Program Trace Optimization with Constructive Heuristics for Combinatorial Problems

James McDermott[1] and Alberto Moraglio[2]

[1] National University of Ireland, Galway `james.mcdermott@nuigalway.ie`
[2] University of Exeter, UK `A.Moraglio@exeter.ac.uk`

Abstract. Program Trace Optimisation (PTO), a highly general optimisation framework, is applied to a range of combinatorial optimisation (COP) problems. It effectively combines "smart" problem-specific constructive heuristics and problem-agnostic metaheuristic search, automatically and implicitly designing problem-appropriate search operators. A weakness is identified in PTO's operators when applied in conjunction with smart heuristics on COP problems, and an improved method is introduced to address this. To facilitate the comparison of this new method with the original, across problems, a common format for PTO heuristics (known as generators) is demonstrated, mimicking GRASP. This also facilitates comparison of the degree of greediness (the GRASP α parameter) in the heuristics. Experiments across problems show that the novel operators consistently outperform the original without any loss of generality or cost in CPU time; hill-climbing is a sufficient metaheuristic; and intermediate levels of greediness are usually best.

Keywords: Constructive heuristics, GRASP, search operators

1 Introduction

Many heuristic and metaheuristic methods have been applied in the field of combinatorial optimisation (COP). Often they achieve good results on problems where exact methods become infeasible. Research in this area is somewhat disunified in that many methods have been individually specialised to many problems: in each case, a combination of domain expertise and algorithmic expertise is required to design suitable representations and operators.

However, many of these methods do share common concepts, such as (in constructive heuristics) sampling of solution elements biased by their cost, and (in metaheuristics) perturbation of existing solutions. It is a natural goal to unify these methods in a single, general approach. It is also desirable to achieve a degree of automation in the application of these methods to problems, rather than requiring algorithmic expertise to be re-applied to each new problem.

Program trace optimisation (PTO) [1] is a recent optimisation framework which is highly general and unifying, and which does (in a sense to be clarified later) automate the work of adapting an algorithm to a problem; and so it responds to these research challenges.

In fact, PTO is not just highly general but in a sense *maximally* general, because it uses a representation which by the Church-Turing thesis is the most general possible – the *program trace*, that is a data structure representing a history of execution of a program. The program in question is a non-deterministic *solution generator*, which plays the role of a genotype-phenotype map, seeing the program trace as a genotype and a candidate solution as a phenotype. To be specific, the generator is a program or function which randomly samples one element from the solution space, with or without bias, for example in a bitstring space it returns a single bitstring chosen randomly. With bias, it becomes a constructive or generative heuristic. PTO thus gives a unifying view.

Heuristic and metaheuristic approaches are commonly hybridised, for example in GRASP [2]. In PTO, both the heuristic and metaheuristic aspects are pluggable, and it is natural in PTO to compare a given heuristic across several metaheuristics. In this paper, we borrow well-known constructive heuristics for several problems as PTO generators. This re-use of known-good ideas is a central goal of PTO. We write all of them in a GRASP-like format, and this gives the ability to understand and compare PTO program traces and search behaviour across problems. It allows for a systematic experiment also on the well-understood "greediness" parameter of GRASP.

In most metaheuristics, the ability to make *small* perturbations is essential to success: the perturbation operator should give a new solution which retains or *re-uses* almost all of the information of the previous solution. But on some COP problems, with typical constructive heuristics as generators, it turns out that the PTO operators do not succeed in re-using as much of this information as they could. The problem is due to the repair mechanism which runs after each genetic variation operator. Therefore, in this paper an improved repair mechanism is designed, which succeeds in re-using as much information as possible.

Reader's guide In the next section, Section 2, we describe PTO itself and previous related work, with an expanded explanation of PTO mechanisms. We then describe in detail our fixed format for generators for COP problems in Section 3. The novel repair mechanism is described in Section 4. Experiments and results are in Section 5, and we conclude with Section 6.

2 PTO and related work

PTO uses a user-supplied non-deterministic generator function as an implicit definition of the representation. PTO sees the sequence of random decisions made during execution of this program as a genotype.

PTO's modular design is shown in Figure 1. The user supplies a problem-specific generator and fitness function. The core of PTO executes the generator once per individual. Each execution gives both a trace (genotype) and a solution (phenotype). Any metaheuristic can be plugged-in as the solver, carrying out its genetic operations on the trace.

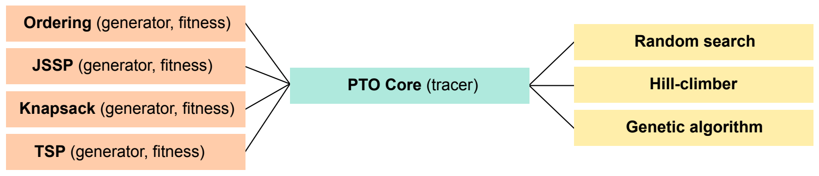
Ordering (generator, fitness)
JSSP (generator, fitness)
Knapsack (generator, fitness)
TSP (generator, fitness)
PTO Core (tracer)
Random search
Hill-climber
Genetic algorithm

Fig. 1: PTO architecture

The idea of using such a sequence of decisions as a genotype was introduced in the Program Optimisation with Dependency Injection (PODI) system [3]. The generators used varied from uniform sampling of the space for symbolic regression problems, to initialisation operators from previous metaheuristic research in communications network design, re-purposed as generators, to idiosyncratic hand-written code originally intended only as a tool for random generation of 3D structures. PTO goes beyond PODI by using a more sophisticated trace representation, allowing improved re-use of genetic material.

