Computer Modeling of Cultural Interaction and Evolution

AiS Challenge
Final Report
April 3rd, 2002

Team 90
Silver High School

Team Members
Roeland Hancock

Teacher
Mrs. Peggy Larisch

Mentors
Mr. Steve Blake
Dr. David Harris

Executive Summary.

Human culture is a dynamic and difficult beast. Sociologists strive to create accurate models of our culture, but have limited data from a limited period of observation, and the general result is an incomplete and poorly explained hypothesis.

Through the use of a large-scale computer model it should be possible to simulate the basic shape of cultural evolution. Such a simulation would provide a very valuable research tool and give great insight into the mysteries of man as a social creature.

In order to produce an effective simulation, and agent-based, or individual level, simulation environment was created. The initial population was allowed perform simple, random interactions, which were then refined over generations using a genetic-algorithm. For simplification the driving force behind culture was assumed to be survival.

Introduction

Culture among human beings is a complex, dynamic and abstract thing, but understanding it is vital to managing our society and understanding the past. Sociologists face difficulties in their analysis of culture due to the long time periods involved. Culture does not, generally, change in the short time period that sociologists have to observe. A sociologist may observe a society for a number of years, decades or even a lifetime, perhaps becoming accustomed to the gradual changes and failing to observe the significance of the change. After this one lifetime of spotty observation there is a discontinuity in the data, the data is now subjected to the interpretations of a new personality, or the society may be forgotten entirely.

By using a computer model, it is possible to gain an insight into cultural changes over the long term. With a supercomputer tens of thousands of generations can be simulated in some degree of detail in the span of perhaps a few weeks. Initial conditions can be changed and new factors introduced at any time and the agent's responses analyzed almost instantly. The sociologist is now free to control every detail and observe every response. He is no longer limited to observation of things beyond his control; he can now formulate and test a hypothesis with great ease and speed.

A slightly similar model, Sugarscape (Epstein 1996), was created in the mid-nineties at the Sante Fe Institute and MIT. Sugarscape, however, was a more general agent modeling experiment, rather than a dedicated attempt to model culture.

Description

To simulate the evolution and adaptation of culture, some adaptive computational technique must be applied. A natural choice is the use of a genetic algorithm. A genetic algorithm's functionality is based on the Darwinian evolution in the natural world and has been shown to be extremely effective in many optimization situations.

A genetic algorithm is applicable to problems which either require an exponential amount of

time to solve or that have no defined method of solution. These problems are often characterized by multiple and complex, sometimes even contradictory constraints, that must be all satisfied at the same time. One notable success of a GA is the solution of the traveling salesman problem.

A GA follows the model of biological natural selection. The factors to be taken into account are given values, initially random, comparable to a chromosomes and forming individuals. These individuals are combined, mutated and evaluated according to the desired solution. These offspring will include more fit individuals. The superior offspring continue to breed, while the lesser fit individuals are eliminated through selective pressure.

It is not necessary to have a formal method of solution when using GA's to approach a problem, only the end goal needs to be defined. The GA is a purely computational and procedural approach.

Breeding occurs by through a simple algorithm. A random point is picked in the genome; this becomes the pivot and divides each genome in two, creating four gene segments, A1, A2, B1, and B2. Two new genomes are created through the sequence A1B2, B1A2. The parents remain in the gene pool in the case of agent reproduction and are removed in cultural reproduction. In the case of cultural breeding the fittest parents remain through the next generation to prevent the unlikely possibility of total degeneration, a technique known as 'favored offspring'.

The simulation is composed of several parts: the environment, the society, the agent, and the culture.

The environment is a very simplistic ecological simulation whose primary purpose is to restrict the population of the society. The environment also incorporates predators to create a risk element, and provides natural resources. The development of technology by the society is assumed to reduce the impact of the environment on the society.

It is important to distinguish between society and culture. A society is a group of individuals who share the same physical environment or who bond together for a common purpose. A culture is the way of life of a society, including customs, ideas and morals. Thus the society is simply a collection of agents, whereas the culture is the structure which governs the behavior and interactions of the agents. The culture derives from the society

The agent is the lowest and most important level of the simulation. An agent represents a single individual. Each agent operates independently attempting to satisfy its own needs and desires while cooperating within the accepted framework of the culture. And agents actions are evaluated according to the culture, a negative rating lessens (but does not eliminate) its chances of reproducing, leading to the eventual removal of its genes from the gene pool. What actions an agent chooses to perform are based on the agents characteristics as well as how it has performed in the past.

