COMPUTER SCIENCE | RESEARCH ARTICLE

Automatic design of algorithms for the traveling salesman problem

Cristian Loyola1, Mauricio Sepúlveda1, Mauricio Solar2, Pierre Lopez3 and Victor Parada1*

Abstract: The automatic generation of procedures for combinatorial optimization problems is emerging as a new alternative to address the hardest problems of this class. One of these problems still offering great computational difficulty is the traveling salesman problem. Its simple presentation masks the great difficulty that exists when solving it numerically. The results obtained so far for this problem are based on the hybridization of known heuristics. However, there is still a need for an experimental breakthrough in the study of all of the possible combinations of heuristics, which represents a huge search space. In this paper, we explore this space using evolutionary computing to automatically design new algorithms for the problem. We carried out a computational experiment to produce the algorithms that not only are competitive with some of the existing heuristics but that also contain several novel structures that directly influence performance.

Subjects: Evolutionary Computing; Operations Research; Machine Design

Keywords: traveling salesman problem; automatic algorithm design; heuristics, hyperheuristics, genetic programming, evolutionary computing

ABOUT THE AUTHORS

Cristian Loyola is an Informatics Engineer from Universidad de Santiago de Chile. Mauricio Sepúlveda is an Informatics Engineer and he has earned a Dr Sc in Engineering Sciences from the same university. His current research interests are on optimization methods.

Mauricio Solar is an Electronic Engineer and earned a MSc and Dr Sc in Computer Science and Systems Engineering. His current research interests are on the search of efficient algorithms to combinatorial problems in astronomy.

Pierre Lopez From the Université Paul Sabatier in Toulouse, he received a BS in Electronics, Electrotechnics, and Automatics, a MS in Control Engineering and Industrial Informatics, and a PhD in Manufacturing Engineering. His research area includes Operations Research, Contrain Programming, multiobjective optimization and optimization of the energy management in multi-source systems.

Victor Parada is a Chemical Engineer, MSc in Production Engineering and Dr Sc in Computer Science and Systems Engineering. His current research interest is on evolutionary computation.

PUBLIC INTEREST STATEMENT

Since the early days of computing, computer programming has been a complex task. Programmers model the problem, design algorithms, write code and implement the solution. Although all are complex stages, algorithm design is perhaps one of the stages that requires a higher level of expertise. In this article, we propose that this task can be automated. To test this idea, new algorithms are designed automatically for the traveling salesman problem. The new algorithms are constructed through genetic programming, a discipline that emulates the behavior of Darwinian evolution to search for the best solution in a complex space. We carried out a computational experiment to produce the algorithms that not only are competitive with some of the existing heuristics but that also contain several novel structures that directly influence performance.
1. Introduction
The determination of the optimal solution of difficult combinatorial optimization problems is still a great computational challenge. In the real world, such problems typically arise in the field of operations research with several applications in logistics and transport. This family of problems has traditionally been faced with mathematical programming techniques. Although these techniques guarantee solving the problem to optimality, they are usually performed with high computational time. Heuristics and metaheuristics require lower computational times for these problems; however, only approximate solutions are guaranteed. The continuous search for the determination of an optimal solution has led to the idea of computationally creating a method for each problem. Thus, new methods can be built by assembling elementary components, which can be heuristics, meta-heuristics or exact methods.

New procedures have been automatically constructed for various optimization problems. In fact, various scheduling problems have been addressed to demonstrate the effectiveness of automatically combined heuristics in comparison to individual heuristics (Nguyen, Zhang, Johnston, & Tan, 2013, 2014; Pickardt, Hildebrandt, Branke, Heger, & Scholz-Reiter, 2013; Salhi & Vázquez Rodríguez, 2014). Additionally, new and efficient constructive heuristics, combined with others that gradually improve a solution, have been proposed for various problems such as graph coloring (Contreras Bolton, Gotica, & Parada, 2013; Sabar, Ayob, Qu, & Kendall, 2012), knapsack (Drake, Hyde, Ibrahim, & Ozcan, 2014; Parada, Herrera, Sepúlveda, & Parada, 2016), timetabling (Burke, Qu, & Soghier, 2014; Qu, Pham, Bai, & Kendall, 2015), cutting and packing problems (Burke, Hyde, Kendall, & Woodward, 2012; Terashima-Marín, Ross, Farias-Zárate, López-Camacho, & Valenzuela-Rendón, 2010) and the competitive traveling salesmen problem (Kendall & Li, 2013). Such methods are also known as hyper-heuristics that have meticulously been classified according to the nature of the space containing the heuristics in Burke et al. (2013). Hyper-heuristics are compounds of different structures that may reveal new knowledge to improve existing methods or to propose new algorithms. However, in the literature, emphasis is placed on numerical results giving less attention to the decoding of the resulting procedures as algorithms. In this article, new algorithms generated as hyper-heuristics for the traveling salesman problem (TSP) are described.

Not only the large number of applications but also the large number of procedures tested with a variety of instances makes TSP an interesting problem to test new algorithmic ideas, such as the automatic generation of algorithms. The TSP consists of determining a minimum cost route that visits each of the cities of a region, ending at the starting point and passing through each of the other points only once. The case where the traveling distance between two cities is the same in both directions is known as the symmetric case. The simplicity of its presentation masks the great difficulty that exists when it is approached numerically (Rego, Gamboa, Glover, & Osterman, 2011). Because of its NP-hardness, a property that TSP shares with many other problems, an exact polynomial algorithm that provides an optimal solution for any instance of the problem has yet to be found (Fortnow, 2009; Garey & Johnson, 1979). Although optimal solutions have been found for many problem instances, doing so has required great computational effort (Reinelt, 2007; The Traveling Salesman Problem, 2016). Even when the requirement of finding the optimal solution is relaxed in the field of heuristics, the generation of efficient algorithms for solving any instance of the problem continues to be a great intellectual challenge. Concretely, if a polynomial algorithm is found that solves one such problem, efficient algorithms may be found for all of the other problems with the same property. For this reason, the TSP has been studied extensively in the literature, and currently many different algorithms and test instances are available as necessary requirement for the automatic generation of an algorithm (Applegate, Bixby, Chvátal, & Cook, 2006).

