Evolving Fixed-parameter Tractable Algorithms

Stefan A. van der Meer [a] Iris van Rooij [a,b] Ida Sprinkhuizen-Kuyper [a,b]

[a] *Radboud University Nijmegen, Department of Artificial Intelligence*
[b] *Radboud University Nijmegen, Donders Institute for Brain, Cognition and Behaviour*

Abstract

One effective means of computing NP-hard problems is provided by fixed-parameter tractable (fpt-) algorithms. An fpt-algorithm is an algorithm whose running time is polynomial in the input size and superpolynomial only as a function of an input parameter. Provided that the parameter is small enough, an fpt-algorithm runs fast even for large inputs. In this paper, we report on an investigation of the evolvability of fpt-algorithms via Genetic Programming (GP). The problem used in this investigation is the NP-hard 2D-Euclidean Traveling Salesman Problem (TSP), which is known to be fpt if the number of points *not* on the convex hull is taken as the parameter. The algorithm evolved in our GP study turns out to have clear characteristics of an fpt-algorithm. The results suggest GP can be utilized for generating fpt-algorithms for NP-hard problems in general, as well as for discovering input parameters that could be used to develop fpt-algorithms.

1 Introduction

Many computational problems, including those figuring in computational cognitive theories, are NP-hard. Traditionally, such NP-hard problems are considered intractable for all but small input sizes [3]. This has led applied computer scientists to focus attention on developing inexact (heuristic) methods for approaching NP-hard problems, and cognitive scientists to reject NP-hard problems as psychologically implausible models of human cognition [14]. However, it is known that certain NP-hard functions can be computable in a time that is polynomial for the overall input size and superpolynomial for only a small aspect of the input, called the *parameter*. Problems for which this holds are called *fixed-parameter tractable* and are said to belong to the complexity class FPT [2]. As long as the parameter is small enough for those instances of interest, an NP-hard problem in FPT can be considered efficiently solvable.

How do we know if a given problem is in FPT for some parameter k? One way of finding out is by designing an algorithm that solves the problem and establish that its running time can be expressed as a polynomial function of the input size, n, and a superpolynomial function of k (i.e., time $O(n^{\alpha} f(k))$, where α is some constant and $f(.)$ is a function depending only on k). Designing such an algorithm can be technically quite challenging, however, especially if the relevant parameter is yet to be discovered. It is for this reason that we investigate here the utility of genetic programming (GP) as a general method for developing or discovering fpt-algorithms for NP-hard problems.

Genetic programming (GP) is an evolutionary computation technique used to evolve computer programs [6, 13]. Populations of programs are evaluated and the fittest individuals are 'bred' to form new populations. Breeding is performed by applying genetic operations such as *crossover*, which creates new programs by recombining random parts of two selected programs, and *mutation*, where a random part of a program is randomly altered to form a new program. We used GP to evolve an algorithm that solves instances of the 2-dimensional Euclidean Traveling Salesman problem (TSP): given a set of points ('cities') in the plane, find the shortest tour visiting all points and returning to its starting point. This problem is known to be NP-hard and in FPT if the number of inner points is taken as a parameter [1]. The inner points of a TSP instance are the points that are in the interior of the convex hull. GP has often been applied to finding heuristic algorithms for TSP (see for example [12]), but to our knowledge no attempts to use GP to find fpt-algorithms exist to this date.

The aim of this research is to test whether or not an fpt-algorithm for TSP can be evolved using GP. Also of interest is whether GP can be used to discover potentially interesting input parameters for use in

developing new fpt-algorithms for NP-hard problems in general. The rest of this paper is organized as follows. Section 2 describes our method, Section 3 provides our results and Section 4 concludes.

2 Method

In GP, when evaluating an evolved program, called an *individual*, it is executed within a context of predefined supporting code, referred to as the *environment*. As this environment does not evolve, its functionality remains constant over all evaluations. The environment combined with the individual forms the algorithm. The *primitive set* is the set of functions, variables and constants available to the GP process for generating programs. As tree-based GP [13] was used, functions are referred to as function nodes (forming internal nodes in a program's tree), while variables and constants are terminal nodes (forming the leaves). Lastly, where the primitive set and environment define the search space, it is the *fitness function* that defines the goal of the search process [13], with individuals assigned a higher fitness value having a higher chance of being selected for breeding.

