

```

/*
 * Classifier.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 "Classifier.h"
using namespace std;

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

int games_played;

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));
}

Classifier::Classifier() {//: rules(NUM_OF_RULES),history(RULE_LENGTH) {
    for (int i = 0; i < NUM_OF_RULES; i++) {
        //rules.setSize(NUM_OF_RULES);
        //history.setSize(RULE_LENGTH);

        // Initialize rule data randomly, and initialize history to match
        // anything as this is an individual that has not yet participated
        // in any games
        rules[i].choice = randInt(2);
        for (int x = 0; x < RULE_LENGTH; x++) {
            rules[i].criteria[x] = randInt(NUM_OF_MOVE_PAIRS);
            history[x] = M_XX;
        }

        // Initialize number of games played as well
        games_played = 0;
    }
} /* end of Classifier::Classifier() constructor */

Classifier::~Classifier() {
} /* end of Classifier::~Classifier() destructor */

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

bool Classifier::matches(int h, int c) {

    switch(h) {
        case M_CC:
            if ( (c == M_XX) || (c == M_XC) || (c == M_CX) || (c == M_CC) )
                return true;
            break;
    }
}

```

```

    case M_CD:
        if ( (c == M_XX) || (c == M_XD) || (c == M_CX) || (c == M_CD) )
            return true;
        break;
    case M_DC:
        if ( (c == M_XX) || (c == M_XC) || (c == M_DX) || (c == M_DC) )
            return true;
        break;
    case M_DD:
        if ( (c == M_XX) || (c == M_XD) || (c == M_DX) || (c == M_DD) )
            return true;
        break;
    case M_XC:
        if ( (c == M_XX) || (c == M_XC) || (c == M_CX)
            || (c == M_CC) || (c == M_DX) || (c == M_DC) )
            return true;
        break;
    case M_XD:
        if ( (c == M_XX) || (c == M_XD) || (c == M_DX)
            || (c == M_DD) || (c == M_CX) || (c == M_CD) )
            return true;
        break;
    case M_CX:
        if ( (c == M_XX) || (c == M_CX) || (c == M_XC)
            || (c == M_CC) || (c == M_XD) || (c == M_CD) )
            return true;
        break;
    case M_DX:
        if ( (c == M_XX) || (c == M_DX) || (c == M_XD)
            || (c == M_DD) || (c == M_XC) || (c == M_DC) )
            return true;
        break;
    default: // if (h == case M_XX)
        return true;
}
return false; // no match found
} /* end of bool Classifier::matches(int,int) function */

bool Classifier::ruleMatches(rule r) {
    for (int i = 0; i < RULE_LENGTH; i++) {
        if (!matches(history[i],r.criteria[i]))
            return false;
    }
    return true;
} /* end of bool Classifier::ruleMatches(rule) function */

int Classifier::decideMove() {
    double balance = 0.0;
    int lengthOfHistoryToMatch;

    if (games_played < RULE_LENGTH)
        lengthOfHistoryToMatch = games_played;
    else
        lengthOfHistoryToMatch = RULE_LENGTH;

    double spec[NUM_OF_RULES];
    double maxSpec = -1;

    if (lengthOfHistoryToMatch == 0) { // This is only true on first game
        for (int i = 0; i < NUM_OF_RULES; i++) {
            if (rules[i].choice == COOPERATE)
                balance++;
            else
                balance--;
        }
        if (balance > 0) // Vote is for COOPERATE

```

```

        return COOPERATE;
    else // Vote is for DEFECT
        return DEFECT;
}

else { // True on all games after the first

    int bestRuleIndex = 0;

    for (int i = 0; i < NUM_OF_RULES; i++) {

        double currentRuleSpecificity = 0.0;
        double currentRuleVotingWeight = 0.0;

        for (int j = 0; j < lengthOfHistoryToMatch; j++) {
            currentRuleSpecificity *= 0.8; // discount value of earlier games.
            if (matches(history[j],rules[i].criteria[j])) {
                switch (rules[i].criteria[j]) {
                    case M_CC:
                        currentRuleSpecificity += 5;
                        break;
                    case M_CD:
                        currentRuleSpecificity += 5;
                        break;
                    case M_DC:
                        currentRuleSpecificity += 5;
                        break;
                    case M_DD:
                        currentRuleSpecificity += 5;
                        break;
                    case M_XC:
                        currentRuleSpecificity += 2;
                        break;
                    case M_XD:
                        currentRuleSpecificity += 2;
                        break;
                    case M_CX:
                        currentRuleSpecificity += 2;
                        break;
                    case M_DX:
                        currentRuleSpecificity += 2;
                        break;
                    default: // a.k.a. case M_XX:
                        currentRuleSpecificity += 0;
                }
            }
        }

        spec[i] = currentRuleSpecificity;
        if (currentRuleSpecificity > maxSpec)
            maxSpec = currentRuleSpecificity;
    }

    if (maxSpec >= 0) { // Rules with maximum specificity vote on the decision

        balance = 0;
        for (int i = 0; i < NUM_OF_RULES; i++) {
            if (spec[i] == maxSpec)
                balance += (rules[i].choice == DEFECT)? 1 : -1;
        }

        if (balance > 0)
            return COOPERATE;
        if (balance < 0)
            return DEFECT;
        return randInt(2);
    }

    else { // There was absolutely no matching, so vote on move as if
        // no games had been played yet.
        cout << "nomatch\n";
    }
}

