Program Trace Optimization

Alberto Moraglio[1] and James McDermott[2]

[1] University of Exeter, UK A.Moraglio@exeter.ac.uk
[2] University College Dublin, Ireland James.McDermott2@ucd.ie

Abstract. We introduce Program Trace Optimization (PTO), a system for 'universal heuristic optimization made easy'. This is achieved by strictly separating the problem from the search algorithm. New problem definitions and new generic search algorithms can be added to PTO easily and independently, and any algorithm can be used on any problem. PTO automatically extracts knowledge from the problem specification and designs search operators for the problem. The operators designed by PTO for standard representations coincide with existing ones, but PTO automatically designs operators for arbitrary representations.

Keywords: Universal optimisation, operator design, genotype-phenotype mappings

1 Introduction

In the 1960s and onwards, researchers in genetic algorithms proposed a vision of them as 'universal solvers', capable of addressing any search and optimization problem. Later, researchers found that this promise could not be delivered because each new problem required a significant investment of time and expertise in tailoring the algorithm to the problem for acceptable performance. This is common to all metaheuristics: solvers succeed only when using an encoding and search operators which are well-chosen for the problem at hand. This is hard work, to be done per problem, and a black art, which requires (often unwritten) expertise in both the problem and the solver.

The PTO vision is to make universal optimisation easy. This is achieved by (i) neatly separating problem specification and solver definition, and (ii) automatically tailoring the solver to the problem by analysing and using the structure of the problem specification. In PTO, the core task for the user is to write a *generator*, which implicitly defines the set of possible solutions, rather than design a *representation and search operators* as in most EAs. PTO will then induce these automatically. For some problems, a good representation and search operators are easy to create and PTO's will be equivalent. In others, a generator is easier to create: it is a more concrete task, perhaps more aligned with the thinking of domain experts, as opposed to metaheuristics experts. In yet other cases, a generator may be an easy way to avoid constraint violations or express search bias difficult to express through operators (e.g. the Heuristic TSP generator of Section 4). PTO's automatic design of search operators deals seamlessly with arbitrarily complex representations. The PTO user does not have to think about

how to optimise, while the researcher working on PTO solvers does not have to think about specific problems.

2 Overview of PTO

PTO has two key ingredients: (i) a *universal solution representation* – the program trace – that decouples problem and solver; (ii) a *naming scheme* on the trace reflecting the problem structure that automatically adapts generic search operators to the problem at hand. In the following, we briefly describe these.

2.1 Universal solution representation

Any search method requires an implicit or explicit representation for generation and manipulation of solutions. PTO uses a universal solution representation that applies to any problem. Metaheuristics defined on such a representation can be applied *unchanged* to any problem, thus becoming universal optimisers. The representation used by PTO is as follows. The user supplies a solution generator, which implicitly defines the set of possible solutions. PTO runs it and traces its execution, resulting in a sequential data structure called a program trace. The trace can be thought of as the sequence of outcomes of (random) decisions made by the generator in producing a particular solution. The trace can be manipulated: it can be 'played back' in the generator to redo the same sequence of decisions and produce the same solution; it can be edited and played back to produce a variant solution; two parent traces can be combined and the result played back to produce a child solution. That is, the trace is a genotype, the solution is the corresponding phenotype, and the playback mechanism in the generator is a developmental mapping; editing and recombination of traces are search operators. The trace is a universal representation that applies to any problem because it is implicit in the problem definition (in the generator) and can be extracted automatically by tracing. No other representation can be more general or more powerful, since the generator can use Turing-complete code.

