

```

/*
 * Automaton.cc      implementation 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>
#include "Automaton.h"
using namespace std;

int cc_count;
int cd_count;
int dc_count;
int dd_count;

inline int randInt(int N) { // Random int x where 0 <= x < N
    return std::rand() % N;
}
inline double randReal() { // Random real x where 0.0 <= x < 1.0
    return ( std::rand() / (RAND_MAX + 1.0));
}

Automaton::Automaton() {
    for (int i = 0; i < NUMBEROFSTATES; i++) {
        nextState[i][0] = randInt(NUMBEROFSTATES);
        nextState[i][1] = randInt(NUMBEROFSTATES);
        move[i] = randInt(2);
        marked[i] = false;
    }
} /* end of Automaton::Automaton() constructor */

void Automaton::resetMarkers() {
    for (int i = 0; i < NUMBEROFSTATES; i++)
        marked[i] = false;
} /* end of void Automaton::resetMarkers() */

void Automaton::setFitness(double fitnessValue) {
    fitness = fitnessValue;
} /* end of void Automaton::setFitness(double) */

void Automaton::increaseFitness(double fitnessValue) {
    fitness += fitnessValue;
} /* end of void Automaton::increaseFitness(double) */

double Automaton::getFitness() {
    return fitness;
} /* end of double Automaton::getFitness() function */

void Automaton::mutate() {
    int pos = randInt(NUMBEROFSTATES);
    switch (randInt(3)) {
        case 0: nextState[pos][0] = randInt(NUMBEROFSTATES); break;
        case 1: nextState[pos][1] = randInt(NUMBEROFSTATES); break;
        case 2: move[pos] = !move[pos];
    }
}

```

```

} /* end of void Automaton::mutate() */

void Automaton::malign() {
    for (int i = 0; i < NUMBEROFSTATES; i++)
        move[i] = DEFECT;
}

void Automaton::crossover (Automaton& partner) {
    int pos = randInt(NUMBEROFSTATES-2) + 1;
    for (int i = pos; i < NUMBEROFSTATES; i++) {
        // Copy current automaton's data into a temporary location
        int tempNextStateZero = nextState[i][0];
        int tempNextStateOne = nextState[i][1];
        int tempMove = move[i];
        // Replace current automaton's data with partner's data
        nextState[i][0] = partner.nextState[i][0];
        nextState[i][1] = partner.nextState[i][1];
        move[i] = partner.move[i];
        // Replace partner's data with data stored in temporary location
        partner.nextState[i][0] = tempNextStateZero;
        partner.nextState[i][1] = tempNextStateOne;
        partner.move[i] = tempMove;
    }
} /* end of void Automaton::crossover(Automaton&) */

bool Automaton::isNice() {
    bool visited[NUMBEROFSTATES]; // States visited
    bool nice = true; // Innocent until proven guilty
    int currentState = 0; // Start at beginning
    for (int i = 0; i < NUMBEROFSTATES; i++) // Start off having not yet
        visited[i] = false; // visited all of the states
    while(nice) {
        if (move[currentState] == COOPERATE) {
            if (!visited[currentState]) {
                visited[currentState] = true; // Mark the current state
                currentState = nextState[currentState][COOPERATE];
                // At this point, niceness has not been determined and so
                // the nextState array will need to continue to be followed
            }
            else // If the currentState was already visited, then the
                break; // individual must therefore be nice.
        }
        else // Individual is first to defect and is therefore not nice.
            nice = false;
    }
    return nice; // Niceness has been determined
} /* end of bool Automaton::isNice() function */

void Automaton::computeFitnesses(Automaton **data, int size) {
    // Reset all of the individual fitnesses to zero
    for (int ct = 0; ct < size; ct++)
        data[ct]->fitness = 0;
    // Pair each individual once with each other player
    // in the population.
    for (int a = 0; a < size; a++) {
        for (int b = a+1; b < size; b++) {
            data[a]->state = 0; // Start state is state zero
            data[b]->state = 0; // Start state is state zero
            for (int i = 0; i < NUMBER_OF_GAMES; i++) {
                // For each game: The move of each player is given by
                // the action associated with its current state
                int moveA = data[a]->move[data[a]->state];
                int moveB = data[b]->move[data[b]->state];
                // Each player chooses its next state based upon its
            }
        }
    }
}