2.1 The environment

For the environment a structure was chosen similar to the one used in [12]. In this environment, a tour is built step by step, with the evolved individual forming a function that determines for each step what the next city in the tour should be. Algorithm 1 contains a pseudocode version of the environment. For each TSP instance used to evaluate a given individual, the environment loops through all cities in the problem instance that have not yet been 'visited' (i.e., that are not yet part of the tour). For each city, the evolved function calculates a score. When all unvisited cities have been scored, the city with the lowest score is selected. This city is added to the tour, and is therefore considered 'visited'. This process repeats itself until the tour includes all cities in the problem instance. In effect the algorithm 'travels' from city to city until it has visited them all and the tour is complete. If at each step the evolved function has given the best score to the correct city, the algorithm has found an optimal tour. In case of a Nearest Neighbor heuristic, it would score each city according to its distance from the 'current' city (i.e., the distance from the city last added to the tour to the city being evaluated), which will not yield an optimal tour for many problem instances.

Algorithm 1 The environment represented in pseudocode.

```
city-start = random city
city-current = city-start
while not visited all cities do
    selected = None
    bestscore = ∞
    for all unvisited cities do
        city-eval = next unvisited city to evaluate
        score = result of evolved function
        if score < bestscore then
            bestscore = score
            selected = city-eval
        end if
    end for
    add selected city to tour
    city-current = selected
end while
return  length of tour
```

This structure was chosen because it constrains the evolved function to the specific task of solving TSP. The structure allows for a wide range of solvers, from purely heuristic (e.g., Nearest Neighbor) variations, to optimal exhaustive searchers. Which exact algorithms can be constructed is constrained by the primitive set.

Name	`distance`
Return type	Number
Child nodes	2 of type city
Description	Returns the distance between the two given cities.
Name	`if-on-convex`
Return type	Number
Child nodes	1 of type city, 2 of type number
Description	If the given city is on the convex hull, returns the result of evaluating the first numeric child node. Else it returns the result of the second.
Name	`for-loop-`X
Return type	Number
Child nodes	1 of type number
Description	Loops through unvisited cities, evaluating the child node for each one and adding up the result to a running total. This total is returned. X is a number referring to the associated variable-node, see Table 2.

Table 1: Function set, domain-specific functions

2.2 The primitive set

Tree-based GP was used, so an evolved algorithm forms a tree structure consisting of any valid combination of function nodes and terminal nodes. Strong typing was used to enforce type requirements of certain nodes [11]. At its root, the tree returns a real number: the calculated score. All function nodes and many terminal nodes return this type, meaning they can form the root of the tree, allowing for a wide variety of possible programs.

The basic concept behind the primitive set was also inspired by the research of [12], in that the primary tool for scoring a city to be used in the evolved function was *distance*. However, unlike their research, in our experiment we required the primitive set to be sufficient for specific types of algorithms other than heuristics such as Nearest Neighbor. The first type is exhaustive algorithms, the second is fpt-algorithms.

2.2.1 Iteration and recursion

Some form of iteration or recursion is needed in order for an evolved program to implement something more complex than a heuristic of a complexity that is linear to the size of its tree. Therefore, both iteration and recursion were implemented and added to the primitive set.

Traditional implementations of iteration in GP [8, 9] do not allow for nested loops, nor do they allow for a loop counter or element reference to be used by other nodes in the tree when iterating over a vector. In this research, complex nested loop structures are of interest, as they allow for more advanced calculations, and more emergent computational complexity. Therefore, a simplified version of the iteration implementation described in [4, 5] was used. A for-loop node was implemented which iterates over all unvisited cities when called. On each iteration, it sets a variable to reference the current unvisited city in the iteration, and evaluates its child node. The result is added to a running total, which is returned at the end of the loop. The variable can be accessed through a special terminal node that can only be generated inside the subtree of a given for-loop node, as it will be linked to that specific loop node and only has a value inside its 'scope'.

