A tunable deceptive problem to challenge Genetic and Evolutionary Computation and other A.I.

Daniel Howard
Howard Science Ltd,
Malvern, UK
dr.daniel.howard@gmail.com

***Abstract*— A deceptive problem with known analytical solution is introduced. Arguably its solution search landscape is such that heuristic methods will find it difficult to search for the solution. The problem is tunable offering a test bed by which to examine the performance of different methods of heuristic and evolutionary search.**

***Keywords*— deceptive problem, Genetic Programming, Evolutionary Computation, A.I, solution landscape, heuristic method, tunable problem, analytical solution, toy problem.**

I. PROBLEM DESCRIPTION

A sequential computer program consisting of a set of instructions with some inter-dependencies between instructions is to be run on a parallel computer. No instruction or underlying algorithm is modified but the instructions must be distributed optimally among a potentially unlimited number of parallel processors, respecting the dependencies, such that the program is run in minimum time, essentially carries out the same computation and outputs the similar results as its sequential version.

In attempting to parallelize the code by exploiting all opportunities and respecting the dependencies, assume that all processors are of the same computational memory capacity and speed, and that all code blocks take the exact same amount of compute time on these processors.

TABLE 1 SIMPLE EXAMPLE WITH SOLUTION

Code Blocks
1.) A = 5
2.) B = 6
3.) F = B - 7
4.) B = 80
5.) D = B*B
6.) E = 2*A

Dependency Matrix

	3	**4**	**5**	**6**
1				**raw**
2	**raw**	**waw**	**rar**	
3		**war**		
4			**rar**	
5				

(1,6);(2,3);(2,4);(2,5);(3,4);(4,5)

Solution

P	**P**
1	**2**
6	**3**
	4
	5

II. ANALYTICAL SOLUTION

Table 1 presents a small but illustrative instance of this class of problem. Ignoring whether true or false it has all three types: Read After Write (raw) or flow data dependencies, e.g. code block 1.) must write the value of A before code 6.) uses it. Also, with B for code blocks 2.) and 3.); and 2.) and 5.). The Write After Write (waw) or output dependence exists as code block 2.) cannot finish after code block 4.) to change the value of B. Write After Read (war) also known as anti-dependence exists since code block 3.) must finish reading B before code block 4.) can overwrite B.

The analytical solution is obtained by considering each row of the dependency matrix in turn, as in Table 2, and assigning to processors ***P*** either sequentially or when there is no dependency for the code block then in parallel.

TABLE 2 OBTAINING THE ANALYTICAL SOLUTION

(1,6)				**(2,3) (2,4) (2,5)**				**(3,4)**				**(4,5)**			
P	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**
1				**1**	**2**			**1**	**2**			**1**	**2**		
6				**6**	**3**	**4**	**5**	**6**	**3**	**5**		**6**	**3**		
									4				**4**		
													5		

III. HEURISTIC SEARCH SOLUTION REPRESENTATION

The search for the analytical solution by heuristic means is difficult if adopting the following representation using two types of functions only: $\mathbf{P^I_J}$ which stands for "parallelize" and $\mathbf{S^I_J}$ which stands for "keep sequential". These take the original sequential code block instructions in order, manipulating them and assigning them to processors.

The subscript **J** indicates how many instructions are in function input and the superscript **I** indicates how many instructions will go one way (left) and the remainder go the other way. For example, if ten instructions should be processed then for P^3_{10} three instructions are run in parallel with the other seven, whereas if for S^3_{10} then three instructions are split sequentially to the other seven.

Consider a candidate solution in this representation, for example:

S^2_6	P^1_2	P^3_4	S^1_3	S^1_2

with a tree representation as in Figure 1. Table 3 shows how to execute it to obtain an overall time equivalent parallel computation (though involving one more processor).