2.2 Naming scheme

The *trace* is a linear structure for any problem; the corresponding *solution* can be any arbitrarily complex structure as the generator can use any construct of the programming language, e.g. data structures, function calls and recursion, and can return any data structure. The linear genotype allows for easy adaptation of existing metaheuristics such as genetic algorithms or particle swarm optimization to work in PTO, while indirectly searching spaces of arbitrarily complex phenotypes. PTO annotates the trace so that it captures the problem structure implicitly expressed by the user in the generator. The key observation is that the control structures used in the code of the generator reflect the structure of the solutions generated, e.g., a solution with a matrix structure may be reflected in the use of nested loops in the generator; a solution with a tree structure may

be reflected in the use of multiple recursion in the generator. To make this information about solution structure available to the search operators acting on the linear genotype, the trace is annotated with the 'execution context' in which each random decision (a trace entry) took place. This execution context contains information about the state of the process (a picture in time of the running program), e.g., including the line of code from which the random number generator was invoked, the values held in the loop indices, and the current stack of (possibly recursive) calls. Search operators for annotated traces preserve the execution context as much as possible, to prevent the equivalent of the disruptive "ripple effect" [10] of Grammatical Evolution (GE). Generic search operators on the trace representation induce domain-specific search operators on the phenotype which are equivalent to known-good operators for standard representations. This mechanism of automatic implicit adaptation of the search operators to the structure of the problem at hand naturally extends to arbitrarily complex solution structures, as any solution structure can be described by a generator using a few fundamental control structures (i.e., sequence, condition, iteration, subroutine), which are handled by the annotation scheme.

2.3 PTO software architecture

PTO has a modular design in three parts. The tracer hooks into the user-supplied generator, and records and plays back program traces. The solver is pluggable: any metaheuristic solver can be plugged-in as the solver. The problem is also pluggable: the user supplies an objective function and a generator. Advances in any component do not require changes in the others.

User interface: PTO automatically makes all design decisions and parameter configurations. This makes the user interface unobtrusive and the learning curve flat, and allows for reuse of existing code, e.g. EA initialisation methods or constructive heuristics as generators (as in PODI [6]).

Tracer: The tracer interfaces to the problem definition by tracing the generator when it is run, and playing back modified traces in the generator. This is achieved by overriding the calls to any function of the Python standard random library used by the generator, so that tracing and playing back are invisible to the user. The tracer annotates the trace entry by dynamic analysis of the running code of the generator as described in Section 3.1. The annotated trace is the interface between the tracer and the solvers. Search operators work on annotated traces but are only allowed to use the labels to align entries with the same label in different traces, and cannot use any other information stored in the labels. This decouples tracer and solvers, allowing the naming scheme (the information stored in the labels) to be modified without the need to alter solvers.

Search algorithm: The requirement for a meta-heuristic to work with PTO is that it can work on the trace representation. This is a *dictionary*, i.e. a variable-length linear structure with entries identified by names rather than by indexes, and with heterogenous entries (real, binary, integer, and permutations – the output domains of random number generators in the Python library). Generalising meta-heuristics to work on the trace representation can be done in a principled

way using the geometric framework [7] so as to retain the original dynamics of the meta-heuristic on the new representation. The meta-heuristics currently in PTO include Genetic Algorithm, Stochastic Hill-Climbers, Geometric Particle Swarm Optimisation [8] and Random Search.

2.4 Related work

PTO builds on and combines previous ideas. The idea of using a sequence of decisions as a genotype was originally introduced in the Program Optimisation with Dependency Injection (PODI) system [6], with emphasis on using it with complex and smart generators, to evolve complex constrained structures. PODI is similar to the *decision chain encoding* [5]. Constructive heuristics and simulated annealing have also been hybridized for vehicle routing problems [1]. PODI can be thought of as a generalisation of GE [9] that, instead of using a grammar to map linear genotypes to complex phenotypes, uses an unrestricted program i.e., the generator, which is much more expressive. GE and by extension PODI can suffer from low locality [11] and the ripple effect [10] as a result of the mapping.