Recursion was implemented in the form of a terminal node. When this node is called while calculating the score for the city under evaluation, it adds the evaluated city to the tour and recursively calls the function holding the algorithm. By doing this, it causes the calculation of the rest of the tour as the algorithm would find it, if the given city were to be added to the tour. The length of this tour is returned, and the node returns this in turn as its result, after removing the evaluated city from the tour.

2.2.2 Primitive set contents

With the evolved program calculating a score using real numbers, it seemed useful to include basic mathematical operations for calculating and combining results from other nodes. The function set contained function nodes for addition, subtraction, multiplication and division, and for min and max operations. All these nodes require two children returning numbers, and return a single number themselves after performing their mathematical operation on the results of the child nodes. Besides these basic operations, certain

Name	`city-current`
Type	City
Description	The current city, i.e., the city last added to the tour.
Name	`city-eval`
Type	City
Description	The city currently being evaluated, i.e., the city that is being scored.
Name	`city-start`
Type	City
Description	The starting city of the tour (static).
Name	`var-`X
Type	City
Description	Associated with a for-loop node further up the tree (identifiable by identical X, see Table 1). Refers to the city that the loop has assigned to the variable before evaluating the subtree this node is in.
Name	`recursion`
Type	Number
Description	Returns the distance of the tour that the algorithm would travel if the city that is being evaluated were selected and traveled to by the algorithm.

Table 2: Terminal set

domain-specific functions were necessary, listed in Table 1. The `distance` node is used to find the distance between two cities. As arguments for this function, several terminal nodes exist that return a given city, listed in Table 2. The relevant nodes for iteration and recursion are also included, and are as defined earlier. Lastly, a node was added that represents the knowledge of the convex hull of a given TSP instance. This `if-on-convex` node checks if a given city is on the convex hull. If so, then it evaluates the first of its subtrees; if not, then it evaluates the other. Therefore, an evolved program using this node can alter its method of calculating a score (and its result) depending on whether the evaluated city is on the convex hull or not.

2.3 Fitness function

Traditionally, GP experiments use a single value to determine the fitness, such as the difference between the length of a shortest tour (i.e., the optimal solution for TSP) and the length of the tour that was found by an individual. In this experiment, however, there are two relevant values: speed and accuracy.

Speed was measured using the number of tree nodes evaluated in creating an individual's tour, where a lower number of evaluations is faster and therefore better (on this measure). Individuals with many loops or with a recursion would evaluate a larger number of nodes, and score worse than a Nearest Neighbor-like individual. Equation 1 shows how the speed measure was calculated, where the number of instances refers to the instances used in evaluating the individual.

$$fitness_{speed} = \frac{\text{number of instances}}{\text{nodes evaluated}} \tag{1}$$

Accuracy was measured as the difference between the length of an optimal solution to the TSP instance the individual just solved, and the length of the tour the individual found. Equation 2 shows the exact calculation.

$$fitness_{accuracy} = \frac{1}{1 + \text{ tour length error}} \tag{2}$$

Note that Equations 1 and 2 are chosen such that $fitness_{speed}$ is decreasing in the number of evaluated nodes, and $fitness_{accuracy}$ is decreasing in the length of the produced tour.

In exploratory runs it became clear that, if both $fitness_{speed}$ and $fitness_{accuracy}$ independently contributed to overall fitness, then the Nearest Neighbor (NN) individuals and exhaustive search individuals, consisting of only three and one nodes respectively, would always be selected for breeding. Apparently their good speed and good accuracy respectively would always 'beat' more complex individuals that were in their initial stages of development. This made it practically impossible for more complex individuals to exist for

longer than a single generation, and therefore difficult for such individuals to evolve into more 'fit' variants. To prevent the search process from fixating on the two extremes of exhaustive versus NN search, lower limits were set on both speed and accuracy. These limits would start at a high level in the beginning of the run, but would become lower with each generation until (i) the accuracy limit would make Nearest Neighbor search unfeasible and (ii) the speed limit would make exhaustive search unfeasible. It was our expectation that the introduction of such strict lower limits would enable the evolution process to go beyond the fastest heuristic approach and the intractable exact approach, and explore instead accurate yet tractable algorithms such as fpt-algorithms. The fitness functions with the additional lower limits are given in Equation 3.