```

```

        for (int i = 0; i < NUM_OF_RULES; i++) {
            if (rules[i].choice == COOPERATE)
                balance++;
            else
                balance--;
        }
        if (balance > 0) // Vote is for COOPERATE
            return COOPERATE;
        else // Vote is for DEFECT
            return DEFECT;
    }
} /* end of int Classifier::decideMove() function */

void Classifier::update(int move_pair_constant) {
    // Shift history array
    for (int i = RULE_LENGTH-2; i > -1; i--) {
        history[i+1] = history[i];
    }

    // Add most recent move
    history[0] = move_pair_constant;

    // Calculate fitness reward and update fitness
    switch(move_pair_constant) {
        case M_CC:
            fitness += 3.0;
            break;
        case M_CD:
            fitness += 0.0;
            break;
        case M_DC:
            fitness += 5.0;
            break;
        case M_DD:
            fitness += 1.0;
    }
} /* end of void Classifier::update(int,bool) */

void Classifier::mutate() {
    /* if (randInt(10) == 1) {
        for (int i = 0; i < NUM_OF_RULES; i++)
            rules[i].choice = DEFECT;
        return;
    } */
    int reorderRules = 0; //randInt(2);
    int pos = randInt(NUM_OF_RULES); // Position in rule set where the
    // mutation will occur

    if (reorderRules == 0) {
        bool modifyCriteria = true;
        int randSwitch = randInt(RULE_LENGTH+1);
        if (randSwitch == RULE_LENGTH)
            modifyCriteria = false;
        if (modifyCriteria)
            rules[pos].criteria[randSwitch] = randInt(NUM_OF_MOVE_PAIRS);
        else
            rules[pos].choice = !rules[pos].choice;
    }
    else {
        int swapPos = randInt(NUM_OF_RULES); // Position in rule set of second
        // rule whose position will be
        // swapped with rule in position pos
        while (swapPos == pos) { // Ensure that pos and swapPos are in two
            // different positions in the rule set
            swapPos = randInt(NUM_OF_RULES);
        }
    }
}

```

```

    }
    rule tempRule = rules[pos]; // Temporary rule swapping location
    rules[pos] = rules[swapPos];
    rules[swapPos] = tempRule;
}
} /* end of void Classifier::mutate() */

void Classifier::crossover (Classifier& partner) {

    int cutChoice = randInt(100); // Used to decide whether to perform an
                                // incision or laceration type cut
    int rulepos = randInt(NUM_OF_RULES-2) + 1;

    if (cutChoice < 50) { // Laceration type cut (50% chance). This type
                        // of cut divides the individual's rules not only
                        // by a cut point in rules, but a cut point in
                        // the criteria of the rule at the rule cut point
        int critpos = randInt(RULE_LENGTH-1) + 1;
        for (int i = rulepos; i < NUM_OF_RULES; i++) {

            if (i == rulepos) {
                for (int j = critpos; j < RULE_LENGTH; j++) {
                    // Copy current classifier's criteria into a temp location
                    int tempCriteria = rules[i].criteria[j];
                    // Replace current classifier's data with partner's data
                    rules[i].criteria[j] = partner.rules[i].criteria[j];
                    // Replace partner's data with data in temp location
                    partner.rules[i].criteria[j] = tempCriteria;
                }
                // Copy current classifier's rule choice into a temp location
                int tempChoice = rules[i].choice;
                // Replace current classifier's data with partner's data
                rules[i].choice = partner.rules[i].choice;
                // Replace partner's data with data in temp location
                partner.rules[i].choice = tempChoice;
            }
            else {
                // Copy current classifier's data into a temporary location
                rule tempRule = rules[i];
                // Replace current classifier's data with partner's data
                rules[i] = partner.rules[i];
                // Replace partner's data with data in temporary location
                partner.rules[i] = tempRule;
            }
        }
    }

    else { // Incision type cut (50% chance). This type of cut divides
        // the individual's rules only by a cut point in the rules.
        for (int i = rulepos; i < NUM_OF_RULES; i++) {
            // Copy current classifier's data into a temporary location
            rule tempRule = rules[i];
            // Replace current classifier's data with partner's data
            rules[i] = partner.rules[i];
            // Replace partner's data with data stored in temporary location
            partner.rules[i] = tempRule;
        }
    }

} /* end of void Classifier::crossover(Classifier&) */

void Classifier::malign() {
    for (int i = 0; i < NUM_OF_RULES; i++)
        rules[i].choice = DEFECT;
} /* end of void Classifier::malign() */

void Classifier::computeFitnesses(Classifier** data, int size, int method) {