The combination of different methods working together seems like a natural approach to face the challenge posed by the TSP. Over the years, several different approaches have emerged for the TSP and initially, most of them dealt with specific heuristics for the problem. Such methods have been successfully cross-applied to other problems related to combinatorial optimization. Later, metaheuristic approaches, which are methods that consider a set of specific rules used to perform the
search for the optimum solution in the solution space, were implemented (Rego et al., 2011). Under mathematical programming, novel ways of obtaining lower and upper bounds that facilitate the task of integer programming algorithms were found. At present, hybridizations of all of these methods are mainly considered, which allows for the generation of very efficient methods for solving some types of problem instances (The Traveling Salesman Problem, 2016). Clearly, many other heuristic combinations are possible, but the space that they constitute is enormous, and each new proposal must be evaluated numerically by computational experiments, which represents a type of “manual” test. Considering the growth in computational capacity, part of this task might be automatically performed, such as by producing algorithms using search methods based on evolutionary computation, specifically by using genetic programming (Affenzeller & Winkler, 2009; Koza, 2003; Poli, Langdon, & McPhee, 2008).

Because an algorithm can be represented as a syntax tree and determining the maximum performance algorithm for a problem is an optimization problem, this algorithm can be found with a search method such as genetic programming (GP). In genetic programming, the problem solutions are structures that represent elementary components of computer programs, and a solution is found through the automatic production of a program, starting from some high-level specifications. The process occurs through structure populations that evolve gradually due to the application of operators of selection, variation and reproduction. Population after population, the structures become more and more specialized in their specific task. Structures capable of solving some specific problems in raw material optimization and schedule definition have emerged from this approach (Burke, Hyde, Kendall, & Woodward, 2010). From this viewpoint, it is possible to make use of the existing heuristics for the TSP and to automatically combine them or parameters of them to create new algorithms through evolution (Aziz, 2015; Ryser-Welch, Miller, & Asta, 2015). However, it is unclear how the elementary components constitute algorithms for the TSP through an evolutionary process guided by genetic programming. Various algorithmic structures could arise; constructive algorithms that build a solution step by step, or neighborhood search algorithms that begin with a feasible solution and improves it at every stage. Having a framework to construct algorithms for the TSP would facilitate the rapid study of many possible algorithms by adequately fitting their components. For example, the best-known heuristic so far may be enhanced by studying the additional functionalities that evolutionary computation can adequately fit into its structure. In this way, interpreting the new resultant algorithms can also lead to the generation of new ideas.

In this paper, new algorithms automatically generated for the TSP are presented. By means of GP, we explore the space constituted by the available heuristics for the TSP to generate algorithms that efficiently and competitively solve a set of typical test instances. The automatically generated procedures find solutions for a set of problem instances. The procedures are represented by abstract syntax trees and are composed of high-level instructions and specific problem-oriented instructions. Thus, from a set of initial trees, new procedures are produced using the operations of evolutionary computation. As the evolutionary process occurs, new and better algorithms are produced. The best algorithms obtained are evaluated with a different set of problem instances.

In the following section, the methodology used for the production of algorithms for the TSP is presented. Section three presents the main findings: the new algorithms and their numerical results. Section four presents the main conclusions.

2. Generation of new algorithms for the TSP

The generation of algorithms is performed through an evolutionary process by combining specific heuristics for the problem with high-level functions. Each algorithm is represented by an abstract syntax tree in which the nodes are the instructions of the algorithm. Internal nodes are high-level functions typically found in any algorithm, while the leaf nodes are functions that act on the solution of the TSP. Figure 1 outlines the procedure based on GP. From a population of algorithms \( P(t) \) at a time \( t \), a new population \( P(t + 1) \) is obtained by applying the operations of selection, reproduction, crossover and mutation. Each new built algorithm is evaluated with a fitness function that measures...
the performance of the algorithm with a group of TSP instances. The process is repeated for a number of generations from an initial population randomly generated. The process converges to the best algorithms for the TSP. Some initial components must be defined: a data structure for storing a TSP solution, a set of functions to compose the trees, the function that measures the performance of an algorithm, a set of problem instances and the PG parameters values.

2.1. Heuristics for the TSP

The heuristics for the TSP have been classified by Rego et al. (2011) under the framework of ejection chains, a technique that has been used in search algorithms to explore large-sized neighborhoods in combinatorial optimization problems (Talbi, 2009). Two groups are identified: the so-called Lin-Kernighan heuristics and the stem-and-cycle heuristics. The former addresses the problem in a repetitive cycle in which the current circuit is modified by deleting a number of edges and replacing them with new ones. In this group, the chain corresponds to a Hamiltonian path. The heuristics of the latter group are a generalization in which the chain structure contains a Hamiltonian path and a cycle, conferring greater versatility to the moves that can be performed to generate neighboring solutions. Both the first and second groups contain basic heuristics that are structurally interesting for generating new algorithms. The following paragraphs describe some algorithms that are directly associated with the functionalities that we attribute to the algorithms in this paper.

2.1.1. Nearest neighbor heuristic
In each stage, the nearest unvisited city is added to the circuit (Ravi & Shukla, 2009). The process is repeated until all of the cities are in the circuit.

2.1.2. Insertion heuristic
In this heuristic, a new city is added between two previously connected cities. There are four variants that arise when considering different methods of selecting the city to be inserted. In the nearest insertion and in the farthest insertion, the selected city is the closest and the farthest, respectively, of the cities in the circuit. In the cheapest insertion, the city is selected such that the increase in the total cost of the circuit is minimized. Finally, in the arbitrary insertion, the city is randomly selected.

