Program Optimisation with Dependency Injection

James McDermott and Paula Carroll

Management Information Systems, Quinn School of Business,
University College Dublin, Ireland. jmmcd@jmmcd.net

Abstract. For many real-world problems, there exist non-deterministic heuristics which generate valid but possibly sub-optimal solutions. The *program optimisation with dependency injection* method, introduced here, allows such a heuristic to be placed under evolutionary control, allowing search for the optimum. Essentially, the heuristic is "fooled" into using a genome, supplied by a genetic algorithm, in place of the output of its random number generator. The method is demonstrated with generative heuristics in the domains of 3D design and communications network design. It is also used in novel approaches to genetic programming.

1 Introduction

In software engineering, unit testing is the practice of providing known-good input-output pairs as tests for individual functions. When a function is non-deterministic, each input does not correspond to a single output, so unit testing cannot be used. The *dependency injection* design pattern provides a solution: the random number generator used by the function is shadowed by a source of non-random numbers, so each input again corresponds to one correct output.

This inspires the idea of searching among the outputs of any non-deterministic program (NDP) by searching among the sequences of values returned by the non-random number generator (NRNG). A genetic algorithm (GA) can be used to supply genomes, and each genome then used as an NRNG, as shown in Fig. 1. This is *program optimisation with dependency injection* (PODI).

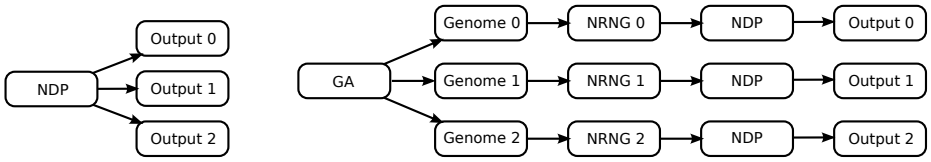

Fig. 1. A non-deterministic program (NDP) can produce multiple outputs (left). In PODI (right), a GA supplies genomes, each giving rise to a non-random number generator (NRNG): given an NRNG instead of an RNG, the NDP produces a *single* output. The GA searches for genomes leading to good outputs.

For the remainder of this paper, the goals are to define the PODI idea and to demonstrate that it can be used for search and optimisation in diverse domains. Sect. 2, next, describes some related work. The PODI method is defined in Sect. 3. Several examples of applying PODI in diverse domains are described in Sect. 4; Sect. 5 analyses the PODI search space; and Sect. 6 gives conclusions.

2 Related Work

Many authors have pursued genetic programming (GP) representations other than the standard tree-based GP of Koza [1]. Examples include linear GP (LGP) [2], Cartesian GP (CGP) [3], and grammatical evolution (GE) [4]. Each uses a linear genome and a specialised genome-program mapping to produce a program. In PODI, the same idea is used, but any non-deterministic program (NDP) whose output is a program can play the role of the genome-program mapping. PODI can also be used in non-GP domains, using an NDP whose outputs are the objects of the domain's search space.

Another strand of research closely related to PODI is the "programming by optimisation" method [5]. A template program is written by hand, in which some parts are concrete and fixed, and for other parts, alternative algorithms and implementations are provided. Optimisation proceeds by searching among the possible fully concrete programs. The aim is to allow the human programmer to work in the familiar creative domain (writing programs by hand), while the computer carries out the tedious and difficult job of determining which alternatives are faster or more correct in a given context. PODI has a similar goal, in that it allows the programmer to work by programming. However, it does not require a specialised template language, and the search mechanism is different.

3 Method

The PODI representation consists of two parts: a GA to be described below, and a pre-written non-deterministic program (NDP).

Running the NDP produces some output to be evaluated as a candidate solution. Depending on the domain, it could be a string, a vector, a program, or any data structure. Running the NDP multiple times will produce multiple candidate solutions: they will differ because the NDP is non-deterministic. By *injecting* (passing in) an extra argument which will play the role of the random number generator, the NDP can be made deterministic. This new argument is an instance of a non-random number generator (NRNG)[1].

The NRNG has the same interface as a standard random number generator (RNG): `randint(a, b)` returns an integer in $[a, b)$; `random()` returns a floating-point number in $[0, 1)$; `choice(L)` returns an element of the list `L`; and so on. The NDP can therefore run without modification. However, the values returned

[1] An example of dependency injection is given by Peter Norvig: `http://www.udacity.com/view#Course/cs212/CourseRev/apr2012/Unit/292001/Nugget/315001`

by the NRNG are not random, nor even pseudo-random. Instead, they are determined by a stored list of integers which come from a genome[2]. The NDP can therefore be regarded as a mapping from genomes to outputs (such as vectors, programs, or other data structures).