PTO goes beyond PODI and the other approaches by seeing the trace as a universal representation, allowing generic meta-heuristics to be used on any problem. The PTO philosophy of separating problem specification and solver mirrors that of Probabilistic Programming (PP) [4]. Connections between PTO and PP run deep. The PTO trace representation and naming scheme are the translation to an optimisation context of a successful PP approach [13] to inference over complex probabilistic models. Links between PP and EC have previously been made, including optimisation on a type of program trace [2]. The idea of automatic implicit design – that generic search operators on the trace representation induce domain-specific search operators at the "phenotype" level – is novel to PTO. This will be described in some detail in Section 3.

PTO differs from other works. There are approaches for the automatic design of search operators for specific problems based on hyper-heuristics e.g., [3]. PTO automatic design is different from these approaches, as it does not rely on searching the space of search operators but rather on extracting and using implicit problem-knowledge from the problem specification. Also PTO design differs from that of mathematical theories for the design of search operators such as the geometric framework [7], as in PTO the design is automatic and implicit rather than manual and explicit. PTO aims at shielding users from choosing parameters. This is not done by automatic off-line parameter tuning and algorithm configuration [12], but rather by choosing default robust parameter values in a principled manner that work well in many situations. PTO aims at being very easily extendible and encompasses a large number of problems and search algorithms, and its design is highly modular. But unlike existing software libraries for metaheuristics, PTO is a self-configuring system which seamlessly adapts to any problem and any arbitrarily complex representation.

The No Free Lunch theorem [14] does not prevent PTO from good universal performance, because as will be described, PTO automatically tailors the operators to the problem. Thus, PTO is effectively not a black-box method.

3 Implicit operator design

Here, we define the naming scheme and generic search operators on traces. For illustrative purposes, we then give two examples using *simplified* naming schemes to show how the naming scheme implicitly adapts generic search operators to problem-specific search operators by using the control structure in the code of the generator. Finally, we discuss how problem knowledge is embedded in PTO.

3.1 Naming scheme

We follow the approach of Wingate et al. [13] and name random choices according to their structural position in the execution trace, which we define in a way roughly analogous to a stack address: a random choice's name is defined as the list of the functions, their line numbers, and their loop indices, that precede it in the call stack. A trace annotated in this way is called a *structured trace*. The formal specification is inductive, as shown in Algorithm 1 (some details omitted).

Algorithm 1 Annotating a structured trace.

Begin executing the generator with empty function, line, and loop stacks.
When entering a function,
- push a unique function id on the function stack
- push a 0 on the line stack.

When moving to a new line, increment the last value on the line stack.
When starting a loop, push a 0 on the loop stack.
When iterating through a loop, increment the last value on the loop stack.
When exiting a loop, pop the loop stack.
When exiting a function, pop the function stack and the line stack.
When a random choice is made, name it with the entire contents of all stacks.

We also define a *linear trace*. It can be seen as a special case of the structured trace, in which each entry is named after its sequential position in the trace, thus more similar to PODI [6]. The search operators defined next work on both linear and structured traces.

3.2 Search operators on named traces

Initialisation runs the generator and traces its execution. **Point mutation** picks a random entry of the trace and replaces its value with a value drawn from the same random call. **Uniform crossover** aligns parent traces on their names (i.e. dictionary keys). For names that appear in both parents, the offspring inherits the corresponding entries from either parent at random, i.e. using a random mask to select. For names that appear in only one parent, the offspring inherits all of them. **Repair** is applied after each alteration of the trace, i.e., after the application of any variation operator. The repair takes place when

running the modified trace in the generator in playback mode to generate the corresponding solution. If there is a mis-match, i.e. the current value comes from a random call other than the one identified by the name, then a new random value is drawn from the correct random call. If the trace is used up before the generator finishes, the trace is extended with new random entries as needed. Excess entries in the trace, not used by the generator, are deleted.

3.3 Example: loops & matrices

The user writes a generator and objective function using standard Python:

```
1: def generator():
2:     s = random.randint(1,5)
3:     return [[random.randint(1,15) for i in range(s)]
4:                                   for j in range(s)]
5: def objective(matrix): return determinant(matrix)
```

In this simple example, the generator outputs a random square matrix, such as those shown below (bottom of page). The objective function (to be maximised) is the determinant, calculated by recursively decomposing into sub-matrices.