The core of the simulation is agent interaction. The simulation is set up on a cyclical basis, each cycle an agent may perform one action from a pre-defined list (see Appendix A). The original intention was to allow the list of possible actions to be dynamically modified, but this proved to be infeasible. Each action has an effect on the society and individual that can be evaluated and

analyzed for impact. The agent receives a score for each action, determined through the algorithm in figure 1.

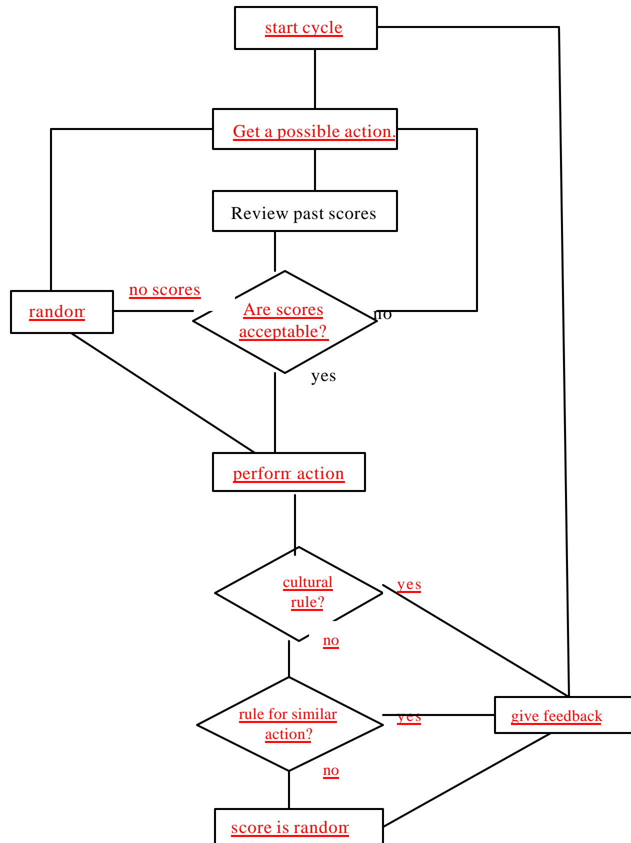

Figure 1

Initially environment society and culture are randomly generated. The society and culture are then reproduced across several separate streams of execution. These reproductions allow different cultures to be compared under the same environmental conditions.

A society is composed of many different types of people. These people shape the culture, but are also shaped by the culture and strive to remain within the socially acceptable norms. This will be represented with the simultaneous adaptation of the agent to the culture and the culture to the agent. To effect this, there must be two almost separate simulations, and two separate courses of evolution, that of the culture and that of the agent.

To successfully use a GA, fitness criteria must be established. The question of what makes one culture better than another must be answered. This is, unfortunately, a very subjective question that is difficult to answer. After much research, discussion with a mentor and analysis of this project's goals, a working definition of the goal was reached. A 'good' culture is defined to be one which facilitates the survival of the society and individual. The fitness for an agent comes from how well the agent works with the present culture.

The program makes use of the object-oriented features of C++, relying on classes and inheritance for data types and algorithms. By using classes, common generic code can be created for a basic algorithm or component of the simulation. A new component can then be easily created, using

the generic code as a parent class.

As genetic algorithms demand heavy computational resources, this project will be executed on theta, a 128-processor supercomputer at Los Alamos. In order to utilize the power of a multi-processor (MP) machine, special programming techniques must be used. A common method of using an MP machine is the message passing interface (MPI), a C, C++ and FORTRAN library that allows a program to distribute tasks across processors and consolidate results.

The main bottleneck in a GA is evaluation, so the fitness algorithm is the logical place to begin parallelization. A central processor maintains control of the program and input/output interactions. This central (root) processor then delegates tasks to other processors, which operate simultaneously, and consolidates the results from each processor. Since there are multiple factors in the fitness algorithm, each factor is assigned a number. The processors responsible for the fitness algorithm are assigned corresponding numbers, and evaluate only the factors matching their number in the pool. Once each factor has been computed, the results are summed by the central processor and used as the final fitness score. Other tasks, such as breeding and environmental simulation, are also sent to specific processors.

Each society is scored based on the survival and reproduction rates. The cultural rules for each society are then bred. A new society based on the parent society is produced and allowed to operate for several generations. After several generations, the societies are compared, the less fit are discarded and new ones are bred.