2.1.3. Savings heuristic
This heuristic starts with a partial route in which a focal city is chosen arbitrarily, to which the salesman returns after every visit to another city. For each pair of non-focal cities, the savings is the distance that the route would be shortened if the salesman went directly from one city to another while skipping the focal city (Clarke & Wright, 1964).

2.1.4. $\lambda$-Opt heuristic
In each stage, $\lambda$ connections of the present route are replaced by new connections to obtain a new, shorter route (Lin, 1965).
2.1.5. Lin-Kernighan heuristic

In this case, no fixed value is used for the $\lambda$-Opt; rather, the heuristic attempts to obtain the best value of $\lambda$ for each movement. The heuristic is based on the fact that each $\lambda$-Opt movement can be represented as a sequence of 2-Opt movements (Lin & Kernighan, 1973).

The TSP has also been addressed through metaheuristics that consider different search techniques, such as simulated annealing (Aarts, Korst, & van Laarhoven, 1988), GRASP (Goldbarg, Goldbarg, & Farias, 2007), neural networks (Jolai & Ghanbari, 2010), Tabu Search (Knox, 1994) and evolutionary computation (Neumann, 2008).

One of the evolutionary computer techniques is genetic programming, where the individuals represent elemental program components that from some specifications of the problem under study, a program to solve it is automatically produced (Poli et al., 2008). From this point of view, it is possible to make use of previously existing heuristics that solve an optimization problem and evolutionarily originate new solution procedures that combine such heuristics.

2.2. A proposed data structure

The efficiency of an algorithm has a direct relation to the data structure used; thus, to automatically design algorithms, a data structure must be available to support its functionality. The following data structures are defined:

- $LC$: A list that contains the cities and their coordinates $(x, y)$.
- $D(n \times n)$: A matrix that stores the distances between each pair of cities.
- $Tour$: A list that stores the generated partial circuit. After execution, $Tour$ contains the circuit generated by the algorithm.
- $CC(x, y)$: The coordinates of the central point of all the cities.

2.3. Design of functions that compose the algorithms

The elemental components of the existing algorithms are translated into two sets of functions. The first set contains the basic instructions in most computer languages and is defined with parameters $P_1$ and $P_2$, as integer variables, considering that a value greater than 0 corresponds to the “true” logical value and that a value equal to 0 corresponds to “false”. All functions return true or false. These functions are While($P_1$, $P_2$), which executes $P_2$ while $P_1$ returns “true”; And($P_1$, $P_2$), which executes $P_1$ and $P_2$ and returns “true” if both return “true” or “false”; and IfThen($P_1$, $P_2$), which executes $P_2$ if the execution of $P_1$ returns “true”.

The second set of functions contains the following functions designed exclusively for the TSP. Two types of functions belonging to this set are designed: constructive terminals and tour modification terminals. The constructive terminals add cities to the Tour while it is incomplete. Using different criteria, it selects a city that is not yet on the Tour and adds it in a determined position. If the tour is already complete, it does not make any modification. It returns the value “1” each time a city is added to the Tour and “0” if the Tour is already complete. The designed terminals are based on typical constructive heuristics for the TSP and are described below:

- best-neighbor(): This terminal adds the city in Tour that is closest to the last city added in Tour. If Tour is empty, the closest pair of cities is added.
- worst-neighbor(): This terminal adds the city in Tour that is farthest to the last city added in Tour. If Tour is empty, the farthest apart pair of cities is added.
- nearest-insertion(): This terminal adds the city in Tour that is closest to one of the cities in Tour. It is based on the “Nearest Insertion” heuristics. If Tour is empty, it adds the pair of cities that are closest to each other.
• **farthest-insertion()**: This terminal adds the city in Tour that is farthest from one of the cities in Tour. It is based on the “Farthest Insertion” heuristics. If Tour is empty, it adds the pair of cities that are farthest apart from each other.

• **near-center()**: This terminal adds to Tour the closest city to the central point CC(x, y).

• **far-center()**: This terminal adds to Tour the farthest city from the central point CC(x, y).

The second type of terminal modifies the tour built up to this point. It gives the tour under construction more variability, and it operates on Tour, changing the order of the cities. It returns the value “1” if a modification is made to the Tour and “0” if not.

• **invert()**: This terminal inverts the order of the cities in Tour, exchanging the first city for the last, the second for the next to last, and so on.

• **2-Opt()**: This is the 2-Optimal adaptation algorithm. The algorithm makes an improvement in the partially constructed tour. A closed circuit is generated with the cities in Tour, where the edges are exchanged. If the algorithm finds an improvement, Tour is updated.

### 2.4. A representation for the new algorithms

To easily operate the algorithms, we represented each one as a tree, which is one of the representations typically used in genetic programming (GP) (Koza, 2003). At the end of the evolutionary process, we manually decoded each algorithm’s corresponding tree.

### 2.5. Design of the fitness function

The fitness function considers the quality, feasibility and size of the algorithm. The quality of an algorithm is measured in terms of the cost minimization of the generated circuit, which reflects the relative error regarding the best-known solution for each instance. Additionally, an algorithm is feasible when it finds valid circuits for the TSP. Feasibility is measured in terms of the relative deviation of the number of cities considered in the generated circuit. The size of an algorithm is important for decoding the corresponding algorithm. Then, the relative error of the number of nodes compared to a desired number of nodes is considered as the third term in the fitness function. Equation (1) reflects these three components. Specifically, let n be the number of instances, zj be the optimum value of the instance j, and uj is the value obtained by the algorithm. Additionally, let cj be the total number of cities of the instance j, and let dj be the number of cities in the final tour produced by the algorithm. Related to the size, let tj be the desired number of nodes, and let ln be the number of nodes generated by the algorithm. The values α and β in [0, 1] are used to weight the relative importance of each term. Then, the fitness function $f_a$ evaluates each algorithm with a set of evolution instances according to (1). We also define the average relative error $ARE_a$, as the error produced by an algorithm a to find a solution for a set of instances (Equation 2), and the error for each instance is denoted by $e_a$.