The idea of hybridising contructive heuristics with metaheuristic search is well-established in the COP literature. GRASP combines a greedy heuristic with path relinking and local search [2]. As a generalisation, biased randomisation and simulated annealing have been combined [4, 5]. PTO with GRASP-like generators differs from GRASP itself by allowing any metaheuristic to be plugged-in. In some hybrid approaches, the greedy constructive procedure is used as a "smart" initialisation, after which the algorithm uses metaheuristic search alone [6]. PTO uses the heuristic throughout the metaheuristic search process.

In the evolutionary computation literature, the genotype-phenotype mapping is a common research topic. "Smart" mappings seek to build problem-awareness (e.g. constraint handling) directly into the algorithm, e.g. [7]. PTO differs from such approaches by using a program trace as the genotype and aims more for universality and modularity – the algorithm is unchanged for new problems.

2.1 The program trace: a universal solution representation

As already explained, in PTO the program trace is 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. Metaheuristics defined on such a representation can be applied *unchanged* to any problem, thus becoming universal optimisers.

The trace representation is a *dictionary*. Each entry is a key-value pair corresponding to one random decision made by a random function call during execution of the generator. The value is simply the output of the random function call, for example an integer or float. The key gives the structural position of that decision in the execution trace, rather like a stack address: it includes the list of functions, their line numbers, and their loop indices, that precede the random call in the call stack. This scheme follows the approach of Wingate et al. [8]. Examples in the context of well-known COP problems are given in Section 3.

2.2 Solvers and search operators

The search operators required by metaheuristics such as hill-climbers and evolutionary algorithms are defined on the trace representation. They are defined in a principled way based on the geometric framework [9].

Initialisation runs the generator and traces its execution. The resulting trace becomes the newly-initialised genotype, and the output of the generator becomes its phenotype. **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. Repair is discussed further in Section 4.

Given these operators, metaheuristic solvers such as random search, hill-climbing, and a GA can be defined. The trace becomes the genotype, and the generator becomes the genotype-phenotype map.

2.3 The role of the generator

The PTO generator can be seen as a genotype-phenotype mapping, where the genotype is the program trace and the phenotype is a candidate solution. PTO search operators work directly on the genotype, but we can also see them as implicitly working on the phenotype. What is their behaviour at that level?

In the framework of Jones [10], a search problem begins with an *object space* – in which the possible solutions are contained. The user is required to provide a representation space, that is a space in which points can be manipulated by search operators, and points can be mapped "forward" to the object space. E.g. in the TSP, the object space is the space of tours, and the representation space is the space of permutations. The distinction captures the extra meaning and structure associated with the object space. Objects in the object space are human-readable; those in the representation space, machine-manipulable. In PTO, the generator *implicitly* defines the object space. The job of the PTO user is to supply a generator which samples from the object space, rather than to design a representation space and operators on it. This workflow may suit domain experts better (but may be less familiar to metaheuristics researchers). One feature of PTO which is interesting for constrained COP problems such as JSSP is that it searches the space of *feasible* solutions. For many such problems it is not difficult to write a generator giving only feasible solutions, but it is difficult to design mutation and crossover operators that given feasible parents are guaranteed to return feasible offspring. PTO's search operators preserve feasibility automatically.

The operators applied by PTO on the trace representation correspond, when viewed at the level of the solution, to operators well-designed for the problem, in the sense that they take advantage of the meaning and structure of the object space. For example, if the solution space is of bitstrings, then a natural generator gives operators equivalent to well-known GA bitstring operators. If a generator uses nested loops, the implicit representation is a matrix; if one uses recursion the implied representation is a tree. Thus the same operators on traces result in meaningful operators for vectors, matrices and trees. This has been demonstrated in previous work [1], giving confidence that the PTO implicit adaptation works well. The case of permutations is different and is treated in Section 4. For a further example, when a greedy, randomised constructive heuristic is used as the generator, PTO can be expected to behave similarly to mutate-and-repair methods already well-known in the COP literature [5, 4]. Although the end result may replicate a known-good method, the benefit for researchers and practitioners is in the automation: good problem-appropriate operators are derived, requiring domain knowledge but not algorithmic knowledge on the part of the user.

