keeps enough tokens to block the expected attacks. Predictions of subsequent attacks are based on four different parameters: θ_{DUpdate} (which determines how quickly the agent adjusts its prediction based on the results of the last round), $\theta_{\text{DPropensity}}$ (which determines how cautious the agent is when predicting subsequent attacks based on attacks to itself), θ_{fearD} (which determines how cautious the agent is when predicting subsequent attacks back on attacks to its friends), and θ_{minD} (which specifies the minimum percentage of tokens the agent should keep).

Full details for how the agent chooses to keep tokens based on these parameters and game events are found in the function `cuantoGuardo` in the supplied C++ code and `cuanto_guardo` in the supplied Python code.

Determine who to attack

The agent next considers using its tokens to attack (steal popularity from) another player. While a player could potentially attack multiple players in a round, CAB limits the number of attacks it makes in a round to a single attack. The function `addAttack` computes who to attack (if any) and how many tokens it should attack with. It also adjusts/finalizes the number of tokens the agent will keep in defense κ_i.

To determine which attack to make, a CAB agent (player i) considers three different forms of attacks: (1) vengeance attacks: attacks designed to take vengeance on players that have inflicted damages on player i; (2) defend-friend attacks: attacks on individuals that have attacked player i's friends. (3) pillage attacks: unprovoked attacks on other players. Vengeance attacks are parameterized by three parameters: $\theta_{rmvMult}$, θ_{vMax}, and $\theta_{\text{vPriority}}$. Defend-friend attacks are parameterized by five parameters: θ_{dfMult}, θ_{dfMax}, $\theta_{\text{dfPriority}}$, $\theta_{\text{attackGGuys}}$, and $\theta_{\text{groupAware}}$. Finally, pillage attacks are based off of six parameters: θ_{pFury}, θ_{pDelay}, $\theta_{\text{pPriority}}$, θ_{pMargin}, $\theta_{\text{pCompanion}}$, and θ_{pFriends}.

The CAB agent first chooses no more than one attack from each of the three forms of attacks. In making this determination, the CAB agent predicts, based on observations from prior rounds, the number of tokens the potential target players are keeping as well as how much other players may attack the target player at the same time. Based on these estimates, the CAB agent predicts whether each attack would be successful in (1) providing profit to the CAB agent (player i) and (2) inflicting damage upon the target player. If the CAB agent does not believe it would benefit based on these estimates from an attack, it discards the idea of carrying out the attack on that player.

After selecting the best attack (if any) from each form of attack, the CAB agent uses the priority parameters $\theta_{\text{vPriority}}$, $\theta_{\text{pPriority}}$, and $\theta_{\text{dfPriority}}$ to determine which attack to execute (assuming the agent found at least one attack it computes would be successful).

Full details for how the agent chooses who (if any) to attack and how many tokens to use in that attack are defined in the function `quienAtaco` in the supplied C++ code and `quien_ataco` in the supplied Python code.

Attacks are only carried out if there are sufficient tokens available. The parameter θ_{Safety} determines whether the agent prioritizes keeping or attacking in the case that the agent does not have enough tokens to both keep $\hat{\kappa}_i$ tokens and make a particular attack.

Give tokens to members of chosen community

After determining how many tokens to keep and how many tokens to use in attack, CAB uses the remainder of its tokens to give to individuals in its chosen community. Two principles determine how the agent distributes its tokens within its group. First, the agent determines how much of its tokens to distribute uniformly among the players in the chosen community or to distribute tokens based on the level of friendship. This is determined by the parameter $\theta_{\text{fixedUsage}}$. Second, in order to not be taken advantage of, the agent limits giving to players that have not reciprocated or cannot (due to low popularity) adequately

reciprocate. The way that such limitations are set is defined by the parameters $\theta_{\text{debtLimits}}$ and $\theta_{\text{limitGive}}$. If the agent cannot find individuals to give all of its remaining tokens to based on the limitations, it keeps those tokens.

Full details for how the agent chooses to give its tokens to players in its target group are defined in the function `groupAllocateTokens` in the supplied C++ code and `group_allocate_tokens` in the supplied Python code.

2.2 Learning Parameter Values

We used a typical genetic algorithm to learn parameter values used by CAB agents in the experiments reported in the main paper. This algorithm is defined as follows:

- A population of 100 CAB agents were evolved for 200 generations. 200 games, each played by 10 randomly selected agents from the agent pool, were played in each generation. Each game lasted 30 rounds.
- In the initial population, all parameters (genes) were randomly selected for each agent (except for the case described in Section 5 of the main paper in which they were all initialized to the handcoded settings, which are given in Table 1 of this document). At the end of each generation, a new population was created using the selection and mutation process described next.
- Selection and Mutation: At the start of subsequent generations, each agent in the new population was created by randomly selecting two agents (parents) from the previous generation, with probability of selection proportional to each agent's fitness based on games it played in the previous generation (see next bullet point). With probability 0.85, a gene for the new agent is selected from one (chosen randomly) of its parents randomly selected from the gene of one of the two parents. With probability 0.12, the gene value was the random selection between the two parents plus a value in the range $[-5, 5]$ (the gene is then capped in the range $[0, 100]$. With remaining probability (0.03), the gene was generated randomly in the range $[0, 100]$.
- Fitness: There are multiple possible goals that an agent could have when playing the JHG. In this paper, we assume the agents want to either maximize their own popularity over time (absolute popularity) and maximize its rank among agents in the game (relative popularity). This creates two different notions of fitness. The fitness of an agent A with respect to absolute popularity in a game lasting T rounds is defined as follows:

$$F(A) = 0.5 \cdot \left(\frac{1}{N} \sum_{t=1}^{N} \mathcal{P}_A(t) + \mathcal{P}_A(T) \right) \tag{17}$$

  That is, fitness with respect to absolute popularity is the average of the mean popularity of the agent in a game (across all rounds) and its ending popularity. This places emphasis on agents to have high popularity at the end of the game, but also considers an agent's popularity over all rounds of the game. The fitness of an agent A with respect to relatively popularity is simply its popularity rank (among players in the game). Specifically, an agent's relative fitness is the number of agents that have lower popularity than it does at the end of the game. We chose to select 90% of parents based on absolute fitness, and the other 10% on relative fitness, but enforced that no new agent had both parents selected using relatively fitness.