$$f_a = \alpha \left( \frac{1}{n} \sum_{j=1}^{n} \beta \left( \frac{z_j - u_j}{z_j} \right) + \left( 1 - \beta \right) \frac{c_j - d_j}{c_j} \right) + \left( 1 - \alpha \right) \frac{t_a - l_a}{t_a}$$  

$$ARE_a = \frac{1}{n} \sum_{j=1}^{n} \left| \frac{z_j - u_j}{z_j} \right|$$

### 2.6. Evaluation of new algorithms

To evaluate the algorithm’s performance during the evolution phase, adaptation instances are used, and to test the best algorithms found, a second group of instances are used. The five adaptation cases have a known optimum solution and have been selected from the TSPLIB library (Reinelt, 2007). Their sizes range from 48 to 76 cities. The selected instances are att48.tsp, eil51.tsp, berlin52.tsp, st70.tsp and pr76.tsp. For the evaluation phase, three groups of instances with different sizes were used and were selected from the same library. Each group has 10 instances and a known optimum solution:
• Group A: rd100.tsp, kroA100.tsp, kroC100.tsp, kroD100.tsp, eil101.tsp, lin105.tsp, ch130.tsp, ch150.tsp, bier127.tsp and pr107.tsp.

• Group B: rd400.tsp, pcb442.tsp, pbm436.tsp, pbn423.tsp, pr439.tsp, rat575.tsp, rbu737.tsp, rbx711.tsp, uy734.tsp and qrl662.tsp.

• Group C: dsj1000.tsp, pr1002.tsp, u1060.tsp, vm1084.tsp, pcb1173.tsp, d1291.tsp, rl1304.tsp, rl1323.tsp, nrw1379.tsp and fl1400.tsp.

2.7. Evolutionary process

The evolutionary process was performed on a population of a fixed number of generations. Specifically, evolution was performed by generating successive populations of 500 individuals each, while considering the operators of selection and crossover with a probability of 0.85 and mutation with a probability of 0.05. The initial population was generated by the ramped half-and-half method in which half of the population was randomly generated with full trees up to a default depth and the other half with partially full trees. The fittest individuals were selected by a tournament, where the best tree was selected from a random sample of trees from the current population. For each tree, one of two types of mutations was randomly selected: “point mutation” or “shrink mutation”. In the first case, a tree node was replaced by a function that uses the same number of parameters, while in the second, a node was replaced by any of the functions that acts directly on a city or on the whole circuit. The crossover between two trees was performed by replacing a node of the first tree by a section of the second.

The evolutionary process was performed on the GPC++ platform. This platform was implemented in the C++ language and contains the basic operations of evolutionary computation that operate on tree structures (Fraser & Weinbrenner, 1993). The platform was executed on Windows XP Service Pack 2 on a computer with a Dual Core 2.5 GHz processor.

3. Results

Algorithm generation presents a systemic convergence during the evolutionary process. The initial fitness values gradually decrease during this process. Figure 2 shows the convergence curves of the 30 runs, each executed with a different seed. Each point of the graph represents the average population fitness for each generation. It is observed that the average fitness of the first generations is generally between 0.5 and 0.6. Most of the algorithms have a variable number of nodes and are generally different from the initially set value of 15. Additionally, most of these initial algorithms are unable to generate a complete tour. Between generations 1 and 20, the convergence curve tends to decrease, showing that the evolutionary process increasingly generates better algorithm populations. Fitness values of approximately 0.5 rapidly decrease to values between 0.095 and 0.19, after which the evolutionary process behavior is kept constant up to the last generations.

Figure 2. Convergence of the evolutionary process.
The size of the produced algorithms easily converges toward the desired size. As established in Equation (1), the algorithms that do not reach the desired size are penalized proportionally. For the algorithms, the average established size was 15 nodes so that the algorithms would be easily read. Figure 3 shows the evolution of the average number of nodes for 30 runs. The graph shows the convergence toward 15 nodes, which occurs in the first generations. Specifically, as of the seventh generation, the average stabilizes at 15 nodes. This means that most of the algorithms inspected during the search have a size of 15 nodes.

The best algorithm was selected from each of the 30 runs. Table 1 shows the results obtained in the evolution process in terms of the relative error in relation to the optimum solution of the five adaptation instances. The average error $ARE_a$, best error and worst error of each algorithm are shown. All of the generated algorithms are able to generate feasible solutions that include the whole set of cities in their circuit; 14 were able to reach the optimum solution for one instance, i.e. number of hits = 1. Although the remaining algorithms did not obtain optimum values, they obtained very small errors; for example, TSP3 had an error of 0.29% and TSP27 had an error of 0.99%. On average, the algorithms exhibited an error of 1.45%. Regarding the execution time, each of the runs took approximately 14–15 min.

The good performance of the algorithms found in the evolution phase is maintained in the evaluation phase with different instances. The 30 algorithms produced during the evolutionary process have been evaluated with the instances of groups A (100 cities), B (500 cities) and C (1000 cities). The algorithms performed better with instances of group A, and the average error of all the algorithms was 4.86% (Table 2). The average execution times were 0.00, 3.01 and 37.47 s for the 30 algorithms with the instances of groups A, B and C, respectively. The best result was obtained by algorithm TSP15, with an error of 0.82%. For the instances of group B, the algorithms presented an average error of 6.77%, and in the larger instances of group C, the error was 6.45%.

