Iterative Cartesian Genetic Programming: Creating general algorithms for solving Travelling Salesman Problems

Patricia Ryser-Welch, Julian F. Miller, Jerry Swan and Martin A. Trefzer

The University of York

Abstract. Evolutionary algorithms have been widely used to optimise or design search algorithms, however, very few have considered evolving iterative algorithms. In this paper, we introduce a novel extension to Cartesian Genetic Programming that allows it to encode iterative algorithms. We apply this technique to the Traveling Salesman Problem to produce human-readable solvers which can be then be independently implemented. Our experimental results demonstrate that the evolved solvers scale well to much larger TSP instances than those used for training.

1 Introduction

Designing effective search algorithms for difficult problems has long been an intensive field of study in computer science [1]. Evolutionary algorithms have been used to optimise or design search algorithms and it is typical for such algorithms to operate on a human-designed template in which new operations are generated at fixed points in the template. Very few have evolved loop-based control flow and attempted to answer John Koza's question: *"Is it possible to automate the decision about [...] the particular sequence of iterative steps in a computer program?"* [2]. Such control flow can be implemented either via iteration or recursion. Recursive approaches to various problems of program induction have been presented in Yu and Clack [3] and Alexander [4] but we are not aware of any direct applications to search problems.

In this paper, we introduce a novel and extended form of a well-known graph-based form of Genetic Programming, Cartesian Genetic Programming (CGP), to encode iterative algorithms. The original motivation for using hyper-heuristics is that they do not require skilled practitioners. The use of CGP is significant in this regard, since it doesn't require knowledge of specialized bloat-handling techniques. We apply this technique to the well-known Traveling Salesman Problem (TSP), a problem domain which has been extensively studied in mathematics and computer science and is often used to benchmark new techniques. Various search algorithms such as Iterative Local Search, Memetic Algorithms, Particle Swarm Optimization, Ant Colony algorithms and other novel metaheuristics have solved a wide variety of TSP benchmark instances that have often less than 2000 cities[6–8]. We use iterative CGP to generate new TSP algorithms:

the quality of an algorithm during the evolution process is determined by a small training set of TSP instances ranging between 200 and 800 cities. Subsequently algorithms are validated on larger unseen instances varying in level of complexity. The easiest instances have 30 cities but the most challenging contain just under 25,000 cities. The contributions of the work presented here are three-fold:

1. Previously CGP has encoded as a *data-flow diagram*, in which information flows through the graph from inputs to outputs [9]. For this work, CGP has been adapted to provide a *flowchart* which represents an iterative algorithm using *"Decision"*, *"Process"* and *"Terminal"* elements.
2. Both instruction ordering and iterative control flow are evolved: groups of instructions can be repeated. The resulting algorithms find good solutions to unseen TSP instances.
3. CGP is used to generate hybrid search algorithms using combinations of local search and binary crossover. We evolve human-readable algorithms that can reach optimal TSP solutions and can be directly translated into other programming languages.

2 Optimisation of algorithms

The goal is to improve some aspect of an algorithm in order to solve problems more efficiently or with fewer resources. It is useful to distinguish here between two distinct search spaces: we use the term *problem solutions* to refer to elements of the *underlying* problem space (e.g. permutations in the case of the TSP) problem and the *algorithm solution* to the generated algorithms. Early approaches in this area have been referred to as *"automatic programming systems"* [2]. The problem solutions are obtained using a solver generated by a technique such as Genetic Programming (GP). Research in this area [2, 10] has largely focused on results in terms of the performance on the underlying search problem, rather than human-readability of the algorithms themselves. More recently, a variety of search methods have been used to automatically configure algorithms via parameter optimization [11, 12]. The latest approaches in this area are increasingly general (e.g. [13, 14]), allowing entire component configurations to be treated as a parameter hierarchy. However, parameter tuning is still currently a rather limited way to optimise an algorithm. The evolution of iterative control flow with CGP offers a more general approach and additionally provides human-readable output without the need for explicit parsimony pressure to combat expression bloat.

A graph-based form of Genetic Programming (GP) has automated machine code with basic loops. This technique was promising, but it has not yet been applied to higher level programming languages [5].

The generation of search algorithms can be considered within the context of *hyper-heuristics*, which are defined as "a search method or learning mechanism for selecting or generating heuristics to solve computational search problems" [15]. Hyper-heuristics can be *selective* or *generative*. The popular conception of selective hyper-heuristics is exemplified by the HYFLEX framework, in which the selection is performed (via an opaque *domain barrier* [15]) from a collection of pre-existing operators (heuristics). There are number of alternative hyper-heuristic frameworks to HYFLEX (e.g. [16, 17]), including selective frameworks with a less-restrictive notion of the domain barrier [18, 19]. In contrast to the selective approach, generative hyper-heuristics create new operators [20] and tend to use nature-inspired mechanisms (such as Learning Classifier Systems or GP) to discover better quality algorithms. This can result in algorithms capable of addressing an entire class of problems [21]. What both approaches have in common is to combine human-designed search components in new ways with the goal of outperforming any individual component. A detailed review of the state-of-art in hyper-heuristics can be found [15, 22]. Ryser-Welch et al [23] and Ross [24] complement these reviews by focusing specifically on hyper-heuristic frameworks.