```

```

switch (method) {

    case EACH_VS_ALL:
        // 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++) {

                for (int i = 0; i < NUM_OF_GAMES; i++) {
                    // For each game: the move of each player is given
                    // by the action given by the () function
                    // of that player.
                    int moveA = data[a]->decideMove();
                    int moveB = data[b]->decideMove();
                    if (moveA == COOPERATE && moveB == COOPERATE) {
                        // Both players are rewarded for cooperating
                        data[a]->update(M_CC);
                        data[b]->update(M_CC);
                        cc_count++;
                    }
                    else if (moveA == DEFECT && moveB == COOPERATE) {
                        // Player B is a sucker for cooperating when player
                        // A was tempted to defect
                        data[a]->update(M_DC);
                        data[b]->update(M_CD);
                        dc_count++;
                    }
                    else if (moveA == COOPERATE && moveB == DEFECT) {
                        // Player A is a sucker for cooperating when player
                        // B was tempted to defect
                        data[a]->update(M_CD);
                        data[b]->update(M_DC);
                        cd_count++;
                    }
                    else if (moveA == DEFECT && moveB == DEFECT) {
                        // Both players are punished for defecting
                        data[a]->update(M_DD);
                        data[b]->update(M_DD);
                        dd_count++;
                    }
                }
                games_played++;
            }

            games_played = 0;
        }
    }

    break; // break out of this switch case

} // end of switch(method)

} /* end of void Classifier::computeFitnesses(Classifier**,int,int) */

void Classifier::printIndividual() {
    cout << endl;
    cout << "Current Individual:" << endl << endl;
    cout << "\tRule Set:" << endl;
    for (int i = 0; i < NUM_OF_RULES; i++) {
        cout << "\t\tRule#" << i+1 << ": ";
        for (int j = 0; j < RULE_LENGTH; j++) {
            switch(rules[i].criteria[j]) {
                case M_CC:
                    cout << "CC";

```

```

        break;
    case M_CD:
        cout << "CD";
        break;
    case M_DC:
        cout << "DC";
        break;
    case M_DD:
        cout << "DD";
        break;
    case M_XC:
        cout << "XC";
        break;
    case M_XD:
        cout << "XD";
        break;
    case M_CX:
        cout << "CX";
        break;
    case M_DX:
        cout << "DX";
        break;
    default: // a.k.a. case M_XX:
        cout << "XX";
    }
    cout << "\t";
}
cout << "--> ";
if (rules[i].choice == COOPERATE)
    cout << "C" << endl;
else
    cout << "D" << endl;
}
cout << endl << "\tHistory:" << endl << "\t\t";
for (int x = 0; x < RULE_LENGTH; x++) {
    switch(history[x]) {
        case M_CC:
            cout << "CC";
            break;
        case M_CD:
            cout << "CD";
            break;
        case M_DC:
            cout << "DC";
            break;
        case M_DD:
            cout << "DD";
            break;
        case M_XC:
            cout << "XC";
            break;
        case M_XD:
            cout << "XD";
            break;
        case M_CX:
            cout << "CX";
            break;
        case M_DX:
            cout << "DX";
            break;
        default: // a.k.a. case M_XX:
            cout << "XX";
    }
    cout << "\t";
}
cout << endl << endl << "\tFitness:" << fitness << endl << endl;
} /* end of void Classifier::printIndividual() */

```

```

void Classifier::transplant(Classifier::Pack newIndividual) {
    for (int i = 0; i < NUM_OF_RULES; i++)

```

```
        rules[i] = newIndividual.rules[i];
    } /* end of void Classifier::labotomy(Classifier::Pack) */

Classifier::Pack Classifier::get() {
    Classifier::Pack pack; // Genetic data is packed into this variable
    for (int i = 0; i < NUM_OF_RULES; i++)
        pack.rules[i] = rules[i];
    return pack;
} /* end of Classifier::Pack Classifier::get() function */
```