Multitarus: Agent Communication Framework

P.B. Lubell-Doughtie[1]
Stanford University

May 19, 2005

Abstract

A multi-agent problem is described in which communication among the agents proves to be essential to its efficient solution. Genetic (Evolutionary) Programming is used to evolve agents that can solve the problem by communicating amongst each other.

I. Introduction

In this paper I ask the question: does the evolution of communication among agents lead to enhanced performances? In addition, if it does lead to enhanced performance, under what circumstances is this so? Enhanced performance includes solving the problem faster, solving the problem more efficiently, or both.

I also wish to determine whether or not communication will evolve naturally in a simulated world when there is an incentive for it to develop. The basic framework for the possibility of communication is provided, and within this framework agents are allowed to evolve their abilities for goal achieving actions aided by communication.

The approach used is to extend Astro Teller's Tartarus program (Teller) to allow for communication. I compare the results seen from those agents that can communicate and those agents that cannot communicate. In Teller's program, a single robot moves up, down, left, and right in a two dimensional grid-world environment. The robot can sense in all eight directions N, NE, …, NW. Its objective is to push as many boxes (which are randomly placed in the environment at the beginning of the game) as possible to the outer edges of the grid environment. The game runs for a fixed number of moves and once completed the likelihood of the agent moving on to the next generation of individuals and playing the game once more (its fitness) is increased by one point for boxes on the outer edge of the grid and two points for boxes in a corner of the grid.

II. The Experimental Setup

A. Acting and Sensing

In Multitarus, the extension of Teller's Tartarus program, the domain is specified by a set of parameters which are set at the beginning of a run of the game. The domain

[1] Thanks to Nils Nilsson and John Koza for their continuing help with these ideas

parameters that can be specified include the width of the grid, the height of the grid, the maximum number of moves an agent can make (the number of time steps per game), the number of boxes in the grid world, and the number of agents in the grid world. Each square within the grid may be empty, contain a box, or contain any number of agents.

The agent can be thought of abstractly as a set of sensors and effectors. Alternatively, the agent may be thought of more concretely as a robot with limited motion capabilities operating within an artificial environment. The agent is able to move in the same ways in which an agent in Tartarus can move (Up, Down, Left, Right). If an agent comes to a wall and moves in the direction of the wall, a time step is taken but the agent does not move and no effect is seen on the environment.

When an agent encounters a box in grid world the agent is able to push the box only if two conditions are satisfied. (1) There is another agent within a certain distance (this distance may be infinity to negate this requirement) and (2) the grid square beyond the box in the direction it is being pushed is empty. If the agent is unable to push the box a time step is taken and the agent and box remain in their previous positions. If the agent is able to push the box then both the agent and the box move one step in the direction the box was pushed.

The agent is able to sense what exists in the eight grid squares that surround it.

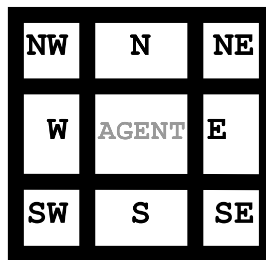


Figure 1: A grid section with sensing areas labeled

A section of the whole grid is displayed above. The cells labeled **N** through **NW**, proceeding clockwise, correspond to grid cells that the agent (labeled by AGENT) can sense. The input to the agent in the cells is determined by what is in the cell.

B. Memory and Communication

Each agent in the environment has its own array-based memory structure which is set at a size of *n* cells. The environment allows for several different types of communication. Because the memory involved in the program is compartmentalized in various separate areas, communication can take place (1) between separate agents through their direct alteration of each others' memory, (2) between separate agents through alteration of an external (to the agents memory) data cell, and (3) within a single agent (see figure below).

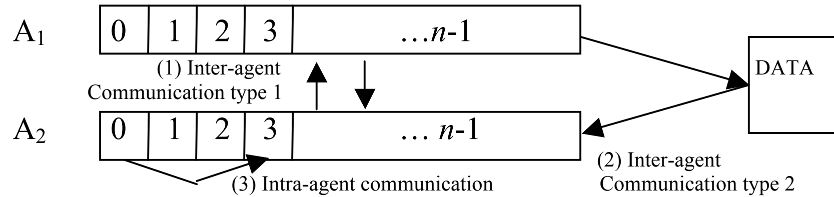


Figure 2: Three types of agent communication

As can be seen above these three types of communication are fundamentally different. Type (1) is the most direct and is analogous to the form of communication we employ in direct face to face communication. For a type of communication analogous to type (2) we may consider one individual writing something in a book or posting something on a message board and another person reading it. Type (3) can be equated with thought where one changes ones own memory structure.

Two classes of functions are involved in communication. There are the intra-agent read and write functions and the inter-agent read and write functions. First the intra-agent functions will be described and then the inter-agent functions will be described in a way that builds upon the intra-agent function descriptions.