PODI searches using a GA where genomes are variable-length integer arrays. Mutation is per-gene int-flip, and crossover selects one crossover point per parent within the section which was used when the parent was passed to the NDP (compare the *used codons* concept in GE). However, *PODI is not just a GA*. It uses a potentially complex genome-phenotype mapping, the NDP. An NDP can use loops and conditionals to divert control flow, and can call its NRNG any number of times. Therefore there is no direct and fixed correspondence between the genome *loci* and the sequence of NRNG calls. This is the key distinction between PODI and a typical GA. PODI can also do true GP in at least three ways (one is demonstrated in Sect. 4.1, and two in Sect. 4.2).

It might appear that in PODI, the genome's control of the NDP's behaviour is no different from the control that would be exerted by a random seed for a typical RNG. After all, in both cases the input (the genome, or the seed) fully determines the behaviour of the NDP. However, PODI is not random search. The difference is in the fact that *the space of PODI genomes has structure*, whereas the space of random seeds does not. That is, small mutations to a random seed will not result in small changes to the NDP's behaviour, and crossover between a pair of random seeds will not lead to behaviour intermediate between the two. The opposite is true of the PODI operators. This argument is developed with experimental support in Sect. 5.

4 Examples

4.1 Emulating Grammatical Evolution

The PODI framework can be seen as a generalisation of GE. To see this, consider Algorithm 1, a standard method (in non-EC contexts) of deriving a string from a non-deterministic grammar. It is non-deterministic, thus stretching the strict definition of "algorithm". Therefore, it can serve as an NDP in PODI. In fact, PODI with Algorithm 1 mimics the behaviour of GE with the "bucket

[2] It is useful to explain a little of the workings of the NRNG. In an object-oriented language such as Python or Java, the NRNG is a subclass of the language's standard RNG. The NRNG constructor accepts an extra argument, a list of integers, which is stored. Only one method needs to be overridden: in Python, it is the RNG's `random()` method, which is required to return a floating-point value in $[0, 1)$. At every call to `random()`, the next of the stored integers is used. It is divided by the constant `maxval`. This constant usually stores the largest representable integer, such as $2^{31} - 1$. However if the user wishes to use a GA with gene values limited to, say, $[0, 100]$, then `maxval` will be set to 100. The result of the division is in $[0, 1)$, as required, and is returned. Methods other than `random`, such as `randint()` and `choice()`, call `random()`, possibly multiple times, and use its output as needed.

Algorithm 1 *Derive*: derive a string from a grammar.

Require: Grammar $G = (R, S, T)$ where R is a mapping from non-terminal symbols to lists of productions, S is a non-terminal start symbol, and T is the terminal set;
Require: Stateful random number generator RNG.
1: **if** $S \in T$ **then**
2: return S
3: **end if**
4: $P \leftarrow R[S]$ {# P is a list of items producible from S}
5: **if** length$(P) = 1$ **then**
6: $p \leftarrow P[0]$
7: **else**
8: $p \leftarrow$RNG.choice(P)
9: **end if**
10: return Concatenate([*Derive*$((R, S', T)$, RNG) for S' in p]) {# The square brackets indicate a *comprehension*: it makes a new list by recursing on each element.}

rule" [6]. Replacing line 8 with the following mimics GE's standard mod rule instead:

8: $p \leftarrow P[\text{RNG.randint}(0, C) \ \% \ \text{length}(P)]$

where C is the maximum value for GE codons, often set to 127 in previous work.

Note that PODI *genes* are in [0, `maxval`), whereas the *codons* returned by the `randint()` call are in $[0, C)$. Therefore, it is not possible to reproduce a particular GE run precisely in PODI by setting identical random seeds. Instead, we wish to demonstrate experimentally that PODI, with the modified version of Algorithm 1 as the NDP, emulates standard GE behaviour.