```

```

        // rules for which state to choose based upon its
        // current state and the move of the opposite player.
data[a]->state
    = data[a]->nextState[data[a]->state][moveB];
data[b]->state
    = data[b]->nextState[data[b]->state][moveA];
if (moveA == COOPERATE && moveB == COOPERATE) {
    // Both players are rewarded for cooperating
    data[a]->fitness += 3.0;
    data[b]->fitness += 3.0;
    cc_count++;
}
else if (moveA == DEFECT && moveB == COOPERATE) {
    // Player B is a sucker for cooperating when player
    // A was tempted to defect
    data[a]->fitness += 5.0;
    data[b]->fitness += 0.0;
    dc_count++;
}
else if (moveA == COOPERATE && moveB == DEFECT) {
    // Player A is a sucker for cooperating when player
    // B was tempted to defect
    data[a]->fitness += 0.0;
    data[b]->fitness += 5.0;
    cd_count++;
}
else if (moveA == DEFECT && moveB == DEFECT) {
    // Both players are punished for defecting
    data[a]->fitness += 1.0;
    data[b]->fitness += 1.0;
    dd_count++;
}
}
}
}
} /* end of void Automaton::computeFitnesses(Automaton**,int,int) */

```

```

void Automaton::printStates(int state, bool reset) {
    if ( (state == 0) && (reset) )
        resetMarkers();
    if (marked[state])
        return;
    else {
        cout << state << ":";
        if(move[state] == 0)
            cout << "C";
        else
            cout << "D";
        cout << " " << nextState[state][COOPERATE] << " "
            << nextState[state][DEFECT] << endl;
        marked[state] = true;
        printStates(nextState[state][COOPERATE],false);
        printStates(nextState[state][DEFECT],false);
    }
} /* end of void Automaton::printStates(int,bool) */

```

```

int Automaton::nicenessIQ() {
    int IQ = (NUMBEROFSTATES*2);
    if (!isNice())
        return 0; // 0 for non-nice individuals
    for (int i = 0; i < NUMBEROFSTATES; i++) {
        int currentState = 0;
        for (int j = i; j > 0; j--) {
            currentState = nextState[currentState][COOPERATE];
        }
        currentState = nextState[currentState][DEFECT];
        if (move[currentState] == DEFECT)

```

```

        IQ += 2;    // Two points for a TIT_FOR_TAT response
    else {
        currentState = nextState[currentState][DEFECT];
        if (move[currentState] == DEFECT)
            IQ++;    // One point for a TIT_FOR_TWO_TATS response
        else
            IQ -= 2; // Subtract two points for failing to retaliate after
                    // two subsequent defections
    }
}
return IQ;
} /* end of int Automaton::nicenessIQ() function */

```

```

void Automaton::transplant(Automaton::Pack newIndividual) {
    for (int i = 0; i < NUMBEROFSTATES; i++) {
        for (int j = 0; j < OPPONENT_MOVES; j++) {
            nextState[i][j] = newIndividual.nextState[i][j];
        }
        move[i] = newIndividual.move[i];
    }
} /* end of void Automaton::transplant(Automaton::Pack) */

```

```

Automaton::Pack Automaton::get() {
    Automaton::Pack pack; // Genetic data is packed into this variable
    for (int i = 0; i < NUMBEROFSTATES; i++) {
        for (int j = 0; j < OPPONENT_MOVES; j++) {
            pack.nextState[i][j] = nextState[i][j];
        }
        pack.move[i] = move[i];
    }
    return pack;
} /* end of Automaton::Pack Automaton::get() function */

```