3 Initial Popularity Distributions

Here, we enumerate the initial popularity distributions for agents used in games played in the evolutionary simulations. The average initial popularity (i.e., the *base popularity*) of all players was always 100. The following five initial popularity distributions were used in the games, chosen at random to begin each game.

- **Equal:** Every player is assigned the same popularity value, which is the base popularity.
- **Random:** Each player is assigned a random (real-valued) initial popularity in the range [1,200]. The popularities are then normalized so that the mean values is the base popularity.
- **High-Low:** The first half of the players are assigned a random real-valued initial popularity in the range [151, 200] (the rich class). The second half of the players are assigned a random real-valued initial popularity in the range [1, 50] (the poor class). These values are then shuffled and the mean is normalized to the base popularity.
- **Step:** Players are assigned increasing popularity values, starting from 1.0 up to the number of players. These values are then shuffled and the mean is normalized to the base popularity.
- **Power:** Players are assigned values based on a power function. Specifically, each entry is set to the reciprocal of the (player index + 1) raised to the power of 0.7. These values are then shuffled and the mean is normalized to the base popularity.

Algorithm 2 CAT token allocation in round τ for player i.

1: **procedure** ALLOCATETOKENS($\mathcal{G}(\tau)$)
2: **if** $\tau = 1$ **then**
3: $s \leftarrow \{(2|I|, i)\}$ `// player keeps all tokens`
4: **else**
5: $\mathbf{c}(\tau) \leftarrow$ detectOtherThieves($\mathcal{G}(\tau)$)
6: $s \leftarrow \{\}$
7: $k_i \leftarrow$ determineDefense($\mathcal{G}(\tau)$) `// determine how many tokens to keep`
8: $s \leftarrow \{(k_i, i)\}$
9: $T_{\text{Left}} = 2|I| - k_i$
10: $\omega_j \leftarrow$ identifyWeakling($\mathcal{G}(\tau), \mathbf{c}(\tau), s$) `// identify player to attack`
11: **if** $\exists \omega_j$ **then**
12: $a_j \leftarrow$ determineAttack($\mathcal{G}(\tau), \mathbf{c}(\tau), s$) `// determine tokens to be use in the attack`
13: $s \leftarrow s \cup \{(a_j, \omega_j)\}$
14: $T_{\text{Left}} = T_{\text{Left}} - a_j$
15: **end if**
16: $s \leftarrow s \cup \{(T_{\text{Left}}, i)\}$
17: **end if**
18: **return** s
19: **end procedure**

4 Conspiring Autonomous Thieves (CATs)

The CAT algorithm described in Section 5 of the main paper is implemented in the code in the file `assassinAgent.h` of the supplied C++ code and `assassinAgent.py` in the supplied Python code.

An overview of the algorithm is provided in Algorithm 2. In the first round, the CAT agent keeps all of its tokens (line 3). In subsequent rounds, it first detects which agents are acting like thieves (line 5). It then determines how many tokens it should keep to block all attacks against it, which are is estimated using how much it was attacked in prior rounds (lines 7-8). It then identifies the weakest individual that is not a CAT (line 10) and has sufficient popularity. If such an individual exists it computes an attack on that individual (lines 12-13). It keeps all remaining tokens (line 16).

Section 5 of the main manuscript describes CAB agents that used handcoded parameter values. These handcoded parameter values are specified in Table 1. They are also provided in the function `assassinDefense` in the file `GeneAgent.h` of the supplied C++ code. Note: To make CAT agents use these parameter values in the C++ code, uncomment line 555 in `GeneAGent.h` and comment out line 556.

5 Additional Results

5.1 Effectiveness of the Genetic Algorithm in the JHG Environment

We have observed that JHG societies seem to perpetually evolve. This somewhat unique characteristic for simulation test-beds introduces challenges when applying traditional optimization techniques. With our utilization of a genetic algorithm to optimize agent strategies, we remained curious about its performance in such a volatile setting.

As mentioned in Section3, we executed the genetic algorithm over 200 iterations, aiming to optimize CAB behaviors (by learning parameter values) in the JHG. Given the fluid nature of societal dynamics in the JHG, a salient question was whether the genetic algorithm would reach convergence. Alternatively, there was a possibility that the algorithm might display cyclical patterns in its optimization process, never truly settling on a singular optimal strategy due to the ever-changing goalpost represented by the societal dynamics.

To investigate this, we display the average popularity of CAB agents at the end of each game an various generations. These popularities are shown in Figure 1. For each of the six independent simulations of the genetic algorithm, we plotted the mean ending popularity based on 50 games along with the standard error. This graphical representation offers insights into the algorithm's trajectory, revealing that it did not converge. The top 10 performing parameter settings in the last generation from each of the six independent simulations were used in the user study.

This exploration is interesting and showcases the complex challenges of creating strategies in the JHG.