For standard GE we use an existing implementation[3]. In both standard GE and PODI-GE we do not use any special initialisation such as ramped half-and-half: instead, we randomly initialise the integer-valued genomes. The initial length is 100 genes. The maximum codon value is 127. Population size is 1000, the number of generations is 40, the one-point crossover probability is 0.7 (one crossover point is chosen in *each* parent, within the portion of its genome used when it was mapped), the per-gene mutation probability is 0.01, the number of elite individuals is 1, and the tournament size is 3. The fitness function is the two-dimensional symbolic regression problem

$$f(x, y) = \frac{1}{1 + x^{-4}} + \frac{1}{1 + y^{-4}}$$

used by Pagie and Hogeweg [7]. There are 676 fitness cases distributed in an even grid across $[-5, 5] \times [-5, 5]$.

[3] http://ponyge.googlecode.com

The grammar is as follows (in division, if the denominator is zero, the numerator is returned):

```
<code> ::= function(x, y) <expr>
<expr> ::= <bop>(<expr>, <expr>) | <uop>(<expr>) | <var> | <const>
<bop> ::= add | subtract | multiply | divide
<uop> ::= square | sin | cos
<var> ::= x | y
<const> ::= 0.1 | 0.2 | 0.3 | 0.4 | 0.5
```

The results are shown in Fig. 2: there is a close correspondence in behaviour between the two methods, both in best fitness values and the number of codons used in the best individuals. There are some small differences in overall behaviour, such as PODI-GE spending a few more generations with a wider standard deviation in the number of used codons (generations 31–35 versus generation 33). Re-running the experiment has not demonstrated a trend in this. We conclude that PODI-GE emulates standard GE behaviour.

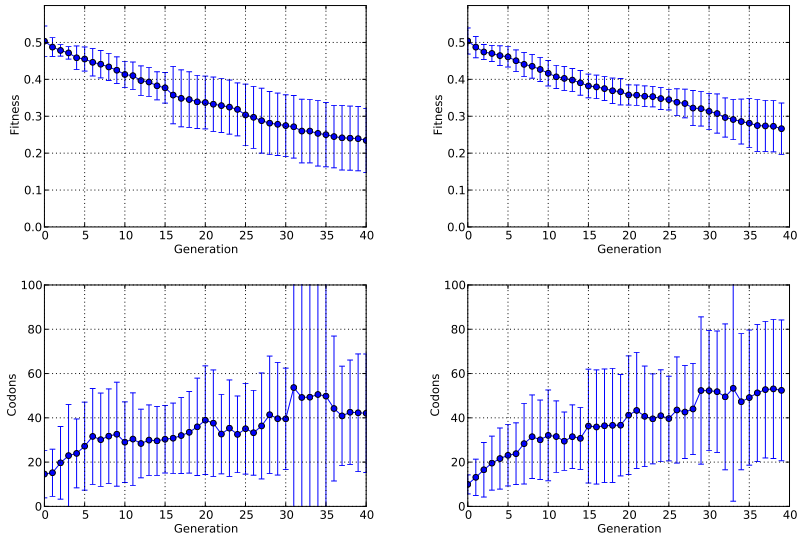

Fig. 2. Mean and standard deviation of best fitness (top) and number of used codons (bottom) across 30 runs. The behaviour of PODI-GE (left) closely matches that of standard GE (right).

4.2 Tree-based Genetic Programming

PODI can also be used for non-grammatical, tree-based forms of GP. It is sufficient to define an NDP which creates GP-style trees. Many GP initialisation operators are suitable, and we have investigated two.

One is the *grow* operator, which recursively chooses nodes randomly from non-terminals and terminals until the depth reaches a limit, then chooses only from terminals. It is well-known, so to save space we do not describe it in detail.

The other is the *bubble-down* operator [8], which (similar to *grow*) creates a tree one node at a time. A new node is created with a randomly-chosen non-terminal label. It is "bubbled down" through the tree according to randomly-generated *direction* and *slots* values which are associated with the existing and new nodes (see Fig. 3) until it reaches an external position. As yet it lacks children. When the number of nodes plus the number of missing children equals the desired tree size, new nodes are given randomly-chosen *terminal* labels and put in place at external positions in left-to-right order. The direction and slots values, used only during tree generation, are then deleted. A complete implementation of the bubble-down algorithm is available online[4].