TABLE 3 CODE EXECUTION WORKED OUT EXAMPLE

		Result		
Input	**Code**	For: 1,2,3,4,5,6		
6	S^2_6	1,2 3,4,5,6		
2	P^1_2	1 3.4.5.6	2	
4	P^3_4	1 3.4.5	2	6
3	S^1_3	1 3 4,5	2	6
2	S^1_2	1 3 4 5	2	6

As all processors are identical and all code blocks are assumed to take the same compute time, this parallelization takes four clock cycles. Only the first processor is fully occupied sequentially computing code blocks 1, 2, 4, 5. It is a solution equivalent to that shown in the last column of Table 1. The evaluation produces a schedule of the code blocks running sequentially or in parallel with respect to one another on different processors.

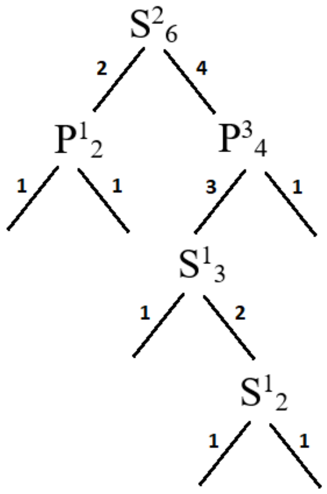

Figure 1 Tree representation of candidate solution.

Before proceeding any further, it is very important to verity that all possible block code distributions are attainable using this solution representation, i.e., through combinations of functions $\mathbf{P^I_J}$ and $\mathbf{S^I_J}$. any valid combination may be obtained. A mathematical proof is needed but for the purpose of this short paper it suffices to give a logical argument why this is so:

Operator $\mathbf{S^I_J}$: Given $\boldsymbol{n}$ input instructions, instruction [$\boldsymbol{i+1}$] depends on prior execution of instruction [$\boldsymbol{i}$] and never vice-versa. Operator $\mathbf{S^I_J}$ always acts on a list of input instructions that is ascending, so its output can never place a prior instruction after a subsequent instruction on a processor.

Operator $\mathbf{P^I_J}$: The operator splits instructions which are allegedly independent. Hence, under the assumption that the number of available processors is equal to the number of instructions, $\boldsymbol{n}$, it is immaterial whether the instruction could have been undertaken earlier on the target processor. If so, then the splitting should have taken place earlier.

For these reasons, if $\boldsymbol{n}$ processors are available then relying solely on this two-operator algebra, a version of the parallelization which is equivalent to the analytical solution or most compute-time-efficient available can always be represented.

IV. Implementation with Genetic Programming

A method such as Genetic Programming [1] is ideally suited to work with this algebra. For this presentation, the implementation used is the Attribute Grammar Genetic Programming method [2]. This method originated in [3] and was developed in [4-5]. It uses standard Genetic Programming [1] to algebraically manipulate a set of input constants. Uniquely, the evaluation of the GP tree produces as its output a variable length vector of constants.

Next, these constants are in turn consulted by a grammar to obtain the parallelization instruction. Bachus-Naur Form of this grammar can be (see [2]) as in Figure 2.

$$
\begin{aligned}
&< s > ::= < E > \\
&< E > ::= < P >< E >< E > \mid < S >< E >< E > \\
&< P > ::= P_0 | \cdots | P_i | \cdots | P_{N/2} \\
&< S > ::= S_0 | \cdots | S_i | \cdots | S_{N/2}
\end{aligned}
$$

Figure 2 Grammar applied to the output of GP, see [2].

One advantage of this scheme over others, such as Grammatical Evolution [6], is that all the tools that are available to GP including standard crossover, working memories, ADFs [7] and Subtree Encapsulation [8], can be used without modification. Therefore, all the findings of the GP method literature to date can be exploited.