The automated design of sizeable algorithms without any external help is beyond the state-of-the art. Suitably expressive algorithms may never terminate or have over-long computations. It is therefore useful to consider an algorithm search-space as consisting or both feasible and infeasible algorithms [2]. When algorithm design is automated, these unwanted occurrences are usually prevented via some forms of constraint. For example, [25–28, 13] restrict the structure of an algorithm to prevent unfeasible sequences being discovered. Syntactic rules control the pattern of the primitives that are combined to form the algorithm-solutions. The body of a loop, the initialisation step, the update step, and sometimes the termination criteria are influenced by evolution. For instance, [29] evolves the body of the loop of ant algorithms using Grammatical Evolution; these algorithms are human readable and strictly restricted to the syntactic rules. Although some good results have been obtained from many of these techniques, the resulting algorithms can be very challenging to understand. In some cases, the chosen algorithm representation (e.g. GP tree) can cause bloat during the evolution, resulting in very large complex algorithms. Other algorithm generation schemes do not express all three elements of a looping construct or restrict the algorithm-solutions to a limited collection of primitives. In the next section, we describe how an extension of CGP can take advantages of properties of this graph-based GP, to relax strict syntactic rules to produce compact iterative algorithms.

3 Iterative Cartesian Genetic Programming

We describe an extension of CGP to the generation of iterative algorithms. In contrast to 'traditional' GP, which operates on expression trees, CGP uses a directed acyclic graph. An integer-based encoding scheme is used to define a two-dimensional grid, representing the adjacency matrix of a set of user-defined nodes. A characteristic of CGP is that it encodes both active and inactive nodes. Inactive nodes are nodes that are not on any paths connecting inputs to outputs; in Fig. 1 the output connects to node 4, but node 5 is inactive (shaded in gray). Nodes may be activated or deactivated during evolution. Each node has a *function gene* indexing a primitive operation in a user-specified look-up table. Nodes are connected in a feed-forward manner from either a previous node or a program input, using at least one *node input.* The *output genes* can connect to any previous nodes or program inputs. The identification of all the active nodes starts from the nodes pointed to by the output genes and continues until an input is reached. All the active nodes are then processed from left to right. In Fig. 1, the decoding step identifies the active nodes 1,2,3 and 4; these are executed in ascending order (i.e. 1,2,3 and 4).The CGP-graph has a fixed length, but the number of active nodes can be anything from zero to the number of nodes (see Fig. 1). Unlike other Genetic Programming techniques, CGP has been shown not to bloat [30, 31].

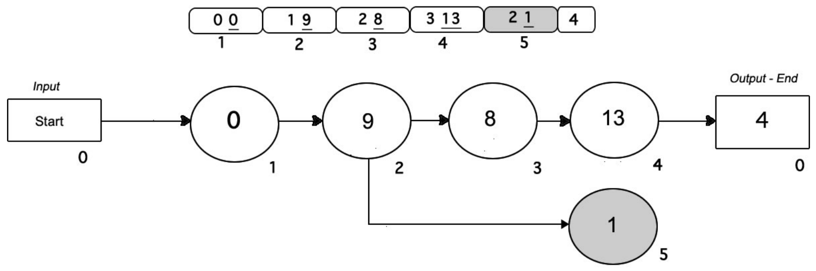

Fig. 1. This CGP graph encodes an algorithm made of 4 primitives, with 1 input and 1 output. All these active nodes (in white) constitute the "process" elements of the flowchart.

A directed graph can fully encode an iterative program: we call this an iterative CGP graph. This allows a cycle to be formed so that loops are possible. The stopping criterion, the iterative update step and the body of the loop are all alterable by evolution. To accomplish this it is necessary for every node to have at least four different types of genes:

Feed-forward connections are standard feed-forward CGP connection genes. They connect the input to the current node with either a previous node or a program input. We refer to these nodes as *process nodes* as they represent a process element of the flowchart.

Branching connections can point to a previous node, a program input, itself, or a suitable subsequent node. They are connection genes which determine the boundaries of the body of a loop and split a CGP graph into smaller sub-sequences. The first operation in the sub-sequence is the operation determined by the function gene of the current node. The last operation in the sub-sequence is the operation defined by the node pointed to by the branching gene of the current node. In these cases we refer to the current node as a *decision node* by analogy with a node in a flowchart which represents a "decision" element.

Function genes are as in standard CGP and encode a primitive operation. Their values correspond with a function look-up table.

Condition genes represent the stopping criteria of loops. A condition look-up table provides a set of Boolean primitives, these indicate whether a loop exits (and control subsequently moves to the next node following the last loop node) or continues to execute the next node inside the loop.