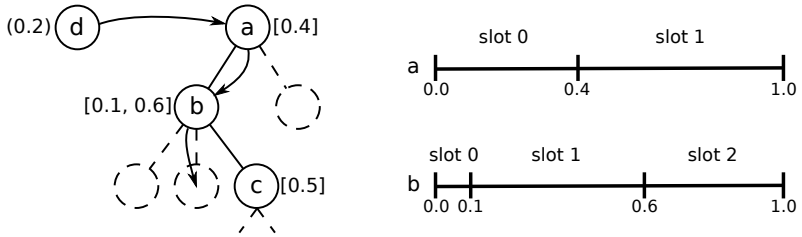

Fig. 3. The bubble-down algorithm. The partially-formed tree so far consists of nodes labelled **a**, **b**, and **c**, with arities 2, 3 and 2. They have randomly-chosen *slots* values, shown in square brackets. A new node has now been created with label **d**, chosen randomly from the non-terminal labels. It has a randomly-chosen *direction* value 0.2 (shown in round brackets). Its slots values are not shown. It must now be bubbled down, and begins at node **a** (top arrow). Since 0.2 lies in slot 0 of node **a** (see right), **d** is bubbled-down to child 0, i.e. it reaches node **b** (middle arrow). The procedure repeats. Node **b** has arity 3, hence 2 slots values, sorted. Since 0.2 lies in slot 1 of node **b** (see right), **d** moves to child 1 (bottom arrow). Since **d** is now external, it stops.

We wish to test PODI where the role of the NDP is played by the grow and the bubble-down algorithms. The test problem is the same symbolic regression problem as in Sect. 4.1, and the settings are the same. The non-terminals and terminals are as in the grammar of that section. For grow, the maximum depth is 6; for bubble-down, the tree size is fixed at 30 nodes. The results are shown in Fig. 4: the mean best fitness for both grow and bubble-down is better than the results shown in Sect. 4.1 for standard GE (t-test with $p < 0.01$).

[4] Download PODI implemented in Python from `https://github.com/jmmcd/PODI`

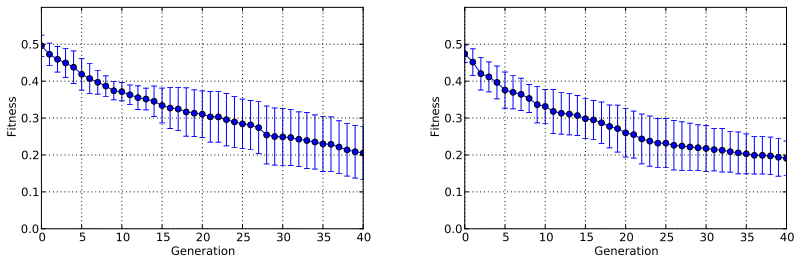

Fig. 4. Results using tree-based GP: mean and standard deviation of the best individual per generation over 30 runs using PODI with *grow* (left), and *bubble-down* (right).

4.3 Communications Networks

The *ring-spur assignment problem* (RSAP) is a problem in the field of next-generation telecommunications networks. Since this is not an applications paper, only a brief description will be given. Starting with an existing physical network, the goal is to form a logical overlay network with a resilient structure, of the lowest possible cost. It must consist of a subset of the physical edges, and conform to a "ring-spur" structure (see Fig. 5). The ring-spur structure means that there is one "tertiary" ring; multiple disjoint "local" rings of up to 8 nodes which each intersect with the tertiary ring; and possibly several "spurs", each a single edge connecting one isolated node to a local ring. The cost is the sum of costs of the edges used, with a penalty per spur.

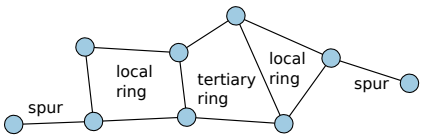

Fig. 5. A valid ring-spur assignment solution: all nodes are included, and the edges form a single tertiary ring intersecting multiple local rings, with some (optional) spurs.

The RSAP was described by Carroll and McGarraghy [9], and integer programming (IP) formulations were given. However for some test problems, the IP method is excessively slow. Kilmartin and Flynn [10] used a variable neighbourhood search method for the problem. They give a non-deterministic initialisation heuristic which attempts to generate valid solutions, and several neighbourhood operators: each generates a new solution, given an existing one.

The initialisation heuristic consists of over 600 lines of Java and works as follows (see [10] for details sufficient for replication). A random node is chosen, and a greedy algorithm over edge costs is used to create a local ring. If it succeeds, the nodes used are removed from the node list. If it fails, the node is removed from the node list and added to a list of spur candidates. The process is repeated: eventually the node list becomes empty. For each spur candidate, a

spur is attempted to some local ring. Finally, a tertiary ring is created starting at a random local ring, and again running a greedy algorithm. In the following experiment, this heuristic is used as the NDP of the PODI method.