However, some caveats apply. The operators designed by PTO will be well-designed for the meaning and structure of the object space only to the extent that this meaning and structure is present in the user-supplied generator. Two generators which are semantically identical but syntactically different may induce different operators. As an example, given a list `L` from which the generator must sample an element, `i = random.choice(range(len(L))); x = L[i]` is semantically equivalent to `x = random.choice(L)`, but the latter expresses problem structure more directly. It is an assumption of PTO that the user will use direct formulations such as this. Also, in its implicit design of operators, PTO uses only the generator, not the objective function. For example, both the n-queens problem and the TSP can be represented as permutation problems, so a generator which uniformly samples permutations could be used equally for both. It would give no advantage on either problem relative to a standard permutation representation. A domain expert might see that for the TSP, a more suitable generator might iteratively build a solution heuristically guided by inter-city distance. This is a method of building problem knowledge into the representation.

2.4 Open questions

Several important research questions remain open. (1) As described, PTO automatically designs operators for each problem. We wish to explore how well PTO performs across COP problems with the minimum input of expertise. (2) A beneficial side-effect of the PTO unifying view is the ability to compare across problems. To further this comparison, in the current paper we make use of a fixed format, borrowed from the core idea of GRASP, for the solution generators for several different problems. These generators are parameterised by a single parameter α which controls greediness. Furthermore, in PTO any metaheuristic can be used in combination with any generator, including combinations which may be rare in the COP literature. We wish to compare performance and the obtained traces and solutions, across problems, searchers and α values. (3) PTO relies for its success on the ability to manipulate the trace data structure. As we will see, in some COP problems the natural generator leads to a trace in which, after manipulation, many elements of the trace cannot be re-used, and so learning is lost. We introduce to PTO a novel mechanism to prevent this and allow strong re-use of trace elements, and we investigate its effect. We do not aim in this paper to compare PTO performance with other approaches.

3 GRASP in PTO

To apply PTO to solve COP problems, we need to provide a problem-specific solution generator. In this paper we use a fixed GRASP-like generator scheme for all problems, as shown in Algorithm 1. It assumes that problem solutions are characterised as aggregations of features: for example, in TSP the cities are features and a solution is a list of cities. At each step, it finds a list of allowed features (a subset of those not yet chosen); calculates their per-feature cost in the presence of the in-progress solution; filters them for low cost, forming the *restricted candidate list* (RCL); and chooses randomly.

Algorithm 1 GRASP's greedy randomised construction scheme

```
1: procedure greedy-randomised-construction()
2:     solution ← empty-solution()
3:     while not complete(solution) do
4:         features ← allowed-features(solution)
5:         for f in features do
6:             costs[f] ← cost-feature(solution, f)
7:         end for
8:         RCL ← { f | costs[f] ≤ min(costs) + α(max(costs) - min(costs))}
9:         solution ← add-feature(solution, random.choice(RCL))      ▷ Randomness
10:    end while
11:    return solution
12: end procedure
```

Algorithm 2 Procedures for the Ordering problem on n items

```
procedure empty-solution()
    return empty list
end procedure
procedure allowed-features(solution)
    return $\{i \mid 1 \leq i \leq n \wedge i \notin \text{solution}\}$
end procedure
procedure cost-feature(solution, f)
    if solution is empty then
        last-item ← 0
    else
        last-item ← last element of solution
    end if
    return |f−last-item+1|                ▷ equal to 0 for consecutive integers
end procedure
procedure complete(solution)
    return solution has length $n$?                              ▷ True or False
end procedure
procedure add-feature(solution, f)
    return append f to solution
end procedure
```

Algorithm 3 New procedures for TSP on n cities with distance matrix D

```
procedure cost-feature(solution, f)
    if solution is empty then
        return 0                          ▷ all cities have zero cost as start city
    else
        last-item ← last element of solution
    end if
    return D[last-item, f]                ▷ travel cost from current to next city
end procedure
```

Algorithm 4 Procedures for a Knapsack problem with items $1 \dots n$, weights w, profits p, and weight limit W

```
procedure allowed-features(solution)
    remaining ← $\{i \mid 1 \leq i \leq n \wedge i \notin \text{solution}\}$
    current ← $\sum\{w[i] \mid i \in \text{solution}\}$
    return $\{i \mid i \in \text{remaining} \wedge \text{current} + w[i] \leq W\}$
end procedure
procedure cost-feature(solution, f)
    return $-p$[f]                ▷ negative: $p$ is a profit but we must return a cost
end procedure
procedure complete(solution)
    current ← $\sum\{w[i] \mid i \in \text{solution}\}$
    min-weight-left ← $\min\{w[i] \mid i \notin \text{solution}\}$
    return solution has length $n$? or current + min-weight-left $> W$
end procedure
```

Algorithm 1 is generic, suitable for several COP problems. It remains to fill in its sub-procedures `empty-solution`(), `complete`(), `allowed-features`(), `cost-feature`(), and `add-feature`() in a problem-specific way. These procedures are shown for several problems in Algorithms 2–4. For TSP, only the `cost-feature` procedure is shown as the others are the same as for Ordering. JSSP procedures are omitted due to lack of space but are specified in our source code available online.