What occurs during a typical read from memory is that the number provided as an argument, which is an integer, assigns the value in that numbered space in the agent's memory to the agent's acting memory retrieval position. The memory retrieval position is the space labeled DATA in Figure 2. It is an extra position in memory set up to hold values that are going to be referenced later, are frequently referenced, or are shared between agents.

When an agent uses the write function the process is slightly more complicated. The first argument which is taken specifies the space in the agent's memory to write into. The second argument which is taken is the data value to be written into the memory location specified by the first argument. The space in the agent's memory is written into and the agent's memory retrieval position is replaced with the value which was in the agent's memory at the location specified by the first argument before it was overwritten with the data value supplied by the second argument. This is done primarily because Teller's Tartarus program operated in this fashion. Programmatically it operates as a way to keep the agents knowledge in circulation; the memory is flushed back into the system.

To demonstrate a write function suppose n = 5, the memory retrieval position is set to [5], agents 1's memory is [0,1,2,3,4] and the function call *Write*([4], [2]) is executed. [4] is the space in the agents memory to write into and [2] is the value to be written. After the function call the memory retrieval space is now [4] because this is the value in space [4] (where array index is done from 0 to n – 1). The agent's memory is now [0,1,2,3,2].

The communication functions operate very similarly to the read and write functions. The read from agent function (*ReadAgent*) takes two arguments. The first argument specifies which agent data will be read from. The second argument specifies which memory cell in that agent will be read. The read memory goes from the other agent's memory cell into the memory retrieval cell of the reading agent.

The write agent function (*WriteAgent*) takes three arguments. The first argument is which agent data is going to be written into. The second argument specifies where that

data will be written in the agent's memory and the third argument contains the actual data to be written. The data that was previously in the cell to be written into is placed in the memory retrieval cell.

C. Genetic Programming

To answer the questions posed in the introduction, namely to evaluate the effectiveness of communication and to see if it develops naturally, a method of evolutionary computing called genetic program was used. This method involves the evolution of simple programs through program branch recombination, reproduction, and mutation (see Koza 1992). Below the *genetic program tableau* for this analysis is presented and explained:

Objective:	Evolve a program which is capable of pushing as many boxes as possible to the outside of a two dimensional environment (version. 1) allowing agents to push boxes when in contact with them (version 2) allowing agents to push boxes only when another agent is within a radius of a certain number of squares.
Terminal Set:	Ephemeral Random Constant, *LowerLeft*, *LowerMiddle*, *LowerRight*, *MiddleLeft*, *MiddleRight*, *UpperLeft*, *UpperMiddle*, *UpperRight*
General Function set:	*Add*, *Equal*, *IfThenElse*, *Less*, *Max*, *Not*, *Sub*
Communication Function set:	*Read*, *Write*, *ReadAgent*, *WriteAgent*
Testing Function set:	*Transport*
Fitness cases:	Position in the environment of each box. A box in a corner earns two points and a box on the edge of the environment earns one point. The box in the corner earns two points because it is earns one point for each wall it is against.
Raw Fitness:	The maximum possible fitness of the individual being tested minus the fitness of that individual.
Adjusted Fitness:	The raw fitness adjusted to a scale from 0 to 1.
Hits:	One hit is scored for each box on the outside walls, two hits for each box in a corner.
Wrapper:	{0 < MoveForward < 6} {7 < TurnLeft < 13} {14 < TurnRight < 19}
Parameters:	Individuals in population M = 1024, Number of generations run G_1 = 101 G_2 = 51, Elitism is used with a pool size of E = 10.
Result Designation:	The best individual so far is designated as the result.
Success Predicate:	An individual will be deemed successful

	when it has the maximum number of hits or is the individual which scores closest to the maximum number of hits.

Table 1: Genetic Programming Tableau

The terminals used in the program are equivalent to those described above in figure 1. Each terminal holds in it the value set at location described by the terminal.

Figure 3: Sample grid portion; terminal description

In figure 3 above the terminal MiddleRight would be 1 indicating that there is a box in that location. The terminal LowerRight would be 4 indicating that agent A2 (or agent n – 2, where n is the value on the map) is in that location. The remainder of the terminals would be 0 indicating that nothing is in that location. If we consider a situation where an agent is on the edge of the grid the squares outside the grid are coded 2. The way in which the general functions work is the same as the way they work in Astro Teller's Tartarus.

The general functions work as follows. *Add* takes two arguments and returns the value of those arguments under addition with an upper limit equal to the number of memory cells minus 1. This is done so that if this value is referenced by a *Read*, *Write*, *ReadAgent*, or *WriteAgent* function no out of bounds errors will occur. *Equal* returns 1 if the two arguments supplied to it are to equal to each other and 0 otherwise. The *IfThenElse* takes three arguments. The syntax is *If* argument one greater than 0 *Then* argument two *Else* argument three. The *Less* function returns 1 if the first argument is less then the second and 0 otherwise. The *Max* function works by returning the first argument if it is greater than the second and doing nothing otherwise. The *Not* function operates by return 1 if the single argument it takes is 0 and returning 0 otherwise. The *Sub* function returns the result of two arguments under subtraction with a strict lower bound of 0; this is again for memory indexing.