Results

The program successfully applies a GA to the culture and agents, providing generally improving fitness scores. The simulation provides different patterns of development based on initial conditions and specifications of how different actions should be interpreted. For an output of the development of both culture and agents see Appendix C.

Conclusions

The project did not achieve the intended results, although it is of some use. Its format for presentation of the data lacks coherence needed to make this the valuable research tool intended. The simulation does successfully apply a genetic algorithm to agents and culture, producing more effective culture and better adjusted agents. It is possible to make a limited observation of how deviance is defined by observing changing scores for individual actions. With more discussions with a sociologist and better development of the user interface this can become an extremely valuable simulation. In its current state the program allows for the observation of how an agent and culture change over time, but does not correlate these results to the causes. With more knowledgeable and experimental use, i.e. by a sociologist, this program would be of more use and relevance.

Recommendations

During the course of this project the original scope of detail was restricted to allow the project to be completed in the designated time frame. A more detailed model would be advisable and of greater use. The goals for the GA are subject to opinion. Other goals may be a more accurate model, or yield alternative, more desirable results.

To allow greater flexibility, the use of variable length genomes was originally intended. After consideration, it was decided that variable length genomes would add an unwanted level of complexity and programming to the project. A future project using variable length genomes would have a much more dynamic structure, more accurate results and allow a wide range of possibilities. Similarly, the action list was intended to be dynamic, introducing new actions as the complexity increased. This could be implemented using a runtime interpreter of codes.

The code uses the STL vector class in several situations where a linked list would be more appropriate.

The program is primitive and lacks a detailed output of the data that is needed for true understanding of cultural development.

References and Acknowledgements

I would like to thank Richard Allen, Steve Blake, Anna Kettenhofen, John Hancock, David Harris and Peggy Largish for their support and assistance throughout the development of this project and LANL for allowing the use of the theta machine.

1. Epstein, Joshua and David Axtell. Growing Artificial Societies. 1996, MIT Press
2. Holland, John. Adaptation in Natural and Artificial Systems. 1992, MIT Press
3. Koza, John. Genetic Programming. 1992, MIT Press.

Appendix A-Agent interactions

KILL	
DIE	
EAT	Required behavior at least once every three cycles. Uses resources.
SLEEP	This is required at least once every four cycles.
GETK	Get resources from the environment or another agent.
GETR	Get knowledge(specialized resource) from environment or another agent.
SHAREK	Trade knowledge with another agent.
SHARER	Trade resources with another agent.
BREED	Reproduce in combination with another agent.
INVENT	Enhance quality of life.