Tracing. When the generator runs to generate a solution, a sequence of random events take place. The sequence of outcomes of the random events is the trace associated with the solution. In the example, the generator makes 1 call to `random.randint(1,5)` and then s^2 calls to `random.randint(1,15)`. The (unannotated) trace is a sequence of the outcomes of these random calls, e.g., `T=(3,9,13,5,1,11,7,3,7,4)`. In this example, the trace (genotype) is a list of integers, and the solution (phenotype) is a square matrix of integers, e.g., trace `T` corresponds to the matrix at bottom of page on left.

Playback. Given a trace, the corresponding solution is found by running the generator in 'playback mode' using the trace to override the source of randomness when random calls are made. This is the genotype-phenotype mapping.

Mutation & linear trace. Point mutation changes a single entry of the trace. E.g., in the trace `T` if 13 changes to 4, then `T` ⇒ `(3,9,4,5,1,11,7,3,7,4)`, corresponding to the change in solution illustrated below on the left. The change in the trace `T` ⇒ `2,9,13,5,1,11,7,3,7,4)` corresponds to the change illustrated in the centre. When the trace is played back in the generator, excess entries are deleted to obtain `(2,9,13,5,1)`. The mutation has changed the size of the matrix from 3 to 2, and the original trace is scanned sequentially (i.e., played back) to fill in the smaller matrix (and the surplus elements are discarded).

$$\begin{pmatrix} 9 & 13 & 5 \\ 1 & 11 & 7 \\ 3 & 7 & 4 \end{pmatrix} \Rightarrow \begin{pmatrix} 9 & 4 & 5 \\ 1 & 11 & 7 \\ 3 & 7 & 4 \end{pmatrix} \; ; \; \begin{pmatrix} 9 & 13 & 5 \\ 1 & 11 & 7 \\ 3 & 7 & 4 \end{pmatrix} \Rightarrow \begin{pmatrix} 9 & 13 \\ 5 & 1 \end{pmatrix} \; ; \; \begin{pmatrix} 9 & 13 & 5 \\ 1 & 11 & 7 \\ 3 & 7 & 4 \end{pmatrix} \Rightarrow \begin{pmatrix} 9 & 13 \\ 1 & 11 \end{pmatrix}$$

This is however not satisfactory, as the mutation operator is treating these square matrices as sequential objects. A more satisfactory mutation operator

would act as illustrated above on the right, extracting a square submatrix of size two from the original one.

Mutation & structured trace. The structured trace aims to fix this problem. For illustrative purposes, in this example we use the simplified naming scheme with line number of the call and loop indices `i` and `j`, i.e., of the form `[line number, i, j]`, instead of the general one in Section 3.1. The original trace `T`, i.e., `(3,9,13,5,1,11,7,3,7,4)`, is then annotated as follows:

`([2,-,-]:3, [3,1,1]:9, [3,1,2]:13, [3,1,3]:5, [3,2,1]:1, [3,2,2]:11, [3,2,3]:7, [3,3,1]:3, [3,3,2]:7, [3,3,3]:4).`

As before, the first element of the trace is mutated from 3 to 2. When played back in the generator, elements of the structured trace are not accessed sequentially but by the context in their names, e.g., when line 3 is executing with `i=1`, and `j=2`, the decision returned from `random.randint(1,15)` is 13. Excess elements not used in play-back are deleted. Since now size=2, when playing back the mutated trace in the generator the values of `i` and `j` that will be encountered are 1 and 2 (never 3), producing the modified trace:

`([2,-,-]:2, [3,1,1]:9, [3,1,2]:13, [3,2,1]:1, [3,2,2]:11).`

Induced phenotypic operators. The structured naming scheme and generic trace operators combine to give induced phenotypic operators tailored to matrices. In mutation, when size changes, the top left square matrix is retained. Crossover works by aligning parent matrices at the top left corner before recombination, as illustrated in Table 1.