For comparison, we also implement a naive GA approach. Here, genomes are of length $3m$ where m is the number of physical edges. Each of the first m genes is a bit and determines whether the corresponding edge is used on the tertiary ring. Each of the next m genes determines whether the corresponding edge is used as a spur. Each of the final m genes is an integer and indicates the index of the local ring on which the corresponding edge occurs, or 0 if unused. In this encoding, a large majority of genomes correspond to invalid networks.

Table 1. Results on the RSAP problem. Lower costs are better.

Problem	IP	PODI	GA	Problem	IP	PODI	GA
pdh	1.36e+06	1.36e+06	3.14e+06	dfn-bwin	105810	113818	195473
pioro40	9586 *	9749	None	di-yuan	412300	412300	None
janos-us	16672	23541	None	ta1	1.14e+07	1.29e+07	None
newyork	1.51e+06	1.77e+06	None	sun	694.99	1277.82	None
cost266	1.236e+07	1.65e+07	None	polska	3487	3740	None
atlanta	5.55e+07	1.042e+08	None	giul39	946 *	1021	None
france	20800	26000	None	dfn-gwin	15724	17428	79868
germany50	549150 *	574460	None	norway	596070	808290	None
nobel-eu	None	None	None	janos-us-ca	None	None	None
zib54	None	None	None	ta2	7.38e+07 *	None	None

Table 1 shows results achieved on the problem instances used in previous work [9, 10]. The IP results are known to be optimal [9] except where marked *. For PODI and the GA, these are the best (lowest cost) results out of 30 runs, with population 1000 and 100 generations, tournament size 10, crossover probability 0.9, mutation probability 0.05, and elitism of 10 individuals.

For some problems, IP methods show that no valid solution exists (those where IP achieves "None" in Table 1). In a few cases PODI achieves the known optimum. The GA approach is the weakest, usually finding no solution. We conclude that PODI is inferior to the IP method, but because of the use of a domain-specific NDP, it is a better starting point than a naive GA. An applications-oriented paper with full details of this experiment is in preparation.

4.4 Generative 3D Design

The design of 3D structures is an interesting application domain for *interactive* EC. GP has been applied by several authors for this application, e.g. [11]. One line of research used GE to explore spaces of 3D designs such as bridges and pylons consisting of uniform variable-length beams [12, 13].

Here, the goal is to explore a space of radially symmetric designs of uniform variable-length beams. An incremental process of hacking and testing led

to the creation of an NDP capable of generating many of the desired designs. It is roughly 170 lines long, badly written and difficult to understand, so is not reproduced here in either code or algorithm form[5]. It seems impossible to produce an equivalent parametric model, grammar, or any other explicit representation. Instead, we see this NDP as a black box, *implicitly* defining a design space amenable to PODI search. Fig. 6 shows a selection of evolved images and a population after 6 generations. No claim is made that the results are optimal in any sense: the evolution was driven by interactive aesthetic selection. Instead, the claim is that the use of the NDP has resulted in a rich and varied search space (Fig. 6 top). Large numbers of undesired (e.g. non-radially symmetric) designs, which would be present in typical naive encodings for this problem, have been eliminated. This greatly improves the interactive evolutionary search. Also, the PODI operators have been shown to have the capacity to converge the population (Fig. 6 bottom).