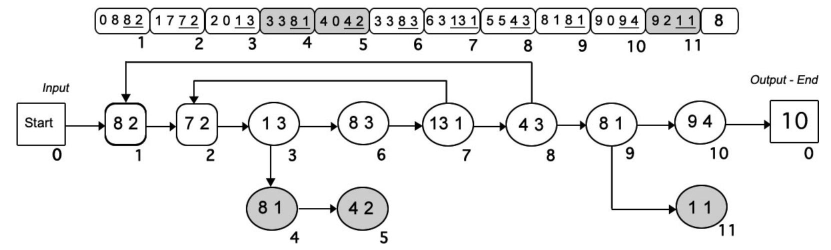

Fig. 2. An iterative CGP graph encodes an algorithm made of 8 primitives starting at node 1 and ending at node 10. Nodes no 4, 5 and 11 are non-coding genes, these are shaded in gray.

The distinction between decision and process nodes plays an important role in the decoding process of an iterative CGP graph. First all the active nodes are identified (by working backwards from the outputs), then the decision nodes are placed so that branching can happen during the decoding process; the index of the decision node is inserted after the last active node of the body of the loop. For example in fig. 2, all the active nodes are executed in the following order: **1**,**2**,3,6,7,**2**,8, **1**,9,10. Also it is assumed that upon the second call of node 1, condition no 2 is also met, causing program execution to move to the next node (9) after the loop terminates at node (8).

1. When an iterative CGP graph does not encode any loops the value of any branching gene is free to point to any nodes and program inputs.
2. For any nodes inside an existing loop, their branching genes can only connect to node with a higher index that is inside the current loop or any previous nodes and program inputs. In fig. 2, the branching gene of nodes 3, 4, 5, and 6 can be valid if its value is lower than the index of the node. It can also point to the right to a node with an index lower than 7.
3. For any nodes outside an existing loop, their branching genes can connect to a node that is outside any existing loops. A valid value for the branching gene of node 1 can only point to the input or nodes 9, 10 or 11

CGP generally uses a (1+λ) evolutionary strategy (this is shown in Algorithm 1). Either point or probabilistic mutation is traditionally used, crossover is not. If an offspring has an equal or better fitness than the parent it is promoted to the next generation [9]. Two basic grammatical rules ensure that either only

Algorithm 1 The $(\mu + \lambda)$ evolutionary strategy used by both versions of CGP. Often there is one parent (μ) and four offspring (λ)

```
 1: Randomly generate individual $i$
 2: Select the fittest individual, which is promoted as the parent (algorithm)
 3: while solution is not found or the generation limit is not reached do
 4:     Mutate the parent to generate offspring
 5:     Generate the fittest algorithm using the following rules:
 6:     if offspring has a better or equal fitness than the parent then
 7:         offspring is chosen as fittest
 8:     else
 9:         The parent remains the fittest
10:     end if
11: end while
```

nested loops are created or new loops do not overlap. This is ensured during the initialization of the iterative CGP population and the mutation of parents to produce new iterative CGP offspring.

4 Discovery of Iterative TSP solvers

The goal of these experiments is to gain insight into how hybrid metaheuristics can be discovered with iterative CGP. We ran our generated algorithms on the TSP instances from well-known benchmarks[1]. The settings of Iterative CGP for the all the tests are given in Table 4. Our proposed method evolves merely a sequence of heuristics, but repeated sub-sequences (or loops). At the end of this process, a generated algorithm can then be extracted, and if desired, re-coded in some conventional programming language. Subsequently, the generated algorithms are evaluated in an independent process using an unseen test. The upper-level process is problem-domain independent and the specialised TSP heuristics used in the lower-layer are described in the next sub-section.

4.1 The Travelling Salesman Problem

This combinatorial problem seeks the shortest possible route that visits each of a list of cities exactly once and returns to the first city, i.e. a Hamiltonian cycle on n cities. A route is are referred to as *a tour* and is typically represented as a permutation on n elements. The problem is naturally represented with a complete weighted graph $G = (V, E)$. Each edge of E defines the link connecting the cities and their weight indicates the distance between two cities u and v; these are retrieved by the distance function $d(u, v)$. The length of a tour is given

[1] d1291, u2152, usa23505 and d18512 are benchmarks from the well-known TSPLIB. The remaining instances are benchmarks from real-life geographical data ; these are wi29, dj38, qa194, zi929, ca4663, ym7663, ja9874, gr9882, sw24978. All these instances can be found at http://www.math.uwaterloo.ca/tsp/world/countries.html and http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html

Table 1. Experimental parameters for Iterative CGP

Parameter	**Value**
Length (no of nodes)	300
Levels-back (no of nodes)	100
Levels-forward (no of nodes)	100
Program inputs	1
Program outputs	1
$\mu + \lambda$	1 + 1
Mutation Rate	0.10
Generations	1500

by the sum of its edge weights. For training purposes, we wish to combine the results across multiple problem instances of different sizes, so we therefore use the popular normalized measure of *relative error* as the fitness value of a TSP solution. This uses the best tour length that is known *a priori*, using the formula $(\text{tourlength} - \text{knownoptimum}) / \text{knownoptimum}$.

