

```

/*
 * Automaton.h    header file
 *
 * Author:    Evan Hourigan                                Copyright (c), 2004.
 *            Hobart & William Smith Colleges
 *            Geneva, NY    14456
 *
 *            Email: hourigan@hws.edu
 * This source code may not be reproduced or modified in any way, shape, or
 * form without written consent from the author.
 */

#include <iostream>
#include <string>
#include <ctime>
using namespace std;

const int OPPONENT_MOVES = 2;
const int NUMBEROFSTATES = 20;
const int COOPERATE = 0;
const int DEFECT = 1;
const int NUMBER_OF_GAMES = 100;
const int MAX_INTERACTIONS = 10;
const int HEAD = 0; // the head of the data array is always zero

// These global variables are being used to track how many times each of the
// four different possible results of each game (i.e. COOPERATE-COOPERATE,
// DEFECT-COOPERATE, COOPERATE-DEFECT, and DEFECT-DEFECT) occur, as well as
// niceness
extern int cc_count;
extern int cd_count;
extern int dc_count;
extern int dd_count;

class Automaton { // The "Chromosome" class for the Population template.

    int state; // The state you are in
    int nextState[NUMBEROFSTATES][OPPONENT_MOVES]; // Determines the next state
                                                    // based upon the move of the
                                                    // opponent while in the
                                                    // current state.
    int move[NUMBEROFSTATES]; // Gives the move associated with each state.
                                // Value is either COOPERATE, or DEFECT.
    bool marked[NUMBEROFSTATES]; // Keep track of states already printed
    double fitness; // Starting at zero, fitness is computed by playing
                    // individuals against each other.

public:

    // Used in multi-process programming. Packs all of the current
    // individual's genetic data into a single struct, which can be type cast
    // to char* and sent to different processes.
    struct Pack {
        int nextState[NUMBEROFSTATES][OPPONENT_MOVES];
        int move[NUMBEROFSTATES];
    };

    // Constructor for an automaton. The nextState arrays are filled randomly
    // with values from 0 to NUMBEROFSTATES-1 for both possible determining
    // moves (either cooperate (0) or defect(1)). The move array is filled
    // with values of either 0 or 1. These arrays are parallel.
    Automaton();

    // Resets the array marking the already-visited states
    void resetMarkers();

```

```

// Instead of calculating fitness individually, fitness is determined by
// by making a population of individuals play the prisoner's dilemma game.
// Each individual plays every other individual, and the sum of the
// game scores for each individual is the fitness value. Hence, every
// time one individual plays another individual, the results of the game
// are added to each individual's fitness with increaseFitness().
void setFitness(double fitnessValue);
void increaseFitness(double fitnessValue);
double getFitness();

void mutate();

// Makes an individual into a defector by changing all the values in the
// move array into DEFECT
void malign();

void crossover (Automaton& partner);

// Determines if an individual is nice (i.e. that it will not be the
// first to defect in a match with another individual) by examining its
// data. Starting at currentState = startingState (i.e. zero), if
// the move of the currentState is DEFECT, then the individual is not
// nice, and no further analysis is needed. If the move of the
// currentState is COOPERATE, and the state has not already been visited,
// then go on to the nextState specified for the opponent's cooperation
// in the current state. However, if the move of the currentState is
// COOPERATE, and the state has already been visited, then the individual
// is in a state loop and therefore must be nice.
bool isNice();

static void computeFitnesses(Automaton **data, int size);

// Traverses and prints out the state graph recursively. If state is
// equal to 0, the start state, and reset is equal to true, then
// resetMarkers() is called to reset all of the markers. Starting at
// state state, if state is not marked, then print a list of information
// in the following format:
//      currentState:action  moveToOnCooperate  moveToOnDefect
// where currentState is the current state, action is the action
// associated with that state (either cooperate or defect),
// moveToOnCooperate is the state to move to if the opponent chooses to
// cooperate, and moveToOnDefect is the state to move to if the opponent
// chooses to defect.
void printStates(int state, bool reset=false);

// Tests the current individual in a variety of ways and returns an int
// value in the range of 0 (dumb) to 2*NUMBEROFSTATES (smart),
// representing the IQ of the current individual. Initially the IQ is
// set at NUMBEROFSTATES, to give each individual a fair chance to prove
// themselves. Then, performs a variety of game-behavior tests, and
// based upon the current individuals behavior, increases or decreases
// the IQ score. If the current individual is not nice (and returns
// FALSE when its isNice() function is called), then this test does not
// apply, and returns a value of -10 indicating a non-nice individual
int nicenessIQ();

// Used in parallel implementation
// Replace genetic data of this worst individual object with the data
// passed as newIndividual
void transplant(Pack newIndividual);

// Used in parallel implementation
// Packs current individuals brains into a struct, and return it as a
// value of type Automaton::Pack

```

```
    Pack get();  
}; /* end of class Automaton */
```