In spite of the adequate performance by the produced algorithms in solving instances of different groups, they do not find an optimum solution for all the studied instances. The algorithms seem to be specialized only in improving solutions of evolution instances that have a smaller size than those used during the evaluation stage. The best results are shown in the “Best Error” column in Table 1. The average error is 0.25%. This percentage is lower than the one obtained during the evaluation phase (Table 2). The best performance was obtained with instances of group A, with an average error of 1.98%. Likewise, the best algorithm produced for a group is not the same as that of the remaining groups. Only algorithm TSP27 stands out as one of the five best algorithms in groups A and C. The “Average Error” column of Table 2 shows an average error of 4.43% for TSP27 in group A and 5.98% in group C. Both percentages are less than the averages for all the algorithms: 4.86 and 6.45%.
Designed algorithms have a novel combination of structures operating on the current TSP solution. In one of them, terminal 2-Opt within a while loop operates in combination with insertion terminals. Thus, a constructive and local search procedure occurs such that each time a new city is inserted into the partial tour immediately, takes place a step of refining the new tour. This use of the terminal is different from the typical use in the literature. The cycle found is observed both in Algorithm 1 and in Algorithm 2, two of the best evolved algorithms. In Algorithm 1 the internal loop while combines 2-Opt with the terminal that adds the terminal closest city to the partial tour. In Algorithm 2, 2-Opt is called every time a new city is added to the tour by using different criteria. Note that 2-Opt is combined with the selection of the farthest neighboring city, but in combination, the city is relocated in the tour whereby the end cost is minimized. Algorithm 1 begins to work with the construction of a sub-tour with a series of constructive terminals in a cycle using two operations. It inserts cities with near-insertion and improves the tour with 2-Opt. The cycle ends when the tour includes the total number of cities, as observed in Algorithm 1 and Figure 4.

### Table 1. Results of the evolution stage

| Algorithm | Area (%) | Best $e_*$ (%) | Worst $e_*$ (%) | Standard deviation | Number of hits | Evolution time [s] |
|-----------|---------|----------------|----------------|-------------------|----------------|--------------------|
| TSP1      | 2.08    | 0.00           | 3.97           | 1.43              | 1              | 898.0              |
| TSP2      | 1.66    | 0.90           | 3.10           | 0.86              | 0              | 880.0              |
| TSP3      | 1.21    | 0.29           | 1.66           | 0.56              | 0              | 868.0              |
| TSP4      | 1.92    | 0.35           | 3.37           | 1.13              | 0              | 877.0              |
| TSP5      | 1.12    | 0.00           | 1.62           | 0.65              | 1              | 867.0              |
| TSP6      | 1.21    | 0.29           | 1.66           | 0.56              | 0              | 892.0              |
| TSP7      | 1.15    | 0.00           | 3.40           | 1.34              | 1              | 899.0              |
| TSP8      | 2.02    | 1.17           | 2.61           | 0.59              | 0              | 877.0              |
| TSP9      | 1.35    | 0.00           | 2.15           | 0.94              | 1              | 787.0              |
| TSP10     | 2.09    | 0.00           | 4.44           | 1.58              | 1              | 860.0              |
| TSP11     | 1.03    | 0.00           | 1.62           | 0.64              | 1              | 982.0              |
| TSP12     | 1.33    | 0.00           | 3.10           | 1.13              | 1              | 942.0              |
| TSP13     | 1.21    | 0.29           | 1.66           | 0.56              | 0              | 950.0              |
| TSP14     | 1.21    | 0.29           | 1.66           | 0.56              | 0              | 884.0              |
| TSP15     | 1.39    | 0.00           | 3.55           | 1.32              | 1              | 898.0              |
| TSP16     | 1.09    | 0.00           | 1.92           | 0.72              | 1              | 887.0              |
| TSP17     | 1.30    | 0.29           | 2.53           | 0.84              | 0              | 925.0              |
| TSP18     | 1.21    | 0.00           | 2.51           | 0.91              | 1              | 885.0              |
| TSP19     | 1.99    | 0.00           | 4.14           | 1.94              | 1              | 883.0              |
| TSP20     | 1.21    | 0.29           | 1.66           | 0.56              | 0              | 917.0              |
| TSP21     | 0.88    | 0.00           | 1.62           | 0.70              | 1              | 867.0              |
| TSP22     | 1.21    | 0.29           | 1.66           | 0.56              | 0              | 911.0              |
| TSP23     | 1.46    | 0.00           | 3.97           | 1.54              | 1              | 903.0              |
| TSP24     | 1.89    | 0.00           | 5.91           | 2.31              | 1              | 879.0              |
| TSP25     | 1.42    | 0.29           | 3.10           | 1.06              | 0              | 858.0              |
| TSP26     | 1.21    | 0.29           | 1.66           | 0.56              | 0              | 871.0              |
| TSP27     | 2.22    | 0.99           | 4.67           | 1.49              | 0              | 871.0              |
| TSP28     | 1.94    | 0.99           | 2.68           | 0.65              | 0              | 893.0              |
| TSP29     | 1.21    | 0.29           | 1.66           | 0.56              | 0              | 868.0              |
| TSP30     | 1.21    | 0.29           | 1.66           | 0.56              | 0              | 898.0              |
| Average   | 1.45    | 0.26           | 2.7            | 0.96              | 0.47           | 889.2              |
Table 2. Results from the evaluation of the algorithms