The only source of randomness in our generators is `add-feature`() (line 9 of Algorithm 1). This is the only part of the generator affecting the trace representation, hence we will have a uniform, sequential trace structure for all problems. The parameter α controls the level of greediness in the heuristic ($\alpha = 0$ is fully greedy, $\alpha = 1$ is fully random), and has the same interpretation across problems (and in previous GRASP literature). As we will see, this fixed generator format also leads to a fixed trace format, aiding our understanding of the algorithm.

3.1 Examples of genotype-phenotype correspondence

In the following, we give two examples to illustrate the information contained in the trace representation (genotype), and how the same genotype corresponds to different types of solutions (phenotypes) for different problems.

Let us consider TSP with 10 cities and with $\alpha = 0.5$. The phenotype `[0, 9, 2, 1, 6, 3, 4, 7, 5, 8]` (tour of cities) corresponds to the following trace:

Address	Type	Value
0	random.choice([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])	0
1	random.choice([2, 4, 8, 9])	9
2	random.choice([1, 2, 3, 4, 5])	2
3	random.choice([1, 3, 4, 5])	1
4	random.choice([5, 6, 7])	6
5	random.choice([3, 4, 5])	3
6	random.choice([4, 5])	4
7	random.choice([7, 8])	7
8	random.choice([5])	5
9	random.choice([8])	8

The trace has three parts: (i) a runtime address (or name) of each entry in the trace; (ii) an entry type which consists of the elementary random generator and the values passed to it as argument; (iii) the value generated by the random generator. In all examples in this paper, the sequential nature of the GRASP generator means that the runtime address simplifies to an incrementing integer. The elementary random generator here is always `random.choice` as it is the only source of randomness in the GRASP generator. The argument passed to it is the RCL available at the moment of the call of the elementary random generator. This argument at different point in the execution is naturally different depending on the features still available at that point (e.g. cities not yet used in the construction of the solution) as well as features that have passed the selection based on the specific heuristic used in the construction (e.g. cities nearer to the last city of the tour under construction). The last column contains the actual values sampled by the elementary random generator with the specific argument. So, e.g., in the second line the value 9 was sampled from the call to `random.choice([2, 4, 8, 9])`.

As a second example, let us consider KNAPSACK with 10 items and with $\alpha = 1.0$ (fully random generator). This removes the effect of the heuristic, leaving only the constraining effects of previously selected items and backpack capacity.

The phenotype (knapsack items) `[9, 3, 1, 0, 2]` gives the following trace:

Address	Type	Value
0	random.choice([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])	9
1	random.choice([0, 1, 2, 3, 4, 5, 6, 7, 8])	3
2	random.choice([0, 1, 2, 4, 5])	1
3	random.choice([0, 2])	0
4	random.choice([2])	2

The interpretation of the trace is analogous to the TSP case. The TSP and Knapsack are different problems, but have the same trace representation as they use the same generator format, and in particular the same source of randomness in the same program execution context. The same is true of the other problems considered in this paper.

4 New repair method in PTO

In the following, we explain the current trace repair method in PTO, and why this needs refining, especially in conjunction with smart generators.

The trace in PTO is annotated with names and types on each entry. The name of an entry provides its 'execution address'. The type of an entry is the elementary generator and the arguments passed to it, which was used to generate the value in the trace at that entry.

Variation operators acting on the trace are typed, i.e., they change the values of the trace in conformity to their entry types, e.g., point mutation replaces the value at an entry in the trace with a new value obtained by re-sampling the associated elementary random generator to that entry (with the input arguments). However, this may still result in invalid traces, which present inconsistencies when expressed into phenotypes, because of the (implicit) 'runtime' dependencies between entries of the trace. For this reason, PTO has a repairing method that is applied to each trace when modified.

The current repairing method is as follows. When generating the corresponding phenotype of a trace (by playing back the trace in the generator), if we get a mismatch between the required type from the generator (i.e., the current elementary random function called in the generator) and the type of the corresponding entry in the trace (identified by its 'execution address'), then the entry of the trace is repaired by replacing its type with the required type, and discarding the value at that entry, which is then regenerated by sampling it from the new type (i.e., new random function and its argument) of the entry.

This repairing method correctly amends traces to produce valid phenotypes, but it may also be quite disruptive as it may require substantial change to a trace to produce a valid trace (see example below). This effectively would correspond to a form of macro-mutation (applied in addition to the intended modification done by variation operator) that may be detrimental for search performance. Ideally, we would like a repair method that produces a valid phenotype while making a minimal change to the trace. This is the aim of the new repair method.

Analogously to the old method, the new method repairs the type of an entry when a mismatch is encountered, however the value at that entry is kept (not re-sampled) if that value *could have been obtained* by running the elementary random function linked with the new entry type. For example, if the type and value of an entry in the trace is `random.choice([1,2,3]) : 2` and the required type from that generator is `random.choice([2,3,4])`, the value 2 at that entry can be "recycled" as it could have been obtained from `random.choice([2,3,4])`, and only the type of the entry is amended, giving `random.choice([2,3,4]) : 2`.

