Robinson, A., and L. Spector. 2002. Using Genetic Programming with Multiple Data Types and Automatic Modularization to Evolve Decentralized and Coordinated Navigation in Multi-Agent Systems. In *Late-Breaking Papers of GECCO-2002, the Genetic and Evolutionary Computation Conference.* Published by the International Society for Genetic and Evolutionary Computation.

Using Genetic Programming with Multiple Data Types and Automatic Modularization to Evolve Decentralized and Coordinated Navigation in Multi-Agent Systems

Alan Robinson
School of Cognitive Science
Hampshire College
Amherst, MA 01002
arobinson@hampshire.edu

Lee Spector
School of Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

Abstract

This work applies PushGP (a multi-type, automatically modularizing genetic programming system) to the 3D Opera problem (a cooperation and navigation multi-agent task involving the movement of a 3D swarm of agents through a constrained exit point). Within this framework we explore the effect of adding task-specific data types to the GP system. In particular, we extend the native types of PushGP to include 3D vectors, and we compare the results with and without this extension to each other and to human-programmed agent controllers.

1 INTRODUCTION

Successfully evolving programs that control agent behavior is tricky, and the focus of much genetic programming research. We are interested in the application of genetic programming techniques to multi-agent system problems that are challenging to human programmers as well as to automatic programming systems. Problems that have human designed solutions provide a particularly rich context in which to evaluate the results of GP runs. Genetic programming can be used to explore the constraints of these problems and to examine the possibility of alternative solutions. Furthermore, if the problem is difficult enough that it merits the effort of manually programming a solution, then it is interesting to see how effective a result a genetic programming system can evolve.

If a genetic programming system is to be used on problems humans find difficult, then it should have available to it the programming features that a human programmer would also have. At a minimum, we hypothesize this means a system supporting program decomposition into modular functions and the ability to operate on multiple data types. Furthermore, if such a system is to find novel solutions then the configuration of the system must be open ended enough to allow evolution to explore a rich solution space. In this work, we examine the application of Push and PushGP (a pairing designed to supply these features) to the 3D Opera problem, a navigation and coordination problem.

The goal of this paper is to explore how providing automatic modularization and multiple data types changes the behavior of the evolved agent, and how the evolved programs compare to solutions and techniques designed manually. When designing a multi-type system an issue is which data-types to include. We experiment with adding a highly domain-specific data-type to PushGP. Specifically, we add a 3D vector data-type, and examine the solutions evolved before and after that addition for differences in performance or behavior. We also examine whether or not PushGP makes use of modularity in the best performing evolved programs.

1.1 WHAT FOLLOWS IN THE NEXT SECTIONS

We will describe the technologies used in this work in the following order: the opera problem, including the specific simulation we used; the Push programming language and the PushGP genetic programming system; the specific encoding of the 3D opera problem for PushGP. We will then present the results of the two experiments that were conducted (one with no vector type, and the other with), and summarize our conclusions.

2 TECHNOLOGIES

2.1 THE 3D OPERA PROBLEM

The Opera problem is a decentralized multi-agent navigation and coordination problem introduced by Crespi, et al. (2001). The problem is defined for any two or greater dimensional space; in our work we focus on the three dimensional version. The task is to move a collection of agents inhabiting a bounded space through a fixed exit point in the minimum possible amount of time, while maintaining a "polite" distance between each agent. (The name of the problem derives from the problem faced by humans trying to leave a crowded opera house after a show.) There is no centralized control of movement; each agent is controlled separately and given only local information about its neighbor's positions. The behavior of each agent must minimize the time taken to reach the exit while at the same time avoiding getting too close to other agents. Since the exit is small in size, overcrowding results if the agents move at maximum speed directly to it. A successful agent must balance high speed with cooperative behavior.

In Crespi, et al. (2002) the 3D opera problem was solved with a gradient descent algorithm on a local potential function that treated the exit as a valley and the nearby agents as hills. By following the path of steepest descent, agents found paths to the exit while maintaining a reasonable average distance between each other. In our work we attempt to use evolution to find solutions to this problem without presupposing a specific algorithmic technique.

2.2 3D OPERA SIMULATION

We wrote our own version of the 3D opera simulation in Lisp to allow easy integration with Push/PushGP. The simulation is functionally and behaviorally modeled on the simulation used by Crespi, but for simplicity is based on uniform-length time steps. The constants used (exit size, room size, etc) are based upon the values typically used previous research.