The communication functions have been described above and work in the implementation as described above. The transport function is included for testing purposes only. It was designed to make the problem more tractable in a shorter time period so that parameter and ADF inclusion effectiveness could be evaluated. It works by changing the location of the agent to the location of the other agent in the environment when it is called.

The wrapper describes how movement in the environment takes place. If the result of the evaluation of the program is a number between 0 and 6 the agent moves forward, if the number is between 7 and 13 the agent turns left, and if the number is between 14 and 19 the agent turns right.

A portion of the runs conducted using the above tableau involved automatically defined functions or ADFs (see Koza 1992). ADFs are a way for genetic programming to create useful subroutines. In the runs in which ADFs were used there is

one main program branch which is able to call the two ADFs. The first ADF is not able to call any functions and it does not take any arguments. The second ADF takes one argument and is able to call the first ADF.

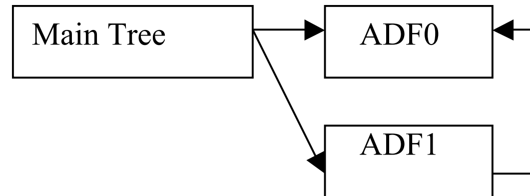


Figure 4: ADF interactions

In all runs tournament selection and elitism are used. Tournament selection is a process which maximizes the diversity in the next generation's population while at the same time gathering fit individuals for movement into the next generation. In tournament selection a group of a certain size, usually 7, is chosen at random and then the individual with the highest fitness value is chosen for reproduction (movement into the next generation without change to its program structure). The group or pool size used for tournament selection in all experiments run was 10. This value was seen to work more effectively then 7. Elitism means that the program is run using breeding policies that try to preserve the best individuals in the population.

The program was run within the ECJ 11 Evolutionary Computation and Genetic Programming Research System.[2] This is a strongly-typed system. This means that the programs created within it are typed and contain multiple different types. In this system the pipelines used for input into a new population were set as follows: crossover 85%, reproduction 10%, and mutation 5%. These numbers were found to be appropriate after experimentation with various settings (specifically, settings without mutation). The ECJ 11 system uses the Mersenne Twister Fast algorithm[3] to generate random numbers, settings, and parameters. This is the randomization algorithm used throughout.

In the grid environment the size is set at 6 wide by 6 high, 6 boxes are randomly placed, n = 20 memory cells are used per agent, there are N = 2 agents, and the number of time steps used per run per agent is set at 2000. During a time step the following program segments run once for each agent in the order in which they are presented below: (1) The environment is updated (2) Agent N's program is run. After all agents have had their turn a complete cycle in the environment has taken place. This complete cycle runs for each time step (2000 times) and this simulation runs M = 1024 times (as defined in parameters), once for each individual in the population. 1024 simulation runs is defined as one generation's run.

III. Description of Experiments

[2] This system was written by Sean Luke, Liviu Panait, Gabriel Balan, Zbigniew Skolicki, Jeff Bassett, Robert Hubley, and Alexander Chircop. More information is available at: http://cs.gmu.edu/~eclab/projects/ecj/.

[3] A description of the Mersenne Twister algorithm can be found at: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf.

A. Nearness Criteria

As delineated in the program objective section of the genetic programming tableau two versions of the game were played. In one version of the game for a block to be able to be pushed there had to be another agent within a certain radius of the agent doing pushing. This version of the game has what I call the "nearness restriction."

In the versions of the game played with the nearness restriction there must be one agent pushing against the box (that is moving in the direction of the box while the square opposite the box is clear) and another agent must be located within a radius of one, two, or three (zero is also possible but this was not analyzed). If we consider Figure 5 below we see that if agent A1 moves to the right on its turn it will be able to push the box under 2 and 3-nearness. Under 1 or 0-nearness (0-nearness is the requirement that the agents be at the same location for the box to be pushed) the box would not be pushable.