$$\begin{gathered}\text{nodes evaluated } > \text{maximum nodes } \vee \\ \text{tour length error } > \text{maximum error } \Rightarrow \\ fitness_{speed} = fitness_{accuracy} = 0\end{gathered} \tag{3}$$

$$\text{otherwise} \Rightarrow fitness_{speed} = \frac{\text{number of instances}}{\text{nodes evaluated}} \ \wedge \ fitness_{accuracy} = \frac{1}{1 + \text{ tour length error}}$$

In comparing two individuals, it is very likely that neither of them may be better in both speed and accuracy, particularly in the earlier generations of a run, and especially considering the existence of the extreme individuals mentioned earlier. A simple criterion was introduced to counteract this effect: Whenever individuals were compared during selection, if neither was better on both speed and accuracy, there would be a chance they were then compared on *either* speed *or* accuracy to find a winner. This probability was set to start at a high level, and decreased as the process advanced in generations.

2.4 Experiment details

The GP experiment was implemented using the evolutionary computation for Java toolkit, ECJ [10]. Many GP parameters were left at the defaults used by Koza [7], such as those involving the building of initial trees in the population. Experiment-specific parameters and their values are listed in Table 3.

Parameter	**Value**
Generations	50
Population size	128
Crossover rate[1]	0.80
Mutation rate[1]	0.10
Reproduction rate[1]	0.10
TSP instance size	7
TSP instances per evaluation	50
Total pool of random instances	500

[1]The crossover, mutation and reproduction rates determine the probability of said genetic operation being used in breeding a new individual. See [7, 13] for details on these operations.

Table 3: GP experiment parameters

The TSP instances used in the experiment each consisted of 7 points. This number was kept deliberately low to ensure that evaluation progressed at a reasonable rate. Larger instances meant that individuals using (exhaustive) recursion would spend a large amount of time per instance, making the evaluation of a large number of individuals take an impractical amount of time. Population size was limited to 128 for the same reason. The instances were generated beforehand, as a set of 7 random coordinates in an area of 500 by 500 points. Every possible number of inner points (0 to 4, as a minimum of 3 points form the convex hull) was equally represented in the pool of instances. This was achieved by randomly generating point sets and disregarding instances that did not match the required number of inner points. For each generated instance both the optimal solution and the points on the convex hull were calculated. Each individual was evaluated on 50 TSP instances randomly selected from a pool of 500 available instances.

3 Results

3.1 Best evolved individual

In the GP experiment, one type of individual was consistently selected as the best individual of a run. The individual's code is shown in Program 1. This individual would generally develop fairly early in the run, between generations 10 and 25 (of 50), and due to its relatively high fitness it would immediately form the best individual of the generation and remain so until the end of the run. The early development is not surprising given the structure of the individual. It is a small tree consisting of only a few nodes, and substantial parts of the tree are formed by nodes that make up two common individuals in the population. The subtree `(distance city-current city-eval)` is equal to the tree of a Nearest Neighbor (NN) individual, shown in Program 2. Similarly, the `(recursion)` node would on its own form the entirety of an exhaustive search individual, shown in Program 3.

Program 1 The program of the best evolved individual.

```
(if-on-convex city-current
    (distance city-current city-eval)
    (recursion))
```

Program 2 The program of the common Nearest Neighbor individual.

```
(distance city-current city-eval)
```

Program 3 The program of the common exhaustive search individual.