| Algorithm | Group A | | | Group B | | | Group C | | |
|-----------|---------|-----------|---------|---------|-----------|---------|---------|-----------|---------|
|           | AREₐ (%) | Best eₐ (%) | Worst eₐ (%) | Standard deviation | AREₐ (%) | Best eₐ (%) | Worst eₐ (%) | Standard deviation | AREₐ (%) | Best eₐ (%) | Worst eₐ (%) | Standard deviation |
| TSP1      | 4.50    | 2.19       | 7.82     | 2.11    | 6.98     | 3.30       | 9.93     | 1.72    | 6.82     | 4.97       | 7.81     | 0.95 |
| TSP2      | 4.68    | 1.03       | 7.00     | 2.04    | 6.53     | 4.22       | 8.64     | 1.65    | 6.30     | 4.52       | 7.82     | 1.01 |
| TSP3      | 5.00    | 3.26       | 6.78     | 1.27    | 6.37     | 4.07       | 7.97     | 1.47    | 6.39     | 4.45       | 8.17     | 1.24 |
| TSP4      | 5.37    | 1.49       | 9.72     | 2.81    | 6.65     | 4.70       | 8.32     | 1.06    | 5.57     | 3.97       | 6.82     | 0.96 |
| TSP5      | 5.01    | 1.03       | 6.94     | 1.85    | 6.39     | 3.91       | 7.97     | 1.26    | 6.40     | 4.45       | 8.17     | 1.14 |
| TSP6      | 5.00    | 3.26       | 6.78     | 1.27    | 6.37     | 4.07       | 7.97     | 1.47    | 6.39     | 4.45       | 8.17     | 1.24 |
| TSP7      | 4.59    | 1.03       | 6.78     | 2.07    | 6.48     | 4.66       | 9.07     | 1.21    | 6.93     | 6.08       | 7.82     | 0.6  |
| TSP8      | 5.48    | 3.32       | 12.61    | 3.07    | 7.96     | 6.57       | 10.89    | 1.29    | 7.64     | 3.80       | 10.88    | 1.94 |
| TSP9      | 5.44    | 1.35       | 8.34     | 2.59    | 7.49     | 6.22       | 9.45     | 1.08    | 6.16     | 4.18       | 7.84     | 1.07 |
| TSP10     | 4.74    | 1.68       | 7.82     | 2.35    | 7.18     | 6.07       | 8.48     | 0.86    | 6.30     | 4.41       | 7.70     | 1.08 |
| TSP11     | 5.04    | 1.03       | 7.17     | 1.73    | 6.73     | 4.66       | 9.23     | 1.48    | 6.38     | 4.45       | 8.17     | 1.16 |
| TSP12     | 4.73    | 1.03       | 6.78     | 2.14    | 6.81     | 3.91       | 8.98     | 1.42    | 6.84     | 5.23       | 7.82     | 0.81 |
| TSP13     | 5.00    | 3.26       | 6.78     | 1.27    | 6.37     | 4.07       | 7.97     | 1.47    | 6.39     | 4.45       | 8.17     | 1.24 |
| TSP14     | 5.00    | 3.26       | 6.78     | 1.27    | 6.34     | 4.07       | 7.69     | 1.43    | 6.42     | 4.76       | 8.17     | 1.19 |
| TSP15     | 3.08    | 0.82       | 7.03     | 1.80    | 7.96     | 6.30       | 9.29     | 0.94    | 7.21     | 4.78       | 8.9      | 1.33 |
| TSP16     | 4.77    | 1.03       | 6.53     | 1.58    | 6.81     | 4.74       | 9.23     | 1.48    | 6.48     | 4.45       | 8.41     | 1.21 |
| TSP17     | 3.72    | 1.65       | 6.51     | 1.76    | 7.02     | 4.59       | 9.34     | 1.73    | 6.79     | 6.03       | 7.96     | 0.64 |
| TSP18     | 4.68    | 1.03       | 7.00     | 2.04    | 6.68     | 4.22       | 8.64     | 1.40    | 6.30     | 4.52       | 7.82     | 1.01 |
| TSP19     | 4.82    | 2.36       | 6.14     | 1.30    | 7.41     | 5.40       | 9.42     | 1.38    | 6.09     | 4.16       | 7.22     | 1.05 |
| TSP20     | 5.00    | 3.26       | 6.78     | 1.27    | 6.34     | 4.07       | 7.69     | 1.43    | 6.42     | 4.76       | 8.17     | 1.19 |
| TSP21     | 5.35    | 3.26       | 6.94     | 1.29    | 6.39     | 4.26       | 8.04     | 1.36    | 6.40     | 4.76       | 8.41     | 1.19 |
| TSP22     | 5.11    | 3.26       | 6.78     | 1.17    | 6.40     | 4.26       | 8.54     | 1.41    | 6.41     | 4.45       | 8.41     | 1.28 |
| TSP23     | 5.14    | 2.06       | 8.96     | 1.86    | 6.97     | 5.32       | 9.08     | 1.30    | 5.51     | 3.41       | 7.08     | 1.23 |
| TSP24     | 4.74    | 1.03       | 6.78     | 1.89    | 6.33     | 4.22       | 7.83     | 1.34    | 6.30     | 4.52       | 7.82     | 1.01 |
| TSP25     | 5.50    | 1.03       | 9.67     | 2.45    | 6.17     | 4.22       | 7.88     | 1.56    | 6.47     | 4.52       | 8.47     | 1.21 |
| TSP26     | 5.00    | 3.26       | 6.78     | 1.27    | 6.36     | 3.87       | 7.97     | 1.50    | 6.39     | 4.45       | 8.17     | 1.24 |
| TSP27     | 4.43    | 1.16       | 10.28    | 2.85    | 7.21     | 5.71       | 9.23     | 0.95    | 5.98     | 3.86       | 7.63     | 1.21 |
| TSP28     | 4.27    | 1.73       | 7.41     | 1.82    | 7.84     | 5.02       | 10.50    | 1.62    | 6.84     | 4.61       | 8.33     | 1.23 |
| TSP29     | 5.00    | 3.26       | 6.78     | 1.27    | 6.34     | 4.07       | 7.69     | 1.43    | 6.42     | 4.76       | 8.17     | 1.19 |
| TSP30     | 5.62    | 1.03       | 9.67     | 2.28    | 6.27     | 4.47       | 7.60     | 1.25    | 6.36     | 4.45       | 8.47     | 1.35 |
| Average   | 4.86    | 1.98       | 7.60     | 1.86    | 6.77     | 4.64       | 8.68     | 1.37    | 6.44     | 4.56       | 8.10     | 1.14 |
Figure 4. Abstract syntax tree of the algorithm TSP11.
Algorithm 1: TSP 11

set $i = 0$
nearest-insertion ();
best-neighbor ();
near-center ();
$i = 3$;
IF (far-center() = 1)
  $i \leftarrow i + 1$;
  IF (worst-neighbor () = 1)
    $i \leftarrow i + 1$;
  IF (worst-neighbor () = 1)
    $i \leftarrow i + 1$;
    WHILE ($i < c$)
      nearest-insertion ();
      $i \leftarrow i + 1$;
      2-Opt ();
    END WHILE
  END IF
END IF
END IF
END IF

The algorithm with the best performance in the group of instances with a size of 100 was Algorithm 2. It has an average error of 3.08% for group A. It performs two consecutive processes; each time a city is inserted, it improves the constructed tour.