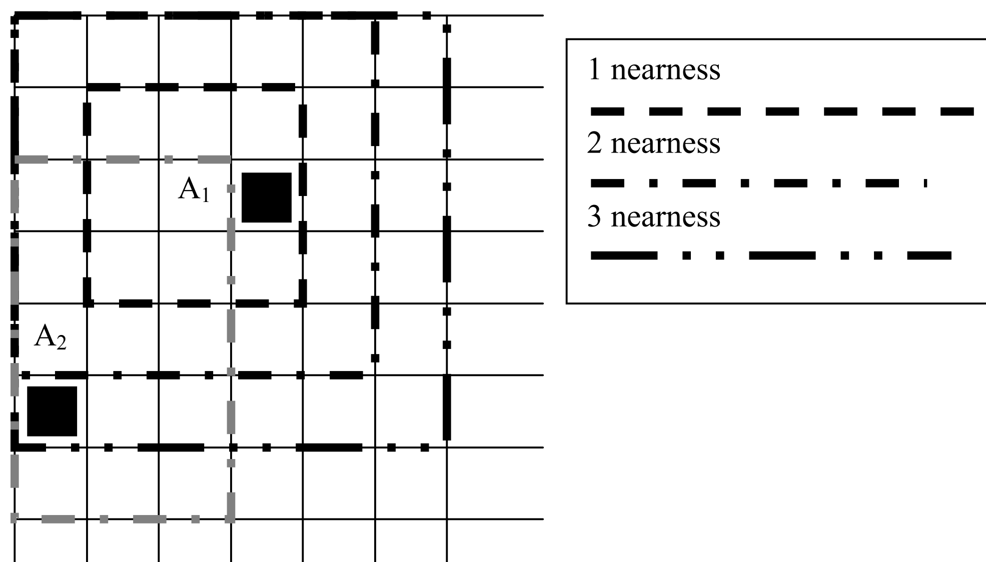


Figure 5: Varying nearness amounts

A subtlety of this implementation of communication is that within one time step of one run the environment may change so that what was initially a block able to be pushed is no longer able to be pushed. Considering figure 5 and suppose that the nearness restriction is set at 2-nearness. We see that before agent one's turn has taken place agent two is able to push the block against the wall down. Now suppose that agent one's turn evaluates to Move Forward and the agent is facing right. After agent one moves agent one will be out of range of agent two under 2-nearness. Agent two will not be able to push the block down in this situation.

A nearness restriction is implemented in order to provide motivation for communication between agents. The reasoning behind this is that agents that are able to work together through communication will be better able to push the blocks to the outside of the grid. This is because those that communicate more effectively will be able to instruct agents to follow them around or move with them or towards them so that pushing blocks is possible.

The program as it is written now implements an ideal and uncommon situation in which what is communicated undergoes no signal loss or degradation of data integrity. This situation is uncommon because in nature, or any non-digital environment, there always exists some type of noise, which precludes perfect signal integrity. As such we must recognize that comparisons between the Multitarus world and the real world will only be valid to within a certain approximation.

B. The Experiments

In order to evaluate the effectiveness of communication, experiments were run with varying amounts of nearness and, with communication and without. It was expected that experiments with a higher amount of nearness (3 as opposed to 2 or 1) would be solved faster and with less use of communication or ADFs.

In the genetic programming runs presented below the agent is tested under four separate sets of programming guidelines, and then the results are analyzed separately. A comparison is made first looking at the effectiveness of ADFs. Then a second comparison is made to look for the effectiveness of communication.

For testing the usefulness of ADFs a comparison is made between allowing communication without ADFs and allowing communication with ADFs. During these runs the nearness requirement is set at 2 to provide motivation. The best of run individual after 50 generations in a prohibiting-ADFs run is shown below:

```
Evaluated: true
Fitness: Raw=8.0 Adjusted=0.11111111 Hits=2
Tree 0:
 (+ (read lower-right) (max lower-right lower-middle))
```

This program adds the result of reading from the memory location specified by what is in the `lower-right` terminal to the greater of the values from the `lower-right` and `lower-middle` terminal. The resulting value is then evaluated according to the wrapper and the agent acts accordingly. This very simplistic program is only obtaining its hits because of the random setup of the board and accomplishes nothing toward the goal.

The best of run individual after 50 generations, with ADFs, is shown below:

```
Evaluated: true
Fitness: Raw=7.0 Adjusted=0.125 Hits=3
Tree 0:
 (less lower-left (IF-then-else lower-left
     transport (less [1] upper-right)))
Tree 1:
 (read-agent (equal [18] transport) (not transport))
Tree 2:
 (+ lower-left (equal (write lower-right lower-left)
     (less middle-right lower-right)))
```

This agent is able to accomplish more hits than without ADFs but this appears not to be because of the ADFs. The important part of this program seems to be the `IF-then-else` function in the main tree, Tree 0. This function uses `transport` if there is a box or a wall or an agent in the lower left corner relative to the agent's orientation. If there is not a box, wall, or agent in the lower left corner relative to the agent's orientation the `IF-then-else` branch returns 1 minus what is in the `upper-right` terminal. Trees 1 and 2 (the ADFs) are never executed because, as can be seen by examining Tree 0, they are never called.

Now that it has been determined that, in a testing environment, programs that are evolved with ADFs are come closer to the solution of the problem, we can test the effectiveness of communication. To test communication we will compare simulations with ADFs and without communication to those with ADFs and with communication. This will be done under 3, 2, and 1-nearness. It is expected that as the nearness requirement becomes less the problem will become more difficult to solve and more ADF and communication usage will occur.

IV. Experimental Results

All experiments were conducted using programs with a main tree and two ADF trees. The parameters are the same as stated above and displayed in the genetic programming tableau. The nearness varies from experiment to experiment.