Table 1: Uniform crossover on structured traces for matrices: (right) recombine aligned traces $p1$ and $p2$ using random mask m to obtain the child trace uc, which when repaired becomes c; (left) phenotypic effect of crossover.

$$p1 = \begin{pmatrix} 9 & 13 & 5 \\ 1 & 11 & 7 \\ 3 & 7 & 4 \end{pmatrix}$$

$$p2 = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$c = \begin{pmatrix} 9 & 2 \\ 1 & 4 \end{pmatrix}$$

name	p1	p2	m	uc	c
$[2,-,-]$	3	2	2	2	2
$[3,1,1]$	9	1	1	9	9
$[3,1,2]$	13	2	2	2	2
$[3,1,3]$	5			5	
$[3,2,1]$	1	3	1	1	1
$[3,2,2]$	11	4	2	4	4
$[3,2,3]$	7			7	
$[3,3,1]$	3			3	
$[3,3,2]$	7			7	
$[3,3,3]$	4			4	

3.4 Example: recursion & expressions

The example in this section illustrates that PTO with linear trace suffers from a disruptive "ripple effect" similar to GE's due to offspring genotype entries being used out of context. This is resolved by using the structured trace which results in implicit design of meaningful search operators for tree structures.

Below is a recursive generator in standard Python for random Boolean expressions on three variables contained in strings.

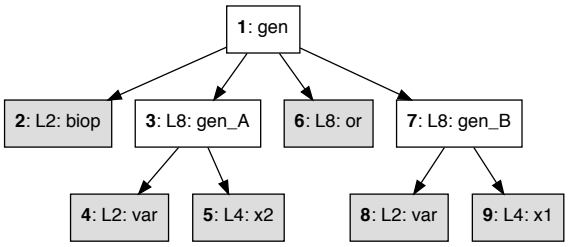
```
1: def gen():
2:   expr_type =  random.choice(['var','uop','biop'])
3:   if expr_type == 'var':
4:     return random.choice(['x1','x2','x3'])
5:   if expr_type == 'uop':
6:     return 'not ' + gen()
7:   if expr_type == 'biop':
8:     return '(' + gen() + random.choice([' and ',' or ']) + gen() + ')'
```

Tracing. Let us consider an example expression `E=(x2 or x1)`. The unannotated trace for this expression is `T=[biop, var, x2, or, var, x1]`.

Mutation & linear trace. The entries of the trace have types, so when we apply point mutation to an entry it can be replaced only with values of the same type e.g., the first entry of `T` with `biop` can be changed only to `var` or `uop` i.e., possible outputs of `random.choice(['var','uop','biop'])`. Point mutation on the *linear* trace can have a global effect. For example, changing `var` in the second entry of `T` to `uop` alters the context of all subsequent elements i.e., results in type-mismatch for all of subsequent elements when the mutated trace is played back in the generator to obtain the corresponding solution. The trace is then repaired by resolving type mismatch errors in the trace by replacing erroneous values by freshly generated values of the correct type, e.g. changing `E=(x2 or x1)` ⇒ `(not x3 and x3)` with trace `[biop, uop, var, x3, and, var, x3]`.

```
[gen, 2] : biop
[gen, 8, gen-A, 2] : var
[gen, 8, gen-A, 4] : x2
[gen, 8] : or
[gen, 8, gen-B, 2] : var
[gen, 8, gen-B, 4] : x1
```

(a)



(b)

Fig. 1: (a) Annotated structured trace for `(x2 or x1)`. To read this we say, e.g., "`x2` is the result of a call on line 4 of the first call (unique ID created by addition of `-A`) to `gen` on line 8 of `gen`." (b) "Derivation" tree. Bold integers indicate execution order; 'L' indicates line number; node labels (e.g. `biop`) indicate results of random calls; edges indicate random calls from parent to child; shading indicates results of terminal random calls.