The objective is not solely to distribute the code statements or blocks among the processors to minimize total elapsed time but also to not violate the dependencies as in Table 1. Once GP evaluates the individual to produce the variable vector of real numbers, once these are consulted using the grammar of Figure 2 to obtain an expression solely in terms of functions $\mathbf{P^I_J}$ and $\mathbf{S^I_J}$., once this expression is evaluated to obtain a putative parallelization strategy as in Table 3 and Figure 1, then this is examined for code dependency violations $\boldsymbol{N_V}$. A fitness of solution measure $\boldsymbol{f}$ is computed by punishing for a longer computation $\boldsymbol{N_{CC}}$ (loosely speaking the number of "clock cycles"):

$$\boldsymbol{f} = - \boldsymbol{P_w}\, \boldsymbol{N_V} - \boldsymbol{N_{CC}}.$$

The constant $\boldsymbol{P_w}$ controls the importance of one term over the other. In practice, a value of $\boldsymbol{P_w}$ = 100 appears useful.

V. Preliminary Numerical Experiments

Consider four tests in Table 4, each a version of the problem. Four test problems: input instructions are numbered sequentially and for example (20,38) denotes the dependency between code block 38 on 20.

ID	n	Dependencies
1	40	(20,38)
2	40	[(i,i+1) & (i,i+21) i=1,19] (20,21)
3	40	(1,10) (10,20) (5,38) (38,40) (1,5) (5,10)
3m	6	(1,2) (2,3) (1,3) (3,4) (2,5) (5,6)

TABLE 4 THE TEST PROBLEMS FOR THE EXPERIMENTS

Analytical solutions can be obtained easily. Dependencies complicate the parallelization. For example, the case 3m can use two or more processors in parallel provided it runs sequentially code blocks 1-4 on some processors and sequentially runs code blocks 5-6 in on some others but code block 5 running in parallel with code block 3 as it is sequential to process 2.

The experiments of Table 4 differ by the extent and nature of their instruction inter-dependency (making for innately more or less complex search spaces) but also and importantly by the number of instructions that are involved in the computations.

Problems ID=1, ID=2 and ID=3, involve $\boldsymbol{n}$ = 40 instructions. Yet for ID=1 only two instructions, $\boldsymbol{n_I}$ = 2, are in a dependency relation! Often in versions of this problem $\boldsymbol{n_I} < \boldsymbol{n}$. In such cases, two test problem experiments are possible: one involving all $\boldsymbol{n}$ instructions, and another involving only participating $\boldsymbol{n_I}$ instructions.

The former problem is presumably a bit harder to solve because the search space is complicated by the presence of more instructions that play no role, but which must be allocate to a processor and clock cycle, although these could ab-initio be assigned to any clock cycle on another processor. Instructions get in the way and $\mathbf{P}^{\mathbf{I}}_{\mathbf{J}}$ and $\mathbf{S}^{\mathbf{I}}_{\mathbf{J}}$ must work at moving them about and away from the 'critical path' in searching for the scheduling that minimizes clock cycles while avoiding violations. Such a problem, therefore, admits numerous equivalent solutions.

Trial runs reveal that the power of GP at discovering solutions is rather insensitive to the fitness penalty number $\boldsymbol{P_W}$ even for small values. Very quickly (within one or two generations depending on the population size and the nature of the constraints) compliant solutions emerge and dominate from then on. The penalty has a threshold nature, either it has the desired effect or it does not. Thus, it was considered not interesting to vary this penalty number further in experiments. The target fitness is determined easily for each of problem.

Table 5 compares experimental results for problem cases 1, 2, 3 and 3m for various population sizes and sets of parallel independent runs (PIRs). PIRs are needed to successfully explore a solution using GP [1]. A PIR stops when a GP discovered solution matches the analytical solution. It also shows what percentage of the PIRs were able to discover the analytical solution within 200,000 generates of steady-state GP (Koza [2] has always preferred use of "generational" GP but this and past work by the author of this paper, e.g. [2-4] always uses "steady-state" GP).

Experiment 3m is a minimalist version of experiment 3. It is apparent that the search space is less complicated because most of the parallel independent runs of experiment 3m attain the target fitness. The table also seems to indicate that the constraints that define problems 1 and 2 make for an easier search space than what defines problem 3.