A. Three-Nearness

The first experiment conducted used 3-nearness. A sample tree allowing communication and graphs comparing runs where communication is allowed and where it is disallowed are provided below for analysis:

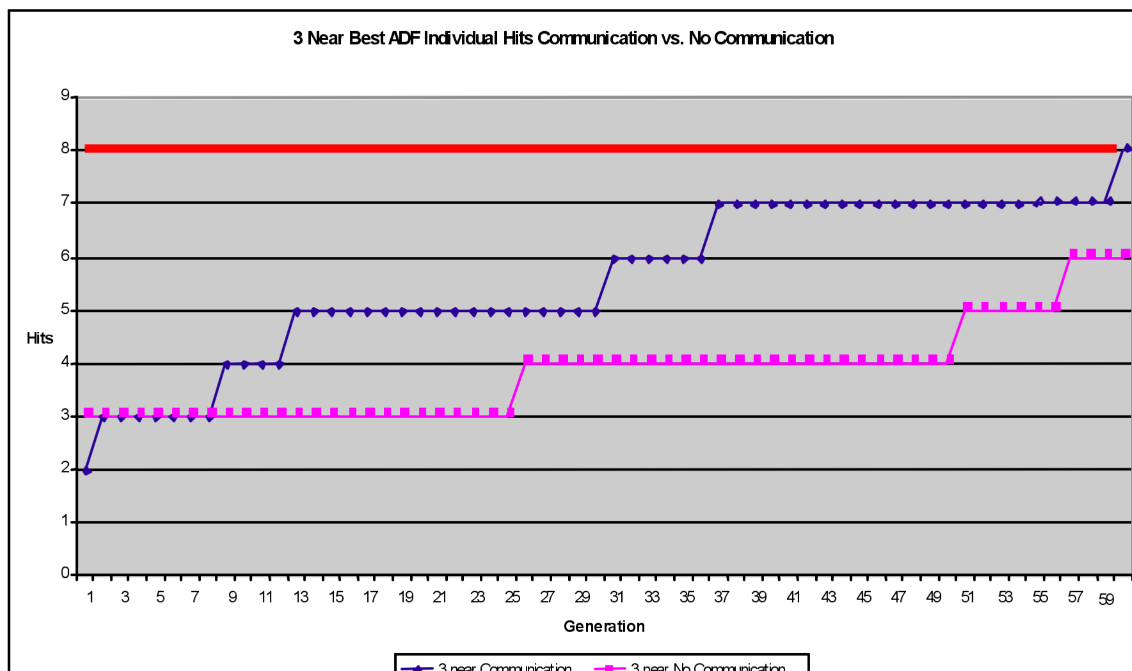
```
Evaluated: true
Fitness: Raw=2.0 Adjusted=0.33333334 Hits=8
Tree 0:
 ADF0[1]
Tree 1:
 (write (not (less [7] [0])) (IF-then-else
     (- [12] [7]) (write-agent (write [19] [12])
     (less [6] [2]) (- (less (write (write (less
     (equal [18] [19]) (less (less [5] [16]) [15]))
     (write [14] [5])) (not (read-agent (less
     [5] [16]) (- [18] [18])))) (equal (IF-then-else
     (- [6] [15]) (write-agent (read [6]) (equal
     [10] [0]) [18]) [11]) (less (IF-then-else
     (write [14] (write [15] [1])) (equal [1]
     [19]) (- [12] [7])) (equal (write-agent [14]
     [12] [10]) [7])))) [11])) (read-agent (read
     [1]) (- [15] [0]))))
Tree 2:
 [19]
```

The program above earns 8 hits. The seemingly important part of this tree is Tree 1 because it contains the majority of the code and is executed through a call to it by the main tree. This ADF's main branch is a `Write` function that places true or 1 in the memory cell found by executing and making use of an `IF-then-else` branch. The first child of the `IF-then-else` is *{12 - 7}* = *2* which is greater than 0 and because of this the `else` branch of the tree is never executed and instead the `then` branch is executed. Within this `then` branch `write-agent` is called and the result of `less [6] [2]` (0) is written into the space of the evaluation of the following tree 1 code segment:

```
(- (less (write (write (less
(equal [18] [19]) (less (less [5] [16]) [15]))
 (write [14] [5])) (not (read-agent (less
 [5] [16]) (- [18] [18])))) (equal (IF-then-else
 (- [6] [15]) (write-agent (read [6]) (equal
 [10] [0]) [18]) [11]) (less (IF-then-else
 (write [14] (write [15] [1])) (equal [1]
 [19]) (- [12] [7])) (equal (write-agent [14]
 [12] [10]) [7])))) [11])
```

The rest of the program contains multiple `write,read,write-agent,read-agent` functions to allow for a good deal of communication between agents and within agents.