Mutation & structured trace. Continuing the example, using the simplified naming scheme `[function, line number]*` instead of the general one, the original trace `T`, i.e., `[biop, var, x2, or, var, x1]`, is annotated as in Fig. 1(a). It can be rendered as a tree as in Fig. 1(b). This is analogous to a GE derivation tree, but distinct structurally: e.g. `x1` is not a child of `var` and one does not read the leaves to obtain the output (recall, it is *returned* by the generator).

When we apply the same point mutation to `T` as before (`var` ⇒ `uop` at the second element, that is at the leaf labelled **4** in Fig. 1(b)), now with the structured trace, the mutation affects only the local context and it is much less disruptive, this time changing `E` e.g. `(x2 or x1)` ⇒ `(not x3 or x1)`.

Table 2: Uniform crossover on structured traces for expressions: recombine aligned traces $p1$ and $p2$ using random mask m to obtain the child trace uc, which when repaired becomes c; the phenotypic effect of crossover is:
$p1$=`(x2 or x1)`
$p2$=`(not x3 and x3)`
c=`(not x3 or x1)`.

name	p1	p2	m	uc	c
[gen, 2]	biop	biop	2	biop	biop
[gen, 8]	or	and	1	or	or
[gen, 8, gen-A, 2]	var	uop	2	uop	uop
[gen, 8, gen-A, 4]	x2			x2	
[gen, 8, gen-B, 2]	var	var	1	var	var
[gen, 8, gen-B, 4]	x1	x3	1	x1	x1
[gen, 8, gen-A, 6, gen, 2]		var		var	var
[gen, 8, gen-A, 6, gen, 4]		x3		x3	x3

Induced phenotypic operators. The structured naming scheme and generic trace operators combine to induce phenotypic operators tailored to nested expressions. The phenotypic operators have a *modularity* property.
Mutation: when a decision node C (a leaf in the tree view of the annotated trace) is mutated, no other change may be needed, so the effect is a point mutation on the expression. In the worst case, mutation invalidates the contents of all subtrees, of parent node P, following C. This is because (i) P (a function call) is scoped and modular, and (ii) the execution order is mirrored in the ordering of siblings of C, so a change in C cannot have an effect on past execution.
Crossover (see Table 2): aligning parents on names corresponds to aligning their common tree structures at root (homologous crossover) with the effect on phenotypes that corresponding sub-expressions between parents are exchanged.

Analogously to these two examples (Section 3.3 on matrices and Section 3.4 on trees), the general naming scheme in Section 3.1 can induce phenotypic operators tailored to any solution structure because it encompasses all control structures i.e., sequential, conditional, nested and recursive function calls, nested loops, and any composition of these, that can describe any generator.

3.5 Implicit problem knowledge

Different generators for the same problem lead to different annotation on traces, different induced phenotypic operators, and so to different performance. PTO assumes that the control structures used by the user coding the generator implicitly reflect an understanding of the inherent structure of the problem and its underlying implicit representation. E.g., the user could have used a flat list representation for the matrix problem. This generator would not embed problem knowledge, and would lead to a structured trace coinciding with the linear trace, hence possibly worse performance. The assumption of PTO is that users will see the patterns and structures of the problem and use these in writing generators.

We believe that this is a realistic assumption, and users will do this naturally and intuitively. PTO then automatically carries out design based on user's intuition on the problem. The resulting phenotypic operators will be effective to the extent that the generator captures the structure of the problem.

Perfecting the naming scheme based on experience of how users use the system in practice, and providing explicit guidelines to the user of how to effectively put structure in generators, is an important line of future research.

4 Experiments and results

Is the structured trace better than the linear trace? Do "smart" generators boost performance? To seek preliminary answers, we test PTO on two very different domains: travelling salesman problems, and symbolic regression with grammatical evolution. We use three solvers: *Random search* (RS), simple *Hill-climbing* (HC), and an *Evolutionary Algorithm* (EA). The budget is set to 20,000 evaluations for all experiments. For the EA, PTO internally sets the number of generations = population size = $\sqrt{20000} = 141$.