5.2 Identifying Bots verses Humans in the User Study

After each game of the human-bot study reported in Section 4 of the main paper, human participants were asked to evaluate the identity of each of the other players in the game. They scored their evaluations on a 7-point Likert scale, with 1 being "definitely human" and 7 being "definitely bot." Over all conditions of the study, the mean evaluation for bot players was 4.20, while the mean evaluation for human players was 4.06. Essentially, human players could not distinguish human players from CAB agents. Figure 2 gives a visual representation of this result.

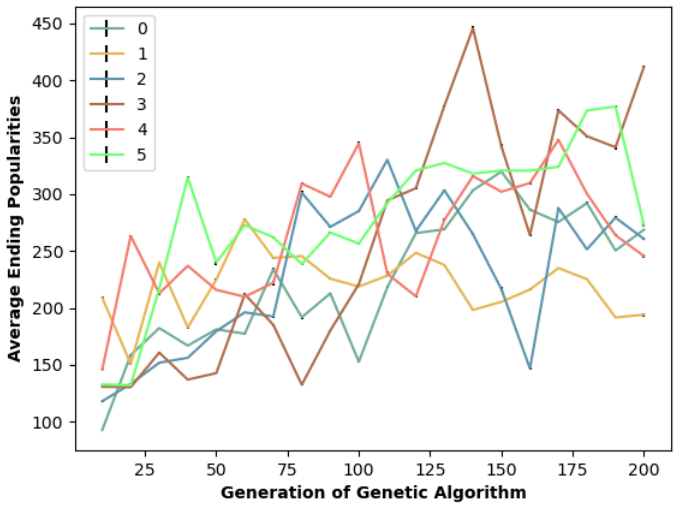


Figure 1: Evolutionary progress of the CAB agents in the Junior High Game (JHG). Specifically, the plot shows the mean ending popularity achieved in different generations, aggregated every five generations. Six separate iterations of the genetic algorithm are shown. This showcases the agent's adaptation to the intricate network, reputation, and power dynamics inherent in the game. Notably, after the initial 100 generations, the algorithm exhibits cyclical behavior, indicating challenges in achieving convergence, and underscoring the complexity of the game's social dynamics.

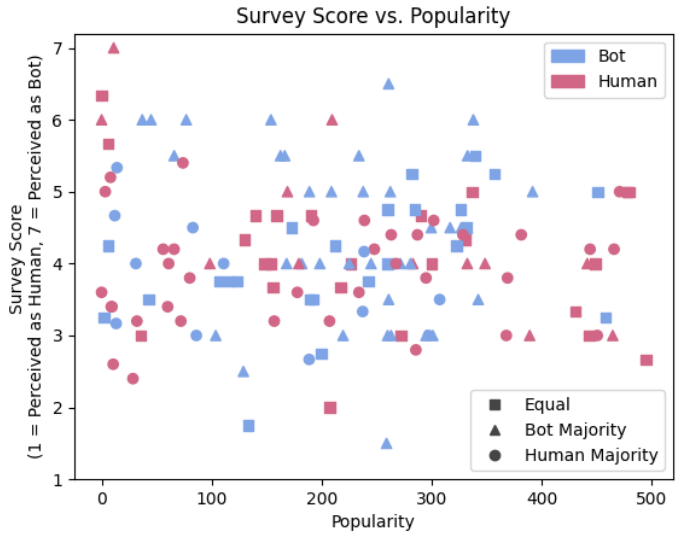


Figure 2: Evaluations by human participants regarding the identify of other players in the human-bot experiment.

6 Visualization and Data of Individual Games of the User Study

This section contains visualizations for all 18 games conducted in the user study. Six games were played in each condition. Visualizations show (above) Player popularities ($\mathcal{P}_i(\tau)$) in each round and (below) the game network in selected rounds. In the game network graphs, nodes (representing players) are positioned by influence, such that players i and j tend to be in close proximity when $\mathcal{I}_{i,j}(\tau)$ and $\mathcal{I}_{j,i}(\tau)$ are high. Larger nodes represent higher relative popularity. Directed edges denote token allocations in the round (i.e., $x_{i,j}(\tau)$), which are green if positive and orange if negative.

Raw data files for the games are available at https://github.com/jakecrandall/IJCAI2024_SM.git

Condition: Human Majority (6 Humans and 2 CAB agents)