Below is a graph of the best individual's number of hits with communication versus the best individual's number of hits without communication and with ADFs in both situations over a time period of 60 generations:



Graph 1: 3-Nearness ADF best individual hits with communication versus no communication. The red line indicates the maximum possible hits.

The darker line, which corresponds to agents that communicate, quickly rises above the lighter line, corresponding to those that cannot communicate. It can be seen that as more generations pass those agents that are allowed to use communication score more hits than those that are not allowed to use communication. These agents therefore come closer to solving the problem at hand.

B. Two-Nearness

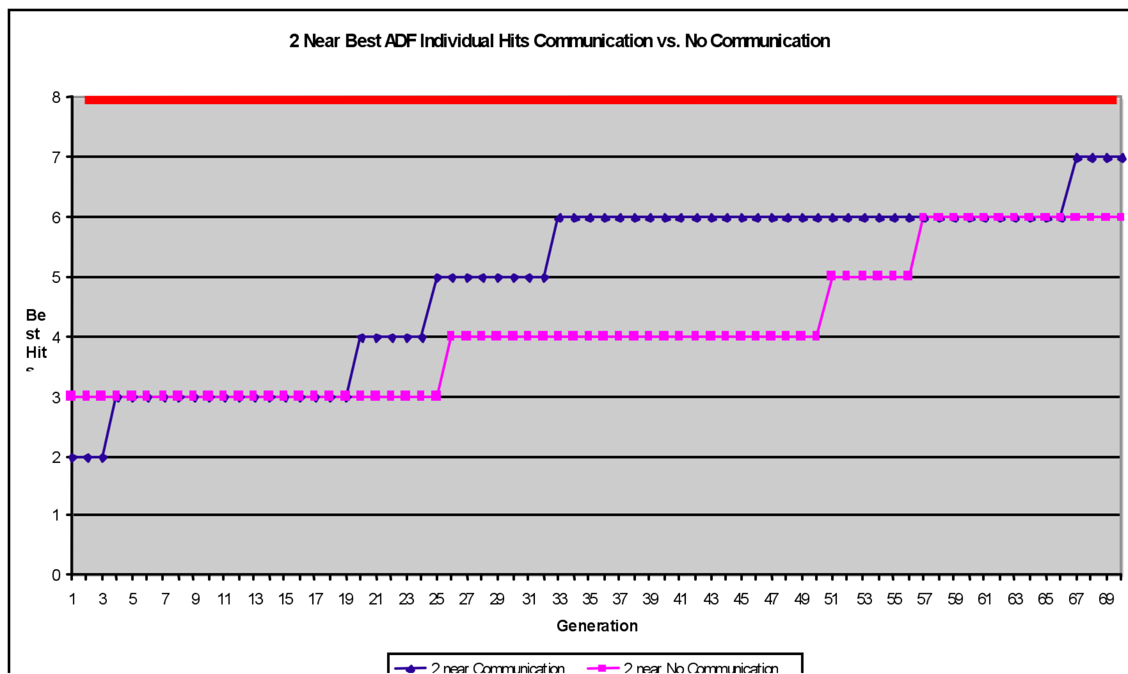
The second experiment conducted was designed to be more difficult and used 2-nearness. A sample tree allowing communication and graphs comparing runs where communication is allowed and where it is disallowed are presented below:

```
Evaluated: true
Fitness: Raw=3.0 Adjusted=0.25 Hits=7
Tree 0:
 (- (write (read-agent ADF0[1] (not (ADF1[2]
     ADF0[1]))) (max (read (ADF1[2] middle-left))
     (write-agent (write (max lower-right upper-middle)
         (ADF1[2] [12])) (max lower-middle lower-left)
         (equal ADF0[1] upper-right)))) (less (read
     (less (IF-then-else middle-right middle-left
         lower-middle)    (write-agent    lower-middle
middle-left
         middle-left))) (write-agent (- (less (ADF1[2]
     ADF0[1]) (IF-then-else (ADF1[2] ADF0[1])
     upper-right  upper-middle))  (read-agent  middle-
right
     (ADF1[2]  ADF0[1])))  (read  (read  (less  (IF-then-
else
     middle-right  middle-left  lower-middle)  (write-
agent
     lower-middle middle-left middle-left))))
     (+ [2] upper-middle))))
Tree 1:
 (not (IF-then-else (read-agent (- [11] [19])
     (- (equal [1] (not [19])) (read-agent (-
         [11] [19]) (- (equal [1] [19]) (write-agent
         (less [19] (IF-then-else (read-agent (- [11]
             [19]) (- (equal [1] (- [11] [19])) (read
             [4]))) (read-agent (- (read-agent [2] [5])
             (IF-then-else  [4]  [16]  [18]))  (-  [11]
[19]))
             (IF-then-else  (read-agent  (-  [11]  [19])
(- equal [1] (less [19] [15])) (read [4])))
                 (read-agent  (-  (read-agent  [2]  [5])
(write-agent  (write  [14]  [3])  (read-agent  (read  [4])
(write [1] [8])) (write [15] [1]))) (read-agent
```

```
                        (read-agent [9] (equal [12] [10]))
(- (read-agent [16] [1]) (equal [17] [6])))) (not (not
(write-agent (less [19] [15]) (write [16] [5]) (-
(read [7]) [19])))))))) (write [16] [5]) (less [9]
          [7])))))) (read-agent (- (read-agent (-
(read-agent [2] [5]) [13]) (not (write-agent (less (-
     [11] [19]) [15]) (write [16] [5]) (- (read
     [4]) [19])))) (IF-then-else [4] [16] [18]))
     (read-agent [16] [1])) (not (not (write-agent
     (less (- [11] [19]) [15]) (write [16] [5])
     (- [11] [19]))))))
Tree 2:
 (+ (+ upper-middle upper-middle) upper-middle)
```