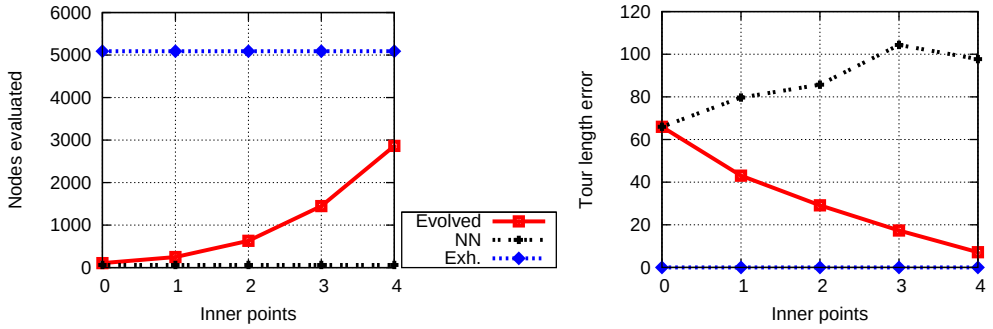
```
(recursion)
```

The behavior of the best evolved individual (Program 1) is straightforward: If the current city is part of the convex hull, it travels to the nearest neighboring city. Otherwise, it recursively builds possible extensions of the tour resulting from travelling to any of the unvisited cities. This recursive process repeats *until* the program encounters again a point on the convex hull, in which case it will extend each of the partial tours constructed so far by traveling to the nearest unvisited neighbor. From all tours constructed in the process, the program determines which is the shortest, and travels to the unvisited city that has that tour as a result. Note that if the instance the algorithm is solving happens to have few inner points, say only 1, it will do much less recursion than an exhaustive solver. At the same time, it will give more accurate results than NN when solving more complex instances with multiple inner points, as the 'look ahead' in the recursion allows it to avoid certain bad choices that NN would make. If tours with such bad choices occur after a recursion has been entered, they will most likely be discarded due to their higher length.

3.2 Fpt-characteristics

Is the best evolved individual an fpt-algorithm for TSP? To address this question, we first consider the algorithm's time behavior: The worst-case[2] time-complexity of the best individual is $O(k!\,(n-k)^2) = O(k!\,n^2)$, where n is the total number of points and k is the number of inner points. For instances with zero inner points, the program behaves as an NN individual (with time-complexity $O(n^2)$), and for instances with a very large number of inner points, performance is nearer an exhaustive search (with time-complexity $O(n!)$). We also investigated the algorithm's average-case time-complexity by running the algorithm on the 500 random instances in the pool. The results, depicted in Figure 1a, show that the average time required for Program 1 to find a tour grows speedily with the number of inner points, with its running time being close to that of the NN heuristic for few inner points and growing closer to the exhaustive algorithm as the number of inner points increase. In sum, the evolved program indeed exploits the number of inner points for

[2]Due to the nature of both the environment and the evolved program itself, an individual's time-complexity depends on the point selected as starting point (which is a random selection). If this point is an inner point, for example, more recursion will be performed than if that city is not visited until later on in the computation.

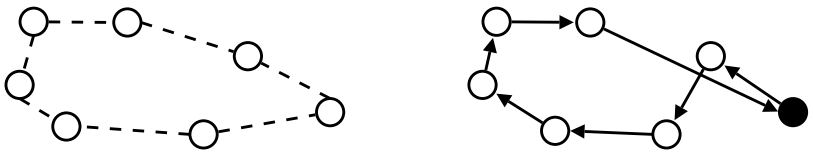


(a) Average number of nodes evaluated (b) Average tour length error

Figure 1: Speed (a) and accuracy (b) as a function of number of inner points, for the evolved program, NN, and exhaustive search, averaged over all 500 random instances and all possible starting cities.

the efficient computation of instances for which that parameter is small, and the running-time behavior is as one expects of an fpt-algorithm (i.e., the running time can be expressed as a polynomial function of input size, n, and a superpolynomial function of only the parameter k).

As it turns out, the best evolved individual does not meet the second criterion for being an fpt-algorithm, viz., exactness. Due to its reliance on NN to select the optimal tour when travelling over the convex hull, it inherits some of NN's flaws. Figure 2 shows an example of a trivial instance (i.e., one without any inner points) where, for a certain starting point, NN fails to find an optimal tour. Such instances are not rare: For only 4 of the 100 generated instances with no inner points, NN is able to find an optimal tour regardless of the starting point, with on average 3.32 out of 7 starting points per instance resulting in a less than optimal tour. Be that as it may, the performance of the best evolved individual is much better than NN for all instances with at least one inner point (see Figure 1b). Hence, even if the best evolved individual is not an exact algorithm for TSP, it clearly outperforms a polynomial-time heuristic, like NN.



(a) Optimal tour (convex hull) (b) NN tour

Figure 2: Example instance with no inner points where NN does not find an optimal tour when starting from a certain city (starting city shown in black).