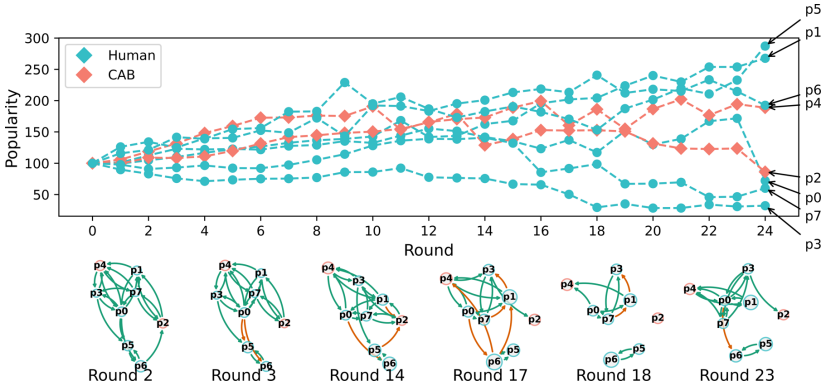


Figure 3: Game code: GCVN

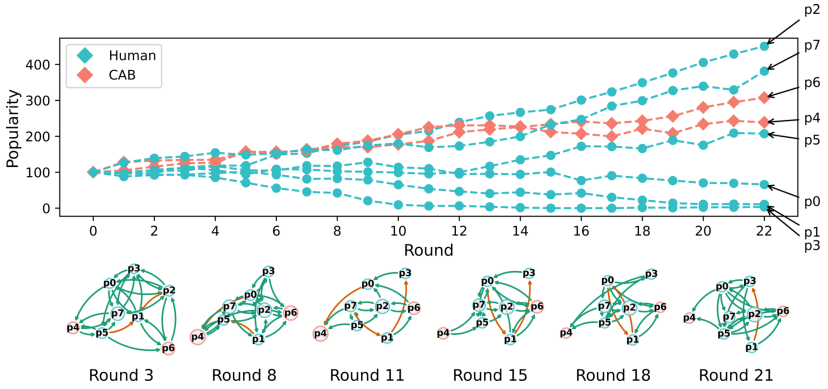


Figure 4: Game code: TDGM

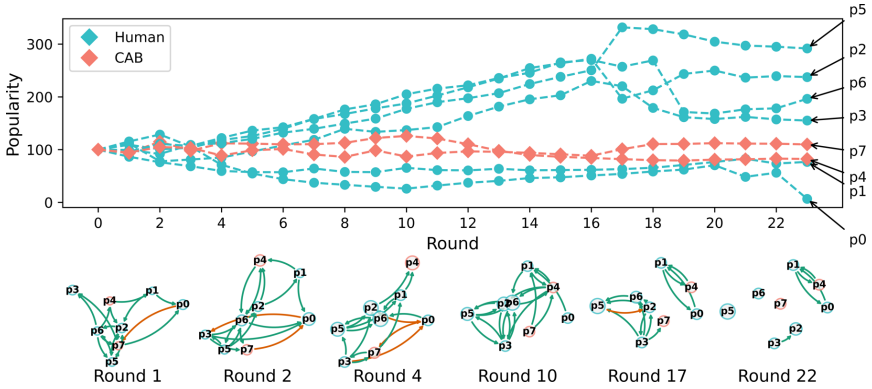


Figure 5: Game code: WRBV

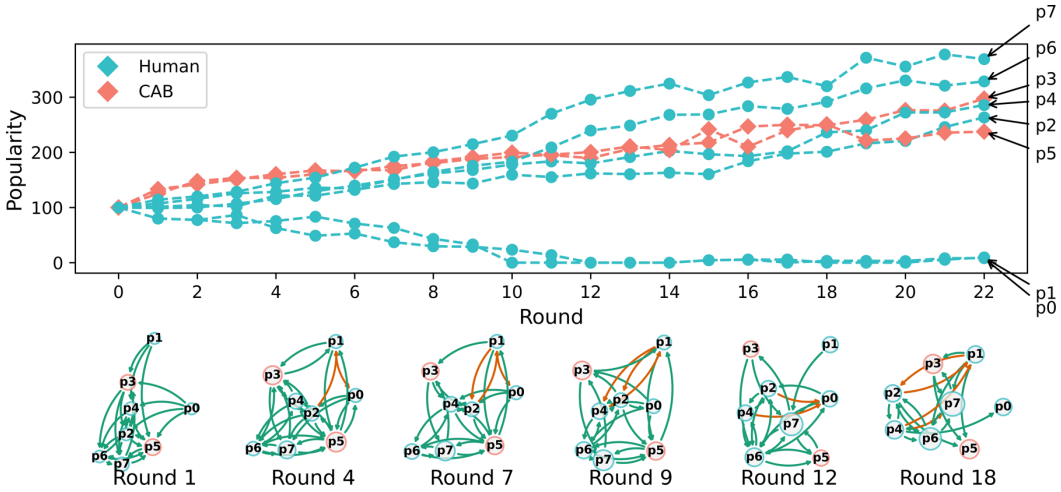


Figure 6: Game code: GJFD

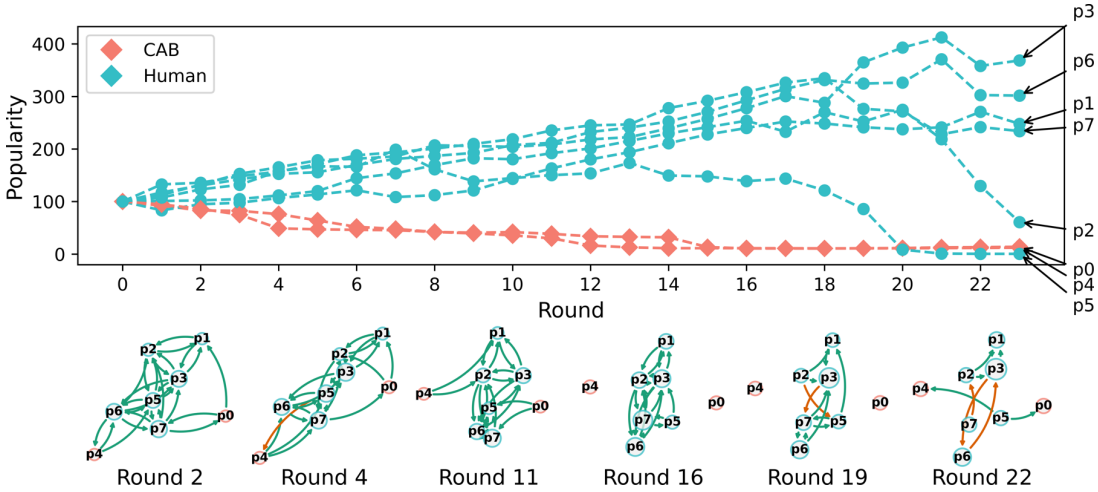


Figure 7: Game code: WQJR

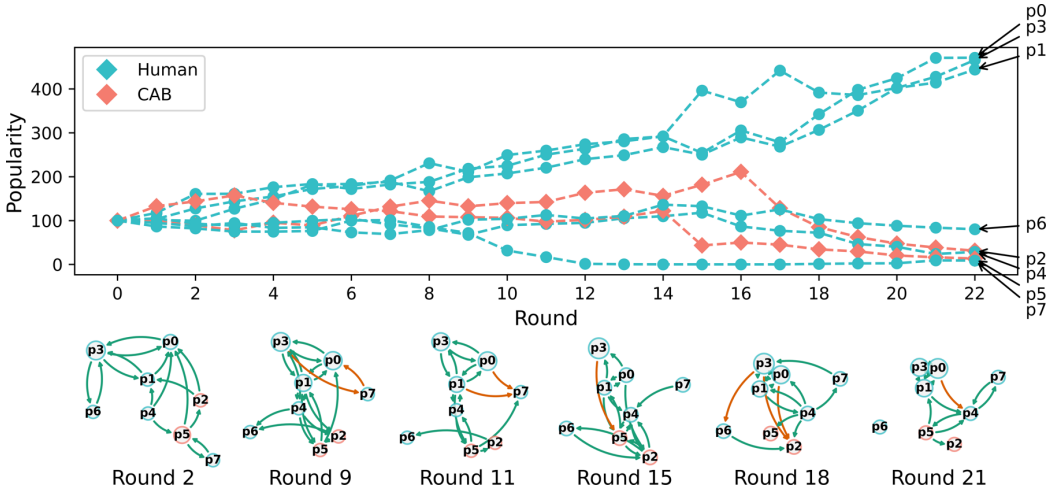


Figure 8: Game code: XTWS

Condition: Bot Majority (2 Humans and 6 CAB agents)