A wide variety of TSP-specific operators have been examined in the literature. *Lin-Kernighan heuristics* are the most studied methods for solving this problem effectively. In these, k edges are deleted and subsequently re-assembled to construct the sub-paths of a new tour with a lower minimum weight [32, 33]. Traditionally those are often referred as *k-opt heuristics.* For example, when $k = 2$, edges between two pairs of cities are reconnected in a different way to obtain a new shorter tour. Solutions obtained from genetic operators (e.g. PMX, as below) can be further improved by local search [6]. Operators given below include those taken from recent work which merged a local search operator with genetic search [7], along with well-known crossover and mutation operators. Table 2 shows all the primitives we will be using for our experiments.

- **Order Based Crossover (OX)** chooses a subtour in one parent and imposes the relative order of the cities of the other parent [34].
- **Partially-Mapped Crossover (PMX)** copies an arbitrary chosen subtour from the first parent into the second parent, before applying minimal changes to construct a valid tour [35, 36].
- **Voting Recombination Crossover (VR)** uses a randomized Boolean voting mechanism to decide from which parents each city is copied from [37].
- **Subtour-Exchange Crossover (SEC)** preserves randomly selected subtours from both parents to construct one new offspring [38].
- **Edge-Assembly Crossover (EAX)** assembles sub-tours together by building intermediary permutations. Each of these permutation are repeatedly minimised [39]
- **Stem-and-cycle ejection Crossover (SCX)** unusually asexually reproduces before improving the child solution using a "Stem-And-Cycle" Local Search approach [7].
- **Insertion Mutation (IM)** moves a randomly chosen city in a tour to randomly selected place [40].
- **Exchange Mutation (EM)** swaps two randomly selected cities [41].

– **Scramble Mutation (SM)** rearranges a random subtour of cities [34]. Hyflex applies this mutation operator on a subtour and on the whole tour.
– **Simple Inversion Mutation (SIM)** implements a 2-opt Lin-Kernighan heuristic.
– **3-point Inverstion Mutation (3IM)** implements a 3-opt heuristic [7].

4.2 Automatic Design of hybrid metaheuristics

In our experiments, operators provided by the Hyflex cross-domain hyper-heuristic framework[2][3] were chosen (see Table 2) as described in previous research [42]. The parameters for our generated metaheuristic were set to 2 offspring, 2 parents, and a maximum of 500 evaluations. The search depth for local search controls the number of iterations used in a local search operators; following preliminary experiments, it was set to 0.89. The intensity of mutation (which defines the numbers of cities shuffled in a permutation) was similarly set to 0.8. These standard parameters tune the performance in general of local searches and mutation operators for any problem domain provided in Hyflex. These parameters are standard choices for the Hyflex system.

Table 2. Function set: List of TSP heuristics used as primitives.

Index	TSP heuristics
0	InsertionMutation()
1	ExchangeMutation()
2	ScrambleWholeTourMutation()
3	ScambleSubtourMutation()
4	SimpleInversionMutation()
6	2-OptLocalSearch()
7	Best2-OptLocalSearch()
8	3-OptLocalSearch()
9	OrderBasedCrossover()
10	PartiallyMapCrossOver()
11	VotingRecombinationCrossOver()
12	SubtourExchangeCrossover()
13	ReplaceLeastFit()
	SelectParents()
15	RestartPopulation()

Table 3. Condition set: Boolean primitives chosen for the stopping criterion.

Index	TSP heuristics
1	Number of evaluations > 0
2	The evaluations fall in the first half of the evolution
3	The evaluations fall in the second half of the evolution
4	Number of evaluations > 0 or no improved neighbouring solutions are available

Tables 2 and 3 provide the heuristics and termination criterion for the generated TSP solvers. For Conditions 2 and 3, the evolution is split into two stages: each phase uses half the available evaluations Condition 4 stops the search when all the evaluations have been used or no shorter tour has been found in the last 50 generations.

[2] http://www.asap.cs.nott.ac.uk/external/chesc2011/
[3] http://www.hyflex.org/chesc2014/

A predefined template (Algorithm 2) guarantees that the generated algorithm initializes and evaluates a population of permutations, before selecting parents (lines 1 to 3 of algorithm 2). The code in lines 4 to 21 execute the iterative algorithm defined by the active nodes of an iterative CGP graph. The last line enforces that shorter tours are promoted in the population before the algorithm ends its run (see line 22 of Algorithm 2). The remaining lines apply the heuristics of the active process and branching nodes (see lines 4 and 23). The 'goto' statements can either jump to the start of a loop, the next heuristics (if there is one) or the first when the stopping criterion is met, or to the first heuristic of the metaheuristic.