Appendix B-Source Code

```
/*(c) 2002 GPL Roeland Hancock

*AIS 2002
*ceval.cpp cultural fitness with MPI
*/

#include <mpi++.h>
#include <vector>
#include "processors.h"
#include "protos.h"

int evaluateCulture(int aspects, int deaths, int births, int population, int
society) {
//importance of various aspects
const float dPercent=-.2;
const float aPercent=.6;
const float pPercent=.15;
const float kPercent=.05;
int score, subScore;

      MPI::Bcast(&aspects, 1, MPI_INT, 0, MPI_COMM_WORLD);
      if (!(apsects < procSize)) { //there are more apsects to evaluate than
processors

            if (procRank==P_FIT_C_EVAL) { //find death rate-negative score
                  if(population < environment.size()) {
                        int pscore= (int)
(population/environment.size()*100);
                  }
                  else {
                        int pscore=(int) (environment.size()/population*-
100);
                  }
                  int kscore=(int)(kPercent*societies[society]->knowledge());
                  subScore=(int)(pscore*pPercent+dscore*dpercent+ kScore);
            }

            }

            if(procRank==P_FIT_C_AGENT) {//find average agent scores- this
should take the longest
            vector<int> scores;

                  for (int i=0; i<societies[society]->size(); i++) {
                        int mask=societies[society]->mask();
                        vector<int> a =agents.members(mask);
                        for(int c=0; c< a.size(); c++) {
                              scores=agents[a[c]]->scores();

      subScore+=(int)(scores.accumulate(scores.begin(), scores.end(),
0)/scores.size());
                        }
                        subScore=(int)(subScore/c);
                        subScore*=(int)(subScore*aPercent);
```

```
            }
        }


    }
    MPI::Reduce(&subScore, &score, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
//get final rating
    return(score);
}



/*(c) 2002 GPL Roeland Hancock

*AIS 2002
*main.cpp establishes root processor, user IO
*/
#include <iostream.h>
#include <mpi++.h>
#include “processors.h”
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include “protos.h”
pid_t work;
int cycle=0;
int go=1;
int choice=0;

int main() {
/*fork execution to keep ui avaliable*/
    if (work=fork()) {//this is the parent
    cout << “Simulator running as " << work << “\n”;

    while (go==1) {
        while (choice==0) {
            cout << "1\tStop simulation\n”;
            cout << "2\tDump current generation to cout\n”;
            cout << “>”;
            cin >>choice;
            cout << “\n”;
        }/*(c) 2002 GPL Roeland Hancock *AIS 2002
*main.cpp establishes root processor, user IO
*/
#include <iostream.h>
#include <mpi++.h>
#include “processors.h”
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include “protos.h”
pid_t work;
int cycle=0;
int go=1;
int choice=0;

int main() {
```

```
/*fork execution to keep ui avaliable*/
      if (work=fork()) {//this is the parent
      cout << “Simulator running as " << work << “\n”;

      while (go==1) {
            while (choice==0) {
                  cout << "1\tStop simulation\n”;
                  cout << "2\tDump current generation to cout\n”;
                  cout << “>”;
                  cin >>choice;
                  cout << “\n”;
            }
            switch choice {
                  case 1:
                        go=0;
                        //halt();
                        break;
                  case 2:
                        dump();
                        choice=0;
                        break;
                  default:
                  choice=0;
            }


      }
      if(!kill(work, SIGKILL);


      else {//this is the child which actually runs the simulation
            MPI::Init();
            int procRank = MPI::COMM_WORLD.Get_rank();
            int procSize = MPI::COMM_WORLD.Get_size();
            MPI::Bcast(&pool, 1, MPI_INT, 0, MPI_COMM_WORLD) {


            MPI::Finalize();
      }
      return (0);
}
            switch choice {
                  case 1:
                        go=0;
                        //halt();
                        break;
                  case 2:
                        dump();
                        choice=0;
                        break;
                  default:
                  choice=0;
            }


      }
      if(!kill(work, SIGKILL);
```

```
      else {//this is the child which actually runs the simulation
            MPI::Init();
            int procRank = MPI::COMM_WORLD.Get_rank();
            int procSize = MPI::COMM_WORLD.Get_size();
            MPI::Bcast(&pool, 1, MPI_INT, 0, MPI_COMM_WORLD) {


            MPI::Finalize();
      }
 return (0);
}

/*(c) 2002 GPL Roeland Hancock

*AIS 2002
*processors.h controls processor pool
this program is designed for 16 processors
*/
#ifndef _PROCESSORS_H
/*each culture runs on its own processor*/
#define P_NODE0   10
#define P_NODE1   11
#define P_NODE2   12
#define P_NODE3   13
/*each culture has its own environment*/
#define P_ENVIRONMENT_0 14
#define P_ENIVIRONMENT_1      15
#define P_ENVIRONMENT_2 16
#deifne P_ENVIRONMENT_3 17

#define P_FIT_C_AGENT   18
#define P_FIT_C_EVAL    19
#define P_FIT_A_BREED   110