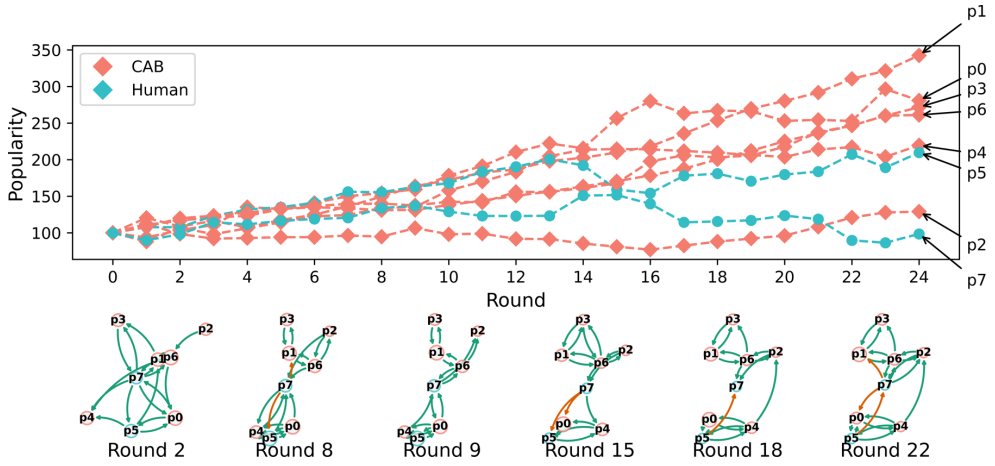


Figure 9: Game code: GXVS

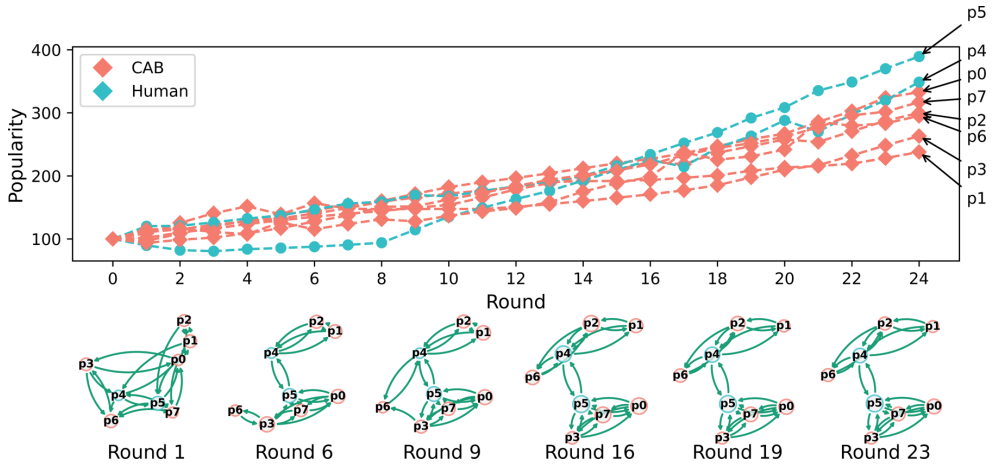


Figure 10: Game code: SHFC

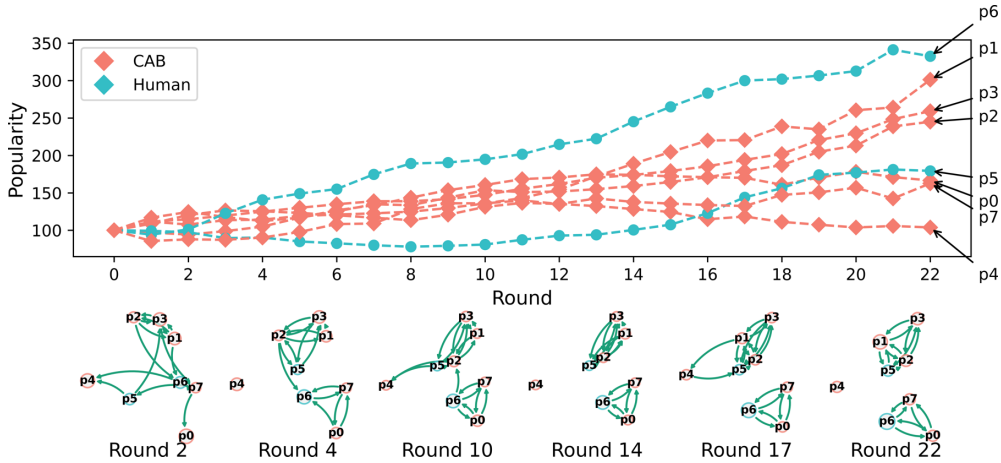


Figure 11: Game code: WCJQ

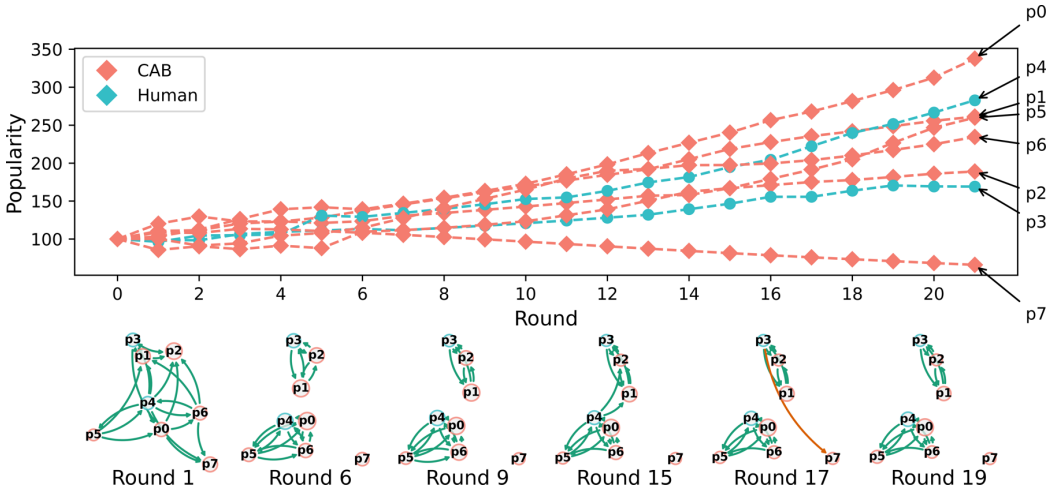