At the start of the simulation agents are distributed throughout the interior of a 10x10x10 unit cube. The simulation is then run for a fixed number of time steps (the number depending on the fitness case). At each step every agent's controller (a Push program) is supplied with the coordinates of its location, the coordinates of the exit, and the coordinates of all agents within a radius of 1.5 units. The agent must respond with a vector describing its next movement in 3D space; if this vector is larger than 0.25 units it is normalized to 0.25 units. At the end of each time step all the agents are moved according to the vectors they returned. No collisions are simulated. After the agents move, however, the number of agents within collision distance of each other (.75 units) is recorded for use in the fitness function. Before the next time step every agent within 0.5 units of the exit is removed from the simulation. The location of the exit varies for different fitness cases, but is always located on a wall of the cube.

2.3 PUSH AND PUSHGP

We used the Push language and the PushGP genetic programming system to evolve programs that control the behavior of agents. Push is a multi-type stack-based language designed for genetic programming research. Push contains 50-odd instructions, including mathematical operations, stack manipulations, Boolean operations, and instructions for conditional code execution. Push instructions pop their arguments from the appropriate stack and push their results back. Multiple stacks (Integer, Float, Code, Boolean, Symbol, Name, and Type), allow for multiple types of data to be processed by a single Push program without syntactic constraints. Instructions either draw from specific stacks, or consult the Type stack to determine which stack(s) to use. The Code stack can be used to manipulate Push code, which can be executed on demand at any time during a program run. By modifying the contents of this stack with Push's rich code manipulation instructions any number of subroutines and modular control structures (all of which may be recursive) can be created on the fly, without the need for pre-specification of the number of modules or the number of parameters that they take.

To drive evolution we used PushGP, a fairly traditional genetic programming environment that evolves Push programs. Fitness-based selection is achieved via tournaments, and new individuals are produced via crossover and mutation operators. Because the Push language inherently supports the creation of subroutines and modular control structures, the PushGP system needs no explicit mechanisms to provide Koza style ADFs. The Push language and the PushGP system are further documented in Spector (2001), Spector and Robinson (2002a), Craford-Marks and Spector (2002), and online at `http://hampshire.edu/lspector/push.html`.

In previous work we have used Push and PushGP to evolve agents for controlling traffic grids (Spector and Robinson, 2002b). In the present work we chose to apply it to the 3D Opera problem because we are interested in observing if its ability to create modular code and its support for multiple data types would be utilized effectively in a more complex domain. Furthermore, we were interested in experimenting with the addition of a vector data type to Push, to see how this would change the behavior of evolved programs that manipulate entities in 3D space.

2.4 PUSHGP PROBLEM ENCODING

We evolved programs that would be fully executed every time step, agent by agent. The input to the program was the coordinates of the agent, the exit, and the other nearby agents. Programs were evaluated for up to 400 Push instructions. The top of the appropriate stack was then treated as the vector of the direction the agent should

move next. In experiment one, all coordinates are trios of values pushed on the float stack. In experiment two, coordinates are pushed as single 3D vectors onto the vector stack.

Table 1: PushGP parameters for all experiments

PARAMETER	VALUE
Population size	5000
Tournament Size	5
Fitness Cases	5
% Mutation	45
% Crossover	45
% Reproduction	10
Instruction Set, standard Push operators	ephem.-random-integer, ephem.-random-float, ephem.-random-boolean, ephem.-random-symbol, convert, get, set, noop, =, rep, swap, pop, dup, max, min, >, <, *, -, +, /, pulldup, pull, not, or, and, size, length, extract, insert, container, contains, subst, position, member, nthcdr, nth, append, list, cons, cdr, car, null, atom, quote, map, if, do*, do, integer, float, boolean, type, name, code
Instruction Set, vector operators	Tovec, tofloat, vshift, vscale, vmul, vadd, vector

Table 2: PushGP vector operators

OPCODE	RESULT
Tovec	Pop the top three items of the float stack and push the result as a vector
Tofloat	Pop the top vector and push the result as three floats on the float stack
vshift	Add the top float to each element of the top vector and pop the float stack.
vscale	Scale the top vector by the top float and pop the float stack
vmul	Multiply and pop the top two vectors and push the result
vadd	Add and pop the top two vectors and push the result

In all experiments the instruction set included both standard Push operators and the vector operators. This kept the search space size similar between both experiments. In experiment one the use of vectors was prevented by converting them to no-ops when executed.

Fitness was defined as `(too-close + (remaining * (too-close + 1) ) )`, with `too-close` as the number of times two agents got close enough to "collide" and `remaining` as the sum of the distance of all remaining agents from the exit after the simulation terminates. We add 1 to `too-close` because otherwise if too-close is zero, fitness would always be zero, independent of the value of `remaining`. This metric was chosen because it rewards improving both `too-close` and `remaining` at the same time, and because zero fitness was only possible if both were zero.

Table 3: The five fitness cases used

NUMBER OF AGENTS	NUMBER OF TIME STEPS
2	25
10	30
10	30
10	30
25	100

Fitness cases differed in terms of the number of agents, how long the simulation ran, and the random initial distributions of agents inside the 10x10x10 cube. The same set of distributions were used for all runs and experiments in this paper to facilitate comparison, and to ensure that evolution selected for more successful programs, rather than programs run with lucky distributions.