For **symbolic regression by GE**, the grammar for n variables is:

```
<expr> ::= <op>(<expr>, <expr>) | <var> | <const>
<op> ::= add | sub | mul | aq
<var> ::= x1 | x2 | ... | xn
<const> ::= 0.0 | 0.1 | ... | 1.0
```

Here, `aq` is the analytic quotient $aq(x, y) = x/\sqrt{1.0 + y^2}$. We use the natural generator which creates a program by making uniform choices among all possible productions at each step. There is no maximum depth or weighting by tree depth. The objective is -RMSE. We report results on several problems. Synthetic instances are polynomials on n variables with degree $d = 4$, for $n = \{1, \ldots, 10\}$, with coefficients uniform in $[0, 1]$ for all possible terms. Well-known benchmarks are also used: Pagie-2D, Vladislavleva-4, Dow Chemical, Tower, and Housing[3]. We compare 3 solvers and 2 trace types, *Linear* (L) and *Structured* (S). We show results on training data only since the goal is to investigate search performance rather than generalisation. Results are shown in Fig. 2. These show that the best combinations use the structured trace: HC/S and EA/S. HC/L does very well, whereas EA/L is very weak. RS does very badly on larger synthetic problems, and for RS the trace type makes no difference. For synthetic problems, the differences are stronger for larger problems, demonstrating PTO scaling.

For **travelling salesman problems** we define two generators. In the *Unbiased* (U) generator, a random permutation is generated by starting with the integers $\{1, \ldots, n\}$, and for each index, swapping with a randomly-chosen later index. In the *Heuristic* (H) generator, a route is constructed by starting with a random starting city, and at each step, choosing the next city randomly from all those remaining, with their probabilities inversely weighted by the distance

[3] Datasets taken from `http://www.github.com/ponyge/ponyge2`.

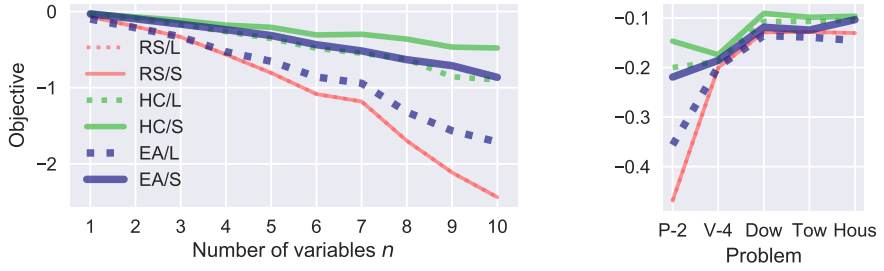


Fig. 2: Regression results (mean of 30) for polynomials (left) and dataset problems (right).

from the current city. Results on 6 TSPLIB[4] instances for 3 solvers and 2 generators are shown in Table 3. On larger problems in particular, EA/H is the best combination: it out-performs each of its components (EA/U and RS/H).

Table 3: TSP results presented as mean (stddev). Lower is better. Integers in instance names (48–575) indicate problem size. Bold indicates best result per-instance.

Solver	Gen	att48	berlin52	eil101	u159	a280	rat575
RS	U	112397 (3521)	23108 (464)	2854 (43)	381521 (5157)	30203 (420)	104536 (935)
RS	H	71150 (2084)	15878 (448)	2029 (40)	222018 (3822)	18542 (304)	64691 (608)
HC	U	67573 (4875)	14699 (827)	**1628** (79)	198736 (8807)	16390 (520)	59456 (1385)
HC	H	**59976** (6092)	**13850** (1110)	1888 (100)	201490 (12426)	18286 (1084)	65858 (2172)
EA	U	68377 (4678)	14103 (913)	1747 (97)	224842 (9276)	18936 (499)	70865 (1183)
EA	H	67503 (5192)	14727 (1065)	1910 (65)	**112444** (6228)	**10086** (279)	**34949** (705)

Overall, the structured trace gives consistently better performance than the linear trace, and "smart" generators can do very well. One surprising result is the good performance of hill-climbing with a linear trace in GE.