Algorithm 2 Template for a hybrid meta-heuristic, with main structure (line 4 to 21) being evolved by an Hyper-Heuristic algorithm.

```
 1: $p_0 \leftarrow GenerateInitialSolution()$;
 2: $p_0 \leftarrow EvaluatePopulation()$;
 3: $t \leftarrow SelectParents()$;
 4: {Start of code generated by Iterative CGP}
 5: goto the first active node
 6: while Not the end of of evolved sequence of heuristics do
 7:    if The current node is a process node then
 8:       Apply the heuristic on $t$ or $p$
 9:       goto the next active node
10:    else
11:       if the current node is a decision node and the last node a loop then
12:          goto the first node of the loop
13:       end if
14:       $StoppingCriterion \leftarrow$ apply condition of the currentNode
15:       if $StoppingCriterion$ is false then
16:          Apply the heuristic on the $t$ or $p$
17:          Go to the next active node
18:       else
19:          Go to the first node after the loop
20:       end if
21:    end if
22: end while
23: {End of code generated by Iterative CGP}
24: $p \leftarrow replaceLeastFit(t, p)$
```

The testing phase was performed on TSP lib instances pr299, pr439 and rat783[4] having 299, 439 and 783 cities respectively. It is well-known that hyper-heuristic evaluation is computationally expensive, so this small subset of instances was chosen for their diverse clustering of cities. The fitness measure used for generated solvers during the training phase is obtained by averaging the relative error values obtained for these instances; each run had a budget of 500

[4] http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

evaluations. The fitness measure for testing is the relative error of the instance under test.

5 Experimental results

Algorithms 3 and 4 show the best iterative algorithms evolved by iterative CGP. As discussed above, the first three lines and the last instruction of these algorithms are part of the template described in Algorithm 2. The remaining instructions of the algorithms were generated during the decoding phase of the iterative CGP graphs.

Algorithm 3 resembles a memetic algorithm; evolution has re-discovered a similar algorithm to the most effective sequential algorithm evolved in our previous research [42]. In fact, lines 6 to 8 cancel out the effect of restarting the population p, if no shorter tour has been found in 50 generations, then the population is initialized again. However the newly-created TSP solutions are replaced immediately by the offspring (t); if and only if the length of their tour is shorter than the new generated individuals in population p.

Algorithm 4 applies two loops that are carried out during the first half of the evolution and fewer evaluations are required. The first loop can be perceived as redundant, but its purpose is to execute only once two Lin-Kernighan operators; one before and one after searching more thoroughly. This occurs in a nested loop, constructed with the Best2-OptLocalSearch() to reduce the length of the tour, before applying ExchangeMutation heuristic. This heuristic should prevent the 3-opt-LocalSearch() finding no available neighbouring solutions and then finding the same local optima again. These new offspring then replace the least fit individuals in the population p.

Algorithms 3 and 4 were translated from their iterative CGP graph form and coded as TSP solvers in the programming language Java$^{\text{TM}}$. The same primitives were retained, but a different set of benchmarks was used for testing. As observed above, hyper-heuristics are notoriously computationally expensive and a representative subset (having different distributions of cities) was chosen so to allow the experiments to be performed within a reasonable time. After some initial experiments, we set our number of evaluations to 6000, so that the search can be performed in a reasonable amount of time. We are aware the search is likely to be short, however, it would be just a matter of increasing the evaluations to solve more instances. For direct comparison, the best performing sequential TSP solver obtained from previous research [42] and the memetic algorithm due to Özcan [43] were also coded in Java. Both algorithms apply the same set of operators, with statistical comparison provided in Table 4, which gives the mean of the best obtained tour lengths over 30 runs and the mean relative error (and its standard deviation) from the best-known tour length.

We can see in Fig. 3 the evolution has constructed algorithms that enhance the strength and ameliorate the weaknesses of the heuristics and conditions listed in tables 2 and 3. Both algorithms start their search with 3-Opt-LocalSearch, to reduce dramatically the length of the tours generated during the initialization process. In Fig. 3, the search descends sharply from around a relative error approximately around 0.20 from the known minimum to a relative error around 0.11 from generation 0 to 1. Hence it appears that evolution has rediscovered some elements of the template applied in Ryser-Welch et al. [42].

Algorithm 3 This algorithm is the outcome of applying algorithm 2 on the iterative graph in fig 2

```
1: $p_0 \leftarrow GenerateInitialSolution()$;
2: $p_0 \leftarrow EvaluatePopulation()$;
3: $t \leftarrow SelectParents()$;
4: {Start of code generated by Iterative CGP}
5: while Number of evaluation left $> 0$ do
6:     $t \leftarrow$ 3-OptLocalSearch($t$)
7:     $p \leftarrow$ restart population
8:     $p \leftarrow$ replaceLeastFit($t$,$p$)
9:     $t \leftarrow$ SelectParents()
10:    $t \leftarrow$ ExchangeMutation($t$)
11: end while
12: {End of code generated by Iterative CGP}
13: $p \leftarrow replaceLeastFit(t, p)$
```