The new repair method is much less disruptive in generators in which elementary random functions are called with arguments that depend on random outcomes in the generator at earlier stages. This is the situation that arises when using GRASP-in-PTO generators (but also in many other cases) in which the list of items available to be sampled at a point in time (the argument of the `random.choice` function) to be added to a solution under construction are those that have not been selected previously (as random outcomes at previous stages in the generator).

Using the old repair method, a single change in a value of the trace (point mutation) triggers a 'snowball' effect of type-mismatch in all the subsequent entries of the trace as the arguments of these types depend on the value that was changed. So, the overall effect of point mutation together with the old repair is effectively a macro-mutation, which is clearly an unintended effect of point mutation.

Using the new repair method, a single change in a value of the trace (point mutation) has only a limited effect on the subsequent entries of the trace, as the arguments of these types even if different are still compatible with the values down the line in the trace. So, in this case the overall effect of the point mutation together with the new repair is a much less disruptive form of mutation.

The exact effects of the old and new repair methods are dependent on the specific generator, but in general the new method is guaranteed to be less disruptive than the old.

In Table 1, we compare the disruption of the old and new repair methods on an illustrative example, that of iteratively sampling from a list to produce a permutation. Here, ? indicates that a resampling event is required, while * indicates "recycling" as described above. As we can see, the new repair mechanism triggers fewer resampling events.

Table 1: Point mutation (top) and 1-point crossover (bottom).

	Trace	Mutated trace	Old repair	New repair
1	[1,2,3,4,5,6,7] : 1	[1,2,3,4,5,6,7] : 1	[1,2,3,4,5,6,7] : 1	[1,2,3,4,5,6,7] : 1
2	[2,3,4,5,6,7] : 2	[2,3,4,5,6,7] : 7	[2,3,4,5,6,7] : 7	[2,3,4,5,6,7] : 7
3	[3,4,5,6,7] : 3	[3,4,5,6,7] : 3	[2,3,4,5,6] : ?	[2,3,4,5,6] : 3*
4	[4,5,6,7] : 4	[4,5,6,7] : 4	[2,3,4,5,6] : ?	[2,4,5,6] : 4*
5	[5,6,7] : 5	[5,6,7] : 5	[2,3,4,5,6] : ?	[2,5,6] : 5*
6	[6,7] : 6	[6,7] : 6	[2,3,4,5,6] : ?	[2,6] : 6*
7	[7] : 7	[7] : 7	[2,3,4,5,6] : ?	[2] : ?
	Phenotype		Phenotype	Phenotype
	(1 2 3 4 5 6 7)		(1 7 ? ? ? ? ?)	(1 7 3 4 5 6 ?)

	Parent trace 1	Parent trace 2	Recombined trace	Old repair	New repair
1	[1,2,3,4,5,6,7] : 1	[1,2,3,4,5,6,7] : 7	[1,2,3,4,5,6,7] : 1	[1,2,3,4,5,6,7] : 1	[1,2,3,4,5,6,7] : 1
2	[2,3,4,5,6,7] : 2	[1,2,3,4,5,6] : 6	[2,3,4,5,6,7] : 2	[2,3,4,5,6,7] : 2	[2,3,4,5,6,7] : 2
3	[3,4,5,6,7] : 3	[1,2,3,4,5] : 5	[1,2,3,4,5] : 5	[3,4,5,6,7] : ?	[3,4,5,6,7] : 5*
4	[4,5,6,7] : 4	[1,2,3,4] : 4	[1,2,3,4] : 4	[3,4,5,6,7] : ?	[3,4,6,7] : 4*
5	[5,6,7] : 5	[1,2,3] : 3	[1,2,3] : 3	[3,4,5,6,7] : ?	[3,6,7] : 3*
6	[6,7] : 6	[1,2] : 2	[1,2] : 2	[3,4,5,6,7] : ?	[6,7] : ?
7	[7] : 7	[1] : 1	[1] : 1	[3,4,5,6,7] : ?	[6,7] : ?
	Phenotype	Phenotype		Phenotype	Phenotype
	(1 2 3 4 5 6 7)	(7 6 5 4 3 2 1)		(1 2 ? ? ? ? ?)	(1 2 5 4 3 ? ?)

5 Experiments and results

The goal of our experiments is to explore the performance of PTO with GRASP-like generators on a range of COP problems, the effect of the novel trace repair mechanism, and the effect of the α parameter. Our goal is not to achieve state-of-the-art results, and in fact we can expect the results to be similar to those achieved by GRASP itself with the same parameters.

Our code is available in Github[3] and a dedicated script for the following experiments is available. The PTO implementation is written in Python, and has been adapted for execution in PyPy[4] for an approximate $8\times$ speed-up relative to pure Python (our bottleneck is algorithmic rather than numerical, so PyPy is more effective than the numerical library Numpy).

5.1 Problems and instances

We have chosen a mix of dataset-based COP problems for realism and synthetic problem instances (of sizes comparable to real-world datasets) for controlled investigation of scalability. The TSP, JSSP and Knapsack problem are well-known.