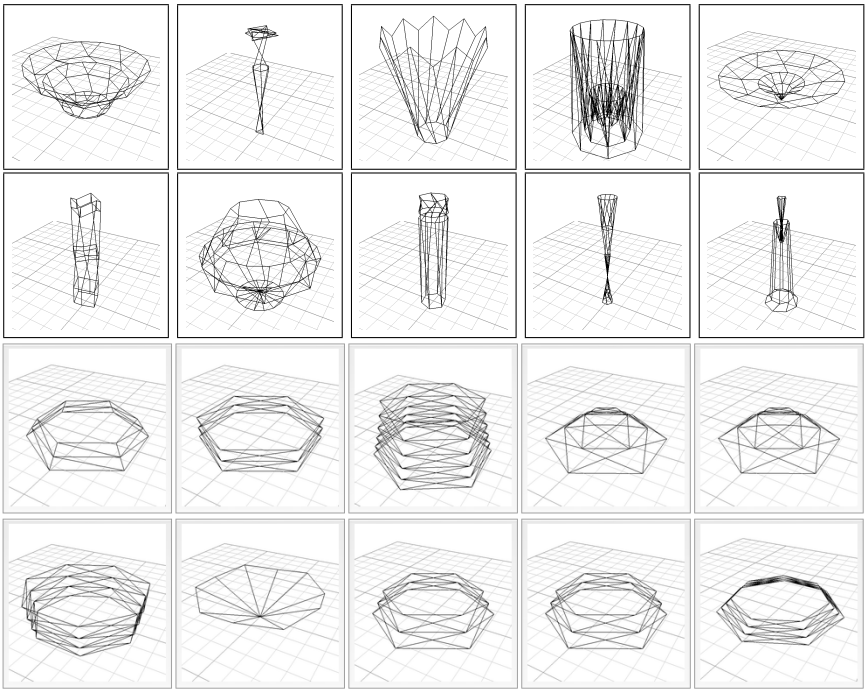

Fig. 6. A selection of evolved results (top 2 rows), demonstrating a wide and varied search space. A single population (bottom 2 rows), demonstrating some convergence.

[5] Download from `https://gist.github.com/4055990`

5 Analysis: the Structure of the PODI Search Space

We have asserted that the PODI search space of integer arrays has structure, and in particular that mutation and crossover are useful operators. In experimental settings, mutation is generally considered useful if it has *locality*, i.e. the semantic distance between an individual and a mutated version is small [14]; crossover is considered useful if it is *geometric*, i.e. the semantic distance between parents and children is minimised [15]. These properties have been formally defined by Moraglio et al. [16].

The partially converged population of Fig. 6 gives some visual support for the claim that the PODI operators are useful. In order to test the claim experimentally, we will perform many mutations and crossovers on randomly-generated individuals and study the distances between the original and new individuals. Our mutation hypothesis is that the expected semantic distance between an individual and a mutated version is less than that between two randomly-generated individuals. Our crossover hypothesis is that the expected semantic distance between a parent and a child is less than that between two randomly-generated individuals. These relatively weak hypotheses, if proven, are sufficient to show that search with PODI is distinct from random search.

This study is restricted to the GP domain as in Sect. 4.2 (a comparison with GE behaviour is left to future work). For each tree-generating NDP (grow and bubble-down), three methods of creating pairs of individuals are used: random generation of the pair; random generation of the first, and mutation to create the second; and random generation of two parents, crossover to create two children, and then taking all four parent-child pairs. For this experiment one-point mutation is used, rather than a per-gene mutation probability. That is, a mutation changes precisely one gene (as in crossover, the mutation is constrained to occur within the used section of the genome).

Four types of distance between pairs are calculated:

- genome distance is the Levenshtein (edit) distance between genomes;
- phenotype distance is the tree-edit distance between generated trees;
- semantic distance is the Euclidean distance between the vectors of the individuals' values at the fitness cases;
- fitness distance is the difference between the fitness values.

10,000 trials are run for each NDP and each method of creating pairs. Trials where an individual is invalid or a change is genotypically or phenotypically neutral are discarded. Table 2 shows the *median* results (not the mean, because semantics and fitness results are liable to include very large outliers). The low values including zero for the median are not unexpected, since a genotypically or phenotypically non-neutral change can still be neutral in semantics or fitness. In all cases, the mutation and crossover pairs are closer together than the randomly generated pairs. Significance is tested using the Mann-Whitney U test with $p < 0.01$. We conclude that search with PODI is not random search.

Table 2. Median distances between tree-based GP individuals generated using the grow algorithm and the bubble-down algorithm: in all cases, the pairs created by mutation or crossover are closer together than the random pairs. Statistical significance at $p < 0.01$ is indicated by * (see text).

NDP	Distance	Random	Mutation		Crossover	
Grow	Genotype	1e+02	1	*	59	*
Grow	Phenotype	45	5	*	36	*
Grow	Semantic	26	1	*	23	*
Grow	Fitness	0.9	0.011	*	0.62	*
Bubble-down	Genotype	1e+02	1	*	59	*
Bubble-down	Phenotype	72	6	*	61	*
Bubble-down	Semantic	26	0	*	24	*
Bubble-down	Fitness	0.79	0	*	0.59	*

6 Conclusions and Future Work

A highly general evolutionary algorithm, *program optimisation with dependency injection*, has been introduced and studied. It has been shown to function as a generalisation of GE, and to be capable of performing tree-based GP in two ways. It has been applied to problems in 3D design and in communications network design, and the structure of its search space has been analysed.