Figure 12: Game code: HNCQ

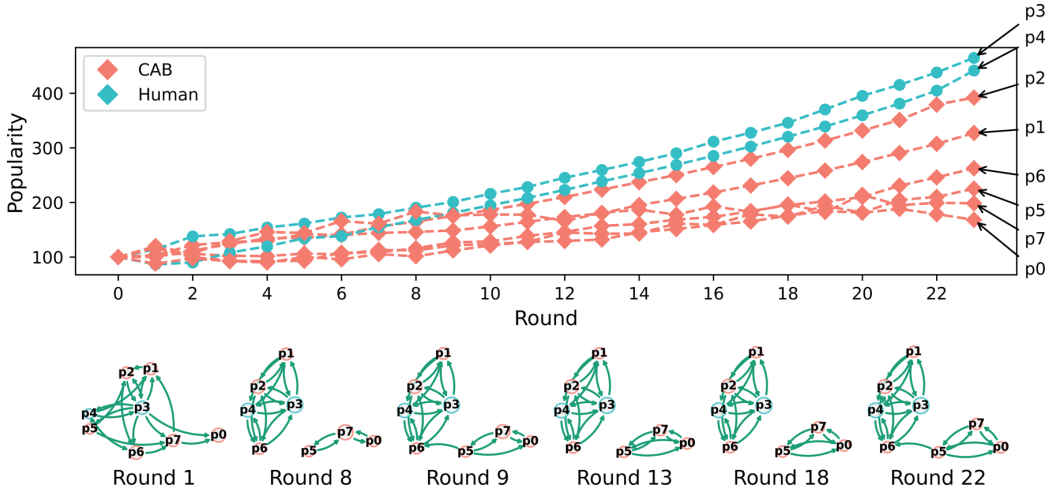


Figure 13: Game code: TKRW

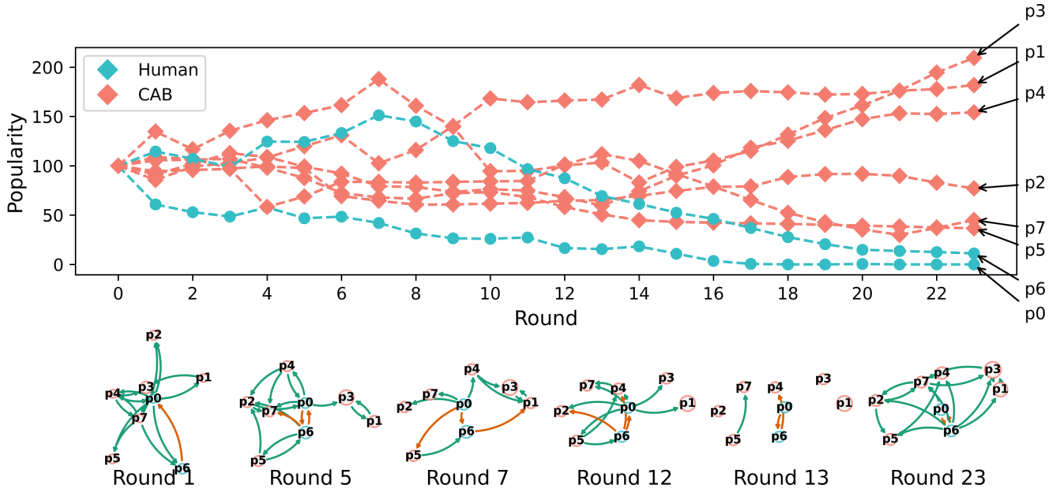


Figure 14: Game code: WHRP

Condition: Even (4 Humans and 4 CAB agents)