The program above earns 7 hits. Tree 0 and Tree 1 of the above program make ample use of the `read`, `write`, `read-agent`, and `write-agent` functions. It is likely that it is because of this usage that the program is able to score the number of hits it did.

Below is a graph of the best individual's number of hits with communication versus without communication and with ADFs in both situations over a time period of 60 generations:



Graph 2: 2-Nearness ADF best individual hits with communication versus no communication. The red line indicates the maximum possible hits.

The darker line, which corresponds to agents that are allowed to communicate, rises above the lighter line (corresponding to agents that cannot communicate). Once above the line it stays above. The agents who were able to communicate therefore come closer to solving the problem at hand.

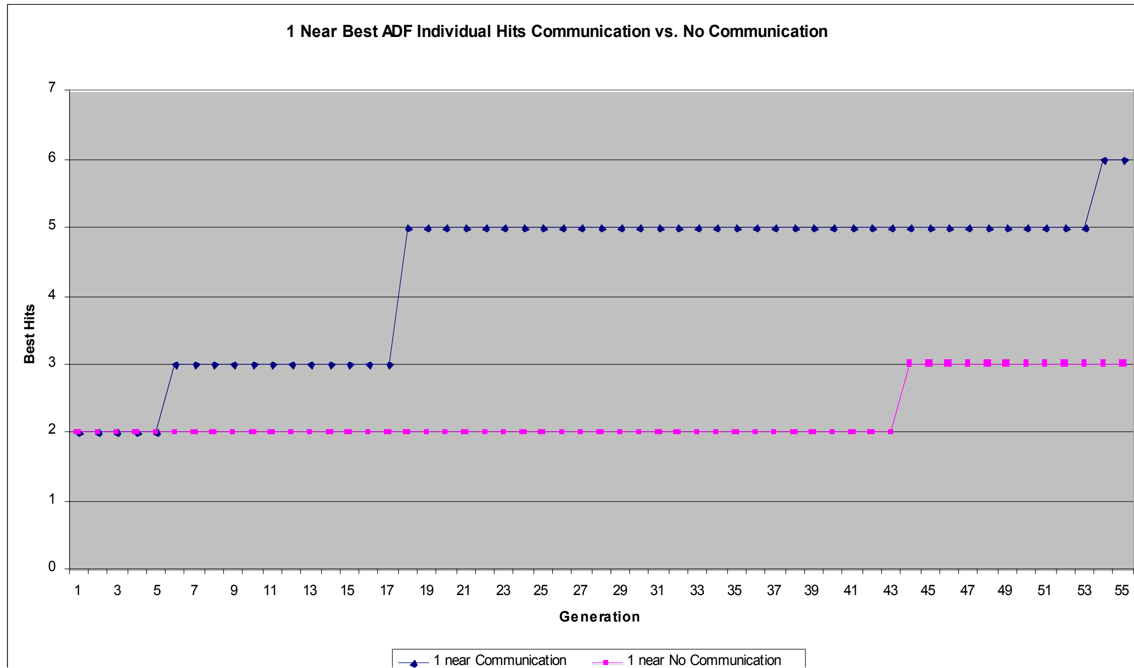
C. One-Nearness

The third experiment conducted was the most difficult; more difficult than 3 and 2-nearness, it used 1-nearness. A sample tree allowing communication and graphs comparing runs where communication is allowed and where it is disallowed are presented below:

```
Evaluated: true
Fitness: Raw=6.0 Adjusted=0.14285715 Hits=4
Tree 0:
 (write-agent (+ (ADF1[2] middle-right) (not
     (less lower-left middle-left))) lower-left
     (- (not middle-right) (read-agent (read-agent
         ADF0[1] middle-left) lower-right)))
Tree 1:
 (equal [2] (write-agent [4] (write-agent
     (write-agent (- (IF-then-else (write-agent
         (- [17] [0]) (- (not [2]) (read [19])) (IF-
then-else
         (read [9]) (IF-then-else (not (- (- [17]
         (write [10] [13])) (read [19]))) (equal
(write [6] [3]) (IF-then-else [11] [15] [0])) (write-
agent
         (not [3]) (read-agent [18] [2]) [0])) [13]))
         (less [15] [3]) (write-agent [14] [10] [5]))
         (not [5])) (- (not [5]) (read [19])) [13])
     (not (- (write [10] [13]) (- [13] (read [19]))))
     [3]) (IF-then-else (not [5]) (equal (write
     [6] [3]) (IF-then-else [11] [15] [0])) (write-
agent
     (not [3]) (- [17] [0]) [0]))))
Tree 2:
 upper-right
```