Algorithm 4 This algorithm is the outcome of applying algorithm 2 on the iterative graph in fig 2

```
1: $p_0 \leftarrow GenerateInitialSolution()$;
2: $p_0 \leftarrow EvaluatePopulation()$;
3: $t \leftarrow SelectParents()$;
4: {Start of code generated by Iterative CGP}
5: while The evaluations fall in the first half of the evolution (node 1) do
6:     $t \leftarrow$ 3-OptLocalSearch($t$) (node 1)
7:     while The evaluations fall in the first half of the evolution (node 2) do
8:         $t \leftarrow$ Best2-OptionLocalSearch($t$) (node 2)
9:         $t \leftarrow$ ExchangeMutation($t$) (node 3)
10:        $t \leftarrow$ 3-OptionLocalSearch($t$) (node 4)
11:        $p \leftarrow$ replaceLeastFit($t$, $p$) (node 5)
12:        $t \leftarrow$ SelectParents() (node 5);
13:    end while
14:    $t \leftarrow$ SimpleInversionMutation($t$) (node 6)
15: end while
16: $t \leftarrow$ 3-OptionLocalSearch($t$) (node 7)
17: $t \leftarrow$ OrderBaseCrossover($t$) (node 8)
18: {End of code generated by Iterative CGP}
19: $p \leftarrow replaceLeastFit(t, p)$
```

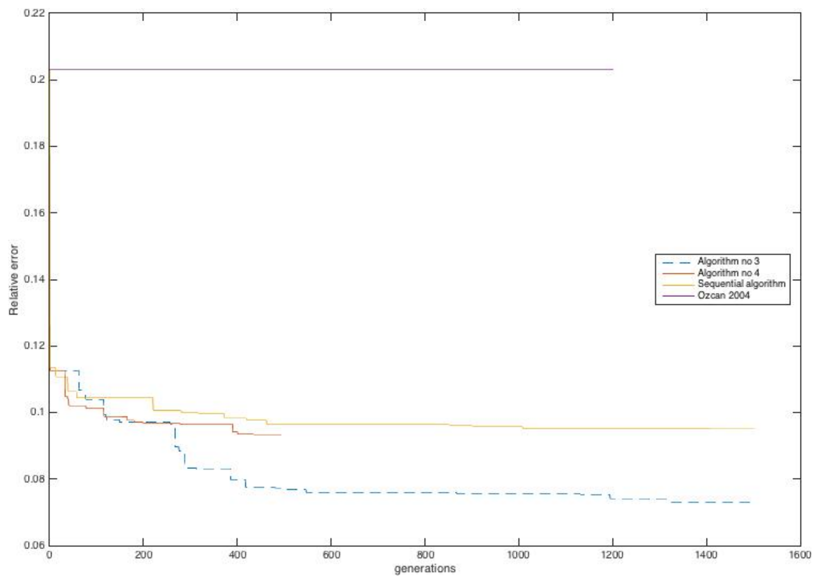

Fig. 3. A comparison of the four algorithms during the search for an optimum tour for the benchmark D1219

The iterative and the sequential algorithms achieved the best average fitness overall with a small standard deviation. For most benchmarks, the iterative algorithm scales well, finding good solutions to some benchmarks larger than the instances used during the training phase. Algorithm 3 has found the best solutions for the instances d1291, ym7663, usa13509 and sw24978; these instances are particularly hard to solve. Algorithm 4 uses many fewer evaluations; the termination criterion stop the loop when half of the evaluations have been used. The algorithm has found better tours than the sequential algorithms for the TSP instances d1291, zi929, ja9874 and usa13509.

We applied the Mann-Whitney U nonparametric test (for $p = 0.05$) to all pairs of algorithms, the results of which are in Table 5. The symbol = indicates that there is no significant difference between (the results of) Alg A and Alg B, $>$ denotes that Alg A is significantly better than Alg B and $<$ that Alg A is significantly worse than Alg B. In general, Algorithms 3 and 4 have found better or similar tours than related previous work [42]. Our approach has generated some iterative metaheuristics that have higher scalability than the best performing sequential TSP solver obtained from this previous research.

Table 4. Mean values of TSP solutions on 30 independent runs. The optimum value was either found by using Concorde or Lin-Kernighan