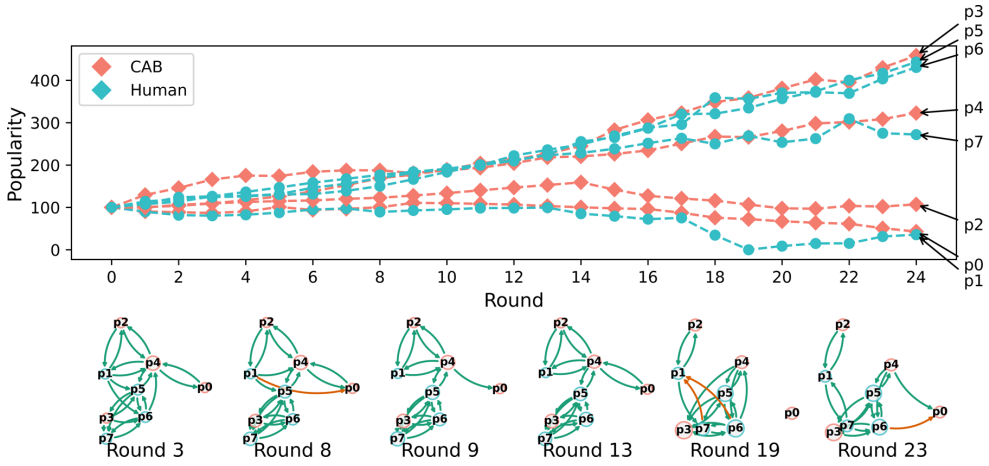


Figure 15: Game code: CXJR

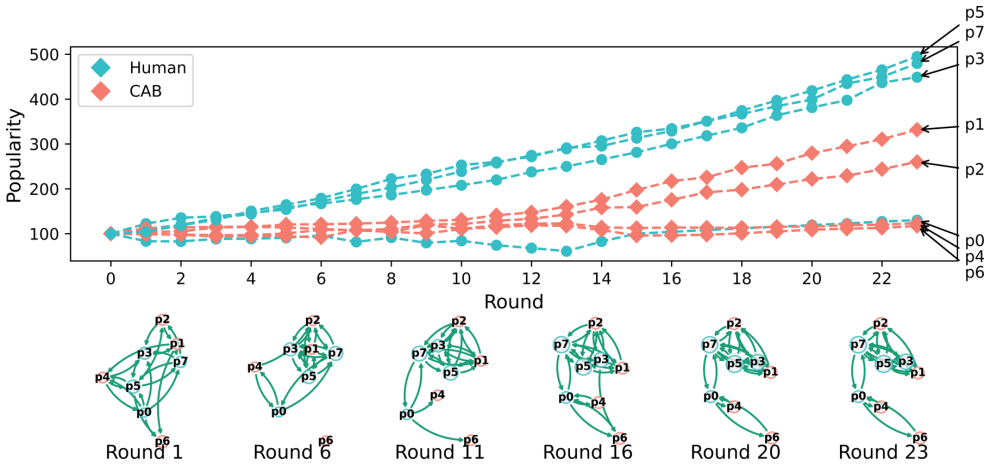


Figure 16: Game code: PSGN

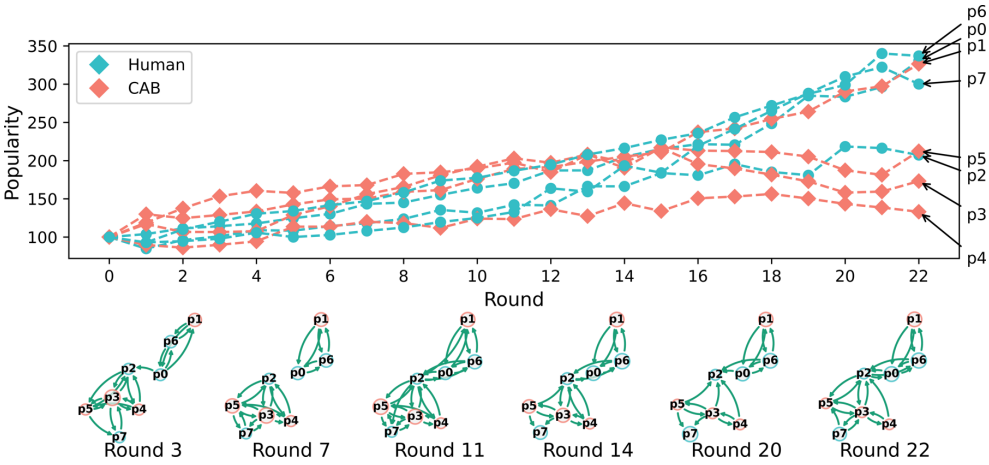


Figure 17: Game code: WQDS

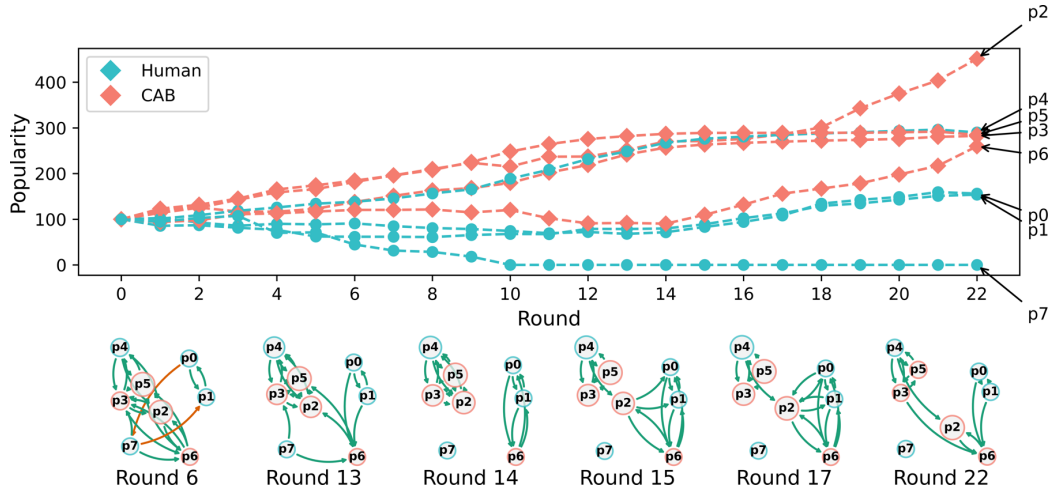
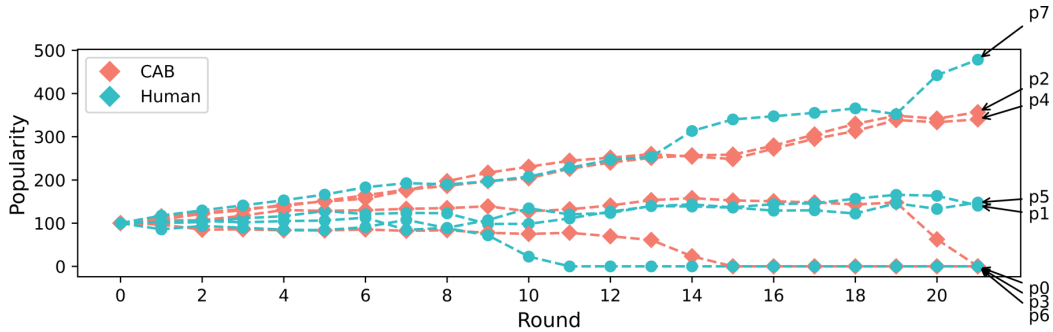


Figure 18: Game code: MCZG



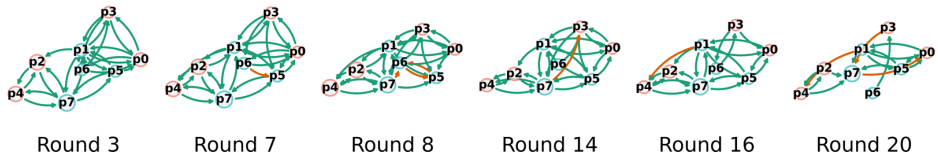