The program above earns 4 hits. Tree 0 and Tree 1 of the above program use intertwined communication function calls with references to the grid environment. It is likely that because of this the program is able to score as many hits as it does considering the difficulty of the problem presented. The best individual for the population was eventually (see graph below) able to earn 6 hits.

Below is a graph of the best individual's number of hits with communication versus without communication and with ADFs in both situations over a time period of 60 generations:



Graph 3: 1-Nearness ADF best individual hits communication versus no communication. Maximum possible hits (8) not on graph.

The darker line, which signifies agents that are allowed to communicate, rises above and then once above the lighter line (signifying agents that cannot communicate) stays above. The agents who were able to communicate therefore come closer to solving the problem at hand. In this case it can be seen that those agents that communicated did significantly better than those that did not.

V. Conclusions

We have seen, that in this multi-agent implementation of the simple Tartarus grid-world environment (provided by Astro Teller), agents that have the ability to communicate with other agents perform better. These agents have a significant advantage that allows them to solve the problem, or approach a solution to the problem, whereas agents without communication capabilities cannot solve the problem or do not come as close to solving the problem. In addition, agents with communication capabilities came closer to solving the problem, or solved the problem, in a fewer number of time-steps than those without communication capabilities.

We can create three classes for multi-agent problems:

1. Those for which communication hinders finding a solution.
2. Those for which communication is irrelevant to finding a solution.
3. Those for which communication helps to find a solution.

The experimental results seen in this research appear to belong to class 3. It seems anomalous that as the difficulty of the problem increases, that is as the nearness requirement decreases from three to two, communication is not more effective. We see

that in 2-nearness communication appears less effective than in 3-nearness. Under 2-nearness the problem presented is neither too difficult for communication to solve the problem nor too simple for communication to hinder the finding of a solution. There exist situations in which communication may hinder the finding of a solution by extraneous functions. Similarly the problem of neither being too difficult nor too simple is present in 3-nearness. A middle ground exists in problem difficulty where communication simply becomes less important; these are the problems that belong to class 2 above.

With that said, there exist some multi-agent problems for which communication between agents results in finding a solution much faster and efficiently than if the agents are not allowed communication. This can be seen above in graph 3. This graph clearly shows the effectiveness of communication in the sharp rise in number of hits received by agents with communication versus those without communication when 1-nearness is used.

We would expect that if the problems where to get easier communication would be less important. This is because the problem as a whole would be easier to solve. The subcomponent of the problem of attracting agents to an agent trying to push a block would have a larger probability of being solved by chance.

In addition, we would suspect that making the problem harder would make the ability to communicate even more important. In this situation the problem would be harder and take longer to solve. Considering the harder 0-nearness problem in which a block can only be pushed when there are two agents at the same location, communication between agents would be useful. This is because communicating to an agent to direct them onto the spot the pushing agent is on would make the problem easier.

VI. Future Work

A possible alteration of the Mulitarus system developed is to change the objective of the game so that, instead of pushing boxes to the outer portion of the environment, the agents gather items in the environment which can either affect them positively or negatively. Whether the items are positive or negative would depend on various characteristics of the item, such as its appearance. This would engender communication because as the agents gathered items they would learn which are positive and which are negative. This information could then be transmitted to other agents so that all the agents would quickly learn to stay away from the negative and head towards the positive. This simulation would be particularly instructive because it seems reasonable that it may be similar to the way in which language and communication developed in the natural world.

A possible extension of the already developed Multitarus system is to run it with altered parameters. Specifically, the number of boxes in the environment, the size of the environment, and the number of agents in the environment could be increased. This would allow for effective use of communication because there would be more space for agents to be outside of the nearness requirement.

References

Koza, J., *Genetic Programming*, MIT Press, Cambridge-U.S.A., 1992.

Koza, J., *Genetic Programming II*, MIT Press?, Cambridge-U.S.A.?, 199X?.

Langdon, W., Qureshi, A., Genetic Programming - *Computers using "Natural Selection" to generate programs*, University College London, http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/surveyRN76.pdf

Norvig, P., Russel, S., Artificial Intelligence, Pearson Education,
Upper Saddle River-U.S.A., 2nd edition, 2003

Teller, Astro, "Learning Mental Models", Carnegie Mellon University

Werner, G., Dyer, M., "Evolution of Communication in Artificial Organisms", *Artificial Life II*, 659, Addison-Wesley Publishing Company, 1992