[3] `https://github.com/program-trace-optimisation/PTO`
[4] `https://pypy.org`

We use the simplest, most canonical version in each case, and in particular we use the 1-dimensional Knapsack. In the Ordering problem of size n, a "toy problem", the solution is a permutation of size n and the goal is simply to assemble the permutation $(12 \ldots n)$. The objective function penalises each entry x_i in the permutation by $|x_{i-1} - x_i + 1|$. Since PTO is phrased as a maximising algorithm, we define Ordering fitness as the negative of that summed penalty. Similarly, TSP fitness is defined as negative cost, and JSSP as negative makespan.

For Ordering, we use the instances of sizes 10, 20, 40, 80, 160, 320. For Knapsack, we generate random instances of the same sizes. For TSP, we have taken 6 instances from TSPLIB[5], named `att48`, `berlin52`, `eil101`, `u159`, `a280`, and `rat575` (where the integer gives the size). For JSSP, we have taken the instances `azb5-azb9` [11] and `yn4` [12] from the OR Library[6] file `jobshop1.txt`, with sizes 10×10, 10×10, 20×15, 20×15, 20×15, 20×20. Although these instances are not large, they are commonly used in modern JSSP research [13].

5.2 Experimental design

We use three solvers plugged-in to PTO: Random Search (RS), Hill-Climbing (HC), and an Evolutionary Algorithm (EA). In HC a move is accepted if better than or equal to the current point. In the EA, we use 0.5-truncation selection, a crossover rate of 1 and a mutation rate of 1 per individual. The budget is set to 20,000 evaluations for all experiments, as in previous work [1]. For the EA, PTO internally sets the number of generations = population size = $\sqrt{20000} = 141$. This is in keeping with the PTO philosophy of minimising the amount of user configuration required.

The novel trace repair mechanism introduced in Section 4 is compared against the original PTO trace repair mechanism. We will refer to these as *Strong* reuse and *Weak* reuse respectively. We compare 5 α values, $\{0.0, 0.1, 0.5, 0.9, 1.0\}$.

We have a total of (4 problems) $\times$ (6 instances per problem) $\times$ (3 solvers) $\times$ (2 re-use approaches) $\times$ (5 α values) $\times$ (20 repetitions) = 14400 runs.

5.3 Results

Results are shown in Fig. 2. By inspection, we have four main results. (1) Strong reuse is better than Weak reuse. The original trace repair mechanism, giving Weak reuse, was too disruptive for the type of generator (constructive heuristic) used on these problems. (2) Table 2 shows also the mean time taken per run. As expected, the time taken increases with problem size. There is no important difference in time taken between the Weak and Strong re-use methods. A further result is not shown: increasing α tends to increase the time taken, since it gives a larger RCL. (3) the HC solver is better than either RS or EA. The use of HC over RS amounts to the difference between using a combined constructive heuristic/search metaheuristic (as in GRASP and PTO) and using a constructive