One of the main advantages of EC in general is that it is a useful "black-box" method: it makes many domains amenable to search and optimisation which are inaccessible to some more specialised methods. In a similar way, PODI opens up a broader class of problems to evolutionary approaches. It is not limited to domains where we can define all three typical EC genetic operators (initialisation, crossover and mutation). With PODI, only an NDP is required. Since an NDP is non-deterministic and outputs an object of the solution space without requiring any "parents", it is functionally similar to an *initialisation* operator.

One of the main *dis*advantages of EC in general is that more specialised methods can often out-perform it where they are applicable. An example is gradient-descent methods, which generally out-perform EC when an explicit gradient can be defined on a convex fitness landscape. Similarly, PODI will often be out-performed by specialised methods in particular domains, as we have seen in the RSAP. However, when faced with a new problem or a variant of an existing one such that no existing heuristics have been developed and shown to solve it well, PODI provides a potentially useful starting-point.

PODI is a generative method, in that the NDP is a complex functional mapping from genome to phenotype. The NDP can range from very simple algorithms, such as grow, to complex ones such as those used here for 3D design and communications network design. Both newly-written and third-party programs have been used as NDPs. The NDP can be used as a medium for the expression of domain-specific knowledge. It has the advantage in this regard that directly writing programs (as opposed to creating representations) is a natural way of working for many domain experts.

The results when running PODI on a symbolic regression problem using grow and bubble-down as NDPs are surprisingly good. The latter also offers a potential advantage related to avoiding bloat: tree size can be placed under direct evolutionary control. This idea will be explored in future work.

Acknowledgements

Thanks are due to Erik Hemberg and Edgar Galván-Lopéz for reading drafts, and to anonymous reviewers, and to the authors of [10] for publishing their code.

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge, Massachusetts (1992)
2. Brameier, M., Banzhaf, W.: Linear genetic programming. Springer (2006)
3. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: EuroGP, Springer (2000) 121–132
4. O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Kluwer Academic Publishers (2003)
5. Hoos, H.H.: Programming by optimisation. Technical Report TR-2010, Department of Computer Science, University of British Columbia (2010)
6. Keijzer, M., O'Neill, M., Ryan, C., Cattolico, M.: Grammatical evolution rules: The mod and the bucket rule. In: Proceedings of EuroGP, Springer (2002) 123–130
7. Pagie, L., Hogeweg, P.: Evolutionary Consequences of Coevolving Targets. Evolutionary Computation **5** (1997) 401–418
8. Hemberg, E., Veeramachaneni, K., McDermott, J., Berzan, C., O'Reilly, U.M.: An investigation of local patterns for estimation of distribution genetic programming. In: Proc. GECCO, ACM (2012)
9. Carroll, P., McGarraghy, S.: A decomposition algorithm for the ring spur assignment problem. International Transactions in Operational Research (2012)
10. Kilmartin, P., Flynn, M.: Quantum annealing in management science & analytics: An investigation of applying QA techniques to the ring spur assignment problem. Master's thesis, University College Dublin Business School (2012)
11. Hornby, G.S., Pollack, J.B.: The advantages of generative grammatical encodings for physical design. In: Proc. CEC, IEEE (2001) 600–607
12. Byrne, J., Fenton, M., Hemberg, E., McDermott, J., O'Neill, M., Shotton, E., McNally, C.: Combining structural analysis and multi-objective criteria for evolutionary architectural design. In: EvoWorkshops, Torino, Italy, Springer (2011)
13. McDermott, J., Byrne, J., Swafford, J.M., Hemberg, M., McNally, C., Shotton, E., Hemberg, E., Fenton, M., O'Neill, M.: String-rewriting grammars for evolutionary architectural design. Environment and Planning B: Planning and Design **39**(4) (2012) 713–731
14. Rothlauf, F.: Representations for Genetic and Evolutionary Algorithms. 2nd edn. Physica-Verlag (2006)
15. Krawiec, K., Lichocki, P.: Approximating geometric crossover in semantic space. In: Proc. GECCO, New York, NY, USA, ACM (2009) 987–994
16. Moraglio, A., Krawiec, K., Johnson, C.: Geometric semantic genetic programming. In: Proc. PPSN XII, Springer (2012) 21–31