4 Conclusion

The program evolved in our GP experiment shows clear characteristics of an fpt-algorithm, even though strictly speaking it is not: The program does not solve all TSP instances optimally, though it is much more accurate than its polynomial-time competitor, the NN heuristic. Also, the program is characterized by an fpt running time of $O(k!\, n^2)$. This result is promising with regards to the utility of GP for developing fpt-algorithms for NP-hard problems in general and discovering relevant parameters that can be used in such algorithms. We think that the fact that the best individual in our experiment was not an exact algorithm for TSP does not detract from this point, because an evolved fpt-heuristic can give a clear suggestion as to the direction in which an fpt-algorithm can be sought. After all, TSP is known to be in FPT if the parameter is the number of inner points [1], and the fpt-heuristic evolved in our experiment used this same parameter to bound its superpolynomial running time. Besides the important step of discovering the parameter, it is conceivable that fpt-like inexact individuals themselves can be transformed into fpt-algorithms; and even if they cannot, an evolved (inexact) fpt-heuristic may still strike a better balance between speed and accuracy for instances of practical interest, than do available polynomial-time heuristics.

References

[1] V.G. Deineko, M. Hoffman, Y. Okamoto, and G.J. Woeginger. The traveling salesman problem with few inner points. In K.-Y. Chwa and J.O. Munro, editors, *Computing and Combinatorics: 10th Annual International Conference, COCOON 2004*, volume 3106 of *LNCS*, pages 268–277, Berlin, 2004. Springer-Verlag.

[2] M.R. Fellows and R.G. Downey. Parameterized complexity after (almost) 10 years: review and open questions. In C.S. Calude et al., editors, *Proceedings of Combinatorics, Computation and Logic, DMTCS'99 and CATS'99*, volume 21 of *Australian Computer Science Communications*, pages 1–33, Singapore, 1999. Springer-Verlag.

[3] M.R. Garey and D.S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W.H. Freeman, New York, NY, 1979.

[4] E. Kirshenbaum. Genetic programming with statically scoped local variables. In D. Whitley et al., editors, *GECCO 2000: Proceedings of the Genetic and Evolutionary Computation Conference, July 10-12, 2000, Las Vegas, Nevada*, pages 459–468, San Francisco, CA, 2000. Morgan Kaufman.

[5] E. Kirshenbaum. Iteration over vectors in genetic programming. Technical Report HPL-2001-327, HP Laboratories Palo Alto, 2001.

[6] J.R. Koza. Genetic Programming: a paradigm for genetically breeding populations of computer programs to solve problems. Technical Report CS-TR-90-1314, Stanford University, Stanford, CA, 1990.

[7] J.R. Koza. *Genetic Programming II: automatic discovery of reusable programs*. MIT Press, Cambridge, MA, 1994.

[8] J.R. Koza and D. Andre. Evolution of iteration in genetic programming. In L.J. Fogel, P.J. Angeline, and T. Baeck, editors, *Evolutionary Programming V: proceedings of the fifth annual conference on evolutionary programming*, Cambridge, MA, 1996. MIT Press.

[9] J.R. Koza, D. Andre, F.H. Bennett III, and M. Keane. *Genetic Programming 3: Darwinian invention and problem solving*. Morgan Kaufman, 1999.

[10] S. Luke et al. *ECJ 17 - A Java-based evolutionary computation system*, 2008. Retrieved from http://www.cs.gmu.edu/ eclab/projects/ecj/.

[11] D.J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[12] M. Oltean and D. Dumitrescu. Evolving TSP heuristics using multi expression programming. In M. Bubak et al., editors, *Computational Science - ICCS 2004: 4th International Conference, Proceedings, Part II*, volume 3037 of *LNCS*, pages 670–673, Berlin, 2004. Springer-Verlag.

[13] R. Poli, W.B. Langdon, N.F. McPhee, and J.R. Koza. Genetic Programming: an introductory tutorial and a survey of techniques and applications. Technical Report CES-475, University of Essex, Computing and Electronic Systems, United Kingdom, 2007.

[14] I. van Rooij. The tractable cognition thesis. *Cognitive Science*, 32:939–984, 2008.