TSP Instance	**Known Optimum** [5]	**Iterative Alg no 3**	**Iterative Alg no 4**	**Ryser-Welch 2015**	**Ozcan 2004**
wi29	27,603	27,603	27,603	27,603	30,704
	relative error	0.000	0.000	0.000	0.001
	standard dev.	0.000	0.000	0.000	0.068
dj38	6,656	6,656	6,656	6,656	7,044
	relative error	0.000	0.000	0.000	0.002
	standard dev.	0.000	0.000	0.000	0.112
qa194	9,352	9,369	9,560	9,378	9,361
	relative error	0.002	0.022	0.004	0.001
	standard dev.	0.001	0.008	0.001	0.021
zi929	95,345	96,472	99,996	97,283	118071
	rel. error	0.011	0.048	0.019	0.240
	standard dev.	0.004	0.009	0.004	0.019
d1291	50,801	56,264	58,562	58,562	58,750
	relative error	0.081	0.112	0.121	0.200
	standard dev.	0.008	0.009	0.029	0.011
u2152	64,253	67,064	69,827	68,732	78,692
	relative error	0.043	0.086	0.069	0.223
	standard dev.	0.006	0.014	0.017	0.015
ca4663	1,209,319	1,277,495	1,331,639	1,304,901	1,547,992
	relative error	0.056	0.101	0.079	0.284
	standard dev.	0.004	0.024	0.015	0.022
ym7663	238,314	260,199	267,905	266,738	266,738
	relative error	0.091	0.124	0.119	0.281
	standard dev.	0.021	0.023	0.033	0.022
ja9874	491,924	533,304	555,201	564,581	625,035
	relative error	0.084	0.128	0.147	0.276
	standard dev.	0.018	0.033	0.046	0.011
gr9882	300,899	327,118	334,135	334,642	383087
	relative error	0.087	0.110	0.112	0.273
	standard dev.	0.018	0.022	0.021	0.023
usa13509	19,982,859	21,083,162	21,465,644	21,320,901	25,109,189
	relative error	0.055	0.074	0.066	0.251
	standard dev.	0.007	0.010	0.011	0.012
d18512	645,238	671,752	676,486	674,104	790,769
	relative error	0.041	0.048	0.044	0.225
	standard dev.	0.003	0.002	0.002	0.013
sw24978	855,597	912,915	927,663	928,355	1,075,056
	relative error	0.066	0.084	0.085	0.256
standard dev.	0.008	0.012	0.012	0.011	

Table 5. Comparison of TSP solvers via Mann-Whitney U, $p = 0.05$.

Instance	**Alg3 vs Alg4**	**Alg3 vs Ryser-Welch [42]**	**Alg4 vs Ryser-Welch [42]**
wi29	=	=	=
dj38	=	=	=
qa194	>	=	>
zi929	>	>	<
d1291	>	>	=
u2152	>	>	>
ca4663	>	>	<
ym7663	>	>	<
ja9874	>	>	>
gr9882	>	>	=
usa13509	>	>	<
d18512	=	=	<
sw24978	>	>	=

6 Conclusion

We have presented a novel approach to evolving metaheuristics, which generates new metaheuristic variants containing evolved looping constructs. We evolved two novel TSP solvers and applied them to benchmark instances of the Travelling Salesman Problem. We show that not only that the method can produce human-readable algorithms (our sequence of operations was readily re-coded in Java), but it can also rediscover effective algorithms and generate new ones. The results of our experiments are promising: from a small training set, solutions equal or close to the actual known optima have been found for the benchmark instances under test. Our next step will be to apply this type of evolutionary hyper-heuristic to other problem domains as well to generate new hybrid metaheuristics and to demonstrate the generality and scalability of the proposed method. For example, personnel scheduling, vehicle routing and numerical optimisation will be considered with a larger range of instance sizes, allowing the potential of this technique to be fully evaluated.

Acknowledgements

The N8 HPC computer cluster used to host our evolutionary cross-domain hyper-heuristics and test their performance was provided and funded by the N8 consortium and EPSRC (Grant No.EP/K000225/1). The Centre is co-ordinated by the Universities of Leeds and Manchester.

References

1. E. Kant. Understanding and automating algorithm design. *IEEE Transactions on Software Engineering*, SE-11(11):1361–1374, 1985.
2. John R Koza and David Andre. Evolution of iteration in genetic programming. In *Evolutionary Programming*, pages 469–478, 1996.
3. Tina Yu and Chris Clack. Recursion, lambda-abstractions and genetic programming. In Riccardo Poli, W. B. Langdon, Marc Schoenauer, Terry Fogarty, and Wolfgang Banzhaf, editors, *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming*, pages 26–30, Paris, France, 14-15 April 1998. CSRP-98-10, The University of Birmingham, UK.
4. Brad Alexander and Brad Zacher. Boosting search for recursive functions using partial call-trees. In *Parallel Problem Solving from Nature–PPSN XIII*, pages 384–393. Springer, 2014.
5. James Alfred Walker, Yang Liu, Gianluca Tempesti, Jon Timmis, and Andy M Tyrrell. Automatic machine code generation for a transport triggered architecture using cartesian genetic programming. *International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS)*, 3(4):32–50, 2012.
6. Gregory Gutin and Daniel Karapetyan. A memetic algorithm for the generalized traveling salesman problem. *Natural Computing*, 9(1):47–60, 2010.
7. E Kasturi and S Lakshmi Narayanan. A novel approach to hybrid genetic algorithms to solve symmetric tsp. *International Journal*, 2(2), 2014.