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

[6] `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/jobshopinfo.html`

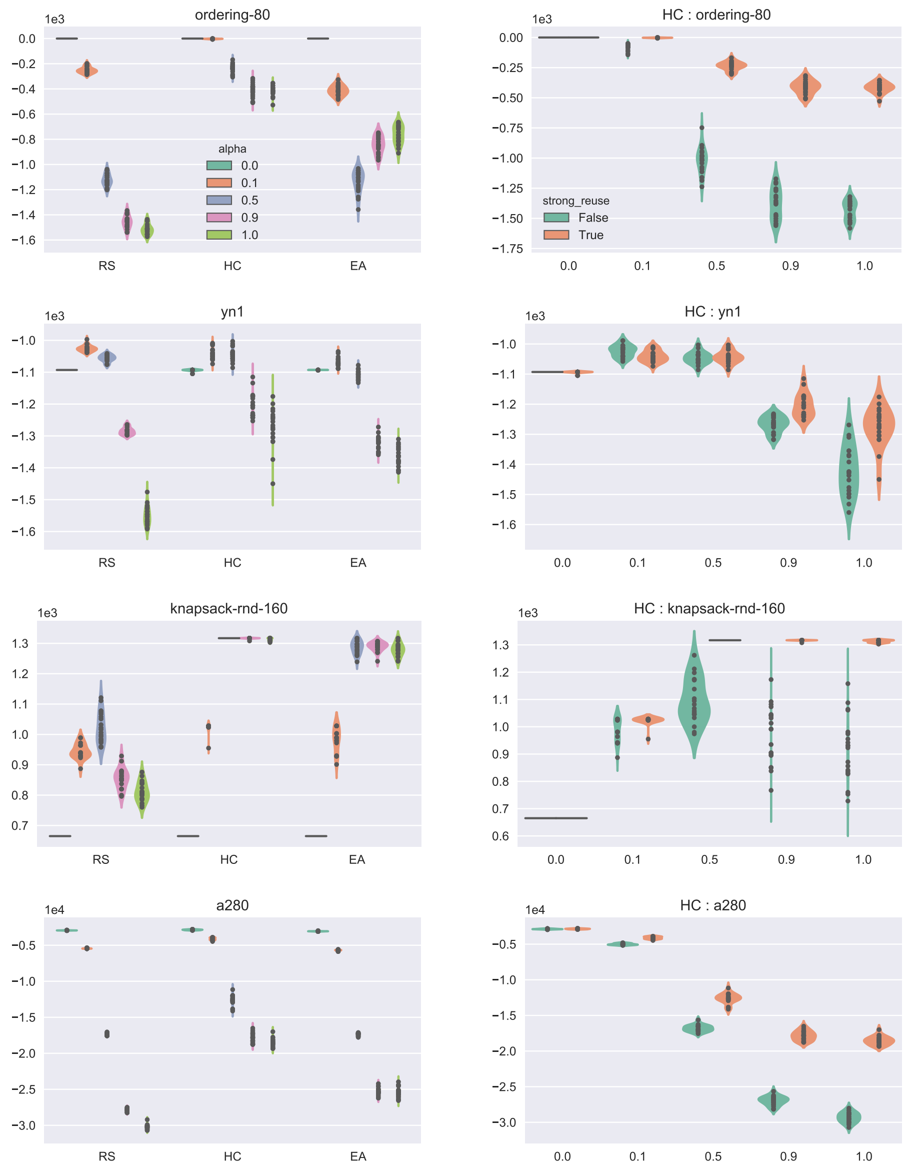

Fig. 2: Results for one instance of each problem type. Top to bottom: Ordering, JSSP, Knapsack, TSP. Left: strong re-use only; analysis by solver and α. Right: HC only; analysis by α and strong versus weak re-use. For all problems, higher values are better (closer to zero, for negative values). Horizontal axes (fitness) scaled by 10^3 or 10^4 as shown.

heuristic alone (with equivalent effort). The advantage is clear-cut. However, an EA does not improve performance further. (4) The other main result is that on the Knapsack and JSSP problems, an intermediate level of greediness ($\alpha = 0.5$) tends to outperform highly greedy construction (small α) or highly randomised construction (large α). For the Ordering problem, a purely greedy approach ($\alpha = 0.0$) solves the problem perfectly. This is the expected result since Ordering is unimodal. It is more surprising that a greedy approach does well on TSP.

Results 1-3 are clear-cut and quite consistent across problems and instances. Result 4 is more problem-dependent, as expected. Perhaps the best setup for difficult problems is the HC solver with $\alpha = 0.5$. For this setup, we give results across all problems in Table 2.

Table 2: Problem (instance), mean objective value and standard deviation, and elapsed time (in seconds), for weak and strong re-use methods. Results are shown for the HC metaheuristic and $\alpha = 0.5$ only. For all problems, a higher objective value is better (closer to zero, for negative values), and a lower time.

	Weak reuse			Strong reuse		
Inst.	Obj.	sd	T(s)	Obj.	sd	T(s)
Ordering						
10	0	0	5	0	0	4
20	-5	6	10	0	0	8
40	-175	26	22	-18	8	19
80	-1018	110	63	-238	36	53
160	-5138	405	190	-1722	178	203
320	-24088	1444	719	-9020	670	736
JSSP						
abz5	-1277	21	77	-1270	19	81
abz6	-1008	19	78	-1000	19	77
abz7	-774	12	322	-783	15	325
abz8	-795	19	324	-793	18	328
abz9	-817	13	323	-817	11	317
yn1	-1046	20	487	-1043	20	469

	Weak reuse			Strong reuse		
Inst.	Obj.	sd	T(s)	Obj.	sd	T(s)
Knapsack						
10	331	0	3	331	0	3
20	460	0	3	460	0	4
40	677	4	6	680	0	6
80	839	45	9	894	0	11
160	1096	81	15	1317	0	23
320	1385	146	30	2015	7	59
TSP						
att48	-64720	3891	28	-46665	2380	24
berlin52	-15564	790	32	-11063	477	27
eil101	-1861	56	88	-1263	79	78
u159	-206070	4381	186	-136750	7680	167
a280	-16806	464	546	-12612	610	506
rat575	-61021	1070	2571	-49457	1187	2448

5.4 Discussion

We have seen that performance of hill-climbing is better than the evolutionary algorithm, overall. Two main hypotheses can be suggested to explain this result.

(1) Perhaps our crossover operator is disruptive, i.e. despite the improved repair mechanism, it fails to retain enough good information from the parents and recombine it in a way that it remains “good”. The PTO crossover operator is a uniform crossover on the dictionary of trace entries. This choice was made because a one-point crossover on trace entries is difficult to define for the general case. However, there are some possibilities to do so, or to do so for the special case of GRASP-like generators. These will be considered in future work.