Algorithm 2: TSP15

set $i = 0$
WHILE ($i < c$)
  IF (nearest-insertion() = 1)
    $i \leftarrow i + 1$;
    2-Opt ();
  END IF
  best-neighbor ();
  $i \leftarrow i + 1$;
  2-Opt ();
  IF (nearest-insertion() = 1)
    $i \leftarrow i + 1$;
    2-Opt ();
  END IF
  worst-neighbor ();
  $i \leftarrow i + 1$;
  2-Opt ();
END WHILE
4. Conclusions

This paper presents new algorithms obtained by genetic programming that provide solutions to the TSP. To this end, a set of functions and terminals that operate directly on the data structure that stores the solution was designed. The search was guided by a fitness function, which considers the relative error produced by each algorithm, to evaluate a set of fitness cases and a function that controls the size of the algorithms. The computational experiment allowed the evolution of thousands of similar optimization algorithms that considered basic constructive heuristics, city selection operations, and the 2-OPT heuristic, which are instruments typically used by the most efficient algorithms currently used to address the problem.

To build algorithms, we designed a set of terminals that are able to insert cities into the tour or exchange the order in a current partial tour. To manipulate the terminals, three functions were designed (IfThen, While, And). All the obtained algorithms have a While cycle in their structure, which enables them to input the complete set of cities into the tour. Most of them perform two actions: first, an insertion and later, an improvement using 2-OPT. Its construction occurs by testing them with a set of fitness instances, which exhibit good performance that remains when they are evaluated with different instances. The best results were obtained with evolutionary instances with sizes of 100 (Group A). In addition, 100 generations are sufficient to achieve convergence towards the best algorithms. The algorithms found are constructive, i.e. they step by step build and refine a TSP solution. Although the algorithms are not able to deliver optimum values for all instances, they do deliver approximate values as solutions, as in any heuristic. It is an expected result because the basic algorithmic components, defined as internal functions, are components of already existing heuristic algorithms. Finding exact algorithms with this methodology would require functions that always ensure the determination of the optimal solution.

Funding
This research was partially funded by project ECOS/ CONICYT [grant number C13E04] and the Complex Engineering Systems Institute ISCI, [grant number POS-004-F]; CONICYT [grant number FB016].

Author details
Cristian Loyola1
E-mail: cristian.lyloyola@usach.cl
Mauricio Sepúlveda2
E-mail: mauricio.sepuveda@usach.cl
ORCID ID: http://orcid.org/0000-0002-5522-3194
Mauricio Solar2
E-mail: mauricio.solar@usm.cl
Pierre Lopez3
E-mail: pierre.lopez@laas.fr
Víctor Parada1
E-mail: victor.parada@usach.cl

1 Departamento de Ingeniería Informática, Universidad de Santiago de Chile, 3659 Av. Ecuador, Santiago, Chile.
2 Departamento de Informática, Universidad Técnica Federico Santa María, Campus San Joaquín, Avenida Vicuña Mackenna 1910, San Joaquín, Santiago, Chile.
3 LAAS-CNRS, Université de Toulouse, Toulouse, France.

Citation information
Cite this article as: Automatic design of algorithms for the traveling salesman problem, Cristian Loyola, Mauricio Sepúlveda, Mauricio Solar, Pierre Lopez & Víctor Parada, Cogent Engineering (2016), 3: 1255165.

References
Aarts, E., Korst, J., & van Laarhoven, P. (1988). A quantitative analysis of the simulated annealing algorithm: A case study for the traveling salesman problem. Journal of Statistical Physics, 50, 187–206. http://dx.doi.org/10.1007/BF01022991
Affenzeller, M., & Winkler, S. (2009). Genetic algorithms and genetic programming: Modern concepts and practical applications. Boca Raton, FL: CRC Press. http://dx.doi.org/10.1201/CHNUMINSIGH
Applegate, D., Bixby, R., Chvatol, V., & Cook, W. (2006). The traveling salesman problem: A computational study. Princeton, NJ: Princeton University Press.
Aritz, Z. A. (2013). Ant colony hyper-heuristics for travelling salesman problem. Procedia Computer Science, 76, 534–538. http://dx.doi.org/10.1016/j.procs.2015.12.333
Burke, E. K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Ozcan, E., & Qu, R. (2013). Hyper-heuristics: A survey of the state of the art. Journal of the Operational Research Society, 64, 1695–1724. http://dx.doi.org/10.1057/jors.2013.71
Burke, E. K., Hyde, M., Kendall, G., & Woodward, J. (2010). A genetic programming hyper-heuristic approach for evolving 2-D strip packing heuristics. IEEE Transactions on Evolutionary Computation, 14, 942–958. http://dx.doi.org/10.1109/TEVC.2010.2041061
Burke, E. K., Hyde, M. R., Kendall, G., & Woodward, J. (2012). Automating the packing heuristic design process with genetic programming. Evolutionary Computation, 20, 63–89. http://dx.doi.org/10.1162/evco_a_00044
Burke, E. K., Qu, R., & Soghier, A. (2014). Adaptive selection of heuristics for improving exam timetables. Annals of Operations Research, 218, 129–145. http://dx.doi.org/10.1007/s10479-012-1140-3
Clarke, G., & Wright, J. (1964). Scheduling of vehicles from a central depot to a number of delivery points. Operations Research, 12, 568–581. http://dx.doi.org/10.1287/opre.12.4.568
Contreras Bolton, C., Gatica, G., & Parada, V. (2013). Automatically generated algorithms for the vertex coloring problem. PLoS ONE, 8, e58551. http://dx.doi.org/10.1371/journal.pone.0058551
Drake, J. H., Hyde, M., Ibrahim, K., & Ozcan, E. (2016). A genetic programming hyper-heuristic for the multidimensional knapsack problem. Kybernetes, 43, 1500–1511. http://dx.doi.org/10.1108/K-09-2013-0201