For the problems that involve all 40 input instructions, GP produces a set of different 'fitness equivalent' solutions, i.e., different possible assignments of input instructions to processors and clock cycles at the same level of fitness.

In Table 5, various GP population sizes, e.g., 200, 500, 3000, 5000 are investigated and results must achieve the analytical solutions: $f = -2.00$ (case 1); $f = -21.00$ (case 2); $f = -4.00$ (cases 3 and 3m). PIRs are the total number of parallel independent runs in the experiment, each starting with a different random seed. The last column gives the success or percentage of runs that find the analytical solution inside of 200,000 generates of steady-state GP (to the nearest percent). Not all runs are successful making this problem a candidate test bed for the relative performance of different heuristic methods.

prob	pop size	PIRs	success	prob	pop size	PIRs	success
1	200	120	98 %	1	500	120	97 %
2	200	120	96 %	2	500	120	95 %
2	3000	120	95 %				
3	200	120	72 %	3	500	120	75 %
3	1000	120	70 %	3	5000	150	69 %
3m	500	150	99 %	3m	1000	150	97 %
3m	3000	150	97 %	3m	5000	150	96 %

TABLE 5 EXEPERIMENTS: NOT ALL PIR SOLVE THE PROBLEM.

VI. CONCLUSIONS

A heuristic search problem with known analytical solution is introduced in this paper. Heuristic methods found it difficult to discover solutions as evidenced in Table 5. The problem requires more analysis. It is tunable thus offering a test by which to examine the performance of different methods of Artificial Intelligence, heuristic solution search, Genetic and Evolutionary Computation.

REFERENCES

[1] Koza J. (1992), Genetic Programming: On the Programming of Computers by Means of Natural Selection: v. 1 (Complex Adaptive Systems), MIT Press, Cambridge.

[2] Howard D., Ryan C., Collins J.J. (2011) Attribute Grammar Genetic Programming Algorithm for Automatic Code Parallelization. In: Lee G., Howard D., Ślęzak D. (eds) Convergence and Hybrid Information Technology. ICHIT 2011. Lecture Notes in Computer Science, vol 6935. Springer, Berlin, Heidelberg.

[3] Howard D., Roberts S.C. (2001), Genetic Programming solution of the convection-diffusion equation. In: Spector L. et al (eds), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001), pp. 24-41. Morgan Kauffman, San Francisco.

[4] Howard D. (2009), Bio-inspired simulation tool for PERT, Proceedings of the 2009 International Conference on Hybrid Information Technology. Pages 537-540, Daejeon, Korea — August 27 - 29, 2009, ISBN: 978-1-60558-662-5.

[5] Baber C., Stanton N., Howard D., Houghton R.J. (2009): paper 15 – Predicting the structure of covert networks using Genetic Programmingm, Cognitive Work Analysis and Social Network Analysis. Paper presented at the NATO RTO Modelling and Simulation Work Group Symposium, Brussels, Oct 15-16, 2009. ISBN 978-92-837-0200-2.

[6] Ryan C., Collins J. J. and O'Neill M., Grammatical Evolution: Evolving Programs for an Arbitrary Language. In Wolfgang Banzhaf and Riccardo Poli and Marc Schoenauer and Terence C. Fogarty editors, Proceedings of the First European Workshop on Genetic Programming, LNCS 1391, 83–95, Paris, (1998).

[7] Koza J. (1994), "Genetic Programming II: Automatic Discovery of Reusable Subprograms", MIT Press.

[8] Roberts S.C., Howard D., Koza J.R. (2001), "Evolving modules in Genetic Programming by subtree encapsulation". In Julian F. Miller and Marco Tomassini and Pier Luca Lanzi and Conor Ryan and Andrea G. B. Tettamanzi and William B. Langdon editors, Genetic Programming, Proceedings of EuroGP'2001, volume 2038, pages 160-175, Lake Como, Italy, 2001. Springer-Verlag.