(2) When hill-climbing does well, it suggests that the landscape is somewhat unimodal. Thus, we may ask: what is the effect on the fitness landscape of using "smart" constructive heuristics as PTO generators? Our speculative answer involves seeing constructive heuristics as epigenetic, developmental processes. In epigenetics, the development of an individual from genotype to phenotype is seen as a process, not an instantaneous step, and it has its own type of optimisation behaviour. The metaphor of rolling downhill – already familiar in the fitness landscape – applies also during development of a single individual. Waddington argued [14] that multiple starting points (genomes) can lead, in some epigenetic landscapes, to a similar end-point (phenotype). This occurs if there are "valleys" or "basins of attraction" in the epigenetic landscape. It is given the name *canalisation*. It gives a form of robustness – a good phenotype can be achieved despite noise in the genotype and environment. We can see constructive heuristics as developmental processes with a canalisation effect. Suppose we have a genotype which gives the optimum phenotype to a TSP, and some noise is added to the genotype, altering one of the edges which is constructed early in the development process. The constructive heuristic will make subsequent choices which are heuristically-guided good ones, and may replicate the effects of those made in the previous individual, leading eventually to a phenotype which shares many of the original phenotype's properties. The improved repair mechanism introduced in this paper accentuates this effect. This *smooths the fitness landscape*, tending to help hill-climbing to perform well. This allows hill-climbing to perform well. In this context, the highly exploitative hill-climbing approach may outperform the exploration-exploitation trade-off chosen by the evolutionary algorithm.

6 Conclusions and future work

In this paper we have explored the use of "smart", GRASP-like, constructive heuristics as generators in the PTO framework, to solve combinatorial optimisation problems. They are a natural fit, and can be seen as a generalisation of previous work, or in a sense as a replication but with automation. We have introduced an improved PTO trace repair mechanism which runs after each genetic operation, and which gives stronger re-use of genetic material relative to the original. After extensive experimentation, we then have four main results:

1. The novel strong re-use method beats weak re-use;
2. Strong re-use does not incur a penalty in computation time;
3. HC beats RS and EA;
4. Medium levels of greediness in the heuristic are often best for real problems.

There are then many lines of research open for the future. Although for convenience in this paper we have introduced a GRASP-like common format for PTO generators for COP problems, generators going beyond this format are also possible. In fact, the freedom of the user to write or supply a generator in any format is a claimed strength of PTO, and in future work novel generators which do not emulate GRASP in this way will be introduced.

If we see the GRASP restricted candidates list as a stepped-uniform distribution on the remaining items ordered by their costs, then we can consider generalising by plugging in a different distribution, such as the triangular distribution suggested by Juan et al. [4]. There is also the possibility of using α to give a threshold on *rank of cost* as opposed to a threshold on *cost*.

The current PTO crossover is a uniform crossover, but a one-point crossover can also be defined, either for the special case of GRASP or for the general case.

Acknowledgements

Thanks to Carlos Fonseca for discussion and to anonymous reviewers. This work was carried out while JMcD was at University College Dublin.

References

1. Moraglio, A., McDermott, J.: Program trace optimization. In: International Conference on Parallel Problem Solving from Nature. pp. 334–346. Springer (2018)
2. Feo, T.A., Resende, M.G.: Greedy randomized adaptive search procedures. Journal of Global Optimization 6(2), 109–133 (1995)
3. McDermott, J., Carroll, P.: Program optimisation with dependency injection. In: Krawiec, K., et al. (eds.) EuroGP. pp. 133–144. Springer (2013)
4. Juan, A.A., Faulin, J., Ferrer, A., Lourenço, H.R., Barrios, B.: MIRHA: multi-start biased randomization of heuristics with adaptive local search for solving non-smooth routing problems. TOP 21(1), 109–132 (2013)
5. 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)
6. Ahuja, R.K., Orlin, J.B., Tiwari, A.: A greedy genetic algorithm for the quadratic assignment problem. Computers & Operations Research 27(10), 917–934 (2000)
7. Bean, J.C.: Genetic algorithms and random keys for sequencing and optimization. ORSA journal on computing 6(2), 154–160 (1994)
8. 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)
9. Moraglio, A.: Towards a geometric unification of evolutionary algorithms. Ph.D. thesis, University of Essex (2008)
10. Jones, T.: Evolutionary Algorithms, Fitness Landscapes and Search. Ph.D. thesis, University of New Mexico, Albuquerque (1995)
11. J. Adams, E.B., Zawack, D.: The shifting bottleneck procedure for job shop scheduling. Management Science 34(391-401) (1988)
12. Yamada, T., Nakano, R.: A genetic algorithm applicable to large-scale job-shop instances. In: Manner, R., Manderick, B. (eds.) Parallel problem solving from nature 2. pp. 281–290. North-Holland, Amsterdam (1992)
13. Bierwirth, C., Kuhpfahl, J.: Extended GRASP for the job shop scheduling problem with total weighted tardiness objective. European Journal of Operational Research 261(3), 835–848 (2017)
14. Waddington, C.H.: Canalization of development and the inheritance of acquired characters. Nature 150(3811), 563 (1942)