#define P_AP_EVAL 111
#define P_AP_EXEC 112
#define _PROCESSORS_H
#endif

/*(c) 2002 GPL Roeland Hancock
*AIS 2002
*main.cpp establishes root processor, user IO
*/
#include <iostream.h>
#include <mpi++.h>
#include "processors.h"
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
pid_t work;
int cycle=0;
int go=1;
int choice=0;
int pec0[2]={P_ENVIRONMENT_0, P_NODE_0};
int pec1[2]={P_ENVIRONMENT_1, P_NODE_1};
int pec2[2]={P_ENVIRONMENT_2, P_NODE_2};
```

```
int pec3[2]={P_ENVIRONMENT_3, P_NODE_3};
MPI_Group p0, p1, p2, p3;
environment eZero, eOne, eTwo, eThree;
culture cZero, cOne, cTwo, cThree;
int main() {
/*fork execution to keep ui avaliable*/
	if (work=fork()) {//this is the parent
	cout << "Simulator running as " << work << "\n";

	while (go==1) {
		while (choice==0) {
			cout << "1\tStop simulation\n";
			cout << "2\tDump current generation to cout\n";
			cout << ">";
			cin >>choice;
			cout << "\n";
		}
		switch (choice) {
			case 1:
				go=0;
				//halt();
				break;
			case 2:
				dump();
				choice=0;
				break;
			default:
			choice=0;
		}


	}


	else {//this is the child which actually runs the simulation
		societies.initialize();
		MPI::Init();
		int procRank = MPI::COMM_WORLD.Get_rank();
		int procSize = MPI::COMM_WORLD.Get_size();
		/* splits into separate groups*/

		MPI::Group_incl(MPI_COMM_WORLD, 2, pec0, p0);
		MPI::Group_incl(MPI_COMM_WORLD, 2, pec1, p1);
		MPI::Group_incl(MPI_COMM_WORLD, 2, pec2, p2);
		MPI::Group_incl(MPI_COMM_WORLD, 2, pec3, p3);
		//run 4 copies of the simulation. each runs for a set period of
time and then reports its fitness
		while (1) {
		MPI::Bcast(&pool, 1, MPI_INT, 0, pec0);

		int fitness=0;
		culture myCulture;
		environment myEnvironment;
		myEnvironment.run();
		myCulture.run();
		MPI::Reduce(&subScore, &score0, 1, MPI_INT, MPI_SUM, 0, pec0);
		MPI::Bcast(&pool, 1, MPI_INT, 0, pec1);
```

```
            int fitness=0;
            culture myCulture;
            environment myEnvironment;
            myEnvironment.run();
            myCulture.run();
            MPI::Reduce(&subScore, &score1, 1, MPI_INT, MPI_SUM, 0, pec1);
            MPI::Bcast(&pool, 1, MPI_INT, 0, pec2);
            int fitness=0;
            culture myCulture;
            environment myEnvironment;
            myEnvironment.run();
            myCulture.run();
            MPI::Reduce(&subScore, &score2, 1, MPI_INT, MPI_SUM, 0, pec2);
            MPI::Bcast(&pool, 1, MPI_INT, 0, pec3);
            int fitness=0;
            culture myCulture;
            environment myEnvironment;
            myEnvironment.run();
            myCulture.run();
            MPI::Reduce(&subScore, &score3, 1, MPI_INT, MPI_SUM, 0, pec3);


            }



            MPI::Finalize();
      }
 return (0);
}



class environment {
public:
      environment();
      ~environment();
      new int predators=10;
      new double size=100;
      new const double growth=1.05;
      new const double regrowth=1.01;
      new const double die=1.2;
      bool overpopulated(int population);
      int run(&society soc);
      bool capcity=0;
      bool regrow=0;
      new double resourcesA;
      int useResource(int & r, int & d);




};
environment::environment() {


environment::~environment(){
```

```
      delete predators;
      delete size;
      delete growth;
      delete regrowth;
}
inline int environment::useResource(int & r, int & d) {
      r=0;
      if(resourcesA) {
            resourcesA--;
            r=1;
      }
      predators/

}
inline bool environment::overpopulated(int population) {
      if ((predators + population) >= size) {
            return(1);
      }
      else {
            return (0);
      }
}
inline int run(&society soc) {

      if (regrow) { //the population exceeded capacity last cycle
            if (overpopulated(population->soc)){ //the population still
exceeds capacity
                  capacity=1;
                  size*=die;
                  regrow=1;
            }
            else {
                  size*=regrowth;
                  regrow=0;
            }
      }
      else if (overpopulated(population->soc)){
                  capacity=1;
                  size*=die;
                  regrow=1;
      }
      else {
            size*=growth;
      }



      resourcesA=size;

/*(c) 2001 GPL Roeland Hancock
agent.cpp AIS 2002 definitions and member functions for
 agents*/
class agent {
 public:
   agent();
   ~agent();
   void setGenome(vector <int> newGenome);
```

```
   history getHistory();
   vector<int> getGenome();
   void addFeedback(int code, int score);
   int doAction();
 private:
   new history actionHistory;
   new vector<int> genome;
   new int parents [2];
   new int brother;
};
inline ~agent() {
   delete actionHistory;
   delete genome;
   delete parents;
   delete brother;

}
void setGenome(vector <int> newGenome) {
   genome=newGenome;
}
vector<int> getGenome() {
   return(genome);
}
int doAction() {
   vector<int> pActions=getActions();
}



/*(c) 2001 GPL Roalnd Hancock
history.cpp AIS 2002  definitions for history and access funcs*/
class history {
public:
       history();
       ~history();
       void addFeedback(int action, int score);
       int[2] getLast(int action);
}
void addFeedback(int action, int score) {
```