Measuring the fitness of all cases on current hardware resources was very slow, so heuristics were used to improve evaluation speed. The fitness for the quickest (and easiest) cases were found first. If the performance was worse than that of a null program, further testing was aborted, and default fitness values were used for the remaining cases. The default values corresponded to a little worse than the fitness of the null program for those cases.

3 EXPERIMENTS

3.1 EXPERIMENT 1: WITHOUT VECTORS

16 runs were initiated, however, due to time constraints only 2 runs were completed to 49 generations, and only 8 additional runs were completed to at least generation 30. Completing more runs would probably not change the results much, however, since most improvement happened very early in the runs. For each run, an individual within 1% of the best fitness of that run was usually found by generation 10, and always by generation

15. Furthermore, all but two runs ended with nearly the same fitness (245), with the exceptions being one run ending with fitness of 240 (the best found in all runs), and another ending with 280.

Though there was some variation between the individuals produced by different runs, all simplified to programs that returned a single fixed vector, regardless of the sensory information provided to the agent. The result of evaluating these programs was all agents moving in a straight line for their entire lifetime. Improvements in fitness consisted of finding a direction that happened to send the most agents into the exit, while minimizing the number of agents getting close enough for collision. Interestingly, in five of the programs evolved, PushGP made use of the DO or DO* instruction, one of its main methods for evolving modular code, though the result of the instruction was only to duplicate one of the numbers that would eventually be used to construct the constant vector. No programs used stack manipulation instructions to explicitly duplicate the contents of the float stack, suggesting that something about programs referencing their own code was more effective than direct stack manipulation.

Although more runs would be needed to be sure, it appears that a constant vector is too strong a local minimum to allow reactive programs to establish themselves and evolve in the experimental framework that we used. This is probably due to the amount of code required to process trios of floats as vectors being unlikely to evolve at the same time that effective use of vectors evolves. We will see that even when provided with vectors in experiment 2, it usually takes several generations before they are utilized successfully.

Perhaps, then, this problem, without the benefit of vector types, is best considered as a challenge problem for GP. In order to do better than a fixed vector, the system must be very effective at construction of modular code and promote the exploration of significantly less fit areas of the search space for long enough to develop the right building blocks.

3.2 EXPERIMENT 2: WITH VECTORS

Six runs were initiated, but due to time constraints only three completed all 49 generations. The rest completed at least 30 generations. There was a wider range in fitness over these runs than in the previous runs, and the number of runs/generations was not the same as in the first experiment, so we must be particularly careful in making comparisons.

Table 4: Fitness results for all experiment 2 runs

Best fitness	**Gen. Evolved at**	**Gen. of fitness < 240**
138	31	8
206	15	0
208	41	6
217	35	8
221	23	4
226	23	8

As seen in Table 4, in every case, the fitness of the best of run individual was better than the very best individual from the first experiment. While more generations were completed for these runs than for the first experiment, on average it took only 6 generations to evolve better performance than the best result from experiment 1. Therefore, the performance advantage cannot be explained by the number of generations completed. Further note that the best of run fitness was usually found by generation 30. Since 10 runs from experiment 1 completed at least 30 generations, the total number of individuals evaluated and important generations completed is roughly the same, if not higher on the part of experiment 1. We conclude, therefor, that while more runs and more consistent runs would be ideal, these results can be used for comparison.

The individuals evolved had highly varied code, however, of the six, four had basically the same behavior. One of the more simple programs that evolved this behavior appeared at generation 41 of one run: `(-0.0053015 -0.043026447 -0.043867588 TOVEC VMUL -0.193120718 VSCALE -0.8248620 VSCALE)`. Note that this program has been simplified from its original form to this functional equivalent in order to save space and facilitate describing its function.

This program, which had a fitness of 208, constructs a small magnitude vector, multiplies it times the agent's location, and then scales the resulting vector by roughly 0.16. The result is used as the direction of motion for the agent. This program moves the agent towards 0,0,0, with agents farthest away from 0,0,0 moving quicker than closer agents. The simulation does not run long enough for agents to actually reach 0,0,0, and the end result is that agents congregate towards a cube spanning between 0,0,0, and the middle of the room. Since in all fitness cases the exits are located roughly near the center of a wall of the cube, this outcome has a lower average distance between all agents and the exit than at the start of the simulation. Since agents don't move very far, however, few get close enough to trigger a "too-close" penalty.