Figure 19: Game code: VBLN

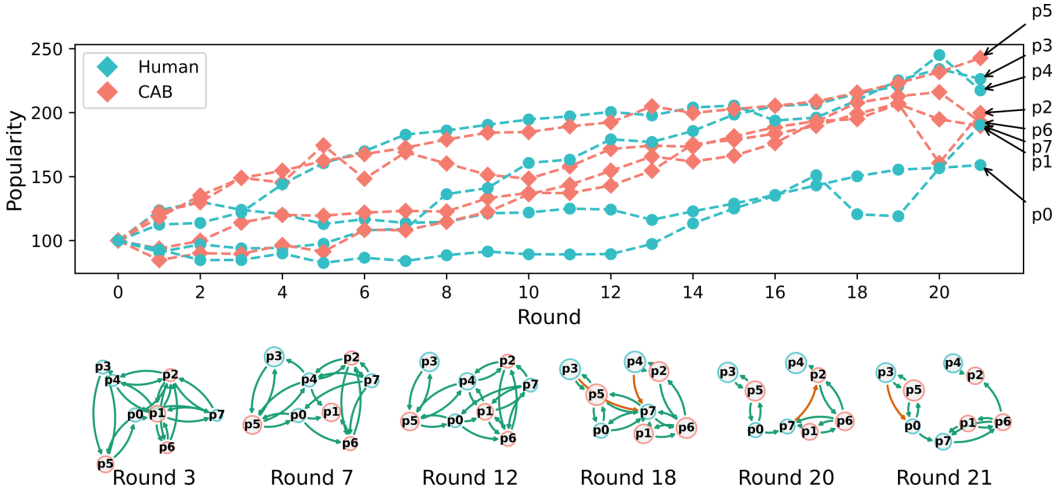


Figure 20: Game code: XSVC

References

[1] *Sociology: Understanding and Changing the Social World*. University of Minnesota Libraries Publishing, Minneapolis, MN, 2016.

[2] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienn Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 10, 2008.