5 Conclusions and future work

PTO provides a novel perspective on heuristic optimization by neatly separating problem specification and search algorithm, and automatically tailoring search operators to the problem at hand. We believe PTO has great potential and envisage an ambitious research plan, organised in three research strands.
(i) Problems: PTO will be extended to other optimisation paradigms such as multi-objective, co-evolution, dynamic and noisy objective functions. A broad range of complex real-world applications will be tested, from logistics to interactive art. Generators for many problems will be borrowed from throughout the fields of heuristics and EC and tested.

[4] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

(ii) Trace: Variant naming schemes will be investigated for performance and for their effect on operator design. The naming scheme will be used in a GE variant which avoids the ripple effect in a principled way. Alternative styles of writing generators will be investigated, and the results provided to users as guidelines. **(iii) Algorithms**: Many more metaheuristics will be plugged-in to PTO, including several already in the geometric framework [7], such as Surrogate-Based Optimisation and Estimation of Distribution Algorithms. Robust parameter settings and self-adaptive parameters will be investigated. Landscape analysis will be used to validate algorithm design and generator choices. The links between PTO and probabilistic programming will be used to import and export ideas.

Finally, PTO will be promoted as a community resource. Code is available at `https://github.com/program-trace-optimisation`.

References

1. de Armas, J., Keenan, P., Juan, A.A., McGarraghy, S.: Solving large-scale time capacitated arc routing problems: from real-time heuristics to metaheuristics. Annals of Operations Research pp. 1–28 (2018)
2. Batishcheva, V., Potapov, A.: Genetic programming on program traces as an inference engine for probabilistic languages. In: ICAGI. pp. 14–24. Springer (2015)
3. Burke, E.K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., Qu, R.: Hyper-heuristics: a survey of the state of the art. Journal of the Operational Research Society **64**(12), 1695–1724 (Dec 2013)
4. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Future of Software Engineering. pp. 167–181. ACM (2014)
5. Janssen, P., Kaushik, V.: Decision chain encoding: evolutionary design optimization with complex constraints. In: EvoMUSART. pp. 157–167. Springer (2013)
6. McDermott, J., Carroll, P.: Program optimisation with dependency injection. In: Krawiec, K., et al. (eds.) EuroGP. pp. 133–144. Springer (2013)
7. Moraglio, A.: Towards a geometric unification of evolutionary algorithms. Ph.D. thesis, University of Essex (2008)
8. Moraglio, A., Di Chio, C., Poli, R.: Geometric particle swarm optimisation. In: Ebner, M., et al. (eds.) EuroGP. pp. 125–136. Springer (2007)
9. O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Kluwer Academic Publishers, Norwell, MA (2003)
10. O'Neill, M., Ryan, C., Keijzer, M., Cattolico, M.: Crossover in grammatical evolution. Genetic Programming and Evolvable Machines **4**(1), 67–93 (Mar 2003)
11. Rothlauf, F., Oetzel, M.: On the locality of grammatical evolution. In: Collet, P., et al. (eds.) EuroGP. pp. 320–330. Springer (2006)
12. Stützle, T., López-Ibáñez, M.: Automatic (offline) configuration of algorithms. In: Laredo, J.L.J., et al. (eds.) GECCO (Companion), pp. 681–702. ACM (2015)
13. Wingate, D., Stuhlmueller, A., Goodman, N.: Lightweight implementations of probabilistic programming languages via transformational compilation. In: Gordon, G., et al. (eds.) AISTATS. PMLR, vol. 15, pp. 770–778 (11–13 Apr 2011)
14. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. Trans. Evol. Comp **1**(1), 67–82 (Apr 1997)