Another program, `(VECTOR POP POP POP POP -0.118044495 VSCALE VMUL)`, which evolved at generation 15 of one run, has a similar fitness to the above

program (206), but a different behavior. It pops the 4 top vectors on the stack, leaving the location of the 3rd closest agent (if any) at the top. It then scales that vector by –0.11, and attempts to multiply that vector times the coordinates of the 4th closest agent (only rarely are there four nearby agents, so usually that instruction is converted into a no-op). The behavior of this program is to move towards 0,0,0, if, and only if, there are at least 3 agents near the current agent. In this way, the program promotes concentrating each agent towards the center of the room whenever that agent might be within penalty distance of another agent, but otherwise doesn't move.

The most fit program (fitness = 138), which evolved on generation 31 of one run, is much more complex: `(((AND x x) (SIZE (DO* x x (x (x x))) (x x ((x x x) ((x (x) (x -0.9737293720245361 -) x x (x (((x) MIN VSCALE) (x) x (VECTOR SUBST)) x (x (x TYPE) (x) (x) x))) (POP) (x x (x) PULL) 0.015931844) x (x)))) ((BOOLEAN) VADD x VSCALE ((DO* POP) (x x x) x x) x) x (x) x) (x) (x x (x) x) x).` Note that the symbol x represents a no-op that has no function when executed, but is structurally necessary in order for the rest of the program to work.

This program, like many of the more complex programs generated by GP systems, is hard to decipher. It has been simplified to remove all functions that do not contribute to fitness, so the presence of modularity instructions (DO*) indicate that it does use modularity to function. Furthermore, it makes heavy use of multiple data-types, switching the stack type from the default to VECTOR, BOOLEAN, and TYPE, showing that it is not just the ability to process vectors, but the ability to process multiple types of data that contributes to the fitness of this program.

Behaviorally, this program operates similarly to the manually designed agents by Crespi, et al. (2002). All agents immediately start to move towards the exit, however, whenever agents get close to each other, they instead retreat away from their neighbors. Notably, the performance of the evolved program is not as good as that designed by Crespi, since agents do at times get too close to neighbors, and not all agents are able to exit. One major deficiency is that even in a simulation of just a few agents, a lot of time is spent backtracking from potential collisions; rarely does an agent exit if it does not have a straight run to the exit. Our result, however, is the product of just six runs, and with more experimentation we can expect improvements in performance.

4 CONCLUSIONS

In this work we set out to evolve agents for the 3D Opera problem. We explored the effects of adding a vector data-type to an already rich multi-type system, and whether or not modularity was useful for the problem in general. Our preliminary results suggest that in the domain of the 3D Opera problem, and perhaps for other 3D agent problems:

- It is difficult to evolve vector manipulation code at the same time as effective vector use.
- A vector data type, when available, is highly selected for.
- Adding vectors improves performance throughout the generations of a run.
- Multiple data-types systems are useful not just because one data-type is specialized to the problem, but because many problems are naturally dealt with in terms of more than one data-type.
- PushGP leverages the ability to create modular code in many situations, including when it is just evolving constants.

There is much room for future work on this research. More runs would shed light on the extent of these effects, and allow for more detailed examinations of the similarities and differences between the programs evolved by PushGP vs. manually designed programs. It would also be interesting to investigate methods for more equally weighing the intertwined metrics that combine to define fitness. Though the fitness equation was designed to reward improving both metrics, it appears that "politeness" was usually more selected for than "exiting". Making the selective pressure more equal may be enough to evolve a fully successful individual, given how close we already came in just six runs of experiment two.

Acknowledgments

This effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30502-00-2-0611. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government. This research was also made possible by generous funding from Hampshire College to the Institute for Computational Intelligence at Hampshire College.

References

Crespi, V., G. Cybenko, and D. Rus (2001). *Decentralized Control and Agent-Based Systems in the framework of the IRVS.* White paper presented at the PI TASK Meeting held in Santa Fe, NM, on April 2001. http://actcomm.thayer.dartmouth.edu/task/crespi/irvs2.pdf

Crespi, V., G. Cybenko, D. Rus, M. Santini (2002). *Decentralized Control for Coordinated flow of Multi-Agent Systems.* Dartmouth Technical Report TR2002-414, January, 2002. Presented at the 2002 World Congress on Computational Intelligence. Honolulu, Hawaii, May 12--17, 2002.

Crawford-Marks, R., and L. Spector. 2002. Size Control via Size Fair Genetic Operators in the PushGP Genetic Programming System. In W. B. Langdon, E. Cantu-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska (editors), *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2002*, San Francisco, CA: Morgan Kaufmann Publishers

Spector, L., and A. Robinson. 2002a. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. In *Genetic Programming and Evolvable Machines*, Vol. 3, No. 1, pp. 7-40

Spector, L., and A. Robinson. 2002b. Multi-type, Self-adaptive Genetic Programming as an Agent Creation Tool. In *Proceedings of the Workshop on Evolutionary Computation for Multi-Agent Systems, ECOMAS-2002*, International Society for Genetic and Evolutionary Computation.