8. Nizar Rokbani, Ajith Abraham, and Adel M Alimil. Fuzzy ant supervised by pso and simplified ant supervised pso applied to tsp. In *Hybrid Intelligent Systems (HIS), 2013 13th International Conference on*, pages 251–255. IEEE, 2013.
9. Julian F Miller, editor. *Cartesian genetic programming*. Springer, 2011.
10. Scott Brave. Evolving recusive programs for tree search. 1996.
11. Manuel López-Ibánez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari. The irace package, iterated race for automatic algorithm configuration. Technical report, Citeseer, 2011.
12. Holger H Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012.
13. Manuel López-Ibánez and Thomas Stützle. The automatic design of multiobjective ant colony optimization algorithms. *Evolutionary Computation, IEEE Transactions on*, 16(6):861–875, 2012.
14. Franco Mascia, Manuel López-Ibáñez, Jérémie Dubois-Lacoste, and Thomas Stützle. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Comput. Oper. Res.*, 51:190–199, November 2014.
15. Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.
16. Jerry Swan and Nathan Burles. Templar - A framework for template-method hyper-heuristics. In *Genetic Programming - 18th European Conference, EuroGP 2015, Copenhagen, Denmark, April 8-10, 2015, Proceedings*, pages 205–216, 2015.
17. J. Swan, J. R. Woodward, E. Özcan, G. Kendall, and E. K. Burke. Searching the Hyper-heuristic Design Space. *Cognitive Computation*, 6(1):66–73, 2014.
18. Jerry Swan, Ender Özcan, and Graham Kendall. Hyperion - a recursive hyper-heuristic framework. In Carlos Coello, editor, *Learning and Intelligent Optimization*, volume 6683 of *Lecture Notes in Computer Science*, pages 616–630. Springer Berlin / Heidelberg, 2011.
19. Alexander E.I. Brownlee, Jerry Swan, Ender Özcan, and Andrew J. Parkes. Hyperion^2: A toolkit for {Meta-, Hyper-} heuristic research. In *Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion*, GECCO Comp '14, pages 1133–1140, New York, NY, USA, 2014. ACM.
20. Peter Ross, Sonia Schulenburg, Javier G. Marín-Blázquez, and Emma Hart. Hyper-heuristics: Learning to combine simple heuristics in bin-packing problems. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 942–948, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.
21. Peter Ross. Hyper-heuristics. In *Search methodologies*, pages 529–556. Springer, 2005.
22. Nelishia Pillay. A review of hyper-heuristics for educational timetabling. *Annals of Operations Research*, pages 1–36, 2014.
23. Patricia Ryser-Welch and Julian F Miller. A review of hyper-heuristic frameworks. In *Proceedings of the 50th anniversary convention of the AISB, 1–4 April 2014, London*, 2014.
24. Peter Ross. Hyper-heuristics. In *Search Methodologies*, pages 611–638. Springer, 2014.
25. William Benjamin Langdon. *Genetic programming and data structures*. PhD thesis, University College London, 1996.
26. Gayan Wijesinghe and Vic Ciesielski. Evolving programs with parameters and loops. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.

27. Jan Larres, Mengjie Zhang, and Will N Browne. Using unrestricted loops in genetic programming for image classification. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.
28. Shinichi Shirakawa and Tomoharu Nagao. Graph structured program evolution with automatically defined nodes. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1107–1114. ACM, 2009.
29. Jorge Tavares and Francisco B Pereira. Automatic design of ant algorithms with grammatical evolution. In *Genetic Programming*, pages 206–217. Springer, 2012.
30. Julian Miller. What bloat? cartesian genetic programming on boolean problems. In *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 295–302, 2001.
31. Andrew James Turner and Julian Francis Miller. Neutral genetic drift: an investigation using Cartesian Genetic Programming. *Genetic Programming and Evolvable Machines*, 16(4):531–558, 2015.
32. S Lin and BW Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, 1973.
33. Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.
34. Lawrence Davis et al. *Handbook of genetic algorithms*, volume 115. Van Nostrand Reinhold New York, 1991.
35. David E Goldberg and Robert Lingle. Alleles, loci, and the traveling salesman problem. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, volume 154. Lawrence Erlbaum, Hillsdale, NJ, 1985.
36. John J Grefenstette. Incorporating problem specific knowledge into genetic algorithms. *Genetic algorithms and simulated annealing*, 4:42–60, 1987.
37. H Miihlenbein and J Kindermann. The dynamics of evolution and learning-towards genetic neural networks. *Connectionism in perspective*, pages 173–197, 1989.
38. K Katayama, H Sakamoto, and H Narihisa. The efficiency of hybrid mutation genetic algorithm for the travelling salesman problem. *Mathematical and Computer Modelling*, 31(10):197–203, 2000.
39. Yuichi Nagata and David Soler. A new genetic algorithm for the asymmetric traveling salesman problem. *Expert Systems with Applications*, 39(10):8947–8953, 2012.
40. David B Fogel. An evolutionary approach to the traveling salesman problem. *Biological Cybernetics*, 60(2):139–144, 1988.
41. Wolfgang Banzhaf. The “molecular” traveling salesman. *Biological Cybernetics*, 64(1):7–14, 1990.
42. Patricia Ryser-Welch, Julian F. Miller, and Shahriar Asta. Generating human-readable algorithms for the travelling salesman problem using hyper-heuristics. In *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference*, GECCO Companion ’15, pages 1067–1074, New York, NY, USA, 2015. ACM.
43. Ender Ozcan and Murat Erenturk. A brief review of memetic algorithms for solving euclidean 2d traveling salesrep problem. In *Proc. of the 13th Turkish Symposium on Artificial Intelligence and Neural Networks*, pages 99–108, 2004.