Fortnow, L. (2009). The status of the P versus NP problem. Communications of the ACM, 52, 78–86. http://dx.doi.org/10.1145/1562164

Fraser, A., & Weinbrenner, T. (1993). GPC++ genetic programming C++ class library. Retrieved from ftp://ftp.cs.bham.ac.uk/pub/authors/WB.Langdon/weinbrenner/gp.html

Garey, M., & Johnson, D. (1979). Computers and intractability: A guide to the theory of NP-completeness. A series of books in the mathematical sciences. San Francisco, CA: WH Freeman and Company.

Goldbarg, E., Goldbarg, M., & Farias, J. (2007). GRASP with path-relinking for the TSP. In K. F. Doerner, M. Gendreau, P. Greistorfer, W. Gutjahr, R. F. Hartl, & M. Reimann (Eds.), Metaheuristics: Progress in complex systems optimization (pp. 137–152). New York, NY: Springer.

Jolai, F., & Ghanbari, A. (2010). Integrating data transformation techniques with Hopfield neural networks for solving travelling salesman problem. Expert Systems with Applications, 37, 5331–5335. http://dx.doi.org/10.1016/j.eswa.2010.01.002

Kendall, G., & Li, J. (2013). Competitive travelling salesmen problem: A hyper-heuristic approach. Journal of the Operational Research Society, 64, 208–216. http://dx.doi.org/10.1057/jors.2012.37

Koza, J. (2003). Genetic programming IV: Routine human-competitive machine intelligence. Massachusetts: Kluwer Academic Pub.

Lin, S. (1965). Computer solutions of the traveling salesman problem. Bell System Technical Journal, 44, 2245–2269. http://dx.doi.org/10.1002/bltj.1965.44.issue-10

Lin, S., & Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. Operations Research, 21, 498–516. http://dx.doi.org/10.1287/opre.21.2.498

Neumann, F. (2008). Expected runtimes of evolutionary algorithms for the Eulerian cycle problem. Computers and Operations Research, 35, 2750–2759. http://dx.doi.org/10.1016/j.cor.2006.12.009

Nguyen, S., Zhang, M., Johnston, M., & Tan, K. C. (2013). A computational study of representations in genetic programming to evolve dispatching rules for the job shop scheduling problem. IEEE Transactions on Evolutionary Computation, 17, 621–639. http://dx.doi.org/10.1109/TEVC.2012.2227326

Nguyen, S., Zhang, M., Johnston, M., & Tan, K. C. (2014). Genetic programming for evolving due-date assignment models in job shop environments. Evolutionary Computation, 22, 105–138. http://dx.doi.org/10.1162/EVCO_a_00105

Parada, L., Herrera, C., Sepúlveda, M., & Parada, V. (2016). Evolution of new algorithms for the binary knapsack problem. Neural Computing, 15, 181–193. http://dx.doi.org/10.1016/j.ins.2014.01-0615-0

Ravi, S., & Shukla, S. (Eds.) (2009). Fundamental problems in computing. Dordrecht: Springer Netherlands. Retrieved from http://www.springerlink.com/index/10.1007/978-1-4020-9688-4

Rego, C., Gamboa, D., Glover, F., & Osterman, C. (2011). Traveling salesman problem heuristics: Leading methods, implementations and latest advances. European Journal of Operational Research, 211, 427–441. http://dx.doi.org/10.1016/j.ejor.2010.09.010

Rényi, A. (2003). A field guide to the theory of NP-completeness. A series of books in the mathematical sciences. San Francisco, CA: WH Freeman and Company.

Ryser-Welch, P., Miller, J. F., & Asta, S. (2015). Generating human-readable algorithms for the travelling salesman problem using hyper-heuristics. Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation (pp. 1067–1074). New York, NY: ACM.

Sabol, N. R., Ayob, M., Qu, R., & Kendall, G. (2012). A graph coloring constructive hyper-heuristic for examination timetabling problems. Applied Intelligence, 37(1), 1–11. http://dx.doi.org/10.1007/s10489-011-0309-9

Salhi, A., & Vázquez Rodríguez, J. A. V. (2014). Tailoring hyper-heuristics to specific instances of a scheduling problem using affinity and competence functions. Memetic Computing, 6, 77–84. http://dx.doi.org/10.1007/s12293-013-0121-7

Talbi, E. (2009). Metaheuristics. Hoboken, NJ: John Wiley and Sons. http://dx.doi.org/10.1007/9780470496916

Terashima-Marín, H., Ross, P., Farías-Zárate, C. J., López-Camacho, E., & Valenzuela-Rendón, M. (2010). Generalized hyper-heuristics for solving 2D regular and irregular packing problems. Annals of Operations Research, 179, 369–392. http://dx.doi.org/10.1007/s10479-008-0475-2

The Traveling Salesman Problem. (2016, September 25). Retrieved from http://www.math.uwaterloo.ca/tsp/index.html

© 2016 The Author(s). This open access article is distributed under a Creative Commons Attribution (CC-BY) 4.0 license.

You are free to:
- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:
- Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made.
- No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.